CodeHacker: Automated Test Case Generation for Detecting Vulnerabilities in Competitive Programming Solutions

Anonymous ACL submission

Abstract

The evaluation of Large Language Models (LLMs) for code generation relies heavily on the quality and robustness of test cases. However, existing benchmarks often lack coverage for subtle corner cases, allowing incorrect solutions to pass. To bridge this gap, we propose **CodeHacker**, an automated agent framework dedicated to generating targeted adversarial test cases that expose latent vulnerabilities in program submissions. Mimicking the *hack* mechanism in competitive programming, CodeHacker employs a multi-strategy approachincluding stress testing, anti-hash attacks, and logic-specific targeting to break specific code submissions. To ensure the validity and reliability of these attacks, we introduce a Calibration Phase, where the agent iteratively refines its own Validator and Checker via self-generated adversarial probes before evaluating contestant code. Experiments demonstrate that CodeHacker significantly improves the True Negative Rate (TNR) of existing datasets, effectively filtering out incorrect solutions that were previously accepted. Furthermore, generated adversarial cases prove to be superior training data, boosting the performance of RL-trained models on benchmarks like LiveCodeBench. All code, datasets, and evaluation scripts will be open-sourced to promote further investigation in LLMs for competitive programming.

1 Introduction

Large Language Models (LLMs, OpenAI, 2024; DeepSeek-AI, 2025) are commonly used for code generations and program reasoning. Several benchmarks have been developed to evaluate whether LLMs can understand problem constraints correctly, derive correct programs logically, and produce executable implementations (Chen et al., 2021a; Li et al., 2022a; Wang et al., 2025a). One challenge is that the evaluation outcome for each LLM can depend sensitively on the quality and coverage of each test case, raising the questions of how test cases should be constructed and validated properly (Liu et al., 2023a).

A substantial amount of effort has been applied to address this challenge. One line of work focuses on constructing larger and more rigorous benchmarks, thereby reducing bias introduced by sparse test cases. For example, LiveCodeBench (Jain et al., 2025) applies stricter evaluation protocols and contamination control, while CodeContests (Li et al., 2022b) and CodeContest+ (Wang et al., 2025c) target competition-level algorithmic problems with more challenging test settings. Another type of work considers automated test case generation (Li et al., 2023a; He et al., 2025; Zhou et al., 2025), attempting to expand test coverage and reach more potential error patterns through input mutation, rule-based construction, or language model generation. Several studies also rely on human experts to analyze program logic and construct targeted counterexamples or adversarial cases, achieving strong discriminative power at the cost of scalability (Ma et al., 2025; Wang et al., 2025b). However, we argue that these existing automated approaches operate fundamentally as Black-box Fuzzers. They rely on probabilistic exploration of the input space, prioritizing statistical coverage to maximize the likelihood of hitting a bug. Their evaluation outcomes often still hinge on whether their test cases happen to cover the corresponding failure cases, rather than on whether the model truly understands the problems semantics and constraint structure.

Increasing the number of test cases or expanding the input space in a coarse-grained manner is insufficient for boosting the reliability of the evaluation. What ultimately matters is whether test cases can target the specific logical vulnerabilities of a concrete implementation (Li et al., 2023b; He et al., 2025). In competitive programming practice, constructing a valid counterexample that

causes a program to fail (i.e., a successful hack) is often no easier than solving the original problem itself: it requires a deep understanding of algorithmic assumptions, boundary conditions, and complexity constraints, which usually demands fine-grained reasoning about program behavior (Wang et al., 2025b). From this perspective, the ability to generate failure-inducing adversarial inputs reflects not only a models understanding of program behavior, but also provides a more discriminative signal for assessing advanced program reasoning capabilities.

While prior work often relies on human experts to generate effective counterexamples (i.e., hacks), we consider the following question: ***can such expert-level hacking be automated?*** We introduce **CodeHacker**, a program-centric adversarial framework for competitive programming. Unlike methods that rely on heuristic test generation, CodeHacker adopts a Code-Aware Adversarial perspective. CodeHacker treats individual programs as first-class objects and systematically searches for failure-inducing counterexamples, addressing the limitations of existing evaluation approaches in terms of both targeting and scalability. Unlike methods that rely on static test cases or heuristic test generation, our framework adopts a program-centric adversarial evaluation perspective, tightly coupling test construction with the actual execution behavior of the code under evaluation. Our main contributions are summarized as follows:

1. We formalize hacking as an adversarial test generation task and design an LLM-driven agent that actively searches for high-value corner cases and logical counterexamples that are difficult to capture with traditional mutation-based or prompt-based generation methods.

2. By deploying the agent on existing code datasets, we demonstrate substantial improvements in both true negative rate (TNR) and true positive rate (TPR), indicating that the generated adversarial cases meaningfully strengthen evaluation robustness.

3. Building on the generated adversarial cases, we further propose CodeHackerBench, a new evaluation setting designed to better characterize models ability to reason about incorrect code and extreme failure scenarios.

2 Related Work

Code Benchmark. The existing code datasets mainly adopt three paradigms. *Manual* test cases such as MBPP (Austin et al., 2021), HumanEval (Chen et al., 2021b) and LiveCodeBench (Jain et al., 2025) are typically handcrafted. Expert-designed cases can better target problem-specific corner cases; manual construction is expensive, hard to automate, and difficult to scale, thus more suitable for small evaluation sets than large training corpora. *Mutation-based* approaches aim to improve coverage by automatically recombining or mutating existing inputs (type-aware input mutation to reduce false positives on MBPP/HumanEval (Liu et al., 2023a), and similar strategies in CodeContests (Li et al., 2022b)), but they may violate complex input constraints, potentially introducing invalid tests and raising false negatives. Finally, *LLM-based* generation produces tests conditioned on problem statements (e.g., Taco Li et al., 2023a) and has also been explored in several generators/meta-benchmarks (e.g., EvalPlus, HardTests, TestCaseEval, LogiCase, and TCGBench Liu et al., 2023b; He et al., 2025; Cao et al., 2025; Sung et al., 2025; Ma et al., 2025) ; yet LLM outputs are not guaranteed to satisfy intricate constraints and are limited by context/output length, making them less suitable for very large or highly structured instances (e.g., million-node graphs).

Adversarial Data in Competitive Programming. Prior work (Hort and Moonen, 2025) introduced *Codehacks*, a dataset constructed by mining historical Codeforces hacks. Due to the inaccessibility of the original victim submissions via public APIs, this dataset relies on post-hoc matching based on observed execution failures. Mutation- or randomly generated test approaches expose false negatives but do not model *adversarial intent* (Liu et al., 2023b, 2025a).

RLHF and RLVR. PPO (Schulman et al., 2017) is a standard RLHF optimizer based on on-policy rollouts and value-function estimation. GRPO (Shao et al., 2024) removes the explicit critic via group-based baselines, and the following works (Yu et al., 2025b; Liu et al., 2025b) further improves training stability. Recent work increasingly adopts RL with verifiable (RLVR) for code generation, often using GRPO-style method (Ekbote et al., 2025; Pennino et al., 2025).

3 Method

3.1 Problem Formulation

We consider a Codeforces-style competitive programming environment where an LLM-based agent plays the role of an attacker that attempts to *hack* existing submissions by generating adversarial test cases.

Let $\mathcal{P}$ denote the set of problems and, for each problem $p \in \mathcal{P}$, let $\mathcal{X}_p$ be its input space and $\mathcal{Y}_p$ the corresponding output space. A contestant submission (program) s is expected to map an input to an output in $\mathcal{Y}_p$, but it may fail during execution. We denote these execution failures as **Runtime Error (RE)**, **Time Limit Exceeded (TLE)**, and **Memory Limit Exceeded (MLE)**. Formally, the submission induces a (partial) mapping:

$$s : \mathcal{X}_p \to \mathcal{Y}_p \cup \{\text{RE}, \text{TLE}, \text{MLE}\}.$$

The online judge implements an evaluation function to verify the submission. If the program executes successfully and produces an output that satisfies the problem's correctness criteria (verified by a standard comparator or a special judge), the verdict is **Accepted (AC)**; otherwise, if the output is rejected as incorrect, the verdict is **Wrong Answer (WA)**. Combined with the execution failures defined above, the evaluation function returns a verdict from the set:

$$J_p(s, x) \in \{\text{AC}, \text{WA}, \text{RE}, \text{TLE}, \text{MLE}\}.$$

We treat all verdicts other than **AC** as failures of the target submission under the given test input.

LLM agent as a hacking policy. An LLM agent with parameters θ interacts with a problem–submission pair (p, s) and generates test inputs in order to trigger a non-AC verdict. We model the agent as a (possibly stochastic) policy

$$\pi_\theta : \mathcal{H}_{p,s} \to \mathcal{X}_p,$$

where $\mathcal{H}_{p,s}$ denotes the interaction history (including the problem statement, the system feedback, and previously proposed tests). At step t, given history $h_t \in \mathcal{H}_{p,s}$, the agent proposes an input

$$x_t \sim \pi_\theta(\cdot \mid h_t),$$

which is evaluated by the judge to obtain $v_t = J_p(s, x_t)$. The interaction proceeds for at most T steps or until a non-AC verdict is observed.

Hack success and success rate. For a fixed problem–submission pair (p, s) and a given agent π_θ, we define the *hack success indicator*

$$H(p, s; \pi_\theta) = \mathbb{I}\big[\exists\, t \le T : J_p(s, x_t) \neq \text{AC}\big]. \quad (1)$$

where $\mathbb{I}[\cdot]$ is the indicator function and the randomness is induced by the policy π_θ (and, if applicable, the environment). Intuitively, the hack succeeds if the agent can find at least one test input within T trials that causes the submission to receive a non-AC verdict.

Given a dataset $\mathcal{D}$ of problem–submission pairs, the overall *hack success rate* (HSR) of the agent is defined as

$$\text{HSR}(\pi_\theta) = \frac{1}{|\mathcal{D}|} \sum_{(p,s)\in\mathcal{D}} \mathbb{E}\big[H(p, s; \pi_\theta)\big],$$

where the expectation is taken over the internal randomness of the LLM agent and, if relevant, the judging system. In our experiments, we empirically approximate $\text{HSR}(\pi_\theta)$ by averaging the observed hack success indicators over all evaluated pairs $(p, s) \in \mathcal{D}$.

