GrandCode: Achieving Grandmaster Level in Competitive Programming via Agentic Reinforcement Learning

Xiaoya Li, Xiaofei Sun, Guoyin Wang[†], Songqiao Su, Chris Shum and Jiwei Li

DeepReinforce Team

Abstract

Competitive programming remains one of the last few human strongholds in coding against AI. The best AI system to date still underperforms the best humans competitive programming: the most recent best result, Google's Gemini 3 Deep Think, attained 8th place even not being evaluated under live competition conditions. In this work, we introduce GrandCode, a multi-agent RL system designed for competitive programming. The capability of GrandCode is attributed to two key factors: (1) It orchestrates a variety of agentic modules (hypothesis proposal, solver, test generator, summarization, etc) and jointly improves them through post-training and online test-time RL; (2) We introduce Agentic GRPO specifically designed for multi-stage agent rollouts with delayed rewards and the severe off-policy drift that is prevalent in agentic RL. GrandCode is the first AI system that consistently beats all human participants in live contests of competitive programming: in the most recent three Codeforces live competitions, i.e., Round 1087 (Mar 21, 2026), Round 1088 (Mar 28, 2026), and Round 1089 (Mar 29, 2026), GrandCode placed first in all of them, beating all human participants, including legendary grandmasters. GrandCode shows that AI systems have reached a point where they surpass the strongest human programmers on the most competitive coding tasks.✉

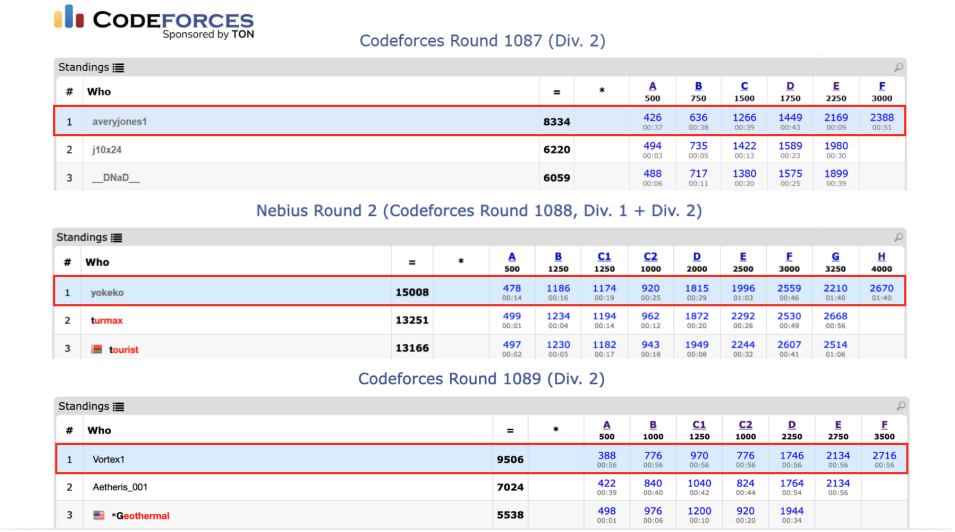


Figure 1: Codeforces standings overview for the three live contests in which GrandCode participated. GrandCode ranked first place in all three contests and being the first to finish all tasks in each of them.

1 Introduction

Despite rapid progress in AI for coding, the strongest current AI systems still fall short of the best human competitors in competitive programming. At the same time, the rapid improvement of large language models [21; 22; 23; 18; 7; 20; 35; 4] has driven substantial gains and has also spurred a growing literature on competitive-programming benchmarks, evaluation, and datasets [26; 6; 36; 17; 33; 14; 32; 19; 5; 40]. AlphaCode achieved a Codeforces rating of approximately 1300, placing it in the top 54% of competitors [16]; AlphaCode2 improved this to the 85th percentile [1]; and OpenAI's o3 ranks 175th globally [24]. Most recently, Gemini 3 Deep Think attained a ranking of 8th place, though this result was obtained on historical problems rather than under live contest conditions.

In this work, we introduce **GrandCode**, a multi-agent reinforcement learning system designed for competitive programming. GrandCode orchestrates a variety of agents and modules, and is optimized through both post-training and online adaptation with test-time RL in an explicitly agentic loop: the hypothesis model proposes structural conjectures, the main solver takes the main responsibility of reasoning and solution generation, the summarization model maintains a compact memory of long context, and the test-case generator produces edge test cases to challenge proposed solutions. The goal of this design is to enable an agentic loop of reasoning, verification, and feedback, and continually refining its solutions.

To address the severe off-policy issue in multi-turn agentic RL, we introduce Agentic GRPO, a variant of Group Relative Policy Optimization [27] that combines immediate reward updates with delayed correction, enabling more effective credit assignment under long, multi-stage rollouts and asynchronous training.

GrandCode is the first AI system to consistently surpass the best human competitors in competitive programming under live contest conditions: in the three most recent Codeforces rounds under standard live contest conditions: Round 1087 on March 21, Round 1088 on March 28, and Round 1089 on March 29, 2026, GrandCode placed first in all three contests, outperforming every human participant, including multiple top-ranked legendary grandmasters.

GrandCode does not emerge in isolation, but is built on a broad ecosystem of prior AI components and systems, which we want to acknowledge. It builds on Qwen 3.5 as the foundation model [35], chosen for its accessible SFT pipeline and multimodal capabilities. We also employ models such as Kimi 2.5 [20], GLM [41], and other closed-source LLMs for data generation in several modules, and incorporates modules and implementation ideas from systems such as Slime [30] and Tinker [29].

The rest of this paper is organized as follows.

- Section 2 describes the Codeforces evaluation setting and GrandCode's contest results.
- Section 3 presents the overall system design.
- Section 4 introduces Agentic GRPO and its delayed correction mechanism for multi-stage agent rollouts.
- Section 5 describes our test-case generation pipeline.
- Section 6 introduces the hypothesis-generation stage.
- Section 7 describes continued training and supervised fine-tuning.
- Section 8 presents multi-component RL orchestration.
- Section 9 describes the supporting RL infrastructure.
- Section 10 discusses test-time RL in live contests.

✉ Email: {xiaoya_li, xiaofei_sun, songqiao_su, chris_shum, jiwei_li}@deep-reinforce.com; † independent researcher.

2 Codeforces Competition Results

2.1 Codeforces

Codeforces[1] is one of the most prominent platforms for competitive programming and hosts frequent public contests with large and highly skilled participant pools. In a typical round, participants are given a sequence of problems with increasing difficulty and must solve them as quickly as possible under strict time and memory limits. Many rounds are organized by division: Div. 1 typically targets higher-rated participants, Div. 2 targets a broader pool of lower-rated participants, and Div. 1+2 rounds combine both groups in a shared contest. Each solution is submitted to an online judge that evaluates the code on hidden test cases and returns only limited feedback, such as whether the submission is wrong or exceeds the time limit. A successful submission must therefore be both **correct** and **efficient**, and earlier accepted submissions receive better scores. Problem statements are often long, combine narrative description with precise constraints and input-output specifications, and may also include figures, as illustrated in Figure 9.

It is worth noting that Codeforces has policies against AI-generated content, and accounts suspected of using AI face removal. High-ranking accounts in the contest are under even tighter scrutiny. To get the final score for the full version, we wait until the human participants have nearly finished the task before submitting the full version.

2.2 Participating Results

GrandCode participated in the three most recent Codeforces live competitions under the contestant IDs `averyjones1` in Round 1087, `yokeko` in Round 1088, and `Vortex1` in Round 1089.

Round	Div.	Date	Time (UTC+3)	Duration	ID
1087	2	Mar. 21, 2026	17:35–19:35	02:00:00	averyjones1
1088	1+2	Mar. 28, 2026	17:45–20:15	02:30:00	yokeko
1089	2	Mar. 29, 2026	17:35–19:50	02:15:00	Vortex1

We report two scores: $S(\text{separate})$, which is obtained by summing the scores of tasks at the time they are completed, i.e., by submitting each solution as soon as it is ready. By *separate*, we mean that the submissions are made independently, with submission and standings details shown in Figures 7 and 8; and $S(\text{joint})$, which is the score based on the full set of submissions in a single account, as shown in Figure 1, with submission and standings details shown in Figure 6. In practice, $S(\text{joint})$ is strictly lower than $S(\text{separate})$ because of waiting time.

It is worth noting that this can also lead to multiple-submission penalties that are not reflected. We believe this effect is small because (1) all tasks are solved within at most four submission attempts, and (2) in the Codeforces scoring system, the penalty for multiple attempts is small compared with the reward for early accepted submissions.

Round	$S(\text{separate})$	$S(\text{joint})$	Finish time
1087	9269	8334	00:51:11
1088	16511	15008	01:40:35
1089	11596	9506	00:56:43

GrandCode achieved the best score in all three contests and was also the first to finish all tasks in each of them. The corresponding $S(\text{separate})$ scores were 9269, 16511, and 11596 for Rounds 1087, 1088, and 1089, respectively, and the corresponding $S(\text{joint})$ scores were 8334, 15008, and 9506. Details for submissions and standings for *joint* is shown in Figiure 1.

[1] `https://codeforces.com/`

3 System Overview

The proposed sytesm combines three learned policies and the test-case generation module:

1. **Main solver** π_{main} corresponds to the core policy that generates reasoning traces and code.
2. **Hypothesis model** $\pi_{\text{hypothesis}}$ proposes intermediate conjectures or structural properties, which will be verified them on small instances. Accepted hypothesis will be injected into the prompt for Main solver.
3. **Summarization model** π_{summary} compresses very long reasoning traces so that hard problems remain tractable in later RL stages.
4. **Test-case generation** constructs adversarial tests, solution-attack tests, and large-size stress cases to evaluate candidate programs before submission.

The overall workflow has two phases:

1. **Post-training**, which can further be divided into three substages:
   (a) **Continued pre-training** on broad competitive programming data to improve the model's general problem-solving ability. We start from existing task datasets, use them as seeds for data expansion, generate additional data with Claude and Gemini, and continue training the Qwen model on the resulting corpus.
   (b) **Supervised fine-tuning** on high-quality (question, thinking, solution) triples. Given (question, solution) pairs, we generate reasoning traces for data expansion and use the resulting triples for supervised fine-tuning, together with auxiliary components such as $\pi_{\text{hypothesis}}$ and π_{summary}.
   (c) **Multi-component reinforcement learning** to jointly optimize the system, enabling the main solver and auxiliary components to collaborate more effectively under the final objective.
2. **Test-time / live-contest solving.** During this stage, the model solves the current problem instance using difficulty-aware routing, direct generation for easy cases, and an online **test-time RL** loop with verification feedback for harder cases.

Figure 2 shows the full pipeline.

Difficulty-based routing We fine-tune a lightweight classifier to assign each task to one of five difficulty levels, where Level 1 denotes the easiest problems and Level 5 the hardest.

4 Agentic GRPO with Immediate Reward and Delayed Correction

In many agentic settings, especially code optimization, it often requires the agent to perform multiple rounds of self-debugging and troubleshooting. As a result, the full process from starting a rollout to receiving its final reward can be extremely slow. This is not only because the generated sequence itself is long, but more importantly because the code must be evaluated multiple times, and each evaluation involves compilation and execution. For many competitive programming tasks, a single evaluation can easily exceed one minute.

