LogicPro: Improving Complex Logical Reasoning via Program-Guided Learning

Jin Jiang[1], Yuchen Yan[2], Yang Liu[3], Jianing Wang[3], Shuai Peng[1], Xunliang Cai[3], Yixin Cao[4], Mengdi Zhang[3], Liangcai Gao[1]*

[1]Wangxuan Institute of Computer Technology, Peking University, Beijing, China,
[2]College of Computer Science and Technology, Zhejiang University, [3]Meituan Group,
[4]Institute of Trustworthy Embodied AI, Fudan University

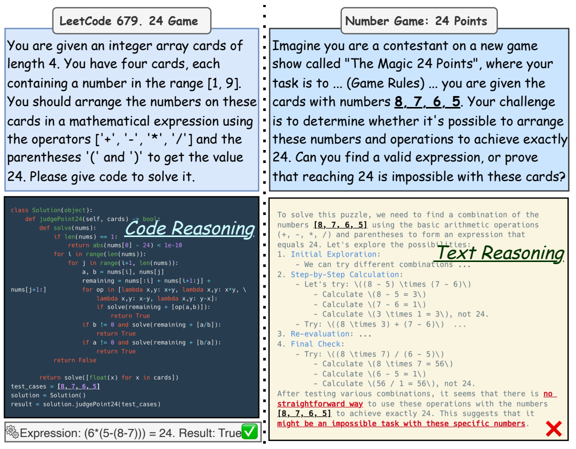
Correspondence: jiangjin@stu.pku.edu.cn, gaoliangcai@pku.edu.cn

Abstract

In this paper, we propose a new data synthesis method called **LogicPro**, which leverages LeetCode-style algorithm Problems and their corresponding Program solutions to synthesize Complex Logical Reasoning data in text format. First, we synthesize complex reasoning problems through source algorithm problems and test cases. Then, standard answers and intermediate variable outputs are obtained for each problem based on standard python solutions and test cases. Finally, with the guidance of code intermediate variables, we synthesize the text reasoning process for each reasoning problems. Through this method, we can synthesize data that is difficult, scalable, effective, and comes with golden standard answers and high-quality reasoning processes. As a result, with our 540K synthesized dataset constructed solely from 2,360 algorithm problems, our approach [1] achieves significant improvements in multiple models for the datasets *BBH*^{27}, *LogicBench*, *DROP*, *AR-LSAT*, and *GSM8K*, etc. outperforming a wide range of existing reasoning datasets.

1 Introduction

With the rapid development of artificial intelligence, Large Language Models (LLMs) (Bi et al., 2024; Liu et al., 2024a) demonstrate excellent performance in reasoning tasks. The success of these models is inseparable from the support of large-scale and high-quality reasoning data. However, data acquisition and processing face numerous challenges in the real world. As a viable alternative, synthetic data (Wang et al., 2024) can effectively alleviate this problem and further enhance (Dubey et al., 2024; Adler et al., 2024) the model's reasoning capabilities.



LeetCode 679. 24 Game

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. Please give code to solve it.

```
class Solution(object):
    def judgePoint24(self, cards) -> bool:
        def solve(nums):
            if len(nums) == 1:
                return abs(nums[0] - 24) < 1e-10
            for i in range(len(nums)):
                for j in range(i+1, len(nums)):
                    a, b = nums[i], nums[j]
                    remaining = nums[:i] + nums[i+1:j] +
nums[j+1:]          for op in [lambda x,y: x+y, lambda x,y: x*y, \
                        lambda x,y: x-y, lambda x,y: y-x]:
                        if solve(remaining + [op(a,b)]):
                            return True
                    if b != 0 and solve(remaining + [a/b]):
                        return True
                    if a != 0 and solve(remaining + [b/a]):
                        return True
            return False

        return solve([float(x) for x in cards])
test_cases = [8, 7, 6, 5]
solution = Solution()
result = solution.judgePoint24(test_cases)
```

```
Expression: (6*(5-(8-7))) = 24. Result: True✅
```

Number Game: 24 Points

Imagine you are a contestant on a new game show called "The Magic 24 Points", where your task is to ... (Game Rules) ... you are given the cards with numbers **8, 7, 6, 5**. Your challenge is to determine whether it's possible to arrange these numbers and operations to achieve exactly 24. Can you find a valid expression, or prove that reaching 24 is impossible with these cards?

```
To solve this puzzle, we need to find a combination of the
numbers [8, 7, 6, 5] using the basic arithmetic operations
(+, -, *, /) and parentheses to form an expression that
equals 24. Let's explore the possibilities:
1. Initial Exploration:
   - We can try different combinations ...
2. Step-by-Step Calculation:
   - Let's try: \((8 - 5) \times (7 - 6)\)
      - Calculate \(8 - 5 = 3\)
      - Calculate \(7 - 6 = 1\)
      - Calculate \(3 \times 1 = 3\), not 24.
   - Try: \((8 \times 3) + (7 - 6)\)  ...
3. Re-evaluation: ...
4. Final Check:
   - Try: \((8 \times 7) / (6 - 5)\)
      - Calculate \(8 \times 7 = 56\)
      - Calculate \(6 - 5 = 1\)
      - Calculate \(56 / 1 = 56\), not 24.
After testing various combinations, it seems that there is no
straightforward way to use these operations with the numbers
[8, 7, 6, 5] to achieve exactly 24. This suggests that it
might be an impossible task with these specific numbers.
```

Figure 1: Left: LeetCode 679. 24 Game - original algorithm problem and standard Python solution. Right: Our synthesized complex reasoning problem: 24-point and g4o-api response.

Synthetic data (Liu et al., 2024b) has wide applications in mathematics and code domains. For mathematical data, synthetic data can be generated through problem-driven methods such as evol-instruct (Luo et al., 2023a; Zeng et al., 2024), problem restatement (Yu et al., 2023), back-translation (Lu et al., 2024), and few-shot examples (Dalvi et al., 2021), or knowledge-driven methods based on knowledge bases (Dalvi et al., 2021) and concept graphs (Tang et al., 2024). For code data, it generates diverse and high-quality instruction data with self-evolutionary methods (Wang et al., 2023; Luo et al., 2023b) or combined with open-source references such as OSS-Instruct (Wei et al., 2024).

In the complex logical reasoning domain, there is relatively less research on data synthesis. Previously, some work uses logical proposition-based soft reasoning methods (Tafjord et al., 2021; Clark et al., 2021) to synthesize training data, aiming to improve models' reasoning ability and interpretability (Saeed et al., 2021; Dalvi et al., 2021). Additionally, Sinha et al. (2019) and Tian et al. (2021) use kinship graphs and first-order logic

*Corresponding author.
[1]Code and data are publicly available at https://github.com/jiangjin1999/LogicPro

to synthesize person relationship reasoning data and logical entailment reasoning data respectively. More recently, Morishita et al. (2023) using formal logic theory, Parmar et al. (2024) covering 25 patterns across different logic types, and Morishita et al. (2024) utilizing formal language programs to synthesise reasoning data.

Previous methods primarily used propositional logic or formal languages as the source of logic. Instead, we find that algorithmic questions and programming languages like LeetCode provide a different source of logic. Algorithmic problems typically involve explicit input-output relationships, recursive and iterative structures, and operations on data structures. All of these constitute a unique pattern of reasoning. At the same time, algorithmic problems are naturally related to real-world task contexts, such as path-planning, data sorting, and resource allocation problems, which can often be mapped to real-world application scenarios.

As shown in Figure 1, we discover that while an algorithm problem can be easily solved through code, LLMs often make mistakes when the same problem is transformed into a specific text reasoning question. Inspired by this observation, we propose a data synthesis method aimed at generating high-quality reasoning problems and reasoning processes by utilizing widely available algorithmic problems and their code solutions.

Our approach consists of data collection and three steps. For data collection, we collect a large number of existing LeetCode algorithmic problems and their code solutions, while collecting or constructing diverse test cases. First, by combining the original algorithm problem with test cases, it is transformed into a specific text reasoning problem. Then, the original standard code solution is combined with the test cases to generate the corresponding python code solution. By running the code, the final output result and the values of key intermediate variables can be obtained. Finally, based on the code output, the model is guided to synthesize the complete reasoning process for the current problem.

This data synthesis method offers significant advantages: sufficient difficulty, scalability, effectiveness, and high-quality reasoning paths. Specifically, the sufficient difficulty is reflected in models performing worse on LogicPro compared to other baseline data; scalability was demonstrated by collecting more algorithmic problems and constructing more test cases to further scale the data; effectiveness is shown through performance improvements across multiple models on multiple Out-of-Distribution (OOD) benchmarks.

Our main contributions are summarized as follows:

1. We propose a novel data synthesis method called LogicPro, which uses LeetCode-style data as seeds to synthesize text-formatted complex logical data through algorithmic problems and program solution.

2. With this approach, we can synthesise a 540K dataset from just 2,360 algorithmic problems that is sufficiently difficult, scalable, and effective, as well as having standard answer and high-quality reasoning process.

