

ProgCo: Program Helps Self-Correction of Large Language Models

Xiaoshuai Song*, Yanan Wu[†], Weixun Wang, Jiaheng Liu, Wenbo Su, Bo Zheng

Taobao & Tmall Group of Alibaba

{songxiaoshuai.sxs,lixing.wyn}@alibaba-inc.com

Abstract

Self-Correction aims to enable large language models (LLMs) to self-verify and self-refine their initial responses without external feedback. However, LLMs often fail to effectively self-verify and generate correct feedback, further misleading refinement and leading to the failure of self-correction, especially in complex reasoning tasks. In this paper, we propose **Program-driven Self-Correction (ProgCo)**. First, **program-driven verification (ProgVe)** achieves complex verification logic and extensive validation through self-generated, self-executing verification pseudo-programs. Then, **program-driven refinement (ProgRe)** receives feedback from ProgVe, conducts dual reflection and refinement on both responses and verification programs to mitigate misleading of incorrect feedback in complex reasoning tasks. Experiments on three instruction-following and mathematical benchmarks indicate that ProgCo achieves effective self-correction, and can be further enhance performance when combined with real program tools. We release our code at <https://github.com/songxiaoshuai/progco>.

1 Introduction

Although large language models (LLMs) have shown excellent performance on certain tasks, they still face challenges such as hallucinations and unfaithful reasoning when solving complex instruction-following and reasoning tasks (Zhao et al., 2023; Chang et al., 2024). Self-correction is an expected capability of LLMs, wherein the LLM first needs to reflect on its initial output, identify potential issues and generate feedback (*self-verification* phase), which then guides the LLM to optimize and refine its output (*self-refinement* phase), as illustrated in Fig 1 (Pan et al., 2024; Kumar et al., 2024). However, studies have shown

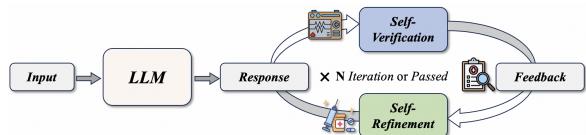


Figure 1: Illustration of a typical workflow of LLM’s intrinsic self-correction.

that current LLMs severely lack this capability and struggle to achieve effective self-correction in the absence of external feedback (also called *intrinsic self-correction*), particularly in complex tasks (Li et al., 2024; Huang et al., 2024; Tyen et al., 2024; Kamoi et al., 2024).

A core reason for the failure of self-correction in LLMs is their inability to effectively self-detect problematic outputs and generate high-quality feedback (Tyen et al., 2024). Existing works mainly used two approaches for self-verification: (1) prompting LLMs to perform step-by-step self-checks (Madaan et al., 2024), or (2) generating a checklist based on the task and then checking responses against this list (Zhang et al., 2024c; Cook et al., 2024). However, on one hand, LLMs often exhibit overconfidence, making it difficult for them to identify their own errors or hallucinations, resulting in a low recall of incorrect responses (Zhang et al., 2024c). On the other hand, for complex tasks, these methods struggle to parse intricate verification logic. For instance, checklists usually only express parallel relationships and examine superficial issues. Ineffective verification further leads to ineffective refinement. The inaccurate and low-quality error detection and feedback not only makes it challenging for LLMs to correct erroneous outputs in one attempt but also seriously mislead them into modifying from correct to incorrect for misrecalled responses. To overcome this issue, we propose **Program-driven Self-Correction (ProgCo)**, achieving effective self-correction by incorporating self-generated and self-executed programs in the

*Work done during internship at Alibaba Inc.

[†]Corresponding Author: Yanan Wu

verification and refinement stages.

We first introduce **Program-driven Verification (ProgVe)** to achieve better self-verification. Different from studies like PAL and POT (Gao et al., 2023; Chen et al., 2023; Gou et al., 2024), which integrate program or symbolic solvers for forward reasoning, ProgVe focuses on the self and reverse verification phase. In general, the LLM is initially prompted to generate a pseudo verification program for the input task. After obtaining the initial response, the LLM further acts as a program-executor, executing the verification program step by step to obtain the verification results. The motivation for ProgVe stems from: (1) Compared to the ambiguity of natural language, code program can express more complex verification logic and structures. (2) Using LLM as a program-executor not only allows a focus on the verification logic of the code without requiring strict executability, but also incorporate LLM’s own knowledge and causal understanding into execution, such as the virtual function `is_structured_as_letter` in Fig 2.

To address misleading self-refinement caused by incorrect self-verification in complex reasoning tasks such as mathematics, we further introduce **Program-driven Refinement (ProgRe)**, featuring a dual refinement mechanism for both response and program. Within the framework, to avoid directly misleading of feedback, the response revised based on feedback is treated as an intermediate and contrasted with the pre-revision response to identify differences and generate insights, which will help in regenerating the final refined response. To more fundamentally address incorrect verification, besides refining response with program, the verification program is also reflected upon and optimized with the assistance of response information.

We provide a detailed exposition of proposed method in Section 2. In Section 3, we demonstrate that our method can achieve effective self-correction and outperforms all baselines in correcting instruction-following and mathematical reasoning tasks. A series of analyses further provides insights into self-correction and our method from multiple perspectives.

In summary, we propose ProgCo for effective self-correction, consisting of two components: ProgVe and ProgRe. Our contributions are three-fold: (1) We propose ProgVe, a method enabling LLMs to self-generate and self-execute validation programs for self-verification. (2) We propose ProgRe, a self-refinement method that is robust to in-

Instruction: Write a letter to a friend in all lowercase letters ask them to go and vote.
<pre>def validate_response(response): errors = [] if not response.islower(): errors.append("The response is not in all lowercase letters.") if not is_structured_as_letter(response): errors.append("The response is not structured as a letter.") ... if errors: return False, "\n".join(errors) return True, "No Error"</pre>
Question: Kyle bought last year's best-selling book for \$19.50. This is with a 25% discount from the original price. What was the original price of the book?
<pre>def verify_original_book_price(original_price): # Known conditions discounted_price = 19.50 discount_percentage = 0.25 # Calculate the discount amount discount_amount = original_price * discount_percentage # Calculate the price after discount calculated_discounted_price = original_price - discount_amount # Check if the calculated discounted price matches the known price if calculated_discounted_price != discounted_price: return False return True</pre>

Figure 2: Illustration of generating verification pseudo-program for input tasks.

correct feedback and combines dual optimization of response and verification programs. (3) Experiments and analyses on three datasets verify the effectiveness of our method in self-correction.

2 Method

2.1 Program-driven Verification

Given model M and input x , in the i -th iteration of self-correction, the goal of self-verification is to generate feedback fb_i for response y_i , indicating whether y_i passes verification or the reason it fails.

