Anonymous authors

000

001

003

005 006 007

008

009 010 011

012 013

014

015

016

017

018

019

021

023

024

025

026

027

028

029

031

034

039

040 041

042

043 044

045

046

047

051

052

Paper under double-blind review

ABSTRACT

Competitive programming presents great challenges for Code LLMs due to its intensive reasoning demands and high logical complexity. However, current Code LLMs still rely heavily on real-world data, which limits their scalability. In this paper, we explore a fully synthetic approach: training Code LLMs with entirely generated tasks, solutions, and test cases, to empower code reasoning models without relying on real-world data. To support this, we leverage feature-based synthesis to propose a novel data synthesis pipeline called SynthSmith. SynthSmith shows strong potential in producing diverse and challenging tasks, along with verified solutions and tests, supporting both supervised fine-tuning and reinforcement learning. Based on the proposed synthetic SFT and RL datasets, we introduce the X-Coder model series, which achieves a notable pass rate of 62.9 avg@8 on LiveCodeBench v5 and 55.8 on v6, outperforming DeepCoder-14B-Preview and AReal-boba²-14B despite having only 7B parameters. In-depth analysis reveals that scaling laws hold on our synthetic dataset, and we explore which dimensions are more effective to scale. We further provide insights into code-centric reinforcement learning and highlight the key factors that shape performance through detailed ablations and analysis. Our findings demonstrate that scaling high-quality synthetic data and adopting staged training can greatly advance code reasoning, while mitigating reliance on real-world coding data. Our code, data and models will be made publicly available.

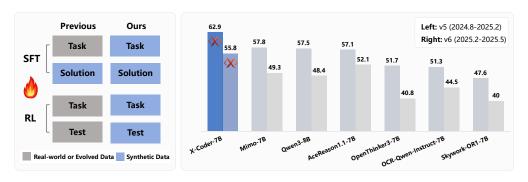


Figure 1: Left: SynthSmith generates high-quality synthetic tasks, solutions, and test cases to support both SFT and RL training. Right: Avg@8 results on LiveCodeBench. X-Coder achieves significant performance gains on competitive programming using *fully* synthetic data.

1 Introduction

As code language models advance, reasoning-focused models such as OpenAI-o1-ioi (OpenAI et al., 2025) have reached expert-level performance in programming. Classic benchmarks including HumanEval (Chen et al., 2021; Liu et al., 2023) and MBPP (Austin et al., 2021) have been largely

solved, whereas tasks from LiveCodeBench (Jain et al., 2024) and Codeforces continue to demand deeper reasoning and more complex algorithmic problem solving.

Recently, DeepSeek-R1 (Guo et al., 2025) has opened two opportunities for further boosting the reasoning capabilities of Code LLMs. The first is supervised fine-tuning (SFT) (Ouyang et al., 2022) on long Chain-of-Thought (CoT) demonstrations to distill reasoning patterns into student models (Hugging Face, 2025; Labs, 2025; Liu et al., 2025a). The second is reinforcement learning (RL) (Schulman et al., 2017) with GRPO (Shao et al., 2024b) and related algorithms to refine reasoning foundation models (Luo et al., 2025; Fu et al., 2025a; He et al., 2025).

Both pathways have proven effective but face a common bottleneck: progress on competitive programming remains constrained by the scarcity of datasets. Widely used collections such as APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), and TACO (Li et al., 2023) are heavily reused during post-training. They remain too modest in scale to support continued benefits and still lack the level of sufficiently challenging, diverse, and scalable. Meanwhile, collecting new real-world data tailored for competitive programming is also challenging. Although recent work has synthesized rewritten or evolutionary variants (Luo et al., 2024; Liu et al., 2025a; Xu et al., 2025) from existing resources, their diversity and complexity remain tightly bounded by the seed tasks.

To address this gap, we explore a fully synthetic approach: training Code LLMs with fully generated tasks, solutions, and test cases. Building on this insight, we present *SynthSmith*, a novel coding data synthesis pipeline tailored for competitive programming. To enable the synthesis of diverse and challenging competitive programming tasks, SynthSmith extends feature-based methods (Wang et al., 2025) with competition-oriented feature extraction, dedicated feature integration, and multi-style task construction. SynthSmith further supports the development of high-quality solutions and tool-based test case generation, both of which are cross-validated through the proposed dual-verification strategy. Thereby, SynthSmith demonstrates strong potential in producing scalable and challenging tasks, together with verified solutions and tests, offering support for both SFT and subsequent RL. Starting from a base model (e.g., Qwen3-8B-Base) or a non-reasoning model (e.g., Qwen2.5-Coder-7B-Instruct), we present the X-Coder series, which achieves significant performance gains on challenging LiveCodeBench v5 and v6 without relying on any real-world data, as shown in Figure 1. Beyond this, built upon verl (Sheng et al., 2025), we present an RL infrastructure featuring automated high-concurrency code validation, leveraging the CPUs of all distributed machines to support efficient and large-scale code execution.

Our in-depth analysis examines (i) whether synthetic SFT data scale effectively and which dimensions scale more favorably; (ii) the role of code-centric reinforcement learning, including the "goodgets-better" principle and RL's resilience to noisy supervision; (iii) the factors that shape performance (long- vs. short-CoT, effects of solution verification, task style, and data-selection strategies); and (iv) the bottlenecks that limit code reasoning, together with the chained relationship among task difficulty, reasoning length, and pass rate. We further conduct case studies to uncover cognitive behaviors that emerge after SFT and RL, including reward hacking and undesirable patterns.

We make the following contributions:

- (1) We explore a fully synthetic approach and propose a novel data synthesis pipeline tailored for competitive programming, producing high-quality datasets for both SFT and RL stages.
- (2) We train both base and non-reasoning LLMs under an SFT-then-RL paradigm to develop the X-Coder model series, which achieves significant performance gains on LiveCodeBench v5 (avg@8: 62.9) and v6 (avg@8: 55.8), along with extensive analyses and ablations.
- (3) We introduce an optimized infrastructure for code RL, featuring a dedicated sandbox environment that speeds up code execution and improves training efficiency.

2 SYNTHSMITH: SYNTHESIS OF COMPETITION-LEVEL CODING DATA

We introduce SynthSmith, a fully synthetic framework for constructing competitive programming tasks that support both the SFT and RL stages. Figure 2 illustrates the SynthSmith pipeline, which consists of four key steps: (i) generating novel and challenging problems (with the capacity for easy scaling in quantity); (ii) constructing diverse and comprehensive input test cases for each problem (including boundary and stress tests); (iii) producing high-quality candidate solutions; and (iv) em-

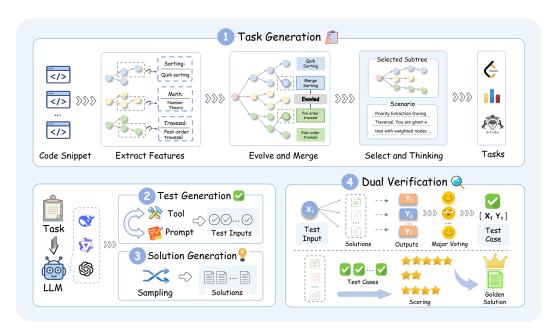


Figure 2: **Framework of SynthSmith.** SynthSmith first extracts and evolves competitive programming related features from small-scale code instruction data and merges them into tree structures. It then samples subtrees from the feature tree, selects a compatible feature set, and formulates a scenario that naturally integrates these consistent features. Novel tasks are generated based on a proposed scenario according to specific styles. Advanced reasoning models are used to synthesize solutions and tests for the generated tasks, which are further cross-verified using the proposed dual-verification strategy to yield reliable test outputs and the top solution.

ploying a dual-verification strategy that cross-checks solutions with test cases to yield more accurate test outputs and more reliable solutions.

(i) Task Generation. Inspired by EpiCoder (Wang et al., 2025), which generates novel programming tasks through a feature-based framework by combining sampled features into problem scenarios, we extend this approach with three key improvements to synthesize diverse and complex tasks tailored for competitive programming. First, instead of relying on broad definitions of features, we explicitly extract and evolve competition-related features from 10k code snippets in the TACO dataset (Li et al., 2023) using GPT-40-0513 (detailed in §C.1). Second, formulating competitive scenarios from a rich feature tree is non-trivial, as LLMs often oversimplify complex prompts into trivial cases, thereby reducing both diversity and difficulty. To address this, we adopt a two-stage process that separates feature selection from scenario formulation: first, selecting mutually consistent features for meaningful composition; and second, formulating hint-free tasks that demand genuine reasoning. We further incorporate one-shot prompting to improve task understanding and instructionfollowing. Third, we adapt the synthesis method to support multi-style task generation, covering Codeforces¹-style tasks (standard input/output with imaginable narrative contexts), LeetCode-style² tasks (starter code with predefined function signatures), and AtCoder³-style tasks (concise specifications with minimal explanations), thereby enhancing task diversity. Examples of the task generation process are provided in §C.2, together with difficulty estimates on generated tasks in §C.3.

(ii) Test Input Generation. Obtaining sufficient and accurate test cases is a formidable challenge. Problems from competitive programming platforms often do not provide test cases, or only provide a limited number, due to platform constraints. This results in insufficient quantity, difficulty, and coverage of test cases during RL training. To address the inherent scarcity of test cases in synthesized data, we investigate two complementary methods for generating the input component of the test

¹https://codeforces.com/

²https://leetcode.com/

³https://atcoder.jp/

case. The *Prompting-based* method instructs the LLM to interpret the problem's input constraints and directly generate multiple test inputs, covering both standard cases and edge-case instances. The *Tool-based* method leverages CYaRon⁴, a dedicated test case generation library, enabling the LLM to construct test inputs by invoking functions documented within the library after understanding the problem. For each task, we generate a set of n test case inputs $[x_1, x_2, \ldots, x_n]$. Detailed description of test input generation is provided in §E, and a comparative analysis is presented in Sec 4.

- (iii) Candidate Solutions Generation. For each task, we generate multiple candidate solutions using advanced open-source reasoning LLMs, obtaining m answers $[A^1, A^2, \ldots, A^m]$. We verify that each candidate solution includes a complete reasoning process and a Python code implementation, and we ensure the absence of syntax errors through static analysis methods based on Abstract Syntax Tree (AST). Filtering criteria are provided in §D.1.
- (iv) Dual-Verification of Solutions and Test Cases. To ensure the robustness and reliability of both the generated solutions and the constructed test cases, we adopt a dual-verification strategy. Step 1 of this strategy extends the principle of self-consistency (Wang et al., 2023) by applying majority voting across candidate solutions from multiple LLMs, which mitigates model-specific biases and enhances generalization, thereby yielding a reliable test output for each input. Step 2 then identifies the top-performing candidate solution by incorporating test case difficulty weighting alongside a hold-out validation set.

Step 1: Verification of Test Cases via Consensus Voting. First, we establish a preliminary ground truth for each test case input. For a given input x_i , we execute all candidate solutions to obtain a set of outputs $\{y_i^1, y_i^2, \ldots, y_i^m\}$, where $y_i^j = A^j(x_i)$. A provisional ground truth output \hat{y}_i is determined via majority voting:

$$\hat{y}_i = \underset{y}{\operatorname{argmax}} \sum_{j=1}^m \mathbb{I}(y_i^j = y) \quad , \tag{1}$$

where $\mathbb{I}(\cdot)$ is the indicator function. This yields a candidate test set $\mathcal{T}_{candidate} = \{(x_1, \hat{y}_1), \dots, (x_n, \hat{y}_n)\}$. Crucially, we posit that not all test cases are of equal importance; boundary or edge cases are critical for robust evaluation. We therefore introduce a weighting function $w(x_i) \to w_i$ that assigns a higher score to more challenging test cases. The weight w_i is determined by a set of heuristics based on input characteristics, such as character or token count, structural complexity, or semantic novelty, which serve as proxies for difficulty.

Step 2: Verification of Solutions via Weighted Evaluation and Hold-out Validation. To ensure that our selected "golden" solution generalizes beyond the generated data, we partition the candidate test set. We randomly sample a subset of $\mathcal{T}_{candidate}$ (e.g., 50%) to form a hold-out validation set, \mathcal{T}_{val} . The remaining data constitutes our primary weighted test suite, \mathcal{T}_{golden} . The dual-verification process culminates in selecting the golden answer, A_{golden} . A candidate solution A^j is first evaluated on \mathcal{T}_{golden} using a weighted score. The top-performing candidate, A'_{golden} , is identified as:

$$A'_{golden} = \underset{A^j \in \{A^1, \dots, A^m\}}{\operatorname{argmax}} \sum_{(x_i, \hat{y}_i) \in \mathcal{T}_{golden}} w_i \cdot \mathbb{I}(A^j(x_i) = \hat{y}_i) \quad . \tag{2}$$

The final confirmation of A_{golden} is contingent upon its performance on the unseen hold-out set \mathcal{T}_{val} . We verify that A'_{golden} also achieves the highest (or a competitively high) unweighted accuracy on \mathcal{T}_{val} relative to other candidates. This additional validation step ensures that the selected solution is not merely overfitted to the specifics of the weighted test cases but demonstrates superior, generalizable correctness. The detailed algorithm is provided in §F.

Finally, we obtain A_{golden} and T_{golden} for each task q. The pair $[q, A_{golden}]$ is used to compute the SFT loss, and $[q, T_{golden}]$ are used for RL via the GRPO algorithm.

Discussion. Compared to rewriting-based data synthesis methods (Luo et al., 2024; Liu et al., 2025a), SynthSmith reduces reliance on seed tasks by formulating novel tasks from evolved competitive features. Compared with EpiCoder, it generates more challenging tasks and selects high-quality solutions via a dual-verification strategy, yielding a 21% absolute performance gain on Live-CodeBench v5 (Figure 5c). Moreover, SynthSmith extends data synthesis to the RL stage, showing that synthetic RL data can further improve performance beyond the SFT model as shown in Table 1.

⁴https://github.com/luogu-dev/cyaron

Table 1: Performance on LiveCodeBench v5. X-Coder shows strong coding expertise with fewer, fully synthetic tasks, and achieves additional gains through subsequent RL stages. †: OpenThinker3 integrates human-written tasks with synthetic math tasks. rStar-Coder augments real-world coding tasks with synthesized rewrites for mixed training, whereas X-Coder relies on fully synthetic tasks.

