Can LLMs Generate High-Quality Test Cases for Algorithm Problems? TestCase-Eval: A Systematic Evaluation of Fault Coverage and Exposure

Zheyuan Yang[♣] **Zexi Kuang**[◇] **Xue Xia**[♡] **Yilun Zhao**[♠]

[♣]Tongji University [◇]Northeastern University [♡]HKUST [♠]Yale University

Abstract

We introduce TestCase-Eval, a new benchmark for systematic evaluation of LLMs in test-case generation. TestCase-Eval includes 500 algorithm problems and 100,000 human-crafted solutions from the Codeforces platform. It focuses on two pivotal tasks: (1) *Fault Coverage*, which measures how well LLM-generated test sets probe diverse input scenarios and cover a wide range of potential failure modes. (2) *Fault Exposure*, which evaluates whether LLMs can craft a tailored test input that reveals a specific incorrect code implementation. We provide a comprehensive assessment of 19 state-of-the-art open-source and proprietary LLMs on TestCase-Eval, offering insights into their strengths and limitations in generating effective test cases for algorithm problems.

Data TestCase-Eval
Code FlowRays/TestCase-Eval

1 Introduction

Algorithmic problem-solving is fundamental to computational fields such as software engineering, data science, and competitive programming (Jimenez et al., 2023; Huang et al., 2023; Jain et al., 2024; Yu et al., 2024a; El-Kishky et al., 2025). The correctness and robustness of algorithmic solutions hinge on the quality of test suites—carefully designed inputs that uncover edge cases, corner conditions, performance limitations, and common failure scenarios (Austin et al., 2021; Hendrycks et al., 2021; Li et al., 2022). Traditionally, crafting such test cases requires significant domain expertise and manual effort. With the rapid advancement of LLMs capable of sophisticated code generation, a crucial question arises: **Can LLMs generate high-quality test cases that match or surpass those designed by human experts?**

We introduce TestCase-Eval, a comprehensive benchmark for systematically evaluating LLMs in test-case generation for algorithmic problems. It comprises 500 up-to-date algorithm problems and 100,000 corresponding real-human crafted solutions, both sourced from the Codeforces platform. As illustrated in Figure 1, TestCase-Eval features two core tasks, each targeting a crucial aspect of test-case quality: (1) *Fault Coverage*, which evaluates whether LLM-generated test cases effectively explore diverse input scenarios, including edge cases and boundary conditions, to expose various types of incorrect solutions. (2) *Fault Exposure*, which evaluates whether an LLM can generate a targeted test input that successfully exposes the flaws in a specific given incorrect solution.

We conduct an extensive evaluation on TestCase-Eval, covering 19 frontier open-source and proprietary LLMs. Our experimental results demonstrate that TestCase-Eval presents a significant challenge, with even top-performing models like Qwen3-32B scoring only 43.8% on the Fault Exposure task—far below human expert performance (93.3%). These findings underscore the inherent difficulty of TestCase-Eval. Furthermore, our in-depth analysis of reasoning LLMs, CoT reasoning, and model performance across different programming languages and error types offers valuable insights for future advancements in the field.

2 Related Work

Prior works on LLM-based test-case generation follows two main directions: (1) Enhancing code generation via self-debugging, where models iteratively refine solutions by generating and analyzing test cases (Chen et al., 2022; Zhang et al., 2023; Shinn et al., 2023; Jiao et al., 2024; Zeng et al., 2025). (2) Improving code evaluation by leveraging LLMs to generate diverse test cases, as used in recent code generation evaluation benchmarks (Liu et al., 2023; Du et al., 2024; Yu et al., 2024b; Jain et al., 2024). Despite these advancements, a sys-

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), pages 1050–1063

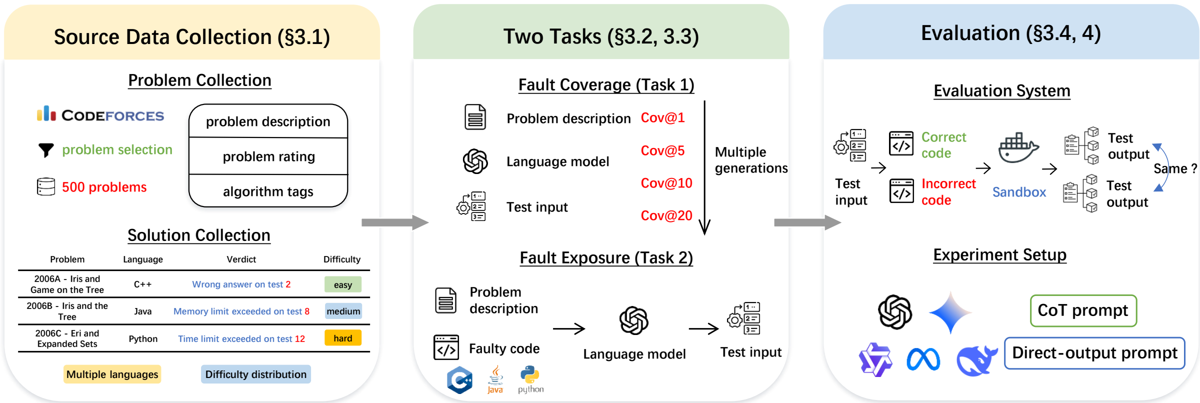

Figure 1: An overview of TestCase-Eval and the research pipeline in this study.