This creates a severe off-policy problem. Asynchronous RL methods such as pipeline-RL [25], where sampling and training proceed concurrently with in-flight weight updates and a single sequence may be generated under multiple policy versions, is a necessary solution, but not enough.

Orthogonal to asynchronous training, we propose *Agentic GRPO* with *Immediate Reward and Delayed Correction*, which is designed to handle multi-stage rollouts in agentic settings under the asynchronous training framework. Suppose we have a multi-stage rollout of the form, the sequence for stage $t \in [1, N]$ is denoted by s_t, with the reward r_t:

$$s_1, r_1, s_2, r_2, \ldots, s_N, r_N.$$

In the standard GRPO, we update the whole trajectory with the final reward r_N and ignore the intermediate rewards $r_1, r_2, \ldots, r_{N-1}$. The core idea behind *Agentic GRPO* is that to have the trainer update the policy as soon as possible once an intermediate reward r_t is available instead of waiting for the final reward r_N. When we finish the whole sequence, we apply a delayed correction term $r_N - r_1, r_N - r_2, \ldots, r_N - r_{N-1}$ to the

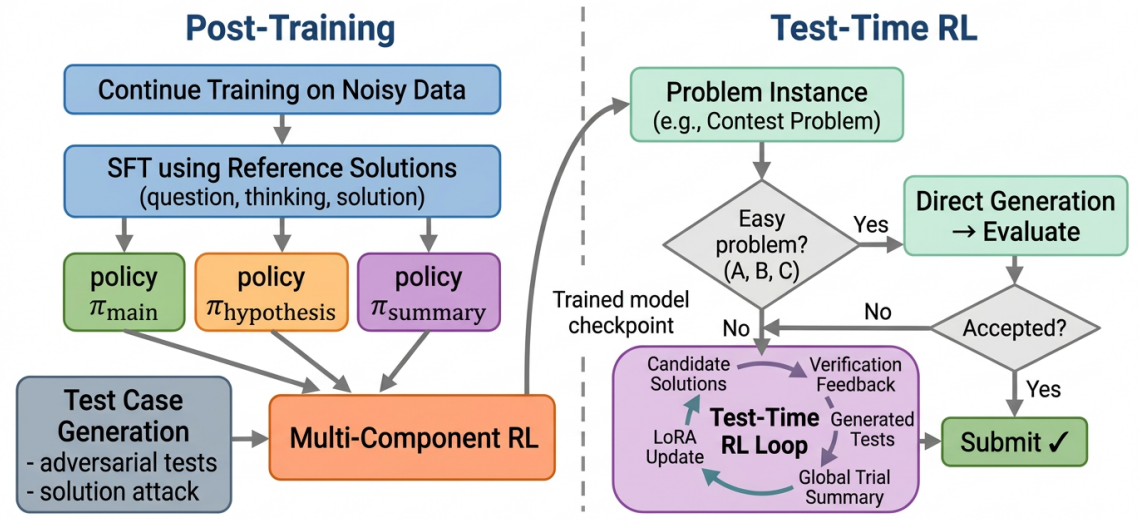


Figure 2: Overview of the full pipeline. In post-training, we continue training on noisy competitive-programming data, perform supervised fine-tuning on reference solutions, train auxiliary hypothesis generation policy $\pi_{\text{hypothesis}}$ and summarization policy π_{summary} and jointly optimize the system with multi-component RL. At test/online-contest time, the model uses direct generation for easy cases, and an online test-time RL loop for harder cases.

trainer. Therefore, for each subsequence s_t with reward r_t, gradient updates will be divided into two stages, *Immediate Reward* and *Delayed Correction*.

Immediate Reward we first update the trainer using the immediate reward r_t. For all s_t of K rollouts,

$$A_t^{(i)} = \frac{r_t^{(i)} - \mu_t}{\sigma_t}, \qquad \mu_t = \frac{1}{K}\sum_{i=1}^{K} r_t^{(i)}, \qquad \sigma_t = \operatorname{std}\left(r_t^{(1)}, \ldots, r_t^{(K)}\right). \tag{1}$$

The corresponding GRPO loss on tokens in s_t is

$$\mathcal{L}_t = -\frac{1}{K}\sum_{i=1}^{K}\sum_{u \in s_t^{(i)}} \min\left(\rho_u^{(i)} A_t^{(i)}, \operatorname{clip}\left(\rho_u^{(i)}, 1-\epsilon, 1+\epsilon\right) A_t^{(i)}\right), \tag{2}$$

where

$$\rho_u^{(i)} = \frac{\pi_\theta(a_u \mid s_u)}{\pi_{\theta_{u,\text{beh}}^{(i)}}(a_u \mid s_u)}. \tag{3}$$

Here $\theta_{u,\text{beh}}^{(i)}$ denotes the behavior policy version that generated token a_u in rollout i, which may vary across tokens under asynchronous pipeline training.

Delayed Correction When the full rollout $s_1, s_2, ..., s_N$ is completed and the final reward r_N becomes available, we use it to correct each earlier stage s_t. We first define the reward correction as follows:

$$\delta_t^{(i)} = r_N^{(i)} - r_t^{(i)} \tag{4}$$

and normalize it as

$$A_t^{(i)} = \frac{\delta_t^{(i)} - \mu_t}{\sigma_t}, \qquad \mu_t = \frac{1}{K}\sum_{i=1}^{K} \delta_t^{(i)}, \qquad \sigma_t = \mathrm{std}\left(\delta_t^{(1)}, \ldots, \delta_t^{(K)}\right). \tag{5}$$

The delayed correction applied to the older stage s_t is

$$\mathcal{L}_t^{\mathrm{corr}} = -\frac{1}{K}\sum_{i=1}^{K} \sum_{u \in s_t^{(i)}} \min\left(\hat{\rho}_u^{(i)} A_t^{(i)}, \mathrm{clip}\left(\hat{\rho}_u^{(i)}, 1-\epsilon_2, 1+\epsilon_2\right) A_t^{(i)}\right), \tag{6}$$

where

$$\hat{\rho}_u^{(i)} = \frac{\pi_{\theta'}(a_u \mid s_u)}{\pi_{\theta_{u,\mathrm{beh}}^{(i)}}(a_u \mid s_u)}, \qquad \epsilon_2 \le \epsilon. \tag{7}$$

Here θ' denotes the current policy after subsequent updates, while $\theta_{u,\mathrm{beh}}^{(i)}$ is the behavior-policy version that originally generated token a_u.

The proposed Agentic GRPO can be used together with asynchronous training methods: the former enables more timely credit assignment in multi-turn agent rollouts, while the latter overlaps sampling and training for higher throughput. A more details theoretical analysis of Agentic GRPO is shown in Appendix C.

5 Test Case Generation

In programming contests, the real judge test cases are hidden, so we must construct its own test cases and pass them before the submission. Therefore, we need to generate edge cases that can expose logical bugs, boundary failures, and incorrect complexity assumptions before submission.

Test case generation faces two main challenges. First, it is difficult to generate genuinely **adversarial test cases** that expose subtle logical errors rather than merely checking superficial correctness. Second, **large-size test cases** are hard to use for verification, because in many settings the only trusted solver available to us is a brute-force implementation, and that solver times out on large inputs, which makes we don't have gold outputs to compare with for large-size inputs.

5.1 Adversarial Test Case Generation

We adopt two strategies for adversarial test generation: **difference-driven test generation** and **solution attack**.

Difference-driven Test Case Generation If a test case can expose a difference between two solutions, it is very likely an adversarial case.

Given a list of candidate solutions sampled during training, we iteratively generate candidate test cases by prompting Claude, GPT, DeepSeek V3, and Kimi 2.5 by to produce inputs that are likely to reveal corner cases. No candidate solution is provided to the LLMs at this stage. We further employ a generator-validator framework inspired by CodeContests+ [33], in which an additional LLM validator is used to filter or refine the generated tests. We then run the generated test cases on all sampled solutions and check whether any test induces different outputs between them. Test cases that trigger such differences are preserved. Whenever such a case is found, we feed it back to the LLM and ask it to describe what edge condition the input may be

Stage	Passed	Total
Base test suite	42	50
After difference-driven generation + solution attack	48	50
After submission feedback + continued online generation	50	50

Table 1: Results on 50 real Codeforces problems using the Codeforces judge as the final criterion.

triggering, and then generate additional tests of a similar kind. In this way, the test pool is gradually enriched with more informative cases, and by the end of optimization for a problem we are able to maintain a strong problem-specific test suite. It is worth noting that this idea is akin to the spirit to the agentic verification approach [19].

Solution Attack In the training set, where a gold solution is available, we further generate adversarial tests by directly comparing the gold solution with each candidate solution. We feed both solutions to an LLM and ask it to analyze their differences, identify potential bugs in the candidate solution, and propose edge cases that are likely to expose those bugs. We carry out this process in a multi-turn conversation mode, allowing the model to iteratively refine its hypotheses and search for stronger adversarial tests. Each generated test case is then executed on both the gold solution and the candidate solution to verify whether it indeed induces a behavioral difference. Test cases that successfully separate the two solutions are preserved, since they provide direct evidence of a real bug in the candidate solution.

Using the adversarial test cases generated above, we further fine-tune a Qwen-3.5-27B model to generate such verified adversarial examples given the problem statement and the candidate solution.

5.2 Test-time Strategies

When we handle a specific contest task, we use the similar online strategy: We prompt the fine-tuned model to generate multiple adversarial test cases conditioned on each generated solution, and after every few candidate solutions, we regenerate part of the test set and refresh the evaluation pool. Test examples that trigger a difference are kept, since they are highly informative indicators of edge cases. A solution is submitted only if it passes all of these test cases.

5.3 Results on Real Codeforces Problems

We evaluate test case genearation on 50 real Codeforces problems by submitting solutions to the real Codeforces website and using the real system as the final criterion. We check whether a solution that passes our generated test suite also passes the hidden official tests. Table 1 summarizes the results. The pass count increases from 42 to 48 after applying difference-driven test case generation and solution attack. For the remaining two failures, we further incorporate submission feedback and continue generating additional test cases online, which raises the pass count to all of 50 tests.

5.4 Large-size Test Cases

In real-time contest, feedback from the submission system can also be harnessed to generate test cases with larger input sizes. A failure due to time limit exceeded often suggests that the generated code snippet may be logically correct but computationally inefficient. If such the code snippet is faster than the brute-force baseline, we can use it as a solver on test cases with larger input sizes, allowing evaluation on inputs that are too large for brute force.