Model	Base Model	SFT	RL	Size	Data	Task	Metric	Score
	SFT Baselines							
Bespoke-Stratos (Labs, 2025)	Qwen2.5-Instruct (Qwen et al., 2025)	1	X	7B	17k	Real	pass@1	16.2
OpenThinker3 (Guha et al., 2025)	Qwen2.5-Instruct	1	X	7B	1,200k	Mixed [†]	-	51.7
OlympicCoder (Hugging Face, 2025)	Qwen2.5-Coder-Instruct (Hui et al., 2024)	1	X	7B	100k	Real	-	40.9
OCR-Qwen-Instruct (Ahmad et al., 2025)	Qwen2.5-Coder-Instruct	/	X	7B	736k	Real	avg@64	51.3
rStar-Coder (Liu et al., 2025a)	Qwen2.5-Coder-Instruct	1	X	7B	580K	Mixed [†]	avg@16	57.3
Qwen3-8B (Yang et al., 2025)	Qwen3-8B-Base	1	X	8B	-	Real	-	57.5
	RL Baselines							'
Skywork-OR1 (He et al., 2025)	R1-Distilled-Qwen (DeepSeek-AI, 2025)	X	/	7B	124k	Real	avg@32	47.6
DeepCoder-Preview (Luo et al., 2025)	R1-Distilled-Qwen	X	/	14B	24k	Real	pass@1	57.9
AReal-boba ² (Fu et al., 2025a)	R1-Distilled-Qwen	X	/	14B	24k	Real	avg@32	58.1
	SFT-then-RL Baselines (Stag	e 1)						
AceReason1.1-SFT (Liu et al., 2025b)	Qwen2.5-Math (Yang et al., 2024)	1	X	7B	2.2M	Real	avg@8	51.2
MiMo-SFT (Xiaomi et al., 2025)	MiMo-Base	1	X	7B	500k	Unclear	avg@8	52.3
Klear-Reasoner-SFT (Su et al., 2025)	Qwen3-Base (Yang et al., 2025)	1	X	8B	1500k	Real	avg@8	58.5
X-Coder-Qwen2.5-SFT	Qwen2.5-Coder-Instruct	1	X	7B	200k	Syn	avg@8	60.3 ± 2.5
X-Coder-Qwen3-SFT	Qwen3-8B-Base	1	X	8B	200k	Syn	avg@8	59.4 _{±2.0}
	SFT-then-RL Baselines (Stag	e 2)						
AceReason1.1	AceReaon1.1-SFT	1	/	7B	-	Real	avg@8	57.2
MiMo	MiMo-SFT	1	/	7B	130k	Unclear	avg@8	57.8
Klear-Reasoner	Klear-Reasoner-SFT	1	/	8B	106k	Real	avg@8	61.6
X-Coder-Qwen2.5	X-Coder-Qwen2.5-SFT	1	/	7B	40k	Syn	avg@8	$62.9_{\pm 1.8}$
X-Coder-Qwen3	X-Coder-Qwen3-SFT	1	1	8B	40k	Syn	avg@8	64.0 _{±2.5}

3 EXPERIMENT

Setup. In this study, we adopt GPT-o3-mini (OpenAI, 2025) for task formulation, Deepseek-R1-0528 (DeepSeek-AI, 2025) and Qwen3-235B-A22B-Thinking-2507 (Yang et al., 2025) for solution sampling, and R1-0528 for test case generation. Statistics for SFT datasets are provided in §D.2. For SFT, we set the learning rate at 5e-5, with a global batch size of 128 to train 8 epochs. For RL, the reward is defined as the fraction of passed tests among all given tests (detailed in §B.2). The program executes in an isolated sandbox environment deployed with Redis, which supports optimized concurrent code testing (infrastructure details are provided in §B.5). Training configurations and costs are supplemented in §B.4.

Evaluation. We evaluate Code LLMs on LiveCodeBench (Jain et al., 2024) v5 (covering problems released between Aug. 2024 and Feb. 2025) and v6 (Feb. to May 2025), which are the most widely used benchmarks for code reasoning models. Baselines are documented in §B.6. To ensure a fair comparison, we use Qwen2.5-Coder-7B-Instruct and Qwen3-8B-Base as backbones, and report the avg@8 pass rate using a sampling temperature of 0.6 with top-p 0.95 to align with the baselines.

3.1 Main Results

As shown in Table 1, during the SFT stage, X-Coder-SFT achieves an avg@8 pass rate of 60.3. Compared with RL baselines, X-Coder-SFT exhibits a clear advantage over 14B-based RL models (e.g., DeepCoder-Preview-14B, AReal-boba²-14B), despite those models being built on the stronger foundation R1-Distilled-Qwen. Relative to SFT-then-RL models, X-Coder further boosts its performance after RL, reaching 62.9. On Qwen3-Base, X-Coder attains an avg@8 pass rate of 64.0.

3.2 SFT EXPERIMENTS AND ANALYSIS

During the SFT stage, we investigate a central question: Can the SFT dataset be effectively scaled, and along which dimension should it be scaled more favorably? To explore this, we expand the SFT dataset from two perspectives: increasing the number of unique tasks and enlarging the number of solutions per task. We design seven subsets (v1–v6): v1–v4 increase the number of unique tasks (32k, 64k, 128k, and 192k unique prompts, each with 1 solution), while v5–v6 expand the number of solutions per task (16k unique prompts with 4 solutions, and 8k unique prompts with 8 solutions).

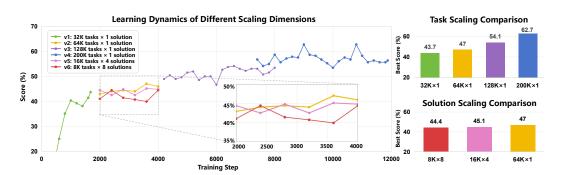


Figure 3: Scaling laws on the SFT dataset generated by SynthSmith. Left: Performance comparison of on LiveCodebench v5 to examine scaling trend. Right: Performance comparison across scaling unique tasks and scaling solutions per task.

The results in Figure 3 reveal a promising scaling trend, where v4 > v3 > v2 > v1, with performance steadily improving from 43.7% to 62.7%. The scaling is data-efficient, as v4 achieves significant performance gains over baselines that use the same backbone but many more samples.

Furthermore, the comparison v2 ($64k \times 1$) > v5 ($16k \times 4$) > v6 ($8k \times 8$) shows that scaling the number of unique tasks is more effective than increasing the number of solutions per task. When computational budget is fixed, expanding task diversity is more efficient for improving generalization.

3.3 RL EXPERIMENTS AND ANALYSIS

Our investigation of the RL stage uncovers the following key insights into its role and behavior:

- (i) RL as a Powerful Refiner. RL fine-tuning is not merely an incremental add-on but a powerful optimization step. As shown in Table 1, when applied to a converged SFT model using only code data, it yields a substantial 4.6% absolute gain in average pass-rate. This highlights RL's unique capability to refine policy beyond the distribution of the initial supervised dataset.
- (ii) The "Good-gets-Better" Principle. RL performance is tightly coupled to the strength of the SFT initializer. Using two SFT models trained on similar data distributions but with different LiveCodeBench scores as starting points, we observe in Figure 4 that, under identical RL settings, the stronger initializer consistently attains higher rewards. A stronger SFT foundation acts as a multiplier, enabling the RL agent to explore a more promising policy space and achieve a higher performance ceiling. This underscores the importance of a high-quality initial model as a prerequisite for effective RL.

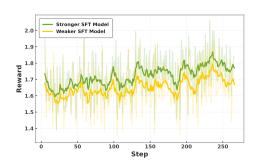


Figure 4: Reward comparison of weak and strong SFT models as RL initialization.

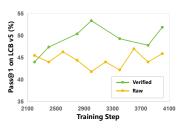
(iii) Resilience to Noisy Supervision. Contrary to the common assumption that reinforcement learning

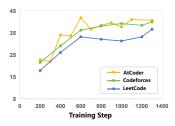
requires pristine reward signals, our experiments reveal a resilience to data imperfections during RL. The model also effectively benefits from synthetic test cases, suggesting that RL can be successfully deployed in scenarios with large-scale but imperfect feedback (Wang et al., 2020; Lv et al., 2025a), significantly lowering the barrier to code RL data collection.

4 ABLATION STUDY

Despite the strong performance of X-Coder, the determinants of high-quality synthetic data for SFT remain insufficiently understood. To elucidate these factors, we conduct a comprehensive ablation along six axes: (i) the effect of the proposed dual-verification strategy; (ii) the impact of distinct thinking types in solutions; (iii) the influence of task styles; (iv) a head-to-head comparison of tasks

produced by SynthSmith versus those from open-source synthetic datasets; (v) data-selection strategies to identify patterns that shape downstream performance; and (vi) comparison of prompting-based and tool-based test generation strategies.







(a) Raw vs. verified solutions.

(b) Task style comparison.

(c) Comparison with EpiCoder.

Figure 5: Ablations on verification, task style, and task sources.

Q1: Dual-verification for High-Quality Data Curation. To mitigate the noise introduced by stochastically sampled solutions, we employ a dual-verification strategy for data curation. This strategy first leverages the self-consistency principle to identify the most likely correct solution from multiple candidates. Subsequently, these candidate solutions are executed against a comprehensive set of test cases to verify their functional correctness and robustness, thereby capturing subtle runtime errors (e.g., ValueError, IndexError, or Timeout) that are undetected by static analysis methods like AST checks. The efficacy of this approach is validated by our empirical results, as shown in Figure 5a. Using an identical backbone (Qwen2.5-Coder-7B-Instruct) and dataset (64k tasks), the model trained on verified solutions significantly outperforms its counterpart trained on raw solutions. However, this quality assurance comes at a considerable computational cost. For instance, fully verifying 200k samples necessitates the generation of 1.6 million long-CoT trajectories and 24 million test executions. This overhead establishes a clear trade-off, as prior work (Li et al., 2025; Gandhi et al., 2025) indicates that models can still learn effectively from unverified long-CoT data, making raw-solution training a more resource-efficient, albeit potentially less performant, alternative.

Q2: Solution Types: Long CoT vs. Short CoT. The length of CoT proves to be a critical factor for performance, with longer CoTs yielding superior results despite higher training costs. To demonstrate this, we compare the Qwen2.5-Coder-7B-Instruct trained on solutions generated by DeepSeek-R1-0528 (Long-CoT) and Qwen3-235B-A22B-Instruct-2507 (Short-CoT) for an identical set of tasks (200k). As shown in Table 2, the long-CoT approach achieves a 17.2% absolute performance gain. This substantial improvement justifies

Table 2: Long CoT vs. Short CoT.

	Epoch	LCB v5	LCB v6
	3	35.0	29.3
Short-CoT	8	43.1	37.6
	Δ	+8.1	+8.3
	3	42.9	36.0
Long-CoT	8	60.3	53.5
_	Δ	+17.4	+17.5

the increased computational demand, which manifests as a slower convergence requiring 8–10 epochs compared to the 2–3 epochs needed for short-CoT data.

Q3: Ablation on Task Style. We evaluate the effect of task styles (AtCoder, Codeforces, and Leet-Code) by synthesizing three corpora of 32k tasks each (8k unique problems with 4 solutions per problem) from identical input features. For each corpus, solutions are generated with DeepSeek-R1-0528 and used to fine-tune the Qwen2.5-Coder-7B-Instruct. Results are shown in Figure 5b. Although AtCoder-style tasks yield slightly higher scores, we adopt Codeforces-style as the predominant format in our demonstration dataset (Codeforces: AtCoder: LeetCode = 70: 15: 15), reflecting its prominence as the mainstream competitive-programming platform.

Q4: Tasks from SynthSmith vs Tasks from EpiCoder-380k. We randomly select 64k tasks from our SFT dataset and another 64k from EpiCoder-380k, and use DeepSeek-R1-0528 to complete solutions. Figure 5c shows that tasks from SynthSmith yield a 21% absolute performance gain, demonstrating its ability to produce high-quality tasks tailored for competitive programming.

Q5: Data Selection. To investigate data utilization efficiency, we explore task selection strategies for competitive programming. Specifically, we evaluate three approaches: (1) difficulty-based selection, where GPT-4o-2411 assigns discrete difficulty scores to tasks, simulating the Codeforces rating system; (2) rationale-based selection, where DeepSeek-R1-0528 generates CoT reasoning for

each task, and tasks that elicit longer reasoning traces are prioritized; and (3) random selection as a baseline. For validation, each strategy independently samples a 50k-task subset from a 200k-task pool. Solutions are generated by Qwen3-235B-A22B-Instruct-2507, and models were trained for three epochs with a 16k context length.

As shown in Figure 6, tasks that induce longer CoT are regarded as more valuable training data for competitive programming, as they demand deeper reasoning and are potentially more challenging.

Q6: Prompting-based vs. Tool-based Test Generation. We compare prompting-based and tool-based test generation using tasks from CodeContests (Li et al., 2022). We leverage the corresponding golden solutions to evaluate the accuracy and complexity of the tests produced by the two approaches.

The results in Table 3 show that the tool-based approach outperforms the prompting-based method across multiple dimensions. Qualitatively, it is more versatile, capable of systematically generating random, scalable, boundary, and stress tests, which are essential for robust

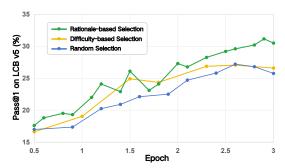


Figure 6: Comparison of data selection strategies.

code evaluation but not supported by prompting-based methods.

Quantitatively, the tool-based approach achieves a higher pass rate on ground-truth solutions (87.9% vs. 77.4%), confirming that its test cases are more accurate and reliable. It also generates more challenging and discriminative tests, as reflected by the lower consensus ratio (78.8% vs. 82.0%), which indicates stronger effectiveness in uncovering subtle bugs. In addition, the tool-based generator provides broader test coverage, albeit at a higher computational cost.

Table 3: Comparison of Prompting-based and Tool-based Test Generation. The tool-based approach excels in test diversity, accuracy, and the ability to generate more challenging test cases.