3.2 CodeHacker Agent

The CodeHacker is an LLM-based agent responsible for generating adversarial test cases designed to exploit weaknesses in a given program. It interacts with the problem statement and the contestants submission to generate inputs that aim to trigger non-AC verdicts such as Wrong Answer (WA), Runtime Error (RE), Time Limit Exceeded (TLE), or Memory Limit Exceeded (MLE).

3.3 Phase I: Evaluation Tool Calibration

Before attempting to hack contestant submissions, CodeHacker must ensure its "ammunition" (test cases) matches problem constraints and its "judgment" (checker) is flawless. We employ an iterative refinement process.

3.3.1 Validator Refinement

The Validator is the gatekeeper that ensures test cases lie within the allowed input space Φ. Similar to the Checker, the agent actively attacks the Validator from two directions to identify weaknesses:

- **Bypass Attack (x_{invalid}):** The agent generates an explicitly invalid input x_{invalid} (e.g., inputs violating constraints like $N > N_{\max}$ or malformed formatting). If the Validator returns *Accepted*, it exposes a False Positive

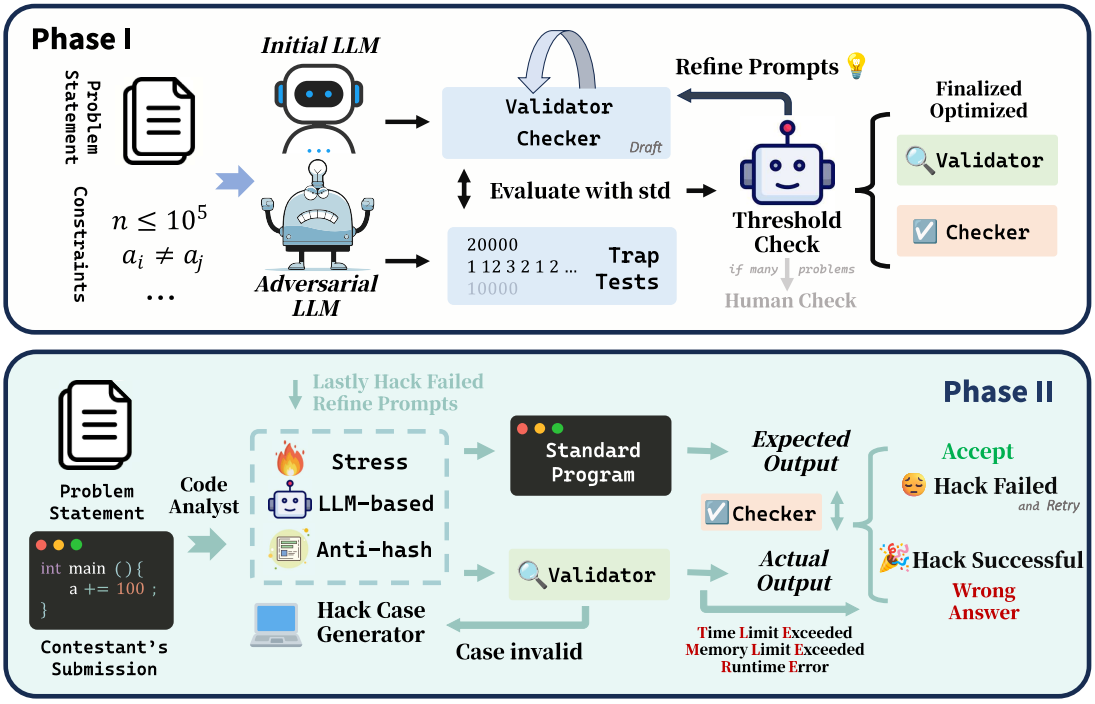

Figure 1: The overall architecture of the CodeHacker framework. **Phase I (Evaluation Tool Calibration):** The agent iteratively refines the judging infrastructure to ensure reliability. This process begins by refining the **Validator** to strictly enforce input constraints, followed by refining the **Checker** to eliminate false verdicts. **Phase II (Adversarial Case Generation):** Utilizing the calibrated tools, the Code Analyst guides three distinct generation strategies (Stress, LLM-based, and Anti-hash) to synthesize adversarial test cases that expose specific vulnerabilities in the contestant's submission.

flaw (the validator is too loose and fails to catch illegal inputs).

- **Rejection Attack (x_{valid}):** The agent generates a valid but tricky input x_{valid} (e.g., legitimate edge cases like $N = 1$, values equal to 0, or maximum constraints). If the Validator returns *Rejected*, it exposes a False Negative flaw (the validator is too strict and incorrectly flags valid inputs).

If the Validator fails in either case, it is patched to strictly align with the problem definition. The adversarial refinement process is detailed in **Algorithm 1** (Appendix B).

3.3.2 Checker Refinement

The Checker verifies the correctness of the programs output. In the refinement loop, the agent acts as a deceptive adversary, probing for two specific types of vulnerabilities:

- **Deception Attack (y_{wrong}):** The agent constructs an incorrect output y_{wrong} that mimics the structure of a valid answer (e.g., correct formatting but wrong value, or satisfying only partial constraints). If the Checker returns *Accepted*, it exposes a False Positive flaw (the checker is too loose).
- **Rejection Attack (y_{true}):** The agent identifies a valid output y_{true} (often an edge case or an alternative valid solution different from the standard reference) that a rigid checker might miss. If the Checker returns *Wrong Answer*, it exposes a False Negative flaw (the checker is too strict).

If either attack succeeds, the Checker is flagged as compromised and subsequently updated using the adversarial samples. The complete dual-attack workflow is outlined in **Algorithm 2** in Appendix B.

Expert Intervention for Complex Verification Logic. While our automated refinement pipeline is highly effective, we observed a small fraction of cases where the LLM struggles to generate correct Validators or Checkers. This typically occurs when the verification logic itself embodies significant algorithmic difficulty. For Validators, constraints may require proving the existence of a solution (e.g., "guarantee the graph has a Hamilto-

nian path"); for Checkers, verifying a contestant's output can be equivalent to solving a secondary hard problem (e.g., checking graph isomorphism or computing geometric intersections with high precision). Since implementing such logic exceeds the typical one-shot capability of LLMs, we employ human expert intervention for these rare instances to manually implement or patch the evaluation tools, ensuring strict adherence to problem semantics.

3.4 Phase II: Adversarial Case Generation

Armed with robust evaluation tools (the calibrated Validator and Checker), CodeHacker proceeds to the execution phase. This phase aims to generate adversarial inputs that specifically trigger non-AC verdicts in the target submission.

3.4.1 Code Analyst

The Code Analyst serves as the strategist. Relying solely on static textual analysis often leads to hallucinations regarding complexity limits or logical edge cases. To ensure rigorous vulnerability identification, we equip the Analyst with a *Dual-Execution Interface*, enabling a hybrid analysis workflow:

- **Behavioral Probing (via C++ Sandbox):** The agent treats the target submission as a black box to verify logical hypotheses. For instance, if the agent suspects the code fails on disconnected graphs, it synthesizes a small probe input (e.g., two isolated nodes) and executes the target binary. Observing a crash (RE) or an incorrect output confirms the bug without guessing.
- **Precise Calculation (via Python Interpreter):** The agent utilizes Python as a computational engine to verify constraints and mathematical properties. This includes: (1) Complexity Verification: Writing scripts to calculate the exact worst-case operation count (e.g., $\sum_{i=1}^{N} \lfloor N/i \rfloor$ for harmonic series) to confirm TLE risks; (2) Boundary Checks: Computing whether specific intermediate values (e.g., C_{2N}^{N}) exceed the 64-bit integer limit to confirm overflows;

By combining behavioral observation with mathematical verification, the Code Analyst formulates a highly reliable hacking plan.

Note on Design: Distinct from the Analyst's minimal logic probes (e.g., $N = 5$), the Generator is essential for *weaponizing* these insights into large-scale hacks (e.g., $N = 10^5$). By outputting executable C++ code rather than raw data, the Generator bypasses LLM context constraints to produce massive, format-compliant datasets.

3.4.2 Hack Case Generators

To maximize the diversity and effectiveness of the attacks, we employ three complementary generation strategies:

Generated by Stress Test: This module randomly or systematically explores the boundary of the input space. It focuses on maximizing input size (N_{max}) or creating specific structural patterns (e.g., fully connected graphs, skewed trees). While naturally testing asymptotic behavior, this strategy is particularly effective at exposing Runtime Errors (e.g., stack overflows, index out-of-bounds) and Wrong Answers caused by integer overflows or precision issues under extreme constraints.

Generated by LLM: Guided by the Code Analyst's plan, the LLM crafts targeted semantic test cases. This method is versatile and capable of targeting all verdict types. Depending on the vulnerability identified by the Analystwhether it is an unhandled hack case (WA), a deep recursion risk or segmentation fault (RE), a sub-optimal algorithmic branch (TLE), or inefficient space management (MLE), the LLM synthesizes inputs specifically designed to trigger the corresponding failure mode.

Generated by Anti-Hash Generator: This module targets submissions utilizing hashing algorithms. For fixed-parameter Polynomial Rolling Hashes, we formulate the collision search as a Shortest Vector Problem (SVP) and apply lattice reduction algorithms (e.g., LLL) (Sugar_fan, 2024) to deterministically construct collisions. Additionally, for scenarios with smaller moduli or non-linear hash functions where lattice reduction is inapplicable, we employ a **Birthday Attack** strategy. By generating a large pool of random inputs, we exploit the *Birthday Paradox* to identify collisions with high probability in $\mathcal{O}(\sqrt{M})$ complexity, where M is the modulus.Mathematical proofs and implementation details are provided in **Appendix H**.

3.5 CodeHackerBench

The **CodeHackerBench** dataset is constructed by mining judgment discrepancies within the

CodeContest$^+$ dataset. By cross-referencing the ground-truth correctness labels against the verdicts returned by the original (weaker) test cases, we identified approximately 6,000 controversial problem-submission pairs.

To ensure the absolute reliability of this benchmark, we implemented a Two-Tier Human Verification Protocol covering all critical components (Hack Cases, Validators, and Checkers):

- **Infrastructure Audit (Refined Tools):** We manually reviewed all of the Refined Validators and Checkers modified during the calibration phase. This ensures that our judging logic serves as the gold standard, strictly adhering to the official problem definitions.
- **Hack Case Verification (Data Sampling):** For the Generated Hack Cases, we conducted a random sampling of 600 instances. These cases were inspected by expert competitive programmers, achieving a 100% validity rate (confirming they adhere to input constraints) and correctly triggering failure modes in the target submissions.