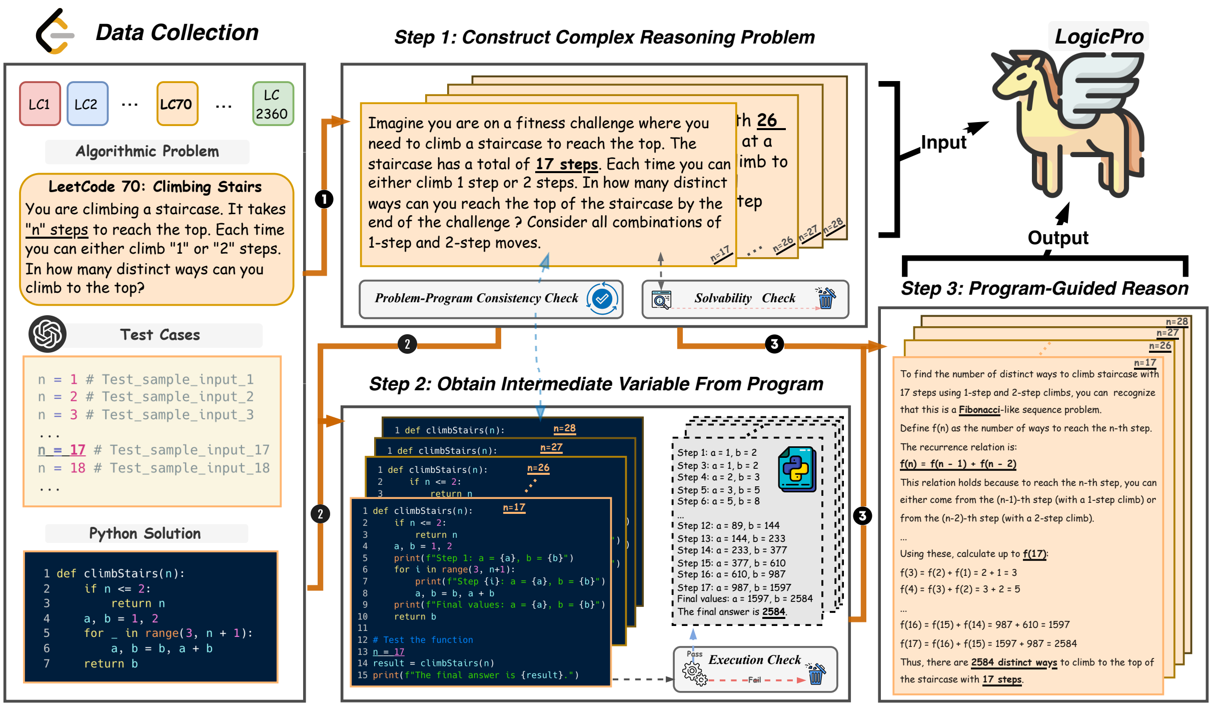
3. The experimental results show that our approach achieves significant improvements in multiple models for the datasets *BBH*^{27}, *LogicBench*, *DROP*, *AR-LSAT*, and *GSM8K*, etc. outperforming a wide range of existing datasets.

2 Approach

In this section, we elaborate on the LogicPro data synthesis process. It mainly includes: data collection and construction of test cases (Section 2.1), how to synthesize textual complex logical reasoning problems (Section 2.2), how to obtain intermediate variable outputs from code (Section 2.3), and how to synthesize high-quality reasoning processes based on intermediate variable outputs and synthesized questions (Section 2.4). An overview of our approach is shown in Figure 2.

2.1 Data Collection

In the data collection phase, source Leet data was collected test cases were constructed. We collect 2,360 official LeetCode problems as initial seeds for LogicPro's synthetic data. However, due to the limited number of test cases in the original data, GPT-4 is used to reconstruct test cases for each question. Specifically, we prompt GPT-4 to generate 150 test cases for each LeetCode problem, sample three times, and then perform consolidation, deduplication, and filtering on the three results. Eventually, 2,360 LeetCode algorithm problems were compiled with standard Python solutions, each containing up to 300 test cases.


Data Collection
LC1
LC2
LC70
LC 2360
Algorithmic Problem
Test Cases
Python Solution
Step 1: Construct Complex Reasoning Problem
Problem-Program Consistency Check
Solvability Check
n=17
n=26
n=27
n=28
Step 2: Obtain Intermediate Variable From Program
Execution Check
Pass
Fail
LogicPro
Input
Output
Step 3: Program-Guided Reason

LeetCode 70: Climbing Stairs
You are climbing a staircase. It takes "n" steps to reach the top. Each time you can either climb "1" or "2" steps. In how many distinct ways can you climb to the top?

```
n = 1 # Test_sample_input_1
n = 2 # Test_sample_input_2
n = 3 # Test_sample_input_3
...
n = 17 # Test_sample_input_17
n = 18 # Test_sample_input_18
...
```

```python
def climbStairs(n):
    if n <= 2:
        return n
    a, b = 1, 2
    for _ in range(3, n + 1):
        a, b = b, a + b
    return b
```

Imagine you are on a fitness challenge where you need to climb a staircase to reach the top. The staircase has a total of 17 steps. Each time you can either climb 1 step or 2 steps. In how many distinct ways can you reach the top of the staircase by the end of the challenge ? Consider all combinations of 1-step and 2-step moves.

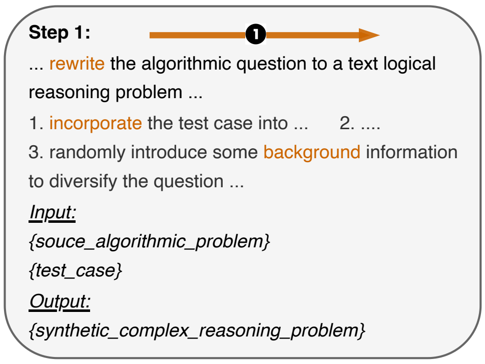
```python
def climbStairs(n):
    if n <= 2:
        return n
    a, b = 1, 2
    print(f"Step 1: a = {a}, b = {b}")
    for i in range(3, n+1):
        print(f"Step {i}: a = {a}, b = {b}")
        a, b = b, a + b
    print(f"Final values: a = {a}, b = {b}")
    return b

# Test the function
n = 17
result = climbStairs(n)
print(f"The final answer is {result}.")
```

```
Step 1: a = 1, b = 2
Step 3: a = 1, b = 2
Step 4: a = 2, b = 3
Step 5: a = 3, b = 5
Step 6: a = 5, b = 8
...
Step 12: a = 89, b = 144
Step 13: a = 144, b = 233
Step 14: a = 233, b = 377
Step 15: a = 377, b = 610
Step 16: a = 610, b = 987
Step 17: a = 987, b = 1597
Final values: a = 1597, b = 2584
The final answer is 2584.
```

To find the number of distinct ways to climb staircase with 17 steps using 1-step and 2-step climbs, you can recognize that this is a Fibonacci-like sequence problem.
Define f(n) as the number of ways to reach the n-th step.
The recurrence relation is:
f(n) = f(n - 1) + f(n - 2)
This relation holds because to reach the n-th step, you can either come from the (n-1)-th step (with a 1-step climb) or from the (n-2)-th step (with a 2-step climb).
...
Using these, calculate up to f(17):
f(3) = f(2) + f(1) = 2 + 1 = 3
f(4) = f(3) + f(2) = 3 + 2 = 5
...
f(16) = f(15) + f(14) = 987 + 610 = 1597
f(17) = f(16) + f(15) = 1597 + 987 = 2584
Thus, there are 2584 distinct ways to climb to the top of the staircase with 17 steps.

Figure 2: An overview of LogicPro (Example: LeetCode-70 Climbing Stairs): 1. Construct Complex Reason Problem (Section 2.2 Step 1), based on source algorithm problems and test cases to synthesize complex reasoning problems. 2. Obtain Intermediate Variable From Program (Section 2.3 Step 2) 3. Program-Guided Reason (Section 2.4 Step 3), synthesizing the final Input-Output from complex reasoning problems and intermediate variable outputs.

2.2 Step 1: Construct Complex Reasoning Problem


Step 1:

... rewrite the algorithmic question to a text logical reasoning problem ...

1. incorporate the test case into ... 2. ....

3. randomly introduce some background information to diversify the question ...

Input:
{souce_algorithmic_problem}
{test_case}
Output:
{synthetic_complex_reasoning_problem}

Figure 3: Schematic Diagram of Step 1.

In the first step, we synthesize complex reasoning problems based on the original algorithm problems and test cases. As shown in Figure 3, the input includes the source LeetCode algorithm problems and specific test case. In this process, the LLM needs to combine test cases and algorithm problems, while randomly adding some background information, to synthesize a complex reasoning problem. Referring to the example in Figure 2, the model combine LeetCode Problem 70 "Climbing Stairs" and the test case input n=17 into a specific text reasoning problem.

For the data flow in this step, 699K complex reasoning problems in text format are generated. Subsequently, after conducting consistency check (5K+) and solvability check, we select 595K qualified problems for the next step.

Problem-Program Consisitency Check The purpose of this module is to examine the consistency between code and text reasoning problem. In the second step, we rewrite the original standard Python solution and generate test case specific python code. To ensure consistency between the rewritten Python code and the text reasoning problems generated in the first step, we perform consistency checks on the code and problems. After inspection, 5K+ data entries are filtered out. Prompt is in Appendix C.2 Figure 12.

Solvability Check This check module is designed to check whether the synthesised questions are solvable. In the first step, a large number of textual reasoning problems were generated by combining the original LeetCode problems with the test cases generated by GPT-4. However, some of these problems are unsolvable or meaningless. This is mainly because the test cases generated by

GPT-4 are not entirely perfect. Although GPT-4 ensures that the generated test cases comply with the problem requirements and code format, these test cases may still deviate from the core testing points of the problems, or the sample length is too long, resulting in unsolvable synthesized problems. After detection, we have screened out and filtered 98K unsolvable problems. Prompt is in Appendix C.2 Figure 13.

2.3 Step2: Obtain Intermediate Variable from Program

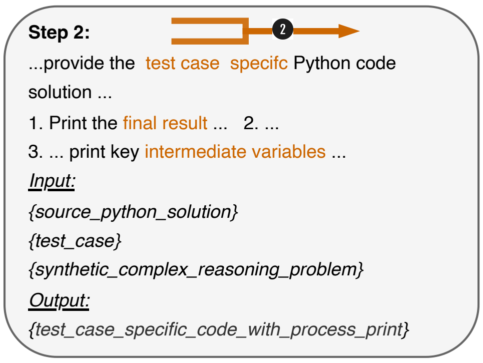

Step 2:
...provide the test case specifc Python code solution ...
1. Print the final result ... 2. ...
3. ... print key intermediate variables ...
Input:
{source_python_solution}
{test_case}
{synthetic_complex_reasoning_problem}
Output:
{test_case_specific_code_with_process_print}

Figure 4: Schematic Diagram of Step 2.