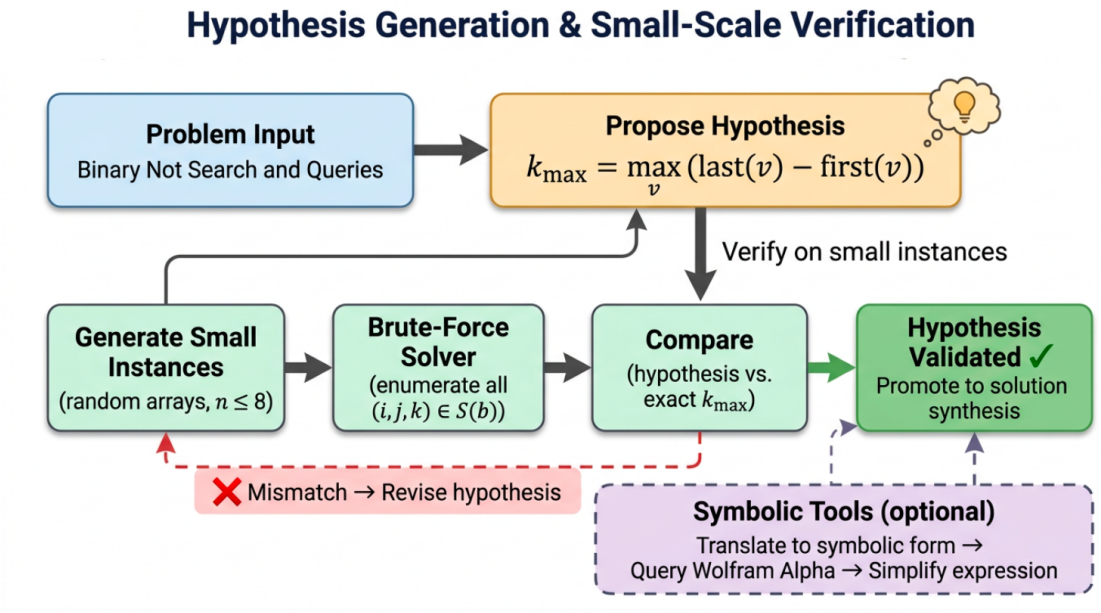


Figure 3: Hypothesis generation and small-scale verification. The agent first proposes a compact characterization ($k_{\max} = \max_v(\text{last}(v) - \text{first}(v))$), then generates small random instances, computes the exact answer via a brute-force solver that enumerates all tuples $(i, j, k) \in S(b)$, and compares against the hypothesized value. A mismatch triggers hypothesis revision and only validated hypotheses are promoted to solution synthesis.

6 Prelude: Hypothesis Generation

6.1 Overview

The first stage of the agent workflow is **hypothesis generation** and **small-scale verification**. Before attempting full solution synthesis, we need to propose intermediate claims, structural properties, or compact mathematical characterizations of the problem, and then checks whether they are valid on small instances. For example, the agent may verify that whether the problem is a dynamic-programming problem in nature, or confirm that the input graph satisfies a structural property such as being undirected. These checks can be easily performed on small-scale inputs using the brute-force algorithm.

This stage can also invoke symbolic tools. When a problem or a broken-down subproblem can can be fully translated to a symbolic form, we will query an external engine such as Wolfram Alpha[2] to simplify or solve the resulting expression. This is especially useful when the main difficulty lies in discovering the right formula or invariant rather than in implementation details.

As a concrete example, consider the Codeforces problem "Binary Not Search and Queries." We may first hypothesize that $k_{\max} = \max_v(\text{last}(v) - \text{first}(v))$. Next we ask the model to generate many small inputs, computes the exact answer with a brute-force solver, and checks whether the conjectured formula matches the

[2] https://www.wolframalpha.com/

Model for Hypothesis Gen	pass@1	pass@5
Qwen-3.5-27B	34%	44%
+SFT	45%	52%
+SFT+RL	52%	57%

Table 2: Pass@1 and pass@5 on a 200-problem evaluation set for hypothesis generation. Supervised fine-tuning substantially improves over the base Qwen-3.5-27B model, and additional RL training brings further gains on both metrics.

true value on those instances. When a mismatch is found, the counterexample is fed back to the model together with the failing hypothesis, and the model is asked to propose a revised conjecture; this verify-and-revise loop repeats until the hypothesis is consistent with all small-scale tests. This process is illustrated in Figure 3.

Hypotheses that survive this iterative validation are injected into the prompt of the main solving thread, providing verified structural insights that guide subsequent solution synthesis. The tools used in this stage include shell execution, Wolfram Alpha, and web search. For search in particular, we apply a filter to avoid directly retrieving historical answers, editorials, or hints for the target problem.

6.2 Training Hypothesis Generation Model

We use a lightweight model for this component rather than allocating a large model to every instance. We denote this model by $\pi_{\text{hypothesis}}$ and use Qwen-3.5-27B as the base model for $\pi_{\text{hypothesis}}$.

SFT We first train $\pi_{\text{hypothesis}}$ with supervised fine-tuning. To construct SFT data, we use Claude to generate multiple candidate hypotheses given only the problem statement, and retain those that are verified to be correct. For problems that come with editorials or reference solutions, we additionally provide these materials to Claude and ask it to produce hypotheses implied by the official reasoning, again retaining only the correct ones. The resulting verified hypotheses are then used as supervision for SFT.

RL After SFT, we further optimize $\pi_{\text{hypothesis}}$ with reinforcement learning using GRPO. At the current stage, the reward for a generated hypothesis h is the proportion of verification tests it passes:

$$r_{\text{verify}}(h) = \frac{1}{|T|} \sum_{t \in T} \mathbf{1}[h \text{ is correct on } t]. \tag{8}$$

At the current stage, $\pi_{\text{hypothesis}}$ is trained independently for simplicity. However, this is inherently imperfect, since generating a correct hypothesis does not necessarily imply that it is helpful to generate the correct solution. In the full system, $\pi_{\text{hypothesis}}$ will later be jointly trained with the overall solver in the RL training stage for the latter.

6.3 Evaluation

We randomly select 200 problems for evaluation. Since each problem comes with a test-based checker, evaluation can be performed directly based on the provided tests. Table 2 reports pass@1 and pass@5 for different models, where pass@1 denotes the success rate of the first generated hypothesis and pass@5 denotes the success rate at which at least one of five generated hypotheses passes the tests. The results show consistent improvements from the base Qwen-3.5-27B model to +SFT and further to +SFT+RL on both metrics.

6.4 Clue Finding Using OEIS

We also employ a side route that does not involve any additional model training. We first compute outputs for a few small values of N, and query the On-Line Encyclopedia of Integer Sequences (OEIS, `https://oeis.org/`) with the resulting sequence to search for useful clues. If the lookup is successful, the returned pattern, formula, or related structural hint is included in the prompt for subsequent solving.

Model	Accept Rate	Level 5 Solved	Weighted Score (0-100)
Gemini 3.1 Pro	75%	7/20	64.3
Claude Opus 4.6	73%	8/20	63.7
GPT-5.4	72%	7/20	63.0
Kimi K2.5	65%	5/20	53.3
DeepSeek V3.2	65%	4/20	52.7
Qwen 3.5-397B	64%	4/20	52.3

Table 3: Accept rates, Level 5 correct answers (out of 20), and scaled weighted scores on 100 benchmark questions. Questions are equally distributed across five difficulty categories. Weighted scores apply a 1-5 multiplier based on difficulty, normalized to a 0-100 scale.

7 Post Training with Continue Training and SFT

Benchmarking Existing Models We select 100 questions, equally distributed across the five difficulty categories, and evaluate proprietary models on this benchmark, including Gemini 3.1 pro, Claude Opus 4.6 and GPT 5.4. Table 3 reports three complementary metrics for each model: the overall accept rate, the number of hardest Level 5 problems solved, and a difficulty-weighted score that assigns weights $1, 2, 3, 4, 5$ to Levels 1–5, respectively. Overall, current frontier models achieve an overall accept rate of roughly 70%–75% and solve about 35%–40% of Level 5 problems.

7.1 Continue Training on Noisy Data

For continued pretraining, we first collect a broad seed set of competitive-programming problems from TACO [13], LeetCode [12], USACO [31], CodeContests [16], IOI [9], and additional problems crawled from various online sources. We then use Gemini 3.1 Pro to expand this seed set into a much larger and more diverse training corpus. Next, we directly prompt Claude 4.6 and Gemini 3.1 to generate detailed thinking processes for these problems, and use the resulting question-thinking-solution tuples to train Qwen 3.5-397B. To make the model familiar with settings where a hypothesis is provided, we randomly convert 20% of the continued-pretraining examples into hypothesis-conditioned cases, in which a hypothesis generated by $\pi_{\text{hypothesis}}$ is incorporated in the prompt before we generate the detailed thinking process. At this stage, the data can be noisy because it is partly generated, and some synthesized reasoning traces or answers may be incorrect. However, the goal of continued pretraining is primarily to improve the model's general competitive-programming ability rather than to provide precise supervision. We leave more fine-grained filtering and high-quality supervision to the later SFT stage. It is worth noting that since the model is trained on the data is generated by Claude 4.6 and Gemini 3.1, the upper bound for the ckpt by the end of continue training is around 73% accepted rate, as shown in Table 3.

7.2 SFT using Reference Solutions

For SFT, we focus only on non-synthetic problems with reference solutions, some of which also come with hints. Our goal is to generate a high-quality thinking process c that can plausibly lead to the provided reference solution. With such tuples, we can directly perform SFT by training the model G_{main} to first predict c given x, and then predict y given (x, c), denoted by $p(y, c \mid x)$.

Solution Matching For each question x, we prompt Gemini 3.1 Pro, Claude Opus 4.6, GPT-5.4, GLM 4.5, Kimi K2.5, and DeepSeek V3.2 multiple times to generate both a reasoning trace c and a solution y. We then compare the generated solution against the gold solution. If the generated solution is equivalent to the gold solution, we keep the associated reasoning trace. Even when the implementation is not textually identical, we still retain it if it is comparable to or better than the reference solution in terms of efficiency, since many problems admit multiple valid solutions.

It is worth noting that this strategy is effective for relatively easy tasks, where we can often recover a solution close to the gold one. For harder tasks, however, the generated solution frequently fails to match the gold solution, making this simple filtering procedure insufficient.

Finding Optimal Thinking Trace c for Hard Problems For each hard question-solution pair (x, y), we first prompt Claude 4.6 or Gemini 3.1 to generate N candidate reasoning contexts $C = [c_1, \ldots, c_N]$ given (x, y). We denote the full forward model by π_{main}, and its original checkpoint by $\pi^{(0)}_{\text{main}}$. We then select c_i based on the following score:

$$s(c_i) = \alpha \cdot \frac{\log p_{\pi_{\text{main}}}(c_i, y \mid x)}{|c_i| + |y|} + \beta \cdot \frac{\log p_{\pi^{(0)}_{\text{main}}}(x \mid c_i)}{|x|}, \tag{9}$$

where $p_{\pi_{\text{main}}}(c_i, y \mid x)$ is computed by the post-trained Qwen 3.5-397B, $p_{\pi^{(0)}_{\text{main}}}(x \mid c_i)$ is computed by the original Qwen 3.5-397B, and both terms are length-normalized. Intuitively, the first term favors reasoning contexts that make the target reasoning and solution likely under the post-trained model, while the second term favors contexts that remain predictive of the original question. The second term is essential for distinguishing genuinely useful reasoning traces from degenerate ones that merely copy or reveal the answer. At the end of this procedure, the tuples are combined into the the SFT training data.

7.3 Training Summarization Model

For hard questions, the thinking trace can easily exceed 100K tokens, which makes both inference and the later RL stage extremely computationally expensive. Additionally, extremely long traces are also harder to optimize with RL. We therefore first train a separate summarization model π_{summary}.

