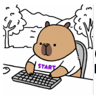

START: Self-taught Reasoner with Tools

Chengpeng Li[1,2*], **Mingfeng Xue**[2*], **Zhenru Zhang**[2], **Jiaxi Yang**[2*], **Beichen Zhang**[2*], **Bowen Yu**[2], **Binyuan Hui**[2], **Junyang Lin**[2], **Xiang Wang**[1†], **Dayiheng Liu**[2†]
[1]University of Science and Technology of China
[2]Qwen Team, Alibaba Inc
{lichengpeng.lcp,liudayiheng.ldyh}@alibaba-inc.com

Abstract

Large Reasoning Models (LRMs) have demonstrated remarkable capabilities in complex reasoning through long chain-of-thought, yet they struggle with precise computations and algorithmic operations. Integrating computational tools with LRMs remains challenging, particularly in activating and enhancing models' tool-use capabilities without compromising their reasoning strengths. We address these challenges through START (Self-taught Reasoner with Tools), introducing two key innovations: (1) Hint-infer, a training-free approach that activates LRMs' latent tool-use capabilities through artificial hints, enabling test-time performance scaling; (2) Hint-RFT, a self-training framework that enables models to learn effective tool utilization through diverse hint patterns and rejection-based data synthesis. Experiments show that START significantly improves state-of-the-art LRMs across challenging benchmarks, including competition-level mathematics (AMC23: 95.0%, AIME24: 75.6%) and graduate-level science questions (GPQA: 64.6%). Our analysis reveals that START not only enhances accuracy but also improves reasoning efficiency through strategic tool utilization, demonstrating broad applicability in complex reasoning scenarios.

1 Introduction

Large Reasoning Models (LRMs) have achieved breakthrough progress in complex reasoning tasks, demonstrating human-like thinking paradigm including iterative reflection, task decomposition, and dynamic strategy shifting in long chain-of-thought (Wei et al., 2022; OpenAI, 2024b; DeepSeek-AI, 2025). However, despite these achievements, current LRMs face fundamental limitations when confronted with tasks requiring precise numerical computations, symbolic manipulation, or complex program execution. (Gou et al., 2024). Computational tools, such as code interpreters (CI), provide LRMs with a paradigm beyond text reasoning, enabling enumeration, precise computation, and algorithm execution through tool interaction (Gou et al., 2024; OpenAI, 2025). By leveraging these capabilities, LRMs can expand their reasoning search space and reduce hallucinations caused by complex calculations.

Integrating computational tools into LRMs remains an open scientific challenge that raises fundamental questions about model capabilities and training strategies. Specifically, we investigate: (1) whether LRMs trained solely on text reasoning data retain the potential for code interpreter utilization; (2) how to synthesize high-quality bootstrapping data for training models with robust tool-use abilities. These questions are crucial for understanding the relationship between language reasoning and computational tool use, as well as developing efficient training paradigms for tool-integrated LRMs. While we only consider Python interpreter as our tool, we conceptualize it not as a single tool but as a versatile **gateway** to a vast ecosystem of specialized libraries—a standard paradigm in the Tool-Integrated Reasoning (TIR) subfield(Gou et al., 2024; Yang et al., 2024) and more details are in Appendix J.1.

To address these questions, we present START (Self-taught Reasoner with Tools). Our initial investigation reveals that state-of-the-art LRMs like QwQ-32B-Preview (Qwen Team, 2024) and DeepSeek-R1-Distill-Qwen-32B (DeepSeek-AI, 2025) struggle to follow instructions for incorporating code interpreter during reasoning. A possible reason is that LRMs typically focus solely on problem-solving during training for complex reasoning tasks, resulting in a loss of generalization in instruction following. To tackle this challenge, we propose a training-free framework named **Hint-infer** that activates models' latent tool-use capabilities. Specifically, by injecting artificial hints such

*Work done during internships at Qwen Team.
†Corresponding author

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 13512–13553
November 4-9, 2025

as "Wait, I can use Python to check if my approach is correct and refine it, if necessary.“‘python" at the end of the thinking process, models spontaneously generate Python code to interact with code interpreters for further verification and computation.

While Hint-infer provides a way to synthesize data with CI calls in long CoT reasoning, it is limited in terms of both the positioning and functionality of tool usage. To enhance the diversity of CI utilization in synthetic data, we propose Hint-RFT (rejection fine-tuning (Yuan et al., 2023; Dong et al., 2023)). Specifically, we summarize six types of hints based on long CoT reasoning patterns (Marjanović et al., 2025) and the role of CI in mathematical reasoning (Gou et al., 2024). Beyond the end of thinking process, we identify "Wait" and "Alternatively" as two high-frequency discourse markers that signal cognitive shifts like (Li et al., 2025a), serving as additional insertion points for hints. To ensure the effectiveness of hint insertion, we employ rejection sampling to retain only the data where hints lead to a significant improvement in reasoning accuracy. Furthermore, by scaling up the training data through another round of RFT, we achieve additional performance gains.

On five challenging mathematical benchmarks, both QwQ-32B-Preview and DeepSeek-R1-Distill-Qwen-32B demonstrate significant performance improvements with Hint-infer, showing test-time scaling capabilities as the number of hint insertions increases. Hint-RFT further enhances model performance through training, achieving average accuracy improvements of 8.6% and 4.4% respectively across these benchmarks. Notably, our methods generalize well to out-of-domain tasks, as evidenced by performance gains on benchmarks like GPQA. When extending to coding tasks, QwQ-32B-Preview shows a 5.9% improvement on LiveCodeBench, demonstrating the broad applicability of our approach.

In summary, our contributions are threefold:

1. We propose a training-free approach called Hint-infer that activates LRMs' latent tool-use capabilities through artificial hints, enabling test-time performance scaling through multiple hint insertions.

2. We introduce Hint-RFT, a self-training framework that enables LRMs to teach themselves effective tool utilization through diverse hint patterns and rejection-based data synthesis.

3. We demonstrate the effectiveness and generalizability of our methods through extensive experiments across mathematical, scientific, and coding tasks, achieving competitive performance with state-of-the-art models.

2 Methodology

Our method consists of three key components: (1) Hint-infer, a training-free approach that enables and scales tool usage through artificial hints during inference; (2) Hint-RFT, a data synthesis framework that combines diverse hint patterns with rejection sampling for model fine-tuning; and (3) RFT, a further refinement stage that scales up training data for additional performance gains. We detail each component in the following subsections, with a primary focus on Hint-infer and Hint-RFT as our main technical contributions.

2.1 Hint-infer

We observe that models like QwQ-32B-Preview and DeepSeek-R1-Distill-Qwen-32B struggle to activate their CI capabilities through direct prompting. Instead, we explore intervening in their reasoning process. Our initial investigation focuses on whether CI can provide performance gains while preserving models' complete thinking process.

Specifically, we intercept the model's output at either the thinking termination point or the last occurrence of "wait" if no explicit termination exists. We then append a hint, such as "Wait, I can use Python to check if my approach is correct and refine it, if necessary.“‘python" (more details in Appendix D). This process can be repeated N times after each reasoning completion, enabling multiple rounds of tool-assisted verification and refinement. The difference of our method and S1 (Muennighoff et al., 2025a) can be seen in Appendix J.

2.2 Hint-RFT

Construct Hint We summarize six types of hints based on long CoT reasoning patterns (Marjanović et al., 2025) and the role of CI in mathematical reasoning (Gou et al., 2024): (1) complex calculations hint for direct computations, (2) self-reflection hint for verification, (3) check logic hint for deduction validation, (4) alternative method hint for exploring different approaches, (5) general hint for basic tool usage, and (6) deeper think hint for thorough analysis. Since mathematical reasoning with tools can be quite complex, these diverse hints enable

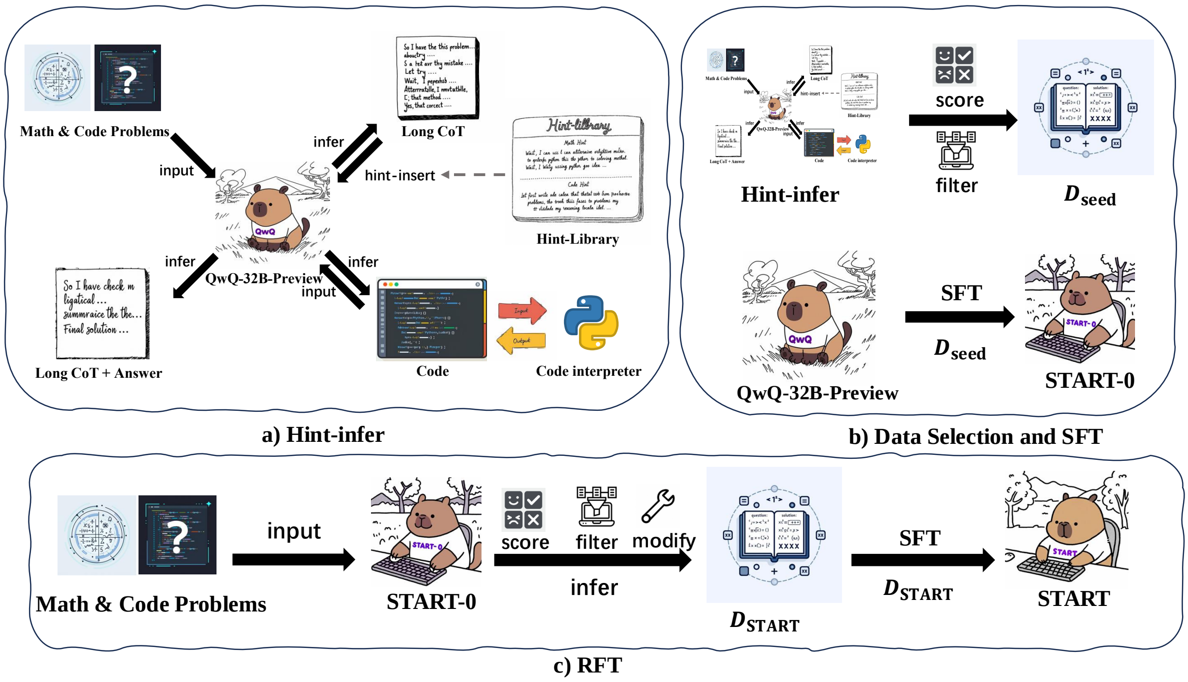

Figure 1: **Training framework for START.** Our framework consists of three components: (**a**) Hint-infer: a training-free approach that injects context-aware hints to activate tool usage during inference, illustrated with QwQ-32B-Preview; (**b**) Hint-RFT: processes Hint-infer outputs to create seed dataset $\mathcal{D}_{\text{seed}}$ for initial fine-tuning, producing START-0; (**c**) RFT: uses START-0 to generate expanded dataset $\mathcal{D}_{\text{START}}$ for final model training. See cases in Appendix G and Appendix I.

the model to adopt different strategies based on the specific situation it encounters. For each hint type, we generate multiple alternative expressions using Qwen-2.5-72B (Yang et al., 2025) to increase linguistic diversity. Some typical hints are list in Figure 2.

Data Curation For mathematical reasoning data curation, we first identify two strategic insertion points for hints: (1) at the end of thinking process, and (2) during the thinking process. To maintain reasoning coherence, we specifically target insertions after conjunctions like *Alternatively* and *Wait*, as these tokens naturally indicate potential shifts in reasoning or consideration of alternative approaches, similar to (Li et al., 2025a).

To ensure the effectiveness of hint insertion, inspired by (Lightman et al., 2024), we adopt an active learning idea. Specifically, we only retain data samples where the inserted hints lead to successful problem-solving for previously incorrect cases, ensuring that each hint insertion makes meaningful contributions to the reasoning process. We verify this robustness through multiple sampling iterations for each hint. Additionally, we filter out responses containing repetitive patterns or incorrect code execution.

RFT To efficiently scale up our training data, we first annotate our training set to obtain a startup dataset D_{seed} containing 10K mathematical reasoning samples. To reduce computational cost, we fine-tune QwQ-32B-Preview on D_{seed} to obtain START-0, which serves as an intermediate model for further data generation. We then employ START-0 to perform rejection fine-tuning (RFT), resulting in a larger dataset D_{START} with 40k mathematical reasoning samples. Finally, we fine-tune the base model on D_{START} to obtain our final model, START.

3 Experiment

3.1 Training data

Our training data consists of math problems sourced from previous AIME problems [1](before 2024), MATH (Hendrycks et al., 2021), and Numina-MATH (LI et al., 2024). We apply the decontamination method as described in (Yang et al., 2024) to the training set in order to minimize potential test data leakage risks. There are a total of 40K math problems, and the specific quantity distribution can be referred to in Appendix A.

