LogicPro: Logical Reasoning Enhanced with Program Examples

Anonymous ACL submission

Abstract

In this paper, we present a novel approach, called **LogicPro**, to enhance Logic reasoning through Program examples to improve multiple complex reasoning tasks simultaneously. We do this effectively by simply utilizing widely available algorithmic problems and their code solutions. First, we constructed diverse input test samples based on algorithmic questions and code solutions. Then, we designed different logic reasoning questions based on the algorithmic problems and test samples. Finally, combining the intermediate variable outputs of the code solutions and the logic reasoning questions, we obtain the final reasoning path through a large language model. Based on this, we are able to construct very rich *SFT* data. At the same time, we construct a diverse and scalable dataset of logical reasoning evaluation by treating each algorithmic question as a reasoning rule. As a result, our approach achieves significant improvements on multiple models for BBH dataset (20+ subsets), GSM8K and HellSwag datasets, and significantly outperforms a wide range of existing logical reasoning datasets. In addition, our eval data distinguishes well between existing models and brings new challenges to the model.

1 Introduction

All men are mortal.
Socrates is a man.
Therefore, Socrates is mortal.

In logic, Aristotle's syllogism is often used to explain deductive reasoning. In addition to deductive reasoning, logical reasoning includes other common forms, such as inductive reasoning, abductive reasoning, and analogical reasoning, which constitute the basic types of logical reasoning in the objective world. Recently, the rapid development of large language models has demonstrated powerful natural language processing capabilities. Logical reasoning, as a unique aspect of human cognition, is one of the key factors in measuring the generalized intelligence of these models. How to improve the models' ability in complex reasoning is directly related to their potential application in various fields.

For large language models, constructing relevant training datasets is the key to improving the logical reasoning ability of the models. Existing studies have conducted supervised training by collecting and constructing a variety of data, including but not limited to: Realistic logical reasoning (e.g. LOGIQA (Mill, 2013), AR-LSAT (Zhong et al., 2021), RECLOR (Yu et al., 2020)), synthetic Logical Reasoning (e.g. EntailmentBank (Dalvi et al., 2021), RuleBERT (Saeed et al., 2021), Adversarial NLI (Nie et al., 2020)), Mathematical Reasoning (e.g. GSM8K (Cobbe et al., 2021), AQUA-RAT (Ling et al., 2017), MATH (Hendrycks et al.)), and Common-Sense Reasoning (e.g. CommonsenseQA (Talmor et al., 2019), MedMCQA (Pal et al., 2022), OpenBookQA (Mihaylov et al., 2018)). Although these data improve the logical reasoning ability of the model to some extent, there are still many problems. Realistic logical reasoning data are limited in data size in the objective world and costly to collect. Synthetic logical reasoning data are typically constructed using a limited set of individual reasoning rules and patterns. Although these rules can be combined in numerous ways, they lack overall diversity and are prone to overfitting to a specific reasoning model. This limitation makes it challenging to improve logical reasoning abilities in out-of-domain scenarios. And some gaps remain between the two domains of mathematical reasoning and logical reasoning, although mathematical reasoning can assist in enhancing logical ability, the help it brings is limited and the enhancement is unstable. Knowledge-based reasoning data is mainly based on knowledge from different disciplines and domains, which is of limited help to complex logical reasoning ability.

For the test data, the eval benchmarks for logical reasoning are overall at the same pace as the training data, and researchers will provide the corresponding eval sets while constructing the logical reasoning training dataset. The above mentioned LOGIQA (Mill, 2013), AR-LSAT-TEST (Wang et al., 2022), RECLOR-DEV (Yu et al., 2020) and ConTRoL (Liu et al., 2021), TaxiNLI (Joshi et al., 2020), and NaN-NLI (Truong et al., 2022) and other test datasets are from the real world and have sufficiently complex patterns of reasoning, but are limited and scarce. HELP (Yanaka et al., 2019), TaxiNLI (Joshi et al., 2020), RuleTaker-dev (Clark et al., 2021) and ProofWriter-dev (Tafjord et al., 2021) are synthetic data, which are large and scalable, but with a more homogeneous reasoning model. Teng et al. (2023) also summarizes some of the above mentioned eval datasets in terms of target types of multiple choice, natural language reasoning and True-or-False. In addition, given the early date of construction of these benchmarks, there may be leakage issues for evaluating large language models based on extensive crawling of the Internet corpus. BBH (Suzgun et al., 2023) selected 23 most challenging tasks from BIG-Bench (Suzgun et al., 2023), which cover general-purpose languages. tasks, which cover general-purpose language comprehension, arithmetic and algorithmic reasoning, and logical reasoning. This dataset has a sufficiently diverse range of reasoning patterns, but its small data size and the need for manual labeling make it difficult to scale. In contrast, mathematical reasoning and intellectual reasoning serve more as auxiliary observation dimensions. Existing benchmarks can already do a good job of evaluating the reasoning ability of existing large models. However, there are no benchmarks that can do all three at the same time: diversity of reasoning rules, large enough data size, and scalability.

On the whole, it is difficult for both the training set and the evaluation set to achieve the three points of diversity of reasoning rules, large enough data volume and scalability at the same time. Considering that code data can well enhance the reasoning ability of large models (Zhang et al., 2024), and inspired by (Hua et al., 2024), training with "concrete" reasoning data has the ability of generalization, which can improve abstract reasoning, while training with abstract data is difficult to generalize to concrete reasoning problems. Therefore, we consider using widely available algorithmic questions and their code data to construct concrete logical reasoning questions from abstract code data. This approach not only further improves the reasoning ability of the model (compared to pure code data), but also simultaneously satisfies the three requirements of diverse reasoning rules, large data volume and scalability.

In this paper, we propose a method to enhance logical reasoning using algorithmic questions and their code. First, we use an open-source model (Llama3-70B-chat) to construct the inputs of multiple test samples based on algorithmic questions and their Python code. Then, we consider the inputs of the test samples and algorithmic questions, and obtain algorithmic questions based on the inputs of different samples as logical reasoning questions by rewriting the model. Subsequently, we consider the test sample input and the code solution to construct the code solution based on the current sample and obtain the final result as the standard answer for the logical reasoning question. Immediately after that, we rewrite the code using the current sample from the previous step and run the rewritten code so that it outputs the values of the important intermediate variables. Finally, combining the outputs of the questions and the intermediate variables, we obtain the final reasoning path. Based on this approach, we constructed a training set and a test set for each algorithmic topic. It can be found that the data constructed by our approach can achieve the three points of diversity of reasoning rules, sufficiently large data size and scalability at the same time.