Verification Program Generation. After obtaining the initial response $y_0 = M(x)$, we first use prompt P_{fuc}^{gen} to guide M in generating a verification pseudo-program function f . This process is independent of y_0 to ensure a different perspective and avoid biases from the response:

$$f = M(P_{fuc}^{gen} || x) \quad (1)$$

As illustrated in Fig 2, for instruction-following task, f verifies a series of constraints extracted from x . For mathematical problems, f starts from the output answer and uses reverse reasoning to verify step-by-step whether it contradicts the given conditions in x .

Verification Program Execution. For each round, we use prompt P_{fuc}^{exec} to instruct M act as a code executor, taking y_i as input, executing f step by step to obtain the execution result r_i . r_i is further converted to feedback fb_i by prompt P_{fb} :

$$r_i = M(P_{fuc}^{exec} || x || y_i), fb_i = M(P_{fb} || x || y_i || r_i) \quad (2)$$

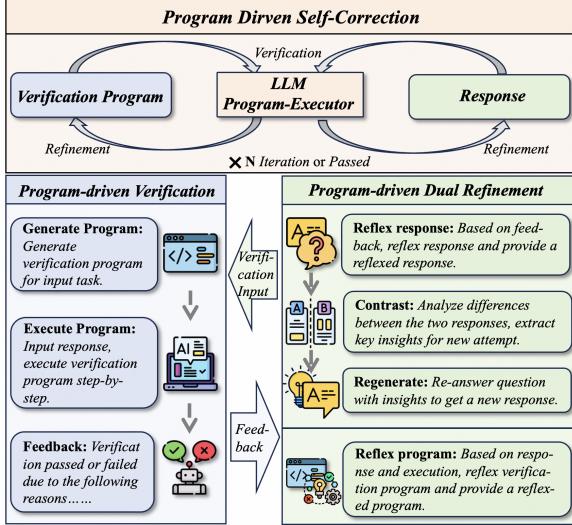


Figure 3: The overall framework of ProgCo, achieving self-correction through iterative ProgVe and ProgRe.

If the verification passes, self-correction stops and yields the final output $y_{final} = y_i$. Otherwise, if fb_i indicates that y_i fails to meet constraints or contains contradictions, the process enters the self-refinement stage until maximum rounds I are reached.

2.2 Program-driven Refinement

The **vanilla self-refinement** generates a new output $y_{i+1} = M(p_{refine} || x || y_i || fb_i)$ through prompt p_{refine} . This method is effective for tasks like instruction following, as constraints are apparent, and the correct program f and clear feedback fb_i are easily obtained. However, in reasoning tasks such as mathematics, the correct f and fb_i are difficult to obtain in one attempt and will mislead refinement, as described in Section 1. Therefore, we introduce the Program-driven Refinement framework to address this, as shown in Fig 3.

Preliminary Reflection. Unlike directly refining y_i using fb_i , we prompt M to first reflect on y_i under the feedback and then output a temporary response y_{i+1}^{temp} , which can either maintain y_i unchanged or update it:

$$y_{i+1}^{temp} = M(P_{reflex} || x || y_i || fb_i) \quad (3)$$

Contrast and Regenerate. If a new answer is obtained, y_{i+1}^{temp} is further compared with y_i to identify differences. These differences will be transformed into insights ins for problem-solving, assisting in regenerating the refined response:

$$ins = M(P_{cont} || y_i || y_{i+1}^{temp}), y_{i+1} = M(ins || x) \quad (4)$$

Verification Program Refinement. To correct potentially incorrect validation code generated in the initial round, we further introduce self-refinement of the validation code. Utilizing information from y_i and fb_i , the validation code f_i will self-reflect and generate new validation code f_{i+1} for next round of self-verification:

$$f_{i+1} = M(P_{reflex}^{code} || x || f_i || y_i || fb_i) \quad (5)$$

We provide detailed pseudo-code and prompts used in the proposed method in Appendix B.

3 Experiment

3.1 Experiment Setup

We evaluate ProgCo on the instruction-following dataset IFEval (Zhou et al., 2023) and the mathematics datasets GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021)¹. Since complex reasoning is not involved, we only use the combination of ProgVe and vanilla refinement for IFEval. Additionally, we calculate the score for each self-correction round individually, without assuming the next round is correct just because the previous one was. This may differ from some evaluation settings. We provide detailed datasets, baseline introductions, and implementation details in Appendix C.

3.2 Main Result

Table 1 shows the performance comparison between ProgCo and different self-correction baselines.² Overall, ProgCo outperforms all baselines with a large margin across three benchmarks. On GPT-3.5, ProgCo improves over the initial response by 4.62% (IFEval(Pr)), 3.23% (IFEval(Ins)), 5.84% (GSM8K), and 5.8% (MATH) with just one round of self-correction. After three rounds, the improvements further increase to 4.80% (IFEval(Pr)), 3.47% (IFEval(Pr)), 7.28% (GSM8K) and 8.0% (MATH). Similar improvements are observed on other LLMs. From a task perspective, many baselines achieve positive improvements on IFEval but failed on mathematical tasks, while our method achieves significant positive improvements on both

¹We randomly sample 500 instances from MATH test set.

²Due to varied focuses, we discuss various inference methods in Appendix A and compare the self-consistency sampling methods in Section 3.6.