tematic study on the standalone capability of LLMs in test-case generation remains an open challenge. A closely related benchmark for algorithmic problem test-case generation is TestEval (Wang et al., 2024), which evaluates test generation for LeetCode problems but relies on traditional Line and Branch Coverage assessment (illustrated in Appendix A.1), which may be inadequate for algorithmic problem settings. For instance, the 6.7B DeepSeek-Coder achieves over 90% in TestEval, with top models nearing 100%. Our work shifts the focus to more challenging CodeForces competition problems and introduces two novel and challenging tasks. Even state-of-the-art models achieve only around 40% on the task of Fault Exposure.

3 TestCase-Eval Benchmark

This section discusses the TestCase-Eval benchmark construction and task settings, as illustrated in Figure 1.

3.1 Source Data Collection

We first outline the data collection process, which involves collecting *problems* with corresponding correct and incorrect *human-written solutions*.

Problem Collection. We collect algorithmic problems from Codeforces contests held between January 1, 2024, and December 30, 2024. This time frame falls outside the pretraining period of most existing foundation models, reducing potential data memorization concerns. Our goal is to curate a dataset that (1) includes a substantial number of incorrect submissions and (2) ensures accurate offline evaluation. To achieve this, we apply a series of filtering steps. First, we exclude problems requiring special judge functionalities (detailed in Appendix B.3), as these allow multiple valid outputs for a single input, often leading to unreliable evaluations. Next, we verify each problem by running ten correct human-written solutions sourced from Codeforces, ensuring they consistently produce identical outputs for the same test inputs. Additionally, we remove problems with fewer than 1,000 online incorrect submissions to ensure a diverse range of mistakes. After this selection process, our final dataset includes a total of 500 problems. Figure 3 in Appendix presents the distribution of problem difficulty ratings.

Human-written Solution Collection. For each problem in TestCase-Eval, we collect 200 incorrect submissions (*i.e.,* human-written solutions along with their evaluation outcomes) from the Codeforces platform. The platform provides detailed error types for each incorrect submission, such as "Memory Limit Exceeded", "Time Limit Exceeded", "Runtime Error", and "Wrong Answer". Additionally, it specifies the test case index where the error occurred (*e.g.,* "Wrong answer on test 5"), with higher indices generally indicating more complex or inherent errors. Leveraging this information, we categorize incorrect submissions into three difficulty levels: *Easy*, *Medium*, and *Hard*. (Detailed difficulty definition and distribution are illustrated in Appendix B.2) To ensure diversity, we select code written in three widely used programming languages: C++, Python, and Java. We begin by crawling the submission logs for each problem, which contain the complete history of contestant submissions. These logs are carefully filtered based on test case failures and programming language criteria, resulting in a comprehensive dataset of 100,000 submissions. (Detailed dataset collection and sampling pipeline are illustrated in Appendix B.2)

We next outline the construction process for the two evaluation tasks in TestCase-Eval.

3.2 Fault Coverage Evaluation (Task 1)

This task assesses the LLM's ability to generate comprehensive test inputs that effectively detect faulty code implementations. Specifically, given the description of an algorithmic problem, the LLM must thoroughly understand the problem and generate a specified number of test cases that maximize coverage of incorrect solution scenarios. Let $\mathcal{T}_N = \{t_1, t_2, \ldots, t_N\}$ represent the set of N test inputs generated by the LLM. For each test input t_i, let $\mathcal{F}(t_i)$ denote the subset of incorrect submissions it detects, drawn from the complete set of incorrect solutions $\mathcal{F}_{\text{total}}$, the final score for this task is defined as the coverage rate of incorrect solutions when generating N test inputs:

$$Cov@N = \frac{\left|\bigcup_{i=1}^{N} \mathcal{F}(t_i)\right|}{|\mathcal{F}_{\text{total}}|}$$

It quantifies the LLM's effectiveness in generating diverse and impactful test cases that expose incorrect implementations.