Given a long reasoning trace c, we partition it into chunks $c^{(1)}, c^{(2)}, \ldots, c^{(n)}$, so that the full example is represented as $(x, c^{(1)}, c^{(2)}, \ldots, c^{(n)}, y)$. The summarization model maintains a progressive summary state $s_1, s_2, \ldots, s_n$, where $\pi_{\text{summary}}(c^{(1)}) \to s_1$, $\pi_{\text{summary}}(s_1, c^{(2)}) \to s_2$, and in general $\pi_{\text{summary}}((s_{t-1}, c^{(t)}) \to s_t$. Intuitively, s_t is a compact summary seen up to chunk t.

There has been existing work that integrates summary generation into training mostly in an end-to-end fashion, including MemAgent [37], Agentic Memory [38], and Composer 2 [3]. In these approaches, summarization usually shares the parameters with the main model, and is optimized jointly with the main model in an end-to-end fashion based on the final answer.

While this end-to-end formulation is natural, its reward is relatively sparse, since supervision is dominated by the terminal outcome of the full trajectory. By contrast, we first take advantage of the existing SFT data to learn as strong a summarization policy as possible, rather than directly entering the RL stage and relying only on the final reward as the training signal. We train the summarizer progressively in multiple stages. This gives denser intermediate supervision in the early stage of training.

Stage 1 We first train each local summarization step with RL. Starting from Qwen-3.5-27B, the policy maps the first chunk $c^{(1)}$ to a summary s_1. Concretely, for a sampled summary s_1, we define the score as:

$$\text{score}(s_1) = \alpha \frac{\log p\left(c^{(2)}, \ldots, c^{(n)}, y \mid s_1, x\right)}{\left|c^{(2)}, \ldots, c^{(n)}\right| + |y|} + \beta \frac{\log p\left(c^{(1)} \mid s_1\right)}{\left|c^{(1)}\right|} \tag{10}$$

The first term encourages s_1 to preserve information needed for the later chunks and the final answer, while the second term penalizes reconstruction error by encouraging s_1 to retain enough information to reconstruct the original chunk.

Model	Accept Rate	Level 5 Solved	Weighted Score (0-100)
Qwen 3.5-397B	64%	4/20	52.2
+continue training	71%	6/20	61.0
+continue training + SFT	73%	7/20	62.5
+continue training + SFT + summary	72%	7/20	61.7

Table 4: Ablation of continued training, SFT, and summary-augmented training on the 100-problem benchmark. Continued training delivers a large gain over the base model, and SFT provides an additional improvement.

We optimize this policy with GRPO. Given a group of rollout rewards $\{r_i\}_{i=1}^{G}$, we normalize them within the group as $A_i = (r_i - \text{mean}(r))/\text{std}(r)$ and update the policy using these relative advantages. We use analogous objectives for later transitions $(s_{t-1}, c^{(t)}) \to s_t$. This stage teaches each summary state to preserve the information necessary for continuing the long reasoning process.

Stage 2 We then train the full progressive chain end to end. We sample the entire summarization trajectory $c^{(1)} \to s_1$, $(s_1, c^{(2)}) \to s_2$, …, $(s_{n-1}, c^{(n)}) \to s_n$, and use the final answer likelihood, e.g. the normalized $\log p(y \mid s_n)$, as the terminal reward for GRPO.

Stage 3 Finally, π_{summary} is integrated into the full RL training of π_{main}, so that summarization and downstream solving can be optimized jointly in the overall pipeline, as will be described in detail in Section 8.

In addition, data with summaries is mixed into the SFT training data, so that π_{summary} becomes familiar with both settings where summaries are present and settings where they are not.

7.4 Evaluation

We evaluate the effect of continued training, SFT, and summary-augmented training on the same 100-problem benchmark described above. Table 4 reports the results. Continued training substantially improves over the base model, raising the overall accept rate from 64% to 71% and the weighted score from 52.2 to 61.0. SFT yields a further improvement, reaching 73% accept rate, 7/20 solved Level 5 problems, and a weighted score of 62.5. We observe a minor performance degradation when the summarization module is incorporated.

7.5 Multimodal Problem Solving

Many competitive-programming problems come with images or diagrams, as illustrated in Figure 9. Since Qwen is a multimodal model, we follow its native multimodal input protocol when such visual content is present. We also experimented with converting images into text descriptions and feeding only the extracted text into the model but find that this text-only conversion significantly underperforms direct multimodal processing. A likely reason is that many images are still visually convoluted and difficult to describe faithfully in text, and the conversion often loses precisely the spatial or structural information necessary for reasoning, as illustrated by the examples in Figure 4.

8 Multi-Component RL Orchestration

The final stage involves multiple components for RL training, including π_{main}, $\pi_{\text{hypothesis}}$, and π_{summary}. We choose multi-component RL for three main reasons: (1) **Smaller auxiliary models:** $\pi_{\text{hypothesis}}$ and π_{summary} use smaller models, which saves substantial compute resources; (2) **Stronger initialization:** $\pi_{\text{hypothesis}}$ and π_{summary} are separately pretrained in earlier stages and are already reasonably strong before the final joint RL stage, so they require only moderate joint tuning together with π_{main}; (3) **Better disentanglement:** separate training makes it easier to isolate the influence of different modules: if one uses only a single final solution reward for π_{summary} and π_{main} and the result is poor, it is unclear whether the

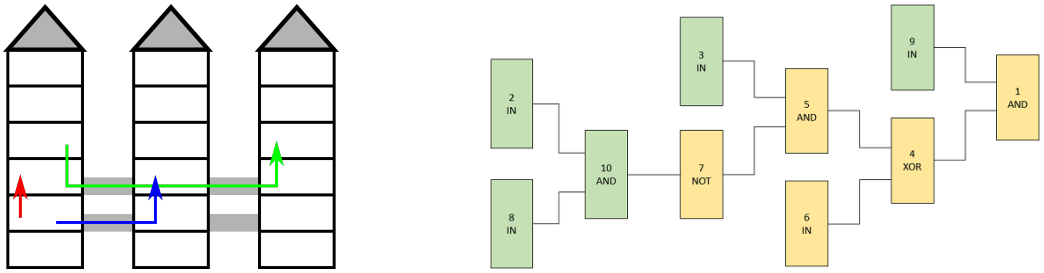


Figure 4: Examples of contest figures whose visual structure is difficult to capture with text-only descriptions.

failure comes from a bad summary or a bad solver. This modular design therefore makes the final RL stage substantially more compute-efficient and easier to optimize.

8.1 Reward for π_{main}

Reward evaluation of a generated code snippet consists of three stages:

1. **Executability.** If a generated code snippet cannot be compiled or executed, it receives score 0.
2. **Correctness.** Correctness is measured by comparing the code snippet output with a reference output. When a gold solution is available, as in training-time evaluation, we use the gold solution to produce the reference output. At test time, when no gold solution is available, we compare against the output of a brute-force solver on small inputs. If a code snippet fails these correctness checks, it receives score 0.
3. **Efficiency.** For a code snippet that pass correctness checking, we compare runtime against a brute-force baseline on the generated tests, whose scale doesn't has to be small. For each test case, the reward is defined as the speedup over the brute-force algorithm; if the generated code times out, we assign a score of 0.1 for that test case. The final score is the average over all test cases.

A detailed formalization in math in shown in Appendix B.

Length Penalty based on Difficulty More difficult questions typically require longer chains of reasoning, so the thinking-length penalty should depend on the problem difficulty. Recall that we fine-tune a lightweight classifier that assigns each task a difficulty level $d \in \{1,2,3,4,5\}$, where larger d indicates a harder problem. We therefore allocate a larger thinking-token budget to harder tasks. Let l denote the thinking length, B the base budget for Level 1 problems, and α the budget growth factor per difficulty level. The difficulty-dependent budget is $B \cdot \alpha^{(d-1)}$, and the penalty is zero when the thinking length stays within this budget and increases only when it is exceeded:

$$Penalty(l,d) = \max\left(0,\ \frac{l - B \cdot \alpha^{(d-1)}}{B \cdot \alpha^{(d-1)}}\right). \tag{11}$$

In this way, easier problems are encouraged to use short, efficient reasoning traces, while harder problems are allowed more extensive thinking before incurring a penalty.

8.2 Orchestrating of π_{main} and $\pi_{\text{hypothesis}}$

We first sample multiple candidate hypotheses from $\pi_{\text{hypothesis}}$. Only hypotheses that pass all verification tests are allowed to continue to the next stage. The reward for the hypothesis-generation stage is

$$r_1(h) = r_{\text{verify}}(h) = \frac{1}{|T|}\sum_{t \in T} \mathbf{1}[h \text{ is correct on } t]. \tag{12}$$

For each passed hypothesis h, we then invoke π_{main} to generate reasoning traces and candidate solutions conditioned on (x, h). In parallel, we also generate rollouts from π_{main} without any hypothesis, which we denote by the empty condition $\varnothing$. Let $S(x, h)$ denote the average score of solutions generated with hypothesis h incorporated, and let $S(x, \varnothing)$ denote the average score of solutions generated without any hypothesis.

We define the total reward for $\pi_{\text{hypothesis}}$ by

$$r(h) = \begin{cases} \alpha r_1(h) + \beta\big(S(x,h) - S(x,\varnothing)\big), & \text{if } h \text{ passes all tests,} \\ \alpha r_1(h), & \text{otherwise.} \end{cases} \tag{13}$$

This design encourages $\pi_{\text{hypothesis}}$ not only to generate locally valid hypotheses, but also to produce hypotheses that measurably improve downstream solution quality when used by π_{main}. Importantly, $\pi_{\text{hypothesis}}$ is allowed to generate hypotheses over multiple rounds with self-correction. This iterative loop lets the hypothesis generator recover from initial mistakes and converge toward hypotheses that are both locally correct and globally useful.

8.3 Orchestrating of π_{summary} and $\pi_{\text{hypothesis}}$

π_{summary} is triggered only for long reasoning sequences. During training, we gradually increase its triggering frequency, together with the curriculum that shifts toward harder problems over time. As a result, early training focuses primarily on $\pi_{\text{hypothesis}}$ and π_{main}, while later training increasingly exposes the system to summary-dependent rollouts. When π_{summary} is triggered, we use the same final reward as for π_{main}. Concretely, if the summary-conditioned rollout ultimately produces code snippet P, then the reward assigned to π_{summary} is the final rollout score $r(P)$ defined for the main solver.

At the systems level, we place the large MoE solver policy π_{main} on a dedicated distributed GPU mesh, since it is the only component that requires both expert parallelism and long-context context parallelism. The auxiliary policies $\pi_{\text{hypothesis}}$ and π_{summary}, both implemented as smaller dense models, are served asynchronously on separate small GPU pools. This avoids fragmenting the main MoE mesh, keeps the main rollout/training pipeline saturated, and allows hypothesis and summary requests to be batched independently. Code execution, brute-force checking, and test generation are handled by a separate CPU sandbox pool.

9 RL Infrastructure

9.1 Asynchronous Training

As discussed in Section 4, we combine Agentic-GRPO with pipeline-RL strategy [25] to improve training efficiency and address the off-line policy problem.