4 Experiments

Dataset. We randomly selected 2000 problems from the CodeContest+ dataset, consisting of:

- **1000 traditional judge problems:** These problems rely on the standard online judging systems (Traditional Judge), where the system typically evaluates the solution by comparing the output against the expected results.
- **1000 special judge problems:** These problems use a special judging mechanism (Special Judge), which involves custom evaluation logic. This often includes additional checks such as output formatting, time limits, or other domain-specific rules.

In our experiments, we specifically focused on the C++ code of the selected problems as C++ is the most widely used among current competitive programming contests for its high efficiency. Additionally, for RL-based evaluations, we used **LiveCodeBench**, consisting of 287 problems from AtCoder. The models were evaluated based on the pass@k metrics and pass@1, pass@5 indicate the percentage of problems solved by the model within the top-k attempts.

Metrics. We evaluate the quality of our benchmark and agent using four key metrics. Detailed formal definitions and calculation methods are provided in **Appendix E**. *(1) True Positive Rate (TPR)* and *(2) True Negative Rate (TNR)*: Measure the discriminative power of the test cases against the ground truth labels. *(3) Validation Pass Rate (VPR)*: Measures the proportion of test cases in a dataset that satisfy the strict problem constraints (validity). Additionally, we report the *Hack Success Rate (HSR)* to quantify agent performance.

Implementation Details. We randomly selected 2,000 problems from CodeContest$^+$. Crucially, to ensure verdict fidelity, we strictly replicated the official Codeforces Windows-based toolchain (MinGW) (Codeforces, 2023). For the generation modules, we set the sampling temperature to 0.7, preventing the agent from collapsing into repetitive patterns and encouraging a broader exploration of the input space. For Reinforcement Learning, we train Qwen3-4B using the DAPO algorithm (Yu et al., 2025a). Complete implementation details, including exact compiler flags, infrastructure setups, and RL hyperparameters, are detailed in **Appendix C** and **Appendix D**.

4.1 Main Results

Adversarial hacking acts as a discriminator for advanced reasoning. Table 2 shows that DeepSeek V3.2 achieves the highest HSR of 64.83%, outperforming GPT-5-Mini (51.40%) and Gemini-3.0 (35.65%). Crucially, we observe a dramatic performance gap when explicit reasoning is disabled: DeepSeek V3.2's success rate plummets to 23.76% (a 2.7× decline) in non-thinking mode. This substantial performance gap between reasoning and non-reasoning models underscores a critical finding: **Hacking is a Reasoning Task**. Complex adversarial inputs such as specific tree topologies to maximize recursion depth or mathematically precise sequences cannot be retrieved from memory or guessed. They require the model to perform *constructive reasoning*: deriving a generative algorithm that satisfies multiple entangled constraints. This explains why standard LLMs fail to generate valid complex hacks, whereas reasoning-enhanced models excel.

Adversarial hacking corrects metric inflation and enhances evaluation rigor. To accurately assess the impact of adversarial test cases, we constructed a new benchmark variant,

Dataset	Overall	Traditional Judge		Special Judge	
	VPR (% ↑)	TPR (% ↓)	TNR (% ↑)	TPR (% ↓)	TNR (% ↑)
CodeContests (Li et al., 2022c)	71.41	98.96	76.33	84.73	77.69
HardTests (He et al., 2025)	97.32	98.33	79.25	86.56	75.37
TACO (Li et al., 2023b)	81.84	96.46	83.70	82.67	85.48
CodeContest^{+} (Wang et al., 2025c)	99.66	96.02	85.72	96.62	84.04
CodeContest^{++} (Ours)	**100.00**	**95.86**	**96.31**	**96.38**	**96.05**

Note: To ensure evaluation fairness: (1) All test cases were pre-filtered by our **refined validator** to exclude invalid inputs; (2) Results in the **Special Judge** columns were evaluated using our **refined checker** to avoid false verdicts caused by original weak checkers.

Table 1: Comparison of Overall Validation Pass Rate (VPR) of original datasets, and correctness metrics (TPR/TNR) across 2000 problems with among different datasets.

Backbone Model	HSR (%)	Avg. #T
Reasoning Models		
DeepSeek V3.2	**64.83**	1.24
GPT-5-Mini	51.40	1.35
Qwen3-Next-80B-A3B	38.37	1.58
Gemini-3.0-Flash-Preview	35.65	1.62
Qwen3-32B	23.64	1.89
Non-Reasoning Models		
DeepSeek V3.2*	**23.76**	2.45
Qwen3-Next-80B-A3B*	20.06	2.68
Qwen3-32B*	10.00	2.91

Table 2: Hack performance across different models. **HSR**: Hack Success Rate. **Avg. #T**: The average number of refinement turns required to successfully hack a submission by directly LLM generated (lower is better, max 5). The symbol * indicates non-thinking mode for hybrid-reasoning models.

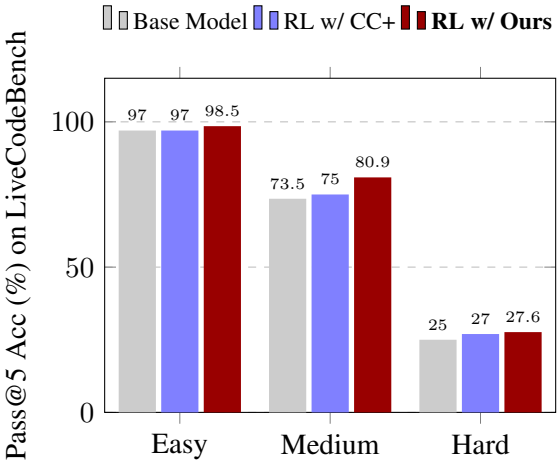

Figure 2: Pass@5 performance comparison on LiveCodeBench. Models trained with adversarial data show consistent improvements. The model trained on our augmented subset achieves the highest accuracy across all difficulty levels.

CodeContests^{++}. We posit that under the strict premises of input validity and verdict correctness (guaranteed by our refined validator and checker), the hallmark of a rigorous benchmark is a significantly higher TNR coupled with a moderately lower TPR. As shown in Table 1, unlike baselines with inflated TPRs (≈99%) that often mask subtle flaws, CodeHacker achieves a superior TNR (e.g., **96.05%** on Special Judge problems). This performance confirms that our framework successfully corrects historical misclassifications: the observed drop in TPR is not a degradation but a correction of *inflated* metrics, exposing latent bugs in previously "Accepted" solutions rather than incorrectly rejecting valid code.

Adversarial data improves RL efficiency and generalization. Table 2 shows that the model RL-trained on the augmented `CodeContests`$^{++}$ consistently outperforms the baseline trained on standard data (see Appendix G for detailed numerical breakdowns). This indicates that high-quality adversarial inputs serve as dense reward signals during reinforcement learning, forcing the policy to optimize for strict boundary condition handling and deep logical robustness rather than simple pattern matching. Crucially, the performance gains on the out-of-distribution LiveCodeBench confirm that learning from hack cases fosters genuine algorithmic reasoning capabilities rather than overfitting to the training set.

4.2 Ablation Study

To rigorously evaluate the contributions of our framework, we conduct a two-fold ablation study. First, we analyze the impact of individual agent modules on the Hack Success Rate (HSR), determining which components are essential for generating valid attacks. Second, we examine

the Sanitation Pipeline step-by-step to understand how each refinement stage improves the reliability (TNR and TPR) of the final benchmark.

Impact of Agent Components We investigate the necessity of the Code Analyst, Refinement Loop, and Stress Test modules by removing them one by one from the CodeHacker agent. As shown in Table 3, the full CodeHacker agent achieves the highest success rate (51.40% using GPT-5-mini).

Variant	Hack Success Rate (%↑)
CodeHacker (Full)	**51.40**
w/o Code Analyst	49.86
w/o Stress Test	50.12
w/o Anti-Hash Generator	51.00
w/o Refinement Loop	46.60

Table 3: Ablation study of CodeHacker agent components.

The Refinement Loop proves to be the most critical component; its removal causes a significant performance drop ($51.40\% \rightarrow 46.60\%$). This highlights the difficulty of generating strictly valid adversarial cases in a single passiterative feedback is essential for correcting invalid inputs. The Code Analyst also plays a vital role ($49.86\% \rightarrow 51.40\%$) by guiding the generator toward logical vulnerabilities rather than blind guessing. The Stress Test module provides a minor but necessary boost ($50.12\% \rightarrow 51.40\%$) by covering asymptotic complexity failures that semantic analysis might miss.

It is worth noting that while the Anti-Hash Generator was triggered in only a small fraction of cases (approximately 0.4% of submissions using rolling hashes), it achieved a 100% success rate in breaking these solutions. This confirms that while niche, domain-specific adversarial modules are essential for achieving perfect coverage against algorithmic shortcuts.

Impact of Dataset Enhancement Pipeline We further analyze how our infrastructure improvementsspecifically the tool calibration (Validator/Checker) and the adversarial data augmentation (Hack Cases) affect evaluation reliability. To isolate the impact of judging logic, we conduct this ablation study specifically on the 1000 problems requiring Special Judges. Table 4 details the cumulative effect of the refined validator, refined checker, and the addition of hack cases.

An interesting phenomenon occurs when introducing the refined validator: the TNR slightly drops from 82.18% to 82.08%. This reveals a *masking effect* in the baseline, where invalid inputs previously caused incorrect solutions to fail, inadvertently counting them as correctly rejected. By filtering these invalid inputs, the refined validator exposes the baseline's true, weaker discriminatory power.

Configuration	TNR (%↑)	TPR (%)
Baseline (CodeContest$^+$)	82.18	95.34
+ Refined Validator	82.08	95.50
+ Refined Checker	84.04	96.62
+ Hack Cases (Ours)	**96.05**	96.38

Table 4: Stepwise analysis of the dataset enhancement pipeline on Special Judge problems. Adding adversarial Hack Cases provides the decisive boost to TNR.

While the refined checker improves verdict precision (raising TPR to 96.62%), the most substantial improvement in robustness comes from the augmentation with hack cases. This step leaps the TNR to 96.05%, confirming that standard test cases lack the coverage to catch subtle logical errors, and that adversarial generation is non-negotiable for rigorous evaluation.