3.3 Fault Exposure Evaluation (Task 2)

This task is inspired by the hacking phase in CodeForces competitions, where participants analyze others' solutions and attempt to "hack" them by providing inputs that reveal flaws in the code. The goal is to assess the LLM's ability to understand both the problem and the specific errors present in the faulty implementation. Given the description of an algorithmic problem and a single faulty code implementation f_i within the sampled set $\mathcal{F}$ (a strategically sampled subset of $\mathcal{F}_{\text{total}}$), Task 2 requires the LLM to generate a single test input t_i to exploit the fault. The *Fault Exposure Rate* is computed as:

$$\text{Fault Exposure Rate} = \frac{1}{|\mathcal{F}|} \sum_{f_i \in \mathcal{F}} e(f_i, t_i), \text{ where}$$

$$e(f_i, t_i) = \begin{cases} 1, & \text{if } t_i \text{ successfully exposes fault in } f_i, \\ 0, & \text{otherwise.} \end{cases}$$

It measures both general test case generation capabilities and targeted fault detection performance.

4 Experiment

4.1 Experiment Setup

We evaluate 11 series of open-source models, including Qwen2.5 (Yang et al., 2024b) and Qwen2.5-Coder (Hui et al., 2024), Qwen3 (Yang et al., 2025), QwQ (Team, 2025), Llama-3.1&3.3 (Meta, 2024), Gemma-3 (Team et al., 2025), DeepSeek-R1 (DeepSeek-AI et al., 2025), Mistral-Small (Jiang et al., 2023), Codestral (Team, 2024), and SeedCoder (Seed, 2025). We also evaluate two series of proprietary models, including GPT-4o (OpenAI, 2024) and GPT-4.1 (OpenAI, 2025). Appendix C.1 details the parameter settings and model configurations. We evaluate the models with both **Direct Output** and **Chain-of-Thought** prompts, with prompting examples presented in Appendix C.2. We utilize the **sandbox environment** from ExecEval (Khan et al., 2023) for code execution and test input evaluation, ensuring secure execution and reliable assessment of results. To approximate **human-expert-level performance** on TestCase-Eval, we randomly sampled 20 problems from the dataset. Two human experts, with Codeforces ratings of 2080 and 2237, independently completed both Task 1 and Task 2 for each problem. Their performance was then averaged to obtain the final assessment.

4.2 Experimental Results and Analysis

Table 1 illustrates the model performance on TestCase-Eval. Our key findings are as follows:

TestCase-Eval presents substantial challenges for current models. The TestCase-Eval benchmark is highly challenging, as evidenced by the significant performance gap between models and human experts on both tasks. This gap is particularly pronounced in Task 2 (Fault Exposure), where human experts achieve a 93.3% fault exposure rate, more than double the best-performing model, Qwen3-32B (43.8%). While Task 1 (Fault Coverage) also shows a considerable gap, models achieve relatively higher scores, suggesting that generating a broad set of test cases is more tractable than triggering specific code flaws. Furthermore, we observe that Task 2 yields more stable and reproducible results across multiple evaluation runs, whereas Task 1 scores exhibit higher variance, likely due to the stochastic nature of generating a diverse set of test inputs.

Models	T1: Fault Coverage				T2: Fault Exposure			
	c@1	c@5	c@10	c@20	Easy	Med.	Hard	**Ovr.**
Human Expert	56.2	85.7	93.5	97.2	95.0	92.5	91.8	93.3
GPT-4.1	45.3	67.5	74.1	80.0	42.9	34.3	30.3	36.5
GPT-4.1-mini	38.8	63.2	68.5	72.6	39.2	32.4	27.4	33.6
GPT-4o	36.4	60.3	69.7	73.5	37.5	30.5	25.2	31.7
Qwen3-8B	46.2	78.5	87.9	92.1	48.6	39.8	33.1	41.3
Qwen3-32B	50.8	82.3	92.6	95.7	52.7	42.5	33.2	43.8
R1-Distill-Qwen-32B	31.9	65.3	75.6	82.6	48.7	39.7	33.9	41.6
QwQ-32B	37.3	58.9	67.6	78.3	49.4	38.0	30.2	40.2
Qwen2.5-7B	38.6	65.4	73.0	79.1	39.8	34.2	29.5	35.0
Qwen2.5-Coder-7B	36.7	63.2	70.5	76.5	37.2	33.1	29.9	33.7
Qwen2.5-32B	44.4	70.9	79.6	88.4	38.8	30.4	25.5	32.3
Qwen2.5-Coder-32B	35.8	66.7	81.8	89.7	40.5	34.0	27.3	34.6
Qwen2.5-72B	38.2	57.8	65.2	73.1	33.6	27.5	24.2	29.0
Llama-3.1-70B	47.8	75.4	84.8	90.9	37.9	33.5	30.5	34.3
Llama-3.3-70B	43.2	72.5	81.2	88.6	33.8	27.9	25.2	29.5
Mistral-Small-24B	35.5	71.9	80.4	88.3	37.4	31.9	28.4	33.1
Codestral-22B	34.8	68.8	87.4	90.8	34.9	28.2	26.4	30.3
Gemma-3-12B	30.4	54.6	61.0	65.3	35.7	31.9	33.3	33.8
Gemma-3-27B	32.4	55.6	64.1	70.7	34.3	28.3	28.3	30.7
Seed-Coder-8B	30.7	63.2	75.6	87.4	34.5	28.1	25.5	29.9

Table 1: Performance of the evaluated LLMs with CoT reasoning on TestCase-Eval. For Task 1, we report Coverage@N; for Task 2, we report fault exposure rate.