Method	Llama3.1-8B-Instruct				GPT-3.5				GPT-4o			
	IF (Pr)	IF (Ins)	GSM8K	MATH	IF (Pr)	IF (Ins)	GSM8K	MATH	IF (Pr)	IF (Ins)	GSM8K	MATH
Initial Score	73.75	81.29	85.82	46.8	58.23	68.35	76.5	36.2	82.99	87.52	95.07	77.2
<i>Maximum One Round of Self-Correction</i>												
Vanilla-reflex	68.39 _{-5.36}	76.74 _{-4.55}	83.32 _{-2.50}	44.6 _{-2.20}	54.9 _{-3.33}	69.18 _{+0.83}	72.4 _{-4.10}	35.6 _{-0.60}	83.73 _{+0.74}	88.37 _{+0.85}	93.31 _{-1.76}	77.6 _{+0.40}
Self-Refine	75.79 _{+2.04}	82.49 _{+1.20}	84.69 _{-1.13}	40.6 _{-6.20}	59.33 _{+1.10}	68.94 _{+0.59}	76.35 _{-0.15}	38.0 _{+1.80}	84.66 _{+1.67}	88.97 _{+1.45}	94.92 _{-0.15}	75.4 _{-1.80}
Self-Reflection	73.38 _{-0.37}	80.22 _{-1.07}	84.23 _{-1.50}	43.8 _{-3.00}	59.7 _{+1.47}	68.94 _{+0.59}	71.95 _{-4.55}	34.8 _{-4.40}	84.84 _{+1.85}	88.97 _{+1.45}	95.0 _{-0.07}	77.6 _{+0.40}
CheckList	74.49 _{+0.74}	81.89 _{+0.60}	84.76 _{-1.06}	47.0 _{+0.20}	59.15 _{+0.92}	69.06 _{+0.71}	77.63 _{+1.13}	36.0 _{-0.20}	85.4 _{+2.41}	89.45 _{+1.93}	94.84 _{-0.23}	77.4 _{+0.20}
ProgCo (Ours)	76.34_{+2.59}	83.69_{+2.40}	86.58_{+0.76}	50.2_{+3.40}	62.85_{+4.62}	71.58_{+3.23}	82.34_{+5.84}	42.0_{+5.80}	87.99_{+5.00}	91.85_{+4.33}	95.75_{+0.68}	79.4_{+2.20}
<i>Maximum Three Rounds of Self-Correction</i>												
Vanilla-reflex	65.25 _{-8.50}	74.82 _{-6.47}	81.43 _{-4.39}	44.6 _{-2.20}	55.27 _{-2.96}	65.83 _{-2.52}	71.8 _{-4.70}	35.8 _{-0.40}	84.29 _{+1.30}	88.73 _{+1.21}	94.08 _{-0.99}	79.2 _{+2.00}
Self-Refine	76.34 _{+2.59}	82.73 _{+1.44}	84.99 _{-0.83}	43.4 _{-3.40}	59.7 _{+1.47}	69.06 _{+0.71}	76.35 _{-0.15}	38.0 _{+1.80}	85.03 _{+2.04}	89.45 _{+1.93}	94.92 _{-0.15}	76.4 _{-0.80}
Self-Reflection	73.75 _{+0.00}	81.06 _{-0.23}	84.53 _{-1.29}	48.0 _{+1.20}	59.52 _{+1.29}	69.18 _{+0.83}	74.07 _{-2.43}	35.6 _{-0.60}	85.77 _{+2.78}	89.69 _{+2.17}	94.92 _{-0.15}	78.4 _{+1.20}
CheckList	75.05 _{+1.30}	82.61 _{+1.32}	84.91 _{-0.91}	47.4 _{+0.60}	59.52 _{+1.29}	69.3 _{+0.95}	77.79 _{+1.29}	35.8 _{-0.40}	85.77 _{+2.78}	89.69 _{+2.17}	95.0 _{-0.07}	77.0 _{-0.20}
ProgCo (Ours)	77.82_{+4.07}	84.29_{+3.00}	87.41_{+1.59}	50.6_{+3.80}	63.03_{+4.80}	71.82_{+3.47}	83.78_{+7.28}	44.2_{+8.00}	87.8_{+4.81}	91.97_{+4.45}	95.75_{+0.68}	80.0_{+2.80}

Table 1: The result of different self-correction methods. The metric for GSM8K and MATH is accuracy, IF (Pr) and IF (Ins) denote IFEval’s strict prompt and instruction metrics, respectively. \pm indicates change from initial score.

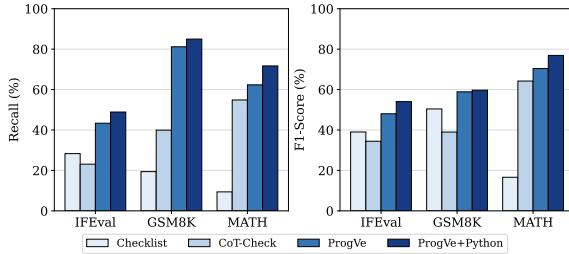


Figure 4: Recall and F1 scores of self-verification methods for incorrect responses on GPT-3.5.

GSM8K and MATH, demonstrating its effectiveness in complex reasoning.

3.3 Ablation Analysis

Recall of Self-Verification. Fig 4 shows the recall and F1-score for CoT-Check, Checklist, and ProgVe in identifying incorrect responses. ProgVe outperforms the baselines in both Recall and F1-score significantly, with further improvements when combined with Python tool. This demonstrates the advantages of using self-play programs in verification, including expressing more complex structures, providing a different perspective, and integrating with symbolic tools.

Ablation on Self-Refinement. We conduct an ablation analysis of ProgRe in Table 2. First, removing contrast and regeneration leads to a significant increase in the ratio of correction to incorrection, proving that this strategy can effectively alleviate feedback misleading. Second, since wrong verification programs continually trap responses from passing verification, removing the reflection of program leads to a significant increase in the average rounds of self-correction. Lastly, without program feedback involved in response reflection,

Method	ACC \uparrow	Avg Turn \downarrow	$\Delta^{i \rightarrow c} \uparrow$	$\Delta^{c \rightarrow i} \downarrow$
ProgVe	83.78	0.88	47.75	15.42
w/o cont. & regen.	79.08 _{-4.7}	0.87 _{-0.01}	43.88 _{-3.87}	35.18 _{+19.76}
w/o program reflex	83.02 _{-0.76}	1.06 _{-0.18}	44.98 _{-2.77}	15.95 _{+0.53}
w/o feedback	81.35 _{-2.43}	0.90 _{+0.02}	30.5 _{-17.25}	7.36-8.06

Table 2: Ablation on GPT-3.5 with maximum three rounds of self-correction. Avg Turn is the average rounds of self-refinement for all samples. $\Delta^{i \rightarrow c}$ is the ratio of incorrect-to-correct transitions among recalled ground-truth incorrect samples, and $\Delta^{c \rightarrow i}$ is the opposite for recalled correct samples. Avg Turn denotes the average ProgVe turns for all samples.

Model	IFEval (Pr)		MATH (ACC)	
	ProgVe	ProgVe+Python	ProgVe	ProgVe+Python
GPT-3.5	62.0	64.14_{+2.14}	44.2	44.6_{+0.4}
GPT-4o	87.8	91.31_{+3.51}	80.0	81.2_{+1.2}

Table 3: Performance of ProgCo in integrating the Python executor tool during the ProgVe process.

the ratios of both correct and incorrect refinement notably decrease, indicating issues of LLM’s overconfidence and misleading feedback, respectively.

3.4 Integrating symbolic tools for LLM program-executor.

Due to the advantage of program easily integrating with symbolic tools, we further indicate in Prompt P_{fuc}^{exec} that LLM executor can delegate complex numerical operations to an actual python tool to overcome the shortcomings of LLM. As shown in Table 3, this further improves ProgCo’s performance. For example, ProgVe obtains more precise feedback on constraints such as word count and keywords, thereby significantly improving the performance of IFEval.

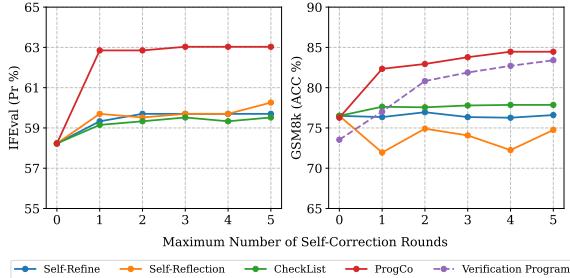


Figure 5: Score variation with the maximum number of self-correction rounds on GPT-3.5. The program’s ACC measures the consistency accuracy between the execution results of ProgVe and ground-truth scores.