5 Conclusion

In this work, we introduced **CodeHacker**, an autonomous framework that iteratively refines competitive programming evaluations via a novel Self-Hacking mechanism. By correcting legacy judging flaws and augmenting the test cases with targeted adversarial inputs, our augmented subset achieves highest True Negative Rate (TNR), offering a significantly more trustworthy standard than existing benchmarks. Furthermore, we presented **CodeHackerBench**, a benchmark designed to evaluate the *adversarial reasoning* capabilities of LLMs. Our experiments demonstrate that the ability to generate valid hacks serves as a strong differentiator for advanced reasoning models, mirroring the cognitive demands of rigorous algorithm design. We hope this work establishes adversarial generation as a critical aspect for evaluating general algorithmic intelligence. Beyond academic evaluation, we envision CodeHacker as a vital component of the next-generation competitive programming infrastructure, assisting both problem setters in quality assurance and contestants in personalized training.

Ethic Statement

We emphasize two critical ethical principles regarding the research and application of automated hacking agents, strictly adhering to the community standards of the competitive programming ecosystem:

Prohibition of Unauthorized Data Scraping. While platforms like Codeforces and QOJ publicly display hack attempts, they explicitly prohibit the use of automated web crawlers or scrapers to harvest such data in bulk. Unauthorized scraping violates the Terms of Service of these platforms and places an unsustainable load on community-maintained servers. Our research strictly respects these prohibitions; we urge future researchers to avoid unauthorized data harvesting and to rely solely on officially released datasets or compliant data access methods to preserve platform integrity.

Mandatory Local Evaluation. It is fundamentally impermissible to evaluate AI agents by submitting generated test cases to public Online Judges. Using public judging queues for automated testing constitutes a Denial of Service (DoS) risk and degrades the service quality for human contestants. Therefore, it is mandatory that all adversarial evaluations be conducted in a local, sandboxed environment. CodeHacker is specifically designed with a Self-Hacking loop to operate entirely offline, ensuring zero interference with live judging infrastructure.

Limitations

While our approach has shown promising results, there are several limitations that need to be addressed in future work. While the CodeHacker agent is capable of generating valuable test cases, it is not infallible. There may still be certain edge cases or subtle algorithmic flaws that the agent fails to identify. Expanding the agents capability to detect more complex failures, especially those related to performance (e.g., time or memory limitations), remains an open challenge. Besides, our current evaluation focuses on a subset of programming languages and problem domains. Expanding the CodeHackerBench benchmark to include additional languages and a broader set of problem types will provide a more comprehensive assessment of the approachs generalizability. As competitive programming and algorithmic challenges evolve, we plan to adapt our framework to ensure it stays up-to-date with the latest trends and challenges in the field.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Yuhan Cao, Zian Chen, Kun Quan, Ziliang Zhang, Yu Wang, Xiaoning Dong, Yeqi Feng, Guanzhong He, Jingcheng Huang, Jianhao Li, Yixuan Tan, Jiafu Tang, Yilin Tang, Junlei Wu, Qianyu Xiao, Can Zheng, Shouchen Zhou, Yuxiang Zhu, Yiming Huang, and 2 others. 2025. Can llms generate reliable test case generators? a study on competition-level programming problems. *Preprint*, arXiv:2506.06821.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021a. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021b. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Codeforces. 2023. Codeforces command lines. `https://codeforces.com/blog/entry/121114`.

DeepSeek-AI. 2025. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Chanakya Ekbote, Vijay Lingam, Behrooz Omidvar-Tehrani, Jun Huan, Sujay Sanghavi, Anoop Deoras, and Stefano Soatto. 2025. Murphy: Multi-turn grpo for self correcting code generation. *Preprint*, arXiv:2511.07833.

Zhongmou He, Yee Man Choi, Kexun Zhang, Jiabao Ji, Junting Zhou, Dejia Xu, Ivan Bercovich, Aidan Zhang, and Lei Li. 2025. Hardtests: Synthesizing high-quality test cases for llm coding. *Preprint*, arXiv:2505.24098.

Max Hort and Leon Moonen. 2025. Codehacks: A dataset of adversarial tests for competitive programming problems obtained from codeforces. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 742–746. IEEE.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*.

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

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

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 dAutume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022a. Competition-level code generation with alphacode. *Science*, 378(6624):10921097.

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 dAutume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022b. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

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 dAutume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022c. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572. Curran Associates, Inc.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023b. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Kaibo Liu, Zhenpeng Chen, Yiyang Liu, Jie M. Zhang, Mark Harman, Yudong Han, Yun Ma, Yihong Dong, Ge Li, and Gang Huang. 2025a. LLM-powered test case generation for detecting bugs in plausible programs. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 430–440, Vienna, Austria. Association for Computational Linguistics.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. 2025b. Understanding r1-zero-like training: A critical perspective. In *2nd AI for Math Workshop @ ICML 2025*.

Zihan Ma, Taolin Zhang, Maosongcao, Junnan Liu, Wenwei Zhang, Minnan Luo, Songyang Zhang, and Kai Chen. 2025. Rethinking verification for LLM code generation: From generation to testing. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

OpenAI. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Federico Pennino, Bianca Raimondi, Massimo Rondelli, Andrea Gurioli, and Maurizio Gabbrielli. 2025. From reasoning to code: Grpo optimization for underrepresented languages. *Preprint*, arXiv:2506.11027.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *Preprint*, arXiv:1707.06347.

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

Sugar_fan. 2024. A tool for hacking rolling hashes with fixed modulos and bases. `https://codeforces.com/blog/entry/129538`.

Sicheol Sung, Aditi, Dogyu kim, Yo-Sub Han, and Sang-Ki Ko. 2025. Logicase: Effective test case generation from logical description in competitive programming. *Preprint*, arXiv:2505.15039.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025a. Openhands: An open platform for ai software developers as generalist agents. *Preprint*, arXiv:2407.16741.

Zihan Wang, Jiaze Chen, Zhicheng Liu, Markus Mak, Yidi Du, Geonsik Moon, Luoqi Xu, Aaron Tua, Kunshuo Peng, Jiayi Lu, and 1 others. 2025b. Aethercode: Evaluating llms' ability to win in premier programming competitions. *arXiv preprint arXiv:2508.16402*.

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

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2025a. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, and 17 others. 2025b. DAPO: An open-source LLM reinforcement learning system at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Shang Zhou, Zihan Zheng, Kaiyuan Liu, Zeyu Shen, Zerui Cheng, Zexing Chen, Hansen He, Jianzhu Yao, Huanzhi Mao, Qiuyang Mang, and 1 others. 2025. Autocode: Llms as problem setters for competitive programming. *arXiv preprint arXiv:2510.12803*.

A Terms and Definitions

To ensure clarity and consistency throughout this paper, we provide formal definitions for standard terms used in the competitive programming ecosystem. The evaluation of a solution typically involves a rigorous interaction between the contestant's code and the Online Judge (OJ) system's components. Understanding the precise roles of the *Validator*, *Checker*, and the mechanics of a *Hack* is essential for grasping the workflow of our CodeHacker framework. Table 5 summarizes these key definitions and their specific contexts within this work.

B Refinement Algorithms

We provide the detailed pseudocode for the iterative refinement of the Validator and Checker below.

C Codeforces C++ Compilation Environments

To ensure full reproducibility and consistency with competitive-programming settings, we faithfully replicated all official Codeforces C++ compiler configurations. Contrary to common Linux-based setups, Codeforces officially employs a Windows-based judging infrastructure. Consequently, we strictly utilized the official flags (e.g., `-Wl,-stack=268435456`) to prevent discrepancies in TLE or RE verdicts.

Algorithm 1 Iterative LLM-based Refinement of Validator

Require: Problem constraints Φ, validation prompt P_{val}, hacking prompt $P_{\text{val-hack}}$, LLM $\mathcal{M}$, max failures K
Ensure: Validator V

```
V ← M(P_val, Φ)
k_fail ← 0
while k_fail < K do
    (x_valid, x_invalid) ← M(P_val-hack, Φ, V)
    is_FP ← (V(x_invalid) == Accepted)
    is_FN ← (V(x_valid) ≠ Accepted)
    if is_FP ∨ is_FN then
        k_fail ← 0
        Update P_val with failure cases x_invalid and x_valid
        V ← M(P_val, Φ)
    else
        k_fail ← k_fail + 1
    end if
end while
return V
```

Algorithm 2 Iterative LLM-based Refinement of Checker

Require: Problem spec Φ, checking prompt P_{c}, hacking prompt $P_{\text{c-hack}}$, oracle $\mathcal{O}$, LLM $\mathcal{M}$, max failures K
Ensure: Checker C

```
C ← M(P_c, Φ)
k_fail ← 0
while k_fail < K do
    (x_cand, y_wrong, y_true) ← M(P_c-hack, Φ, C)
    is_FP ← (C(x_cand, y_wrong) == Accepted)
    is_FN ← (C(x_cand, y_true) ≠ Accepted)
    if is_FP or is_FN then
        k_fail ← 0
        y_gt ← O(Φ, x_cand)
        Update P_c with failure case (x_cand, y_wrong, y_true, y_gt)
        C ← M(P_c, Φ)
    else
        k_fail ← k_fail + 1
    end if
end while
return C
```

D RL Training Hyperparameters

We employed the **DAPO** algorithm (Yu et al., 2025a) to train the Qwen3-4B base model. To accommodate the lengthy context often required in competitive programming (e.g., long problem statements and complex logic), we set the maximum generation length to 24,000 tokens. The training was conducted with a global batch size of 32 and an actor learning rate of 5×10^{-7}. During the exploration phase, we sampled $N = 8$ solutions per prompt. We adopted a dual clipping strategy with $\epsilon_{\text{low}} = 0.2$ and $\epsilon_{\text{high}} = 0.3$. The reward is rule-based: the model receives a reward of $+1$ if the generated code successfully passes all

Table 5: Common Terms in Competitive Programming

Term	Description
Submission	In programming competitions, a submission is a program submitted by a contestant. This submission is evaluated by a judging system, resulting in a verdict such as *Accepted (AC)*, *Wrong Answer (WA)*, *Time Limit Exceeded (TLE)*, *Runtime Error (RE)*, or *Compile Error (CE)*, among others.
Test case	A test case is used to check whether the participants submission is correct. It usually consists of input data and the corresponding correct output (reference answer).
Test input	The input data of a test case will be fed into the contestants program. The programs output will then be compared with the reference output to determine correctness.
Test output	The output data of a test case is the correct answer corresponding to the input data. For problems with multiple correct solutions, the reference output typically represents one of the possible correct answers.
Validator	A validator is a program used to check whether the input data satisfies the problem constraints and format.
Generator	A generator is a program used to produce test input automatically, especially when the input data is large, random, or complex and cannot be easily handcrafted.
Checker	A checker is a program used to determine if a contestants output is correct. Usually, it simply compares the contestants output with the expected output, but for problems with multiple solutions (e.g., floating-point answers or different valid orderings), it may include more complex judging logic.
Hack	In *Codeforces*, a hack is a special mechanism available during the hacking phase of some rounds. Participants can test other contestants solutions by providing a test case that causes them to fail. If a hack succeeds, the hacker gains additional points, and the hacked participant may lose points or their solution becomes marked as *hacked*. Hacks help ensure solution robustness and encourage writing well-tested code.