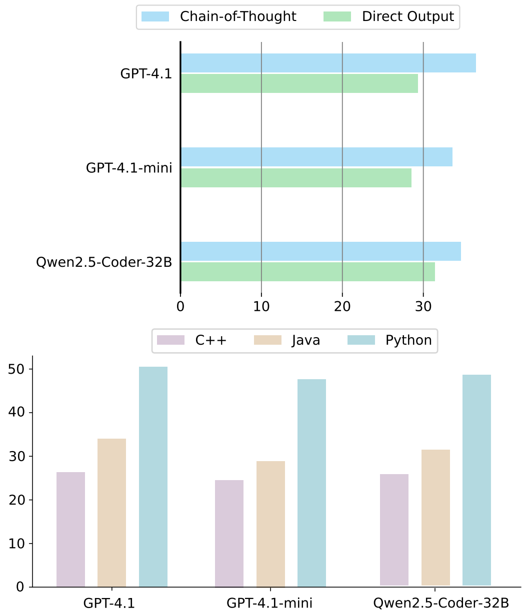

Figure 2: (Top) Performance comparison between CoT prompting and direct-output prompting for Task 2. (Bottom) Overall model performance using CoT prompting across C++, Java, and Python in Task 2.

Open-source models compete with or surpass proprietary counterparts. Our results indicate that leading open-source models are highly competitive. In Task 1 (Fault Coverage), several open-source models, including Qwen3-32B (50.8 cov@1) and Llama-3.1-70B (47.8 cov@1), outperform the best proprietary model, GPT-4.1 (45.3 cov@1). The trend continues at higher N values, where Qwen3-32B's cov@20 score of 95.7 significantly surpasses GPT-4.1's 80.0. In the more reasoning-intensive Task 2, while the Qwen3 series leads, GPT-4.1 shows strong performance with a 36.5% fault exposure rate, outperforming all other general-purpose open-source models like Llama-3.1-70B (34.3%). This highlights a competitive landscape where proprietary models do not hold a universal advantage in our benchmark.

Reasoning LLMs outperform general-purpose LLMs on both tasks. Reasoning-oriented models, such as the Qwen3 series, demonstrate superior performance on both tasks. Notably, Qwen3-32B achieves the highest scores in Task 1 across all metrics (e.g., 50.8 cov@1 and 95.7 cov@20), clearly surpassing strong general-purpose models such as Llama-3.1-70B, as well as proprietary models like GPT-4.1. This performance gap becomes even more pronounced in Task 2, which demands deeper analytical capabilities. Qwen3-32B and R1-Distill-Qwen-32B attain the top two fault exposure rates, at 43.8% and 41.6% respectively, with a substantial margin over all other evaluated models. These results suggest that reasoning models excel because they are better equipped to analyze algorithmic problem descriptions, systematically identify possible fault patterns, and generate high-quality test inputs.

CoT prompts vs direct-output prompts. Our experiments reveal that CoT prompting significantly outperforms direct-output prompting in generating test cases (Figure 2). This advantage stems from CoT's structured reasoning process, which guides the model through intermediate steps before arriving at the final output. Such an approach is particularly beneficial for the complex tasks in TestCase-Eval, where systematic thinking is crucial. When generating test cases for fault exposure, CoT prompting led to more effective fault detection, especially in challenging edge cases. This suggests that models benefit from explicit reasoning steps, as they help decompose intricate problems and improve fault exposure rate in nuanced scenarios.

Comparison of fault exposure results across different programming languages. In Fault Exposure task, model performance varies across programming languages. As shown in Figure 2, models generally achieve higher fault exposure rates on Python solutions, likely due to Python's dynamic typing, flexible syntax, and interpreted execution, which facilitate the generation of diverse test cases that reveal faults. In contrast, C++ and Java exhibit lower fault exposure rates, possibly due to their strict type enforcement, manual memory management, and compiled execution, which can limit the likelihood of generating inputs that reveal subtle faults. Addressing these differences through targeted adaptation could help improve fault detection across a wider range of programming languages.

Performance breakdown to four major error types. Table 2 provides a detailed breakdown of model performance on task 2 across four major error types: Wrong Answer (WA), Runtime Error (RE), Time Limit Exceeded (TLE), and Memory Limit Exceeded (MLE). A clear trend emerges from the data: models generally demonstrate stronger capabilities in detecting logical and execution-related faults (WA and RE) compared to resource-based faults (TLE and MLE). This suggests that current LLMs are generally better at detecting logical or edge-case errors than time or memory inefficiencies.