[1] https://huggingface.co/datasets/gneubig/aime-1983-2024

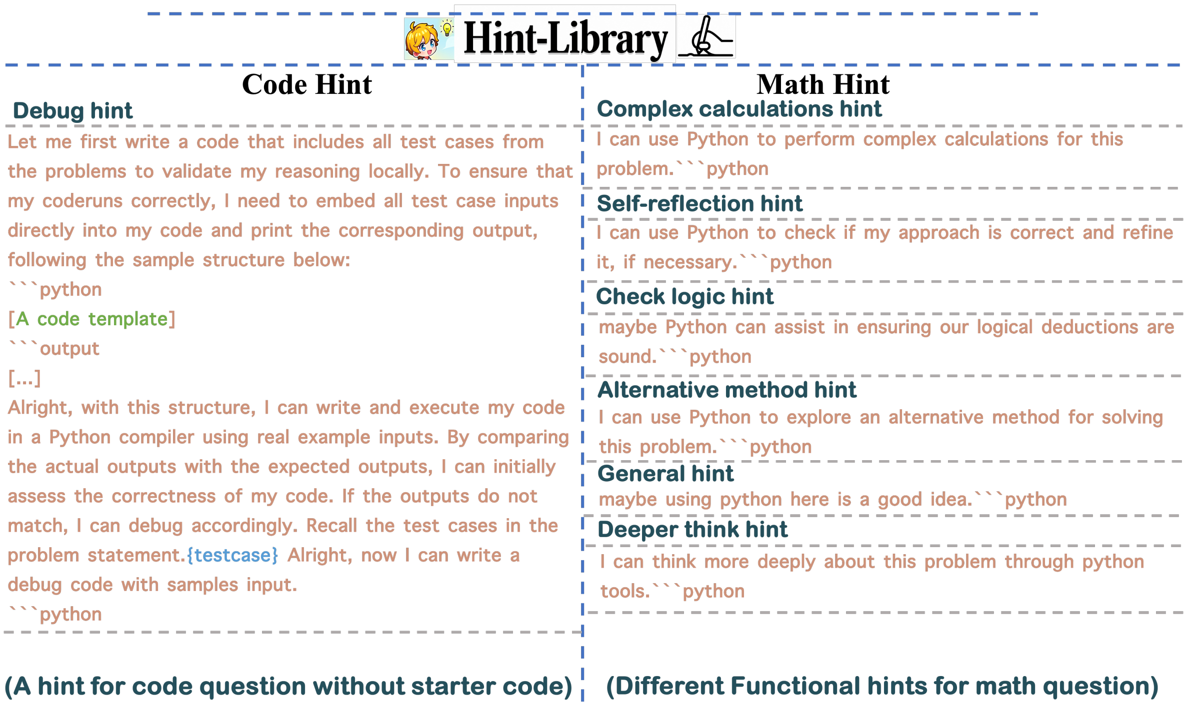

Figure 2: **Some typical hints**. Code generation tasks: Debug hint guides test case review and local code validation. The code template is in D. Math reasoning: Domain-specific hints (e.g., Complex Calculations, Self-Reflection, Logic Check, Alternative Methods) steer code-aided reasoning behaviors.

3.2 Benchmarks

We evaluate our method on both in-domain and out-of-domain benchmarks that require complex reasoning with computational tools.

In-domain Mathematical Benchmarks We select competition-level mathematics datasets including AMC23 [2], AIME24 [3], AIME25 [4], and MATH500 (Lightman et al., 2024). These benchmarks cover various mathematical topics like algebra, calculus, number theory, probability, and geometry.

Out-of-domain Scientific Benchmark We use GPQA (Rein et al., 2023), which contains 448 graduate-level multiple-choice questions in biology, physics, and chemistry. This benchmark is particularly challenging, with domain experts achieving less than 75% accuracy (OpenAI, 2024b).

3.3 Baselines

We compare START with state-of-the-art language models including general-purpose LLMs (GPT-4o (OpenAI, 2024a), Llama3.3-70B (Dubey et al., 2024), DeepSeek-V3-671B (DeepSeek-AI et al., 2024)) and specialized reasoning models (o1 (OpenAI, 2024b) DeepSeek-R1 (DeepSeek-AI, 2025), QwQ-32B-Preview (Qwen Team, 2024)), DeepSeek-R1-Distill-Qwen-32B (DeepSeek-AI, 2025),s1 (Muennighoff et al., 2025a),Qwen-2.5-MATH-72B-TIR (Yang et al., 2024), rStar-Math-7B (Guan et al., 2025).

3.4 Implementation

We select QwQ-32B-Preview and DeepSeek-R1-Distill-Qwen-32B as our base models, producing START-Preview and START-R1 respectively through START frame work in Figure 1. For detailed training hyperparameters, please refer to H. During inference, we use a maximum sequence length of 32,768, limit tool usage to 6 times, and set top_p=0.95 and temperature=0.6 for decoding.

3.5 Main Results

Table 2 presents the comprehensive evaluation results of START against various baseline models. Our key findings are as follows:

In-domain Performance START demonstrates consistent improvements across all mathematical benchmarks. Specifically, START-32B-Preview

[2] https://huggingface.co/datasets/AI-MO/aimo-validation-amc

[3] https://huggingface.co/datasets/AI-MO/aimo-validation-aime

[4] https://huggingface.co/datasets/TIGER-Lab/AIME25

Table 1: Scores on GPQA in various subjects.

Model	Physics	Chemistry	Biology
QwQ-32B-Preview	73.8	41.9	68.4
Search-o1	77.9	**47.3**	**78.9**
START	**80.0**	**47.3**	68.4

achieves significant gains over its base model QwQ-32B-Preview, with improvements of +14.8%, +7.1%, +14.0%, and +3.8% on AIME24, AIME25, AMC23, and MATH500 respectively. Similarly, START-32B-R1 enhances the performance of DeepSeek-R1-32B, achieving state-of-the-art results on AMC23 (95.0%) and competitive performance on other benchmarks.

Out-of-domain Generalization On GPQA, START-32B-Preview shows a significant improvement of 5.5% over its base model, matching the performance of search-o1-32B. As detailed in Table 1, START achieves the highest score in Physics (80.0%), while search-o1-32B(based on START-32B-Preview) excels in Biology. This pattern aligns with the nature of different disciplines: Physics problems often require extensive computational reasoning where Python-based tools excel, while Biology questions rely more on knowledge-based reasoning where search capabilities prove more beneficial. Beyond our main experiments, we further evaluate START's generalization capabilities on diverse reasoning tasks (detailed results in Appendix F).

3.6 Generalization

Framework Generalization To explore the generalization capability of our START framework, we extend our approach to programming tasks using QwQ-32B-Preview as the base model. We design specialized code hints (detailed in Appendix E) that promote self-debugging capabilities, encouraging models to verify solutions against test cases and make necessary adjustments.

The model is trained on programming datasets from Codeforces [5], code contests [6] and LiveCodeBench (before July 2024) (Jain et al., 2024). We evaluate on 112 problems from LiveCodeBench (August-November 2024). As shown in Table 3, START-Preview maintains the strong performance on easy problems (92.3%) while achieving substantial improvements on medium-difficulty problems (+38.6%). The improvement on hard problems is modest (+2.0%), suggesting that while our approach effectively enhances code debugging capabilities, solving complex programs remains challenging.

Tool Generalization Our framework demonstrates strong generalization capabilities in tool learning. Specifically, the model learns a generalizable *principle* of when and how to seek external help, allowing it to autonomously invoke tools it was not exposed to during training. Our training data synthesis already involves a diverse set of libraries such as `sympy`, `numpy`, and `scipy`. The most compelling evidence of tool generalization is the model's emergent behavior on OOD tasks. For instance, when evaluated on the GPQA science benchmark, the model spontaneously invoked the `rdkit` cheminformatics library over 250 times to solve problems related to chemical molecule manipulation, despite this tool being **entirely absent** from the training data. The spontaneous and correct use of a novel tool, as detailed in Table 4, is strong evidence that `START` is not merely memorizing Python library calls but is learning a more abstract, generalizable tool-use capability.

3.7 Analysis of Hint-infer

We analyze the effectiveness of Hint-infer through two perspectives: its direct impact on base models and its test-time scaling capabilities.

Hint-infer vs. Fine-tuned Models Our experiments reveal that while base models like QwQ-32B-Preview possess latent tool-use capabilities, these abilities are difficult to activate through standard prompting. Figure 3 (left) shows that Hint-infer significantly improves QwQ-32B-Preview's performance across all benchmarks, with notable gains on AMC23 (+12.5%), AIME24 (+10.0%), and AIME25 (+13.3%). Notably, both base models achieve these improvements while requiring fewer average tokens, demonstrating that tool usage enhances not only accuracy but also reasoning efficiency. However, these improvements are moderate compared to the fine-tuned START-Preview, suggesting that while Hint-infer can effectively activate tool usage, fine-tuning through Hint-RFT better unlocks the model's full potential.

[5]https://codeforces.com/problemset

[6]https://github.com/google-deepmind/code_contests

Table 2: Performance comparison. Pass@1 results are reported for all benchmarks, with AIME24, AIME25, and AMC23 averaged over 16 samples, and MATH500 over 1 sample. * indicates results from official releases. For unofficial results, we evaluate models locally using the same inference settings as START.

Model	**Tool-Use**	*In-domain*				*Out-of-domain*	
		AIME24	**AIME25**	**AMC23**	**MATH500**	**GPQA**	**Avg**
Baselines							
GPT-4o*	✗	9.3	-	-	60.3	50.6	-
DeepSeek-V3-671B*	✗	39.2	-	-	90.2	59.1	-
Llama3.3-70B	✗	36.7	-	47.5	70.8	43.4	-
Qwen-2.5-MATH-72B-TIR*	✓	40.0	-	70.0	88.1	-	-
rStar-Math-7B	✓	26.7	-	47.5	78.4	-	-
search-o1-32B*	✓	56.7	-	85.0	86.4	63.6	-
DeepSeek-R1-671B*	✗	79.8	70.0	-	97.3	71.5	-
s1-32B*	✗	50.0	33.3	-	93.0	59.6	-
Improvement							
QwQ-32B-Preview	✗	50.0	40.0	80.0	90.6	58.1	63.7
START-32B-Preview	✓	64.8 (+14.8)	47.1 (+7.1)	94.0 (+14.0)	94.4 (+3.8)	63.6 (+5.5)	72.3 (+8.6)
DeepSeek-R1-32B	✗	72.3	54.2	88.9	94.3	61.6	74.3
START-32B-R1	✓	75.6 (+3.3)	63.3 (+9.1)	95.0 (+6.1)	95.1 (+0.8)	64.6 (+3.0)	78.7 (+4.4)

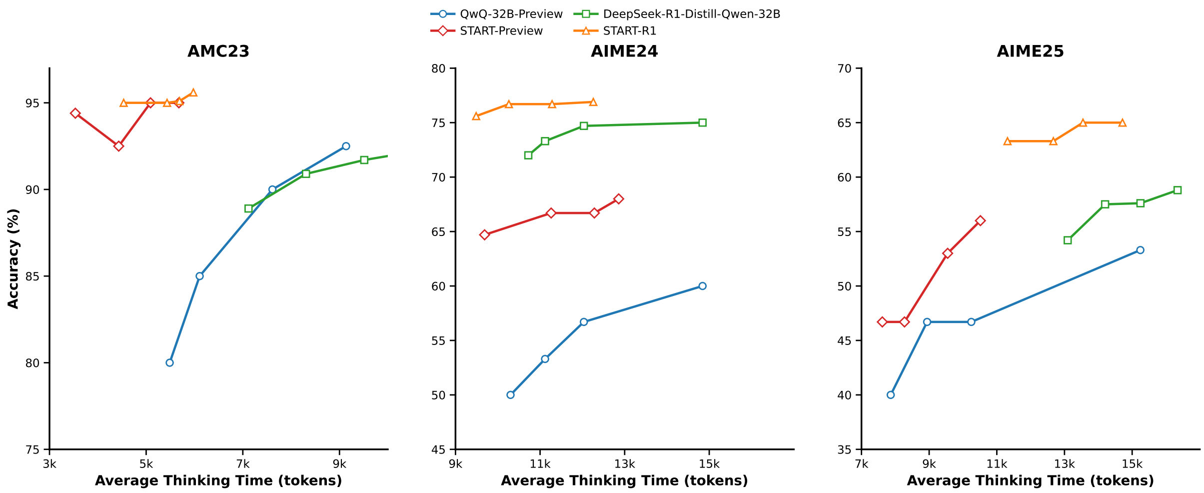

Figure 3: Test time scaling for QwQ-32B-Preview,START-Preview, DeepSeek-R1-Distill-Qwen-32B and START-R1 on challenge math bench marks via Hint-infer.

Table 3: Scores on questions of different difficulty levels on LiveCodeBench.

Model	Easy	Medium	Hard
QwQ-32B-Preview	**92.3**	46.0	10.2
START-Preview	**92.3**	**84.6**	**12.2**

Test-time Scaling Analysis A key advantage of our `Hint-infer` method is its ability to enable test-time performance scaling. As illustrated in Figure 3, the data points for each model-dataset pair represent the model's accuracy after **0, 1, 2, or 3 rounds** of hint intervention are sequentially applied at test time. Each additional hint provides the model with another opportunity to use tools for refinement, leading to consistent performance improvements with a corresponding increase in thinking time. Unlike previous approaches that simply extend reasoning through generic tokens (Muennighoff et al., 2025a), our method actively promotes targeted tool utilization at each refinement step.

Interestingly, we observe that this scaling effect is more pronounced in the base models than in our fine-tuned `START` models. For instance, the base `QwQ-32B-Preview` model shows steady accuracy improvements with increased thinking time across all benchmarks, whereas the fine-tuned `START-Preview` model's performance quickly plateaus. This suggests that the fine-tuned

Table 4: Frequency of Python Libraries Used by the START model on the GPQA Benchmark. The model spontaneously and correctly used `rdkit`, a tool not present in its training data.

Library	Count	Brief Description
`rdkit`	252	Chemical molecule manipulation (Not in training)
`sympy`	224	Symbolic mathematics
`numpy`	192	Numerical arrays and computation
`math`	144	Basic mathematical functions
`scipy`	112	Scientific and engineering computation

models have already internalized the reflective, tool-using behaviors during training, thus reducing the marginal benefit of additional hints during inference.