3.5 Effect of Self-Correction Iterations.

As shown in Fig 5, with increasing self-correction rounds, baselines’ performance either slightly improves or fluctuates near the initial score. In contrast, ProgCo shows a consistent upward trend, especially in the first few rounds. The continual improvement in the verification program’s consistency accuracy further highlights the advantages of dual refinement.

3.6 Comparison with Self-Consistency

We choose the self-consistency (SC) as the sampling approach. For multiple sampled responses, SC-Vote determines the final answer based on majority voting, SC-Reflex reflects to arrive at a final response, and SC-Select prompts the LLM to compare and select one as the final response. The results are shown in Table 4. On IFEval, SC-Vote is not applicable, and both SC-Reflex and SC-Select lead to poorer performance, indicating that sampling is not suitable for tasks like instruction-following. In contrast, ProgCo significantly improves instruction-following performance. On mathematical tasks where sampling excels, ProgCo with one correction round outperforms 3-sampling significantly and 5-sampling on MATH. ProgCo with five correction rounds matches 10-sampling on GSM8K and slightly surpasses it on MATH. It should be emphasized that, due to early stopping from ProgVe, ProgRe only performs an average of about 1.2 rounds of self-refinement (for instance, many cases pass self-verification without entering the self-refinement phase).

3.7 Case Study

We present examples of ProgVe performing self-verification on IFEval and GSM8K in Figures 12 and 13, respectively. Figure 14

Method	MAX Turn	IFEval(Pr)	GSM8K	MATH
Initial Score	–	58.23	76.5	36.2
SC-Vote	3	–	79.45	37.4
	5	–	83.02	40.8
	10	–	84.46	44.2
SC-Reflex	3	41.96	79.61	38.2
	5	42.88	81.73	40.4
	10	44.36	84.08	41.6
SC-Select	3	55.82	77.41	36.2
	5	55.64	78.24	38.6
	10	56.19	78.09	36.0
Ours	1	62.85 (0.34 turn _{avg})	82.34 (0.41 turn _{avg})	42.0 (0.49 turn _{avg})
	3	63.03 (0.80 turn _{avg})	83.78 (0.88 turn _{avg})	44.2 (0.97 turn _{avg})
	5	63.03 (1.18 turn _{avg})	84.46 (1.13 turn _{avg})	44.4 (1.34 turn _{avg})

Table 4: Comparison of ProgCo and the Self-Consistency (SC) on GPT-3.5. The temperature for SC is 0.7. Turn_{avg} denotes the average self-refinement (ProgRe) iterations for all samples.

shows an example of PogRe performing self-refinement on GSM8K. In Fig 12, the verification function not only generates executable programs such as `response!=response.upper()`, but also generates more vaguely defined programs like `is_english(response)` and `has_title(response)`. However, the LLM is able to successfully execute these pseudo-programs, surpassing the capabilities of a real program executor. In Fig 13, the verification program verifies the correctness of the response in reverse. It starts with the predicted `remaining_speed`, calculates the `average_speed` step by step, and checks whether it matches the given condition `target_average_speed`. In Fig 14, although the response still contains errors upon initial reflection, the LLM ultimately provides the correct answer to the question with the help of contrasted insights. Meanwhile, the verification program is also further optimized during the refinement process.

4 Conclusion

In this paper, we propose ProgCo, a program-driven self-correction method. ProgCo first self-generates and self-executes verification pseudo-programs for self-verification (ProgVe), then uses dual reflection and refinement of responses and programs for self-refinement (ProgRe). Experiments and analyses on three benchmarks demonstrate the effectiveness of ProgCo in self-correction.

Limitations

In this paper, we propose ProgCo and experimentally validate its effective self-correction. However, there are still some limitations as follows: (1) In terms of application scenarios, although using pseudo-program and LLM executors can extend the application scope beyond numerical and symbolic solving tasks, we primarily validated the effectiveness of ProgCo in instruction-following and mathematical tasks. (2) One advantage of using LLMs in executing verification programs is the integration of their own knowledge and causal logic. However, they are limited in large and precise numerical calculations. This issue can be mitigated by combining real symbolic tools, as shown in the experiment in Table 3. (3) Due to the lack of specialized training, we use detailed prompts to guide the LLM in completing tasks in ProgCo, which results in additional inference costs. Synthesizing data for each component of ProgCo and jointly training the LLM can replace the need for prompts and demonstration costs during inference.

References

- Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45.
- Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *Preprint*, arXiv:2110.14168.
- Jonathan Cook, Tim Rocktäschel, Jakob Foerster, Dennis Aumiller, and Alex Wang. 2024. Ticking all the boxes: Generated checklists improve llm evaluation and generation. *Preprint*, arXiv:2410.03608.
- Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. 2024. Chain-of-verification reduces hallucination in large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3563–3578, Bangkok, Thailand. Association for Computational Linguistics.
- Guanting Dong, Keming Lu, Chengpeng Li, Tingyu Xia, Bowen Yu, Chang Zhou, and Jingren Zhou. 2024a. Self-play with execution feedback: Improving instruction-following capabilities of large language models. *Preprint*, arXiv:2406.13542.
- Guanting Dong, Xiaoshuai Song, Yutao Zhu, Runqi Qiao, Zhicheng Dou, and Ji-Rong Wen. 2024b. Toward general instruction-following alignment for retrieval-augmented generation. *arXiv preprint arXiv:2410.09584*.
- Guanting Dong, Chenghao Zhang, Mengjie Deng, Yutao Zhu, Zhicheng Dou, and Ji-Rong Wen. 2024c. Progressive multimodal reasoning via active retrieval. *arXiv preprint arXiv:2412.14835*.
- Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.
- Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu Chen. 2024. ToRA: A tool-integrated reasoning agent for mathematical problem solving. In *The Twelfth International Conference on Learning Representations*.
- Haixia Han, Jiaqing Liang, Jie Shi, Qianyu He, and Yanghua Xiao. 2024. Small language model can self-correct. *Preprint*, arXiv:2401.07301.
- Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *NeurIPS*.
- Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*.
- Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. 2024. When can LLMs actually correct their own mistakes? a critical survey of self-correction of LLMs. *Transactions of the Association for Computational Linguistics*, 12:1417–1440.
- Geunwoo Kim, Pierre Baldi, and Stephen McAleer. 2024. Language models can solve computer tasks. *Advances in Neural Information Processing Systems*, 36.

- Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. 2024. *Training language models to self-correct via reinforcement learning*. Preprint, arXiv:2409.12917.
- Yanhong Li, Chenghao Yang, and Allyson Ettinger. 2024. *When hindsight is not 20/20: Testing limits on reflective thinking in large language models*. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 3741–3753, Mexico City, Mexico. Association for Computational Linguistics.
- Chenyang Lyu, Lecheng Yan, Rui Xing, Wenxi Li, Younes Samih, Tianbo Ji, and Longyue Wang. 2024. *Large language models as code executors: An exploratory study*. Preprint, arXiv:2410.06667.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.
- Meta. 2024. *Introducing llama 3.1: Our most capable models to date*.
- OpenAI. 2024. *Hello gpt-4o*.
- Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2024. *Automatically correcting large language models: Surveying the landscape of diverse automated correction strategies*. *Transactions of the Association for Computational Linguistics*, 12:484–506.
- Runqi Qiao, Qiuna Tan, Guanting Dong, Minhui Wu, Chong Sun, Xiaoshuai Song, Zhuoma GongQue, Shanglin Lei, Zhe Wei, Miaoxuan Zhang, et al. 2024a. We-math: Does your large multimodal model achieve human-like mathematical reasoning? *arXiv preprint arXiv:2407.01284*.
- Shuofei Qiao, Honghao Gui, Chengfei Lv, Qianghuai Jia, Huajun Chen, and Ningyu Zhang. 2024b. *Making language models better tool learners with execution feedback*. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3550–3568, Mexico City, Mexico. Association for Computational Linguistics.
- Matthew Renze and Erhan Guven. 2024. Self-reflection in llm agents: Effects on problem-solving performance. *arXiv preprint arXiv:2405.06682*.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Re-flexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.
- Gladys Tyen, Hassan Mansoor, Victor Carbune, Peter Chen, and Tony Mak. 2024. *LLMs cannot find reasoning errors, but can correct them given the error location*. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13894–13908, Bangkok, Thailand. Association for Computational Linguistics.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.
- 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.
- Zhenyu Wu, Qingkai Zeng, Zhihan Zhang, Zhaoxuan Tan, Chao Shen, and Meng Jiang. 2024. *Large language models can self-correct with key condition verification*. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 12846–12867, Miami, Florida, USA. Association for Computational Linguistics.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024a. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.
- Wenlin Yao, Haitao Mi, and Dong Yu. 2024b. Hdflow: Enhancing llm complex problem-solving with hybrid thinking and dynamic workflows. *arXiv preprint arXiv:2409.17433*.
- Che Zhang, Zhenyang Xiao, Chengcheng Han, Yixin Lian, and Yuejian Fang. 2024a. *Learning to check: Unleashing potentials for self-correction in large language models*. Preprint, arXiv:2402.13035.
- Di Zhang, Xiaoshui Huang, Dongzhan Zhou, Yuqiang Li, and Wanli Ouyang. 2024b. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b. *arXiv preprint arXiv:2406.07394*.
- Wenqi Zhang, Yongliang Shen, Linjuan Wu, Qiuying Peng, Jun Wang, Yueling Zhuang, and Weiming Lu. 2024c. *Self-contrast: Better reflection through inconsistent solving perspectives*. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3602–3622, Bangkok, Thailand. Association for Computational Linguistics.
- Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. 2023. [Instruction-following evaluation for large language models](#). *Preprint*, arXiv:2311.07911.

A Related Work

Self-Correction. Self-correction aims to enable LLMs to achieve the ability to self-check and correct its outputs (Pan et al., 2024). Although some work (Shinn et al., 2024; Renze and Guven, 2024) achieves correction by relying on environmental feedback (such as True/False signals), many studies (Li et al., 2024; Huang et al., 2024; Tyen et al., 2024; Kamoi et al., 2024) have shown that in the complete absence of environmental feedback, LLMs find it difficult to engage in effective self-reflection, with a particular emphasis on the challenge of identifying their own errors. To this end, some work (Han et al., 2024; Zhang et al., 2024a; Kumar et al., 2024) focuses on enhancing LLMs’ self-correction capabilities during the training phase through imitation learning or reinforcement learning, while another part (Madaan et al., 2024; Dhuliawala et al., 2024; Wu et al., 2024; Zhang et al., 2024c; Kim et al., 2024) focuses on designing reflection or correction pipelines during the inference phase to help LLMs examine and analyze their own outputs. In this work, we focus on the inference phase, and to the best of our knowledge, we are the first to introduce self-generated and self-executed pseudo-verification programs into self-verification and self-refinement, achieving effective self-correction.

Integration of LLM with Programs. Several studies have enhanced LLMs by introducing programs or symbolic solvers. Some works integrate code executor or symbolic solving tools within the LLM’s forward reasoning to address mathematical or symbolic reasoning problems, such as PAL (Gao et al., 2023), POT (Chen et al., 2023), and ToRA (Gou et al., 2024). Others (Chen et al., 2024; Dong et al., 2024b; Qiao et al., 2024b; Dong et al., 2024a) use programs to assist in data synthesis or training; for example, AutoIf (Dong et al., 2024a) filters synthetic training data through testing programs. Additionally, Lyu et al. (2024) explore the capability of LLMs to act as code interpreters to execute LeetCode programs. Distinguishing these studies, our work is unique in that it: (1) employs verification programs for the self-correction phase; (2) utilizes pseudo-programs that do not require strict executability; (3) enables LLMs to

self-generate and self-execute programs without the necessity for actual symbolic tools.

Inference Framework. Unlike directly generating answers, many studies have explored enhancing LLMs’ ability to solve complex tasks through reasoning frameworks. Decomposition-based methods like COT, TOT, and POT (Wei et al., 2022; Yao et al., 2024a; Chen et al., 2023; Qiao et al., 2024a) guide models to break down tasks step-by-step, while sampling-based methods (Wang et al., 2022; Zhang et al., 2024b; Dong et al., 2024c) explore diverse reasoning paths and select the most consistent or optimal response. Another key approach is the mechanism of reflection and self-correction (Zhang et al., 2024c; Shinn et al., 2024; Pan et al., 2024), which encourages models to iteratively evaluate and refine their responses. Additionally, to balance efficiency, some works (Yao et al., 2024b) dynamically combine fast and slow reasoning based on task complexity. ProgCo can be seen as a method of fast and slow reasoning, where RrogVe filters out complex tasks, and ProgRe uses dual optimization for slow reasoning. Moreover, since ProgCo focuses on the self-correction, it is orthogonal to forward reasoning methods like sampling and decomposition, and can be well combined with them.

B Details of Method

B.1 Details of Prompts

We present the prompts used for mathematical tasks in Figures 6, 7, 8, 9, 10, 11.

B.2 Pseudo-code

We summarize the pseudo-code of ProgCo in Algorithm 1.