In the second step, we input Python code solutions, test cases, and synthesized reasoning problems to generate test case specific python code. As shown in Figure 4, we require the model to generate python code solutions related to the test cases and ensure that the code prints the final result to the result variable. In addition, to simulate the human reasoning process, the model is required to print key intermediate variables in the code.

For example, for the climbing stairs problem shown in Figure 2, the model needs to generate python code for the specific test case of climbing 17 stairs, and print intermediate variables in the code, namely the calculation process of the Fibonacci sequence at each step. Finally, by executing the code, the output of the intermediate variables can be obtained, which will assist in the reasoning synthesis for the next step.

For the data flow of this step, the final answer to each problem is obtained by running the combined code that integrates the test cases with the standard Python solution. Meanwhile, we modify the code to print important intermediate variable values. Then, through execution check, we finally obtain 544K data points, including standard answers and code intermediate variable outputs for each problem.

Execution Check This check module is designed to detect errors that occur during code execution. In the step 2, the output of intermediate variables in the code was obtained by modifying the original standard Python solution for these errors, we perform data filtering, which can be divided into two categories: The first category involves modified code that encounters errors during execution and still fails to run properly after multiple sampling attempts; The second category involves code that executes correctly but still contains error messages in the intermediate variable outputs due to the use of try-catch functionality. After detection, these two types of issues result in the filtering of 50K error codes.

2.4 Step3: Program-Guided Reasoning

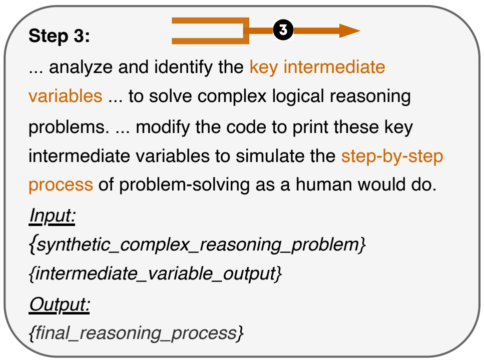

Step 3:
... analyze and identify the key intermediate variables ... to solve complex logical reasoning problems. ... modify the code to print these key intermediate variables to simulate the step-by-step process of problem-solving as a human would do.
Input:
{synthetic_complex_reasoning_problem}
{intermediate_variable_output}
Output:
{final_reasoning_process}

Figure 5: Schematic Diagram of Step 3.

In the third step, the input consists of the complex reasoning problem generated in the first step and the intermediate variable output obtained in the second step. These elements are used to synthesize the final reasoning process. As shown in Figure 5, the model (Llama3.1-70B-Instruct) answers synthesized complex reasoning questions by analyzing intermediate variable data.

Taking the example in Figure 2, based on the calculation process of the Fibonacci equation in the intermediate variable output, the model successfully synthesized a high-quality reasoning process as the final output of LogicPro data. The goal of this step is to generate golden reasoning processes with logic and accuracy, thereby ensuring the completeness and credibility of the reasoning.

For the data flow in this step, based on the text-format complex reasoning problems and corresponding code intermediate variable outputs obtained from previous steps, we use this important intermediate variable information to guide the model in generating high-quality reasoning processes for complex reasoning problems. Finally, a high-quality resoning quetion-answer datasets with 540k input-output pairs is obtained.

3 Experiments

3.1 Experimental Setup

3.1.1 Training

This section elaborates on the experimental settings related to training. We introduce: (1) the baseline datasets used for comparison with our LogicPro; (2) experimental design to verify the effect of different synthetic data. (3) the model configurations used in training and analysis; (4) the parameters and implementation details of the training process.

Baseline Data To comprehensively evaluate the effectiveness of our method, we select multiple mainstream logical reasoning data synthesis methods as baseline comparisons, including RuleTakers (Clark et al., 2021), LogicNLI (Tian et al., 2021), ProofWriter (Tafjord et al., 2021), CLUTRR (Sinha et al., 2019), RuleBert (Saeed et al., 2021), LogicBench (Parmar et al., 2024), and FLD (Morishita et al., 2023). The detailed information of these datasets can be found in Appendix A.1 Table 5.

Experimental Design To better simulate the actual training scenarios of LLMs, we use two types of base data: first, 100K general domain data extracted from OpenHermes-2.5 (Teknium, 2023), and second, LogiCoT (Liu et al., 2023) as the specialized base data for logical reasoning domain. By mixing these different base data with baseline data and our synthetic data for training separately, we systematically analysed the impact of different synthetic data on model performance.

Models We conduct systematic experimental research on foundation models of various scales and architectures. The main experiments employ four representative models: Qwen-7B and Llama3-8B as small-scale representatives, and Qwen2-72B and Llama3-70B as large-scale model representatives. For in-depth understanding of model behaviors, we select Qwen2-7B and Llama-8B for fine-grained analysis experiments.

Training Details In terms of training implementation, we use Megatron-LM as the training framework with the following configurations: a cosine learning rate schedule is adopted with an initial learning rate of 1e-5, a warmup ratio of 0.03, and the learning rate decays to 0; the maximum sequence length is set to 8192, with a global batch size of 128, and the number of training epochs is set to 3. All experiments are completed with Supervised Fine-tuning (SFT) on a computing cluster consisting of 32 NVIDIA A100 GPUs.

3.1.2 Evaluation

This section details the experimental setup associated with the evaluation. We present: (1) the benchmarks used to evaluate the different synthetic data; and (2) the implementation details of the evaluation process.

Benchmarks To comprehensively evaluate the model's complex logical reasoning capabilities, ten representative benchmark datasets for testing are selected , including BBH[27] (Suzgun et al., 2023), LogicBench (Parmar et al., 2024), DROP (Dua et al., 2019), AR-LSAT (Zhong et al., 2021), Boardgamqa (Kazemi et al., 2024), FOLIO (Han et al., 2022), GSM8K (Cobbe et al., 2021), Multi-LogicEval (Patel et al., 2024), ProofWriter (Tafjord et al., 2021), and MATH (Hendrycks et al., 2020). Notably, for our synthetic data LogicPro, all benchmark are out-of-domain tests. In comparison, some baseline data, same as some benchmark, are synthesized based on propositional logic or first-order logic. Additionally, LogicBench's training and test sets are completely in-domain. Furthermore, BBH, as a core benchmark for evaluating models' complex logical reasoning capabilities, includes 27 challenging reasoning tasks. Based on this, we apply weights to all data subsets to calculate the final average score. Due to space limitations, only the evaluation results of seven benchmarks are presented in the main text, with complete results available in Appendix A.2 Table 6.

Evaluation Details In the inference phase, we use the vLLM (Kwon et al., 2023) framework for deployment. The inference configuration adopts greedy decoding strategy and sets the maximum generation length to 4096 tokens. For the evaluation of model output, we adopt Qwen-2.5-72B as the model evaluator to score. The specific evaluation prompt template can be found in Figure 14.

3.2 Main Results

Table 1 shows the main results where LogicPro outperforms previous synthesis methods across multiple benchmarks. On the representative Big Bench Hard benchmark, LogicPro improves the

Model	**BBH^{27}**	**LogicBench**	**DROP**	**ARLSAT**	**#BoardQA**	**FOLIO**	**GSM8K**	**#Avg**
Qwen2-7B-RuleTakers	45.4	59.1	65.7	16.5	42.2	44.6	80.9	45.8
Qwen2-7B-LogicNLI	43.3	71.3	67.4	17.8	45.3	41.7	81.6	45.0
Qwen2-7B-ProofWriter	40.8	68.6	64.3	17.0	36.9	36.3	80.9	43.2
Qwen2-7B-CLUTRR	43.0	72.0	64.0	17.0	**51.9**	41.2	80.4	45.0
Qwen2-7B-RuleBert	46.2	69.1	67.4	16.5	43.8	40.7	81.7	47.3
Qwen2-7B-LogicBench	44.7	*95.9	67.4	17.8	41.4	38.7	**82.1**	46.6
Qwen2-7B-FLD	42.0	69.5	**68.3**	14.8	34.9	45.6	80.0	43.8
Qwen2-7B-LogicPro (ours)	**50.9**	**73.5**	**68.3**	**19.1**	48.1	**46.1**	81.5	**51.2**
Llama3-8B-RuleTakers	38.5	59.9	65.9	12.6	**47.3**	43.3	67.9	40.3
Llama3-8B-LogicNLI	40.4	54.0	65.3	12.6	41.0	56.0	69.3	41.8
Llama3-8B-ProofWriter	37.2	62.1	66.4	**15.2**	31.3	47.8	69.1	39.5
Llama3-8B-CLUTRR	40.5	61.1	66.6	10.4	43.7	56.0	69.5	42.2
Llama3-8B-RuleBert	34.7	48.8	66.5	**15.2**	43.6	51.5	68.9	37.4
Llama3-8B-LogicBench	41.0	*93.5	66.2	10.9	38.6	**62.7**	69.8	43.7
Llama3-8B-FLD	35.7	67.8	61.2	13.5	39.1	50.0	64.6	38.2
Llama3-8B-LogicPro (ours)	**45.0**	**67.9**	**68.8**	**15.2**	44.3	48.3	**74.2**	**46.2**
Qwen2-72B-RuleTakers	61.3	72.4	76.6	19.6	61.0	49.3	88.5	61.3
Qwen2-72B-LogicNLI	61.7	80.7	77.0	21.3	60.4	58.2	87.3	61.6
Qwen2-72B-ProofWriter	61.8	75.5	77.2	16.5	55.0	44.0	88.0	61.5
Qwen2-72B-CLUTRR	68.1	79.0	78.4	24.4	61.0	59.7	**89.4**	66.7
Qwen2-72B-RuleBert	67.8	74.1	76.5	19.6	55.2	**62.7**	88.0	66.5
Qwen2-72B-LogicBench	67.1	*97.0	77.9	24.8	57.3	60.5	88.4	67.2
Qwen2-72B-FLD	65.4	72.4	76.3	17.0	45.5	53.7	86.9	63.6
Qwen2-72B-LogicPro (ours)	**72.4**	**81.7**	**79.6**	**27.4**	**66.4**	55.2	89.1	**70.4**
Llama3-70B-RuleTakers	51.5	69.1	75.6	19.1	61.5	53.9	86.8	52.7
Llama3-70B-LogicNLI	58.5	69.9	78.0	17.8	58.5	58.3	84.5	57.8
Llama3-70B-ProofWriter	55.3	31.9	75.3	15.2	58.7	51.0	65.4	53.2
Llama3-70B-CLUTRR	57.8	71.8	74.0	20.9	62.1	**61.3**	75.1	57.3
Llama3-70B-RuleBert	56.0	68.7	75.0	13.9	51.5	51.0	85.4	55.1
Llama3-70B-LogicBench	61.4	*93.2	78.4	21.3	58.7	50.0	84.6	60.8
Llama3-70B-FLD	57.2	74.4	75.0	16.5	46.3	56.9	85.5	56.4
Llama3-70B-LogicPro (ours)	**63.7**	**72.7**	**78.8**	**22.3**	**65.0**	54.2	**87.6**	**62.4**