Validating the Data Synthesis Strategy A key assumption of our `Hint-RFT` framework is that hint-driven interventions are a constructive method for generating high-quality training data. To validate this, we analyzed the impact of applying hints to existing reasoning trajectories during our data generation process. Our findings, summarized in Table 5, show that this intervention is overwhelmingly beneficial.

The analysis reveals that a hint is over **three times more likely to correct an erroneous reasoning trace** (a 33.0% success rate in fixing errors) than it is to introduce an error into a correct one (a 10.4% failure rate). This strong positive net effect confirms that our approach is robust and effectively leverages hint-driven tool use to synthesize high-quality, corrective reasoning trajectories. This result provides a quantitative justification for our rejection-based fine-tuning strategy.

Analysis of Efficiency While the integration of external tools inherently introduces computational overhead, our analysis reveals that the `START` framework achieves a favorable trade-off, leading to a final model that is significantly more efficient in terms of token generation. By strategically replacing verbose, step-by-step textual reasoning with concise and precise tool calls, `START` models produce shorter and more direct solutions. This consistently leads to a higher accuracy ceiling while consuming fewer total tokens than the baseline. For instance, on the AMC23 benchmark, our `START-Preview` model achieves **94.0%** accuracy using an average of only **3,534** tokens. In stark contrast, the baseline `QwQ-32B-Preview` model requires over **5,487** tokens to reach a much lower accuracy of 80.0%. Since token generation is a primary driver of latency, this significant reduction in token count represents a substantial efficiency gain. More results can be seen at the starting points of each model in Figure 3.

3.8 Ablation Study

Ablation on Hints To investigate the effectiveness of different hint types, we conduct ablation studies using QwQ-32B-Preview as the base model. We randomly sample 5000 math training examples for each hint type while ensuring consistent data volume across experiments. Table 6 shows the performance of different hint variants.

Results demonstrate that all hint types contribute positively to model performance, with average improvements ranging from +1.8% to +6.1%. Among single hint types, self-reflection hints achieve the best average performance (71.3%), showing particularly strong improvements on AIME25 (+8.2%) and AMC23 (+9.6%). Deeper thinking hints and alternative method hints also demonstrate strong performance, especially on complex problems like AIME24. The mixed approach, which combines all hint types, achieves the best overall performance (72.4%), suggesting that diverse hint patterns help cover different aspects of mathematical reasoning.

Notably, even when trained with only self-reflection hints, the model exhibits diverse tool usage behaviors in testing. This suggests that hint-infer does not overfit to a single behavior pattern. While this work demonstrates the effectiveness of simple handcrafted hints with random insertion, developing more sophisticated hint design and insertion strategies remains a promising direction for future research.

Ablation on Active Learning We investigate the effectiveness of active learning in Hint-RFT by comparing two data collection strategies (with same data amount): (1) recalling all samples with correct final answers, and (2) only retaining samples where hint insertion leads to successful problem-solving for previously incorrect cases. Using QwQ-32B-Preview as the base model, we conduct experiments on mathematical benchmarks, as shown in Table 7.

Table 5: Analysis of Hint Intervention Outcomes During Rejection Sampling. Hints are significantly more likely to correct an error than to introduce one.

Initial CoT State	Outcome After Hint Intervention	Rate
Correct	Becomes Wrong (Error introduced)	10.4%
Correct	Stays Correct	89.6%
Wrong	**Becomes Correct (Error fixed)**	**33.0%**
Wrong	Stays Wrong	67.0%

Table 6: **Ablation study on different hint types.** Results show the performance of QwQ-32B-Preview with different hint variants on mathematical benchmarks. Mixed approach combines all hint types while maintaining the same total data volume.

Method	MATH500	AIME24	AIME25	AMC23	Avg
QwQ-32B-Preview	90.6	50.0	40.0	80.0	65.2
Complex calculation hint	91.5	52.3	41.2	83.2	67.0
Self-reflection hint	92.8	54.6	48.2	89.6	71.3
Check logic hint	92.4	53.3	43.7	86.9	69.1
Alternative method hint	93.1	55.5	45.6	87.5	70.4
General hint	91.8	51.6	42.6	83.2	67.3
Deeper thinking hint	93.1	58.4	44.4	85.5	70.3
Mixed	93.4	56.8	47.8	91.5	72.4

Results show that while both strategies improve over the baseline, the active learning approach leads to better performance across all benchmarks. START-0-Preview with active learning achieves an average improvement of +3.8% over the baseline, compared to +2.4% without active learning, demonstrating the importance of selective data curation in tool-augmented reasoning.

Ablation on Data Format To isolate the impact of tool usage from data quantity, we conduct an ablation study comparing pure reasoning and tool-augmented reasoning approaches. Using the same training data , we fine-tune QwQ-32B-Preview through standard text-based RFT (QwQ-32B-Preview-RFT) and our tool-augmented approach (START-Preview). As shown in Table 8, QwQ-32B-Preview-RFT achieves similar performance to the base model (65.2% vs. 65.2% average accuracy), while START-Preview shows substantial improvements (+9.9% average accuracy). These results suggest that the performance gains of START primarily stem from its tool-use capabilities rather than the expanded training data, highlighting the importance of integrating computational tools in complex reasoning tasks.

4 Related Work

Large Reasoning Models Large Language Models have demonstrated remarkable capabilities in complex reasoning through Chain-of-Thought (CoT) prompting (Wei et al., 2022). This has been further enhanced by Long Chain-of-Thought (OpenAI, 2024b; DeepSeek-AI, 2025; Team et al., 2025; Qwen Team, 2024), where models exhibit advanced cognitive behaviors such as reflection, verification, and multi-path exploration. Recent works have successfully scaled these capabilities through various approaches: QwQ-32B-Preview (Qwen Team, 2024), DeepSeek-R1 (DeepSeek-AI, 2025), and InternThinker (Cai et al., 2024) leverage fine-tuning and reinforcement learning to enhance reasoning abilities, while Open-R1 (Huggingface, 2025), S1 (Muennighoff et al., 2025b), and LIMO (Ye et al., 2025) demonstrate the effectiveness of distillation for smaller models. However, these text-based reasoning approaches often struggle with precise numerical computations and complex algorithmic operations, leading to potential hallucinations and accuracy issues in mathematical and scientific reasoning tasks. This limitation highlights the need for integrating computational tools with language models while preserving their

Table 7: **Ablation study on active learning strategy.** Results compare QwQ-32B-Preview with START-0-Preview trained with and without active learning (ac). Active learning retains only samples where hint insertion improves model performance.

Method	MATH500	AIME24	AIME25	AMC23	Avg
QwQ-32B-Preview	90.6	50.0	40.0	80.0	65.2
START-0-Preview w/o ac	91.3	52.3	41.2	85.5	67.6
START-0-Preview	**92.3**	**53.1**	**42.7**	**87.8**	**69.0**

Table 8: **Ablation study on data format.** Comparison between pure reasoning (RFT) and tool-augmented reasoning (START) using the same training data, demonstrating the importance of tool usage rather than data quantity.

Method	MATH500	AIME24	AIME25	AMC23	Avg.
QwQ-32B-Preview	90.6	50.0	40.0	80.0	65.2
QwQ-32B-Preview-RFT	91.8	53.3	33.3	82.5	65.2
START-Preview	**94.4**	**64.8**	**47.1**	**94.0**	**75.1**

sophisticated reasoning capabilities.

Tool-integrated Reasoning To address computational inaccuracies in LLMs, recent works have explored integrating external tools into the reasoning process. Studies have demonstrated the benefits of code-based pre-training (Shao et al., 2024) and post-training (Chen et al., 2023; Gou et al., 2024; Liao et al., 2024; Li et al., 2024). Formal verification tools like Lean have also shown promise in mathematical proof verification (Xin et al., 2024; Wu et al., 2024).

Additionally, while rStar (Guan et al., 2025) focuses on concatenating short CoTs into long CoTs and integrating tool usage through process rewards model and MCTS, START builds upon o1-style long CoT to internalize complex human reasoning processes within the CoT framework itself. These approaches are relatively orthogonal and could potentially be complementary in future research.

Concurrent Work Concurrent with our work, several studies have explored related directions in tool-augmented reasoning. Works like ToRL (Li et al., 2025b), AutoCode4Math (Wang et al., 2025), and ZTRL (Mai et al., 2025) investigate zero-shot reinforcement learning approaches for tool utilization. While these studies demonstrate promising results, they differ fundamentally from START in that they train from base models rather than enhancing existing LRMs, potentially missing out on the sophisticated reasoning capabilities already present in state-of-the-art reasoning models.

Recent works like Retool (Feng et al., 2025) and STILL3 (Chen et al., 2025) focus on addressing cold-start challenges in generating code-integrated reasoning data. While these approaches make valuable contributions to data generation strategies, START's hint-based approach offers a unique perspective by directly activating and improving latent tool-use capabilities in existing models.

5 Conclusion

We present START, a framework that effectively activates and enhances the tool-use capabilities of LLMs through two key components: Hint-infer and Hint-RFT. Our approach demonstrates that large reasoning models possess latent tool-use abilities that can be activated without training through strategically placed hints, and these capabilities can be further enhanced via targeted fine-tuning with rejection sampling. Through extensive experiments, START achieves substantial improvements across various benchmarks, surpassing state-of-the-art models on several mathematical and scientific reasoning tasks.

Our analysis reveals that the framework's effectiveness stems primarily from enhanced tool utilization, which in turn leads to more efficient reasoning with reduced token consumption. Furthermore, we show that this hint-based activation enables test-time performance scaling, allowing models to achieve better results through multiple rounds of tool interaction without additional training. These insights validate our approach and highlight the potential of strategic interventions to unlock and refine the latent abilities of LLMs.

6 Limitations

While START demonstrates strong performance across various reasoning tasks, we acknowledge several limitations of our current work. First, the computational overhead of tool usage, while offset by improved reasoning efficiency, may need to be considered in resource-constrained scenarios. Our current implementation requires additional computation for code execution, though this is typically balanced by the reduced number of reasoning steps and improved accuracy. Additionally, while our hint insertion methodology proves effective, there might be room for more systematic approaches to optimize insertion points and frequencies. For instance, developing adaptive strategies for determining optimal hint insertion timing based on problem complexity and reasoning progress could potentially further improve efficiency. Future work could explore these directions while maintaining the core benefits of our framework, potentially leading to even more efficient tool-augmented reasoning systems.

7 Acknowledgement

We thank all anonymous reviewers for their helpful comments and suggestions. This research is supported by the National Science and Technology Major Project (2023ZD0121102).

References

Zheng Cai, Maosong Cao, Haojiong Chen, Kai Chen, Keyu Chen, Xin Chen, Xun Chen, Zehui Chen, Zhi Chen, Pei Chu, Xiaoyi Dong, Haodong Duan, Qi Fan, Zhaoye Fei, Yang Gao, Jiaye Ge, Chenya Gu, Yuzhe Gu, Tao Gui, Aijia Guo, Qipeng Guo, Conghui He, Yingfan Hu, Ting Huang, Tao Jiang, Penglong Jiao, Zhenjiang Jin, Zhikai Lei, Jiaxing Li, Jingwen Li, Linyang Li, Shuaibin Li, Wei Li, Yining Li, Hongwei Liu, Jiangning Liu, Jiawei Hong, Kaiwen Liu, Kuikun Liu, Xiaoran Liu, Chengqi Lv, Haijun Lv, Kai Lv, Li Ma, Runyuan Ma, Zerun Ma, Wenchang Ning, Linke Ouyang, Jiantao Qiu, Yuan Qu, Fukai Shang, Yunfan Shao, Demin Song, Zifan Song, Zhihao Sui, Peng Sun, Yu Sun, Huanze Tang, Bin Wang, Guoteng Wang, Jiaqi Wang, Jiayu Wang, Rui Wang, Yudong Wang, Ziyi Wang, Xingjian Wei, Qizhen Weng, Fan Wu, Yingtong Xiong, Chao Xu, Ruiliang Xu, Hang Yan, Yirong Yan, Xiaogui Yang, Haochen Ye, Huaiyuan Ying, Jia Yu, Jing Yu, Yuhang Zang, Chuyu Zhang, Li Zhang, Pan Zhang, Peng Zhang, Ruijie Zhang, Shuo Zhang, Songyang Zhang, Wenjian Zhang, Wenwei Zhang, Xingcheng Zhang, Xinyue Zhang, Hui Zhao, Qian Zhao, Xiaomeng Zhao, Fengzhe Zhou, Zaida Zhou, Jingming Zhuo, Yicheng Zou, Xipeng Qiu, Yu Qiao, and Dahua Lin. 2024. Internlm2 technical report. *Preprint*, arXiv:2403.17297.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Trans. Mach. Learn. Res.*, 2023.

Zhipeng Chen, Yingqian Min, Beichen Zhang, Jie Chen, Jinhao Jiang, Daixuan Cheng, Wayne Xin Zhao, Zheng Liu, Xu Miao, Yang Lu, Lei Fang, Zhongyuan Wang, and Ji-Rong Wen. 2025. An empirical study on eliciting and improving r1-like reasoning models. *Preprint*, arXiv:2503.04548.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2024. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Hanze Dong, Wei Xiong, Deepanshu Goyal, Yihan Zhang, Winnie Chow, Rui Pan, Shizhe Diao, Jipeng Zhang, Kashun Shum, and Tong Zhang. 2023. RAFT: reward ranked finetuning for generative foundation model alignment. *Trans. Mach. Learn. Res.*

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. 2024. The Llama 3 herd of models. *CoRR*, abs/2407.21783.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. 2025. Retool: Reinforcement learning for strategic tool use in llms. *Preprint*, arXiv:2504.11536.