To control the effect of earlier generated off-policy tokens, we apply a staleness weight $w(d_t)$ that down-weights tokens according to their age d_t and drops them entirely once a threshold is exceeded:

$$w(d_t) = \begin{cases} 1, & \text{if } d_t \le K_1, \\ \exp\big(-\lambda(d_t - K_1)\big), & \text{if } K_1 < d_t \le K_2, \\ 0, & \text{if } d_t > K_2. \end{cases} \tag{14}$$

where K_1 is the threshold beyond which staleness is penalized and K_2 is the hard max-lag threshold. We then weight the token-level GRPO loss as

$$L_t = \min\big(r_t A_t,\ \text{clip}(r_t, 1-\epsilon^-, 1+\epsilon^+) A_t\big) \cdot w(d_t), \tag{15}$$

where

$$r_t = \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\theta_t^{\text{beh}}}(y_t \mid x, y_{<t})} \tag{16}$$

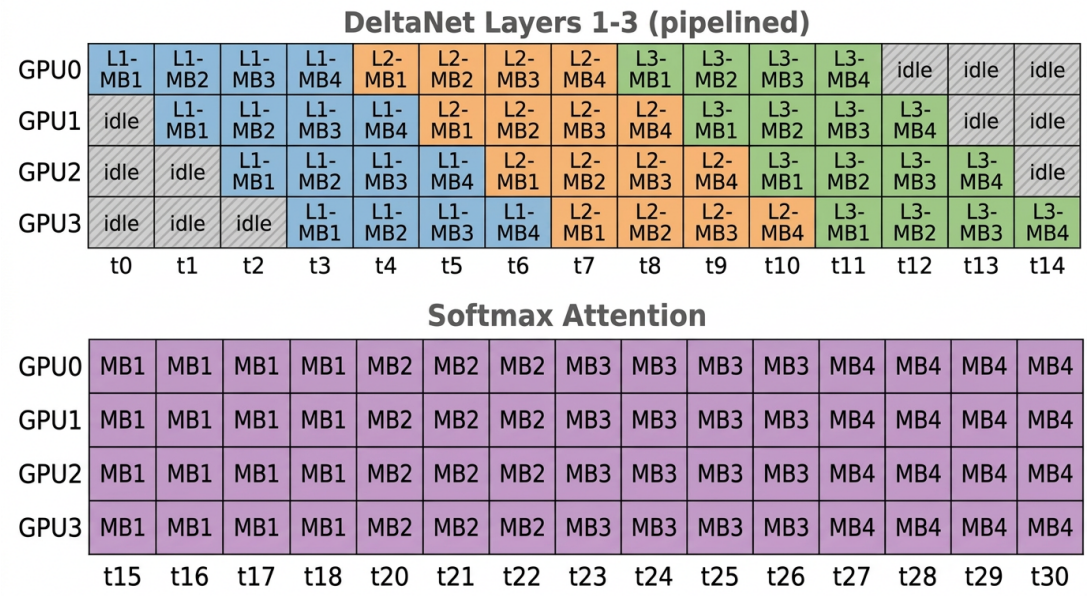


Figure 5: An illustration of pipelined context parallelism for one block with 3 DeltaNet layers (L1, L2, L3) + 1 softmax attention layer with 4 CP ranks and 4 micro-batches for illustration purposes. In the DeltaNet phase, each GPU processes micro-batches (MB) in a staggered pipeline, passing the recurrent state forward; startup and drain bubbles (gray) are confined to the triangular corners. The softmax attention layer is executed with synchronized all-to-all communication at full utilization.

is the token-level policy ratio, where θ_t^{beh} denotes the behavior-policy version that generated token y_t, whose age is d_t. It is worth noting that the token-dropping rule in Eq. 14 works naturally together with Agentic-GRPO. Intermediate rewards might be not affected, because they are used immediately when they become available. By contrast, delayed correction terms may be dropped if they arrive too late and exceed the staleness threshold.

9.2 Pipelined CP for DeltaNet+Softmax Attention Mixture

Because thinking tends to be super long for hard problems, we primarily use context parallelism (CP) to address the long-context issue.

Hybrid architectures that combine linear attention with softmax attention are becoming increasingly common, balancing the efficiency of linear attention with the stronger modeling capacity of softmax attention. For example, each block in Qwen3.5 contains three DeltaNet layers followed by one softmax attention layer. Since DeltaNet is a linear recurrent operator, standard context parallelism is not a natural fit: CP rank t cannot start processing until CP rank $t-1$ has finished and passed along the required recurrent state. Inspired by pipeline parallelism, we therefore use a pipelined CP design by pipelining multiple batches across CP ranks. Operationally, one block consists of three phases: Phase 1 pipelines the three DeltaNet layers with full overlap across micro-batches, Phase 2 inserts a synchronization barrier before attention, and Phase 3 executes the softmax attention layer. This hybrid design preserves the sequential semantics of DeltaNet while amortizing pipeline bubbles over many micro-batches. In a representative setting with 4 CP ranks and 4 micro-batches, one block achieves about 90% overall utilization, and the efficiency further improves when RL training supplies many rollout micro-batches. Figure 5 illustrates the schedule for 4 CP ranks and 4 micro-batches.

Micro-batches	DeltaNet Efficiency	Overall Efficiency	Bubble Overhead
2	67%	87%	13%
4	80%	90.3%	9.7%
8	89%	94%	$\sim$6%
16	94%	97%	$\sim$3%
N (large)	$N/(N+3)$	$\to 100\%$	$\to 0$

Table 5: Efficiency of pipelined context parallelism for the 3DeltaNet+Softmax Attention mixture as the number of micro-batches increases.

9.3 Others

Balanced Difficulty/Length in Batching Thinking length is highly correlated with problem difficulty. As a result, mixing easy and hard problem within the same batch can lead to substantial imbalance in sequence length. This is also a problem for data parallelism, since different DP workers may otherwise receive batches with very different compute costs. Therefore, in both RL training and test-time execution, we organize not only the instances within each batch to have similar difficulty levels, but also align the difficulty of batches across DP workers. At the beginning, when no runtime statistics are yet available, we batch primarily by difficulty. We then record the max completion time of its rollout, and when the same instance is revisited in later rounds, we batch it according to the measured completion time from the previous round.

Dynamic CP Because thinking length varies substantially across difficulty levels, the optimal amount of context parallelism also differs across batches. Using a single fixed CP size for all batches would therefore be inefficient: easy batches may over-parallelize short sequences, while hard batches may under-parallelize very long ones. We therefore adopt a dynamic CP strategy [10], adjusting the CP size based on the difficulty level for each batch.

Expert Routing Stability and Load Balancing To avoid routing instability during RL training, we freeze the router entirely and update only the expert feed-forward parameters. This keeps expert assignments fixed throughout RL, preventing routing drift between rollout and training and avoiding additional instability from changing expert load patterns.

10 Test-Time/Live-contest RL

At test time, or equivalently during a live contest, our strategy is centered on test-time RL [2; 39; 8].

10.1 Test-time RL v.s. Post-training RL

Although RL appears in both the post-training and test-time stages, the two settings have different objectives. *Scope:* post-training RL is used to improve the model's general competitive-programming ability across many problems, whereas Test-time RL is used only to solve the specific problem instance at hand, and it optimizes a separate set of parameters for each task instance. *Optimization target:* post-training RL optimizes *expected reward* over rollouts, denoted by

$$\max_{\theta} \mathbb{E}_{s\sim\pi_\theta(\cdot|x)}[R(s)], \tag{17}$$

while test-time RL is primarily concerned with obtaining the single best solution, characterized by the max or best-of-N reward as follows:

$$\max_{\theta} \mathbb{E}\left[\max_{i=1,\ldots,N} R(s_i)\right], \qquad s_i \sim \pi_\theta(\cdot \mid x), \tag{18}$$

accordingly, the emphasis shifts from improving average reward to finding a rollout with the highest final reward.

Stage	Accept Rate	Level 5 Solved	Weighted Score (0-100)
Post-training	72%	7/20	61.7
Full RL training	81%	13/20	72.3
After test-time RL	85%	15/20	73.5

Table 6: Progression from post-training to full RL training and then to test-time RL on the benchmark. Test-time RL is applied only when direct generation is insufficient.

10.2 Smoothing the Best-of-N Reward

Directly optimizing the exact best-of-N objective is unrealistic for RL: if one keeps only the single best rollout and discards the others, the reward becomes extremely sparse and training becomes unstable. Related work such as TTT-Discover [39] also proposes to smooth the max-style objective, using weights of the form $w_{\beta(s)}(a) = \frac{e^{\beta(s)R(s,a)}}{\mathbb{E}_{\pi_\theta(\cdot|s)}\left[e^{\beta(s)R(s,a)}\right]}$. Here, we use a simpler rank-based relaxation. Given N sampled rollouts with rewards $R^{(1)} \geq \cdots \geq R^{(N)}$ sorted by rank, we optimize

$$\mathcal{W}(\theta) = \mathbb{E}\left[\sum_{j=1}^{N} w_j(\lambda)\, R^{(j)}\right], \tag{19}$$

where the weight assigned to rank j is

$$w_j(\lambda) = \frac{e^{-\lambda j}}{\sum_{m=1}^{N} e^{-\lambda m}}. \tag{20}$$

Here λ is a hyperparameter controlling how sharply the weight concentrates on the top-ranked rollouts. When $\lambda = 0$, all weights are uniform and the objective reduces exactly to standard RL based on the average reward. As λ increases, the weight mass shifts toward the best-ranked rollouts, and the objective increasingly approximates best-of-N optimization. We gradually increase λ during training, providing a seamless transition from average-reward optimization to max-reward optimization.

10.3 Setups

At test time, we optimize LoRA parameters rather than updating the full model. Unlike post-training RL, where the context consists primarily of the problem itself, test-time RL conditions not only on the current question but also on historical candidate solutions generated for this same problem together with their scores [39; 15; 28]. This gap between RLs at post-training and test-time conditioning creates a paradigm shift, which justifies the use of LoRA as a lightweight mechanism for fast adaptation.

Summarizing RL Trials In addition to historical candidate solutions themselves, we also feed a global summary of the full search history, including which strategies have been explored, how often they have been tried, what works, and what does not. This idea is akin to the recently popular concept of skills for training LLMs [34; 11]. This summary is updated online: at each step, we combine the previous summary with the newly generated candidates to form a new summary. In the current system, this global summarization is handled by a fixed policy rather than being optimized online. This can significantly improve RL efficiency by making better use of a small number of rollouts, which is especially important in the online test-time setting where available time is limited.

10.4 Live-contest Strategies

Balancing Direct Generation and Test-Time RL At test time, we want to produce a correct submission as quickly as possible, since earlier accepted submissions receive higher scores. Because test-time RL is

expensive, we do not use it for every problem. For easier problems, such as early contest problems, we first try direct generation with larger batch size running in parallel and evaluate them. We use test-time RL only when direct generation is not enough.

$\pi_{\text{hypothesis}}$, and π_{summary} are fixed at test-time, and only π_{main} is updated.

10.5 Results

Table 6 summarizes the progression from the post-training model to full RL training and then to test-time RL. Starting from the post-training result of 72% accept rate, 7/20 solved Level 5 problems, and a weighted score of 61.7, full RL training substantially improves performance to 81%, 13/20, and 72.3, respectively. Applying test-time RL yields a further gain to 85% overall accept rate, 15/20 solved Level 5 problems, and a weighted score of 73.5. These results suggest that offline RL training already provides a large improvement in core problem-solving ability, while test-time RL is especially helpful on the hardest problems.