Table 1: Main results of LogicPro with baseline data. Where #BoardQA represents BoardgameQA and #Avg represents Average. **Bold** denotes the best score in that baseline, underline denotes the second highest score, and * denotes same-distribution data. See Table 6 for more details.

average performance by 2.3% - 4.7% compared to the previous best baseline across different model type and scales, and shows at least 10% improvement over general baseline data. On LogicBench (a benchmark measuring propositional logic capabilities), LogicPro achieves the best performance except when LogicBench itself is used as the training set. On FOLIO (a benchmark measuring first-order logic capabilities), except for Qwen2-7B, the performance of the other three models trained with LogicPro is inferior to other baseline data. This may be because some baseline synthetic data is essentially generated based on first-order logic, making their data distribution closer to FOLIO, leading to better performance. On GSM8K (mathematical reasoning benchmark), different synthetic data has relatively minor impact on model performance. On OOD benchmarks such as DROP (reading comprehension reasoning) and ARLSAT (law school admission test reasoning), LogicPro also demonstrates advantages across multiple model foundations, further validating its performance on out-of-distribution tasks. The weighted average results across all benchmarks show that on the Qwen2 foundation, 7B and 72B models improve by 3.9%-8% and 3.1%-9.1% respectively compared to baselines; on the Llama3 foundation, 8B and 70B models improve by 2.5%-8.8% and 1.6%-9.7% respectively compared to baselines.

Additionally, for different scale models, we find that our LogicPro still shows advantages on large-scale models. This indicates that although large-scale models possess stronger reasoning capabilities, our data still provides important value for improving their performance. It is speculated that this may be related to the high difficulty level of LogicPro itself, with detailed analysis in section 4.4.

4 Analysis

4.1 Ablation Study

Qwen2-7B	**BBH**^{27}	**LogicBench**	**#Avg**
Source_LeetCode	41.0	63.4	42.2
LogicPro$_{\text{w/o Inter-Var}}$	48.2$_{+7.2}$	69.6$_{+6.2}$	49.0$_{+6.8}$
LogicPro$_{\text{w. Inter-Var}}$	**50.9**$_{+9.9}$	**73.5**$_{+10.1}$	**51.2**$_{+9.0}$
Llama3-8B	**BBH**^{27}	**LogicBench**	**#Avg**
Source_LeetCode	36.6	51.4	38.5
LogicPro$_{\text{w/o Inter-Var}}$	44.0$_{+7.4}$	63.6$_{+12.2}$	45.1$_{+6.6}$
LogicPro$_{\text{w. Inter-Var}}$	**45.0**$_{+8.4}$	**67.9**$_{+16.5}$	**46.2**$_{+7.7}$

Table 2: Ablation study. Source_LeetCode is the source 2360 LeetCode algorithm questions and code solution. LogicPro$_{\text{w/o Inter-Var}}$ and LogicPro$_{\text{w. Inter-Var}}$ indicate whether the construction process uses intermediate variables. In other words, LogicPro$_{\text{w/o Inter-Var}}$ is a direct distillation of the text problems in LogicPro and LogicPro$_{\text{w. Inter-Var}}$ is our final LogicPro data. Llama3.1-70B-Instruct is the model to distill and generate final reasoning trajectories.

As shown in Table 2, we conduct ablation studies to validate LogicPro's effectiveness in synthesizing questions and generating high-quality reasoning processes. First, we compare Source_LeetCode with LogicPro$_{\text{w/o Inter-Var}}$ to analyze the effect on synthesizing text reasoning questions. For BBH, our method improves the performance of Qwen2-7B and Llama3-8B by 7.21% and 7.34% respectively, with significant improvements of 6.8% and 6.6% on the Average metric. Subsequently, we compare LogicPro$_{\text{w/o Inter-Var}}$ with LogicPro$_{\text{w. Inter-Var}}$ to analyze the effect of code intermediate variables in the reasoning process. The results show that introducing code intermediate variables as guidance can further improve data quality compared to directly distilling synthesized questions. Specifically, compared to LogicPro$_{\text{w/o Inter-Var}}$, it achieves improvements of 2.68% and 1.04% on BBH, and improvements of 2.2% and 1.1% on the Average metric.

4.2 Performance Gain on All Baseline Data

As shown in Table 3, we combine all baseline data to evaluate the effect of introducing LogicPro. The experimental results show that on top of integrating all baseline data, the addition of LogicPro can further improve model performance. On the BBH task, it brings improvements of 5.6% and 4.7% respectively, while for average performance, it improves by 5.2% and 3.6% respectively.

Qwen2-7B	**BBH**^{27}	**LogicBench**	**#Avg**
All_Baseline	43.1	92.8	47.1
All* + LogicPro	**48.8**$_{+5.7}$	**96.3**$_{+3.5}$	**52.3**$_{+5.2}$
Llama3-8B	**BBH**^{27}	**LogicBench**	**#Avg**
All_Baseline	42.9	94.3	47.6
All* + LogicPro	**47.6**$_{+4.7}$	**95.0**$_{+0.7}$	**51.2**$_{+3.6}$

Table 3: Results of continuous performance improvement on existing data. All_Baseline represents that we mix all baseline data from the main experiment. All* + LogicPro represents that we further mix LogicPro data for comparison.

It is worth emphasizing that our goal is not to propose a method to replace existing synthetic data, but rather to introduce a novel data synthesis strategy. Different from previous synthetic data, our data's logic comes from LeetCode algorithm problems and programming languages, rather than propositional logic and formal languages. As the results above, our data complements existing data to further enhance the model's complex reasoning capabilities.

4.3 Data Scaling Analysis

In Figure 6, we demonstrate the impact of three data scaling methods on BBH and Average metrics to analyze how the total data amount, number of algorithm problems, and number of test cases in data synthesis methods affect model performance. Overall, the *Data* scaling method shows a steady upward trend, indicating that the synthesized data after mixing effectively improves model performance, while also revealing potential for further improvement through expanded data scale.

In contrast, the *Leetcode Problems* scaling method shows more fluctuation. As the number of Leetcode problems increases from 30% to 100%, both BBH and Average metrics display multiple patterns of decline followed by increase. This may be due to distributional differences in the data generated from different algorithm problems, leading to fluctuations in the mixed data results.

The *Test Cases* method generally shows an upward trend, but model performance declines after

Model	#Rule	LogicNLI	#Proof	CLUTRR	RuleBert	LogicBench	FLD	LogicPro
Qwen2-7B-Instruct	73.0	62.8	64.5	46.3	53.6	74.5	54.1	**39.3**
Qwen2-72B-Instruct	78.3	78.9	73.8	72.0	54.1	77.3	69.1	**46.3**
Llama3-8B-Instruct	69.0	63.3	62.1	67.0	47.8	75.5	51.5	**41.2**
Llama3-70B-Instruct	80.6	79.6	78.9	83.8	65.5	82.8	64.4	**55.6**
Average	75.2	71.2	69.8	67.3	55.2	77.5	59.8	**45.6**

Table 4: Results from different baseline data and LogicPro's difficulty comparisons on four open source models. #Rule represents RuleTakers and #Proof represents ProofWriter. The metrics for the results in the table are accuracy rates, and smaller is better.

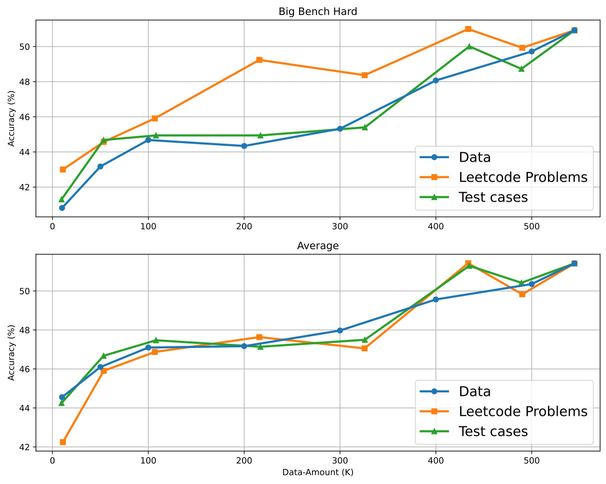


Figure 6: The results of the three scaling methods, *Data*, *Leetcode Problems* and *Test Cases*, on BBH and Average. *Data* means we randomly sample 10K-50K-100K∼500K-540K from LogicPro; *Leetcode Problems* means that we sample 2%-10%-20%∼90%-100% according to the number of algorithmic problems; *Test cases* means that we select all algorithmic problems and sample the number of test cases for each problem according to 2%-10%-20%∼90%-100%.

the number of test cases increases to 80%. This may be attributed to two factors: First, there is diminishing marginal utility of test cases. With a fixed number of algorithm problems, simply increasing test cases may reach a performance improvement bottleneck, and only further increasing the number of algorithm problems can effectively breakthrough. Second, the impact of test case quality: The large number of test cases directly generated by GPT4 are limited in quality. Although increasing quantity brings some improvement, constructing higher-quality test cases is still needed for more significant optimization.