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, ICLR 2024, Vienna, Austria, May 7-11, 2024*.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. 2025. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *Preprint*, arXiv:2501.04519.

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. In *NeurIPS Datasets and Benchmarks*.

Huggingface. 2025. Open r1.

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

Chengpeng Li, Guanting Dong, Mingfeng Xue, Ru Peng, Xiang Wang, and Dayiheng Liu. 2024. Dotamath: Decomposition of thought with code assistance and self-correction for mathematical reasoning. *CoRR*, abs/2407.04078.

Jia LI, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. 2024. Numinamath. `[https://github.com/project-numina/aimo-progress-prize](https://github.com/project-numina/aimo-progress-prize/blob/main/report/numina_dataset.pdf)`.

Xiaoxi Li, Guanting Dong, Jiajie Jin, Yuyao Zhang, Yujia Zhou, Yutao Zhu, Peitian Zhang, and Zhicheng Dou. 2025a. Search-o1: Agentic search-enhanced large reasoning models. *Preprint*, arXiv:2501.05366.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025b. Torl: Scaling tool-integrated rl. *Preprint*, arXiv:2503.23383.

Minpeng Liao, Chengxi Li, Wei Luo, Jing Wu, and Kai Fan. 2024. MARIO: math reasoning with code interpreter output - A reproducible pipeline. In *ACL (Findings)*, pages 905–924. Association for Computational Linguistics.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*.

Xinji Mai, Haotian Xu, Xing W, Weinong Wang, Yingying Zhang, and Wenqiang Zhang. 2025. Agent rl scaling law: Agent rl with spontaneous code execution for mathematical problem solving. *Preprint*, arXiv:2505.07773.

Sara Vera Marjanović, Arkil Patel, Vaibhav Adlakha, Milad Aghajohari, Parishad BehnamGhader, Mehar Bhatia, Aditi Khandelwal, Austin Kraft, Benno Krojer, Xing Han Lù, Nicholas Meade, Dongchan Shin, Amirhossein Kazemnejad, Gaurav Kamath, Marius Mosbach, Karolina Stańczak, and Siva Reddy. 2025. Deepseek-r1 thoughtology: Let's think about llm reasoning. *Preprint*, arXiv:2504.07128.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025a. s1: Simple test-time scaling.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025b. s1: Simple test-time scaling. *Preprint*, arXiv:2501.19393.

OpenAI. 2024a. Hello GPT-4o.

OpenAI. 2024b. Learning to reason with LLMs.

OpenAI. 2025. Openai o3-mini.

Qwen Team. 2024. QwQ: Reflect deeply on the boundaries of the unknown.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. 2023. GPQA: A graduate-level Google-proof Q&A benchmark. *CoRR*, abs/2311.12022.

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

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, Congcong Wang, Dehao Zhang, Enming Yuan, Enzhe Lu, Fengxiang Tang, Flood Sung, Guangda Wei, Guokun Lai, Haiqing Guo, Han Zhu, Hao Ding, Hao Hu, Hao Yang, Hao Zhang, Haotian Yao, Haotian Zhao, Haoyu Lu, Haoze Li, Haozhen Yu, Hongcheng Gao, Huabin Zheng, Huan Yuan, Jia Chen, Jianhang Guo, Jianlin Su, Jianzhou Wang, Jie Zhao, Jin Zhang, Jingyuan Liu, Junjie Yan, Junyan Wu, Lidong Shi, Ling Ye, Longhui Yu, Mengnan Dong, Neo Zhang, Ningchen Ma, Qiwei Pan, Qucheng Gong, Shaowei Liu, Shengling Ma, Shupeng Wei, Sihan Cao, Siying Huang, Tao Jiang, Weihao Gao, Weimin Xiong, Weiran He, Weixiao Huang, Wenhao Wu, Wenyang He, Xianghui Wei, Xianqing Jia, Xingzhe Wu, Xinran Xu, Xinxing Zu, Xinyu Zhou, Xuehai Pan, Y. Charles, Yang Li, Yangyang Hu, Yangyang Liu, Yanru Chen, Yejie Wang, Yibo Liu, Yidao Qin, Yifeng Liu, Ying Yang, Yiping Bao, Yulun Du, Yuxin Wu, Yuzhi Wang, Zaida Zhou, Zhaoji Wang, Zhaowei Li, Zhen Zhu, Zheng Zhang, Zhexu Wang, Zhilin Yang, Zhiqi Huang, Zihao Huang, Ziyao Xu, and Zonghan Yang. 2025. Kimi k1.5: Scaling reinforcement learning with llms. *Preprint*, arXiv:2501.12599.

Haozhe Wang, Long Li, Chao Qu, Fengming Zhu, Weidi Xu, Wei Chu, and Fangzhen Lin. 2025. Learning autonomous code integration for math language models. *Preprint*, arXiv:2502.00691.

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.

Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Jiayu Wang, Dahua Lin, and Kai Chen. 2024. Internlm2.5-stepprover: Advancing automated theorem proving via expert iteration on large-scale LEAN problems. *CoRR*, abs/2410.15700.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. 2024. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *CoRR*, abs/2405.14333.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, et al. 2024. Qwen2.5-Math technical report: Toward mathematical expert model via self-improvement. *CoRR*, abs/2409.12122.

Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. 2025. Limo: Less is more for reasoning. *Preprint*, arXiv:2502.03387.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. 2023. Scaling relationship on learning mathematical reasoning with large language models. *Preprint*, arXiv:2308.01825.

A Training set of START

Table 9 presents the composition of our training dataset. For mathematical reasoning, we collect problems from three main sources: AIME problems before 2024 (890 samples), MATH dataset (Hendrycks et al., 2021) (7,500 samples), and Numina-MATH (LI et al., 2024) (28,505 samples). This combination provides a diverse range of mathematical problems spanning various difficulty levels and topic areas.

For programming tasks, we incorporate data from Codeforces (7,505 problems), code contests (2,011 problems), and LiveCodeBench before July 2024 (Jain et al., 2024) (558 problems). These sources offer a comprehensive coverage of coding challenges with different complexity levels. In total, our training set consists of 49,969 problems, with approximately 74% focusing on mathematical reasoning and 26% on programming tasks.

B More results about Hint-infer

Building upon the observed trends, the detailed results in Table B further underscore the efficacy of the QwQ-Hint-infer and START-Hint-infer methods across diverse challenging reasoning tasks. Specifically, for datasets such as aime24, aime25, gpqa, amc23, MATH500, and LiveCodeBench, QwQ-32B-Preview consistently demonstrates performance enhancements with each subsequent round of hint insertion. For instance, aime24 improves from 50.0% in Round 0 to 60.0% in Round 3, and MATH500 shows a marginal yet steady increase from 90.6% to 92.4% over the same rounds. This consistent upward trend highlights the method's ability to incrementally refine the model's reasoning capabilities through iterative hint integration.

In contrast, the START-Hint-infer approach exhibits a more varied performance across different datasets. While there are improvements in some areas, such as AIME25, where the Pass@1 metric reaches 60.0% by Round 3 and LiveCodeBench sees an increase from 47.3% to 50.0%, other datasets like GPQA and LiveCodeBench show relatively modest gains and even no gains. This disparity suggests that the effectiveness of Hint-infer may be contingent on the inherent characteristics of the dataset and the nature of the reasoning tasks involved.

C Prompting Methods for Data annotation

In our initial exploration for data annotation, we investigated several conventional prompting methods to trigger Large Reasoning Models (LRMs) to generate long Chain-of-Thought (CoT) solutions integrated with Python tool calls. We evaluated three primary strategies:

1. **Direct Prompting**, which uses a simple instruction like, "Please integrate programs to solve the problem above".

2. **Well-designed Prompting**, which provides detailed instructions on tool use, adapted from search-o1 (Li et al., 2025a).

3. **In-context Prompting**, which provides few-shot demonstrations to guide the model's output format.

However, our experiments revealed that these preemptive prompting methods were overwhelmingly ineffective at reliably activating tool use in long-reasoning contexts. As shown in Table 11, both direct and sophisticated prompts almost completely failed to elicit tool invocation, with success rates falling below 4%. In stark contrast, our post-hoc `Hint-infer` method demonstrated a **100% success rate** in triggering the models to produce Python code. This provides strong quantitative evidence that the specific strategy of intervention is critical. A post-hoc, reflective hint is a significantly more effective trigger for activating latent tool-use capabilities than standard, instruction-based prompts.

D Hint-infer for test time scaling

The three rounds hints of GPQA and MATH for Hint-infer are: *Wait, I can use Python to check if my approach is correct and refine it, if necessary.“‘python*, *Wait, I need to utilize Python code again to meticulously check to make sure I understand the question correctly as well as reasoning correctly.“‘python* and *Wait, I can think more deeply about this problem through python tools.“‘python*.

E Code Task Hints

For code problem with starter code, the code template is

Table 9: Sources of Dataset D

Source	Quantity
AIME problems (before 2024)	890
MATH (Hendrycks et al., 2021)	7500
Numina-MATH (LI et al., 2024)	28505
Code Data	
Codeforces	7505
Code contests	2011
LiveCodeBench (before July 2024) (Jain et al., 2024)	558
Total	49969

Table 10: Comparison of QWQ-Hint-infer and START-Hint-infer on challenging reasoning tasks, including PhD-level science QA, math, and code benchmarks. We report Pass@1 metric for all tasks.

Dataset	**QwQ-32B-Preview**				**START**			
	Round 0	**Round 1**	**Round 2**	**Round 3**	**Round 0**	**Round 1**	**Round 2**	**Round 3**
aime24	50.0%	53.3%	56.7%	60.0%	66.7%	66.7%	66.7%	70.0%
aime25	40.0%	47.1%	47.1%	53.3%	47.1%	47.1%	60.0%	60.0%
gpqa	58.5%	58.6%	59.6%	59.6%	63.6%	61.6%	60.6%	61.6%
amc23	80.0%	85.0%	90.0%	92.5%	95.0%	92.5%	95.0%	95.0%
MATH500	90.6%	92.0%	92.0%	92.4%	94.4%	95.0%	95.6%	95.2%
LiveCodeBench	41.4%	42.0%	42.0%	42.0%	47.3%	48.2%	50.0%	50.0%

Table 11: Quantitative Comparison of Different Prompting Methods in Triggering Tool Use. Our post-hoc `Hint-infer` method shows 100% reliability.

Prompting Method	**Tool Use % (QWQ-32B-Preview)**	**Tool Use % (R1-distill-qwen-32B)**
Direct Prompt	0%	0%
Well-designed Prompt	3.3%	2.4%
START (Hint-infer)	**100%**	**100%**

```
{startcoder}
# Test the example inputs
solution = Solution()
# Example input1
test_input1 = ...
# Example input2
test_input2 = ...
# Print output
print(solution.function_name(test_input1))
print(solution.function_name(test_input2))
# Check the output
'''
'''output
[...]
```

For code problem without starter code, the code template is

```
def function_name(parameters):
#Implementation\n
# Test the example inputs
solution = Solution()
# Example input1
test_input1 = ...
# Example input2
test_input2 = ...
# Print output
...(same as above)
```

F More Experiments on Generalization

START shows consistent improvements across logical reasoning (LogicBench: +1.5%), theoretical computer science (TheoremQA: +2.4%), and tabular math word problems (TabMWP: +1.1%). These results suggest that the benefits of our framework extend beyond the primary evaluation domains, demonstrating the general applicability of tool-augmented reasoning.

Table 12: **START's generalization to additional reasoning tasks.**

Model	LogicBench	TheoremQA*	TabMWP
QwQ-32B-Preview	50.6	49.3	95.1
START	**52.1**	**51.7**	**96.2**
Improvement	(+1.5)	(+2.4)	(+1.1)

G Case Study For START

G.1 A case form AMC23

A case form AMC23

Problem:
In the state of Coinland, coins have values 6, 10, and 15 cents. Suppose x is the value in cents of the most expensive item in Coinland that cannot be purchased using these coins with exact change. What is the sum of the digits of x?
Solution:
So I've got this problem from the state of Coinland, where they have coins worth 6, 10, and 15 cents. The question is asking for the value of the most expensive item that you can't buy using these coins with exact change. Then, I need to find the sum of the digits of that value.
First off, I need to understand what this is asking. It seems like a problem related to the concept of the Frobenius coin problem, which is about finding the largest amount that cannot be obtained using given coin denominations. In the classic case, with two coprime denominations, there's a formula to find this number, but here we have three denominations: 6, 10, and 15 cents.
I know that for two coprime numbers, a and b, the largest number that can't be expressed as a non-negative integer combination of a and b is $ab - a - b$. But here, we have three numbers, and they aren't all pairwise coprime. For instance, 6 and 10 share a common factor of 2, and 6 and 15 share a common factor of 3. So, the standard Frobenius number formula doesn't directly apply.
Maybe I should start by looking at the greatest common divisor (GCD) of these numbers. Let's find the GCD of 6, 10, and 15.
The GCD of 6 and 10 is 2.
The GCD of 2 and 15 is 1.
So, the overall GCD is 1. That means that eventually, any sufficiently large number can be expressed as a combination of these coins, since their GCD is 1.
But, there might be some gaps before we reach a certain point where all numbers can be expressed.
Perhaps I should look for the Frobenius number for these three denominations. However, I recall that for more than two denominations, there isn't a general formula for the Frobenius number. So, I might need to approach this problem differently.
One approach could be to consider the possible combinations of these coins and see which numbers can be expressed. Maybe I can start by listing out the possible sums.
Let's list out some combinations:
- 0 coins: 0 cents
- 1 coin: 6, 10, or 15 cents
- 2 coins: $6+6=12$, $6+10=16$, $6+15=21$, $10+10=20$, $10+15=25$, $15+15=30$ cents
- 3 coins: $6+6+6=18$, $6+6+10=22$, $6+6+15=27$, $6+10+10=26$, $6+10+15=31$, $6+15+15=36$, $10+10+10=30$, $10+10+15=35$, $10+15+15=40$, $15+15+15=45$ cents
- And so on...