2 Approach

The whole of our method is divided into five steps as shown in the figure 1. Through these steps, we are able to generate logical reasoning questions and answer pairs containing reasoning processes from algorithmic problem questions and code. And divide them into supervised training dataset and evaluation dataset.

2.1 Step 1: Constructing Test Sample Inputs

In the first step, our inputs are LeetCode algorithm questions and corresponding Python code solutions. As shown in the figure 2, we provide the algorithmic questions and Python code to the large open source model and ask the model to construct 30 test sample inputs at a time in a specific format. Specifically, we set the temperature to 0.7, perform multiple inference, extract test sample inputs from

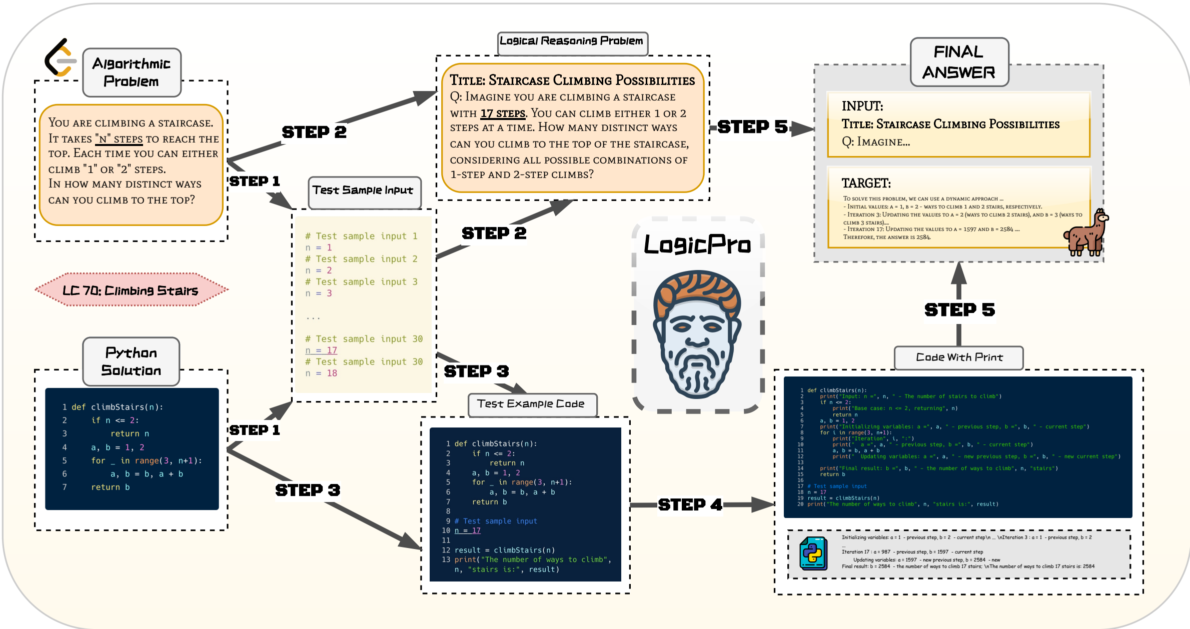

Figure 1: An overview of LogicPro

Step 1: Construct Test Sample Inputs

I have an algorithmic problem and its python code, please help me construct thirty different test sample inputs.
1. The constructed test sample inputs need to fulfill the requirements of the algorithmic problem and be compatible with the provided Python code.
2. Please enclose the constructed test sample inputs in the following python format; please enclose each test sample input individually.
```python
# Test sample input 1
``` ...

3. Ensure that all test samples are unique and as diverse as possible based on the topic and python code.
...

Figure 2: Constructing Test Sample Inputs

Step 2: Construct Logical Reasoning Problem

I have an algorithmic question and a corresponding test input; please rewrite the algorithmic question as a text-only logical reasoning question based on the test input.

Instructions:

1. Please incorporate the test input into the description ...
2. Please first give the name of this logical reasoning ...

Reference case I:

- algorithmic question: Given a sequence containing only (,), {{, }}, [,], <, >, complete the rest of the sequence, making sure that all the parentheses are properly closed and in the right order.
- test input: "< > (([[({{ }}) [< >]]"

Text-Only logical reasoning question:

Title: Correctly close a Dyck-n word.

Q: Complete the rest of the sequence, making sure that the parentheses are closed properly. Input: < > (([[({{ }}) [< >]]

Reference case II:

....
Refer to the above example of rewriting an algorithmic question into a text-only logical reasoning question based on test input:

Figure 3: Constructing Logical Reasoning Problems

the results of the multiple inference, and integrate these sample inputs to form the final test sample.

In conjunction with the example in the figure 1, we input a question description similar to the one in LeetCode for *Climbing Stairs* [1], along with the corresponding Python question solution. Based on these two points, the model will give possible test sample inputs, e.g. $n = 17$.

2.2 Step 2: Constructing Logical Reasoning Problems

For the second step, our inputs are LeetCode algorithmic questions and one of the constructed test sample inputs. As hinted in the figure 3, we ask the model to fuse the test sample input into the algorithmic question description. Also, we provide a rewrite sample of the *close a Dyck-n word* task as a reference case in the context of the prompt. Specifically in Figure 1, the model rewrites the *Climbing Stairs* algorithmic question as a concrete logical reasoning problem based on the test sample input ($n = 17$).

[1] https://leetcode.com/problems/climbing-stairs/description/

Step 3: Construct Text Example Code

I have a piece of <u>Python code</u> and a <u>test case input</u>. Please provide the modified code that can directly run this test sample based on the original Python code.
- Please ensure that the generated code can be executed directly.
- Please ensure that after running the code, the output result of the algorithm is returned through the variable `result`.

Figure 4: Constructing Test Sample Code

2.3 Step 3: Constructing Test Sample Code

In the third step, our inputs are the Python code solution and one of the constructed test sample inputs. As in the prompt in Figure 4, we ask the model to rewrite the Python code solution to fit the constructed test sample input. For example, for the *Climbing Stairs* problem in Figure 1, the rewritten code can be run directly at $n = 17$ and output the final result ($stairs = 2584$)

In particular, we will run the code here and collect standardized answers for different questions as a reference for subsequent training and evaluation sets.

2.4 Step 4: Rewriting the Code to Print Intermediate Variables

Step 4: Rewriting the Code to Print Intermediate Variable

Please modify the following code so that it <u>prints out important variables and their detailed descriptions related to the algorithm</u> at appropriate places.
1. Important variables refer to those critical for understanding the algorithm's logic, ...
2. Ensure that the printed information includes not only the names of the variables ...
3. Ensure that the printed information is closely related to the algorithm logic ...