Notably, the best-performing models, Qwen3-32B and R1-Distill-Qwen-32B, exhibit significantly higher accuracy on Wrong Answer type. Given that WA constitutes the largest proportion of error types, their superior performance in this category is the primary driver of their high overall accuracy (43.8% and 41.6%, respectively). This finding underscores that the strength of these advanced reasoning models lies in their enhanced ability to construct precise test cases that target and expose logical inconsistencies within code.

Models	WA	RE	TLE	MLE	Ovr.
GPT-4.1	42.0	35.4	20.9	25.1	36.5
GPT-4.1-mini	39.3	32.0	17.4	24.6	33.6
GPT-4o	37.1	29.0	16.4	25.1	31.7
Qwen3-8B	48.0	39.0	22.8	26.9	41.3
Qwen3-32B	52.2	38.7	21.2	22.3	43.8
R1-Distill-Qwen-32B	48.0	37.8	23.9	30.3	41.6
QwQ-32B	49.1	35.1	16.4	14.9	40.2
Qwen2.5-7B	38.7	37.4	23.6	31.4	35.0
Qwen2.5-Coder-7B	35.5	39.8	26.5	36.0	33.7
Qwen2.5-32B	36.8	34.7	18.0	28.6	32.3
Qwen2.5-Coder-32B	38.9	38.3	21.0	26.9	34.6
Qwen2.5-72B	33.0	30.5	16.8	20.6	29.0
Llama-3.1-70B	36.5	33.9	27.7	36.6	34.3
Llama-3.3-70B	32.9	30.2	19.3	24.6	29.5
Mistral-Small-24B	35.8	37.4	23.1	40.0	33.1
Codestral-22B	33.1	35.0	20.3	31.4	30.3
Gemma-3-12B	35.8	35.1	27.7	30.9	33.8
Gemma-3-27B	33.5	33.3	22.3	21.7	30.7
Seed-Coder-8B	32.8	31.7	20.5	31.4	29.9

Table 2: Performance breakdown of evaluated LLMs on task 2 (fault exposure), reported by four error types.

5 Conclusion

We introduce TestCase-Eval, a new benchmark for evaluating LLMs in test-case generation for algorithmic problems, with a focus on Fault Coverage and Fault Exposure. Our results show that all the evaluated LLMs struggle with harder faults, highlighting the challenge of automated test-case generation. Although CoT prompting enhances performance, a substantial gap remains between frontier LLMs and human experts. These findings emphasize the need for further research to enhance LLMs' capabilities in generating high-quality test cases and their practical application in software testing.

Limitations

While TestCase-Eval provides a comprehensive benchmark for evaluating LLMs in test-case generation, several limitations warrant further investigation: (1) Our evaluation primarily emphasizes quantitative performance indicators, such as fault coverage and exposure rates, which might not capture the nuanced failure modes that may arise in LLM-generated test cases. Future work could include more detailed error analyses to uncover specific failure patterns and model weaknesses. (2) The difficulty levels (Easy, Medium, Hard) in TestCase-Eval are determined by the test case index where an incorrect solution first fails. While this provides a reasonable estimate of error complexity, it does not explicitly categorize the types of mistakes. (3) Our benchmark focuses on correctness-based faults and does not systematically test performance bottlenecks (*e.g.,* time limit exceeded, memory limit exceeded). Although some incorrect submissions fail due to resource constraints, we do not explicitly assess whether LLMs generate test cases that effectively expose computational complexity flaws. (4) While TestCase-Eval targets test-case generation, practical software testing also requires identifying the root cause and location of bugs. Future work could extend this to more holistic debugging and fault localization tasks.

References

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

Bei Chen, Fengji Zhang, A. Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *ArXiv*, abs/2207.10397.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Jun-Mei Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiaoling Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bing-Li Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dong-Li Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Jiong Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, M. Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shao-Kang Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wen-Xia Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyu Jin, Xi-Cheng Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yi Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yu-Jing Zou, Yujia He, Yunfan Xiong, Yu-Wei Luo, Yu mei You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yao Li, Yi Zheng, Yuchen Zhu, Yunxiang Ma, Ying Tang, Yukun Zha, Yuting Yan, Zehui Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhen guo Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zi-An Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A code efficiency benchmark for code large language models. In *Neural Information Processing Systems*.

OpenAI Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub W. Pachocki, Jerry Tworek, Lorenz Kuhn, Lukasz Kaiser, Mark Chen, Max Schwarzer, Mostafa Rohaninejad, Nat McAleese, o3 contributors, Oleg Murk, Rhythm Garg, Rui Shu, Szymon Sidor, Vineet Kosaraju, and Wenda Zhou. 2025. Competitive programming with large reasoning models.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Xiaodong Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *ArXiv*, abs/2105.09938.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2023. Mlagentbench: Evaluating language agents on machine learning experimentation. In *International Conference on Machine Learning*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Shanghaoran Quan, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *ArXiv*, abs/2409.12186.