This could take a while, and I might miss some numbers. Maybe there's a better way.
Another thought: since the GCD is 1, I know that every number greater than or equal to a certain number can be expressed as a combination of these coins. I need to find the largest number that cannot be expressed this way.
Perhaps I can look at the problem in terms of modulo arithmetic. For example, since 6 is divisible by 3, 10 is not, and 15 is divisible by 3, I can consider how these combine in terms of modulo 3.
Wait, but that might not be directly helpful. Let me think differently.
I've heard of the concept of the "Frobenius number" for two coins, but for three or more, it's more complicated. Maybe I can reduce this problem to a combination of smaller problems.

Let me consider pairs of these denominations and find their Frobenius numbers, then see how that helps.
First, consider only 6 and 10 cent coins. Since $\text{GCD}(6, 10) = 2$, which is greater than 1, the Frobenius number for these two is not defined because you can't make all even numbers greater than a certain point with just 6 and 10. Wait, no. Actually, since their GCD is 2, you can make any even number greater than or equal to a certain value.
Wait, the Frobenius number formula $ab - a - b$ applies only when a and b are coprime. Here, 6 and 10 are not coprime, so that formula doesn't apply.
Similarly, for 6 and 15, $\text{GCD}(6, 15) = 3$, which is greater than 1.
And for 10 and 15, $\text{GCD}(10, 15) = 5$.
So, none of the pairs are coprime.
This complicates things.
Alternatively, maybe I can look at the problem in terms of linear combinations.
I need to find the largest x such that there are no non-negative integers a, b, c satisfying $6a + 10b + 15c = x$.
This seems tricky. Maybe I can look for a pattern in the possible sums.
Let me try to list out the possible sums in order:
$0, 6, 10, 12, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 30$, etc.
Wait, I can use Python to generate specific examples that fit the problem's conditions.

Python Code

```
from itertools import product
def can_be_paid(x, coins):
    for a in range(x//coins[0] + 1):
        for b in range(x//coins[1] + 1):
            for c in range(x//coins[2] + 1):
                if a*coins[0] + b*coins[1] + c*coins[2] == x:
                    return True
    return False
coins = [6, 10, 15]
max_x = 100
# arbitrary upper limit to search for the largest x that cannot be paid
possible_x = []
for x in range(1, max_x):
    if not can_be_paid(x, coins):
        possible_x.append(x)

print((max(possible_x), sum(int(digit) for digit in str(max(possible_x)))))
```

Output

```
(29, 11)
```

From the code, I found that the largest value that cannot be purchased using coins of 6, 10, and 15 cents is 29 cents. The sum of the digits of 29 is $2 + 9 = 11$.
So, the answer is 11.
Final Answer

$$\boxed{11}$$

G.2 A case form AIME24

A case form AIME24

Problem:
Let p be the least prime number for which there exists a positive integer n such that $n^4 + 1$ is divisible by p^2. Find the least positive integer m such that $m^4 + 1$ is divisible by p^2.
Solution:

I'm trying to solve this problem. It says: Let p be the least prime number for which there exists a positive integer n such that $n^4 + 1$ is divisible by p^2. Find the least positive integer m such that $m^4 + 1$ is divisible by p^2.
First, I need to find the smallest prime p such that there's some positive integer n making $n^4 + 1$ divisible by p^2. Then, using that p, I have to find the smallest positive integer m where $m^4 + 1$ is divisible by p^2.
So, to start, I need to understand when $n^4 + 1$ is divisible by p^2. That means p^2 divides $n^4 + 1$, or in other words, $n^4 \equiv -1 \pmod{p^2}$.
I know that for primes, certain properties hold due to Fermat's Little Theorem and Euler's Theorem, but I'm not sure how to apply them here directly because we're dealing with p^2, not just p.
Maybe I should look at the equation $n^4 \equiv -1 \pmod{p^2}$. This implies that $n^8 \equiv 1 \pmod{p^2}$, because if $n^4 \equiv -1$, then squaring both sides gives $n^8 \equiv 1 \pmod{p^2}$.
So, the order of n modulo p^2 divides 8, meaning the order is 1, 2, 4, or 8. But since $n^4 \equiv -1 \pmod{p^2}$, the order can't be 1, 2, or 4, because in those cases, n^4 would be 1, not -1. Therefore, the order must be 8.
This means that 8 divides the order of the multiplicative group modulo p^2, which is $\phi(p^2) = p(p-1)$, where ϕ is Euler's totient function.
So, 8 divides $p(p-1)$. Since p is prime, there are a few cases to consider:
1. $p = 2$
2. p is an odd prime, and 8 divides $p - 1$
Let's consider $p = 2$ first.
If $p = 2$, then $p^2 = 4$. We need to check if there exists an n such that $n^4 + 1$ is divisible by 4.
Let's compute $n^4 + 1$ modulo 4:
- If n is even, n^4 is divisible by 16, so $n^4 \equiv 0 \pmod 4$, thus $n^4 + 1 \equiv 1 \pmod 4$, which is not divisible by 4.
- If n is odd, $n^4 \equiv 1 \pmod 4$, so $n^4 + 1 \equiv 2 \pmod 4$, which is also not divisible by 4.
Therefore, there is no positive integer n such that $n^4 + 1$ is divisible by 4. So, $p = 2$ is not a valid choice.
Next, consider odd primes where 8 divides $p - 1$. That is, $p \equiv 1 \pmod 8$.
The smallest such primes are 17, 41, 73, etc.
Alternatively, there could be other primes where p divides $n^4 + 1$, but for p^2 to divide $n^4 + 1$, it's more restrictive.
Wait a minute, maybe I should consider quadratic residues or higher power residues.
Alternatively, perhaps I should look for primes where -1 is a fourth power residue modulo p^2.
I recall that for primes, -1 is a quadratic residue if $p \equiv 1 \pmod 4$, but here we're dealing with fourth powers.
Maybe I should consider the Legendre symbol or higher power residues.
Alternatively, perhaps I can use Hensel's Lemma to lift solutions from modulo p to modulo p^2.
First, I need to find primes p such that there exists an n with $n^4 \equiv -1 \pmod p$, and then check if that solution can be lifted to modulo p^2.
So, start by finding primes p for which $n^4 \equiv -1 \pmod p$ has a solution.
This is equivalent to saying that -1 is a fourth power residue modulo p.

The multiplicative group modulo p is cyclic of order $p-1$, so $n^4 \equiv -1 \pmod p$ has a solution if and only if there exists an integer k such that $4k \equiv \frac{(p-1)}{2} \pmod{p-1}$, since -1 is a square modulo p if $p \equiv 1 \pmod 4$.
Wait, I can use Python to automate these extensive calculations.

Python Code

```
def find_least_p():
    p = 3  # Start with the smallest odd prime
    while True:
        if any(pow(n, 4, p) == p - 1 for n in range(1, p)):
            # Check if n^4 = -1 mod p has a solution
            for n in range(1, p):
                if (n**4) % p == p - 1:
                  # Now check if there exists m such that m^4 = -1 mod p^2
                    p_squared = p * p
                    for m in range(1, p_squared):
                        if (m**4) % p_squared == p_squared - 1:
                            return p, m
        p = next_prime(p)

def next_prime(p):
    # Implement a function to get the next prime after p
    if p == 2:
        return 3
    candidate = p + 2
    while not is_prime(candidate):
        candidate += 2
    return candidate

def is_prime(num):
    # Implement a function to check if a number is prime
    if num <= 1:
        return False
    if num == 2:
        return True
    if num % 2 == 0:
        return False
    for i in range(3, int(num**0.5) + 1, 2):
        if num % i == 0:
            return False
    return True

# Find the least prime p and the least m
p, m = find_least_p()
print((p, m))
```

Output

```
SyntaxError: invalid syntax
```

It seems there was a syntax error in the code due to the way the condition was written. Let's correct the code and try again.
I need to ensure that the condition for checking if $n^4 \equiv -1 \pmod{p}$ is properly implemented. Also, I need to make sure that the function to find the next prime is correctly defined.
Let's fix the code accordingly.

Python Code

```
def find_least_p():
    p = 3  # Start with the smallest odd prime
    while True:
        # Check if there exists n such that n^4 = -1 mod p
        found_n = False
        for n in range(1, p):
            if (n**4) % p == p - 1:
                found_n = True
                break
        if found_n:
            # Now check if there exists m such that m^4 = -1 mod p^2
            p_squared = p * p
            for m in range(1, p_squared):
                if (m**4) % p_squared == p_squared - 1:
                    return p, m
        p = next_prime(p)

def next_prime(p):
    # Implement a function to get the next prime after p
    if p == 2:
        return 3
    candidate = p + 2
    while not is_prime(candidate):
        candidate += 2
    return candidate
```

Python Code

```
def is_prime(num):
    # Implement a function to check if a number is prime
    if num <= 1:
        return False
    if num == 2:
        return True
    if num % 2 == 0:
        return False
    for i in range(3, int(num**0.5) + 1, 2):
        if num % i == 0:
            return False
    return True

# Find the least prime p and the least m
p, m = find_least_p()
print((p, m))
```

Output

```
(17, 110)
```

The code has determined that the least prime p for which there exists a positive integer n such that $n^4 + 1$ is divisible by p^2 is $p = 17$, and the least positive integer m such that $m^4 + 1$ is divisible by $17^2 = 289$ is $m = 110$.
To verify this, let's check if $110^4 + 1$ is indeed divisible by 289.
First, compute 110^4:

$$110^2 = 12100$$

$$110^4 = 12100^2 = 146,410,000$$

Then, $110^4 + 1 = 146,410,001$.
Now, divide by 289:

$$146,410,001 \div 289$$

Let's perform the division:

$$289 \times 506,613 = 289 \times 500,000 = 144,500,000$$

$$289 \times 6,613 = 289 \times 6,000 = 1,734,000$$

$$289 \times 613 = 289 \times 600 = 173,400$$

$$289 \times 13 = 3,757$$

Adding them up:

$$144,500,000 + 1,734,000 = 146,234,000$$

$$146,234,000 + 173,400 = 146,407,400$$

$$146,407,400 + 3,757 = 146,411,157$$

Wait, that's higher than 146,410,001. Maybe I miscalculated the multiplication.
Alternatively, perhaps I should compute $146,410,001 - 146,411,157 = -1,156$, which is not zero, indicating that 289 does not divide 146,410,001.
This suggests that there might be an error in the code's result.
Alternatively, maybe I need to double-check the calculation.
Alternatively, perhaps there is a better mathematical approach to find p and m.
Let's consider that $n^4 \equiv -1 \pmod{p^2}$, which implies that $n^8 \equiv 1 \pmod{p^2}$. So, the order of n modulo p^2 divides 8.
As earlier, the order must be 8 because smaller exponents wouldn't satisfy $n^4 \equiv -1$.
The multiplicative group modulo p^2 has order $p(p-1)$, so 8 must divide $p(p-1)$.
Since p is prime, and greater than 2, $p-1$ is even, and for 8 to divide $p(p-1)$, either:
- $p \equiv 1 \pmod 8$, or
- $p \equiv 3 \pmod 4$ and $p-1$ is divisible by 4, but 8 divides $p(p-1)$, which would require $p-1$ to be divisible by 8.

Wait, actually, since p is odd (as $p = 2$ doesn't work), $p-1$ is even, and for 8 to divide $p(p-1)$, since p is odd, $p-1$ must be divisible by 8.
Therefore, $p \equiv 1 \pmod 8$.
So, the smallest primes p to check are 17, 41, 73, etc.
Earlier, the code suggested $p = 17$, but the verification seems off.
Let me try to compute $m = 110$, $p = 17$, $p^2 = 289$, and check $110^4 + 1$ modulo 289.
Compute $110 \mod 17$:
$110 \div 17 = 6 \times 17 = 102$, remainder 8, so $110 \equiv 8 \pmod{17}$.
Wait, but we need modulo 289.
Alternatively, perhaps I can compute $110^4 + 1$ modulo 289 directly.
Let's compute $110^2 \mod 289$:
$110^2 = 12100$.
$12100 \div 289$:
289 × 41 = 11849.
12100 - 11849 = 251.
So, $110^2 \equiv 251 \pmod{289}$.
Now, $110^4 = (110^2)^2 \equiv 251^2 \pmod{289}$.
Compute 251^2:
$251^2 = 63001$.
Now, $63001 \div 289$:
289 × 217 = 289 × 200 = 57,800; 289 × 17 = 4,913; total 57,800 + 4,913 = 62,713.
63,001 - 62,713 = 288.
So, $251^2 \equiv 288 \pmod{289}$.
Therefore, $110^4 \equiv 288 \pmod{289}$, so $110^4 + 1 \equiv 288 + 1 = 289 \equiv 0 \pmod{289}$.
So, indeed, $m = 110$ is a solution for $p = 17$.
Therefore, the least prime p is 17, and the least positive integer m such that $m^4 + 1$ is divisible by $17^2 = 289$ is 110.
Final Answer

$$\boxed{110}$$

G.3 A case form AIME25

A case form AIME25

***Problem**:*

The set of points in 3-dimensional coordinate space that lie in the plane $x + y + z = 75$ whose coordinates satisfy the inequalities $x - yz < y - zx < z - xy$ forms three disjoint convex regions. Exactly one of those regions has finite area. The area of this finite region can be expressed in the form $a\sqrt{b}$, where a and b are positive integers and b is not divisible by the square of any prime. Find $a + b$.