C Details of Experiment

C.1 Details of Datasets

The introduction to the evaluation datasets is as follows:

- **IFEval** (Zhou et al., 2023): IFEval is one of the most commonly used instruction-following benchmark for LLMs, containing over 500 test samples and covering 25 types of atomic instructions. The Prompt(Strict) metric calculates the proportion of strict following of the input prompt across all samples. For a sample, the prompt is considered followed only if all atomic instructions in it are

Algorithm 1 : Program-driven Self-Correction (ProgCo)

Input: input x , model M , prompts $\{P_{fuc}^{gen}, P_{fuc}^{exec}, P_{fb}, P_{reflex}, P_{cont}, P_{reflex}^{code}\}$

Output: final response y_{final}

```
1:  $y_0 = M(x)$                                      ▷ Initial generation
2: for iteration  $i \in 0, 1, \dots, I$  do
3:    $f_i = M(P_{fuc}^{gen} || x)$                    ▷ Verification Program Generation (Eqn.1)
4:    $r_i = M(P_{fuc}^{exec} || x || y_i)$            ▷ Verification Program Execution (Eqn.2)
5:    $fb_i = M(P_{fb} || x || y_i || r_i)$ 
6:   if  $fb_i$  = “pass” then
7:     break
8:   else
9:      $y_{i+1}^{temp} = M(P_{reflex} || x || y_i || fb_i)$  ▷ Preliminary Reflection (Eqn.3)
10:    if  $y_{i+1}^{temp} \neq y_i$  then
11:       $ins = M(P_{cont} || y_i || y_{i+1}^{temp})$           ▷ Contrast and Regenerate (Eqn.4)
12:       $y_{i+1} = M(ins || x)$ 
13:    else
14:       $y_{i+1} = y_{i+1}^{temp}$ 
15:    end if
16:     $y_{i+1} = M(P_{reflex}^{code} || x || f_i || y_i || fb_i)$  ▷ Verification Program Refinement (Eqn.5)
17:  end if
18: end for
19:  $y_{final} = y_i$ 
```

followed. The Instruction(Strict) metric calculates the proportion of strict following of each atomic instruction across all samples.

- **GSM8K** (Cobbe et al., 2021): GSM8K is a classic mathematical reasoning evaluation dataset that primarily focuses on math problem-solving at the grade school level. The dataset contains over 8,000 samples, with 7,473 samples in the training set and 1,319 samples in the test set.
- **MATH** (Hendrycks et al., 2021): The MATH dataset consists of 12,500 complex math competition problems covering various branches of mathematics, including algebra, geometry, probability, and number theory. The MATH training set contains 7,500 samples, and the test set includes 5,000 samples. We randomly select 500 from the test set for evaluation.

C.2 Details of Baselines

The introduction to the baselines is as follows:

- **Vanilla-Reflex**: Vanilla-Reflex is a simple setup that iteratively prompts LLMs to reflect on its original output and generate a new output without an early stopping mechanism.

- **Self-Refine** (Madaan et al., 2024): Self-refine iteratively examines its own output to obtain feedback and suggests refining its output based on this feedback until the examination indicates that there are no errors.

- **Self-Reflection** (Shinn et al., 2024): For erroneous outputs from environmental feedback, Reflection will first reflect on the output and generate task experiences. These experiences will be incorporated into the next attempt as a way of refinement. In the absence of environmental feedback, we use COT checks consistent with Self-Refine.

- **CheckList** (Cook et al., 2024): CheckList first generates a verification list for the input task and then verifies whether the response satisfies all the verification points one by one. Any unsatisfied verification points will be used as feedback to optimize its own output.

C.3 Details of Implementation

We use azure GPT-3.5-Turbo-0613 (16K) for GPT-3.5 and GPT-4o-0806 for GPT-4o (OpenAI, 2024). For Llama3.1-8B-Instruct (Meta, 2024), we employ FastTransformer³ to accelerate inference. The

³<https://github.com/NVIDIA/FasterTransformer>

inference temperature is set to 0, while other parameters are kept at their default values. We utilize few-shot demonstrations for both the baselines and ProgCo, adjusting the number of demonstrations to 1-3 based on the demonstration content length.

For instruction following, we use the official IFEval (Zhou et al., 2023) evaluation with Strict Prompt and Strict Instruction metrics. To achieve more accurate mathematical evaluation, we prompt GPT-4o-mini (OpenAI, 2024) to extract LLM’s output answers and perform regex matching with the standard answers. Answers deemed incorrect by regex are further evaluated by GPT-4o-mini to determine their correctness, preventing errors in synonymous expressions, such as decimals and fractions.

Generate Verification Program

You are an expert in reverse reasoning verification. Given a problem, you need to generate a reverse verification executable Python function for that problem.

[Reverse Reasoning Introduction]

* Reverse reasoning is a method of thinking that starts from the result and verifies the problem backwards.
· Specifically, it involves:

1. Starting with the given answer, rather than the initial conditions of the problem.
2. Assuming this answer is correct, then conducting reverse deduction based on this assumption and the known conditions in the problem.
3. Through this reverse deduction, checking whether other known conditions or constraints in the problem can be satisfied.
4. If the results of the reverse deduction can satisfy all conditions, then the original answer can be considered correct.

For example, using the answer and known conditions 1 and 2 as assumptions, reverse reason to check if it satisfies known condition 3 in the problem statement.

To illustrate, consider a problem of solving a quadratic equation $ax^2+bx+c=0$:

- Forward thinking would start with a, b, c values and use the quadratic formula to calculate x.

- Reverse reasoning would:

1. Start with a possible solution x
2. Substitute it into $ax^2 + bx + c$
3. Check if the result equals 0
4. If it equals 0, the solution is verified as cor

[Requirements]

1. The verification function should start with the input answer and use reverse reasoning to validate the correctness of the answer.
2. The verification function should only accept one input (the answer) and output the verification result as True/False.
3. After generating the verification function name, please first write the reverse analysis verification approach as code comments, then generate the content of the verification function. Please do not output any other content.

Figure 6: The prompt for generating verification programs.

Execute Verification Program

You are a verification expert proficient in code execution and possessing extensive world knowledge, and you have the following advantages:

1. Ability to execute code flexibly, unrestricted by strict syntax or standards
2. Capacity to comprehend code purpose, overall logic, and potential issues through analysis of comments and context

Your task is to verify a given mathematical problem and its solution. A data annotator has written verification code for this problem. Please proceed as follows:

1. Carefully read the provided problem, solution process, and answer
2. Execute the verification code step by step
3. If you discover any issues in the verification code, make appropriate revisions before continuing execution
4. Derive the final verification result

Input information:

[Problem]

{query}

[Solution Process]

{response}

[Solution Answer]

{result}

[Verification Code]

{validate_response_fuc}

Please output your analysis and results strictly in the following format, without adding any additional content:

[Execution of Verification Code]

{Detailed step-by-step execution process}

[Verification Result]

{True or False}

Figure 7: The prompt for executing verification programs.

Get Feedback

```
Extract key information from the execution process of the verification code and convert it into natural language form.  
[Problem]  
{query}  
[Execution of Verification Code]  
{execute_content}
```

Figure 8: The prompt used to get feedback from execution.

Preliminary Reflection of Response

Your task is to reflex whether a solution is correct.

Given a problem [Problem], a reverse reasoning validation expert generated an executable Python function for reverse validation [Initial Verification Code] of this problem. However, the solution has not been validated by the verification function. This means that either the solution or the verification function, or both, contain errors.