Figure 5: Rewriting the Code to Print Intermediate Variables

In step 4, our input is the test sample code constructed in step 3. The model rewrites the original test sample code according to the prompt shown in the figure 5 so that it can print out important intermediate variable values. For example, for the *Climbing Stairs* problem in Fig 1, the rewritten code should output the values of a and b for each iteration step and their corresponding descriptions.

Considering that the length of intermediate steps varies from one algorithmic problem to another, some problems may print out very long intermediate variables. For this reason, we set up two sets of prompts to improve the test sample code and filter the variable printouts according to the result length. For the case that the token length of both sets of printout results is within 4096, we choose the set with longer printout results. While for the questions with excessively long printout results, we choose the set with shorter printout results.

2.5 Step 5: Constructing the Final Answer

Step 5: Construct The Final Answer

There is a <u>logical reasoning question</u> and the intermediate variable output of its code solution. Please answer this logical reasoning question based on the <u>intermediate variable output of the code</u>.

Instructions:

1. Refer to the code's intermediate variable outputs. ...
2. ...
...

Reference case:

- Logical reasoning question:

Title: Correctly close a Dyck-n word

Q: Complete the rest of the sequence, making sure that the parentheses are closed properly. Input: < > (([[({{ }}) [< >]]
- Code intermediate variables:

```
Initial stack: []
Initial result: < > ( ( [ [ ( {{ }} ) [ < > ] ]
Stack updated: ['<']
...
Result updated: < > ( ( [ [ ( {{ }} ) [ < > ] ]]))
Final result: < > ( ( [ [ ( {{ }} ) [ < > ] ]))
```
- Logical Reasoning Question Answer:

```
We should process each input one by one and keep track of the stack configuration.
0: empty stack
1: < ; stack: <
...
15: ] ; stack: ( ( [...
So the answer is ] ) ).
```

Figure 6: Constructing the Final Answer

In step 5, we input the logical reasoning problem constructed in step 2 and the intermediate variable output constructed in step 4. As shown in Fig. 6, we ask the model to refer to the intermediate variable outputs to assist the larger model in better logical reasoning. For the *Climbing Stairs* problem in Fig. 1, a more accurate and logical reasoning step can be given after considering the answers from the intermediate variable output.

2.6 Dataset

Based on the above process, we constructed the training set (LogicPro-Train) and the evaluation set (LogicPro-Eval) respectively. After completing Step 3, we filter and divide them according to certain rules. Specifically, we extract 5 input samples from the test samples of each algorithm question as the test set (10740). The rest of the samples will be extracted with an upper limit of 30 as the training set (70286). For the test set, we will run the test sample code directly after step 3 and use the result as the standard answer. For the training set, the subsequent processing steps are performed.

Overall, our method has advantages in the complexity of reasoning rules and data size. Based on our construction method, we can expand the possible test sample inputs without limit. In Table 5 of Appendix B.1, we compare LogicPro with other datasets. Only our dataset provides sufficient data size while ensuring that the reasoning rules are sufficiently complex.

3 Experiments

3.1 Evaluation Setup

3.1.1 Train Datasets

In order to verify the effectiveness of the LogicPro training set, we collected a collection of generic and logic supervised fine-tuning data from open-source sources. The generic data were mainly from OpenHermes-2.5 (Teknium, 2023). We first extracted all the *alpaca* data from OpenHermes, and then randomly sampled from the rest of the data to bring the total number of data up to 100k. The logical data was then taken from several open source logical reasoning datasets (Mill, 2013; Zhong et al., 2021; Yu et al., 2020; Dalvi et al., 2021; Nie et al., 2020; Ling et al., 2017; Talmor et al., 2019; Pal et al., 2022). We randomly selected 100,000 pieces of data as logic data. Given that many existing logic question datasets lack reasoning processes, directly using them for hybrid training may not effectively validate the usefulness of the new LogicPro data. Therefore, we used Llama-3-70B-Instruct to rewrite the collected data to construct the final logical reasoning dataset. The above generalized and logical data were mixed to generate *SFT* data (as Gen_Logic) for training the baseline model. Subsequently, we mixed the constructed LogicPro training data to verify its effectiveness.

To further validate the effectiveness of LogicPro-Train, we categorized the open source reasoning data into four dimensions for in-depth comparison. The first dimension is real-world logical reasoning data, including the larger LOGIQA (Mill, 2013), RECLOR (Yu et al., 2020) and AR-LSAT (Wang et al., 2022). The second dimension is synthetic data, including ProofWriter (Tafjord et al., 2021), RuleBERT (Saeed et al., 2021) and RuleTaker (Clark et al., 2021). The third dimension is mathematical reasoning data, including MAWPS (Koncel-Kedziorski et al., 2016), GSM8K (Cobbe et al., 2021), ASDIV (Miao et al., 2020), SVAMP (Patel et al., 2021) and AQUA -RAT (Ling et al., 2017). The fourth dimension is knowledge reasoning, covering openbookqa, strategyqa, tatqa, and pubmedqa. here we split the reasoning data in more detail than the more diverse logic data above, and use the unsampled full set of data for comparative verification.

3.1.2 Eval Datasets

We evaluated the model on BBH(Suzgun et al., 2023), GSM8K(Cobbe et al., 2021), and HellSwag(Zellers et al., 2019). BBH serves as a core benchmark for evaluating the logical reasoning ability of the model, and contains 23 challenging reasoning tasks. The task types are rich enough to serve as Out of Domain evaluation criteria. These task types cover natural language quizzes and multiple choice questions. However, given the wide variety of subsets of the BBH, it is difficult to effectively reflect the logical reasoning ability of the model by looking at multiple subset averages of the BBH alone. (While subsets of certain domains may have improved, one or two subsets may have significantly declined, resulting in no significant improvement in the BBH average.) Therefore, we extracted four representative BBH subsets for comparative analysis. We use **BOOL**, **CASUAL**, **SORT**, and **TRACKING** to denote the data subsets of BBH: boolean expressions, causal judgment, word sorting, and tracking shuffled object, respectively. GSM8K is used to assist in observing the mathematical reasoning ability of the model.

Specifically, all of our evaluation experiments were conducted in the form of zero-shot CoTs.

3.1.3 Metrics

On all evaluation tasks, we report the accuracy of the predicted answers. For GSM8K, we obtain the results by rule extraction and compute the corresponding metrics by exact matching. For BBH and HellSwag, we use an internal scoring model for evaluation. The inputs to this model are standard answers and modeled responses, and the output is a score (0 or 1).