11 Conclusion

In this work, we introduced GrandCode, a multi-agent reinforcement learning system for agentic competitive programming. Its performance comes from two main ingredients: a coordinated agentic loop of reasoning, hypothesis generation, summarization, test-case generation, and verification; and Agentic GRPO, which addresses delayed rewards and severe off-policy drift in long, multi-stage agent rollouts.

Across offline benchmarks, continued training, supervised fine-tuning, full RL, and test-time RL all contribute meaningful gains, especially on hard problems. Under standard live contest conditions, GrandCode ranked first in all three recent Codeforces rounds in which it participated, outperforming all human contestants, including top legendary grandmasters. Taken together, these results suggest that agentic reinforcement learning, when combined with strong verification and online adaptation, can push coding systems beyond top human performance in real-time environments.

References

[1] AlphaCode Team, Google DeepMind. AlphaCode 2 technical report. Technical report, Google DeepMind, 2023. URL `https://storage.googleapis.com/deepmind-media/AlphaCode2/AlphaCode2_Tech_Report.pdf`.

[2] Anonymous. TTRL: Test-time reinforcement learning. *arXiv preprint arXiv:2504.16084*, 2025.

[3] Aaron Chan, Ahmed Shalaby, Alexander Wettig, et al. Composer 2 technical report. *arXiv preprint arXiv:2603.24477*, 2026.

[4] DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2025.

[5] Zhihao Dou, Qinjian Zhao, and Sumon Biswas. Algoforge: Specializing code generation agents through collaborative reinforcement learning. OpenReview, 2025. ICLR 2026 withdrawn submission.

[6] 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. *arXiv preprint arXiv:2502.06807*, 2025.

[7] Google DeepMind. Gemini 2.5: Our newest gemini model with thinking. `https://deepmind.google/blog/gemini-2-5-our-most-intelligent-ai-model/`, 2025. Technical blog post.

[8] Thomas Hubert, Rishi Mehta, David Silver, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 2025. doi: 10.1038/s41586-025-09833-y.

[9] IOI. International Olympiad in Informatics (IOI). https://www.ioinformatics.org/, 2026. Accessed 2026-03-25.

[10] Chenyu Jiang, Zhenkun Cai, Ye Tian, Zhen Jia, Yida Wang, and Chuan Wu. DCP: Addressing input dynamism in long-context training via dynamic context parallelism. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pp. 221–236, 2025.

[11] So Kuroki, Taishi Nakamura, Takuya Akiba, and Yujin Tang. Agent skill acquisition for large language models via CycleQD. *arXiv preprint arXiv:2410.14735*, 2024. ICLR 2025.

[12] LeetCode. LeetCode. https://leetcode.com/, 2026. Accessed 2026-03-25.

[13] Ge Li, Zhi Jin, Guang Liu, Chen Lyu, Zhihong Sun, Tao Huang, Bo-Wen Zhang, Jie Fu, and Rongao Li. TACO: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*, 2023.

[14] Xiangyang Li, Xiaopeng Li, Kuicai Dong, Quanhu Zhang, Rongju Ruan, Xinyi Dai, Xiaoshuang Liu, Shengchun Xu, Yasheng Wang, and Ruiming Tang. Humanity's last code exam: Can advanced LLMs conquer human's hardest code competition? *arXiv preprint arXiv:2506.12713*, 2025.

[15] Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving CUDA optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*, 2025.

[16] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

[17] 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. *arXiv preprint arXiv:2505.21297*, 2025.

[18] Llama Team, AI @ Meta. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[19] Zeyao Ma, Jing Zhang, Xiaokang Zhang, Jiaxi Yang, Zongmeng Zhang, Jiajun Zhang, Yuheng Jing, Lei Zhang, Hao Zheng, Wenting Zhao, Junyang Lin, and Binyuan Hui. Scaling agentic verifier for competitive coding. *arXiv preprint arXiv:2602.04254*, 2026.

[20] Moonshot AI. Kimi k2.5 release. https://www.moonshot.ai/news/kimi-k2-5-release, 2025. Technical report.

[21] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[22] OpenAI. GPT-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[23] OpenAI. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

[24] OpenAI. Openai o3 and o4-mini system card. Technical report, 2025. URL https://cdn.openai.com/pdf/2221c875-02dc-4789-800b-e7758f3722c1/o3-and-o4-mini-system-card.pdf.

[25] Alexandre Piché, Ehsan Kamalloo, Rafael Pardinas, Xiaoyin Chen, and Dzmitry Bahdanau. Pipelinerl: Faster on-policy reinforcement learning for long sequence generation. *arXiv preprint arXiv:2509.19128*, 2025.

[26] Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, Zekun Wang, Jian Yang, Zeyu Cui, Yang Fan, Yichang Zhang, Binyuan Hui, and Junyang Lin. CodeElo: Benchmarking competition-level code generation of LLMs with human-comparable elo ratings. *arXiv preprint arXiv:2501.01257*, 2025.

[27] 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. *arXiv preprint arXiv:2402.03300*, 2024.

[28] Anja Surina, Amin Mansouri, Lars Quaedvlieg, Amal Seddas, Maryna Viazovska, Emmanuel Abbe, and Caglar Gulcehre. Algorithm discovery with LLMs: Evolutionary search meets reinforcement learning. *arXiv preprint arXiv:2504.05108*, 2025.

[29] Thinking Machines Lab. Tinker. `https://github.com/thinking-machines-lab/tinker`, 2025. Open-source training API.

[30] THUDM. slime. `https://github.com/THUDM/slime`, 2026. Open-source reinforcement learning post-training framework.

[31] USACO. USA Computing Olympiad (USACO). `http://www.usaco.org/`, 2026. Accessed 2026-03-25.

[32] Zihan Wang, Jiaze Chen, Zhicheng Liu, Markus Mak, Yidi Du, Geonsik Moon, Luoqi Xu, Aaron Tua, Kunshuo Peng, Jiayi Lu, Mingfei Xia, Boqian Zou, Chenyang Ran, Guang Tian, Shoutai Zhu, Yeheng Duan, Zhenghui Kang, Zhenxing Lin, Shangshu Li, Qiang Luo, Qingshen Long, Zhiyong Chen, Yihan Xiao, Yurong Wu, Daoguang Zan, Yuyi Fu, Mingxuan Wang, and Ming Ding. AetherCode: Evaluating LLMs' ability to win in premier programming competitions. *arXiv preprint arXiv:2508.16402*, 2025.

[33] Zihan Wang, Siyao Liu, Yang Sun, Hongyan Li, and Kai Shen. CodeContests+: High-quality test case generation for competitive programming. *arXiv preprint arXiv:2506.05817*, 2025.

[34] Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, Zeyu Zheng, Cihang Xie, and Huaxiu Yao. SkillRL: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*, 2026.

[35] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[36] Lei Yang, Renren Jin, Ling Shi, Jianxiang Peng, Yue Chen, and Deyi Xiong. ProBench: Benchmarking large language models in competitive programming. *arXiv preprint arXiv:2502.20868*, 2025.

[37] Hongli Yu, Tinghong Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, and Hao Zhou. MemAgent: Reshaping long-context LLM with multi-conv RL-based memory agent. *arXiv preprint arXiv:2507.02259*, 2025.

[38] Yi Yu, Liuyi Yao, Yuexiang Xie, Qingquan Tan, Jiaqi Feng, Yaliang Li, and Libing Wu. Agentic memory: Learning unified long-term and short-term memory management for large language model agents. *arXiv preprint arXiv:2601.01885*, 2026.

[39] Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

[40] Shenyu Zheng, Ximing Dong, Xiaoshuang Liu, Gustavo Oliva, Chong Chun Yong, Dayi Lin, Boyuan Chen, Shaowei Wang, and Ahmed E. Hassan. When elo lies: Hidden biases in codeforces-based evaluation of large language models. *arXiv preprint arXiv:2602.05891*, 2026.

[41] Zhipu AI and Tsinghua University KEG. GLM-4: Open multilingual multimodal chat lms. https://github.com/zai-org/GLM-4, 2024. Project page.

A Submission Details

Figure 6 shows the standings pages together with the corresponding submission pages for the *joint* setup in the three live Codeforces contests, where all codes need to be submitted in a single account. These screenshots provide direct evidence of the contest identities, submission accounts, scores, accepted solutions, and the times at which the full problem sets were completed. The three paired screenshots correspond to Round 1087 under the ID `averyjones1`, Round 1088 under the ID `yokeko`, and Round 1089 under the ID `Vortex1`.

B Code Reward

Reward for a generated code snippet can summarized formally as follows. Let P denote a generated code snippet, and let $T = \{t_1, \ldots, t_m\}$ denote the generated test set. For each test case t_i, define the reference output

$$r_i = \begin{cases} y^\star(t_i), & \text{if a gold solution } y^\star \text{ is available,} \\ b(t_i), & \text{otherwise, where } b \text{ is a brute-force solver.} \end{cases} \tag{21}$$

Define the executability and correctness indicators by

$$E(P) = \begin{cases} 1, & \text{if } P \text{ compiles and executes,} \\ 0, & \text{otherwise,} \end{cases} \qquad C(P) = \begin{cases} 1, & \text{if } P(t_i) = r_i \text{ for all correctness-check tests,} \\ 0, & \text{otherwise.} \end{cases} \tag{22}$$

For each efficiency test case t_i, define

$$s_i(P) = \begin{cases} \dfrac{\tau_b(t_i)}{\tau_P(t_i)}, & \text{if } P \text{ finishes within the time limit,} \\ 0.1, & \text{if } P \text{ times out,} \end{cases} \tag{23}$$

where $\tau_b(t_i)$ and $\tau_P(t_i)$ denote the runtimes of the brute-force solver and the generated code snippet, respectively. The final score is then

$$R(P) = \begin{cases} 0, & \text{if } E(P) = 0, \\ 0, & \text{if } C(P) = 0, \\ \dfrac{1}{m} \sum_{i=1}^{m} s_i(P), & \text{otherwise.} \end{cases} \tag{24}$$

C Analysis on agentic-GRPO

In standard GRPO, all tokens in the full trajectory $s_1, s_2, \ldots, s_N$ are updated using the final reward r_N with a single normalized advantage:

$$A^{(i)} = \frac{r_N^{(i)} - \mu_N}{\sigma_N}, \qquad \mu_N = \frac{1}{K} \sum_{i=1}^{K} r_N^{(i)}, \qquad \sigma_N = \mathrm{std}\left(r_N^{(1)}, \ldots, r_N^{(K)}\right). \tag{25}$$