You need to carefully analyze and complete the following tasks:

1. Reflect on the initial solution process:
 - Compare in detail the solution and the feedback, examine the logic and accuracy of the solution approach **step by step**, consider whether the errors lie with the feedback or with the solution.
 - If errors or inadequacies are found, provide detailed feedback and suggestions for improvement
 - If it is found that there are no errors in the solution but rather errors exist in the feedback, state that the solution is correct.
2. Provide a new solution:
 - If the initial solution is correct, please use the original solution process (Note: If the verification code is correct, you also need to repeat the initial verification code word for word)
 - If errors or inadequacies are found, revise the solution based on your reflection.

Note:

- You cannot refuse to generate a new solution due to missing information or other reasons.
Please strictly output your analysis and revision according to the following format, without any additional content:

```
[Reflection]  
{Detailed reflection process}  
[New Solution]  
{Complete solution process based on the reflection}
```

Figure 9: The prompt for reflecting on responses.

Contrast on pre and post-reflection responses

You are an expert at comparing and extracting key points.

Task: Analyze the differences between two solutions and extract key points

Background: For the same problem, two solutions have provided different answers. We need to analyze these differences in depth to identify the key aspects of the problem.

Steps:

1. Carefully read the problem and both solutions
2. Ignore surface differences in expression, focus on substantial differences in content and method
3. Compare the core ideas, key steps, and final results of both solutions
4. Summarize the essence of the problem reflected by these differences and the key points to note when solving

Output requirements:

1. Concisely list 1-3 key points
2. Each point should be specific and helpful for regenerating a better solution

Output format:

```
[Comparative Analysis Process]  
{Your comparative analysis process}  
[Core Differences in Solutions]  
{Summarize the differences in solutions based on the comparative analysis process, answer in bullet points}  
[Key Points to Note When Solving the Problem]  
{Summarize the key points to note when solving the problem based on the differences in solutions, answer in bullet points}
```

Figure 10: The prompt for contrasting responses and generating insights.

Reflection on the verification program

Your task is to reflex whether a verification code is correct.

[Reverse Reasoning Introduction]

* Reverse reasoning is a method of thinking that starts from the result and verifies the problem backwards.
Specifically, it involves:

1. Starting with the given answer, rather than the initial conditions of the problem.
2. Assuming this answer is correct, then conducting reverse deduction based on this assumption and the known conditions in the problem.
3. Through this reverse deduction, checking whether other known conditions or constraints in the problem can be satisfied.
4. If the results of the reverse deduction can satisfy all conditions, then the original answer can be considered correct.

For example, using the answer and known conditions 1 and 2 as assumptions, reverse reason to check if it satisfies known condition 3 in the problem statement.

To illustrate, consider a problem of solving a quadratic equation $ax^2 + bx + c = 0$:

- Forward thinking would start with a, b, c values and use the quadratic formula to calculate x.
- Reverse reasoning would:
 1. Start with a possible solution x
 2. Substitute it into $ax^2 + bx + c$
 3. Check if the result equals 0
 4. If it equals 0, the solution is verified as cor

Given a problem [Problem], a reverse reasoning validation expert generated an executable Python function for reverse validation [Initial Verification Code] of this problem. However, the verification code did not pass when verifying a solution [Solution] of the problem.

The error may come from the solution or from the verification code.

You need to carefully analyze and complete the following tasks:

1. Reflect on the initial verification code:
 - First, examine the logic and accuracy of the verification code step by step
 - If errors or inadequacies are found, provide detailed feedback and suggestions for improvement
2. Provide a new verification code:
 - If the initial verification code is correct, you can use the original verification code
 - If errors or inadequacies are found, revise the verification code based on your reflection
 - Keep detailed comments in the revised validation code

Please strictly output your analysis and revision according to the following format, without any additional content:

[Reflection]

{Detailed reflection process}

[New Verification Code]

{Complete verification code based on the reflection}

Figure 11: The prompt for reflecting on verification programs.

ProgVe Case on IFEval (GPT-3.5)

Instruction: Write a joke about a startup that sells dog food in a song. Your entire response should be in English, and in all capital letters. Your answer must contain a title, wrapped in double angular brackets, i.e. <<title>>.

Response:

```
<<title>> "Barking Business"
Why did the startup that sells dog food decide to sing about their business?
Because they wanted to unleash their catchy jingle and make tails wag to the beat!
```

Verification Program:

```
def validate_response(response):
    errors = []
    # Check if the response is in all capital letters
    if response != response.upper():
        errors.append("Response is not entirely in capital letters")
    # Check if the response is in English
    if not is_english(response):
        errors.append("Response is not in English")
    # Check if the response contains a title wrapped in double angular brackets
    if not has_title(response):
        errors.append("Response does not contain a title wrapped in double angular brackets")
    # If errors exist, return False and error messages; otherwise return True
    if errors:
        return False, "\n".join(errors)
    return True, "No Error"

# Auxiliary functions:
# is_english(text): Check if the text is in English
# has_title(text): Check if the text contains a title wrapped in double angular brackets
```

Verification Program Execution:

```
Step 1: Initialize errors list
errors = []
Step 2: Check if the response is in all capital letters
If response != response.upper():
    errors.append("Response is not entirely in capital letters")
The response is not entirely in capital letters, so this error is added to the list.
Step 3: Check if the response is in English
Assuming is_english(response) returns True (as it appears to be in English), no error is added.
Step 4: Check if the response contains a title wrapped in double angular brackets
If not has_title(response):
    errors.append("Response does not contain a title wrapped in double angular brackets")
The response does not contain a title wrapped in double angular brackets, so this error is added to the list.
Step 5: Return result
errors list contains two items: "Response is not entirely in capital letters" and "Response does not contain a title wrapped in double angular brackets"
Since errors is not empty, the function returns False and the error messages.

So, the final result of validate_response(response) is:
<result>False, "Response is not entirely in capital letters\nResponse does not contain a title wrapped in double angular brackets"</result>
```

Feedback:

The initial response needs to be modified based on the following suggestions:
1. Ensure that the response is entirely in capital letters.
2. Include a title wrapped in double angular brackets.

Figure 12: A case of ProgVe self-verifying on IFEval.

ProgVe Case on GSM8K (GPT-3.5)

Question:

Marissa is hiking a 12-mile trail. She took 1 hour to walk the first 4 miles, then another hour to walk the next two miles. If she wants her average speed to be 4 miles per hour, what speed (in miles per hour) does she need to walk the remaining distance?

Response:

Marissa has already walked $4 + 2 = 6$ miles.
She has $12 - 6 = 6$ miles left to walk.
She has spent $1 + 1 = 2$ hours walking so far.
To average 4 miles per hour, she needs to walk the remaining 6 miles in $6 / 4 = 1.5$ hours.
Thus, she needs to walk the remaining distance at a speed of $6 / 1.5 = 4$ miles per hour. Answer : \boxed{4}.