In summary, to further improve the performance of the model, it is necessary to enhance the quality of test cases while increasing the number of algorithmic problems. This can break through the bottleneck of data expansion. On this basis, the model's capabilities can be continuously enhanced.

4.4 Difficulty Comparison with Baseline Data

As mentioned above, LogicPro is a sufficiently difficult dataset. To further validate this point, in Table 4 we compare the performance between various baseline datasets and LogicPro across four open-source models. Specifically, we randomly sample 5K samples from each baseline dataset and LogicPro for evaluation, where LogicPro uses a uniform sampling strategy to ensure each algorithm problem is fairly selected.

The results show that four open-source models Qwen2-7B-Instruct, Qwen2-72B-Instruct, Llama3-8B-Instruct, and Llama3-70B-Instruct perform quite well on existing baseline datasets, with average accuracy exceeding 50%, and some datasets even reaching over 70%. However, their performance drops significantly on the LogicPro dataset, with an average accuracy of only 45.6%. Notably, as mentioned in the "Main Results" section, LogicPro still poses significant challenges for Large Language Models, which may explain why there is still considerable space for improvement in their performance on large-scale models.

5 Related work

5.1 Synthetic Reasoning Data

Synthetic data has played an important role in enhancing LLMs' reasoning capabilities (Dubey et al., 2024), especially in mathematics and coding domains.

For mathematical reasoning, synthetic data generation includes problem-driven methods such as evol-instruct (Luo et al., 2023a; Zeng et al., 2024), problem restatement (Yu et al., 2023), back-translation (Lu et al., 2024), and few-shot examples (Dalvi et al., 2021), as well as knowledge-driven methods that rely on knowledge bases (Dalvi et al.,

2021) or concept graphs (Tang et al., 2024) to generate new problems by sampling key reasoning points (Huang et al., 2024).

In terms of code reasoning (Chen et al., 2025), from Code Alpaca's (Wang et al., 2023) use of self-instruct to generate 20K instruction data based on seed tasks, to WizardCoder's (Luo et al., 2023b) use of Code evol-instruct to generate more complex tasks, to Magicoder's use of oss-instruct (Wei et al., 2024) to extract 75K instructions from open source code. Synthesizing data continuously improves the model's code reasoning capabilities.

In contrast, there is less research on synthetic data for complex logical reasoning. Learning combines formal logic theory (Morishita et al., 2023) to synthesize data using basic reasoning rules to train language models' multi-step reasoning abilities. LogicBench (Parmar et al., 2024) not only constructs logical reasoning benchmark datasets but also provides synthetic data based on formal logic.

LeetCode-style algorithm problems contain rich reasoning processes. This paper synthesizes high-quality complex logical sreasoning data based on the formal logic of programming languages to enhance models' reasoning capabilities.

5.2 Symbolic Reason Enhances LLM Reason

Symbolic language was initially used for formal logical reasoning, mathematical computation, and program verification, playing a crucial role in early artificial intelligence applications such as expert systems (Hatzilygeroudis and Prentzas, 2004) and automated theorem provers (Loveland, 2016). With the rapid development of LLMs, symbolic reasoning, as a structured reasoning approach, has further enhanced the reasoning capabilities of large models through their integration.

There are two main approaches to combining symbolic reasoning with LLMs. The first approach utilizes symbolic language for planning and reasoning. Some research combines LLMs with symbolic solvers like Python (Gao et al., 2023) and SAT (Ye et al., 2023) to solve mathematical (He-Yueya et al., 2023) and logical reasoning problems (Pan et al., 2023), reducing the reasoning burden on models. Additionally, some work employs symbolic language as a planning (Hao et al., 2024; Wen et al., 2024) tool to strengthen LLMs' reasoning capabilities.

The second approach involves using symbolic language to generate training data. Early research (Tafjord et al., 2021; Clark et al., 2021) adopted neural network-based soft reasoning methods to synthesize training data from logical rules expressed in natural language, improving reasoning capabilities and interpretability (Saeed et al., 2021; Dalvi et al., 2021). Recently, more research has explored the application of symbolic methods in synthetic training data. Li et al. (2024) formalizes seed data and replaces certain variables to synthesize mathematical reasoning data; Morishita et al. (2023) constructs synthetic corpora based on formal logic; Shao et al. (2024) generates inductive reasoning data through case-to-code mapping; Parmar et al. (2024) provides more comprehensive propositional logic benchmarks; Morishita et al. (2024) strengthens LLM reasoning capabilities through principled synthesis of logical corpora. These studies have advanced the development of logical reasoning capabilities.

In this paper, we further explore the application of symbolic methods in data synthesis, utilizing formal logical information embedded in LeetCode algorithm problems to synthesize complex logical reasoning data.

6 Conclusion and Future Work

This paper presents a new data synthesis method called LogicPro. This approach utilizes LeetCode-style algorithm problems and solutions to generate complex logical reasoning data. By synthesizing a 540K dataset from just 2,360 seed problems, our approach ensures scalability, difficulty, and high-quality reasoning paths. Results show that LogicPro enhances model performance across multiple reasoning benchmarks, including BBH[27], LogicBench, DROP, AR-LSAT, and GSM8K, outperforming a wide range of existing datasets.

For future work, considering the vast number of algorithmic problems in the real world, such as problems from Luogu, ACM competitions, and various online judges, we can collect more algorithmic problem data to further expand LogicPro's dataset. Additionally, our data includes both result signals (code execution results as **standard answers**) and process signals (intermediate variables during code execution), which may provide new insights for reinforcement learning.

Limitations

Our method explores an algorithm based on LeetCode-style approach to synthesize complex logical reasoning data. In the future, one of our improvement directions is to build more comprehensive and diverse test cases. However, the generation of test cases itself is an independent research field. More advanced test generation techniques are expected to further enhance the quality and generalization ability of synthetic datasets. Furthermore, although LeetCode's official 2,360 problems have achieved significant results, there are still numerous high-quality algorithm problems in the real world, such as Luogu, ACM competitions, and various Online Judge (OJ) platforms. Meanwhile, new algorithm problems continue to emerge. If these resources can be fully utilized, the quality and coverage of synthetic data will be further enhanced, leading to a larger data scale and better performance.

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

Acknowledgments

This work is supported by the projects of Beijing Science and Technology Program (Z231100007423011) and National Natural Science Foundation of China (No. 62376012), which is also a research achievement of State Key Laboratory of Multimedia Information Processing and Key Laboratory of Science, Technology and Standard in Press Industry (Key Laboratory of Intelligent Press Media Technology).

References

Bo Adler, Niket Agarwal, Ashwath Aithal, Dong H Anh, Pallab Bhattacharya, Annika Brundyn, Jared Casper, Bryan Catanzaro, Sharon Clay, Jonathan Cohen, et al. 2024. Nemotron-4 340b technical report. *arXiv preprint arXiv:2406.11704*.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Xiancai Chen, Zhengwei Tao, Kechi Zhang, Changzhi Zhou, Wanli Gu, Yuanpeng He, Mengdi Zhang, Xunliang Cai, Haiyan Zhao, and Zhi Jin. 2025. Revisit self-debugging with self-generated tests for code generation. *arXiv preprint arXiv:2501.12793*.

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.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2368–2378.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Wenfei Zhou, James Coady, David Peng, Yujie Qiao, Luke Benson, et al. 2022. Folio: Natural language reasoning with first-order logic. *arXiv preprint arXiv:2209.00840*.

Yilun Hao, Yang Zhang, and Chuchu Fan. 2024. Planning anything with rigor: General-purpose zero-shot planning with llm-based formalized programming. *arXiv preprint arXiv:2410.12112*.

Ioannis Hatzilygeroudis and Jim Prentzas. 2004. Neuro-symbolic approaches for knowledge representation in expert systems. *International Journal of Hybrid Intelligent Systems*, 1(3-4):111–126.

Joy He-Yueya, Gabriel Poesia, Rose Wang, and Noah Goodman. 2023. Solving math word problems by combining language models with symbolic solvers. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS'23*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2020. hendrycks2measuring. *Sort*, 2(4):0–6.

Yiming Huang, Xiao Liu, Yeyun Gong, Zhibin Gou, Yelong Shen, Nan Duan, and Weizhu Chen. 2024. Key-point-driven data synthesis with its enhancement on mathematical reasoning. *arXiv preprint arXiv:2403.02333*.

Mehran Kazemi, Quan Yuan, Deepti Bhatia, Najoung Kim, Xin Xu, Vaiva Imbrasaite, and Deepak Ramachandran. 2024. Boardgameqa: A dataset for natural language reasoning with contradictory information. *Advances in Neural Information Processing Systems*, 36.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Zenan Li, Zhi Zhou, Yuan Yao, Xian Zhang, Yu-Feng Li, Chun Cao, Fan Yang, and Xiaoxing Ma. 2024. Neuro-symbolic data generation for math reasoning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024a. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*.

Hanmeng Liu, Zhiyang Teng, Leyang Cui, Chaoli Zhang, Qiji Zhou, and Yue Zhang. 2023. Logicot: Logical chain-of-thought instruction tuning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 2908–2921.

Ruibo Liu, Jerry Wei, Fangyu Liu, Chenglei Si, Yanzhe Zhang, Jinmeng Rao, Steven Zheng, Daiyi Peng, Diyi Yang, Denny Zhou, et al. 2024b. Best practices and lessons learned on synthetic data for language models. *arXiv preprint arXiv:2404.07503*.

Donald W Loveland. 2016. *Automated theorem proving: A logical basis*. Elsevier.

Zimu Lu, Aojun Zhou, Houxing Ren, Ke Wang, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2024. Mathgenie: Generating synthetic data with question back-translation for enhancing mathematical reasoning of llms. *arXiv preprint arXiv:2402.16352*.

Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Jianguang Lou, Chongyang Tao, Xiubo Geng, Qingwei Lin, Shifeng Chen, and Dongmei Zhang. 2023a. Wizardmath: Empowering mathematical reasoning for large language models via reinforced evol-instruct. *arXiv preprint arXiv:2308.09583*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023b. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*.

Terufumi Morishita, Gaku Morio, Atsuki Yamaguchi, and Yasuhiro Sogawa. 2023. Learning deductive reasoning from synthetic corpus based on formal logic. In *Proceedings of the 40th International Conference on Machine Learning*, pages 25254–25274.

Terufumi Morishita, Gaku Morio, Atsuki Yamaguchi, and Yasuhiro Sogawa. 2024. Enhancing reasoning capabilities of llms via principled synthetic logic corpus. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. 2023. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Mihir Parmar, Nisarg Patel, Neeraj Varshney, Mutsumi Nakamura, Man Luo, Santosh Mashetty, Arindam Mitra, and Chitta Baral. 2024. Logicbench: Towards systematic evaluation of logical reasoning ability of large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13679–13707.

Nisarg Patel, Mohith Kulkarni, Mihir Parmar, Aashna Budhiraja, Mutsumi Nakamura, Neeraj Varshney, and Chitta Baral. 2024. Multi-logieval: Towards evaluating multi-step logical reasoning ability of large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 20856–20879.

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.

Yunfan Shao, Linyang Li, Yichuan Ma, Peiji Li, Demin Song, Qinyuan Cheng, Shimin Li, Xiaonan Li, Pengyu Wang, Qipeng Guo, et al. 2024. Case2code: Learning inductive reasoning with synthetic data. *CoRR*.

Koustuv Sinha, Shagun Sodhani, Jin Dong, Joelle Pineau, and William L Hamilton. 2019. Clutrr: A diagnostic benchmark for inductive reasoning from text. In *Proceedings of the 2019 Conference on Empirical*

Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pages 4506–4515.

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.

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.

Zhengyang Tang, Xingxing Zhang, Benyou Wang, and Furu Wei. 2024. Mathscale: Scaling instruction tuning for mathematical reasoning. In *Forty-first International Conference on Machine Learning*.

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

Jidong Tian, Yitian Li, Wenqing Chen, Liqiang Xiao, Hao He, and Yaohui Jin. 2021. Diagnosing the first-order logical reasoning ability through logicnli. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3738–3747.

Ke Wang, Jiahui Zhu, Minjie Ren, Zeming Liu, Shiwei Li, Zongye Zhang, Chenkai Zhang, Xiaoyu Wu, Qiqi Zhan, Qingjie Liu, et al. 2024. A survey on data synthesis and augmentation for large language models. *arXiv preprint arXiv:2410.12896*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *The 61st Annual Meeting Of The Association For Computational Linguistics*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning*.

Jiaxin Wen, Jian Guan, Hongning Wang, Wei Wu, and Minlie Huang. 2024. Codeplan: Unlocking reasoning potential in large langauge models by scaling code-form planning. *CoRR*.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2023. Satlm: satisfiability-aided language models using declarative prompting. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, pages 45548–45580.

Longhui Yu, Weisen Jiang, Han Shi, YU Jincheng, Zhengying Liu, Yu Zhang, James Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2023. Metamath: Bootstrap your own mathematical questions for large language models. In *The Twelfth International Conference on Learning Representations*.

Weihao Zeng, Can Xu, Yingxiu Zhao, Jian-Guang Lou, and Weizhu Chen. 2024. Automatic instruction evolving for large language models. *arXiv preprint arXiv:2406.00770*.

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. *arXiv preprint arXiv:2104.06598*.

A Details of Experiment

A.1 Baseline Data Statistics

In Table 5, we statistics on the size of the baseline data, the logical sources of the synthetic data.

Dataset	**Size**	**Logic Source**	**Source**
RuleTakers	480K	Soft Rule	(Clark et al., 2021)
LogicNLI	48K	FOL	(Tian et al., 2021)
ProofWriter	580K	Soft Rule (proof)	(Tafjord et al., 2021)
CLUTRR	50K	Kinship Graph	(Sinha et al., 2019)
RuleBert	310K	Soft Rule	(Saeed et al., 2021)
LogicBench	12K	Propositional Logic	(Parmar et al., 2024)
FLD	300K	Formal Logic	(Morishita et al., 2023)
LogicPro (ours)	540K	Program Logic	LogicPro

Table 5: Data Statistics.

A.2 Complete Main Results

Given space constraints, we show the complete results for all benchmarks in Table 6.

Results for All Subsets of BBH In the table 7, we give the histogram results for all baseline data and LogicPro on the 27 subsets of BBH.

Results for Base Data In section 3.1, paragraph Experimental design, we mentioned that in order to simulate the actual large model training manufacturers, we choose the generic domain and logical reasoning domain data as the base data. The results of the base data without adding any synthetic data are given in Table 7.

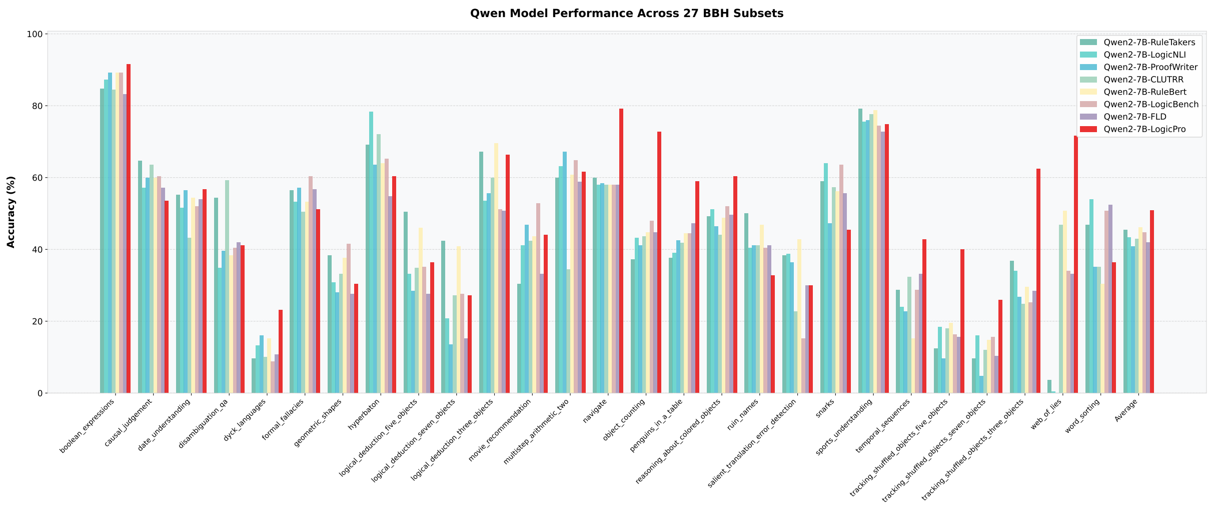
B Details of Analysis

Complete results of the ablation study The complete results on all benchmarks of the ablation study are presented in Table 8.

Complete results of the Performance Gain The complete results of Performance Gain on All Baseline Data on all baselines are in Table 9.

Model	BBH27	LogicBench	DROP	AR-LSAT	BoardgameQA	FOLIO	GSM8K	Multi-LogiEval	ProofWriter	MATH	Average
Qwen2-7B-RuleTakers	45.4	59.1	65.7	16.5	42.2	44.6	80.9	53.8	20.0	40.8	45.8
Qwen2-7B-LogicNLI	43.3	71.3	67.4	17.8	45.3	41.7	81.6	48.7	36.5	40.6	45.0
Qwen2-7B-ProofWriter	40.8	68.6	64.3	17.0	36.9	36.3	80.9	43.4	63.5	40.3	43.2
Qwen2-7B-CLUTRR	43.0	72.0	64.0	17.0	51.9	41.2	80.4	56.3	36.9	40.1	45.0
Qwen2-7B-RuleBert	46.2	69.1	67.3	17.8	43.8	40.7	81.7	59.3	34.6	40.7	47.3
Qwen2-7B-LogicBench	44.7	95.9*	67.4	17.8	41.4	38.7	82.1	67.3	18.0	41.3	46.6
Qwen2-7B-FLD	42.0	69.5	68.3	14.8	34.9	45.6	80.0	50.5	37.3	40.6	43.8
Qwen2-7B-LogicPro (ours)	50.9	73.5	68.3	19.1	48.1	46.1	81.5	62.0	28.5	41.6	51.2
Llama3-8B-RuleTakers	38.5	59.9	65.9	12.6	47.3	43.3	67.9	64.2	29.4	19.8	40.3
Llama3-8B-LogicNLI	40.4	54.0	65.3	12.6	41.0	56.0	69.3	62.3	35.0	19.8	41.8
Llama3-8B-ProofWriter	37.2	62.1	66.4	15.2	31.3	47.8	69.1	62.8	45.4	19.5	39.5
Llama3-8B-CLUTRR	40.5	61.1	66.6	10.4	43.7	56.0	69.5	63.1	35.1	20.0	42.2
Llama3-8B-RuleBert	34.7	48.8	66.5	15.2	43.6	51.5	68.9	65.4	28.2	19.5	37.4
Llama3-8B-LogicBench	41.0	93.5*	66.2	10.9	38.6	62.7	69.8	65.2	38.9	20.2	43.7
Llama3-8B-FLD	35.7	67.8	61.2	13.5	39.1	50.0	64.6	60.0	38.4	17.8	38.2
Llama3-8B-LogicPro (ours)	45.0	67.9	68.8	15.2	44.3	48.3	74.2	68.0	37.7	23.0	46.2
Qwen2-72B-RuleTakers	61.3	72.4	76.6	19.6	61.0	49.3	88.5	60.7	70.6	54.6	61.3
Qwen2-72B-LogicNLI	61.7	80.7	77.0	21.3	60.4	58.2	87.3	64.3	48.9	53.6	61.6
Qwen2-72B-ProofWriter	61.8	75.5	77.2	16.5	55.0	44.0	88.0	63.0	70.7	55.0	61.5
Qwen2-72B-CLUTRR	68.1	79.0	78.4	24.4	61.0	59.7	89.4	59.2	54.0	56.2	66.7
Qwen2-72B-RuleBert	67.8	74.1	76.5	19.6	55.2	62.7	88.0	71.7	60.2	55.2	66.5
Qwen2-72B-LogicBench	67.1	97.0*	77.9	24.8	57.3	60.5	88.4	87.9	57.2	55.4	67.2
Qwen2-72B-FLD	65.4	72.4	76.3	17.0	45.5	53.7	86.9	60.0	56.5	54.7	63.6
Qwen2-72B-LogicPro (ours)	72.4	81.7	79.6	27.4	66.4	55.2	89.1	72.1	54.3	55.8	70.4
Llama3-70B-RuleTakers	51.5	69.1	75.6	19.1	61.5	53.9	86.8	74.4	33.1	36.3	52.7
Llama3-70B-LogicNLI	58.5	69.9	78.0	17.8	58.5	58.3	84.5	67.5	31.3	34.5	57.8
Llama3-70B-ProofWriter	55.3	31.9	75.3	15.2	58.7	51.0	65.4	50.3	49.3	24.6	53.2
Llama3-70B-LogicPro (ours)	63.7	72.7	78.8	22.3	65.0	54.2	87.6	72.3	33.5	40.5	62.4