In the LogicPro-Eval evaluation, we also utilize the scoring model to calculate the metrics

Base Model	*SFT* Data	**BOOL**	**CAUSAL**	**SORT**	**TRACKING**	**BBH**	**GSM8K**	**HellSwag**	**Average**
GPT-4	-	95.0	67.7	73.0	96.5	74.9	94.2	86.0	83.9
ChatGPT	-	87.5	64.67	52.5	70.0	50.25	65.28	76.0	66.6
Qwen1.5-7B	Gen_Logic	80.5	53.3	38.0	31.5	44.8	65.28	54.5	50.4
	w. LogicPro	84.5	54.5	48.0	36.0	45.7	65.51	61.3	55.0
Llama-2-7B	Gen_Logic	71.5	54.5	22.5	30.5	36.6	27.2	51.5	42.0
	w. LogicPro	74.0	52.1	10.0	32.5	36.3	30.1	52.8	41.1
Llama-3-8B	Gen_Logic	65.5	56.9	26.5	41.5	47.2	65.5	54.8	51.1
	w. LogicPro	66.5	57.5	77.0	53.5	50.3	68.8	59.0	61.8
Yi-1.5-9B	Gen_Logic	77.5	52.1	53.0	42.5	52.5	74.4	73.0	60.7
	w. LogicPro	80.0	52.1	56.5	41.5	53.2	77.3	79.0	62.8
llama-2-13B	Gen_Logic	58.0	55.7	38.5	37.0	40.4	43.0	45.5	45.4
	w. LogicPro	58.0	56.9	48.0	35.5	41.9	45.7	49.0	47.9
Qwen1.5-14B	Gen_Logic	87.5	62.3	58.2	52.0	52.4	70.3	70.5	64.7
	w. LogicPro	84.0	62.9	62.7	53.0	53.3	72.8	71.5	65.7

Table 1: Results for LogicPro-Train on Different Models.

3.2 Baslines

For Proprietary Models, we show results from SoTA LLMs such as OpenAI's GPT-4 and ChatGPT (gpt-3.5-turbo). For Open-Source Models, our models include Qwen1.5 (7B-13B), Llama3-8B, Llama2 (7B-13B), and Yi-9B. All of our experiments are trained on base models without *SFT*. The relevant training parameter settings are detailed in the Appendix.

Model	**LogicPro-Eval**
GPT-4	0.4629
Qwen1.5-7B-Chat	0.2864
Llama-3-8B-Instruct	0.2776
Qwen1.5-14B-Chat	0.2759
Llama3-70B-Instruct	0.3762
Qwen1.5-72B-Chat	0.3161

Table 2: Results on LogicPro-Eval; Zero-hot CoT evaluation

3.3 Main Results

3.4 LogicPro on Different Models

Table 1 shows the results of LogicPro-Train on different pedestal models. Overall, our model achieves significant improvement on almost all pedestal models except Llama2-7B. the average BBH improves steadily by 1-2 percentage points. On the **SORT** subset, almost all models gained significant boosts, with Llama3-8B improving by 50 percentage points. As an auxiliary observation for mathematical reasoning, GSM8K shows that all models improved after LogicPro training, which reveals to some extent the intrinsic connection between different reasoning tasks.

3.5 LogicPro vs. Different Data

Table 3 shows the results of comparing LogicPro with other logical inference data. Overall, LogicPro outperforms all other inference data. On the BBH average, the TRACKING subset, and GSM8K, LogicPro's results are slightly lower than the mathematical inference data. However, on the other three subsets, LogicPro significantly outperforms the mathematical reasoning data. Considering the diversity and complexity of logical reasoning, while mathematical data can enhance logical reasoning, logical reasoning needs more data with more diversity like LogicPro.

3.6 LogicPro-Eval

Table 2 shows the results of three open-source models and one closed-source model on LogicPro-Eval. Overall, LogicPro-Eval shows significant differences between models of different sizes, with GPT-4 performing significantly ahead.The Llama series of models outperforms the Qwen1.5 series of models in general. However, the results of Qwen1.5-7B and Qwen1.5-14B are not as expected, although considering the overall poor performance of the Qwen1.5 series on LogicPro-Eval, there is no sig-

Logic Data	**BOOL**	**CAUSAL**	**SORT**	**TRACKING**	**BBH**	**GSM8K**	**HellSwag**	**Average**
-	80.5	53.3	38.0	31.5	44.8	65.28	54.5	50.4
Realistic Logical	79.5	49.1	41.0	32.5	45.7	65.8	58.3	53.1
Synthetic Logical	83.0	52.1	34.5	31.5	45.0	66.0	48.3	51.5
Mathematical	80.0	47.3	39.0	**36.5**	**45.8**	**66.1**	60.0	53.5
Knowledge Reasoning	80.0	52.1	36.5	30.5	44.7	65.7	56.0	52.2
LogicPro	**84.5**	**54.5**	**48.0**	36.0	45.7	65.51	**61.3**	**55.0**

Table 3: Results on LogicPro-Train vs. Different Logic Data. Base Model: Qwen1.5-7B. Baseline Data: Gen_Logic.

nificant difference between the 14B model and the 72B model, which suggests that the LogicPro-Eval correlation capability is a much-needed improvement for the Qwen1.5 series as a whole .

In addition, all open-source models as well as GPT-4 did not reach 50% accuracy on LogicPro-Eval (GPT-4 had more than 70% accuracy on BBH), which suggests that LogicPro-Eval poses a completely new challenge for existing models.

4 Analysis

4.1 Ablation Study

SFT Data	**BBH**	**GSM8K**	**HellSwag**
general_10w	34.5	65.59	58.3
Gen_Logic	44.8	65.28	54.5
w. code	44.6	65.43	56.75
w. code*30	45.0	65.78	55.25
w. LogicPro_IO	45.0	66.51	58.25
w. LogicPro_COT	43.1	63.82	59.75
w. LogicPro_Final	45.7	65.51	61.3

Table 4: Results of ablation study on different *SFT* data. general_10w: Collected 10w general sft data; Gen_Logic: general_10w mixed 10w open source collection of logic data. **w.** denotes the mixing of different data based on **Gen_Logic**. Base Model: Qwen1.5-7B.