Compiler	Arch	Version	Compilation Command
GNU G++14	32-bit	6.4.0	`-static -DONLINE_JUDGE -Wl,-stack=268435456 -O2 -std=c++14`
GNU G++17	32-bit	7.3.0	`-static -DONLINE_JUDGE -Wl,-stack=268435456 -O2 -std=c++17`
GNU G++20	64-bit	11.2.0 (WinLibs)	`-Wall -Wextra -Wconversion -static -DONLINE_JUDGE -Wl,-stack=268435456 -O2 -std=c++20`
GNU G++23	64-bit	14.2 (MSYS2)	`-Wall -Wextra -Wconversion -static -DONLINE_JUDGE -Wl,-stack=268435456 -O2 -std=c++23 -lstdc++exp`

Table 6: C++ compilation environments reproduced from the official Codeforces judge configuration.

test cases, and 0 otherwise.

We provide the detailed hyperparameters used for the DAPO training in Table 7. All experiments were conducted on a cluster of NVIDIA H100 GPUs.

Hyperparameter	Value
Algorithm	DAPO
Base Model	Qwen3-4B
Train Batch Size	32
Samples per Prompt (N)	8
Actor Learning Rate	5×10^{-7}
Max Generation Length	24,000
Clip Range ($\epsilon_{\text{low}}, \epsilon_{\text{high}}$)	0.2, 0.3

Table 7: Hyperparameters for Reinforcement Learning (DAPO).

E Metric Definitions and Calculations

To ensure the reproducibility and clarity of our evaluation, we provide the formal definitions for all metrics used in the experimental section.

Let $\mathcal{D}$ be the dataset of problem-submission pairs. For a specific submission $s \in \mathcal{D}$, let $J_{\text{orig}}(s)$ denote the verdict provided by the original dataset (ground truth), and $J_{\text{new}}(s)$ denote the verdict obtained using our augmented test cases (CodeHacker generated cases).

We define the binary correctness function $V(s)$ as:

$$V(\text{verdict}) = \begin{cases} 1(\text{Correct}) & \text{if verdict is } \textbf{Accepted} \\ 0(\text{Incorrect}) & \text{otherwise.} \end{cases}$$

E.1 True Positive Rate (TPR) and True Negative Rate (TNR)

These metrics evaluate the discriminative power of the test case with respect to the original ground truth labels.

True Positive Rate (TPR): The proportion of solutions originally marked as correct that are also accepted by the new test case.

$$\text{TPR} = \frac{\sum_{s \in \mathcal{D}_{\text{pos}}} \mathbb{I}[V(J_{\text{new}}(s)) = 1]}{|\mathcal{D}_{\text{pos}}|}$$

where $\mathcal{D}_{\text{pos}} = \{s \in \mathcal{D} \mid V(J_{\text{orig}}(s)) = 1\}$. A TPR significantly lower than 100% implies that the new test case has successfully hacked solutions that were previously thought to be correct (exposing False Positives in the original data).

True Negative Rate (TNR): The proportion of solutions originally marked as incorrect that are correctly rejected by the new test case.

$$\text{TNR} = \frac{\sum_{s \in \mathcal{D}_{\text{neg}}} \mathbb{I}[V(J_{\text{new}}(s)) = 0]}{|\mathcal{D}_{\text{neg}}|}$$

where $\mathcal{D}_{\text{neg}} = \{s \in \mathcal{D} \mid V(J_{\text{orig}}(s)) = 0\}$. A low TNR indicates that a test case is too weak, allowing incorrect solutions to pass (False Positives).

E.2 Validation Pass Rate (VPR)

VPR measures the quality of the test cases themselves, specifically ensuring they adhere to problem constraints. Let T_{orig} be the set of all test cases in the original dataset. Let $\mathcal{V}(t)$ be our **Refined Validator**, which returns 1 if an input t satisfies all problem constraints and 0 otherwise.

$$\text{VPR} = \frac{\sum_{t \in T_{\text{orig}}} \mathcal{V}(t)}{|T_{\text{orig}}|} \times 100\%$$

A VPR less than 100% indicates that the original dataset contained invalid test cases that were subsequently filtered out in our benchmark.

E.3 Hack Success Rate (HSR)

HSR measures the agent's ability to trigger a failure in a submission. For a specific set of target submissions S_{target}:

$$\text{HSR} = \frac{\sum_{s \in S_{\text{target}}} \mathbb{I}[\exists x \in X_{\text{gen}} : J(s, x) \neq \text{AC}]}{|S_{\text{target}}|}$$

where X_{gen} is the set of adversarial inputs generated by the agent for submission s.

F Data Decontamination and Benchmark Independence

To ensure the validity of our reinforcement learning results and strictly prevent data leakage, we implemented a **Source-Based Decontamination Protocol**.

Source Separation. Our training and evaluation datasets are derived from disjoint algorithmic contest platforms, ensuring structural independence:

- **Training Set (`CodeContests`$^{++}$):** The augmented training data is exclusively sourced from **Codeforces** problems.
- **Evaluation Set (LiveCodeBench):** For the evaluation of RL-trained models, we specifically utilized the subset of LiveCodeBench consisting entirely of **AtCoder** problems.

Since Codeforces and AtCoder are distinct platforms with unique problem sets, checking systems, and editorial styles, there is effectively **zero overlap** between our training and evaluation data. This physical separation guarantees that the performance improvements observed in Table **??** are due to the generalization of adversarial reasoning capabilities rather than memorization of specific problem patterns.

G Detailed Reinforcement Learning Results

In this section, we provide the comprehensive numerical results for the Reinforcement Learning evaluation on LiveCodeBench. Table 8 details the pass@1 and pass@5 accuracy across three difficulty levels (Easy, Medium, Hard).

H Mathematical Derivation of Anti-Hash Generator

In this section, we detail the construction used to generate hash collisions for polynomial rolling hashes with fixed moduli $\{p_i\}_{i=0}^{n-1}$ and bases $\{q_i\}_{i=0}^{n-1}$.

H.1 Problem Definition

The hash function for a string a of length L is defined as:

$$h_i(a) = \left(\sum_{j=0}^{L-1} a_j q_i^j \right) \pmod{p_i}$$

Our goal is to find two distinct strings a and b such that $h_i(a) = h_i(b)$ for all i. This is equivalent to finding a difference array $d = a - b$ satisfying:

1. $\sum_{j=0}^{L-1} d_j q_i^j \equiv 0 \pmod{p_i}$ for all i.
2. $|d_j| < 26$ (assuming lowercase English letters) for all j, and $d \neq 0$.

H.2 Lattice Construction

We formulate this as a Shortest Vector Problem (SVP). We construct a lattice basis matrix M of size $(n + L) \times (n + L)$. To prioritize satisfying the modular equations (i.e., forcing the remainder to be 0), we introduce a large weight factor λ.

The matrix M is defined as a block matrix:

$$M = \begin{bmatrix} \lambda Q & I \\ \lambda P & 0 \end{bmatrix}$$

where:

- Q is an $L \times n$ matrix where $Q_{ji} = q_i^j \pmod{p_i}$.
- I is an $L \times L$ identity matrix representing the coefficients of d.
- P is an $n \times n$ diagonal matrix where $P_{ii} = p_i$, representing the moduli constraints.
- 0 is an $n \times L$ zero matrix.

H.3 Reduction and Reconstruction

The rows of M span a lattice. A vector v in this lattice has the form:

$$v = (\lambda \cdot R, d)$$

where R represents the remainder of the polynomial evaluation modulo p_i. We aim to find a vector where $R = 0$ (meaning the hash difference is a multiple of the modulus) and d is small (non-zero but within character range).

By multiplying the top-left and bottom-left blocks by a sufficiently large λ, we penalize any non-zero remainder R. We then apply the L^2 **reduction algorithm** (a variant of LLL) to reduce

Training Stage	Easy pass@1	Easy pass@5	Medium pass@1	Medium pass@5	Hard pass@1	Hard pass@5
Base Model (Qwen3-4B)	91.04	97.01	63.24	73.53	11.84	25.00
RL w/ CodeContests^{+}	94.03	97.01	58.82	75.00	15.13	26.97
RL w/ CodeContests^{++}	**94.03**	**98.51**	**66.18**	**80.88**	**17.11**	**27.63**

Table 8: Detailed Pass@k performance on LiveCodeBench across different difficulty levels. Comparing the Base Model against versions trained via RL using standard vs. adversarial datasets.

the basis M. The shortest vector in the reduced basis typically yields a vector with $R = 0$ and a valid difference array d. From d, we construct strings a and b by setting $a_j = \max(0, d_j)$ and $b_j = \max(0, -d_j)$ (shifted by a base character), guaranteeing a collision.

H.4 Birthday Attack Strategy

While Lattice Reduction is effective for polynomial rolling hashes with large moduli (e.g., 2^{64}), it requires the hash function to have a specific linear structure. For generic hash functions or smaller moduli (e.g., $M \approx 10^9$), we utilize a probabilistic approach based on the **Birthday Paradox**.

The Birthday Paradox states that in a set of n randomly chosen elements, where each element is drawn uniformly from a domain of size M, the probability that at least two elements are identical (a collision) is approximately:

$$P(\text{collision}) \approx 1 - e^{-\frac{n(n-1)}{2M}} \approx 1 - e^{-\frac{n^2}{2M}}$$

To achieve a collision probability of $P \approx 0.5$ (50%), the required number of generated test cases n is:

$$n \approx \sqrt{2M \ln 2} \approx 1.177\sqrt{M}$$

For a typical 32-bit integer hash ($M \approx 4 \times 10^9$), the agent only needs to generate approximately $n \approx 75,000$ inputs to find a collision with high confidence. This is computationally trivial for the CodeHacker agent.