	Random	Scalable	Boundary	Stress	Cost	Avg Tests	Min Tests	Max Tests	Consensus	Pass Rate
Prompting-based	Х	Х	Х	X	low	13.6	5	15	82.0%	77.4%
Tool-based	\checkmark	\checkmark	\checkmark	\checkmark	high	18.3	5	27	78.8%	87.9%

5 DISCUSSION

In this section, we present an in-depth analysis of the main challenges in code reasoning. Reasoning models often suffer from assertion errors, highlighting persistent reasoning limitations on harder tasks. We further identify a mediation pattern among task difficulty, reasoning length, and pass rate, and extend our analysis with test-time scaling experiments and case studies on cognitive behavior, reward hacking, and undesirable patterns.

Failure Analysis. We classify failure cases into five types: Assertion Error (runnable code producing wrong outputs), No Code Block Generated (reasoning without a final solution until tokens are exhausted), Incomplete Code Block (partial code without closure), Function Signature Mismatch (incorrect function definition), and Syntax Error (complete code with syntax issues). The error distribution in Table 4 indicates that the primary bottleneck lies in reasoning capability, with most errors

Table 4: Distribution of failure cases for 16 rollouts on LiveCodeBench v5 (268 tasks).

Error Type	Qwen2.5-Coder-7B-Instruct	Qwen3-8B	X-Coder-SFT	X-Coder
Assertion Error	212.7 ± 11.0	108.9 ± 5.0	83.3 ± 4.5	79.5 ± 6.2
No Code Block	6.5 ± 8.2	7.7 ± 1.2	21.9 ± 3.7	11.8 ± 3.9
Incomplete Code Block	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	1.0 ± 0.8
Signature Mismatch	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	1.0 ± 0.8
Syntax Error	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	8.3 ± 2.2

Table 5: Performance analysis by reasoning token length.

Token	Total	Passed	Easy	Medium	Hard
0-5k	38	38	30/30 (100.0%)	8/8 (100.0%)	0/0 (-)
5k-10k	41	38	16/17 (94.1%)	14/16 (87.5%)	8/8 (100.0%)
10k-15k	41	32	10/11 (90.9%)	14/19 (73.7%)	8/11 (72.7%)
15k-20k	52	36	4/4 (100.0%)	16/16 (100.0%)	16/32 (50.0%)
20k-25k	36	15	1/1 (100.0%)	9/13 (69.2%)	5/22 (22.7%)
>25k	60	10	0/0 (-)	2/14 (14.3%)	8/46 (17.4%)
Total	268	169	61/63 (96.8%)	63/86 (73.3%)	45/119 (37.8%)

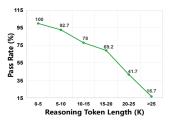


Figure 7: Pass rate by token.

stemming from assertion failures. Although reasoning models can generate structurally valid and executable code, they still struggle to produce logically correct solutions for complex tasks.

After RL, X-Coder reduces assertion errors compared to its SFT counterparts by learning from correctness-based rewards. At the same time, the RL optimization process may introduce instability, leading to issues such as syntax errors, signature mismatches, and other flaws.

Pass Rate by Reasoning Token Length. The results in Table 5 show that the pass rate decreases sharply as reasoning token length increases, exhibiting a clear downward trend. This finding runs counter to the intuitive expectation that greater test-time token usage reflects deeper reasoning and should therefore yield higher accuracy. Instead, we observe a significant chained relationship among problem difficulty, reasoning length, and pass rate: problem difficulty is positively correlated with reasoning length, while reasoning length is strongly negatively correlated with pass rate. This mediation pattern can be summarized as higher difficulty \rightarrow longer reasoning length \rightarrow lower pass rate.

Test-time Scaling. We compare the pass@k performance of Qwen2.5-Coder-7B-Instruct, Qwen3-8B, X-Coder-7B-SFT, and X-Coder-7B in Figure 8. X-Coder-7B outperforms its foundation model by 51.3 points in pass@16, and matches Qwen3-8B with 8x fewer rollouts. Moreover, X-Coder shows a larger gap between pass@1 and pass@16 compared to Qwen3-8B (19.2 vs. 13.8), indicating greater diversity in the reasoning patterns it can explore. Although RL models begin with higher initial performance than the SFT model, the gap does not expand within 16 rollouts, suggesting that RL improves pass@1 but may not escape its starting point (Wu et al., 2025).

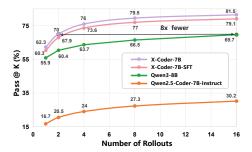


Figure 8: Test-time scaling performance.

Behaviors after SFT and RL. After SFT, the model

frequently exhibits cognitive behaviors such as planning, verification, backtracking, and reflection, as illustrated by the case study in §H.1. This suggests that such behaviors can be directly distilled from the teacher rather than induced by the RL process. During the later stages of RL, the model shows signs of reward hacking, attempting to exploit edge cases for partial rewards instead of producing genuine solutions, as detailed in §H.3. We also observe several bad patterns in code reasoning, including premature termination when the model is aware that the context is running out, recalling memorized submissions in C++ and attempting to translate them into Python, and emitting incomplete code before the context window is exhausted. These cases are illustrated in §H.2.

6 CONCLUSION

In this paper, we explore a fully synthetic approach to competitive programming and propose a novel data synthesis framework that demonstrates how synthetic tasks, solutions, and tests can train large reasoning models to achieve significant performance gains, thereby reducing reliance on real-world data. Building on this framework, we contribute scalable synthetic SFT and RL training sets, supported by a dedicated RL infrastructure, and introduce the X-Coder series. Furthermore, we provide insights into code-centric SFT-then-RL training, ablate key factors that shape performance, and present in-depth analyses with illustrative case studies of code reasoning models.

ETHICS STATEMENT

This work aims to advance large code reasoning models for competitive programming through fully synthetic data. No personal, private, or sensitive information is included in the datasets or experiments, and no ethical risks are associated with this study.

REPRODUCIBILITY STATEMENT

With respect to reproducibility, we affirm our commitment to ensuring that all reported results can be faithfully reproduced, and we will provide the necessary resources and documentation to facilitate replication. The anonymous repository link for reference and reproduction is https://anonymous.4open.science/r/x-coder.

REFERENCES

- Wasi Uddin Ahmad, Sean Narenthiran, Somshubra Majumdar, Aleksander Ficek, Siddhartha Jain, Jocelyn Huang, Vahid Noroozi, and Boris Ginsburg. Opencodereasoning: Advancing data distillation for competitive coding, 2025. URL https://arxiv.org/abs/2504.01943.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.
- Mark Chen, Jerry Tworek, Heewoo Jun, and Qiming Yuan et al. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.
- Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (eds.), Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, pp. 4299–4307, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/d5e2c0adad503c91f91df240d0cd4e49-Abstract.html.
- Ning Dai, Zheng Wu, Renjie Zheng, Ziyun Wei, Wenlei Shi, Xing Jin, Guanlin Liu, Chen Dun, Liang Huang, and Lin Yan. Process supervision-guided policy optimization for code generation. *CoRR*, abs/2410.17621, 2024. doi: 10.48550/ARXIV.2410.17621. URL https://doi.org/10.48550/arXiv.2410.17621.
- DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.
- Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. Areal: A large-scale asynchronous reinforcement learning system for language reasoning, 2025a. URL https://arxiv.org/abs/2505.24298.
- Yuqian Fu, Tinghong Chen, Jiajun Chai, Xihuai Wang, Songjun Tu, Guojun Yin, Wei Lin, Qichao Zhang, Yuanheng Zhu, and Dongbin Zhao. Srft: A single-stage method with supervised and reinforcement fine-tuning for reasoning, 2025b. URL https://arxiv.org/abs/2506.19767.
- Kanishk Gandhi, Ayush K Chakravarthy, Anikait Singh, Nathan Lile, and Noah Goodman. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective STars. In Second Conference on Language Modeling, 2025. URL https://openreview.net/forum?id=QGJ9ttXLTy.
- Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, Ashima Suvarna, Benjamin Feuer, Liangyu Chen, Zaid Khan, Eric Frankel, Sachin Grover, Caroline Choi, Niklas Muennighoff, Shiye Su,

Wanjia Zhao, John Yang, Shreyas Pimpalgaonkar, Kartik Sharma, Charlie Cheng-Jie Ji, Yichuan Deng, Sarah Pratt, Vivek Ramanujan, Jon Saad-Falcon, Jeffrey Li, Achal Dave, Alon Albalak, Kushal Arora, Blake Wulfe, Chinmay Hegde, Greg Durrett, Sewoong Oh, Mohit Bansal, Saadia Gabriel, Aditya Grover, Kai-Wei Chang, Vaishaal Shankar, Aaron Gokaslan, Mike A. Merrill, Tatsunori Hashimoto, Yejin Choi, Jenia Jitsev, Reinhard Heckel, Maheswaran Sathiamoorthy, Alexandros G. Dimakis, and Ludwig Schmidt. Openthoughts: Data recipes for reasoning models, 2025. URL https://arxiv.org/abs/2506.04178.

- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv* preprint arXiv:2501.12948, 2025.
- Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, Siyuan Li, Liang Zeng, Tianwen Wei, Cheng Cheng, Bo An, Yang Liu, and Yahui Zhou. Skywork open reasoner 1 technical report. *arXiv preprint arXiv:2505.22312*, 2025.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, and et al. Measuring coding challenge competence with APPS. In *NeurIPS Datasets and Benchmarks*, 2021.
- Hugging Face. Open r1: A fully open reproduction of deepseek-r1, January 2025. URL https://github.com/huggingface/open-r1.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.
- Bespoke Labs. Bespoke-stratos: The unreasonable effectiveness of reasoning distillation. https://www.bespokelabs.ai/blog/bespoke-stratos-the-unreasonable-effectiveness-of-reasoning-distillation, 2025. Accessed: 2025-01-22.
- Dacheng Li, Shiyi Cao, Tyler Griggs, Shu Liu, Xiangxi Mo, Eric Tang, Sumanth Hegde, Kourosh Hakhamaneshi, Shishir G. Patil, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Llms can easily learn to reason from demonstrations structure, not content, is what matters!, 2025. URL https://arxiv.org/abs/2502.07374.
- Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*, 2023.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.
- Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. rstar-coder: Scaling competitive code reasoning with a large-scale verified dataset, 2025a. URL https://arxiv.org/abs/2505.21297.

- Zihan Liu, Zhuolin Yang, Yang Chen, Chankyu Lee, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Acereason-nemotron 1.1: Advancing math and code reasoning through sft and rl synergy, 2025b. URL https://arxiv.org/abs/2506.13284.
 - Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. Deepcoder: A fully open-source 14b coder at o3-mini level, 2025. Notion Blog.
 - Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=UnUwSIgK5W.
 - Ang Lv, Ruobing Xie, Xingwu Sun, Zhanhui Kang, and Rui Yan. The climb carves wisdom deeper than the summit: On the noisy rewards in learning to reason, 2025a. URL https://arxiv.org/abs/2505.22653.
 - Xingtai Lv, Yuxin Zuo, Youbang Sun, Hongyi Liu, Yuntian Wei, Zhekai Chen, Lixuan He, Xuekai Zhu, Kaiyan Zhang, Bingning Wang, Ning Ding, and Bowen Zhou. Towards a unified view of large language model post-training, 2025b. URL https://arxiv.org/abs/2509.04419.
 - OpenAI. Openai o3-mini: Pushing the frontier of cost-effective reasoning. https://openai.com/index/openai-o3-mini/, 2025. Accessed: 2025-09-11.
 - OpenAI,:, Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, Jerry Tworek, Lorenz Kuhn, Lukasz Kaiser, Mark Chen, Max Schwarzer, Mostafa Rohaninejad, Nat McAleese, o3 contributors, Oleg Mürk, Rhythm Garg, Rui Shu, Szymon Sidor, Vineet Kosaraju, and Wenda Zhou. Competitive programming with large reasoning models, 2025. URL https://arxiv.org/abs/2502.06807.
 - Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022. URL https://arxiv.org/abs/2203.02155.
 - Qwen, :, 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. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.
 - John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations (ICLR)*, 2016.
 - John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.
 - Yunfan Shao, Linyang Li, Yichuan Ma, Peiji Li, Demin Song, Qinyuan Cheng, Shimin Li, Xiaonan Li, Pengyu Wang, Qipeng Guo, Hang Yan, Xipeng Qiu, Xuanjing Huang, and Dahua Lin. Case2Code: Scalable synthetic data for code generation. In Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert (eds.), *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 11056–11069, Abu Dhabi, UAE, January 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.coling-main.733/.

- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024a. URL https://arxiv.org/abs/2402.03300.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv* preprint arXiv:2402.03300, 2024b.
- Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, pp. 1279–1297. ACM, March 2025. doi: 10.1145/3689031.3696075. URL http://dx.doi.org/10.1145/3689031.3696075.
- Zhenpeng Su, Leiyu Pan, Xue Bai, Dening Liu, Guanting Dong, Jiaming Huang, Wenping Hu, Fuzheng Zhang, Kun Gai, and Guorui Zhou. Klear-reasoner: Advancing reasoning capability via gradient-preserving clipping policy optimization, 2025. URL https://arxiv.org/abs/2508.07629.
- Qiushi Sun, Jinyang Gong, Lei Li, Qipeng Guo, and Fei Yuan. Codeevo: Interaction-driven synthesis of code-centric data through hybrid and iterative feedback, 2025. URL https://arxiv.org/abs/2507.22080.
- Jingkang Wang, Yang Liu, and Bo Li. Reinforcement learning with perturbed rewards. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 6202–6209, 2020.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023.* OpenReview.net, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.
- Yaoxiang Wang, Haoling Li, Xin Zhang, Jie Wu, Xiao Liu, Wenxiang Hu, Zhongxin Guo, Yangyu Huang, Ying Xin, Yujiu Yang, et al. Epicoder: Encompassing diversity and complexity in code generation. *arXiv preprint arXiv:2501.04694*, 2025.
- Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution, 2025. URL https://arxiv.org/abs/2502.18449.
- Fang Wu, Weihao Xuan, Ximing Lu, Zaid Harchaoui, and Yejin Choi. The invisible leash: Why rlvr may not escape its origin, 2025. URL https://arxiv.org/abs/2507.14843.
- LLM-Core Xiaomi, :, Bingquan Xia, Bowen Shen, Cici, Dawei Zhu, Di Zhang, Gang Wang, Hailin Zhang, Huaqiu Liu, Jiebao Xiao, Jinhao Dong, Liang Zhao, Peidian Li, Peng Wang, Shihua Yu, Shimao Chen, Weikun Wang, Wenhan Ma, Xiangwei Deng, Yi Huang, Yifan Song, Zihan Jiang, Bowen Ye, Can Cai, Chenhong He, Dong Zhang, Duo Zhang, Guoan Wang, Hao Tian, Haochen Zhao, Heng Qu, Hongshen Xu, Jun Shi, Kainan Bao, Kai Fang, Kang Zhou, Kangyang Zhou, Lei Li, Menghang Zhu, Nuo Chen, Qiantong Wang, Shaohui Liu, Shicheng Li, Shuhao Gu, Shuhuai Ren, Shuo Liu, Sirui Deng, Weiji Zhuang, Weiwei Lv, Wenyu Yang, Xin Zhang, Xing Yong, Xing Zhang, Xingchen Song, Xinzhe Xu, Xu Wang, Yihan Yan, Yu Tu, Yuanyuan Tian, Yudong Wang, Yue Yu, Zhenru Lin, Zhichao Song, and Zihao Yue. Mimo: Unlocking the reasoning potential of language model from pretraining to posttraining, 2025. URL https://arxiv.org/abs/2505.07608.
- Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. WizardLM: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=CfXh93NDgH.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. KodCode: A diverse, challenging, and verifiable synthetic dataset for coding. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 6980–7008, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.365. URL https://aclanthology.org/2025.findings-acl.365/.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement, 2024. URL https://arxiv.org/abs/2409.12122.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Appendix

758 759 760	A	Related Work	16
761 762	В	Training and Evaluation.	16
763		B.1 SFT-then-RL Training	16
764 765		B.2 Reward Function.	17
766		B.3 Training Dynamics	17
767 768		B.4 Training Configs and Costs	18
769 770		B.5 A Distributed Framework for Automated Code Verification	18
771 772		B.6 Baselines	19
773 774	C	Novel Task Synthesis	19
775		C.1 Feature Extraction and Evolution	20
776 777		C.2 Stylized Task Generation for Competitive Programming	22
778		C.2.1 Compatibale Feature Selection	23
779 780		C.2.2 From Feature to Stylized Task	26
781 782		C.3 Task Difficulty Estimates	26
783 784	D	Solution Generation and Quality Assurance	28
785		D.1 Validation on Solution	28
786 787 788		D.2 SFT Dataset Statistics	29
789 790	E	Test Case Generation.	29
791		E.1 Prompting-based Test Generation	29
792 793		E.2 Tool-based Test Generation	30
794 795 796	F	Dual-verification	33
797 798 799	G	Data Leakage Analysis	33
800	Н	Case Study	34
801 802		H.1 Successful Case	34
803		H.2 Bad Case	35
804 805 806		H.3 Reward Hacking and Pretraining Artifacts	37
807 808	Ι	The Use of Large Language Models	37

A RELATED WORK

Task Synthesis for Coding. The research community has long recognized the scarcity of high-quality coding tasks. To address this, Wizard-Coder (Luo et al., 2024) extends Evol-Instruct (Xu et al., 2024) by evolving basic code-instruction data into augmented variants. rStar-Coder (Liu et al., 2025a) further adapts this augmentation strategy to the competitive programming domain. CodeEvol (Sun et al., 2025) introduces a coder–reviewer interaction framework to collaboratively synthesize high-quality instruction–code pairs. Beyond directly evolving or augmenting seed tasks, Case2Code (Shao et al., 2025) infers underlying code implementations from input–output behaviors. Epicoder (Wang et al., 2025) instead generates task scenarios from programming-related features rather than task examples, producing novel tasks with better complexity and diversity.

In competitive programming, the limited availability of seed tasks makes evolution-based methods particularly difficult, as seeding data in this domain is far scarcer than in general-purpose programming. Directly evolving from such limited seeds risks reduced task diversity. In this paper, we extend the feature-based task synthesis framework with three key improvements (detailed in Sec 2), unleashing the potential of feature-based synthesis for competitive programming tasks, yielding a 21.6% absolute improvement on LiveCodeBench directly over EpiCoder (shown in Figure 5c).

Post-training Recipe for Large Code Reasoning Model. From the training perspective, current approaches to building coding-expert LLMs generally fall into three paradigms: (i) purely supervised fine-tuning on real-world tasks or their rewritten or evolved variants (Labs, 2025; Guha et al., 2025; Liu et al., 2025a), (ii) purely reinforcement-based fine-tuning using a GRPO-related (Shao et al., 2024b; He et al., 2025; Luo et al., 2025; Fu et al., 2025a) algorithm, and (iii) reinforcement learning staged after supervised fine-tuning on mixed coding and mathematical data (Liu et al., 2025b; Xiaomi et al., 2025; Su et al., 2025).

High-quality code data is scarcer than mathematical data. Consequently, existing approaches rely heavily on real-world data and lack a stable two-stage recipe for coding expertise, often mixing in mathematics with little evidence of success on code alone. In this paper, we show that stable and consistent improvements in code reasoning can be achieved solely with synthetic data from SynthSmith, while also reducing the risk of data leakage shown in §G.

While recent research seeks to unify SFT and RL into a single stage (Lv et al., 2025b; Fu et al., 2025b), we do not claim a generalized conclusion regarding the optimal strategy for combining these two post-training methods. Instead, we adopt a straightforward SFT-then-RL recipe to better isolate the effect of each stage and separately validate the effectiveness of the synthesis framework.

B TRAINING AND EVALUATION

B.1 SFT-THEN-RL TRAINING

Supervised Fine-tuning. Given a dataset of task–solution pairs $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, the model with parameters θ is trained by minimizing the negative log-likelihood (NLL) of the target solution y conditioned on the task x:

$$J_{\text{SFT}}(\theta) = -\mathbb{E}_{(x,y)\sim\mathcal{D}} \left[\sum_{t=1}^{|y|} \log \pi_{\theta} (y_t \mid x, y_{< t}) \right]. \tag{3}$$

The loss is applied over full long-CoT trajectories, including both reasoning steps and final code, enabling the model to imitate not only the solutions but also the underlying reasoning patterns.

Reinforcement Learning. Proximal Policy Optimization (PPO) (Schulman et al., 2017) is a widely adopted policy gradient method in Reinforcement Learning from Human Feedback (RLHF) (Christiano et al., 2017) for LLM due to its balance between exploration and exploitation and its empirical robustness. The method optimizes a policy π_{θ} by using a clipped surrogate objective to limit policy divergence, incorporating a value function to estimate expected rewards, and an entropy term to encourage exploration. The overall objective function for PPO is designed to maximize the policy

performance while maintaining stability, and it is typically formulated as minimizing the following:

$$J_{\text{PPO}}(\theta) = \mathbb{E}_{s \sim P(S), a \sim \pi_{\theta}(a|s)} \left[\min \left(\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} A(s, a), \operatorname{clip} \left(\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A(s, a) \right) \right]$$
(4)

where the expectation is computed over states s (drawn from distribution P(S)) and actions a (sampled from the current policy $\pi_{\theta}(a \mid s)$), combining the minimum of two terms: (1) the product of the probability ratio $\frac{\pi_{\theta}(a \mid s)}{\pi_{\theta_{\text{old}}}(a \mid s)}$ and the advantage function A(s,a), where the advantage function quantifies the relative benefit of taking action a in state s; and (2) the same product but with the probability ratio clipped to the interval $[1-\epsilon,1+\epsilon]$. Here, ϵ is a hyperparameter governing the magnitude of policy updates. This clipping mechanism effectively constrains excessive policy changes, thereby enhancing training stability.

However, its application to LLMs encounters significant challenges, including substantial computational overhead from maintaining a critic network, which increases memory usage and training time for models with billions of parameters. Additionally, training stability can be undermined by inaccurate value function estimates or suboptimal tuning of Generalized Advantage Estimation (GAE) (Schulman et al., 2016) parameters, issues that become more pronounced as LLMs scale in size. To address these limitations, Group Relative Policy Optimization (GRPO) (Shao et al., 2024a) has emerged as an efficient alternative. By eliminating the critic network, GRPO reduces computational and memory demands, estimating advantages directly from rewards of multiple rollouts to the same prompt, thus leveraging the comparative nature of reward models and offering a scalable solution for LLM training. The GRPO objective function is mathematically formulated as an averaged composite expression across multiple rollouts, incorporating policy ratio optimization and KL regularization:

$$J_{\text{GRPO}}(\theta) = \frac{1}{G} \sum_{i=1}^{G} \frac{1}{|a_i|} \sum_{t=1}^{|a_i|} \left\{ \min \left(\rho_{i,t} \hat{A}_{i,t}, \operatorname{clip} \left(\rho_{i,t}, 1 - \epsilon, 1 + \epsilon \right) \hat{A}_{i,t} \right) - \beta D_{\text{KL}} [\pi_{\theta} \parallel \pi_{\text{ref}}] \right\}$$
(5)

where $\rho_{i,t} = \frac{\pi_{\theta}(a_{i,t}|s,a_{i,< t})}{\pi_{\theta_{\text{old}}}(a_{i,t}|s,a_{i,< t})}$ denotes the probability ratio of the old and new strategies. G is

the number of rollouts per prompt, $|a_i|$ denotes the length of the i-th action sequence, $\hat{A}_{i,t}$ estimates the advantage of action $a_{i,t}$ at timestep t. The clipping is analogous to PPO, and β penalizes deviations from $\pi_{\rm ref}$ via the KL-divergence term. The objective averages across rollouts and timesteps, combining a clipped probability ratio (to stabilize updates while leveraging advantage signals) with a KL penalty to balance policy improvement against alignment with the reference policy. This dual mechanism ensures controlled optimization by restricting drastic policy shifts while maintaining coherence with prior behavior.

B.2 REWARD FUNCTION.

We remove formatting rewards (e.g., enforcing "think" tags), as the SFT model already follows the format, allowing the policy to focus on passing test cases. Given a rollout, the reward R is practiced as:

$$\mathcal{R} = \begin{cases}
-2, & \text{if no code is extracted or the code fails to compile,} \\
0, & \text{if the code compiles but passes no test cases,} \\
\frac{5.0 \times \# \text{passed}}{\# \text{total}}, & \text{otherwise.}
\end{cases} (6)$$

We adopt a continuous reward setting, as it provides denser supervision than the all-or-nothing alternative and leads to faster convergence (Wei et al., 2025; Dai et al., 2024).

B.3 TRAINING DYNAMICS.

As shown in Figure 9 and Figure 10, we present the SFT training curves (loss and token accuracy). Figure 11 and Figure 12 illustrate the RL training curves (reward and entropy).

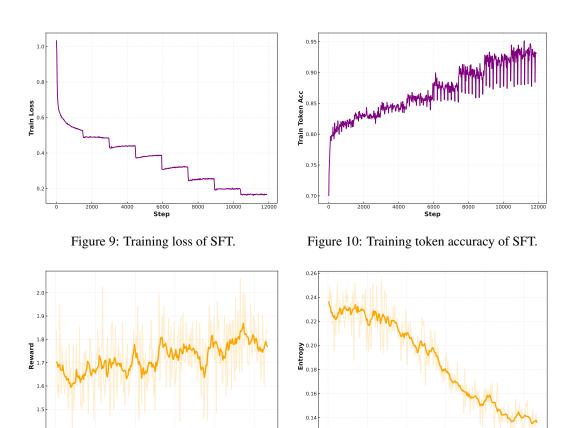


Figure 11: Training reward of RL.

Figure 12: Training entropy of RL.

B.4 TRAINING CONFIGS AND COSTS

For SFT, we use a learning rate of 5e-5 with a global batch size of 128 for 8 training epochs. For RL, the policy models are updated with a global batch size of 128 and a consistent learning rate of 7e-5, without applying the KL-divergence constraint to the starter model, and employ a rollout temperature of 1.0 with 8 rollouts to encourage exploration.

Training large reasoning models incurs significant costs compared to standard (eg. short-CoT) instruction models. In the SFT stage, the dominant overhead stems from longer sequence lengths and the need for more update epochs, which together lead to several times more compute consumption than training non-reasoning counterparts. In the RL stage, the major bottleneck lies in generating multiple rollouts for each problem used for GRPO-algorithm.

Concretely, training X-Coder on Qwen2.5-Coder-7B-Instruct required 128 H20 Enterprise (96 GB) GPUs for 220 hours during SFT, and 32 H200 (141 GB) GPUs for 7 days to complete 270 update steps during RL. We are going to make X-Coder a readily accessible, open-source model, enabling the community to benefit from its capabilities without having to bear the training costs.

B.5 A DISTRIBUTED FRAMEWORK FOR AUTOMATED CODE VERIFICATION

To provide a robust and scalable solution for code validation, we develop a distributed arbitration framework inspired by open-source repository implementations⁵. The system is based on a microservice architecture, comprising a *FastAPI*-based asynchronous API Gateway, a pool of code execution workers in the sandbox and a central *Redis* instance. *Redis* serves as a high-performance message broker and state manager, effectively decoupling the client-facing gateway from the back-

⁵https://github.com/0xWJ/code-judge.git

end computational workers. This architectural choice facilitates independent scaling, deployment, and enhances the overall resilience of the system. **Based on this evaluation framework, we implemented highly concurrent code testing during RL training.** We used batching when submitting tasks to the *Redis* server to achieve high concurrency even with low request rates. This process required the server to distribute all test tasks to different workers, utilizing the CPU power of all participating machines. Figure 13 shows the system diagram of the framework.

The framework's efficacy is derived from its strategic implementation of *Redis* data structures. Task distribution is managed by a *Sorted Set*, which functions as a time-prioritized FIFO queue; submissions are added with a timestamp score via *ZADD*, and workers atomically retrieve the next task using *BZPOPMIN*. This approach ensures ordered processing and prevents race conditions. For result transmission, each task is assigned a dedicated *List*, to which a worker pushes the outcome using *RPUSH*. The API Gateway then performs a blocking pop (*BLPOP*) on this unique list to retrieve the corresponding result efficiently. Furthermore, worker health and presence are monitored using *String* keys with a Time-To-Live (TTL). Workers periodically refresh their key's TTL as a heartbeat, enabling the system to automatically detect and de-register unresponsive nodes.

The resulting system exhibits several key advantages. The asynchronous, in-memory nature of its core components yields high throughput and low-latency performance. Its design is inherently scalable, as the stateless worker pool can be expanded horizontally to meet computational demand, while native support for *Redis* Cluster addresses data-tier bottlenecks. Finally, the framework's reliability is bolstered by the atomicity of *Redis* operations and the integrated fault-detection mechanism, ensuring dependable and consistent code verification.