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

Albert Qiaochu Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, L'elio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *ArXiv*, abs/2310.06825.

Fangkai Jiao, Geyang Guo, Xingxing Zhang, Nancy F. Chen, Shafiq Joty, and Furu Wei. 2024. Preference optimization for reasoning with pseudo feedback. *ArXiv*, abs/2411.16345.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *ArXiv*, abs/2310.06770.

Mohammad Abdullah Matin Khan, M Saiful Bari, Do Xuan Long, Weishi Wang, Md. Rizwan Parvez, and Shafiq R. Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *ArXiv*, abs/2303.03004.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom, Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de, Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey, Cherepanov, James Molloy, Daniel Jaymin Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de, Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378:1092 – 1097.

Jiawei Liu, Chun Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *ArXiv*, abs/2305.01210.

Meta. 2024. The llama 3 herd of models. *ArXiv*, abs/2407.21783.

OpenAI. 2024. Hello gpt-4o.

OpenAI. 2025. Introducing gpt-4.1 in the api.

ByteDance Seed. 2025. Seed-coder: Let the code model curate data for itself.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Neural Information Processing Systems*.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. 2025. Gemma 3 technical report.

Mistral AI Team. 2024. Codestral: Hello, world!

Qwen Team. 2025. Qwq-32b: Embracing the power of reinforcement learning.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2024. Testeval: Benchmarking large language models for test case generation. *ArXiv*, abs/2406.04531.

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

Qiu. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Ke-Yang Chen, Kexin Yang, Mei Li, Min Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yunyang Wan, Yunfei Chu, Zeyu Cui, Zhenru Zhang, and Zhi-Wei Fan. 2024a. Qwen2 technical report. *ArXiv*, abs/2407.10671.

Qwen An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxin Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yi-Chao Zhang, Yunyang Wan, Yuqi Liu, Zeyu Cui, Zhenru Zhang, Zihan Qiu, Shanghaoran Quan, and Zekun Wang. 2024b. Qwen2.5 technical report. *ArXiv*, abs/2412.15115.

Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. 2024a. Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation.

Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. 2024b. Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation. *ArXiv*, abs/2412.21199.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. Acecoder: Acing coder rl via automated test-case synthesis.

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *ArXiv*, abs/2305.14591.

A Related Work

A.1 Line and Branch Coverage

In the realm of software testing, two pervasive metrics are Line Coverage and Branch Coverage. These are often used to evaluate the adequacy of test cases in executing program code.

Specifically, **Line Coverage** measures the percentage of lines of code that have been executed by a set of test cases. It provides insight into which lines of the codebase are actually executed during testing, aiming to ensure that all parts of the code are tested at least once. **Branch Coverage** takes a more granular approach by focusing on the control structures within the code, such as if statements and switch cases. It evaluates whether each possible branch (i.e., each path through a control structure) has been executed. This metric ensures that all possible execution paths are tested.

While these metrics are invaluable in traditional software testing, they fall short in the context of competitive algorithm problems for several reasons: (1) Algorithm Complexity and Diversity: Competitive algorithm problems often involve complex data structures and intricate algorithmic logic that cannot be fully represented by simple line or branch execution. The focus is on the correctness and efficiency of the algorithm, rather than merely executing each line or branch of code. (2) Outcome-Oriented Nature: The primary goal in algorithm competitions is to solve problems correctly and efficiently, not just to achieve high code coverage. An algorithm may achieve high line and branch coverage but still fail to solve the problem correctly or efficiently. (3) Diversity of Test Cases: Algorithm competition problems require testing against a wide variety of edge cases and specific inputs. The generation and evaluation of these test cases extend beyond the scope of simple line and branch coverage metrics, which may not adequately reflect the comprehensiveness of the test cases in ensuring algorithmic correctness and robustness. Traditional Line and Branch Coverage metrics may be inadequate in the context of algorithmic problems. Therefore, we propose two new tasks, Fault Coverage and Fault Exposure, along with corresponding evaluation metrics.

B TestCase-Eval Benchmark

We provide a detailed explanation of the problems and human-written solutions within the dataset.

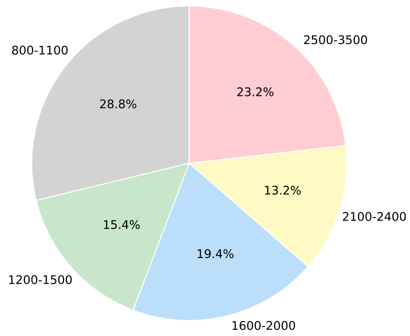

Figure 3: Distribution of Problem Difficulty Levels.

B.1 Problems

Each problem sourced from the Codeforces platform comprises several key elements: 1) title, 2) time limit, 3) memory limit, 4) problem description, 5) input format, 6) output format, 7) test case examples, and 8) optional note. We utilize all of this data to form the `problem_description` string, which acts as the input for the LLM.