Table 6: Complete evaluation results for all benchmark.



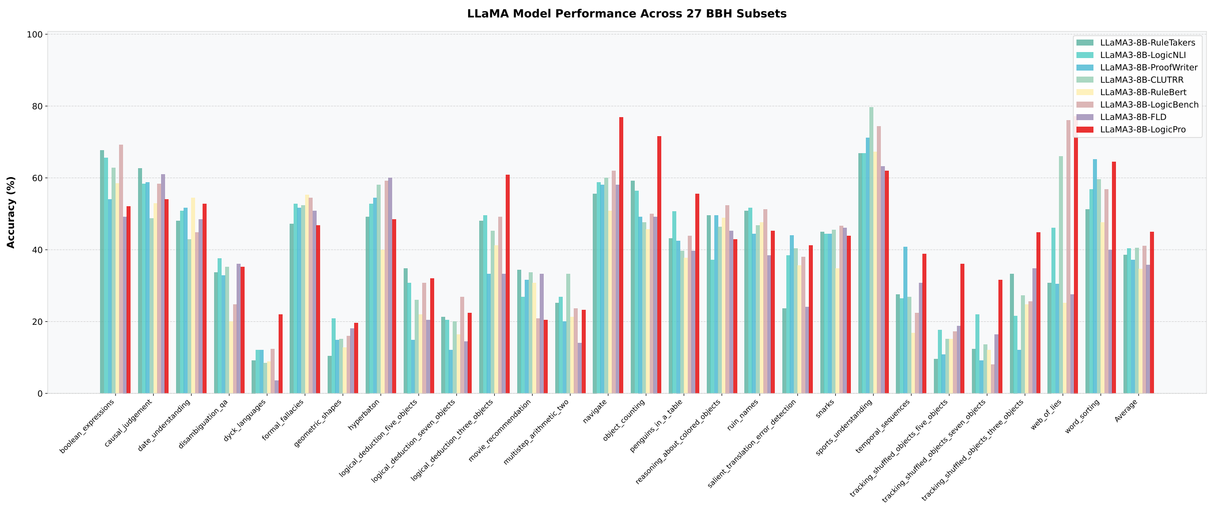


Figure 7: Qwen2-7B and Llama3-8B model performance across 27 BBH subset.

Model	BBH	LogicBench	DROP	ARLSAT	BoardgameQA	FOLIO	GSM8K	Multi-LogiEval	ProofWriter	MATH	Average
Qwen2-7B-base	41.4	68.0	65.3	16.8	42.4	38.96	79.4	52.9	28.2	40.9	43.1
Qwen2-7B-LogicPro	50.9	73.5	68.3	19.1	48.1	46.1	81.5	62.0	28.5	41.6	51.2
Model	**BBH**	**LogicBench**	**DROP**	**ARLSAT**	**BoardgameQA**	**FOLIO**	**GSM8K**	**Multi-LogiEval**	**ProofWriter**	**MATH**	**Average**
Llama3-8B-base	40.5	62.3	65.3	12.4	40.0	43.73	69.7	60.1	30.8	20.0	41.6
Llama3-8B-LogicPro	45.0	67.9	68.8	15.2	44.3	48.3	74.2	68.0	37.7	23.0	46.2

Table 7: Result of not adding any synthetic data (only two types of base data).

Qwen2-7B	BBH	LogicBench	DROP	AR-LSAT	BoardgameQA	FOLIO	GSM8K	Multi-LogiEval	ProofWriter	MATH	Average
Source_LeetCode	41.0	63.4	58.4	18.3	32.8	38.2	79.7	54.8	26.2	40.6	42.2
LogicPro-w/o.Inter-Var	48.2	69.6	62.9	16.5	47.6	50.7	80.5	62.9	30.0	40.1	49.0
LogicPro-w.Inter-Var	50.9	73.5	68.3	19.1	48.1	46.1	81.5	62.0	28.5	41.6	51.2
Llama3-8B	**BBH**	**LogicBench**	**DROP**	**AR-LSAT**	**BoardgameQA**	**FOLIO**	**GSM8K**	**Multi-LogiEval**	**ProofWriter**	**MATH**	**Average**
Source_LeetCode	36.6	51.4	62.1	12.8	38.9	48.2	69.5	62.4	31.6	20.5	38.5
LogicPro-w/o.Inter-Var	44.0	63.6	66.8	13.6	43.5	56.0	73.9	63.1	36.6	20.0	45.1
LogicPro-w.Inter-Var	45.0	67.9	68.8	15.2	44.3	48.3	74.2	68.0	37.7	23.0	46.2

Table 8: Complete results of the ablation study.

C Prompts

C.1 Complete Prompts for The Three Steps in The Main Methodology

The complete prompts for data collection and steps 1, 2, and 3 are in Figure 8, 9, 10, and 11.

C.2 Prompts in the remaining modules

The complete prompts for Problem-Program Consisitency Check and Solvability Check are in Figure 12 and 13.

In the evaluations, the prompt used for model evaluation is shown in Figure 14.

D LogicPro Examples

We give our synthetic data samples in figures 15 and 16, synthesised from two LeetCode algorithm questions, game 24 and stair climbing.

Qwen2-7B	BBH	LogicBench	DROP	ARLSAT	BoardgameQA	FOLIO	GSM8K	Multi-LogiEval	ProofWriter	MATH	Average
All_Baseline_Data	43.1	92.8	63.6	15.2	48.4	60.5	79.2	65.7	67.9	39.9	47.1
All_Baseline_Data + LogicPro	48.8	96.3	66.8	20.4	55.4	64.2	80.6	71.1	67.9	41.0	52.3
Llama3-8B	**BBH**	**LogicBench**	**DROP**	**AR-LSAT**	**BoardgameQA**	**FOLIO**	**GSM8K**	**Multi-LogiEval**	**ProofWriter**	**MATH**	**Average**
All_Baseline_Data	42.9	94.3	66.9	17.0	48.2	64.9	78.4	74.4	69.1	39.7	47.6
All_Baseline_Data + LogicPro	47.6	95.0	68.6	16.5	51.6	61.9	79.4	76.8	66.4	40.2	51.2

Table 9: Complete results of the Performance Gain.

Data Collection: Obtain Test Cases

I have an algorithmic problem and its python code, please help me construct 150 different test case inputs.
1. The constructed test case inputs need to fulfill the requirements of the algorithmic problem and be compatible with the provided Python code.
2. Please enclose the constructed test case inputs in the following python format; please enclose each test case input individually.

~~~
  ```python
  # Test case input 1
  # Your input here
  ```
  ```python
  # Test case input 2
  # Your input here
  ```
  ...
  ```python
  # Test case input 150
  # Your input here
  ```
~~~

3. Ensure that all test cases 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 cases adhere to the constraints provided in the problem description.
7. Provide only the input for the test cases, do not include the output.
8. Please do not omit the output. Give each specific test cases.

Input:
{souce_algorithmic_problem}
{source_python_solution}
Output:
{test_cases}

Figure 8: Complete prompts for Data Collection: construct test cases.

Step 1: Construct Complex Reasoning Problem

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

Instructions:
1. Please incorporate the test case 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 case.
3. Please randomly introduce some background information to diversify the question, such as mentioning a hypothetical scenario, a story, or real-life application related to the logical reasoning task.

Please rewrite the algorithmic question into a text-only logical reasoning question based on test case:
Input:
{souce_algorithmic_problem}
{test_case}
Output:
{synthetic_complex_reasoning_problem}

Figure 9: Complete prompts for step 1.

Step 2: Obtain Intermediate Variable From Program

Step 2.1: Obtain Test Case Specific Code
Please provide a Python code solution for the current complex logical reasoning problem, referring to the given Python code.
- The code should print the final result.
- Please give the final python code in the following format:
  ```python
  ```
- Please ensure that after running the code, the output result of the algorithm is returned through the variable `result`.

Input:
{synthetic_complex_reasoning_problem}
{source_python_solution}
Output:
{test_case_specific_code}

Step 2.2: Print Intermediate Variable
Please analyze and identify the key intermediate variables used in the Python code to solve complex logical reasoning problems. Based on your analysis, modify the code to print these key intermediate variables to simulate the step-by-step process of problem-solving as a human would do.
- Please give the final python code in the following format:
  ```python
  ```

Input:
{synthetic_complex_reasoning_problem}
{test_case_specific_code}
Output:
{test_case_specific_code_with_process_print}

Figure 10: Complete prompts for step 2.