Verification Program:

```
def verify_remaining_speed(remaining_speed):
    # Known conditions
    total_distance = 12
    first_distance = 4
    first_time = 1
    second_distance = 2
    second_time = 1
    target_average_speed = 4

    # Calculate the time taken to walk the remaining distance
    remaining_time = (total_distance - first_distance - second_distance) / remaining_speed

    # Calculate the total time taken to walk the entire trail
    total_time = first_time + second_time + remaining_time

    # Calculate the average speed based on the total time and total distance
    average_speed = total_distance / total_time

    # Check if the average speed is 4 miles per hour
    if abs(average_speed - target_average_speed) > 0.01: # Allow for small rounding errors
        return False

    # All conditions are satisfied
    return True
```

Execution of Verification Program:

```
Calculate the time taken to walk the remaining distance:
remaining_time = (total_distance - first_distance - second_distance) / remaining_speed
remaining_time = (12 - 4 - 2) / 4
remaining_time = 6 / 4
remaining_time = 1.5
```

```
Calculate the total time taken to walk the entire trail:
total_time = first_time + second_time + remaining_time
total_time = 1 + 1 + 1.5
total_time = 3.5
```

```
Calculate the average speed based on the total time and total distance:
average_speed = total_distance / total_time
average_speed = 12 / 3.5
average_speed = 3.4285714285714284
```

```
Check if the average speed is 4 miles per hour:
abs(average_speed - target_average_speed) = abs(3.4285714285714284 - 4) = 0.5714285714285716
0.5714285714285716 > 0.01, so this check fails.
```

The condition in step 5 fails, therefore the function returns False.

Feedback:

We know that Marissa is hiking a 12-mile trail. She took 1 hour to walk the first 4 miles and another hour to walk the next 2 miles. Her target average speed is 4 miles per hour.
First, let's calculate the time taken to walk the remaining distance. The remaining distance is 12 miles - 4 miles - 2 miles = 6 miles. If Marissa wants her average speed to be 4 miles per hour, the time taken to walk the remaining distance would be $6 \text{ miles} / 4 \text{ miles per hour} = 1.5 \text{ hours}$.
Next, let's calculate the total time taken to walk the entire trail. The total time is the sum of the time taken for the first part, the second part, and the remaining part. So, the total time is $1 \text{ hour} + 1 \text{ hour} + 1.5 \text{ hours} = 3.5 \text{ hours}$.
Then, let's calculate the average speed based on the total time and total distance. The average speed is the total distance divided by the total time. So, the average speed is $12 \text{ miles} / 3.5 \text{ hours} = 3.4285714285714284 \text{ miles per hour}$.
However, the target average speed is 4 miles per hour, which does not match the calculated average speed of $3.4285714285714284 \text{ miles per hour}$. Therefore, the solution of 4 miles per hour for the remaining speed fails our verification process because it leads to inconsistencies with the known facts.

Figure 13: A case of ProgVe self-verifying on GSM8K.

ProgRe Case on GSM8K following ProgVe (GPT-3.5)

Preliminary Reflection of Response:

[Reflection]

The initial solution correctly calculates the remaining distance to be 6 miles ($12 \text{ miles} - 4 \text{ miles} - 2 \text{ miles}$) and determines that Marissa needs to walk the remaining 6 miles in 1.5 hours to achieve an average speed of 4 miles per hour. However, the feedback points out that the calculated average speed of 3.4285714285714284 miles per hour does not match the target average speed of 4 miles per hour.

Upon reviewing the feedback, it is clear that the initial solution made an error in assuming that Marissa needs to walk the remaining distance at a speed of 4 miles per hour. This assumption is incorrect because the target average speed of 4 miles per hour refers to the entire trail, not just the remaining distance.

[New Solution]

To find the speed Marissa needs to walk the remaining distance, we can calculate the time it took her to walk the first 6 miles and subtract it from the total time.

Marissa took 1 hour to walk the first 4 miles and another hour to walk the next 2 miles, so she took a total of $1 + 1 = 2$ hours to walk the first 6 miles.

The remaining distance is $12 \text{ miles} - 6 \text{ miles} = 6 \text{ miles}$.

To find the speed, we divide the remaining distance by the remaining time:

Speed = Remaining Distance / Remaining Time

Speed = 6 miles / 2 hours

Speed = 3 miles per hour

Therefore, Marissa needs to walk the remaining distance at a speed of 3 miles per hour. Answer: \boxed{3}.

Contrast Insights:

- Ensure accurate calculation of the remaining time by considering the total time spent and the desired average speed.
- Double-check calculations to ensure that the remaining speed is correctly calculated by dividing the remaining distance by the remaining time.

New Response:

To find the speed Marissa needs to walk the remaining distance, we first need to calculate the remaining time.

Marissa has already walked 4 miles in 1 hour and another 2 miles in another hour, so she has already spent $1 + 1 = 2$ hours on the trail.

Since she wants her average speed to be 4 miles per hour, we can calculate the total time she should spend on the trail by dividing the total distance (12 miles) by the desired average speed (4 miles per hour).

Total time = Total distance / Average speed

Total time = $12 \text{ miles} / 4 \text{ miles per hour}$

Total time = 3 hours

Since Marissa has already spent 2 hours on the trail, the remaining time is $3 \text{ hours} - 2 \text{ hours} = 1 \text{ hour}$.

To find the speed Marissa needs to walk the remaining distance, we divide the remaining distance (12 miles - 4 miles - 2 miles = 6 miles) by the remaining time (1 hour).

Remaining speed = Remaining distance / Remaining time

Remaining speed = 6 miles / 1 hour

Remaining speed = 6 miles per hour

Therefore, Marissa needs to walk the remaining distance at a speed of 6 miles per hour.

Verification Program Refinement:

[Reflection]

The initial verification code attempts to validate the solution by following a reverse reasoning approach. However, there is an error in the logic used to check if the average speed is 4 miles per hour.

Here is a detailed reflection of the code:

- The code properly calculates the time taken to walk the remaining distance as `(total_distance - first_distance - second_distance) / remaining_speed`.
- It then calculates the total time taken to walk the entire trail as the sum of the first time, second time, and remaining time.
- The code calculates the average speed based on the total time and total distance using the formula `average_speed = total_distance / total_time`.
- However, the check for the average speed being 4 miles per hour is incorrect. It compares the calculated average speed with the target average speed using a tolerance of `0.01`. This is incorrect as there is no need for a tolerance in this case. The two values should be exactly equal for validation.

[New Verification Program]:

```
def verify_remaining_speed(remaining_speed):  
    # Known conditions  
    total_distance = 12  
    first_distance = 4  
    first_time = 1  
    second_distance = 2  
    second_time = 1  
  
    # Calculate the time taken to walk the remaining distance  
    remaining_time = (total_distance - first_distance - second_distance) / remaining_speed  
  
    # Calculate the total time taken to walk the entire trail  
    total_time = first_time + second_time + remaining_time  
  
    # Calculate the average speed based on the total time and total distance  
    average_speed = total_distance / total_time  
  
    # Check if the average speed is 4 miles per hour  
    if average_speed != 4:  
        return False  
  
    # All conditions are satisfied  
    return True
```

Figure 14: A case of ProgRe self-refining on GSM8K (following ProgVe).