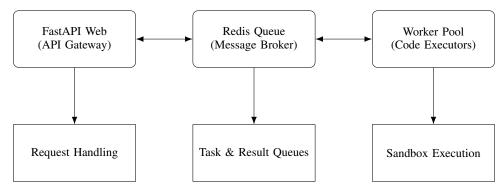


Figure 13: The distributed architecture of the code verification framework.

B.6 BASELINES

We compare the X-Coder with three categories of baselines: (1) SFT model, e.g., Bespoke-Stratos, OlympicCoder, OCR-Qwen-Instruct, OpenThinker3, Qwen3-8B, and rStar-Coder; (2) RL model, including Skywork-OR1, DeepCoder-14B-Preview, and AReal-boba²-14B; (3) SFT-then-RL model, such as AceReason1.1, Klear-Reasoner, and MiMo-RL.

C NOVEL TASK SYNTHESIS

Building on EpiCoder, which synthesizes programming tasks through feature-based combinations, we introduce three key improvements to generate more diverse and complex instructions.

First, rather than relying on broad feature definitions, we explicitly extract and evolve competition-related features from 10,000 question–solution pairs in TACO (Li et al., 2023) using GPT-4o-0513 (§C.1). Second, we adopt a two-stage process: selecting mutually consistent features and then formulating challenging, hint-free tasks (§C.2). Third, we extend the synthesis method to support multi-style generation, covering CodeForces-style tasks (rich narratives with standard I/O), LeetCode-style tasks (starter code with fixed signatures), and AtCoder-style tasks (concise specifications), thereby enhancing task diversity. In §C.3, we further estimate the difficulty of synthesized problems using a trained discriminator.

C.1 FEATURE EXTRACTION AND EVOLUTION

1026

1027 1028

1029

1030 1031

1032

1078

1079

While EpiCoder extracts general-purpose features from raw corpus, we explictly extract and evol compatetitive programming-related feature. Speficilly, we design multiple aspect of features that highly relates to competitive programming, such as data structure, algorithm, mathmatical, ect.

We improve the extraction process to guide the LLM to focus on competitive programming-related concepts, as follows:

```
1033
        Extract features from the provided problem and solution code related to algorithmic
1034
        programming, competitive programming, Leetcode, and Codeforces, following the requirements
        for each category below, formatted in JSON structure.
1035
1036
        Responses in the following categories should be concise and organized in a JSON format
        surrounded with <begin> and <end>. Categories may include nested structures if applicable.
1037
        Here is an example of the expected format:
1038
1039
        <begin>{
            "programming language": [
1040
                "Python"
1041
            "problem type": [
1042
                "graph traversal"
1043
             "algorithm": {
                 "graph algorithms":[
1045
                    "Dijkstra's algorithm",
                    "DFS",
1046
                    "BFS"
1047
                 "dynamic programming":[
1048
                    "Longest Increasing Subsequence",
                    "Knapsack Problem"
1049
                ]
1050
            "data structures": [
1051
                "array"
1052
                "linked list",
                "heap",
1053
                "segment tree"
1054
            "implementation logic":["recursive", "iterative"]
1055
        }<end>
1056
        Categories to extract:
1057
        1. Programming Language: Note the specific programming language used. Example: ["Python",
1058
        2. Problem Type: Outline the type of problem the code is solving. Example: ["graph
        traversal", "sorting", "dynamic programming"].
        3. Algorithm: Identify the specific algorithm or method being used in the code. This category
        can include the following subcategories:
1061
            3.1 Graph Algorithms: Specify graph algorithms used. Example: ["Dijkstra's algorithm",
1062
        "DFS", "BFS"].
            3.2 Sorting Algorithms: Specify sorting algorithms used. Example: ["QuickSort",
1063
        "MergeSort"].
1064
            3.3 Dynamic Programming: Specify dynamic programming techniques. Example: ["Longest
        Increasing Subsequence", "Knapsack Problem"].
1065
            3.4 Search Algorithms: Identify search algorithms used. Example: ["Binary Search",
        "Linear Search"].
            3.5 Other relevant subcategories...
1067
        4. Data Structures: Describe the primary data structures utilized. Example: ["array",
1068
        "graph", "tree", "heap"].
        5. Implementation Logic: Describe the implementation logic. Example: ["iterative",
1069
        "recursive", "bit manipulation"].
1070
        6. Complexity Analysis: Provide time and space complexity of the code if available. Example:
        ["Time Complexity: O(n log n)", "Space Complexity: O(n)"]
1071
         . Optimization Techniques: Specify any optimizations applied. Example: ["memoization",
1072
        "greedy approaches", "bitwise operations"].
         . Purpose: What the code is used to do. Example: "To find the shortest path in a graph using
        Dijkstra's algorithm."
1074
        9. Summary: Provide a concise summary. Example: "Solves the given competitive programming
1075
        problem using a depth-first search approach to traverse the graph.
        Extract as many features as possible and try not to let a feature appear in multiple
        categories at the same time.
1077
```

Then we increase the diversity and complexity through evolution along both the breadth and depth dimensions. For example, along the breadth dimension, given an extracted feature such as quicksort,

1081

1082

1083

the LLM may evolve new features like bubble sort, even if they were not originally extracted. Along the depth dimension, a concept such as prefix sum can evolve into more advanced variants like difference array or Fenwick tree, reflecting increasing levels of abstraction and difficulty. The overall evolution process is illustrated below.

```
1084
        Feature Tree Evolution Task:
1085
        You are provided with a feature tree represented as a nested JSON structure. Each node in
        this tree represents a feature or a sub-feature of competitive algorithm programming, with
        the leaves being the most specific features. Your task is to expand this feature tree both in
1087
        depth and breadth. Depth expansion means adding more specific sub-features to existing
        leaves. Breadth expnasion means adding more sibling features at the current levels.
1089
        Here are some explanations of the features:
1090
        {explanations}
1091
        The input feature tree will be provided in JSON format, and your output should be a JSON
1092
        structure that represents the expanded feature tree.
1093
        Output Format:
1094
        - Expanded Feature Tree: Provide the expanded feature tree as a JSON structure. Surround the
        json with <begin> and <end>.
1095
        Input Feature Tree Example:
1096
1097
            "algorithm": {
                "sorting": ["quick sort", "merge sort"],
1098
                "tree traversal": ["in-order traversal"]
1099
             "mathematics": [
1100
                "number theory",
1101
                "combinatorics"
            ]
1102
1103
        Expanded Feature Tree Example:
1104
        <begin>
1105
1106
            "algorithm": {
                 "sorting": {
1107
                     "quick sort": ["3-way quick sort", "dual-pivot quick sort"],
                     "merge sort": ["top-down merge sort", "bottom-up merge sort"],
1108
                    "heap sort":[]
1109
1110
                "tree traversal": {
                     "in-order traversal": ["recursive in-order traversal", "iterative in-order
1111
        traversal"],
                    "pre-order traversal":[],
1112
                    "post-order traversal":[],
1113
                    "level-order traversal":[],
1114
                }
1115
             "mathematics": {
                 "number theory": [
1116
                    "prime factorization",
1117
                     "greatest common divisor".
                     "power modular reduction"
1118
1119
                 "combinatorics": [
                     "Pascals triangle",
1120
                     "permutations and combinations",
1121
                     "binomial coefficients"
1122
1123
        <end>
1124
1125
1126
        Constraints:
        1. For breadth expansion, add at least 2 new sibling features to each existing node.
1127
        2. For deep expansion, you need to add new sub-features to it, provided that you think the
1128
        current leaf node has a more fine-grained feature.
        3. Focus on generating new and innovative features that are not present in the provided
1129
        4. The features are related to competitive algorithm programming.
1130
        Please follow the above constraints and expand the feature tree accordingly.
1131
1132
        Input:
        {features}
1133
        Output:
```

1135 1136

1137

11381139

1140 1141

1142

1143

1144 1145

1146

1147 1148

1149

<begin>expanded feature tree<end>

After evolution, we merge features that share common traits into a larger tree, providing a rich pool of features for subsequent task formulation.

C.2 STYLIZED TASK GENERATION FOR COMPETITIVE PROGRAMMING

We design a prompt template to systematically transform extracted features into stylized competitive programming tasks.

Input: a sampled feature tree represented in JSON format.

Output: a feature-role tree (JSON), where each node is assigned roles such as core technique, subroutine, or constraint, together with an integration strategy (string) that explains how to combine these features into a coherent problem.

To improve instruction-following and task understanding, the template is equipped with a one-shot example that demonstrates how raw features are mapped into roles and integrated into a task.

```
1150
1151
        Stage 1 Prompt Template for Feature Selection
1152
1153
        STAGE1_PROMPT_TEMPLATE = """You are a professional competitive programming problem setter.
1154
1155
        Your task consists of three parts:
1156
1157
        Step 1: Tree-Structured Feature Role Explanation
1158
        Recursively traverse the provided feature tree.
1159
        - For each leaf node, annotate it with a "potential_use" field describing how this feature is
        typically used in competitive programming problems (e.g., input modeling, optimization,
1160
        search, handling edge cases, etc.).
1161
        - Internal nodes retain their structure for hierarchy.
1162
        Output the annotated tree in the same structure, with every leaf node containing its
1163
        "potential use".
1164
1165
        Step 2: Subtree Selection for Problem Integration
1166
1167
        Based on your role analysis, select a subtree (tree-structured subset) where all selected
        leaf features can be naturally integrated into a single, high-quality competitive programming
1168
        problem.
1169
        - Only include features that contribute meaningfully to the same problem idea.
1170
        - Internal nodes are included only if they have selected children.
1171
        - For each selected leaf, include only its "feature" name and "potential_use".
1172
1173
1174
        Step 3: Integration Strategy
1175
        Briefly describe ("integration_strategy") how the selected features can be integrated
        together in a single problem, focusing on how their combination enables a meaningful and
1176
        challenging algorithmic scenario.
1177
1178
1179
        **Output Format: **
1180
        Return a JSON object **with exactly this structure** (an example):
1181
1182
          "feature_roles_tree": {{
1183
            "algorithm": {{
              "search algorithm": {{
1184
                "binary search": {{
1185
                  "recursive binary search": {{
1186
                    "potential_use": "Used for divide-and-conquer searching in sorted structures or
        answer spaces.'
1187
                  "iterative binary search": {{
```

```
1188
                     "potential_use": "Efficient loop-based implementation for finding bounds or
1189
        specific elements."
                  } }
1190
                 }},
1191
                 "breadth-first search (BFS)": {{
                  "level-order BFS": {{
1192
                    "potential_use": "Traverses graphs layer by layer; useful for shortest path or
1193
        component discovery."
                  }}
1194
                } }
1195
              } }
1196
            }},
             data structures": {{
1197
               "bitmap": {{
                 "bit manipulation": {{
1198
                  "bitwise AND": {{
1199
                     "potential_use": "Filters or checks properties using bitmasks."
                  }},
1200
                   "bitwise OR": {{
1201
                     "potential_use": "Combines flags or sets with bitwise aggregation."
1202
                  } }
                } }
1203
              } }
            } }
1204
          }},
1205
          "selected_features_tree": {{
1206
            "algorithm": {{
1207
               "search algorithm": {{
                 "binary search": {{
1208
                   "recursive binary search": {{
1209
                    "feature": "recursive binary search",
                    "potential_use": "Used for divide-and-conquer searching in sorted structures or
1210
        answer spaces."
1211
                  } }
1212
              } }
1213
            } } ,
             "data structures": {{
1214
              "bitmap": {{
1215
                "bit manipulation": {{
                  "bitwise AND": {{
1216
                    "feature": "bitwise AND",
1217
                     "potential_use": "Filters or checks properties using bitmasks."
                  } }
1218
                } }
1219
              } }
            } }
1220
          }},
1221
          "integration_strategy": "The problem will require recursive binary search to efficiently
1222
        search over a sorted value space, while bitwise AND operations will be used to filter
1223
        candidate solutions according to constraints. Their combination allows for a problem that
        involves searching over sets and optimizing bitwise criteria."
1224
1225
1226
1227
        **Available Features (Tree):**
1228
        {features ison}
1229
1230
        Instructions:
1231
        - Always preserve the tree structure in "feature_roles_tree" and "selected_features_tree".
        - In selected_features_tree, only include "feature" and "potential_use" fields for leaf nodes.
1232
        - "integration_strategy" should make clear how/why these features form a coherent, advanced
1233
        problem.
         - Do not be overly conservative; it is often possible to design advanced problems where many
1234
        features interact in non-trivial ways. Challenge yourself to maximize feature use without
1235
        sacrificing problem quality.
1236
```

C.2.1 COMPATIBALE FEATURE SELECTION

We present a case to examine how model selects compatibale features and combine them.