Implementation: The agent generates two sets of random strings, S_A and S_B. It computes the hash values for all strings in S_A and stores them in a hash map. Then, it computes hashes for strings in S_B and checks for existence in the map. This *meet-in-the-middle* approach efficiently discovers collisions for single-hash checks commonly found in weaker solutions.

I Case Study

I.1 A Weak Checker

In this section, we analyze a weak checker from the problem *Codeforces 25_B*. The problem allows multiple valid output formats, making strict string comparison insufficient.

I.1.1 Problem: Phone Numbers

Description: Given a string of n digits ($2 \leq n \leq 100$), divide it into groups of length 2 or 3, separated by hyphens ('-').
Output Requirements:

- The output must contain only digits and hyphens.
- Every group must have a length of exactly 2 or 3.
- The concatenation of the groups (removing hyphens) must exactly match the original string.

I.1.2 Vulnerability Analysis

The original weak checker (Figure 3) attempts to validate the output by splitting the string by hyphens. However, it lacks robust parsing logic for edge cases:

1. It may crash or misbehave if the output contains consecutive hyphens or leading/trailing hyphens.
2. It fails to rigorously check for non-digit characters (e.g., whitespace or letters) in some implementations.

Our refined checker (Figure 4) performs character-level validation, ensuring that no illegal characters exist and that the structure strictly follows the grouping rules before content verification.

I.2 A Wrong Validator

In this section, we examine a validator flaw in *Codeforces 309_C*. The validator restricts input values more strictly than the problem statement allows, causing valid test cases to be rejected.

```cpp
#include "testlib.h"
#include <bits/stdc++.h>
using namespace std;
void readAndCheckAnswer(InStream& stream, const string& pn, const string& who) {
    string line = stream.readLine();
    // Remove spaces
    line.erase(remove(line.begin(), line.end(), ' '), line.end());
    // Split line into groups on '-'
    vector<string> groups;
    stringstream ss(line);
    string group;
    while (getline(ss, group, '-')) {
        groups.push_back(group);
    }
    // For each group, check that length is 2 or 3 and consists of digits
    for (size_t i = 0; i < groups.size(); ++i) {
        if (groups[i].length() < 2 || groups[i].length() > 3) {
            stream.quitf(_wa, "%s: group %d has invalid length %zu (should be 2 or 3)", who.c_str(), int(i + 1), groups[i].length());
        }
        if (!all_of(groups[i].begin(), groups[i].end(), ::isdigit)) {
            stream.quitf(_wa, "%s: group %d contains non-digit characters", who.c_str(), int(i + 1));
        }
    }
    // Concatenate all groups and check if it matches pn
    string reconstructed;
    for (const auto& g : groups) {
        reconstructed += g;
    }
    if (reconstructed != pn) {
        stream.quitf(_wa, "%s: concatenated groups do not match original phone number", who.c_str());
    }
}
int main(int argc, char* argv[]) {
    registerTestlibCmd(argc, argv);
    int n = inf.readInt(2, 100);
    string pn = inf.readToken(format("[0-9]{%d}", n), "phone number");
    readAndCheckAnswer(ans, pn, "jury's answer");
    readAndCheckAnswer(ouf, pn, "participant's answer");
    quitf(_ok, "Correct answer");
}
```

Figure 3: Weak Checker in $CodeContest^{+}$ for problem Codeforces 25_B

I.2.1 Problem: Memory Management

Description: You are given an array A of N integers and an array B of M integers. Each element $b_j \in B$ represents a memory block of size 2^{b_j}.

Constraints:

- $1 \leq N, M \leq 10^6$.
- Elements of A: $1 \leq a_i \leq 10^9$.
- Elements of B: $0 \leq b_j \leq 29$. (Crucially, $2^0 = 1$ is valid).

I.2.2 Vulnerability Analysis

The original validator (Figure 5) incorrectly enforces the range $[1, 60]$ for array B. This logic explicitly forbids $b_j = 0$, effectively disallowing memory blocks of size $2^0 = 1$. However, the problem statement allows $b_j = 0$. This False Negative error prevents the system from testing edge cases involving the smallest unit of memory, potentially masking bugs in solutions that fail to handle size 1 blocks.

Figure 5 shows the original validator code in CodeContest^{+}. In this validator, the program checks the value of each element in b to ensure that $2^{b_j} \leq 10^9$. However, the range for b_j is incorrectly limited to $[1, 60]$, excluding the case where $b_j = 0$. This results in the rejection of valid test cases where $b_j = 0$.

In our revised version of the validator, shown in Figure 6, we have updated the validator to allow for $b_j = 0$ by adjusting the input range for b_j from

$[1, 60]$ to $[0, 29]$. This change ensures that the case where $b_j = 0$ is now accepted, thus allowing $2^0 = 1$ to pass the validation. This adjustment corrects the original validator's oversight and ensures that all valid test cases are processed correctly.

This case study illustrates the importance of properly defining the valid input ranges and corner cases in the validator. The original error highlights how minor oversights in constraint handling can lead to unnecessary test failures, affecting the overall reliability of the contest system. By fixing this, we ensure that all valid edge cases are properly accepted, making the validator more robust and reliable.

I.3 Another Wrong Validator

This case study uses *Codeforces 177_C2* to illustrate a discrepancy where the validator is too permissive compared to the problem statement, potentially leading to Time Limit Exceeded (TLE) on valid solutions.

I.3.1 Problem: Party

Description: A social network with N people. There are k pairs of friends and m pairs of enemies. Find the maximum size of a valid party group.

Constraints:

- $N \leq 2000$.
- $k \leq 100,000$ (The number of friendship pairs).
- The graph of friends must be simple (no self-loops, no duplicate edges).

I.3.2 Vulnerability Analysis

The problem statement explicitly limits k to $100,000$. However, the maximum possible edges in a graph of $N = 2000$ nodes is $N(N-1)/2 \approx 2 \times 10^6$. The original validator (Figure 7) calculates the theoretical maximum edges (`max_edges`) and allows k to go up to this value. This means the validator accepts test cases with up to 2 million edges, far exceeding the stated limit of 100,000. Solutions optimized for $k = 10^5$ (e.g., using adjacency lists) might unexpectedly TLE when fed 2×10^6 edges.

In our revised version of the validator, shown in Figure 8, we have updated the validator to properly handle the k constraint. Specifically, we now enforce that k cannot exceed 100000, as stated in the problem description. This ensures that the validator only accepts valid test cases and prevents the Time Limit Exceeded (TLE) errors caused by excessively large values of k.

This case study highlights the importance of aligning the validator with the problem's constraints to avoid unnecessary errors. By ensuring that k does not exceed the value of 100,000, as specified in the problem description, we ensure that all valid test cases are accepted while preventing performance issues and TLE errors. This adjustment improves the reliability of the validator and ensures consistency with the problem's defined limits.

```cpp
#include "testlib.h"
#include <bits/stdc++.h>
using namespace std;
void readAndCheckAnswer(InStream& stream, const string& pn, const string& who) {
    string line = stream.readLine();
    if (line.empty()) {
        stream.quitf(_wa, "%s: empty output", who.c_str());
    }
    // Validate characters: no whitespace allowed, only digits and '-'
    for (size_t i = 0; i < line.size(); ++i) {
        unsigned char uc = static_cast<unsigned char>(line[i]);
        if (std::isspace(uc)) {
            stream.quitf(_wa, "%s: output contains whitespace characters", who.c_str());
        }
        if (line[i] != '-' && !std::isdigit(uc)) {
            // Print the offending character; if non-printable, show its code
            if (std::isprint(uc)) {
                stream.quitf(_wa, "%s: output contains invalid character '%c'", who.c_str(), line[i]);
            } else {
                stream.quitf(_wa, "%s: output contains invalid character with code %d", who.c_str(), int(uc));
            }
        }
    }
    // Split line into groups on '-'
    vector<string> groups;
    string cur;
    for (char c : line) {
        if (c == '-') {
            groups.push_back(cur);
            cur.clear();
        } else {
            cur.push_back(c);
        }
    }
    groups.push_back(cur);
    // For each group, check that length is 2 or 3 and consists of digits
    for (size_t i = 0; i < groups.size(); ++i) {
        int len = int(groups[i].length());
        if (len < 2 || len > 3) {
            stream.quitf(_wa, "%s: group %d has invalid length %d (should be 2 or 3)", who.c_str(), int(i + 1), len);
        }
        if (!all_of(groups[i].begin(), groups[i].end(), [](char ch) { return std::isdigit(static_cast<unsigned char>(ch)); })) {
            stream.quitf(_wa, "%s: group %d contains non-digit characters", who.c_str(), int(i + 1));
        }
    }
    // Concatenate all groups and check if it matches pn
    string reconstructed;
    for (const auto& g : groups) {
        reconstructed += g;
    }
    if (reconstructed != pn) {
        stream.quitf(_wa, "%s: concatenated groups do not match original phone number", who.c_str());
    }
}
int main(int argc, char* argv[]) {
    registerTestlibCmd(argc, argv);
    int n = inf.readInt(2, 100);
    string pn = inf.readToken(format("[0-9]{%d}", n), "phone number");
    readAndCheckAnswer(ans, pn, "jury's answer");
    readAndCheckAnswer(ouf, pn, "participant's answer");
    quitf(_ok, "Correct");
}
```

Figure 4: Our Checker for problem Codeforces 25_B

```
#include "testlib.h"
#include <bits/stdc++.h>
using namespace std;
int main(int argc, char* argv[]) {
    registerValidation(argc, argv);
    int n = inf.readInt(1, 1000000, "n");
    inf.readSpace();
    int m = inf.readInt(1, 1000000, "m");
    inf.readEoln();
    vector<int> a = inf.readInts(n, 1, 1000000000, "a_i");
    inf.readEoln();
    vector<int> b = inf.readInts(m, 1, 60, "b_j");
    inf.readEoln();
    for (int i = 0; i < m; i++) {
        long long s = 1LL << b[i];
        ensuref(s <= 1000000000LL, "2^%d is %lld, which is greater than 1e9", b[
            i], s);
    }
    inf.readEof();
    return 0;
}
```

Figure 5: Wrong Validator in CodeContest+for problem Codeforces 309_C

```
#include "testlib.h"
#include <bits/stdc++.h>
using namespace std;
int main(int argc, char* argv[]) {
    registerValidation(argc, argv);
    int n = inf.readInt(1, 1000000, "n");
    inf.readSpace();
    int m = inf.readInt(1, 1000000, "m");
    inf.readEoln();
    vector<int> a = inf.readInts(n, 1, 1000000000, "a_i");
    inf.readEoln();
    vector<int> b = inf.readInts(m, 0, 29, "b_j");
    inf.readEoln();
    inf.readEof();
    return 0;
}
```