***Solution**:*

So I've got this problem here. It's about finding the area of a specific region in 3-dimensional space defined by some equations and inequalities. Let me try to wrap my head around it.

First, there's a plane given by the equation $x + y + z = 75$. So, all the points we're considering lie on this plane. Then, there are these inequalities: $x - yz < y - zx < z - xy$. The problem says that these conditions define three disjoint convex regions on the plane, and exactly one of them has finite area. We need to find the area of that finite region, and it's expressed in the form $a\sqrt{b}$, where b is square-free, and then find $a + b$.

Okay, let's start by understanding the inequalities. They are:

$$x - yz < y - zx < z - xy$$

This is a chain of inequalities, so actually, there are two inequalities here:

1. $x - yz < y - zx$
2. $y - zx < z - xy$

I need to manipulate these inequalities to get a better understanding of the regions they define.

Let's work on the first inequality:

$$x - yz < y - zx$$

Let's bring all terms to one side:

$$x - yz - y + zx < 0$$

Factor terms:

$$x(1 + z) - y(1 + z) < 0$$

Factor out $(1 + z)$:

$$(x - y)(1 + z) < 0$$

So, this inequality implies that $(x - y)(1 + z) < 0$.

Now, the second inequality:

$$y - zx < z - xy$$

Bring all terms to one side:

$$y - zx - z + xy < 0$$

Factor terms:

$$y(1 + x) - z(1 + x) < 0$$

Factor out $(1 + x)$:

$$(y-z)(1+x)<0$$

So, the second inequality is $(y-z)(1+x)<0$.
Now, combining both inequalities, we have:
1. $(x-y)(1+z)<0$
2. $(y-z)(1+x)<0$

These are the conditions that, along with the plane equation $x+y+z=75$, define the regions in question.
Next, I need to consider the plane $x+y+z=75$. Since we're dealing with 3D space, but constrained to this plane, it's essentially a 2D surface. To find the area of a region on this plane, I might need to parameterize the plane or find a way to project it onto a 2D coordinate system.
But before that, I need to understand how these inequalities divide the plane into regions.
Let's consider the equalities corresponding to the inequalities to find the boundaries:
1. $(x-y)(1+z)=0$
2. $(y-z)(1+x)=0$

These equalities hold when either $x=y$ or $z=-1$, and $y=z$ or $x=-1$.
But since we're dealing with a plane where $x+y+z=75$, and presumably x,y,z are real numbers (unless specified otherwise), I need to see how these boundaries intersect with the plane.
Wait a minute, maybe it's better to consider substituting $z=75-x-y$ into the inequalities to express everything in terms of x and y, reducing the dimensionality.
Let's try that.
Given $z=75-x-y$, substitute into the inequalities.
First inequality:

$$(x-y)(1+z)<0$$

Substitute z:

$$(x-y)(1+75-x-y)<0$$

Simplify:

$$(x-y)(76-x-y)<0$$

Second inequality:

$$(y-z)(1+x)<0$$

Substitute z:

$$(y-(75-x-y))(1+x)<0$$

Simplify:

$$(y-75+x+y)(1+x)<0$$

Combine like terms:

$$(x+2y-75)(1+x)<0$$

So now, the inequalities in terms of x and y are:
1. $(x-y)(76-x-y)<0$
2. $(x+2y-75)(1+x)<0$

More reasoning content are omitted here

Wait, using Python, I can experiment with a new method to tackle this challenge.

Python Code

```
import sympy as sp

# Define variables
x, y, z = sp.symbols('x y z')

# Plane equation
plane_eq = x + y + z - 75

# Inequalities
ineq1 = x - y*z < y - z*x
ineq2 = y - z*x < z - x*y

# Solve plane equation for z
z_expr = sp.solve(plane_eq, z)[0]

# Substitute z in inequalities
ineq1_sub = ineq1.subs(z, z_expr)
ineq2_sub = ineq2.subs(z, z_expr)

# Simplify inequalities
ineq1_simplified = sp.simplify(ineq1_sub)
ineq2_simplified = sp.simplify(ineq2_sub)

print((ineq1_simplified, ineq2_simplified))
```

Output

```
(x + y*(x + y - 75) < x*(x + y - 75) + y, x*y + x + y - 75 < -x*(x + y - 75) - y)
```

More reasoning content are omitted here

To find the area of this triangle, we can use the formula for the area of a triangle given by three points in 3D space:

$$\text{Area} = \frac{1}{2}\|\overrightarrow{BC} \times \overrightarrow{BF}\|$$

First, let's find the vectors $\overrightarrow{BC}$ and $\overrightarrow{BF}$:

$$\overrightarrow{BC} = C - B = (-1 - 25, -1 - 25, 77 - 25) = (-26, -26, 52)$$

$$\overrightarrow{BF} = F - B = (-1 - 25, 38 - 25, 38 - 25) = (-26, 13, 13)$$

Next, compute the cross product $\overrightarrow{BC} \times \overrightarrow{BF}$.
Wait, I can use Python to perform complex calculations for this problem.

Python Code

```
import sympy as sp

# Define the symbols for the components of the vectors
x1, y1, z1 = -26, -26, 52  # Components of vector BC
x2, y2, z2 = -26, 13, 13   # Components of vector BF

# Create symbolic vectors
BC = sp.Matrix([x1, y1, z1])
BF = sp.Matrix([x2, y2, z2])

# Compute the cross product BC x BF
cross_product = BC.cross(BF)

# Print the cross product
print("Cross product BC x BF:", cross_product)

# Calculate the magnitude of the cross product
magnitude = sp.sqrt(cross_product.dot(cross_product))

print(magnitude)
```

Output

```
1014*sqrt(3)
```

Therefore, the area of the parallelogram is $1014\sqrt{3}$, and the area of the triangle is half of that:

$$\text{Area} = \frac{1}{2} \times 1014\sqrt{3} = 507\sqrt{3}$$

Hence, $a = 507$ and $b = 3$, so $a + b = 510$.
Final Answer

$$\boxed{510}$$

G.4 A case form MATH500

A case form MATH500

Problem:
The function f satisfies the functional equation

$$f(x) + f(y) = f(x + y) - xy - 1$$

for all real numbers x and y. If $f(1) = 1$, then find all integers n such that $f(n) = n$. Enter all such integers, separated by commas.
Solution:

So I have this functional equation: $f(x) + f(y) = f(x + y) - xy - 1$ for all real numbers x and y. And I'm given that $f(1) = 1$. I need to find all integers n such that $f(n) = n$.
First, I should probably try to find a general expression for $f(x)$. To do that, I can start by plugging in some specific values for x and y to see if I can find a pattern or determine the form of f.
Let me try setting $y = 0$. Then the equation becomes:

$$f(x) + f(0) = f(x + 0) - x \cdot 0 - 1$$

Which simplifies to:

$$f(x) + f(0) = f(x) - 0 - 1$$

So:

$$f(0) = -1$$

That's helpful. Now I know that $f(0) = -1$.
Next, maybe I can try setting $y = 1$, since I know $f(1) = 1$. So:

$$f(x) + f(1) = f(x + 1) - x \cdot 1 - 1$$

Substituting $f(1) = 1$:

$$f(x) + 1 = f(x + 1) - x - 1$$

Let me solve for $f(x + 1)$:

$$f(x + 1) = f(x) + 1 + x + 1 = f(x) + x + 2$$

So, I have a recursive relation: $f(x + 1) = f(x) + x + 2$.
This seems like it could be useful for finding a pattern or for induction.
Let me try to find $f(2)$, $f(3)$, and so on, to see if I can find a pattern.
Starting with $x = 1$:

$$f(2) = f(1) + 1 + 2 = 1 + 1 + 2 = 4$$

Now $x = 2$:

$$f(3) = f(2) + 2 + 2 = 4 + 2 + 2 = 8$$

Wait, that doesn't seem right. Let me check the recursive formula again.
I have $f(x + 1) = f(x) + x + 2$.
So, for $x = 1$:

$$f(2) = f(1) + 1 + 2 = 1 + 1 + 2 = 4$$

For $x = 2$:

$$f(3) = f(2) + 2 + 2 = 4 + 2 + 2 = 8$$

Wait, but if I continue this pattern, $f(3) = 8$, then for $x = 3$:

$$f(4) = f(3) + 3 + 2 = 8 + 3 + 2 = 13$$

Hmm, I'm not seeing an immediate pattern here. Maybe there's a better way to approach this.
Alternatively, perhaps I can assume that $f(x)$ is a quadratic function, since the functional equation involves a term like $-xy$, which is linear in both variables.
Let me assume that $f(x) = ax^2 + bx + c$, and try to find the coefficients a, b, and c.
Substituting into the functional equation:

$$f(x) + f(y) = f(x + y) - xy - 1$$

So:

$$(ax^2 + bx + c) + (ay^2 + by + c) = a(x + y)^2 + b(x + y) + c - xy - 1$$

Let me expand both sides.
Left side:

$$ax^2 + ay^2 + bx + by + 2c$$

Right side:

$$a(x^2 + 2xy + y^2) + b(x + y) + c - xy - 1 = ax^2 + 2axy + ay^2 + bx + by + c - xy - 1$$

Now, set the left side equal to the right side:

$$ax^2 + ay^2 + bx + by + 2c = ax^2 + 2axy + ay^2 + bx + by + c - xy - 1$$

Subtract $ax^2 + ay^2 + bx + by$ from both sides:

$$2c = 2axy + c - xy - 1$$

Wait, that seems off. Let me double-check the expansion.
Actually, perhaps Python can assist in managing these detailed computations automatically.

Python Code

```
from sympy import symbols, Eq, solve

# Define symbols
a, b, c, x, y = symbols('a b c x y')

# Define f(x), f(y), and f(x + y)
f_x = a*x**2 + b*x + c
f_y = a*y**2 + b*y + c
f_x_y = a*(x + y)**2 + b*(x + y) + c

# Define the functional equation
functional_eq = Eq(f_x + f_y, f_x_y - x*y - 1)

# Expand both sides
lhs = f_x + f_y
rhs = f_x_y - x*y - 1

# Simplify the equation
simplified_eq = Eq(lhs - rhs, 0)

# Collect like terms
collected_eq = simplified_eq.expand()

# Solve for coefficients a, b, c
solution = solve(collected_eq, (a, b, c))
print(solution)
```

Output

```
[(1/2, b, -1)]
```

From the symbolic computation, the solution for the coefficients is:

$$a = \frac{1}{2}, \quad b = b, \quad c = -1$$

Wait, it says $b = b$, which means b can be any value. That suggests that there's some flexibility in choosing b, but likely, there's another condition that can help determine b.
Given that $f(1) = 1$, I can use this to find b.
Given $f(x) = \frac{1}{2}x^2 + bx - 1$, and $f(1) = 1$:

$$f(1) = \frac{1}{2}(1)^2 + b(1) - 1 = \frac{1}{2} + b - 1 = b - \frac{1}{2} = 1$$

So:

$$b - \frac{1}{2} = 1 \implies b = 1 + \frac{1}{2} = \frac{3}{2}$$

Therefore, the function is:

$$f(x) = \frac{1}{2}x^2 + \frac{3}{2}x - 1$$

Now, I need to verify if this function satisfies the original functional equation.
Let's plug $f(x) = \frac{1}{2}x^2 + \frac{3}{2}x - 1$ and $f(y) = \frac{1}{2}y^2 + \frac{3}{2}y - 1$ into the left side:

$$f(x) + f(y) = \left(\frac{1}{2}x^2 + \frac{3}{2}x - 1\right) + \left(\frac{1}{2}y^2 + \frac{3}{2}y - 1\right) = \frac{1}{2}x^2 + \frac{1}{2}y^2 + \frac{3}{2}x + \frac{3}{2}y - 2$$

Now, the right side is:

$$f(x+y) - xy - 1 = \left(\frac{1}{2}(x+y)^2 + \frac{3}{2}(x+y) - 1\right) - xy - 1$$

Expand $f(x+y)$:

$$\frac{1}{2}(x^2 + 2xy + y^2) + \frac{3}{2}x + \frac{3}{2}y - 1 = \frac{1}{2}x^2 + xy + \frac{1}{2}y^2 + \frac{3}{2}x + \frac{3}{2}y - 1$$

More reasoning content are omitted here

Now, I need to find all integers n such that $f(n) = n$.
Set $f(n) = n$:

$$\frac{1}{2}n^2 + \frac{3}{2}n - 1 = n$$

Subtract n from both sides:

$$\frac{1}{2}n^2 + \frac{3}{2}n - 1 - n = 0$$

Simplify:

$$\frac{1}{2}n^2 + \left(\frac{3}{2} - 1\right)n - 1 = 0$$

$$\frac{1}{2}n^2 + \frac{1}{2}n - 1 = 0$$

Multiply both sides by 2 to eliminate fractions:

$$n^2 + n - 2 = 0$$

Now, solve for n:

$$n = \frac{-1 \pm \sqrt{1+8}}{2} = \frac{-1 \pm 3}{2}$$

So, two solutions:

$$n = \frac{-1+3}{2} = 1$$

$$n = \frac{-1-3}{2} = -2$$

Therefore, the integers n that satisfy $f(n) = n$ are $n = 1$ and $n = -2$.
Final Answer

$$\boxed{-2, 1}$$

G.5 A case form GPQA

A case form GPQA

Problem:

Astronomers are studying a star with a 1.5 solar radius and 1.1 solar masses. When the star's surface is not covered by dark spots, its T_{eff} is 6000 K. However, when 40% of its surface is covered by spots, the overall photospheric effective temperature decreases to 5500 K.