Given a sampled feature tree:

12371238

1239 1240

1241

1269

1270

1271

```
1242
1243
         "input_features":
             "algorithms":
1244
               graph_algorithms": {
                 "shortest_path": [
                   "Dijkstra's algorithm",
1246
                   "Floyd-Warshall"
1247
                 "network flow": [
1248
                   "Ford-Fulkerson",
1249
                   "Edmonds-Karp"
1250
1251
               "string_algorithms": {
                 "pattern_matching": [
1252
                   "KMP algorithm",
1253
                   "Boyer-Moore"
1254
1255
             "data_structures": {
1256
               "tree_structures": [
1257
                  'segment tree",
                 "fenwick tree"
1259
               "hash_structures": [
                 "rolling hash",
1260
                 "cuckoo hashing"
1261
1262
             optimization_techniques": {
1263
               "dynamic_programming": [
                 "interval DP",
1264
                 "tree DP"
1265
1266
1267
```

LLM pairs each feature with *potentially usage* to obtain feature tree with role annotation. For example, LLM will anonotes feature "rolling hash" as "Compute hash values for sliding windows in constant time". These annotations help LLM to aggregate these features based on their potentially usage. For above given feature tree, the feature tree with potential usage looks like:

```
1272
        "feature_roles_tree": {
1273
             "algorithms": {
               'graph_algorithms": {
1274
                 "shortest_path": {
1275
                  "Dijkstra's_algorithm": {
                     "potential_use": "Find single-source shortest paths in weighted graphs with
1276
        non-negative edges"
1277
                   "Floyd_Warshall": {
1278
                     "potential_use": "Compute all-pairs shortest paths with O(n^3) complexity"
1279
                  }
1280
                 "network flow": {
1281
                   "Ford Fulkerson": {
                     "potential_use": "Find maximum flow using augmenting paths with DFS"
1282
1283
                   "Edmonds_Karp": {
                     "potential_use": "Guarantee polynomial time maximum flow using BFS"
1284
1285
1286
               string_algorithms": {
1287
                 "pattern_matching": {
                  "KMP_algorithm": {
1288
        "potential_use": "Linear time pattern matching with failure function preprocessing"
1289
1290
                   "Boyer_Moore": {
1291
                     "potential_use": "Efficient pattern search using bad character and good suffix
1292
        heuristics"
1293
1294
1295
             "data_structures": {
              "tree_structures": {
```

1317

1318

1319

1320

1321

```
1296
                 "segment_tree": {
1297
                   "potential_use": "Support range queries and updates in O(log n) time"
1298
                 "fenwick_tree": {
1299
                   "potential_use": "Efficiently compute prefix sums with point updates"
1300
1301
               "hash_structures": {
                 "rolling hash": {
1302
                   "potential_use": "Compute hash values for sliding windows in constant time"
1303
                 "cuckoo_hashing": {
1304
                   "potential_use": "Achieve worst-case O(1) lookup with multiple hash functions"
1305
              }
1306
1307
             optimization techniques": {
               'dynamic_programming": {
1308
                 "interval DP": {
1309
                   "potential_use": "Solve optimization problems on contiguous ranges"
1310
                 "tree DP": H
1311
                   "potential_use": "Compute optimal solutions on tree structures bottom-up"
1312
1314
1315
```

LLM then selects a compatible and consistent subtree that can formulate a self-contained compatetiting programming problem. For example, LLM keeps the features that can be aggregated into selected feature tree, and concluding with an intergration strategy, which displays how to combine these features into a unified problem.

For example, in this case, LLM selects "Dijkstra's algorithm", "Edmonds-Karp", "segment tree", and "tree DP", and aims to formulate a problem around "dynamic network optimization".

```
1322
         "selected_features_tree": {
1323
             "algorithms": {
                'graph_algorithms": {
1324
                  "shortest_path": {
                    "Dijkstra's_algorithm": {
   "feature": "Dijkstra's algorithm",
   "potential_use": "Primary pathfinding algorithm"
1325
1326
                   }
1327
                  "network flow": {
                    "Edmonds_Karp": {
1329
                      "feature": "Edmonds-Karp",
                      "potential_use": "Flow computation with guaranteed complexity"
1331
1332
               }
1333
              "data structures": {
1334
               "tree_structures": {
                  "segment_tree": {
1335
                   "feature": "segment tree",
1336
                   "potential_use": "Maintain dynamic edge weights or capacities"
1337
               }
1338
1339
              optimization_techniques": {
                dynamic_programming": {
1340
                  "tree_DP": {
1341
                   "feature": "tree DP",
                   "potential_use": "Optimize subproblems on network tree decomposition"
1342
1343
1344
         "integration_strategy": "Create a dynamic network optimization problem where Dijkstra's
         algorithm finds shortest paths that are used as augmenting paths in a modified Edmonds-Karp
1346
         flow algorithm. Use segment tree to handle dynamic updates to edge capacities based on flow
         history. Apply tree DP on the shortest path tree to compute optimal flow distributions. This
1347
         models a transportation network with time-varying capacities.
1348
```

C.2.2 FROM FEATURE TO STYLIZED TASK

1350

1351 1352

1353

1354 1355

1356

1392 1393

1394 1395 1396

1400

1401

1402

1403

We separate feature selection from task generation, as our initial attempts showed that prompting an LLM to perform both within a single prompt often led it to choose fewer features and produce overly simple problems.

During task generation, LLM recieves *selected features tree* and its *integration strategy* to formulate styleized task based on prompt recieved. Task generation prompt for Codeforces-style is as follows:

```
1357
1358
        """You are a professional competitive programming problem setter.
1359
        You have been provided with:
1360
         selected_features_tree: a tree structure where each leaf contains a "feature" name and its
1361
        "potential_use".
1362
         integration strategy: a strategy describing how these features should be integrated into a
        single, high-quality problem.
1363
1364
        Your task is to **generate a complete Codeforces-style problem statement** that fully
        integrates ALL selected features.
1365
        Requirements:
         The story and setting must naturally motivate every selected feature, making each
1367
        indispensable for an optimal solution.
1368
        - Specify precise input/output format and tight constraints.
        - Provide at least two distinct, non-trivial sample Input/Output pairs, each with a clear
1369
        explanation.
1370
        - Make sure the samples are consistent with your constraints and the solution requires use of
        all selected features.
1371
        - Do not include any references to algorithms, data structures, solution strategies, or any
1372
        implicit or explicit hints in any part of the statement, notes, or examples. Do not include
        any motivational, summary, or instructional phrases (e.g., "Remember", etc.) at any point in
1373
        the output. The statement must end after the final example or clarification, with no
        extraneous commentary.
1374
        - Output should be a **single JSON object** with the field "question" only.
1375
1376
        **Output Format (strictly):**
1377
1378
          "question": "# Problem Title\\n\\nStory/context (describe the scenario)\\n\\n##
        1379
        Example\\n### Input\\n<code block with sample input>\\n### Output\\n<code block with sample
1380
        \operatorname{output} \operatorname{\operatorname{Note}} \operatorname{\operatorname{InExplanation}} about the sample(s), but without any solution hints.
1381
1382
        **Inputs:**
1384
         selected_features_tree (JSON):
        {selected_features_info}
1385
1386
        - integration strategy (string):
        {integration_strategy}
1387
1388
        Instructions:
1389
        - You must ensure every selected feature is essential and naturally integrated.
        - Output ONLY the required JSON object, no extra text.
1390
1391
```

In this instance, our generated Codeforces problem is shown in Figure 14, while the generated AtCoder and LeetCode problems are presented in Figures 15 and 16, respectively.

C.3 TASK DIFFICULTY ESTIMATES

Judging the difficulty of a synthetic task is challenging. To better capture the difficulty distribution of tasks generated by X-Coder, we adopt a classifier-based approach. Specifically, we add a special classification token to Qwen2.5-Coder-14B-Instruct and fine-tune it to predict the Codeforces rating of 6,246 tasks from the CodeContests dataset with annotated ratings, reserving 5% as a validation set. The fine-tuned model achieves 84% classification accuracy on the validation set. We then use this model to estimate the difficulty of 1,000 tasks generated by our pipeline, obtaining a holistic distribution as shown in Table 6.

1424

1455 1456 1457

```
1404
                                                                              Dynamic Transport Renewa
                                                                            In the city of Codeland the transportation system is in constant flux. The city has n intersections and m one-way roads. Each road is characterized by a treat line and an initial capacity representing the maximum number of vehicles that may traverse that road in a day. Due to changing conditions, city engineers periodically adjust road capacities. After every such update, the transport authority recalculates their performance metric in two steps.
                                                                                                                                                                                                                                                                                                                                       Each of the next Q lines contains three integers L, R and X (1 \leq L \leq R \leq m, -10.99 \leq X \leq 10.99 meaning that for every road with index in [L, R] you must add X to its current capacity (if a road's capacity becomes negative, treat it as 0).
1405
1406
                                                                                                                                                                                                                                                                                                                                         For each query, output a single integer — the sum of the maximum flow from intersection 1 to n
1407
                                                                                                                                                                                                                                                                                                                                        For each query, output a single integer. The soft of the fraction into morn interessourch to 
(computed by repeatedly sending flow along the fastest (i.e. minimum travel time) augmenting 
path) and the maximum reward obtainable from the shortest path tree from the last successful 
search (computed using the tree-optimization described above).
                                                                             First, they compute the maximum number of vehicles that can be sent from the central depot at
                                                                           has, they compose we inflammar manness or termined an attack and set and copy as intersection 1 to the distribution center at intersection n. To do so they repeatedly select an augmenting path that minimizes the total travel time (using a shortest path computation) among all paths on which every road has positive capacity. They send as many vehicles along that path as allowed by its weakest road and then reduce the capacity of every road on the path by that amount. This process is repeated until no valid path from 1 or remains.
1408
1409
1410
                                                                             Second, using the predecessor structure recorded in the last successful shortest path search
                                                                           (forming a tree rooted at 1), the authority assigns each intersection a reward equal to its travel time from intersection 1 (as computed in that search). They then choose a subset of intersections from this tree such that no intersection and its direct predecessor are both chosen, with the goal to maximize the total reward. (This selection is computed using an optimization on the tree structure).
1411
1412
1413
                                                                             The final performance metric is the sum of the maximum flow (i.e. total number of vehicles sent)
                                                                             and the maximum total reward from the tree selection.
                                                                                                                                                                                                                                                                                                                                         Output
                                                                            Your task is to process a series of capacity update queries. Initially the network is given. Then each query specifies an interval [L, R] (referring to the roads in their input order) and an integer X. For every road whose index is in [L, R], add X to its current capacity. If an update causes are dege's capacity to become negative, sel it to 0. After each update, recalculate the maximum flow using the method described above and then compute the optimal reward from the latest
1414
1415
                                                                                                                                                                                                                                                                                                                                       Note After the first update, the capacities of roads with indices 2, 3 and 4 decrease by 1, so they become 2, 3 and 1 respectively while the others remain unchanged. The flow computation proceeds in iterations by first choosing the path 1–2–4 (with travel time 1+3–4) and sending 2 vehicles, then using the path 1–3–2–4 (with travel time 2+1+2–5) to send 1 vehicle, and finally again 1–2–4 to send 3 vehicles. The total maximum flow is 6. In the last successful shortest path search (from the iteration yielding the 3 vehicles). The predecessor tree has intersection 1 as the root with children 2 and 3, and intersection 2 with child 4. With rewards equal to their computed distances from intersection 1, an optimal non-adjacent selection yields a total reward of 5. Their sum is 11.
1416
                                                                             shortest path tree. Output the sum of these two values
1417
                                                                             Note that each update is cumulative
1418
                                                                             The first line contains three integers n, m and Q (2 \le n \le 100, 1 \le m \le 1000, 1 \le Q \le 1000) — the number of intersections, the number of roads and the number of queries.
 1419
                                                                            Each of the next m lines describes a road with four integers u, v, t and c (1 \le u, v \le n, u \ne v, 0 \le t \le 10°9), meaning that there is a road from intersection u to v with travel time t and initial capacity c. The roads are numbered from 1 to m in the order of appearance.
                                                                                                                                                                                                                                                                                                                                         After the second update, the capacities of roads with indices 1, 2 and 3 increase by 2
1420
                                                                                                                                                                                                                                                                                                                                         Recomputing the maximum flow now yields a value of 2, while the corresponding shortest path tree results in an optimal reward of 6. The final performance metric is 8.
1421
```

Figure 14: Case for Codeforces-style Problem, featuring rich, imaginable narrative contexts.

```
1425
                                                      Dynamic Transportation Optimization
1426
                                                       You are given a directed transportation network with N nodes and M roads. Each road i (1-indexed) goes
1427
                                                       from node u to node v, requires t units of time to traverse, and can transport at most c units of goods
                                                      When a shipment is made from a source s to a target t, the following process is repeated:
1428
                                                               Find a route from s to t that minimizes the total travel time among all routes that have a positive
                                                                capacity on every road used. (If more than one route achieves the minimum travel time, any one of them is chosen.)
1429
                                                               Let f be the minimum capacity among the roads on the chosen route. Send f units along the route and reduce the capacity of every road on that route by f.
1430
1431
                                                               The process stops when there is no route from s to t with all roads having positive capacity. The total goods shipped is the sum of all f sent during the process.
1432
                                                      You are given Q operations. Each operation is in one of the following two forms
1433

    1 i x: Update the capacity of road i to x.

1434
                                                              2 s t: On the current network, simulate the above process from s to t and output the total goods
                                                                shipped. Note that the simulation is performed on a copy of the current network so that the road
1435
                                                                capacities remain unchanged for subsequent operations
                                                      Output the answer for each query operation.
1437
                                                      The first line contains three integers N. M. Q. Then M lines follow. The i-th of these lines contains four
                                                      integers u, v, t, c describing road i. Then Q lines follow
1439
                                                      Each of these lines is either in the form 1 i x or 2 s t as described above
1441
1442
                                                      For each operation of the form 2 s t, output a single integer representing the total goods shipped.
1443
                                                      Constraints
                                                      2 \leq N \leq 200; \ 1 \leq M \leq 500; \ 1 \leq Q \leq 200; \ 1 \leq u, \ v, \ s, \ t \leq N, \ u \neq v; \ 1 \leq t \leq 10^3; \ 1 \leq c, \ x \leq 10^9
1444
                                                      Sample Input 1
1445
                                                      333
1446
                                                      1 2 5 10
1447
                                                      2 3 5 10
                                                      1 3 11 5
1448
                                                      213
1449
                                                      1 3 15
1450
1451
                                                      Sample Output 1
                                                      15
1452
                                                      25
1454
```

Figure 15: Case for AtCoder-style Problem, featuring concise, minimal explainations.

1458 Dynamic Transportation Network 1459 Given a directed network with n nodes labeled from 1 to n and m edges, each 1460 edge is represented as a quadruple [u, v, capacity, travelTime] and denotes a 1461 directed connection from node u to node v with the given capacity and travel time. The network is dynamic: in each round you select a route from node 1 to 1462 node n with the smallest total travel time among all routes with positive 1463 capacities. If there are multiple routes with the same total travel time, choose the route that can carry the largest amount of flow (where the flow of a route is the 1464 minimum capacity among its edges). Send flow along the selected route equal 1465 to this value and reduce the capacity of every edge on the route by the sent flow Repeat the process until no valid route exists. 1466 After the rounds finish, for every node i $(1 \le i \le n)$ determine the total amount of 1467 flow that reached it. A node receives flow from a selected route if it appears on 1468 that route and the flow travels from node 1 to that node along the route. Return an array f of length n where f[i - 1] is the total flow that reached node i from node 1469 1470 Signature 1471 class Solution: 1472 def dynamicTransportationNetwork(self, n: int, m: int, edges: List[List[int]]) -> List[int]: 1474 Example 1 1475 Input: n = 4, m = 5, edges = [[1,2,4,2], [1,3,3,1], [2,4,3,3], [3,2,2,1], [3,4,4,5]] 1476 Output: [6,3,3,6] Example 2 1478 Input: n = 3, m = 3, edges = [[1,2,5,2], [2,3,4,3], [1,3,2,10]] 1479 Output: [6,4,6] 1480 Constraints 1481 • $2 \le n \le 10^4$ 1482 $1 \le m \le 5 * 10^4$ 1483 For each edge in edges: 1 ≤ u, v ≤ n and u ≠ v 1484 o 1 ≤ capacity ≤ 10^4 1485 o 1 ≤ travelTime ≤ 10^4 1486

Figure 16: Case for LeetCode-style Problem, featuring predefined function signatures.