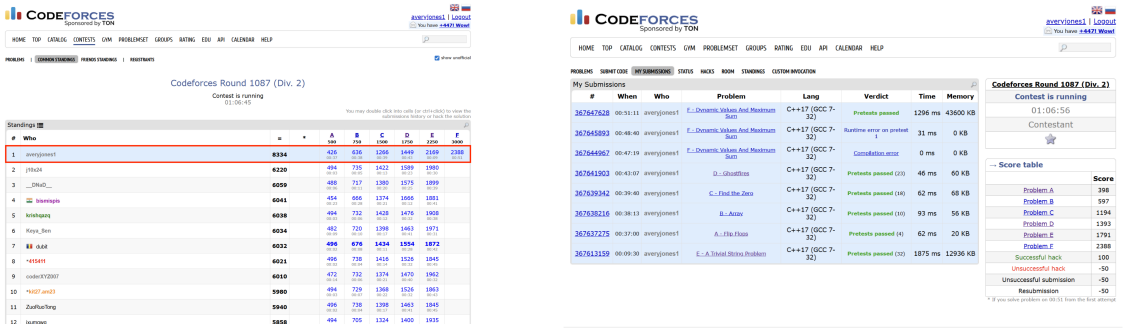
In Agentic GRPO, tokens in stage s_t receive two updates: an immediate reward advantage normalized by σ_t, and a delayed correction advantage normalized by $\sigma_{\delta_t} = \mathrm{std}(\delta_t^{(1)}, \ldots, \delta_t^{(K)})$. The unnormalized signals decompose exactly:

$$\left(r_t^{(i)} - \mu_t\right) + \left(\delta_t^{(i)} - \mu_{\delta_t}\right) = r_N^{(i)} - \mu_N, \tag{26}$$

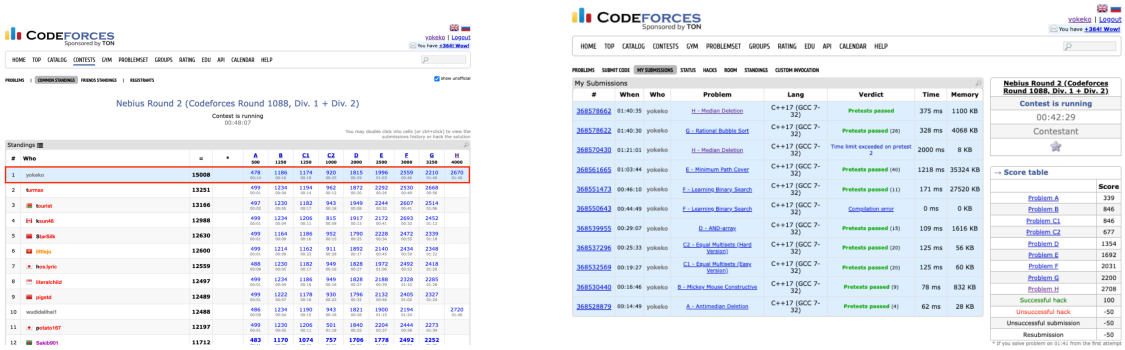
since $\mu_t + \mu_{\delta_t} = \mu_N$. However, because each phase normalizes with its own group statistics, the sum of the two normalized advantages does not recover the standard GRPO advantage:

$$\frac{r_t^{(i)} - \mu_t}{\sigma_t} + \frac{\delta_t^{(i)} - \mu_{\delta_t}}{\sigma_{\delta_t}} \neq \frac{r_N^{(i)} - \mu_N}{\sigma_N}, \tag{27}$$

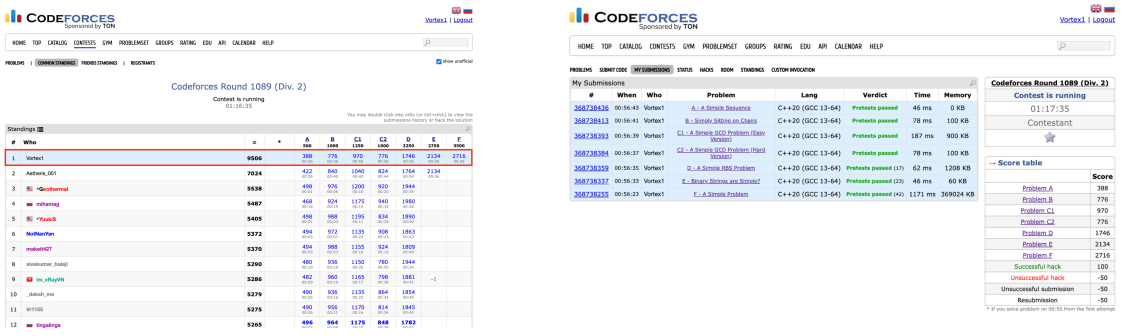
as in general $\sigma_t \neq \sigma_N \neq \sigma_{\delta_t}$. We argue that this discrepancy is a feature rather than a limitation. Independent normalization ensures that each signal as the immediate stage quality and the marginal contribution of future stages—is scaled to its own natural magnitude. If, for example, the immediate rewards r_t concentrate in a narrow range while the corrections δ_t exhibit high variance (or vice versa), a shared normalizer σ_N would let one signal dominate the other. Per-phase normalization gives each stage *equal voice* in the gradient, providing more balanced credit assignment across stages regardless of their respective reward scales.



(a) Round 1087, ID `averyjones1`.



(b) Round 1088, ID `yokeko`.



(c) Round 1089, ID `Vortex1`.

Figure 6: Standings and submission pages for GrandCode in the three live Codeforces contests. The score corresponds to $S(\text{joint})$, which is based on the full set of submissions in a single account.

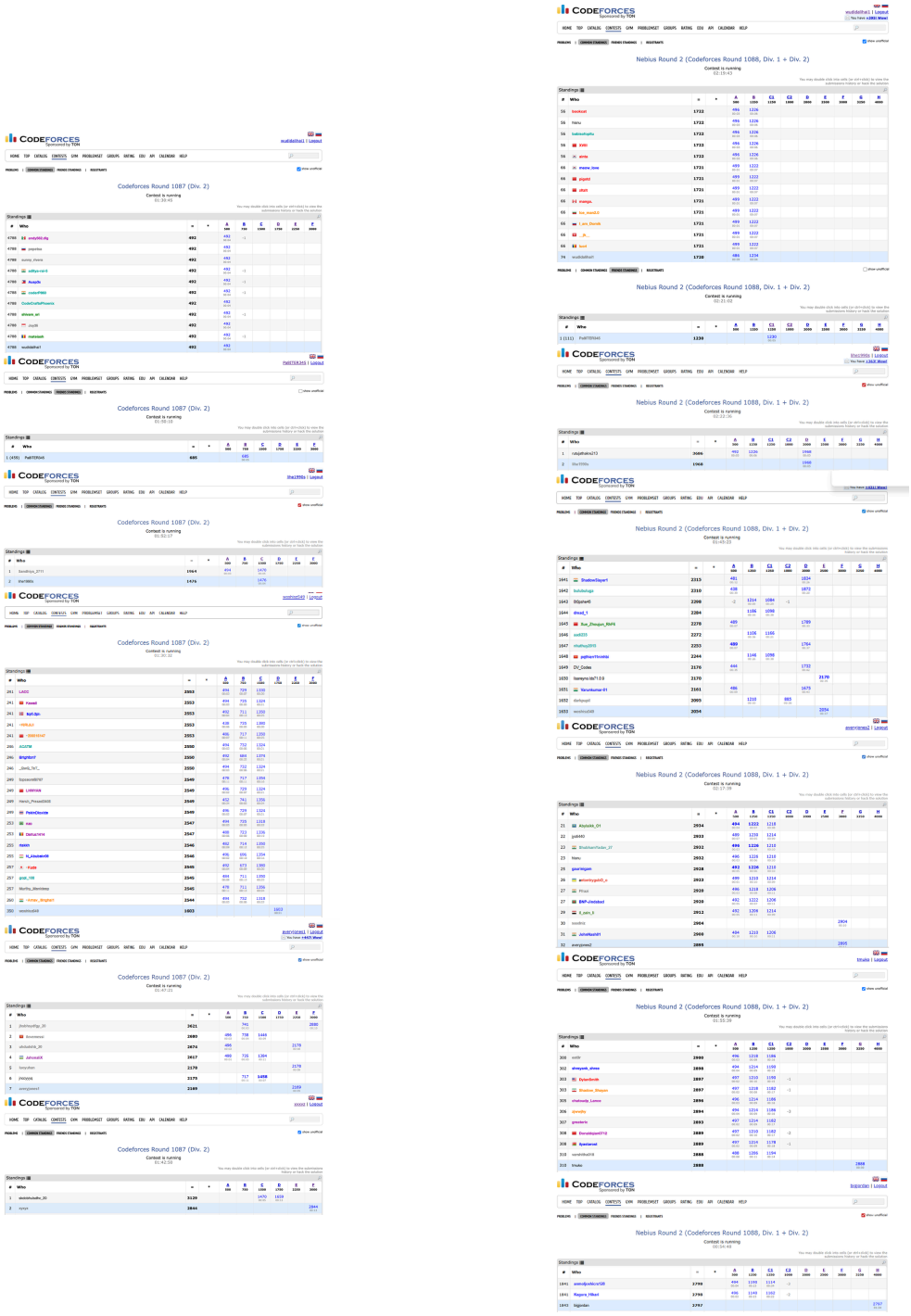


Figure 7: Standings and submission pages for GrandCode. The score corresponds to $S(\text{seperate})$, the score achieved by each solution as soon as it is ready. (a) Round 1087. A–492, B–685, C–1476, D–1603, E–2169, F–2844; $S(\text{seperate})$: 9269. (b) Round 1088. A–486, B–1234, C1–1230, C2–959, D–1968, E–2054, F–2895, G–2888, H–2797; $S(\text{seperate})$: 16511.

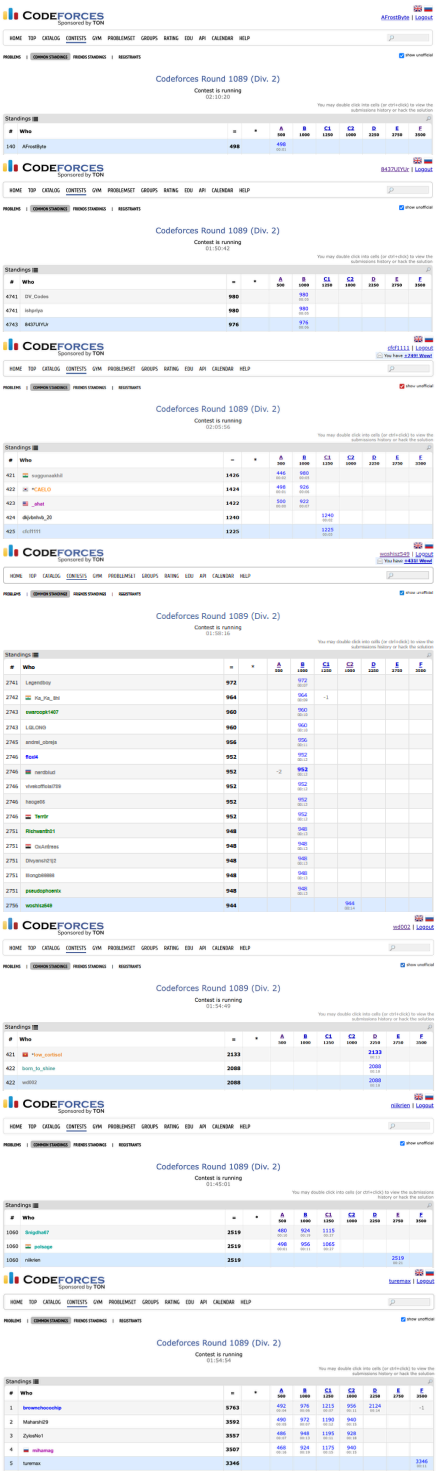


Figure 8: Standings and submission pages for GrandCode in Round 1089. The score corresponds to $S(\text{seperate})$, the score achieved by each solution as soon as it is ready. A – 498, B – 976, C1 – 1225, C2 – 944, D – 2088, E – 2519, F – 3346; $S(\text{seperate})$: 11596.