We performed a detailed ablation analysis of LogicPro-Train on Qwen 1.7-7B. First, we compared the results using generic 100k data, generic 100k mixed with logical 100k data, respectively. In addition, the code data itself is considered to enhance the inference of the model. To demonstrate that our LogicPro-Train data improves logical reasoning more compared to raw code. We transformed the raw code data into code quiz data and performed the same hybrid training to validate it. Meanwhile, in order to exclude the effect of training data volume, we oversampled the code data (2360*30) to make it close to LogicPro-Train's data volume (70286) and performed comparative training. The results in Table 4 show that our data significantly outperforms the code data itself on the logical reasoning task. Finally, we investigated the effect of different answer formats, comparing the results via intermediate code variables with the results of direct input-output (IO) and CoT rewriting using Llama3-70B-Instruct. The results show that LogicPro-Train (intermediate variable construction) outperforms direct rewriting.

4.2 Analysis of LogicPro-Eval

In the results table in Chapter 3.6, it can be seen that LogicPro-Eval is able to distinguish existing models very well. However, beyond the ability to distinguish between different models, how can LogicPro-Eval provide better feedback on the reasoning ability of a model? Unlike BBH, which has only 27 subsets, LogicPro-Eval has more than 2,600 rules, making it difficult to analyze them one by one. Therefore consider finding some dimensions to categorize from the original algorithmic questions.

First, regarding the difficulty of the code questions, as shown in the leftmost subplot of Figure 7, it can be seen that overall, the accuracy of the model on LogicPro-Eval decreases as the difficulty of the code questions increases. This reveals a potential correlation between code abstraction logic and LogicPro construction data.

Second, regarding the input type of the code questions, as shown in the middle subplot of Fig. 7, it can be seen that overall, there is no significant difference in the effect between different input types.

Then, regarding the time complexity of the code questions, as shown in the subplot on the right side of Figure 7, LogicPro-Eval has a weak associa-

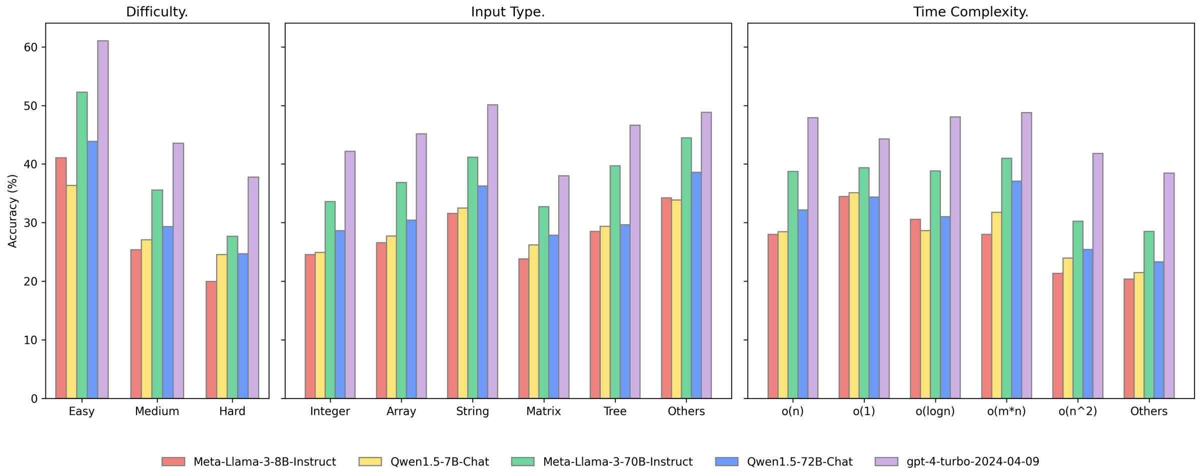

Figure 7: LogicPro-Eval results at different levels of difficulty/input type/time complexity

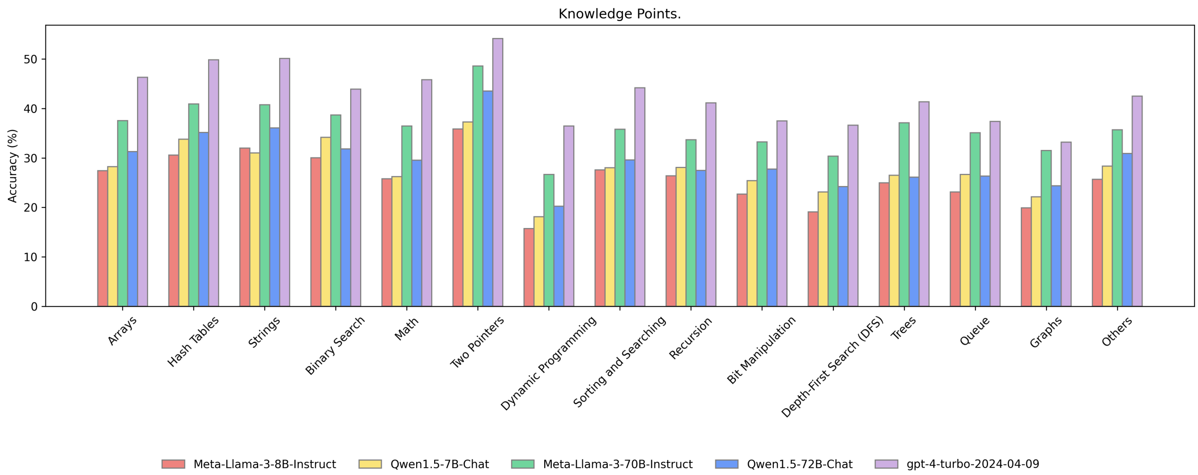

Figure 8: LogicPro-Eval results at different knowledge points

tion with time complexity. The model as a whole performs poorly on problems with high time complexity (e.g., $O(n^2)$). This may be due to the fact that the output length of questions with high time complexity tends to be longer, which leads to a decrease in the effectiveness of questions with high complexity.

Finally, regarding the knowledge points involved in the code questions, LogicPro-Eval is somewhat related to the knowledge points as shown in Figure 8. The overall results in the first few columns are better than in the latter columns. However, the modeling is also not done well in the dynamic programming problem, which humans are not very good at either

Inevitably, the effects of the four dimensions of difficulty, input type, time complexity, and knowledge point may be coupled. However, effective observation of the dimensions can to some extent help us better recognize the model's capability and further improve the model.

5 Conclusion

In this paper, we present LogicPro, which enhances logical reasoning through code cases. With this approach, we can construct datasets that combine the triple points of complexity of reasoning rules, large volume, and scalability, and extend them into LogicPro-Train and LogicPro-Test datasets. The training dataset can bring significant improvements on models of various sizes and origins. The testing dataset can effectively differentiate between existing models, while also bringing new challenges in logical reasoning to the models.