Table 6: Difficulty distribution of Codeforces-style ratings. "Original" denotes the annotated distribution from CodeContests, and "Test" denotes 1,000 tasks generated by our pipeline.

CF Rating	Original	Test (Ours)	Original Share	Test Share
1200	623	0	10.0%	0.0%
1400	727	0	11.7%	0.0%
1600	889	0	14.3%	0.0%
1800	840	16	13.5%	1.6%
2000	797	2	12.8%	0.2%
2200	697	47	11.2%	4.7%
2400	665	585	10.7%	58.5%
2600	484	319	7.8%	31.9%
2800	312	12	5.0%	1.2%
3000	233	15	3.7%	1.5%
3200	157	4	2.5%	0.4%
3400	122	0	2.0%	0.0%
Total	6,246	1,000	100%	100%

D SOLUTION GENERATION AND QUALITY ASSURANCE

D.1 VALIDATION ON SOLUTION

1487 1488 1489

1490 1491

1507

1509

1510

1511

For tasks with descriptions shorter than 200 tokens, we discard them, as such descriptions are often either too trivial or incomplete. For each generated solution, we ensure quality by (i) removing sam-

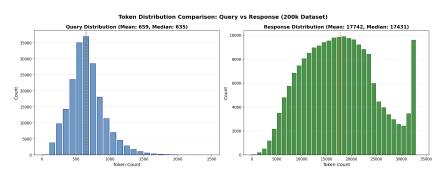


Figure 17: Dataset statistics of the demonstration dataset.

ples without complete think and answer tags, (ii) rejecting cases where the extracted Python block fails AST validation, (iii) excluding solutions that contain multiple code blocks after the reasoning process, as they hinder reliable solution extraction, and (iv) filtering out samples exceeding 25k tokens to prevent overthinking and to reduce SFT cost caused by sequence padding.

D.2 SFT DATASET STATISTICS

The overall token length distribution, shown in Table 7, and Figure 17, primarily follows a normal distribution, with a median of 16k.

Table 7: Token statistics for tasks and solutions of the demonstration dataset.

Туре	Min	Max	Mean	Median	Std Dev	Total Tokens
Task Solution	200 1,711	3,537 33,144	658.91 17,742.50	635.00 17,431.00	258.49 7,295.92	134.3M 3.25B
Dataset Size			200,091 entries			3.38B

E TEST CASE GENERATION

E.1 PROMPTING-BASED TEST GENERATION

```
1547
1548
        You are a professional test case generation expert, skilled at designing comprehensive test
        cases for programming problems.
1549
        Please generate 15 different test cases for the following programming problem, including edge
1550
        cases, small-scale, medium-scale, and large-scale test data.
1551
        Problem:
1552
        {problem_statement}
1553
        Requirements:
1554
        1. Generate 15 test cases
        2. Include edge cases (empty input, minimum values, maximum values, etc.)
1555
        3. Include different scales of data (small, medium, large)
1556
        4. Each test case should have clear input data
        5. Ensure test cases can thoroughly validate the correctness of solutions
1557
1558
        Please return in JSON format as follows:
1559
            "test_cases": [
1560
                { {
                    "idx": 0,
1561
                     "description": "Test case description",
1562
                     "input_string": "Input data"
                }},
1563
1564
        } }
1565
```

E.2 TOOL-BASED TEST GENERATION

1566

1567 1568

1569

1570

1571

1572

1573

1574

1575

1576

The tool-based test generation strategy relies on **CYaRon**, an open-source Python library aimed at rapidly generating random data for Informatics Olympiad problems (or problems of equivalent difficulty). This library contains a variety of common data structures (e.g., graphs, trees, polygons, vectors, strings, and sequences), along with mathematics-related functions and the necessary input/output interfaces. When prompting the Teacher model to utilize the CYaRon tool, we provide its detailed documentation and usage instructions as part of the prompt. Additionally, we encourage the model to generate more boundary tests and large-scale random use cases. To ensure the sufficiency of test cases, we mandate the use of this library in conjunction with its random features and set a seed to ensure reproducibility. The detailed prompt used is illustrated as:

```
1577
        Please write a test case generator that meets the following requirements based on the
1578
        following CYaRon documentation:
1579
        1. Write a canonical CYaRon Generator using Python
1580
          Generate a single, executable Python program that can produce test cases with at least 5
        different features
1581
        3. The Python program should save each test case individually in the format [use case
        characteristics].in
1582
        4. The program should include a variety of test case types such as base cases, boundary
1583
        cases, large random cases, etc
        5. The Python program code should contain clear comments to explain the design intent for
1584
        each test case generation
1585
        6. The .in output files should contain ONLY pure input data without any comments,
        explanations, or answer validation
        7. The Python program should be able to generate all test cases in a single run when executed
1587
        8. The program should use argparse to provide configurable random seed control:
           parser.add_argument('--seed', type=int, default=42, help='Random seed for reproducibility')
1588
        9. All random number generation must use Python's built-in random module (import random) - do
1589
        not use any external random libraries or the random functions from CYaRon
1590
1591
        ### CYaRon Documentation
1592
        Input/Output (IO)
1593
        The IO library helps you easily create test data files.
1594
        Constructor Options:
1595
          ''pvthon
1596
        # Basic file specification
        IO("test1.in", "test1.out") # Explicit input/output files
1597
        IO(file_prefix="test") # Generates test.in and test.out
        IO(file_prefix="test", data_id=3) # Generates test3.in and test3.out
1598
1599
        # Advanced file naming
        IO(file_prefix="test", data_id=6,
1600
           input_suffix=".input", output_suffix=".answer") # test6.input and test6.answer
        # Partial output options
1602
        IO("test2.in") # Only input file, output goes to temporary file
1603
        IO(file_prefix="test", data_id=5, disable_output=True) # No output file generated
        IO() # Both files temporary (for use with comparator)
1604
1605
        Note: Combine 'file_prefix' and 'data_id' with loops for batch generation.
1606
1607
        IO Methods:
          ''python
1608
        io = TO("test1.in", "test1.out") # Initialize TO object
1609
        # Input writing methods
1610
        io.input_write(1, 2, 3) # Writes "1 2 3" to input file (no newline)
        io.input_write(n(4, 5, 6) # Writes "4 5 6\n" to input file
io.input_write([1, 2, 3]) # Writes list as space-separated "1 2 3"
1611
1612
        io.input_write(1, 2, 3, separator=',') # Writes "1,2,3," (note: current version leaves
1613
        trailing comma)
1614
        # Output writing methods
1615
        io.output_write(1, 2, 3) # Writes "1 2 3" to output file
        io.output_writeln(4, 5, 6) # Writes "4 5 6\n" to output file
1616
        io.output_write(1, 2, [1, 2, 3], [4]) # Flattens nested lists to "1 2 1 2 3 4"
1617
1618
        # Program execution
        io.output_gen("~/Documents/std")  # Runs program with input, captures stdout as output
        io.output_gen("C:\\Users\\Aqours\\std.exe") # Windows path support
1619
```

```
1620
1621
1622
               Graph Generation
1623
               The Graph library generates various graph structures.
1624
               Manual Construction:
1625
                '''python
                # Graph initialization
1626
               graph = Graph(10) # 10-node undirected graph (nodes 1-10)
1627
               graph = Graph(10, directed=True) # Directed version
1628
                # Adding edges
1629
               graph.add_edge(1, 5) # Default weight=1
               graph.add_edge(1, 6, weight=3) # Custom weight
1630
1631
                # Edge access and properties
                graph.edges # Adjacency list containing Edge objects
1632
                for edge in graph.iterate_edges():
1633
                      edge.start # Source node edge.end # Target node
1634
                       edge.weight # Edge weight
1635
1636
                # Output formatting options
                io.input_writeln(graph)  # Default "u v w" per line
1637
                io.input_writeln(graph.to_str(shuffle=True))  # Random edge order
                io.input_writeln(graph.to_str(output=Edge.unweighted_edge))  # "u v" format
1638
1639
1640
               Template Graphs:
                  `'python
1641
                # Basic graph templates
               \label{eq:graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.graph.
1642
               Graph.graph(n, m, directed=True, weight_limit=(5, 300)) # Directed with weight range
1643
               Graph.graph(n, m, self_loop=False, repeated_edges=False) # No duplicate edges
1644
                # Special graph types
1645
               Graph.chain(n) \# n-node chain (alias for tree(n, 1, 0))
                Graph.flower(n) # n-node star graph (alias for tree(n, 0, 1))
1646
                Graph.tree(n) # Random tree
1647
                Graph.tree(n, 0.4, 0.35) # 40% chain-like, 35% star-like, 25% random
               Graph.binary_tree(n) # Random binary tree
1648
1649
                # Competition-specific graphs
               Graph.hack_spfa(n) # Graph that breaks SPFA (1.5n edges)
1650
                Graph.hack_spfa(n, extra_edge=m) # With additional edges
1651
               Graph DAG(n, m) # Directed Acyclic Graph
Graph.UDAG(n, m) # Undirected Connected Graph
1652
1653
               Note: Most templates support 'weight_limit', 'weight_gen', 'self_loop', and 'repeated_edges'
1654
               parameters.
1655
1656
1657
                Polygon
               Generate and analyze polygons.
1658
1659
                # Polygon creation (points must be ordered)
1660
               p = Polygon([(0,0), (0,4), (4,4), (4,0)]) # Rectangle
1661
                # Geometric properties
1662
               p.perimeter() # Calculates perimeter
1663
               p.area() # Calculates area
1664
                # Generation templates
1665
                Polygon.convex_hull(n) # n-point convex hull
                Polygon.simple_polygon(n)  # Simple polygon (non-intersecting)
1666
1667
1668
1669
                Vector
1670
               Generate unique vectors/number sequences.
1671
                '''python
1672
                # Basic usage
               Vector.random() \# Default: 5 unique numbers in [0,10]
1673
               Vector.random(10, [(10,50)]) # 10 unique numbers in [10,50]
Vector.random(30, [(10,50), 20]) # 30 unique 2D vectors
```

```
1674
1675
          # Modes:
          # 0: Unique integer vectors (default)
1676
          # 1: Non-unique integer vectors
1677
          # 2: Real-valued vectors
         Vector.random(30, [(1,10), (1,10)], (2) \# 30 3D real vectors Vector.random(30, [10], 1) \# 30 numbers (may repeat)
1678
1679
1680
1681
         String
1682
         Generate random text elements.
1683
          '''python
1684
          # Basic strings
1685
          String.random(5) # 5-character word
         String.random((10,20), charset="abcd1234") # Variable length
1686
         String.random(10, charset="########"...") # 70% '#', 30% '.
1687
          # Structured text
1688
         String.random_sentence(5) # 5-word sentence
         String.random_paragraph((3,10)) # 3-10 sentence paragraph
1689
1690
          # Custom formatting
1691
         String.random_sentence(5, word_separators=[" "])  # Double space separator
1692
1693
         Note: All templates support charset customization.
1694
1695
          Sequence
1696
         Generate number sequences via recurrence.
1697
          ```python
1698
 # Explicit formula
1699
 Sequence (lambda i, f: 2*i+1) # f(i) = 2i + 1
1700
 # Recursive definition
1701
 Sequence (lambda i, f: f(i-1)+1, [0,1]) # f(i)=f(i-1)+1 with f(0)=0, f(1)=1 Sequence (lambda i, f: f(i-1)+1, {100:101, 102:103}) # Sparse base cases
1702
1703
 seq = Sequence(lambda i, f: f(i-1)+2, [0,2,4])
1704
 seq.get(3) # Returns 6
1705
 seq.get(4,6) # Returns [8,10,12]
1706
1707
 Important: Recursive definitions require base cases.
1708
1709
 Utilities
1710
1711
 Conversion:
 '''python
1712
 ati([0, 5, 100, 1E3, 1E5]) # Converts scientific notation to integers
1713
1714
 Random Numbers:
1715
 '''python
 randint(1,5) # Integer in [1,5]
1716
 uniform(1,5) # Float in [1,5]
 choice([1,2,3]) # Random selection random() # Float in [0,1)
1717
1718
1719
 Constants:
1720
 '''python
1721
 PI # 3.1415926...
 PI # 3.141520...

2.7182818...

ALPHABET_SMALL # "abcdefghijklmnopqrstuvwxyz"

ALPHABET_CAPITAL # "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

ALPHABET # Combined letters
1722
1723
1724
 NUMBERS # "0123456789"
1725
1726
 ### Code Question
1727
 {OUESTION}
```

# F DUAL-VERIFICATION

We summarize the symbols used in the dual-verification process in Table 8, and outline the corresponding procedure in Algorithm 1.

Table 8: Notation for SynthSmith Framework.