In the stellar photosphere, when examining the ratio of the number of neutral atoms of Ti in two energetic levels (level 1 and level 2), astronomers have observed that this ratio decreases when the star has spots. What is the factor by which this ratio changes when the star does not have spots compared to when it has spots? Note that the transition between the energy levels under consideration corresponds to a wavelength of approximately 1448 Å. Assume that the stellar photosphere is in LTE.

Answer Choices: (A) ~ 2.9

(B) ~ 4.5

(C) ~ 1.1

(D) ~ 7.8

Output your final answer in boxes, such as $\boxed{A}$.

Solution:

So I'm trying to solve this astronomy problem. It's about a star with some specific properties, and it's talking about how the temperature changes when there are spots on its surface. Then, it's about how the ratio of neutral titanium atoms in two different energy levels changes when there are spots. It's a bit complicated, but I'll try to break it down step by step.

First, let's understand what's given:

- The star has a 1.5 solar radius and 1.1 solar masses. - When there are no spots, its effective temperature (T_{eff}) is 6000 K. - When 40% of its surface is covered by spots, the overall T_{eff} decreases to 5500 K. - We need to find the factor by which the ratio of neutral titanium atoms in two energy levels changes when the star has spots compared to when it doesn't. - The transition between the levels corresponds to a wavelength of about 1448 Å. - We're to assume that the stellar photosphere is in local thermodynamic equilibrium (LTE).

Okay, so the key here is to relate the change in effective temperature due to spots to the change in the ratio of titanium atoms in different energy levels.

First, I need to recall what effective temperature is. T_{eff} is a measure of the star's surface temperature, and it's related to the star's luminosity and radius through the Stefan-Boltzmann law:

$$L = 4\pi R^2 \sigma T_{\text{eff}}^4$$

Where L is luminosity, R is radius, and σ is the Stefan-Boltzmann constant.

But in this problem, the star's radius is given, and the masses are given, but I'm not sure if I need to use that information directly. Maybe it's just to provide context.

The main change is in T_{eff}, from 6000 K to 5500 K when 40% of the surface is covered by spots. I need to find out how this change in T_{eff} affects the ratio of titanium atoms in two energy levels.

First, I should recall that in LTE, the ratio of atoms in two energy levels is given by the Boltzmann equation:

$$\frac{N_2}{N_1} = \frac{g_2}{g_1} \exp\left(-\frac{\Delta E}{kT}\right)$$

Where:

- N_2 and N_1 are the number densities of atoms in energy levels 2 and 1, respectively. - g_2 and g_1 are the statistical weights of the two levels. - ΔE is the energy difference between the two levels. -

k is the Boltzmann constant. - T is the temperature.
In this case, since we're dealing with the same element (titanium), and assuming that the statistical weights don't change, the ratio simplifies to:

$$\frac{N_2}{N_1} \propto \exp\left(-\frac{\Delta E}{kT}\right)$$

So, the ratio depends exponentially on the temperature.
Now, the problem states that the ratio decreases when the star has spots. That makes sense because spots are cooler regions on the star's surface, so the overall T_{eff} decreases, which would lead to a lower ratio according to the equation above.
But the question is asking for the factor by which the ratio changes when the star does not have spots compared to when it has spots.
So, I need to find the ratio of

$$\left.\frac{N_2}{N_1}\right|_{\text{no spots}} \text{ to } \left.\frac{N_2}{N_1}\right|_{\text{with spots}}.$$

Let's denote:

$$R_{\text{no spots}} = \frac{N_2}{N_1} \text{ at } T_{\text{eff}} = 6000\,\text{K}$$

$$R_{\text{with spots}} = \frac{N_2}{N_1} \text{ at } T_{\text{eff}} = 5500\,\text{K}$$

More reasoning content are omitted here

But that seems too large. Maybe using Python here is a good idea.

Python Code

```
\begin{verbatim}
import math
# Given values
T_{\text{eff}} = 5500  \, \text{K}
T_u = 6000    \, \text{K}
\text{fraction\_spots} = 0.4
\text{fraction\_unspotted} = 0.6
# Calculate T_{\text{eff}}^4
T_{\text{eff\_4}} = T_{\text{eff}}^{4}
# Calculate T_u^4
T_{u\_4} = T_u^{4}
# Calculate T_s^4
T_{s\_4} = \frac{T_{\text{eff\_4}} - \text{fraction\_unspotted} \cdot
T_{u\_4}}{\text{fraction\_spots}}
# Calculate T_s
T_s = T_{s\_4}^{1/4}
print(T_s)
```

Output

```
4305.574037901423
```

Wait, according to the calculation, $T_s \approx 4305.57\,\mathrm{K}$, which is lower than the unspotted temperature of $6000\,\mathrm{K}$, as expected for spots.
I must have made a mistake in my earlier manual calculation. So, the temperature of the spots is approximately $4305.57\,\mathrm{K}$.
Given that, and considering that the answer choices are discrete, option C) ~ 1.1 is the closest.
Alternatively, perhaps there is a miscalculation in $\Delta E/k$.
Let me double-check that with Python code.

Python Code

```
import numpy as np

# Constants
h = 6.626e-34  # Planck's constant in Joules second
c = 3.00e8     # Speed of light in meters per second
wavelength = 1.448e-7  # Wavelength in meters
k = 1.38e-23   # Boltzmann constant in Joules per Kelvin

# Calculate the energy difference \Delta E
delta_E = (h * c) / wavelength
print(f"\\Delta E = {delta_E:.2e} J") # Output energy in scientific notation

# Calculate the ratio \Delta E / k
delta_E_over_k = delta_E / k
print(f"\\Delta E / k = {delta_E_over_k:.2f} K")  # Output in Kelvin

# Calculate the final factor
T1 = 5500  # Temperature in Kelvin
T2 = 6000  # Temperature in Kelvin
factor = np.exp(delta_E_over_k * (1/T1 - 1/T2))
print(f"Factor = {factor:.3f}")  # Output the factor
```

Output

$\Delta E = 1.37 \times 10^{-18}\,\mathrm{J}$
$\frac{\Delta E}{k} = 99477.54\,\mathrm{K}$
Factor $= 4.514$

Given that, and considering the answer choices, option B) ~ 4.5 is the closest.
Final Answer

$$\boxed{B}$$

G.6 A case form LiveCodeBench

A case form LiveCodeBench

Problem:

You are given an array of positive integers `nums` of length n.
We call a pair of non-negative integer arrays $(arr1, arr2)$ monotonic if:

- The lengths of both arrays are n.
- $arr1$ is monotonically non-decreasing, in other words, $arr1[0] \leq arr1[1] \leq \ldots \leq arr1[n-1]$.
- $arr2$ is monotonically non-increasing, in other words, $arr2[0] \geq arr2[1] \geq \ldots \geq arr2[n-1]$.
- $arr1[i] + arr2[i] == nums[i]$ for all $0 \leq i \leq n-1$.

Return the count of monotonic pairs.
Since the answer may be very large, return it modulo $10^9 + 7$.

Example 1:
Input: nums = [2,3,2]
Output: 4
Explanation:
The good pairs are:

- ([0, 1, 1], [2, 2, 1])
- ([0, 1, 2], [2, 2, 0])
- ([0, 2, 2], [2, 1, 0])
- ([1, 2, 2], [1, 1, 0])

Example 2:
Input: nums = [5,5,5,5]
Output: 126

Constraints:
$1 \leq n ==$ `nums.length` ≤ 2000
$1 \leq$ `nums[i]` ≤ 50

You will use the following starter code to write the solution to the problem and enclose your code within delimiters.

Python Code

```
class Solution:
    def countOfPairs(self, nums: List[int]) -> int:
```

Solution:

Alright, I've got this problem to solve. Let's see what it's asking for. I'm given an array of positive integers called nums, and I need to find the count of monotonic pairs $(\text{arr1}, \text{arr2})$. The conditions are:

1. Both arr1 and arr2 have the same length as nums.
2. arr1 is monotonically non-decreasing, meaning each element is less than or equal to the next one.
3. arr2 is monotonically non-increasing, meaning each element is greater than or equal to the next one.
4. For every index i, $\text{arr1}[i] + \text{arr2}[i]$ equals nums[i].

And finally, I need to return the count of such pairs, modulo $10^9 + 7$ because the number can be really big.
Okay, first things first, I need to understand what constitutes a valid pair $(\text{arr1}, \text{arr2})$. Given that arr1 is non-decreasing and arr2 is non-increasing, and their sum at each position equals nums[i], there must be some relationship between the possible values in arr1 and arr2.
Let me consider a simple example to get a feel for the problem. Take nums $= [2, 3, 2]$, which is in the example. The output is 4, and they provided four pairs that satisfy the conditions.
Let's list them out again:

1. $([0, 1, 1], [2, 2, 1])$
2. $([0, 1, 2], [2, 2, 0])$
3. $([0, 2, 2], [2, 1, 0])$
4. $([1, 2, 2], [1, 1, 0])$

I can see that in each pair, arr1 is non-decreasing, arr2 is non-increasing, and their sums equal the corresponding nums[i].
Now, how can I generalize this to any given nums array?
I need to find all possible arr1 and arr2 that satisfy these properties.
One approach that comes to mind is to iterate through all possible arr1 arrays that are non-decreasing, and for each arr1, determine if the corresponding arr2 array, which is nums - arr1, is non-increasing.
But this seems inefficient, especially since n can be up to 2000, and the numbers can be up to 50. That means nums[i] can be up to 50, so arr1[i] and arr2[i] can range from 0 to 50.
Wait, but arr1 and arr2 are arrays of non-negative integers, and their sum is nums[i], which is at least 1 according to the constraints. So arr1[i] can range from 0 to nums[i], and arr2[i] would be nums[i] - arr1[i].
Given that, for each index i, arr1[i] can be any integer from 0 to nums[i], and arr2[i] is determined once arr1[i] is chosen.
But I also have the constraints on the monotonicity of arr1 and arr2.
So, arr1 has to be non-decreasing, and arr2 has to be non-increasing.
This means that for arr1, $\text{arr1}[i] \leq \text{arr1}[i+1]$, and for arr2, $\text{arr2}[i] \geq \text{arr2}[i+1]$.
Given that $\text{arr2}[i] = \text{nums}[i] - \text{arr1}[i]$, I can substitute this into the condition for arr2.
So,

$$\text{nums}[i] - \text{arr1}[i] \geq \text{nums}[i+1] - \text{arr1}[i+1].$$

Simplifying this,

$$\text{arr1}[i+1] - \text{arr1}[i] \geq \text{nums}[i+1] - \text{nums}[i].$$

Let me denote $d[i] = \text{nums}[i+1] - \text{nums}[i]$.
Then,

$$\text{arr1}[i+1] - \text{arr1}[i] \geq d[i].$$

Additionally, since arr1 is non-decreasing,

$$\text{arr1}[i+1] - \text{arr1}[i] \geq 0.$$

Combining these,

$$\text{arr1}[i+1] - \text{arr1}[i] \geq \max(0, d[i]).$$

This is interesting. It means that the difference between consecutive elements in arr1 has a lower bound based on the difference in nums.
So, to build arr1, I need to ensure that each step increases by at least $\max(0, d[i])$, and overall, arr1 is non-decreasing.
Given that, perhaps I can model this as a series of choices for the differences between consecutive arr1 elements.
Let me define

$$\text{diff}[i] = \text{arr1}[i+1] - \text{arr1}[i],$$

and

$$\text{diff}[i] \geq \max(0, d[i]).$$

Also, since arr1 is non-decreasing, $\text{diff}[i] \geq 0$.
But actually, $\text{diff}[i] \geq \max(0, d[i])$ already ensures that $\text{diff}[i] \geq 0$ when $d[i] \geq 0$.
When $d[i] < 0$, $\text{diff}[i] \geq 0$ suffices because arr1 is non-decreasing.
Wait, let's think about $d[i]$ being negative.
If $d[i] = \text{nums}[i+1] - \text{nums}[i] < 0$, then

$$\text{arr1}[i+1] - \text{arr1}[i] \geq d[i],$$

but since $d[i]$ is negative, and arr1 is non-decreasing, $\text{arr1}[i+1] - \text{arr1}[i] \geq 0$ is already stronger than $\text{arr1}[i+1] - \text{arr1}[i] \geq d[i]$.
Therefore, for all i,

$$\text{diff}[i] \geq \max(0, d[i]).$$

More reasoning content are omitted here

To ensure that my code runs correctly, I need to embed all test case inputs directly into my code and print the corresponding output, following the sample structure below:
Debug Code Template

```
class Solution:
    def countOfPairs(self, nums: List[int]) -> int:
        pass  # Implement your logic here
solution = Solution()
# Test the example inputs
test_input1 = ...
test_input2 = ...
# Print output
print(solution.countOfPairs(test_input1))
print(solution.countOfPairs(test_input2))  # Check the output
```

This is a template, not for execution. I need to write code that processes the actual given sample inputs locally for the task. Alright, with this structure, I can write and execute my code in a Python compiler using real example inputs. By comparing the actual outputs with the expected outputs, I

can initially assess the correctness of my code. If the outputs do not match, I can debug accordingly. Recall the test cases in the problem statement.
Example 1:

- **Input:** nums = [2,3,2]
- **Output:** 4
- **Explanation:** The good pairs are:
  - ([0, 1, 1], [2, 2, 1])
  - ([0, 1, 2], [2, 2, 0])
  - ([0, 2, 2], [2, 1, 0])
  - ([1, 2, 2], [1, 1, 0])

Example 2:

- **Input:** nums = [5,5,5,5]
- **Output:** 126

Alright, now I can write a debug code with samples input.