Additionally, we analyze the distribution of problem difficulty ratings, which is illustrated in Figure Figure 3.

B.2 Human-written Solutions

To ensure both the representativeness and diversity of error patterns in our benchmark, we developed a comprehensive dataset collection and sampling pipeline.

For each selected Codeforces problem, we first crawled the complete submission logs to collect a representative set of user-submitted incorrect solutions. Specifically, for each problem, we initially sampled 100 incorrect solutions for each of the three major programming languages (C++, Python, and Java), resulting in a preliminary pool. We then applied multiple rounds of filtering and cleaning to ensure quality and diversity. This process yielded a final set of 118,611 human-written incorrect solutions across 500 algorithmic problems, amounting to an average of 237 solutions per problem. These submissions reflect genuine programming errors from a diverse pool of users, capturing a broad spectrum of error types and difficulty levels observed in real-world programming scenarios.

We imposed strict criteria on the sampled solutions: each must be semantically valid and executable, passing compilation and basic test cases without syntax or runtime errors, and failing only under specific, non-trivial input conditions. This design ensures our benchmark targets input-sensitive faults—precisely those that require sophisticated and diverse test input generation to detect.

To construct a manageable yet representative subset for Task 2, we performed stratified sampling for each problem. We began by analyzing the distribution of incorrect solutions by error type and difficulty, which we defined based on the index of the first failed test case. Guided by this analysis, we sampled 20 incorrect solutions per problem, ensuring balanced representation across three major programming languages and maintaining proportional coverage of both error types and difficulty levels.

Codeforces problems typically include between a dozen and over two hundred test cases, each comprising a set of inputs and expected outputs. For a given submission, the verdict "Wrong answer on test 5" indicates that the solution passed the first four test cases but failed on the fifth. The index of the first failed test case thus serves as a crucial signal for assessing both the difficulty of a solution and the effectiveness of generated test cases.

Based on this index, we categorize solutions into three levels of difficulty: *Easy*, *Medium*, and *Hard*. Specifically, we sort all human-written solutions for each problem by the index of the first error, assigning the bottom 40% to *Easy*, the middle 30% to *Medium*, and the top 30% to *Hard*.

B.3 Special Judge

In competitive programming platforms like Codeforces, certain problems permit multiple valid test outputs for a single test input. To validate such outputs, a special judge is employed. This custom code evaluates the correctness of each output, as straightforward comparison to a reference output is inadequate due to the problem's complexity. Figure 4 illustrates a problem that necessitates a special judge.

For accurate offline evaluation, we excluded all problems requiring a special judge. Such problems can cause inconsistent assessments since they allow multiple correct outputs for the same input.

B.4 Multiple Test Cases

In competitive programming platforms (especially Codeforces), multiple test cases within a single test input are a standard feature. As illustrated in Figure 4, problem input specifications often begin with instructions such as "Each test contains multiple test cases" emphasizing the expectation that solutions correctly process a batch of cases in one execution. Through manual inspection of our dataset, we verified that 439 out of the 500 problems inherently require handling multiple test cases. To ensure comprehensive test coverage, we designed our prompt in Appendix C.2 to explicitly guide LLMs in generating diverse test cases in a single test input.

B. Kevin and Geometry

time limit per test: 1 second

memory limit per test: 256 megabytes

Kevin has n sticks with length $a_1, a_2, \ldots, a_n$.

Kevin wants to select 4 sticks from these to form an isosceles trapezoid* with a positive area. Note that rectangles and squares are also considered isosceles trapezoids. Help Kevin find a solution. If no solution exists, output -1.

* An isosceles trapezoid is a convex quadrilateral with a line of symmetry bisecting one pair of opposite sides. In any isosceles trapezoid, two opposite sides (the bases) are parallel, and the two other sides (the legs) are of equal length.

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \le t \le 10^4$). The description of the test cases follows.

The first line of each test case contains a single integer n ($4 \le n \le 2 \cdot 10^5$).

The second line contains n integers $a_1, a_2, \ldots, a_n$ ($1 \le a_i \le 10^8$).

It is guaranteed that the sum of n over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, output 4 integers — the lengths of sticks. If no solution exists, output -1.

If there are multiple solutions, print any of them.

This indicts the problem needs a special judge

Example

input

```
7
4
5 5 5 10
4
10 5 10 5
4
1 2 3 4
4
1 1 1 3
6
4 2 1 5 7 1
6
10 200 30 300 30 100
4
100000000 100000000 1 2
```

output

```
5 5 5 10
5 5 10 10
-1
-1
1 1 4 5
-1
100000000 100000000 1 2
```

Note

In the first test case, you can form an isosceles trapezoid with bases of length 5 and 10, and two legs of length 5.

In the second test case, you can form an isosceles trapezoid with two bases of length 5 and two legs of length 10. A rectangle is considered an isosceles trapezoid here.