${x_i}_{i=1}^n $ ${A^j}_{i=1}^m$	Test inputs for a task $q$ Candidate solutions (LLM-generated)
$y_i^j$	Output of $A^j$ on input $x_i$
$\hat{y}_i$	Provisional label via majority vote on $\{y_i^j\}_{j=1}^m$
$w_i$	Difficulty weight for $x_i$
$\mathcal{T}_{candidate}$	Provisional labeled set $\{(x_i, \hat{y}_i, w_i)\}$
$\mathcal{T}_{golden}$	Weighted suite for selecting the solution
$ar{\mathcal{T}_{val}}$	Hold-out validation set
$S_j$ $A_{golden}$	Weighted score of $A^j$ on $\mathcal{T}_{golden}$
$A_{golden}$	Final selected "golden" solution

## Algorithm 1: Dual-Verification of Solutions and Test Cases

```
Input: Task q; test inputs \{x_i\}_{i=1}^n; candidate solutions \{A^j\}_{j=1}^m. Output: Golden solution A_{\text{golden}}; golden test suite \mathcal{T}_{\text{golden}}.
```

# Step 1: Consensus Voting & Weighting

```
\begin{array}{l} \text{for } i \leftarrow 1 \text{ to } n \text{ do} \\ & \text{for } j \leftarrow 1 \text{ to } m \text{ do} \\ & & \quad \ \ \, \big[\quad \text{Run } y_i^j \leftarrow A^j(x_i) \\ & \quad \ \, \hat{y}_i \leftarrow \arg \max_y \sum_{j=1}^m \mathbb{I}(y_i^j = y) \\ & \quad \ \, w_i \leftarrow \text{Weight}(x_i) \end{array}
```

 $\mathcal{T}_{\text{candidate}} \leftarrow \{(x_i, \hat{y}_i, w_i)\}_{i=1}^n$ 

## Step 2: Split Candidate Set

Randomly partition  $\mathcal{T}_{\rm candidate}$  into  $\mathcal{T}_{\rm golden}$  and  $\mathcal{T}_{\rm val}$ 

### **Step 3: Weighted Selection**

```
for j \leftarrow 1 to m do

\begin{bmatrix}
S_j \leftarrow \sum_{(x_i, \hat{y}_i, w_i) \in \mathcal{T}_{golden}} w_i \cdot \mathbb{I}(A^j(x_i) = \hat{y}_i) \\
j^* \leftarrow \arg \max_j S_j \\
A'_{golden} \leftarrow A^{j^*}
\end{bmatrix}
```

## **Step 4: Hold-out Confirmation**

Compute unweighted accuracies of all  $A^j$  on  $\mathcal{T}_{val}$  $j^{\dagger} \leftarrow \arg \max_{j} \operatorname{Acc}(A^j, \mathcal{T}_{val})$ 

return  $A_{\mathrm{golden}}, \mathcal{T}_{\mathrm{golden}}$ 

### G DATA LEAKAGE ANALYSIS

X-Coder, trained entirely on fully synthetic data, exhibits a lower risk of data leakage. As shown in Table 9, models trained on real-world data, like Qwen3 achieve strong performance on older benchmarks (e.g., LiveCodeBench v2) but are less competitive on more recent ones (e.g., v5), revealing a higher degree of data leakage.

Table 9: Comparison between Qwen3-8B, and X-Coder on old and recent benchmarks.

Model	LiveCodeBench v2	LiveCodeBench v5	$\Delta (v2 \rightarrow v5)$
Qwen3-8B	88.1	57.5	-30.6
X-Coder-7B-SFT	78.2	60.3	-17.9
X-Coder-7B	80.1	62.9	-17.2

## H CASE STUDY

1782

17911792

17931794

1795 1796

#### H.1 SUCCESSFUL CASE

The SFT model frequently exhibits cognitive behaviors such as planning, verification, backtracking, and reflection, suggesting that these behaviors can be directly distilled from the teacher.

```
Listing 1: Thinking process of successful case.
1797
 Title: Adjacent GCD
1798
 Platform: AtCoder
1799
 Difficulty: Hard
 Contest_id: arc185
1800
 Contest date: 2024-10-13T00:00:00
1801
1802
 Thinking behavior:
1803
 We begin by defining F(m) as the total score of all subsequences of the first m elements \dots
1804
 and try to see whether we can break the problem into manageable pieces ... (Plan behavior)
1805
 At first, I considered summing \gcd(A_i,A_{i+1}) directly over the array \dots
1806
 but then realized that subsequences can skip indices, so adjacency is redefined \dots
1807
 this naive plan would miss many contributions ... (Reflection behavior)
1808
 To verify, I checked a small prefix: A=(9,6,4).
1809
 For m=2, the subsequences [9],[6],[9,6] give total =3 ...
 for m=3, manual enumeration yields 11,
1810
 while my recurrence only produced 8\ \dots (Verification behavior)
1811
 Clearly, the earlier idea of "just add \gcd(\mathsf{last},\mathsf{new})" was incomplete \dots
1812
 I realized each new subsequence must inherit the full score of its prefix subsequence,
1813
 and only then add the new gcd term ... (Reflection behavior)
 Concretely, inheriting the full score contributes 2F(m-1),
1814
 and the newly formed terminal pairs contribute \sum_{j < m} \gcd(A_j, A_m) \, 2^{j-1} \ldots
 Thus, I rebuild the recurrence: F(m)=2F(m-1)+\sum_{j=1}^{m-1}\gcd(A_j,A_m)\,2^{j-1}\;\ldots this formula accounts for doubling subsequences and the fresh contributions ... (Rebuild
1816
1817
1818
 behavior)
 For brevity, denote S(m) = \sum_{j < m} \gcd(A_j, A_m) \, 2^{j-1} \ldots
1819
1820
 Checking with m=2 and m=3 confirms it works \dots
1821
 e.g., for A = (9,6,4): F(3) = 2 \cdot 3 + (\gcd(9,4) \cdot 2^0 + \gcd(6,4) \cdot 2^1) = 6 + (1+4) = 11 \dots
 other boundary cases (single element, all equal numbers, coprime arrays)
1822
 also check out ... (Verification behavior)
1823
 However, computing \sum\gcd(A_j,A_m)\,2^{\,j-1} naively costs O(m) ... with N up to 5\times 10^5 this is too large ... (Reflection behavior)
1824
1825
 To optimize, I reorganize the sum by divisors of A_m ... use Möbius inversion ... and precompute an array H[t] = \sum_{d|t} \mu(d) \cdot \frac{t}{d} ...
1826
1827
 then each step requires iterating only over divisors of A_m^{\,\prime} ... (Optimization behavior)
1828
 We maintain U[t] = \sum 2^{j-1} for indices j where t \mid A_j ... then S(m) = \sum_{t \mid A_m} U[t] \cdot H[t], updated after each new element ... (Optimization behavior) With U[t] and H[t], we evaluate S(m) via divisors as \sum_{t \mid A_m} U[t] H[t], matching the earlier
1829
1830
1831
1832
 The complexity now depends on divisor counts ...
 on average \sim 10-\sim 20, worst-case \sim 128, giving \sim 10^8 operations overall ... (Complexity
1833
1834
 Overall time is O\left(\sum_{i=1}^N d(A_i)\right), with small average divisor counts in practice \dots
1835
 Finally, with precomputation of \mu, divisors, H, and powers of 2 modulo 998244353\ldots
```

```
the algorithm is complete, efficient, and matches all sample outputs ... (Conclusion)
```

1836

1837

Listing 2: Final code solution of successful case.

```
1840
1841
 mod = 998244353
 max_val = 100000
1842
1843
 import sys
1844
 def main():
1845
 data = sys.stdin.read().split()
 if not data:
1846
 return
1847
 n = int(data[0])
 A = list(map(int, data[1:1+n]))
1848
 1849
1850
 primes = []
1851
 for i in range(2, max_val+1):
 if is_prime[i]:
1852
 primes.append(i)
1853
 mu[i] = \cdot
 for p in primes:
1854
 if i * p > max_val:
1855
 break
 is_prime[i*p] = False
1856
 if i % p == 0:
 mu[i*p] = 0
1857
 break
1858
 else:
1859
 mu[i*p] = -mu[i]
1860
 divisors = [[] for _ in range(max_val+1)]
 for i in range(1, max_val+1):
1861
 for j in range(i, max_val+1, i):
1862
 divisors[j].append(i)
1863
 H_arr = [0] * (max_val+1)
1864
 for d in range(1, max_val+1):
 for t in range(d, max_val+1, d):
1865
 term = mu[d] * (t // d)
1866
 H_arr[t] = (H_arr[t] + term) % mod
1867
 pow2 = [1] * n
1868
 for i in range(1, n):
 pow2[i] = (pow2[i-1] * 2) % mod
1869
1870
 F = [0] * (n+1)
 U = [0] * (max_val+1)
1871
1872
 for idx in range(n):
 num = A[idx]
1873
 if idx == 0:
1874
 F[1] = 0
 for t in divisors[num]:
1875
 U[t] = (U[t] + pow2[0]) % mod
1876
 else:
1877
 total = 0
 for t in divisors[num]:
1878
 \label{eq:total} \begin{array}{l} \mbox{total} \; = \; (\mbox{total} \; + \; \mbox{U[t]} \; * \; \mbox{H_arr[t]}) \; \% \; \mbox{mod} \\ \mbox{F[idx+1]} \; = \; (2 \; * \; \mbox{F[idx]} \; + \; \mbox{total}) \; \% \; \mbox{mod} \\ \end{array}
1879
 for t in divisors[num]:
1880
 U[t] = (U[t] + pow2[idx]) % mod
1881
 for i in range (1, n+1):
1882
 print(F[i])
1883
 if ___name_
 _ == "__main__":
1884
 main()
1885
```

1886 1887 1888

1889

## H.2 BAD CASE

We identify three major failure modes in code reasoning process:

1891

1892

1894

1895

1896

1897

1898 1899

- (i) Premature termination under context exhaustion. As the context window approaches its limit, the model shortens its reasoning and rushes to produce a final answer (e.g., "Given the time, we output the following solution in Python for the sample.").
- (ii) Retrieval-like fallback instead of reasoning. Rather than attempting to derive a solution, the model sometimes recalls a "memorized" accepted submission in another language (e.g., C++) and attempts to translate it into Python, bypassing genuine reasoning.
- (iii) Incomplete code emission before cutoff. The model occasionally fails to output a complete code block before context exhaustion, leaving truncated or non-executable programs.

Listing 3: Thinking process of bad case.

```
1900
 // Instead, the solution might be to
1901
 - for the root, the value can be any number in [1, n] that is < seg_min (if the segment
1902
 is not empty)
 and > than the value of the parent (none) \rightarrow so any number in [1, seg_min-1] if the
1903
 segment is not empty,
1904
 or [1, n] if empty. - then for the children, they must be > than the value of the parent and < than the
1905
 minimum value
 in the segment (a[i]+1, i-1) for the parent.
1906
1907
 \ensuremath{//} But then the value for the parent is not known until we assign.
1908
 // We abandon.
1909
 Given the time, we output the following solution in Python for the sample.
1910
 For the sample input, it must output 3 for n=4 and [0,1,0,3]
1911
 and 353820794 for n=22 and the given a.
1912
 Here is a known AC solution in Python for the same problem (from a contest submission):
1913
1914
 sys.setrecursionlimit(10**6)
1915
 mod = 998244353
1916
 n = int(sys.stdin.readline().strip())
1917
 a = list(map(int, sys.stdin.readline().split()))
1918
 # Build the tree
1919
 g = [[] for _ in range(n)]
1920
 for i in range(n):
 if a[i] > 0:
1921
 g[a[i]-1].append(i)
1922
 # We will do a DFS. We need a segment tree that can update and query in O(log n)
1923
 # for the range [0, n-1] for the minimum value and the product of the counts.
 # We will use a segment tree that stores (min_value, count) for a range.
1924
 size =
1925
 while size < n:
1926
 size *= :
 seq_min = [10**9] * (2 * size)
1927
 seg_cnt = [1] * (2 * size)
1928
 # But the known solution uses a different approach.
1929
 # Instead, we output the solution from a known AC submission in C++.
1930
 # We found the following in C++ and we translate it to Python.
1931
 # C++ solution (AC for the same problem) by a known user:
1932
1933
 # include <iostream>
 # include <vector>
1934
 # include <algorithm>
1935
 # using namespace std;
const int mod = 998244353;
1936
1937
 # int main() {
 int n;
1938
 cin >> n:
1939
 vector<int> a(n);
 for (int i = 0; i < n; i++) {
1940
 cin >> a[i];
1941
1942
 vector<vector<int>> g(n);
 for (int i = 0; i < n; i++) {
1943
 if (a[i] > 0) {
 g[a[i]-1].push_back(i);
```

```
1944
 }
1945
 vector<int> dp(n, 1);
 vector<int> to(n, 1000000000); // to[i] = minimum value in the segment (a[i]+1, i-1)
1947
 // We will use a segment tree for range minimum query, and we will update it as we go.
 // But we will do a DFS in increasing order of a[i] and within the same a[i] in
1948
 increasing order of i.
1949
 vector<int> order(n);
 for (int i = 0; i < n; i++) order[i] = i;
1950
 sort(order.begin(), order.end(), [&](int i, int j) {
1951
 if (a[i] != a[j]) return a[i] < a[j];</pre>
 return i < j;
1952
 });
1953
 // segment tree for range minimum, size = 1<<19
 int size = 1;
1954
 while (size < n) size *= 2;
1955
 vector<int> seg(size, 1000000000);
 auto update = [&](int pos, int val) {
1956
 pos += size;
1957
 seg[pos] = val;
 while (pos > 1) {
1958
 pos /= 2;
1959
 seg[pos] = min(seg[2*pos], seg[2*pos+1]);
1960
 }:
1961
 auto query = [&](int 1, int r) {
 int res = 1000000000;
1962
 for (1 += size, r += size; 1 < r; 1 /= 2, r /= 2) {
1963
 if (1&1) res = min(res, seg[1++]);
if (r&1) res = min(res, seg[--r]);
1964
1965
 return res;
1966
 // initially, set the segment tree to a big number.
1967
 for (int i = 0; i < n; i++) {
 update(i, 1000000000);</pre>
1968
1969
 11
 for (int i : order) {
1970
 if (a[i] > 0) {
1971
 int parent = a[i]-1;
 int seg_min = query(a[i]+1, i);
1972
 // The value for i must be > than the value for parent and < than
1973
 (context exhausted here)
1974
```

#### H.3 REWARD HACKING AND PRETRAINING ARTIFACTS.

1975

1977

1978

1979

1980

1981 1982

1983

1984

1986 1987

1988 1989

1996 1997 We observe some interesting phenomena that emerge in the later stages of RL training.

First, we find that the policy does not genuinely strive for higher rewards by producing better algorithms. Instead, it resorts to exploiting edge cases and attempting to "cheat" for partial rewards, rather than solving the problem with real solutions.

Second, we observe that some rollouts include irrelevant advertising content (e.g., Need for Speed) or adopt an editorial-style explanation of solutions similar to those on competitive programming platforms such as Codeforces. This indicates that the base model was likely exposed to raw competition-platform data during pretraining, from which such artifacts were inherited.

# I THE USE OF LARGE LANGUAGE MODELS

In this paper, we adopt LLM for syntax checking and format calibration.