Limitations

Our approach explores a novel way of augmenting reasoning or constructing reasoning data. However, in step 5, we rely only on rewrites of open-source models, which can sometimes be problematic. For example, the model may say "from intermediate variables" and then give the final answer directly from the code print as if it were cheating, instead of reasoning step by step. We tried several approaches and found that this phenomenon cannot be avoided. However, we noticed that in all the cases we tried, GPT-4 always did this step well. However, considering the API cost associated with the large amount of data, we did not choose to use GPT-4 for the rewriting of step 5. This may be an important limitation facing the current dataset.

Ethics Statement

This study is based on data from 2360 algorithmic questions on the fully open-source LeetCode platform. All data are from publicly available sources and do not involve any personal privacy information. Our study strictly adheres to the terms of use and privacy policies of the platforms from which the data was sourced. We ensure that the rights of all users and platform regulations are respected during data collection and processing. Through the use of publicly available data, we aim to advance academic research and education, and promote progress in the field of algorithms and computer science

References

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Peter Clark, Oyvind Tafjord, and Kyle Richardson. 2021. Transformers as soft reasoners over language. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 3882–3890.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Bhavana Dalvi, Peter Jansen, Oyvind Tafjord, Zhengnan Xie, Hannah Smith, Leighanna Pipatanangkura, and Peter Clark. 2021. Explaining answers with entailment trees. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7358–7370.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Minlie Huang, Nan Duan, Weizhu Chen, et al. 2023. Tora: A tool-integrated reasoning agent for mathematical problem solving. *arXiv preprint arXiv:2309.17452*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *Sort*, 2(4):0–6.

Wenyue Hua, Kaijie Zhu, Lingyao Li, Lizhou Fan, Shuhang Lin, Mingyu Jin, Haochen Xue, Zelong Li, JinDong Wang, and Yongfeng Zhang. 2024. Disentangling logic: The role of context in large language model reasoning capabilities. *arXiv preprint arXiv:2406.02787*.

Pratik Joshi, Somak Aditya, Aalok Sathe, and Monojit Choudhury. 2020. Taxinli: Taking a ride up the nlu hill. In *Proceedings of the 24th Conference on Computational Natural Language Learning*, pages 41–55.

Rik Koncel-Kedziorski, Subhro Roy, Aida Amini, Nate Kushman, and Hannaneh Hajishirzi. 2016. Mawps: A math word problem repository. In *Proceedings of the 2016 conference of the north american chapter of the association for computational linguistics: human language technologies*, pages 1152–1157.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. 2017. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 158–167.

Hanmeng Liu, Leyang Cui, Jian Liu, and Yue Zhang. 2021. Natural language inference in context-investigating contextual reasoning over long texts. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 13388–13396.

Hanmeng Liu, Jian Liu, Leyang Cui, Zhiyang Teng, Nan Duan, Ming Zhou, and Yue Zhang. 2023a. Logiqa 2.0—an improved dataset for logical reasoning in natural language understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*.

Hanmeng Liu, Ruoxi Ning, Zhiyang Teng, Jian Liu, Qiji Zhou, and Yue Zhang. 2023b. Evaluating the logical reasoning ability of chatgpt and gpt-4. *arXiv preprint arXiv:2304.03439*.

Hanmeng liu, Zhiyang Teng, Ruoxi Ning, Jian Liu, Qiji Zhou, and Yue Zhang. 2023. Glore: Evaluating logical reasoning of large language models.

Shen-Yun Miao, Chao-Chun Liang, and Keh-Yih Su. 2020. A diverse corpus for evaluating and developing english math word problem solvers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 975–984.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *Conference on Empirical Methods in Natural Language Processing*.

John Stuart Mill. 2013. *A system of Logic, Ratiocinative and Inductive: Being a Connected View of the Principles of Evidence, and the Methods of Scientific Investigation*. Harper and Brothers, Publishers.

Yixin Nie, Adina Williams, Emily Dinan, Mohit Bansal, Jason Weston, and Douwe Kiela. 2020. Adversarial NLI: A new benchmark for natural language understanding. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4885–4901, Online. Association for Computational Linguistics.

Ankit Pal, Logesh Kumar Umapathi, and Malaikannan Sankarasubbu. 2022. Medmcqa: A large-scale multi-subject multi-choice dataset for medical domain question answering. In *Conference on health, inference, and learning*, pages 248–260. PMLR.

Keiran Paster, Marco Dos Santos, Zhangir Azerbayev, and Jimmy Ba. 2023. Openwebmath: An open dataset of high-quality mathematical web text.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are nlp models really able to solve simple math word problems? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics.

Mohammed Saeed, Naser Ahmadi, Preslav Nakov, and Paolo Papotti. 2021. Rulebert: Teaching soft rules to pre-trained language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 1460–1476.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, YK Li, Y Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. 2022. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc Le, Ed Chi, Denny Zhou, et al. 2023. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 13003–13051.

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

Oyvind Tafjord, Bhavana Dalvi, and Peter Clark. 2021. Proofwriter: Generating implications, proofs, and abductive statements over natural language. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 3621–3634.

Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. 2019. Commonsenseqa: A question answering challenge targeting commonsense knowledge. In *Proceedings of NAACL-HLT*, pages 4149–4158.

Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. 2022. Galactica: A large language model for science.

Teknium. 2023. Openhermes 2.5: An open dataset of synthetic data for generalist llm assistants.

Zhiyang Teng, Ruoxi Ning, Jian Liu, Qiji Zhou, Yue Zhang, et al. 2023. Glore: Evaluating logical reasoning of large language models. *arXiv preprint arXiv:2310.09107*.

Thinh Hung Truong, Julia Otmakhova, Timothy Baldwin, Trevor Cohn, Jey Han Lau, and Karin Verspoor. 2022. Not another negation benchmark: The nan-nli test suite for sub-clausal negation. In *Proceedings of the 2nd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 12th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 883–894.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023a. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models.

Peiyi Wang, Lei Li, Zhihong Shao, RX Xu, Damai Dai, Yifei Li, Deli Chen, Y Wu, and Zhifang Sui. 2023b. Math-shepherd: A label-free step-by-step verifier for llms in mathematical reasoning. *arXiv preprint arXiv:2312.08935*.

Siyuan Wang, Zhongkun Liu, Wanjun Zhong, Ming Zhou, Zhongyu Wei, Zhumin Chen, and Nan Duan. 2022. From lsat: The progress and challenges of complex reasoning. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*.

Siyuan Wang, Zhongyu Wei, Yejin Choi, and Xiang Ren. 2024. Can llms reason with rules? logic scaffolding for stress-testing and improving llms. *arXiv preprint arXiv:2402.11442*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. 2024. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data.