G. Toothless Time limit: 2 s | Memory: 256 MB | I/O: standard input / standard output

Let n, m, a, b be positive integers with $a \leq b$. Toothless is drawing in an $n \times m$ grid of sand that is initially all white. In one move, he may do the following:

- Select any currently white cell and color it black if it has at most one black cell among its edge-adjacent neighbors.

Because he is particular about his art, Toothless thinks some cells in the grid are *needy*, and these cells must end up black. Of the cells that aren't needy, he also thinks some of them are *special*. A cell has value b if it is special, and a otherwise. **No special cell borders a needy cell by an edge.**

Let P be the sum of the values of all the cells in the grid that **aren't** needy. Because Toothless is very particular about his art, he wants to make some number of moves such that:

- Each of the needy cells becomes black after all moves are done,
- The total value of cells that are colored black (**including** needy cells) is at least $\frac{2}{3} \cdot P$.

Compute any sequence of moves that Toothless could make. Tests are generated in a such way, that it is guaranteed that all of the needy cells in the input can be shaded black after some number of moves. Furthermore, it can be shown that for the given constraints, a satisfying sequence of moves always exists.

Input. Each test contains multiple test cases. The first line contains the number of test cases t ($1 \leq t \leq 10^4$). The description of the test cases follows. The first line of each test case contains four integers n, m, a, and b ($1 \leq n, m \leq 2 \cdot 10^3$, $1 \leq n \cdot m \leq 2 \cdot 10^3$, $1 \leq a \leq b \leq 5 \cdot 10^5$) — the dimensions of the grid and the values of non-special and special cells, respectively. The i-th of the next n lines each contain a string s_i — a string of exactly m characters depicting the i-th row of cells.

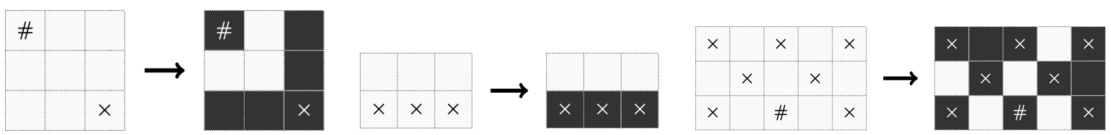
- The j-th character is '`#`' if the cell at (i, j) is needy.
- The j-th character is '`x`' if the cell at (i, j) is special.
- Otherwise, the j-th character is '`.`'.

It is guaranteed that no needy cell is adjacent to a special cell. Tests are generated in a such way, that it is guaranteed that all of the needy cells in the input can be shaded black after some number of moves. It is guaranteed that the sum of $(nm)^2$ over all test cases does not exceed $4 \cdot 10^6$.

Output. For each test case, print the following. In the first line, print the number of moves op ($0 \leq op \leq n \cdot m$) you want to make. For each of the op moves, print a line containing two integers i, j ($1 \leq i \leq n$, $1 \leq j \leq m$), indicating that you wish to color the cell at (i, j) black.

Input	Output
3 3 3 1 5 #.. ... ..x 2 3 1 2 ... xxx 3 5 8 9 x.x.x .x.x. x.#.x	6 1 1 3 1 3 2 3 3 2 3 1 3 3 2 1 2 2 2 3 10 1 1 1 2 1 3 2 2 2 5 1 5 3 5 2 4 3 3 3 1

Note. In the first test case, there is a needy cell in the top left corner and a special cell in the bottom right corner. Of the cells that aren't needy, there are 7 non-special cells worth 1 each and 1 special cell worth 5, so $P = 12$. So, we need to shade cells with a total value of at least $\frac{2}{3} \cdot 12 = 8$; the given sequence of moves achieves a value of 10. In the second test case, there are no needy cells, but there are three special cells in the bottom row of cells. There are 3 non-special cells worth 1 each and 3 special cells worth 2, so $P = 9$. So, we need to shade cells with a total value of at least $\frac{2}{3} \cdot 9 = 6$; the given sequence of moves achieves a value of 6. In the third test case, there is one needy cell at $(3, 3)$, and 7 special cells. Of the cells that aren't needy, there are 7 non-special cells worth 8 each and 7 special cells worth 9, so $P = 7 \cdot 8 + 7 \cdot 9 = 119$. So, we need to shade cells with a total value of at least $\frac{2}{3} \cdot 119 = 79\frac{1}{3}$; the given sequence of moves achieves a value of 87.



(a) Test 1 (b) Test 2 (c) Test 3

Figure 9: Problem G. Toothless (Round 1088, Div. 1+2): complete statement, sample I/O, and example illustrations. Each panel shows the grid before (left) and after (right) moves; dark cells are colored black.

D Submitted Code

Round 1089-A

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int a;
    cin >> a;
    while (a--) {
        int b;
        cin >> b;
        vector<int> c(b + 1), d(b + 1);
        for (int i = 1; i <= b; i++) {
            cin >> c[i];
            d[c[i]] = i;
        }

        int e = 0;
        int f = 0;
        int g = 1;
        while (g <= b) {
            if (d[g] < g) e++;
            int h = g - e;
            if (h > f) f = h;
            g++;
        }

        cout << f << '\n';
    }

    return 0;
}
```

Round 1088-E

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef unsigned long long u64;

#if defined(_WIN32)||defined(_WIN64)
#define getchar_unlocked getchar
#endif

struct R {
    template<class T>
    bool read(T& x) {
        int c = getchar_unlocked();
        if (c == EOF) return false;
        while (c <= ' ') {
            c = getchar_unlocked();
            if (c == EOF) return false;
        }
        T v = 0;
        while (c > ' ') {
            v = v * 10 + (T)(c - '0');
            c = getchar_unlocked();
        }
```

```
        x = v;
        return true;
    }
};

struct S {
    static const int INF = 1e9;

    struct Fr {
        int a;
        u64 b;
        int c;
        int d;
        bool e;
    };

    int n;
    vector<u64> a;
    vector<vector<int>> b;
    vector<int> c;
    vector<int> d;
    vector<vector<pair<u64,int>>> e;
    vector<Fr> f;

    S(int n) : n(n), a(n+1,0), b(n+1), c(n+1,0), d(n+1,0), e(n+1) {}

    int fc(int u, u64 g) {
        auto& v = e[u];
        for (int i = 0; i < (int)v.size(); i++)
            if (v[i].first == g) return i;
        return -1;
    }

    int qr(int s, u64 g0) {
        if (g0 == 1) return INF;
        int p = fc(s, g0);
        if (p != -1) return e[s][p].second;

        f.clear();
        f.push_back({s, g0, 0, 0, false});

        while (!f.empty()) {
            Fr& cur = f.back();
            int cc = fc(cur.a, cur.b);
            if (cc != -1) { f.pop_back(); continue; }

            if (!cur.e) {
                if (gcd(a[cur.a], cur.b) == 1ULL) {
                    e[cur.a].push_back({cur.b, INF});
                    f.pop_back();
                    continue;
                }
                cur.e = true;
                cur.c = 0;
                cur.d = 0;
            }

            bool pushed = false;
            auto& ch = b[cur.a];
            while (cur.c < (int)ch.size()) {
                int v = ch[cur.c];
                u64 ng = gcd(cur.b, a[v]);
                if (ng == 1ULL) { cur.c++; continue; }
                int ci = fc(v, ng);
                if (ci == -1) {
                    f.push_back({v, ng, 0, 0, false});
```

```
                    pushed = true;
                    break;
                }
                cur.d = min(cur.d, e[v][ci].second - c[v] - 1);
                cur.c++;
            }
            if (pushed) continue;

            e[cur.a].push_back({cur.b, d[cur.a] + cur.d});
            f.pop_back();
        }

        p = fc(s, g0);
        return e[s][p].second;
    }
};

int main() {
    R in;
    int t;
    in.read(t);

    while (t--) {
        int n;
        in.read(n);

        S sv(n);
        vector<int> ind(n+1, 0);
        long long ec = 0;
        bool bad = false;

        for (int u = n; u >= 1; u--) {
            int k;
            in.read(sv.a[u]);
            in.read(k);
            ec += k;

            sv.b[u].reserve(k);
            int bs = 1;

            for (int i = 0; i < k; i++) {
                int v;
                in.read(v);
                sv.b[u].push_back(v);
                if (v <= u || v > n) bad = true;
                else {
                    ind[v]++;
                    if (ind[v] > 1) bad = true;
                }
                bs += sv.c[v];
            }

            sv.d[u] = bs;
            sv.e[u].reserve(4);
            sv.c[u] = sv.qr(u, sv.a[u]);

            printf("%d\n", sv.c[u]);
            fflush(stdout);
        }

        if (ec != n-1) bad = true;
        if (ind[1] != 0) bad = true;
        for (int i = 2; i <= n; i++) {
            if (ind[i] != 1) { bad = true; break; }
        }

```

```
        if (bad) {
            printf("-1\n");
            fflush(stdout);
            return 0;
        }
    }

    return 0;
}
```

Round 1088-F

```
#include <bits/stdc++.h>
using namespace std;

const int P = 676767677;

long long pw(long long a, long long b) {
    long long r = 1;
    for (; b > 0; b >>= 1) {
        if (b & 1) r = r * a % P;
        a = a * a % P;
    }
    return r;
}

struct A {
    vector<long long> a, b;
    A(int n) : a(n+1), b(n+1) {
        a[0] = 1;
        for (int i = 1; i <= n; i++)
            a[i] = a[i-1] * i % P;
        b[n] = pw(a[n], P-2);
        for (int i = n; i >= 1; i--)
            b[i-1] = b[i] * i % P;
    }
    long long c(int n, int k) const {
        if (n < 0 || k < 0 || k > n) return 0;
        return a[n] * b[k] % P * b[n-k] % P;
    }
};

struct B {
    int a, b, c;
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int t;
    cin >> t;
    vector<pair<int,int>> q(t);
    int mx = 0;
    for (int i = 0; i < t; i++) {
        cin >> q[i].first >> q[i].second;
        mx = max(mx, q[i].first + q[i].second);
    }

    A cb(mx);

    for (int i = 0; i < t; i++) {
        int n = q[i].first, m = q[i].second;
        long long base = cb.c(n+m-1, m-1);
```

```
        long long ans = 0;

        vector<B> st;
        st.push_back({1, n, 1});

        while (!st.empty()) {
            B cur = st.back();
            st.pop_back();

            int l = cur.a, r = cur.b, d = cur.c;
            int mid = (l + r) >> 1;
            int ls = mid - l;
            int rs = r - mid;
            int ss = r - l + 1;

            long long w = base;
            if (l > 1) w -= cb.c(n+m-ls-2, m-1);
            if (r < n) w -= cb.c(n+m-rs-2, m-1);
            if (l > 1 && r < n) w += cb.c(n+m-ss-2, m-1);
            w %= P;
            if (w < 0) w += P;

            ans = (ans + w * d) % P;

            if (l <= mid-1) st.push_back({l, mid-1, d+1});
            if (mid+1 <= r) st.push_back({mid+1, r, d+1});
        }

        cout << ans % P << '\n';
    }

    return 0;
}
```