Figure 6: Our Corrected Validator for problem Codeforces 309_C

```
#include "testlib.h"
#include <bits/stdc++.h>
using namespace std;
int main(int argc, char* argv[]) {
    registerValidation(argc, argv);
    int n = inf.readInt(2, 2000, "n");
    inf.readEoln();
    int64_t max_edges = int64_t(n) * (n - 1) / 2;
    int k = inf.readInt(0, max_edges, "k");
    inf.readEoln();
    set<pair<int,int>> friendship_pairs;
    for (int i = 0; i < k; i++) {
        int u = inf.readInt(1, n, "u_i");
        inf.readSpace();
        int v = inf.readInt(1, n, "v_i");
        inf.readEoln();
        ensuref(u != v, "A person cannot be friends with themselves: u=%d, v=%d", u, v);
        int x = min(u, v);
        int y = max(u, v);
        pair<int,int> p = make_pair(x, y);
        ensuref(friendship_pairs.count(p) == 0, "Duplicate friendship pair (%d, %d)", x, y);
        friendship_pairs.insert(p);
    }
    int m = inf.readInt(0, max_edges, "m");
    inf.readEoln();
    set<pair<int,int>> dislike_pairs;
    for (int i = 0; i < m; i++) {
        int u = inf.readInt(1, n, "u_i");
        inf.readSpace();
        int v = inf.readInt(1, n, "v_i");
        inf.readEoln();
        ensuref(u != v, "A person cannot dislike themselves: u=%d, v=%d", u, v);
        int x = min(u, v);
        int y = max(u, v);
        pair<int,int> p = make_pair(x, y);
        ensuref(dislike_pairs.count(p) == 0, "Duplicate dislike pair (%d, %d)", x, y);
        ensuref(friendship_pairs.count(p) == 0, "Pair (%d, %d) cannot be both friends and dislike each other", x, y);
        dislike_pairs.insert(p);
    }
    inf.readEof();
    return 0;
}
```

Figure 7: Wrong Validator in CodeContest^{+}for problem Codeforces 177_C2

```cpp
#include "testlib.h"
#include <bits/stdc++.h>
using namespace std;
int main(int argc, char* argv[]) {
    registerValidation(argc, argv);
    int n = inf.readInt(2, 2000, "n");
    inf.readEoln();
    int64_t max_edges = min(100000ll, int64_t(n) * (n - 1) / 2);
    int k = inf.readInt(0, max_edges, "k");
    inf.readEoln();
    set<pair<int,int>> friendship_pairs;
    for (int i = 0; i < k; i++) {
        int u = inf.readInt(1, n, "u_i");
        inf.readSpace();
        int v = inf.readInt(1, n, "v_i");
        inf.readEoln();
        ensuref(u != v, "A person cannot be friends with themselves: u=%d, v=%d", u, v);
        int x = min(u, v);
        int y = max(u, v);
        pair<int,int> p = make_pair(x, y);
        ensuref(friendship_pairs.count(p) == 0, "Duplicate friendship pair (%d, %d)", x, y);
        friendship_pairs.insert(p);
    }
    int m = inf.readInt(0, max_edges, "m");
    inf.readEoln();
    set<pair<int,int>> dislike_pairs;
    for (int i = 0; i < m; i++) {
        int u = inf.readInt(1, n, "u_i");
        inf.readSpace();
        int v = inf.readInt(1, n, "v_i");
        inf.readEoln();
        ensuref(u != v, "A person cannot dislike themselves: u=%d, v=%d", u, v);
        int x = min(u, v);
        int y = max(u, v);
        pair<int,int> p = make_pair(x, y);
        ensuref(dislike_pairs.count(p) == 0, "Duplicate dislike pair (%d, %d)", x, y);
        ensuref(friendship_pairs.count(p) == 0, "Pair (%d, %d) cannot be both friends and dislike each other", x, y);
        dislike_pairs.insert(p);
    }
    inf.readEof();
    return 0;
}
```

Figure 8: Our Corrected Validator for problem Codeforces 177_C2

J Case Study: Heuristic Failure in Number Theory

In this section, we examine a case where a solution relies on flawed mathematical heuristics and floating-point arithmetic for a number theory problem. This case study illustrates how **CodeHacker** can detect vulnerabilities that arise from false generalizations, which are often missed by weak random test cases.

J.1 Problem: Quadratic Set

The problem asks us to find the maximum size subset of $\{1, 2, \ldots, n\}$ such that the product of their factorials is a perfect square. The constraints are $n \leq 10^6$.

A correct solution typically requires examining the prime factorization of the factorials (specifically the parity of prime exponents) using techniques like XOR hashing (Zobrist Hashing). However, the submission below attempts to solve the problem using a greedy heuristic based on $n \pmod 4$ and simple floating-point square checks.

J.2 The Vulnerable Submission

The code in Figure 9 attempts to guess which elements to remove (at most 2) to make the product a square. It relies on the helper function `is_perfect_square` to check conditions derived from loose patterns observed in small numbers.

J.3 Vulnerability Analysis

This submission exhibits a **Logic Error** rooted in a **False Generalization**.

- **Mathematical Flaw:** The condition $\prod a_i! = k^2$ depends on the parity of prime factors. The code tries to satisfy this by checking if specific arithmetic combinations of N (e.g., $N + 2$ or $N(N/2 - 1)$) are perfect squares. There is no number-theoretic basis guaranteeing that these specific checks cover all cases where the factorial product becomes a square.

- **Floating Point Risk:** The use of `remainderf` and `pow/sqrt` introduces precision risks, although the primary failure here is logical.

The Hack. Standard small test cases often satisfy the modulo patterns hardcoded in the solution. However, CodeHacker's **Stress Test** module, configured to explore high-magnitude inputs with specific prime properties, identifies $N = 998787$ as a failure case. For $N = 998787$, the code enters the `else` branch ($998787 \pmod 4 = 3$). The heuristics fail to find the optimal removal set, or incorrecty identify the removal set due to the lack of rigor in the 'is_perfect_square' logic applied to non-factorial numbers. The correct solution requires a hashing approach to track the parity of prime factors up to N, which this code completely lacks.

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

// Flaw 1: Floating point precision issues and logic error
// This function checks if a number is a perfect square using sqrt
// which is unreliable for large integers and irrelevant for the
// factorial product property required by the problem.
bool is_perfect_square(unsigned long long int n) {
  if (pow((long int)(sqrt(n)), 2) == n) {
    return true;
  }
  return false;
}

void solve() {
  unsigned long int n;
  cin >> n;
  vector<unsigned long int> arr = {};
  unsigned long int halfi = (long int)n / 2;

  // Flaw 2: Heuristic logic based on modulo 4
  // The code assumes that the set of removed numbers can always
  // be found among {halfi, n, halfi+1, 2, ...} based on n % 4.
  // This generalization holds for small N but fails for complex cases.
  unsigned long long int asdf = (unsigned long long int)2 * (halfi) * (halfi -
      1);

  if (remainderf(n, 4) == 0) {
    arr = {halfi};
  } else if (n == 1) {
    arr = {};
  } else if (remainderf(n, 4) == 1) {
    arr = {halfi, n};
  } else if ((n % 4) == 2) {
    if (is_perfect_square((unsigned long long int)(n + 2)))
      arr = {halfi + 1};
    else if (is_perfect_square((unsigned long long int)(n * (halfi - 1))))
      arr = {halfi - 2};
    else
      arr = {halfi, 2};
  } else {
    // This branch handles n % 4 == 3
    if (is_perfect_square((unsigned long long int)(n + 1)))
      arr = {halfi + 1, n};
    else if (is_perfect_square(asdf))
      arr = {n, halfi - 2};
    else if (is_perfect_square((halfi - 1) * n))
      arr = {halfi - 2, n - 2};
    else
      arr = {2, halfi, n};
  }

  vector<int> ans = {};
  for (int i = 1; i < n + 1; i++) {
    if (find(arr.begin(), arr.end(), i) == arr.end()) ans.push_back(i);
  }
  cout << ans.size() << endl;
  for (int el = 0; el < ans.size(); el++) {
    cout << ans[el] << ' ';
  }
  cout << endl;
}

int main() {
  solve();
  return 0;
}
```

Figure 9: Submission for "Quadratic Set" using flawed heuristics.

K Case Study: Logical Oversight in Construction

In this section, we present a successful hack generated by CodeHacker against a heuristic solution for *Codeforces 1388A*. This case illustrates how the agent identifies corner cases where a greedy construction strategy violates distinctness constraints.

K.1 Problem: Captain Flint and Crew Recruitment

Description: Given an integer N, represent it as the sum of 4 **distinct** positive integers, such that at least 3 of them are "nearly prime".
Definitions:

- A number is "nearly prime" if it is the product of two distinct prime numbers (e.g., $6 = 2 \cdot 3$, $10 = 2 \cdot 5$, $14 = 2 \cdot 7$).

K.2 Vulnerability Analysis

The submitted code (Figure 10) adopts a static greedy strategy. It always attempts to use the three smallest nearly primes: 6, 10, and 14. Their sum is 30. The code logic sets the fourth number to be $x = N - 30$.
The Flaw: The problem requires all 4 integers to be **different**. The code fails to check if the calculated fourth number x coincides with one of the fixed numbers $\{6, 10, 14\}$.

- If $x = 6 \implies N = 36$, output is `6 10 14 6` (Duplicate 6).
- If $x = 10 \implies N = 40$, output is `6 10 14 10` (Duplicate 10).
- If $x = 14 \implies N = 44$, output is `6 10 14 14` (Duplicate 14).

CodeHacker successfully identified these collision points and generated the adversarial inputs $N \in \{36, 40, 44\}$.

Correct Approach: A robust solution must handle these collisions. For example, if $N - 30$ causes a collision (e.g., $N = 36$), one can replace the fixed set $\{6, 10, 14\}$ (sum 30) with $\{6, 10, 15\}$ (sum 31), making the fourth number $N - 31$. For $N = 36$, this yields `6 10 15 5`, which are all distinct.

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

int main() {
    int t; cin >> t;
    while (t--) {
        int n; cin >> n;
        // The code correctly identifies that any N <= 30 is impossible
        // because 6+10+14+1 = 31 is the minimum possible sum.
        if (n < 31) {
            cout << "NO\n";
        } else {
            cout << "YES\n";
            // BUG: This construction assumes the 4th number (n-30)
            // will never collide with 6, 10, or 14.
            // For N=36, it prints "6 10 14 6", which has duplicates.
            cout << "6 10 14 " << n - 30 << "\n";
        }
    }
}
```

Figure 10: The vulnerable submission fails to handle collision cases.