Hitomi Yanaka, Koji Mineshima, Daisuke Bekki, Kentaro Inui, Satoshi Sekine, Lasha Abzianidze, and Johan Bos. 2019. Help: A dataset for identifying shortcomings of neural models in monotonicity reasoning. In *Proceedings of the Eighth Joint Conference on Lexical and Computational Semantics (* SEM 2019)*, pages 250–255.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Weihao Yu, Zihang Jiang, Yanfei Dong, and Jiashi Feng. 2020. Reclor: A reading comprehension dataset requiring logical reasoning. *arXiv preprint arXiv:2002.04326*.

Lifan Yuan, Ganqu Cui, Hanbin Wang, Ning Ding, Xingyao Wang, Jia Deng, Boji Shan, Huimin Chen, Ruobing Xie, Yankai Lin, Zhenghao Liu, Bowen Zhou, Hao Peng, Zhiyuan Liu, and Maosong Sun. 2024. Advancing llm reasoning generalists with preference trees.

Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. 2023. Mammoth: Building math generalist models through hybrid instruction tuning. *arXiv preprint arXiv:2309.05653*.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4791–4800.

Xinlu Zhang, Zhiyu Zoey Chen, Xi Ye, Xianjun Yang, Lichang Chen, William Yang Wang, and Linda Ruth Petzold. 2024. Unveiling the impact of coding data instruction fine-tuning on large language models reasoning. *arXiv e-prints*, pages arXiv–2405.

Wanjun Zhong, Siyuan Wang, Duyu Tang, Zenan Xu, Daya Guo, Jiahai Wang, Jian Yin, Ming Zhou, and Nan Duan. 2021. Ar-lsat: Investigating analytical reasoning of text.

Kun Zhou, Beichen Zhang, Jiapeng Wang, Zhipeng Chen, Wayne Xin Zhao, Jing Sha, Zhichao Sheng, Shijin Wang, and Ji-Rong Wen. 2024. Jiuzhang3. 0: Efficiently improving mathematical reasoning by training small data synthesis models. *arXiv preprint arXiv:2405.14365*.

A Related work

A.1 Reasoning of LLMs

Reasoning, which servers as a fundamental ability of LLMs, determines the strength to solve complex real-world problems. Enhancing the reasoning ability of LLMs can mainly be divided into two ways.

Improve Reasoning of LLMs by Prompting. The reasoning ability of LLMs can be significantly stimulated by giving them different prompts, such as Chain-of-Thought(Wei et al., 2022), Plan-and-Solve(Wang et al., 2023a), etc. It is also possible to assist the model in reasoning by providing it with some external tools(Yao et al., 2022; Chen et al., 2022; Gou et al., 2023). These methods do not require parameter modification on LLMs, but do some control during LLM's reasoning to get a more reliable reasoning process and a better final result.

Improve Reasoning of LLMs by Training. Continuing pre-training provides a means to enhance the internal reasoning ability of LLMs from a knowledge perspective(Taylor et al., 2022; Paster et al., 2023). The ability of reasoning could be further enhanced by fine-tuning with instruction pairs related reasoning(Yue et al., 2023; Yuan et al., 2024). Reinforcement learning with two types of reward models: Outcome Reward Model (ORM)(Le et al., 2022; Shao et al., 2024) and Process Reward Model (PRM)(Lightman et al., 2023; Wang et al., 2023b), have also been used to improve the model's reasoning accuracy at various granularity. In addition, synthesised data from LLMs(Xin et al., 2024; Zhou et al., 2024) demonstrates the possibility of improving reasoning of LLMs themselves.

A.2 Logic Reasoning of LLMs

Logical reasoning epitomizes the art of deducing new insights from existing knowledge by adhering to specific principles and laws. This process does not necessitate a robust knowledge base. Instead, it emphasizes the precision and meticulousness with which conclusions are inferred from one piece of information to another.

Training Data of Logic Reasoning. There are various open-source available datasets for different types of logical reasoning tasks. LogiQA2.0(Liu et al., 2023a) is a complex logical reasoning dataset built from Chinese Civil Service Exam questions. ReClor(Yu et al., 2020), a dataset built on standardized graduate admission examinations, contains reading comprehension tasks requiring logical reasoning. ULogic(Wang et al., 2024) is logical reasoning dataset constructed from diverse inferential rules, which could improve various commonsense reasoning tasks.

Evaluation of Logic Reasoning. LogiEval(Liu et al., 2023b) and GLoRE(liu et al., 2023) combines several logical reasoning datasets, evaluating the logical reasoning of LLMs from multiple dimensions. Big-Bench Hard(Suzgun et al., 2022; Srivastava et al., 2022) is a diverse evaluation set that incorporates logical reasoning tasks such as logical deduction and logical fallacy detection.

B Data

B.1 Data Comparison

As shown in Fig. 5, we compare four types of data with LogicPro in terms of three dimensions: data size, data source and reasoning rule complexity. The results show that our method performs well in terms of data size, reasoning rule complexity and scalability.

C Prompts

Dataset	Size	Synthetic	Complexity Level of Reasoning Rules
		Realistic Logical Reasoning	
LOGIQA	8,678	not	complex (China Civil Service Exam)
RECLOR	6,138	not	complex (GMAT and LSAT)
FOLIO	1,435	not	medium (First-order logic)
DEER	1200	not	complex (Inductive reasoning)
E-KAR	1155	-	complex (Analogical Reasoning)
		Synthetic Logical Reasoning	
ProofWriter	20,192	yes	Simple (Entailment Tree)
PrOntoQA	-	yes	Simple (First-Order Logic)
RuleTaker	27363	yes	Simple
RuleBERT	310,000	yes	Simple
Clutrr	53,518	yes	Simple
		Mathematical Reasoning	
GSM8K	8,792	not	complex (Multi-step math reasoning)
AQUA-RAT	100,000	-	complex (Math reasoning with NL rationale)
ASDiv	2,305	not	complex (Multi-step math reasoning)
SVAMP	1,000	not	complex (Multi-step math reasoning)
		Commonsense Reasoning	
CommonsenseQA	12,247	-	medium (ConceptNet)
OpenBookQA	5,957	-	medium (Open-book knowledges)
LogicPro (our)	81,026	yes	complex (Logic from Code)

Table 5: Comparison of four types of datasets and LogicPro.