In the third test case, there are no sticks with the same length. It's impossible to form an isosceles trapezoid.

In the fourth test case, it's impossible to form an isosceles trapezoid with a positive area.

Figure 4: An example of a problem that needs a special judge.

C Experiment Setup

C.1 Evaluated Model Configuration

Table 3 details the configuration of each evaluated model. Across all experiments, the temperature is set to 1.0 to ensure diversity in the LLM-generated test cases. The maximum output length is generally configured to 2048 tokens, which suffices for most standard models. However, for reasoning models like QwQ-32B and R1-Distill-Qwen-32B, this maximum output length is extended to 18,000 tokens to accommodate their long CoT reasoning mechanisms. All inference processes are conducted on two NVIDIA A100-80G GPUs.

C.2 CoT and Direct Output Prompts

Model	Citation	Version
GPT-4.1	OpenAI (2025)	gpt-4.1-2025-04-14
GPT-4.1-mini	OpenAI (2025)	gpt-4.1-mini-2025-04-14
GPT-4o	OpenAI (2024)	gpt-4o-2024-11-20
Qwen3-8B	Yang et al. (2025)	Qwen/Qwen3-8B
Qwen3-32B	Yang et al. (2025)	Qwen/Qwen3-32B
R1-Distill-Qwen-32B	DeepSeek-AI et al. (2025)	deepseek-ai/DeepSeek-R1-Distill-Qwen-32B
QwQ-32B	Team (2025)	Qwen/QwQ-32B
Qwen2.5-7B	Yang et al. (2024a)	Qwen/Qwen2.5-7B-Instruct
Qwen2.5-Coder-7B	Hui et al. (2024)	Qwen/Qwen2.5-Coder-7B-Instruct
Qwen2.5-32B	Yang et al. (2024b)	Qwen/Qwen2.5-32B-Instruct
Qwen2.5-Coder-32B	Hui et al. (2024)	Qwen/Qwen2.5-Coder-32B-Instruct
Qwen2.5-72B	Yang et al. (2024b)	Qwen/Qwen2.5-72B-Instruct
Llama-3.1-70B	Meta (2024)	meta-llama/Llama-3.1-70B-Instruct
Llama-3.3-70B	Meta (2024)	meta-llama/Llama-3.3-70B-Instruct
Mistral-Small-24B	Jiang et al. (2023)	mistralai/Mistral-Small-24B-Instruct-2501
Codestral-22B	Team (2024)	mistralai/Codestral-22B-v0.1
Gemma-3-12B	Team et al. (2025)	google/gemma-3-12b-it
Gemma-3-27B	Team et al. (2025)	google/gemma-3-27b-it
Seed-Coder-8B	Seed (2025)	ByteDance-Seed/Seed-Coder-8B-Instruct

Table 3: Model List.

The Chain-of-Thought Prompt in Task1

Task:
Generate a challenging test input for the algorithm problem:
{problem_description}

Instructions:
- Focus on edge cases or scenarios that maximize the failure probability in faulty solutions.
- Due to the output length limit, you should generate a small-scale test input that is complete and valid.
- Output the test input directly, not code to generate it.

Output format:

````
```plaintext
{test input}
```
````

Think step by step.

Figure 5: The Chain-of-Thought prompt used in Task1.

The Direct Output prompt in Task1

Task:
Generate a challenging test input for the algorithm problem:
{problem_description}

Instructions:
- Focus on edge cases or scenarios that maximize the failure probability in faulty solutions.
- Due to the output length limit, you should generate a small-scale test input that is complete and valid.
- Output the test input directly, not code to generate it.

Output format:

````
```plaintext
{test input}
```
````

Only output the test input, no explanations.

Figure 6: The Direct Output prompt used in Task1.

The Chain-of-Thought prompt in Task2

Task:
Generate a challenging test input that exposes the bug in the buggy code of the algorithm problem:
Algorithm Problem: {problem_description}
Buggy Code: {buggy_code}

Instructions:
- Focus on edge cases or scenarios that maximize the failure probability in faulty solutions.
- Due to the output length limit, you should generate a small-scale test input that is complete and valid.
- Output the test input directly, not code to generate it.

Output format:

````
```plaintext
{test input}
```
````

Think step by step.

Figure 7: The Chain-of-Thought prompt used in Task2.

The Direct Output prompt in Task2

Task:
Generate a challenging test input that exposes the bug in the buggy code of the algorithm problem:
Algorithm Problem: {problem_description}
Buggy Code: {buggy_code}

Instructions:
- Focus on edge cases or scenarios that maximize the failure probability in faulty solutions.
- Due to the output length limit, you should generate a small-scale test input that is complete and valid.
- Output the test input directly, not code to generate it.

Output format:

````
```plaintext
{test input}
```
````

Only output the test input, no explanations.

Figure 8: The Direct Output prompt used in Task2.