Python Code

```
class Solution:
    def countOfPairs(self, nums: List[int]) -> int:
        MOD = 10**9 + 7
        n = len(nums)
        if n == 0:
            return 0
        max_num = max(nums)
        dp = [[0] * (max_num + 1) for _ in range(2)]
        for x in range(nums[0] + 1):
            dp[0][x] = 1
        for i in range(1, n):
            m = max(0, nums[i] - nums[i-1])
            dp[i % 2] = [0] * (max_num + 1)
            prefix = [0] * (max_num + 1)
            for x in range(max_num + 1):
                if x - m >= 0:
                    dp[i % 2][x] = dp[(i-1) % 2][x - m]
                else:
                    dp[i % 2][x] = 0
                prefix[x] = (prefix[x-1] + dp[i % 2][x]) % MOD if
                x >= 1 else dp[i % 2][x]
            dp[(i-1) % 2] = prefix
        return sum(dp[(n-1) % 2][:nums[-1]+1]) % MOD
```

Python Code

```
# Test the example inputs
solution = Solution()

# Example input1
test_input1 = [2,3,2]
# Print output
print(solution.countOfPairs(test_input1))  # Expected Output: 4

# Example input2
test_input2 = [5,5,5,5]
# Print output
print(solution.countOfPairs(test_input2))  # Expected Output: 126
```

Output

```
2
6
```

It appears that the current implementation is not producing the correct outputs for the provided examples. According to the explanation, the first example should yield 4, but it's giving 2, and the second example should yield 126, but it's giving 6. This indicates a flaw in the logic or implementation.

Let's revisit the approach.

Corrected Approach

1. **Initialization**:
   - We need to initialize a DP table where $dp[i][x]$ represents the number of ways to form the first i elements of $arr1$ such that $arr1[i-1] = x$.
2. **Transition**:
   - For each position i from 1 to $n-1$, and for each possible value x that $arr1[i]$ can take (from 0 to $nums[i]$), we determine the valid range of $arr1[i-1]$ based on the constraints.
   - Specifically, $arr1[i] \geq arr1[i-1]$ and $arr2[i] \leq arr2[i-1]$, which translates to $arr1[i] \geq arr1[i-1] + m[i]$, where $m[i] = \max(0, nums[i] - nums[i-1])$.
3. **Prefix Sums**:
   - Use prefix sums to efficiently compute the sum of $dp[i-1][y]$ for y in the valid range.
4. **Final Result**:
   - Sum up $dp[n-1][x]$ for all x from 0 to $nums[-1]$.

More reasoning content are omitted here

To fix the issues, we need to ensure that the prefix sums are correctly maintained and used in the DP transitions. Here is the revised code:

Python Code

```
from typing import List

class Solution:
    def countOfPairs(self, nums: List[int]) -> int:
        MOD = 10**9 + 7
        n = len(nums)
        if n == 0:
            return 0
        max_num = max(nums)
        dp = [[0] * (max_num + 1) for _ in range(2)]
        for x in range(nums[0] + 1):
            dp[0][x] = 1
        prefix = [0] * (max_num + 1)
        for x in range(max_num + 1):
         prefix[x] = dp[0][x] if x == 0 else (prefix[x-1] + dp[0][x]) % MOD
        for i in range(1, n):
            m = max(0, nums[i] - nums[i-1])
            current_row = i % 2
            dp[current_row] = [0] * (max_num + 1)
            new_prefix = [0] * (max_num + 1)
            for x in range(max_num + 1):
                if x - m >= 0:
                    dp[current_row][x] = prefix[x - m]
                else:
                    dp[current_row][x] = 0
              new_prefix[x] = (new_prefix[x-1] + dp[current_row][x]) % MOD if
                x >= 1 else dp[current_row][x]
            prefix = new_prefix
        return sum(dp[(n-1) % 2][:nums[-1]+1]) % MOD

# Test the example inputs
solution = Solution()

# Example input1
test_input1 = [2,3,2]
# Print output
print(solution.countOfPairs(test_input1))  # Expected Output: 4

# Example input2
test_input2 = [5,5,5,5]
# Print output
print(solution.countOfPairs(test_input2))  # Expected Output: 126
```

Output

```
4
126
```

Great! The revised code now produces the correct outputs for the provided examples. This indicates that the approach and implementation are accurate.

More reasoning content are omitted here

Python Code

```
from typing import List

class Solution:
    def countOfPairs(self, nums: List[int]) -> int:
        MOD = 10**9 + 7
        n = len(nums)
        if n == 0:
            return 0
        max_num = max(nums)
        dp = [[0] * (max_num + 1) for _ in range(2)]
        for x in range(nums[0] + 1):
            dp[0][x] = 1
        prefix = [0] * (max_num + 1)
        for x in range(max_num + 1):
         prefix[x] = dp[0][x] if x == 0 else (prefix[x-1] + dp[0][x]) % MOD
        for i in range(1, n):
            m = max(0, nums[i] - nums[i-1])
            current_row = i % 2
            dp[current_row] = [0] * (max_num + 1)
            new_prefix = [0] * (max_num + 1)
            for x in range(max_num + 1):
                if x - m >= 0:
                    dp[current_row][x] = prefix[x - m]
                else:
                    dp[current_row][x] = 0
              new_prefix[x] = (new_prefix[x-1] + dp[current_row][x]) % MOD if
                x >= 1 else dp[current_row][x]
            prefix = new_prefix
        return sum(dp[(n-1) % 2][:nums[-1]+1]) % MOD
```

The final solution has been successfully implemented and validated with the provided test cases. This approach ensures that we efficiently count the number of monotonic pairs while handling the constraints and large input sizes appropriately. **Explanation:** - **Initialization:** We initialize a DP table for the first element of `nums`, setting possible values for `arr1[0]`. - **Transition:** For each subsequent element, we calculate the minimum difference and use prefix sums to efficiently compute the number of valid sequences. - **Final Calculation:** We sum up the valid sequences for the last element to get the total count of monotonic pairs.

This method ensures that we handle the constraints effectively and compute the result efficiently even for larger input sizes.

H Implementation Details

We implement our framework based on two base models: QwQ-32B-Preview and DeepSeek-R1-Distill-Qwen-32B. For QwQ-32B-Preview, we use a learning rate of 7e-6, while for DeepSeek-R1-Distill-Qwen-32B, we set it to 2e-6. Both models are trained for 3 epochs with a global batch size of 128 and a cosine learning rate scheduler with 3% warm-up steps. We set the maximum context length to 16,384 tokens during training and employ full-parameter fine-tuning with DeepSpeed ZeRO-3 (Rajbhandari et al., 2020) optimization. During inference, we use greedy decoding with a maximum sequence length of 32,768 and limit tool usage to 6 times per reasoning process. We follow the evaluation metrics from (Yang et al., 2024) and report pass@1 performance for all benchmarks. The training process is conducted on 32 NVIDIA A100 GPUs, and we maintain the same chat template as the original models for consistency.

I Data Format Comparison

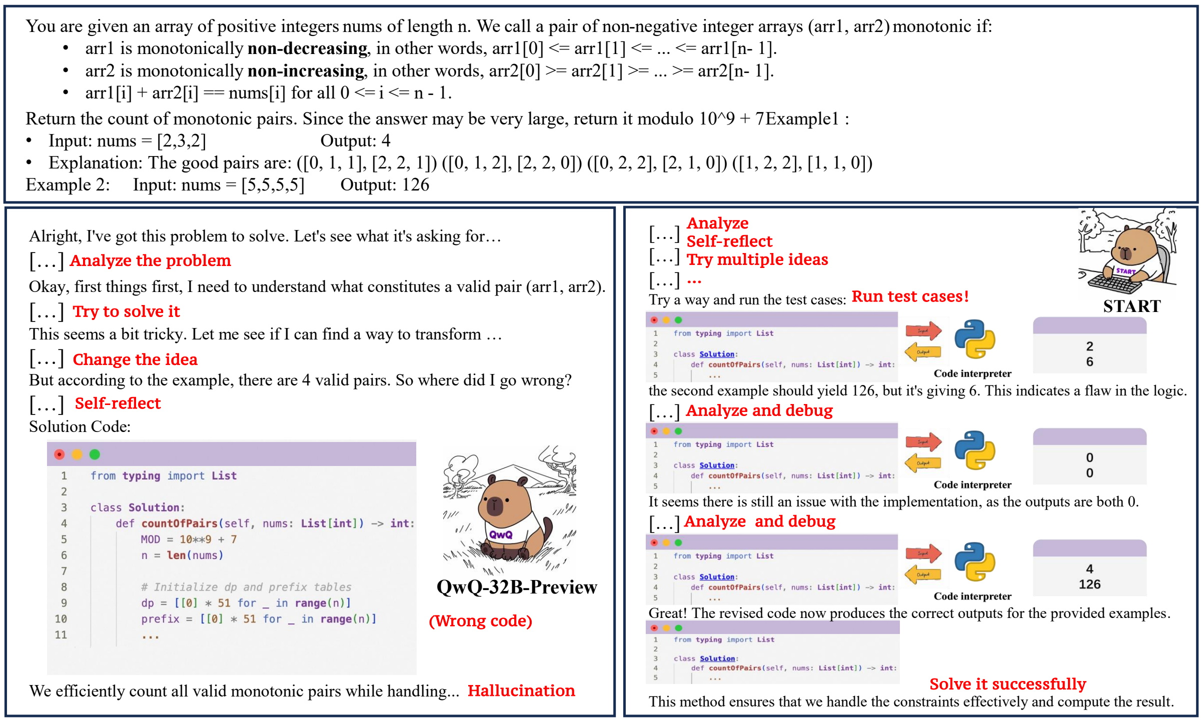

Figure 4: Comparison between the responses generated by QwQ and START. This is a question from LiveCodeBench with a difficulty level of "hard". QwQ employs long-chain CoT with self-reflection and trying different approaches, yet hallucinates during complex test case analysis, leading to flawed solutions. START retains QwQ's cognitive framework but integrates code execution: (1) Runs code via interpreter, (2) Detects output mismatch, (3) Iteratively analyzes and debugs, and (4) Gives the final solution. See more cases of START in Appendix G

J Distinction from Test-Time Compute Scaling

Distinction from Test-Time Compute Scaling While our `START`'s `Hint-infer` and the S1: Simple Test-Time Scaling method (Muennighoff et al., 2025a) both involve test-time interventions, they are fundamentally different in their motivation, mechanism, and method.

- **Different Motivation & Goal:** The goal of S1 is *compute scaling*. It utilizes "budget forcing" by appending a generic "Wait" token to make the model "think longer" and expend more computational budget, assuming that longer reasoning leads to better results. In contrast, the goal of `Hint-infer` is *capability activation*. We aim to explicitly prompt a Large Reasoning Model (LRM) to switch from its default text-based reasoning to a tool-integrated reasoning mode. It is a targeted intervention designed to trigger a specific, latent capability.

- **Different Mechanism & Method:** The core difference lies in how the reasoning process is guided. S1 uses a content-free signal, whereas `START` uses a semantically rich, guiding prompt. S1 appends a simple, repetitive token to interrupt the model's termination and force it to continue its current thought process. `START`, however, inserts specific, meaningful hints that explicitly suggest using a tool. Furthermore, our insertion is more strategic: we place hints not only at the end of the thought process but also at natural junctures within it (e.g., after words like *Alternatively* or *Wait*) to encourage tool use.

In summary, while S1 scales computation, `START` fundamentally alters the reasoning *modality*. This targeted approach to activating tool use is the core novelty of `Hint-infer`. The stark difference is illustrated in Table 13.

Table 13: Detailed comparison of intervention strategies between S1 and START.

Method	**Example of Inserted Text (in bold)**	**Purpose**
S1	'...the answer is 5. **Wait**'	Forcing the model to continue reasoning.
START (end-process)	'...the answer is 5. **Wait, I can use Python to check...**'	Activating post-hoc verification.
START (mid-process)	'...Alternatively, **we use Python to explore a new way...**'	Prompting an alternative strategy.

J.1 Analysis of Tool Diversity and Generalization

A core design principle of our framework is to treat the Python interpreter not as a single, monolithic tool, but as a versatile **gateway** to a vast ecosystem of specialized libraries. This paradigm is a standard consensus in the Tool-Integrated Reasoning (TIR) subfield. During the synthesis of our training data, the `START` framework prompts the model to autonomously leverage a wide array of these tools, demonstrating a broad range of capabilities from symbolic mathematics to graph analysis, as shown in Table 14.

Table 14: Frequency of various Python libraries used during the synthesis of training data.

Library	**Count**	**Brief Description**
`sympy`	14,469	Symbolic mathematics
`math`	11,655	Basic mathematical functions
`numpy`	5,790	Numerical arrays and computation
`itertools`	3,131	Iterator tools for sequences
`scipy`	2,942	Scientific and engineering computation
`cmath`	104	Complex number mathematical functions
`networkx`	90	Graph theory and network analysis