Step 1: Construct Test Sample Inputs

I have an algorithmic problem and its python code, please help me construct thirty different test sample inputs.
1. The constructed test sample inputs need to fulfill the requirements of the algorithmic problem and be compatible with the provided Python code.
2. Please enclose the constructed test sample inputs in the following python format; please enclose each test sample input individually.
```python
# Test sample input 1
# Your input here
```
```python
# Test sample input 2
# Your input here
```
...
```python
# Test sample input 30
# Your input here
```
3. Ensure that all test samples are unique and as diverse as possible based on the topic and Python code.
4. Consider various aspects of the input type to ensure diversity, such as:
- Range of values: Include small, medium, and large values, as well as edge cases.
- Special cases: Consider cases like empty input, maximum allowed input size, or inputs that might cause edge conditions.
- Pattern variations: If the input is a sequence, vary the sequence patterns (e.g., sorted, reverse-sorted, random order).
- Combining elements: If the input is a composite data structure (e.g., array of strings), combine different types of elements.
5. Generate inputs with varying difficulty levels (low, medium, high) considering the problem statement and the provided Python code:
- Low difficulty: Simple and straightforward inputs that cover basic scenarios.
- Medium difficulty: Moderately complex inputs that include more diverse and realistic scenarios.
- High difficulty: Complex inputs that test edge cases and challenging conditions.
6. Ensure that all test samples adhere to the constraints provided in the problem description.
7. Provide only the input for the test samples, do not include the output.

Algorithmic Questions Title:

{algorithmic_problems}

python solution:

{python_solution}

Figure 9: Step 1: Constructing Test Sample Inputs

Step 2: Construct Logical Reasoning Problem

I have an algorithmic question and a corresponding test input; please rewrite the algorithmic question as a text-only logical reasoning question based on the test input.

Instructions:

1. Please incorporate the test input into the description of the algorithm question;
2. Please first give the name of this logical reasoning task; then give the question that contains the test input.

Reference case I:

- algorithmic question: Given a sequence containing only (,), {{, }}, [,], <, >, complete the rest of the sequence, making sure that all the parentheses are properly closed and in the right order.
- test input: "< > (([[({{ }}) [< >]]"

- text-only logical reasoning question：

Title: Correctly close a Dyck-n word.

Q: Complete the rest of the sequence, making sure that the parentheses are closed properly. Input: < > (([[({{ }}) [< >]]

Reference case II:

- algorithmic question: You are given an integer array `cards` of length `4`. You have four cards, each containing a number in the range `[1, 9]`. You should arrange the numbers on these cards in a mathematical expression using the operators `['+', '-', '*', '/']` and the parentheses `'('` and `')'` to get the value 24. You are restricted with the following rules: * The division operator `'/'` represents real division, not integer division.
- test input: "[4, 1, 8, 7]"

- text-only logical reasoning question：

Title: Achieve the Target Value

Q: You are presented with four cards, each bearing a number within the range of 1 to 9. Using the numbers on these cards, form a mathematical expression by arranging them with the operators `+`, `-`, `*`, and `/`, as well as parentheses `(` and `)`, such that the resulting value of the expression is 24. Note the following rules:
- Division operator `/` represents real division, not integer division.
- Each operation must be performed between two numbers (no unary operations).
- Numbers cannot be concatenated to form multi-digit numbers.
Given the cards with numbers [4, 1, 8, 7], determine if it is possible to form an expression that evaluates to 24.
Can you find such an expression, or prove that it cannot be done?

Refer to the above example of rewriting an algorithmic question into a text-only logical reasoning question based on test input:
- algorithmic question: {algorithmic_question}
- test input: {test_sample_input}

- text-only logical reasoning question:

Figure 10: Step 2: Constructing Logical Reasoning Problems

Step 3: Construct Text Example Code

I have a piece of Python code and a test case input. Please provide the modified code that can directly run this test sample based on the original Python code.
- Please ensure that the generated code can be executed directly.
- Please ensure that after running the code, the output result of the algorithm is returned through the variable `result`.

Test case input:
{test_sample_input}

python code:

{python_solution}

Figure 11: Step 3: Constructing Test Sample Code

Step 4: Rewriting the Code to Print Intermediate Variable

Please modify the following code so that it prints out important variables and their detailed descriptions related to the algorithm at appropriate places.
1. Important variables refer to those critical for understanding the algorithm's logic, such as loop counters, function inputs and outputs, key condition judgments, and variables indicating state changes.
2. Ensure that the printed information includes not only the names of the variables but also their roles and meanings within the algorithm, to better understand the execution process of the code.
3. Ensure that the printed information is closely related to the algorithm logic and does not include irrelevant content (such as code errors and exceptions).
4. Ensure that the printed information is detailed enough.

python code:

{test_example_code}

Figure 12: Step 4: Rewriting the Code to Print Intermediate Variables

Step 5: Construct The Final Answer

There is a logical reasoning question and the intermediate variable output of its code solution. Please answer this logical reasoning question based on the intermediate variable output of the code.

Instructions:

1. Refer to the code's intermediate variable outputs. Use the information provided to help you answer the logical reasoning questions.
2. First, outline your approach to solving the logical reasoning task. Then, provide the exact reasoning process step by step.
3. Do not use code to solve this logical reasoning problem. Instead, use the provided intermediate variable outputs to guide your answer.
4. Do not mention “intermediate variables” in your answer. Focus on solving the logical reasoning question directly.
5. Avoid phrases like “From the intermediate variables” in the answer. Just use them(intermediate variables) to help you answer the logical reasoning question.

Reference case:

- Logical reasoning question:

Title: Correctly close a Dyck-n word

Q: Complete the rest of the sequence, making sure that the parentheses are closed properly. Input: < > (([[({{ }}) [< >]]
- Code intermediate variables:

```
Initial stack: []
Initial result: < > ( ( [ [ ( {{ }} ) [ < > ] ]
Stack updated: ['<']
...
Stack updated: ['(', '(', '[']
Result updated: < > ( ( [ [ ( {{ }} ) [ < > ] ]]
Result updated: < > ( ( [ [ ( {{ }} ) [ < > ] ]])
Result updated: < > ( ( [ [ ( {{ }} ) [ < > ] ]]))
Final result: < > ( ( [ [ ( {{ }} ) [ < > ] ]))
```

- Logical Reasoning Question Answer:

```
We should process each input one by one and keep track of the stack configuration.
0: empty stack
1: < ; stack: <
...
15: ] ; stack: ( ( [
Now, we have reached the end. The final stack is "( ( ["
We will need to pop out "[", "(", "(" one by one in that order.
So, we need "]", ")", ")". So the answer is ] ) ).
```

Refer to the above case to give a solution to a logical reasoning question:
- Logical reasoning question:
{logic_reasoning_problem}
- Code intermediate variables:

```
{code_print}
```

Figure 13: Step 5: Constructing the Final Answer