Step 3: Program-Guided Reasoning

Please analyze and identify the key intermediate variables used in the Python code to solve complex logical reasoning problems. Based on your analysis, modify the code to print these key intermediate variables to simulate the step-by-step process of problem-solving as a human would do.
- Please give the final python code in the following format:
  ```python
  ```
- No need to output Example Output.

Input:
{synthetic_complex_reasoning_problem}
{intermediate_variable_output}
Output:
{final_reasoning_process}

Figure 11: Complete prompts for step 3.

Problem-Program Consistency Check

I have a programming problem statement and its corresponding code implementation. Both the problem and the code have been rewritten. Please carefully read the problem statement and the code, and verify if the rewritten code fully matches the rewritten problem description.

Follow these steps to complete the task:
1. **Understand the problem statement**: Summarize the requirements of the rewritten problem, including the input, output, and the functionality to be implemented.
2. **Analyze the code logic**: Explain what the code does, including how it handles input, its core logic/algorithm, and the output it produces.
3. **Check for consistency**: Compare the rewritten problem with the code and determine whether they match. If they don't match, identify the inconsistencies and explain why.

Return the result in **JSON format** with the following structure:

```json
{
"is_consistent": true, // or false
"issues": [          // If inconsistencies exist, list specific issues; if consistent, leave the array empty
   {
   "type": "LogicMismatch", // Example issue types: LogicMismatch (logic doesn't match), MissingDetails (problem lacks details), etc.
   "description": "Detailed description of the issue"
   }
]
}
```

Rewritten Problem Statement:
{synthetic_complex_reasoning_problem}

Rewritten Code:
{test_case_specific_code}

Please evaluate and return the verification result in the specified JSON format.

Figure 12: Prompts for Problem-Program Consisitency Check.

Solvability Check

Your task is to act as a professional reasoning evaluator. Your goal is to determine if the given problem is a valuable reasoning task. Consider the following two aspects:

1. **Problem Type**: The given problem must pertain to logical or inferential reasoning.
2. **Solvability**: The given reasoning problem must be solvable based on the conditions provided in the problem. (You may attempt to solve it step by step to verify its solvability.)

Please reason step by step and conclude with either ‘Yes’ or ‘No’.

Given Problem:
{synthetic_complex_reasoning_problem}

Figure 13: Prompts for Solvability Check.

Model Evaluation

[Instruction]
Please act as an impartial judge and evaluate the quality of the response provided by an AI assistant to the user question displayed below. Your evaluation should consider the correctness. You will be given a reference answer and the assistant's answer. Begin your evaluation by comparing the assistant's answer with the reference answer, you must identify whether the response is correct or wrong by strictly following this format: "<correct or wrong>", for example: "<correct>" or "<wrong>".

[Question]
{question}

[The Start of Reference Answer]
{answer}
[The End of Reference Answer]

[The Start of Assistant's Answer]
{completion}
[The End of Assistant's Answer]

Figure 14: Prompt for Model Evaluation.

Number Game: 24 Points

Question:

Imagine you are a participant in a mind-bending game show where the objective is to reach a target number using mathematical operations. You are given four cards, each displaying a number between 1 and 9.
Your task is to arrange these numbers into a mathematical expression using the operators `+`, `-`, `*`, and `/`, as well as parentheses `(` and `)`, in such a way that the result equals 24.
Keep the following rules in mind:

- The division operator `/` is used for real division, not integer division.
- Each operation must involve exactly two numbers (i.e., no unary operations are allowed).
- You cannot combine numbers to form multi-digit numbers.

Given the cards with numbers [9, 5, 8, 2], determine whether it is possible to create an expression that evaluates to 24. Can you devise such an expression, or prove that it's impossible to do so?

Answer:

Step 1: Understand the problem and the rules
The problem requires arranging four given numbers (9, 5, 8, 2) into a mathematical expression using the operators `+`, `-`, `*`, and `/`, as well as parentheses, to get a result equal to 24. The division operator `/` performs real division, each operation involves exactly two numbers, and numbers cannot be combined to form multi-digit numbers.

Step 2: Consider all possible combinations of numbers and operations
To find an expression that evaluates to 24, we need to consider all possible combinations of the given numbers and the allowed operations. This includes trying different orders of operations and different pairings of numbers for each operation.

Step 3: Start with the first two numbers and try all possible operations
Begin with the numbers 9 and 5. We can try adding, subtracting, multiplying, or dividing these numbers. The results are:

- Addition: 9 + 5 = 14
- Subtraction: 9 - 5 = 4
- Multiplication: 9 * 5 = 45
- Division: 9 / 5 = 1.8

Step 4: Continue with the next numbers and try all possible operations
Now, we have the results from the previous step (14, 4, 45, 1.8) and the remaining numbers (8, 2). We need to try all possible operations between these results and the remaining numbers.

Step 5: Analyze the results from Step 3 and continue with the most promising ones
From Step 3, we see that 14 is a promising result because it is closer to 24 than the other results. Let's continue with 14 and try operations with 8 and 2.

- Using 14 and 8: 14 + 8 = 22 (too high), 14 - 8 = 6 (too low), 14 * 8 = 112 (too high), 14 / 8 = 1.75 (not useful).
- Using 14 and 2: 14 + 2 = 16 (too high), 14 - 2 = 12 (too low), 14 * 2 = 28 (too high), 14 / 2 = 7 (not useful).

Step 6: Consider alternative combinations Since the direct approach with 14 did not yield a straightforward solution, let's consider alternative combinations, such as using 8 and 2 first.

- 8 + 2 = 10
- 8 - 2 = 6
- 8 * 2 = 16
- 8 / 2 = 4

Step 7: Combine the results from alternative combinations with the remaining numbers
Now, let's combine these results with 9 and 5.

- Using 10 (from 8 + 2) and 9: 10 + 9 = 19 (too high), 10 - 9 = 1 (too low), 10 * 9 = 90 (too high), 10 / 9 = 1.11 (not useful).
- Using 10 (from 8 + 2) and 5: 10 + 5 = 15 (too high), 10 - 5 = 5 (too low), 10 * 5 = 50 (too high), 10 / 5 = 2 (not useful).

Step 8: Re-evaluate the approach based on the given intermediate variables Given the intermediate variables, we see a path that leads to 24: 9 + 5 = 14, then 8 + 2 = 10, and finally 14 + 10 = 24. This suggests a specific order of operations that achieves the target.

The final answer is: $\boxed{Yes}$

Figure 15: Data Example for 24 Game.

Climbing Stairs

Question:
Imagine you are climbing a staircase that has 22 steps. Each time you take a step, you can either move up 1 step or 2 steps. In how many distinct ways can you reach the top of the staircase if you start from the bottom?
Consider the scenario where you are undergoing a physical fitness challenge that involves climbing stairs. The goal is to determine all possible step combinations you can take to reach the 22nd step, starting from the ground level. How many unique ways can you complete this challenge?

Answer:
To solve this problem, let's break it down into manageable steps and analyze the pattern of climbing the stairs.

Step 1: Understand the Problem
We have a staircase with 22 steps, and we can climb either 1 step or 2 steps at a time. We need to find out how many distinct ways we can reach the top of the staircase, starting from the bottom.

Step 2: Identify the Base Cases
- If there's only 1 step, there's only 1 way to climb it (1 step at a time).
- If there are 2 steps, there are 2 ways to climb them (either 2 steps at once or 1 step at a time, twice).

Step 3: Recognize the Pattern For any step beyond the second, the number of ways to reach that step is the sum of the number of ways to reach the step immediately before it (since we can climb 1 step from there) and the number of ways to reach the step 2 steps before (since we can climb 2 steps from there).

Step 4: Apply the Pattern to the Problem
Let's denote the number of ways to reach the nth step as W(n). Based on our pattern:
- W(1) = 1 (only one way to climb 1 step)
- W(2) = 2 (two ways to climb 2 steps: 1+1 or 2)
- For any n > 2, W(n) = W(n-1) + W(n-2) because we can reach the nth step from either the (n-1)th step (by taking 1 step) or the (n-2)th step (by taking 2 steps).

Step 5: Calculate for n = 22
We need to calculate W(22) using the formula W(n) = W(n-1) + W(n-2), starting from the base cases W(1) = 1 and W(2) = 2, and iteratively applying the formula until we reach W(22).

Step 6: Iterative Calculation
- W(3) = W(2) + W(1) = 2 + 1 = 3
- W(4) = W(3) + W(2) = 3 + 2 = 5
- W(5) = W(4) + W(3) = 5 + 3 = 8
- W(6) = W(5) + W(4) = 8 + 5 = 13
- W(7) = W(6) + W(5) = 13 + 8 = 21
- W(8) = W(7) + W(6) = 21 + 13 = 34
- W(9) = W(8) + W(7) = 34 + 21 = 55
- W(10) = W(9) + W(8) = 55 + 34 = 89
- W(11) = W(10) + W(9) = 89 + 55 = 144
- W(12) = W(11) + W(10) = 144 + 89 = 233
- W(13) = W(12) + W(11) = 233 + 144 = 377
- W(14) = W(13) + W(12) = 377 + 233 = 610
- W(15) = W(14) + W(13) = 610 + 377 = 987
- W(16) = W(15) + W(14) = 987 + 610 = 1597
- W(17) = W(16) + W(15) = 1597 + 987 = 2584
- W(18) = W(17) + W(16) = 2584 + 1597 = 4181
- W(19) = W(18) + W(17) = 4181 + 2584 = 6765
- W(20) = W(19) + W(18) = 6765 + 4181 = 10946
- W(21) = W(20) + W(19) = 10946 + 6765 = 17711
- W(22) = W(21) + W(20) = 17711 + 10946 = 28657

Step 7: Conclusion
There are 28657 distinct ways to climb a staircase of 22 steps, where you can climb either 1 step or 2 steps at a time, starting from the bottom.

Figure 16: Data Example for Climbing Stairs.