L Prompt Templates

In this section, we present some prompt templates.

Checker Hack Generator Prompt

Please review the provided checker code and identify logical flaws. Each flaw should target a different logical issue that causes the checker to incorrectly judge the output.

 # Inputs
Problem Description: `{problem_description}`
Checker Code: `{checker_code}`

 # Reasoning Steps

1. **Understand Requirements**: Analyze the problem to determine if multiple valid solutions are allowed (Special Judge).
2. **Analyze Checker Logic**: Look for missing checks (too permissive) or overly rigid comparisons against standard output (too strict).
3. **Construct Hack**:
   - **False Positive**: Construct an *invalid* output that the checker accepts.
   - **False Negative**: Construct a *valid* alternative output (different from std) that the checker rejects.
4. **Verify**: Ensure your constructed output strictly follows the format.

 # Target Bug Types

Type 1: False Positive (Too Permissive)
The checker wrongly **ACCEPTS** an invalid output.

Type 2: False Negative (Too Strict)
The checker wrongly **REJECTS** a valid output. This often happens when the checker requires the output to be identical to the standard solution, ignoring other valid permutations or solutions.

 # Output Format (JSON)

```
{
  "test_cases": [
    {
      "strategy": "Checker ignores array sorting (False Positive)",
      "bug_type": "false_positive",
      "test_input": "<INPUT>",
      "fake_output": "<WRONG_OUTPUT_ACCEPTED>"
    },
    {
      "strategy": "Checker rejects valid reverse order (False Negative)",
      "bug_type": "false_negative",
      "test_input": "<INPUT>",
      "fake_output": "<VALID_ALT_OUTPUT_REJECTED>"
    }
  ]
}
```

Validator Hack Generator Prompt

Please review the provided validator code and identify logical flaws. Each flaw should target a different logical issue that causes the validator to incorrectly judge the input case.

 # Inputs
Constraints Description: `{problem_description}` (Focus on Input Format and Data Ranges)
Validator Code: `{validator_code}`

 # Reasoning Steps

1. **Extract Constraints**: List every constraint from the text (e.g., $1 \leq N \leq 10^5$, graph is connected, no duplicate edges).
2. **Audit Code**: Check if the Validator enforces each constraint using `ensuref()` or strict reading methods.
3. **Construct Hack**:

- **Type 1 (False Positive)**: Generate an **ILLEGAL** input that the Validator likely **ACCEPTS** (e.g., Validator uses `int` but input is 10^{10}).
- **Type 2 (False Negative)**: Generate a **LEGAL** input that the Validator likely **REJECTS** (e.g., Validator checks $N < 100$ but text says $N \leq 100$).

 # Bug Types

False Positive (Too Loose) The Validator accepts an input that violates the constraints. (This is dangerous as it crashes solutions).

False Negative (Too Strict) The Validator rejects a perfectly valid input.

 # Output Format (JSON)

```
{
  "test_cases": [
    {
      "strategy": "The problem requires a DAG (no cycles), but the validator only
          checks connectivity. This input contains a cycle.",
      "bug_type": "false_positive",
      "test_input": "3 3\n1 2\n2 3\n3 1",
      "expected_validity": "invalid" // This input SHOULD be invalid
    }
  ]
}
```

Code Analyst System Prompt

You are an expert Code Auditor for Competitive Programming. Your goal is to identify bugs (WA, TLE, RE, MLE) in a C++ submission by interacting with an execution environment.

 # Communication Protocol This is a multi-turn dialogue. In each turn, you must output a **Thought** followed by a **Tool Call** (or a Final Report). The system will execute your tool and return the `[Tool Output]` in the next message.

 # Available Tools

`run_python(script`$_{code}$)
Executes **arbitrary Python code**. You can use loops (`for/while`), define functions, imports (math, itertools), etc.

`run_cpp(input_content)`
Compiles and runs the target C++ code with the provided `input_content` (string). Returns: `stdout`, `stderr`, and `ExitCode`.

`finish(code_analysis_report)`
Submits the final vulnerability report.

 # One-Shot Example

User: Problem: Sum of squares $1^2 + \cdots + N^2$. $N \leq 10^6$. Code uses `int`. **Assistant:** Thought: The formula is $N(N+1)(2N+1)/6$. I need to check if $N = 10^6$ overflows a 32-bit signed int (2×10^9).

Call:

```
run_python("""
sum = 0
for i in range(1, 10000001):
    sum += i * i
print(sum)
""")
```

User: [Tool Output] 333333833333500000

Assistant: Thought: The result is $\approx 3.3 \times 10^{17}$, which far exceeds 2×10^9. This confirms Overflow. Now I'll probe the code with a small case to see if logic is correct otherwise. Expected: $1^2 + 2^2 + 3^2 = 14$.

Call: `run_cpp("3")`

User: [Tool Output] Stdout: "14", ExitCode: 0

Assistant: Thought: Small case works, but large case will definitely overflow.

Call: `finish("The code implements the summation formula correctly but risks overflow for` N `close to upper bounds. While` $N = 10^6$ `fits, any intermediate calculation without modulo might exceed limits. The vulnerability is Integer Overflow on large inputs.")`

 # Your Task
Problem: `{problem_description}`
Code: `{target_code}`

Stress Test Generator Prompt

You are an expert in Competitive Programming. Your task is to write a **Randomized Test Case Generator** in C++. This generator will be used for "Stress Testing" (Fuzzing) a target solution.

 # Problem Description
`{problem_description}`

 # Your Task
Write a complete C++ program that prints a SINGLE valid test case to standard output.
 # Requirements for the Generator Code:

1. **Randomization**: Initialize the random seed using `mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());` or `srand(time(0))`.
2. **Validity**: The output MUST strictly follow the problem's input format and constraints (e.g., if the graph must be a tree, ensure no cycles and connectivity).
3. **Scale**: Bias the generation towards **large inputs** (close to the maximum N) to test for Time Limit Exceeded (TLE) errors, but occasionally generate small edge cases.
4. **Robustness**: Avoid undefined behavior in your generator (e.g., ensure `rand() % 0` never happens).

 # Helper Functions Recommendation
You may define helper functions like `long long rand(long long a, long long b)` to generate numbers in range $[a, b]$.

 # Output Format
Provide ONLY the C++ code block. Start with `#include <bits/stdc++.h>`.

Generator Prompt: Bug Discovery & Test Case Generation

You are an expert in competitive programming. Your task is to find a bug in the following code and generate a test case that will expose it.

 # Problem Description:
`{problem_description}`

 # Incorrect Code:

```
{incorrect_code}
```

 # Hash Collision Data (Optional):
Our anti-hash generator has found two strings which have same hash values. `{Hash Collision Data}`

Your Task
Construct a test case (input) that:

1. Satisfies all the constraints of the original problem
2. Will cause the buggy implementation to produce an **incorrect result**
3. A correct solution would handle properly

Types of Errors to Target
The buggy code may fail in one of these ways:

- **WA (Wrong Answer)**: Produces incorrect output due to logical errors, wrong algorithms, or edge case mishandling
- **TLE (Time Limit Exceeded)**: Takes too long to execute, often due to inefficient algorithms or infinite loops
- **MLE (Memory Limit Exceeded)**: Takes need too much memeory space, often due to increasing space needed in the program.
- **RE (Runtime Error)**: Crashes during execution due to array out of bounds, division by zero, stack overflow, etc.

Your test case should be designed to trigger one of these error types.

 # Output Format:
Provide a complete C++ program that generates the test input. The program should:

- Always include `bits/stdc++.h` first
- Have a `main()` function that outputs the test case

- Follow the input format specified in the problem

Example Output Format
Your generator can be as simple or complex as needed. Here are examples:
Simple Example (Direct Output):

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

int main() {
    // Bug: code assumes n <= 1000, but constraint allows up to 10^5
    // This test case uses n = 100000 to trigger array overflow
    cout << "100000" << endl;
    return 0;
}
```

Complex Example (Algorithmic Generation):

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

int main() {
    // Bug: O(n^2) algorithm will TLE on large inputs
    // Generate maximum size test case with worst-case pattern
    int n = 200000;
    cout << n << endl;

    // Generate adversarial pattern: descending order
    for (int i = n; i >= 1; i--) {
        cout << i;
        if (i > 1) cout << " ";
    }
    cout << endl;
    return 0;
}
```

Note: Your generator can include loops, conditionals, calculations, or any logic needed to construct the test case.

Additional Guidelines

1. **Analyze the bug**: First identify what type of bug exists (Logic errors → WA, Inefficient loops → TLE, etc.)
2. **Target the weakness**: Design your test case to specifically trigger that bug
3. **Stay within constraints**: Your test case must be valid according to the problem constraints
4. **Maximize impact**: Choose values that make the error obvious and deterministic

Now, write the C++ generator based on this advice. Think step by step.

Anti-Hash Generator

You are a Cryptanalysis and Competitive Programming Expert specializing in **Hash Collisions**. Your target is to break a solution that uses **Polynomial Rolling Hash** by generating two different strings with the same hash value (Collision).

 # Workflow

1. **Identify Hashing**: Locate the hash calculation logic (usually `h = (h * B + c) % M`).
2. **Extract Parameters**:
    - **Base (B)**: The multiplier (e.g., 31, 131, 13331, or a random value).
    - **Modulus (M)**: The modulo (e.g., $10^9 + 7$, $10^9 + 9$).
    - **Character Set (C)**: The valid alphabet (e.g., 'a'-'z').
    - **Mapping (C)**: The character-to-integer conversion logic (e.g., `s[i] - 'a' + 1` or raw ASCII values).
    - *Note*: If the code uses `unsigned long long` without explicit modulo, $M = 2^{64}$.

 # Input
Problem: `{problem_description}`
Code: `{target_code}`

 # Output Format (JSON)

```
{
  "vulnerability_type": "Hash Collision",
  "hash_parameters": {
      "base": "List of Integer (e.g., [131, 13331])",
      "modulus": "List of Integer or expression (corresponding one-to-one with
          base, e.g., [998244353, 2^64])",
      "character set": "a,b,c,...,x,y,z"
      "mapping": "{'a':1,'b':2,...}"
  }
}
```