ProBench: An Unbiased Benchmark for Evaluating Large Language Models in Competitive Programming

Anonymous ACL submission

Abstract

With the emergence of reasoning-optimized language models such as OpenAI-o1 and DeepSeek-R1, the reasoning and coding capabilities of large language models (LLMs) have seen substantial advancements. To address the deficiencies in existing benchmarks when evaluating the correctness of LLM-generated code, including insufficient complexity and rigor of test cases, as well as the lack of professionalism and consistency in evaluation environments, which lead to evaluation biases such as false positives and unfairness, we introduce ProBench, an unbiased benchmark for evaluating LLMs in competitive programming. ProBench constructs a complete, efficient, and automated online code evaluation pipeline that directly leverages the native judging services of competitive programming platforms to obtain authentic and reliable test results, thereby ensuring unbiased evaluation. Furthermore, ProBench comprises a comprehensive collection and integration of competitive programming problems sourced from Codeforces, Luogu, and Nowcoder, substantially enhancing the diversity of problem types and languages. Leveraging this carefully constructed benchmark, we evaluate mainstream LLMs across multiple dimensions. Our experimental results demonstrate that o4-mini achieves the highest overall score of 56.55. Further analysis points out the decoupling between the basic code generation capability and reasoning capability of LLMs, and based on this insight, it provides valuable guidance for the continuous development of reasoning-oriented language models.

1 Introduction

LLMs are transforming the way programmers approach coding tasks. These models can generate code snippets from natural language descriptions (Jiang et al., 2024a), debug and explain existing code (Jiang et al., 2024b), and provide suggestions for software design (Hou et al., 2024). Notably, models specifically designed for advanced reasoning, such as OpenAI-o1 (OpenAI, 2024) and DeepSeek-R1 (DeepSeek, 2025), demonstrate significant improvements over general-purpose LLMs in tasks requiring sophisticated cognitive processes. By simulating human reasoning and producing detailed intermediate steps before arriving at final solutions, these models excel in addressing complex coding challenges that demand high-level reasoning capabilities.

As LLMs continue to advance in their coding capabilities (Rozière et al., 2023; Hui et al., 2024; Guo et al., 2024), the complexity of early benchmark (Austin et al., 2021b; Chen et al., 2021a) has gradually become ineffective in distinguishing model performance. To address this limitation and to explore the performance boundaries of LLMs in coding, researchers have developed benchmarks derived from more challenging problems sourced from programming competitions. These benchmarks are designed to more effectively expose the limitations of current LLMs and enable more nuanced performance comparisons (Hendrycks et al., 2021a).

However, existing benchmarks lack the complex and specialized test cases (Liu et al., 2023; Jain et al., 2025) that are critically relied upon to evaluate the code generated by LLMs. These test cases are typically constructed and maintained collaboratively by professional problem setters and global participants, who consider intricate and stringent boundary conditions, and remain proprietary within the platforms. Consequently, benchmark developers must construct alternative test cases independently, yet these alternatives often exhibit limitations in both quantity and coverage, thereby introducing false positives in code evaluation. Meanwhile, existing benchmarks lack professional and consistent evaluation environments. Not only are they unable to accurately measure algorithmic efficiency and produce inconsistent evaluation re-

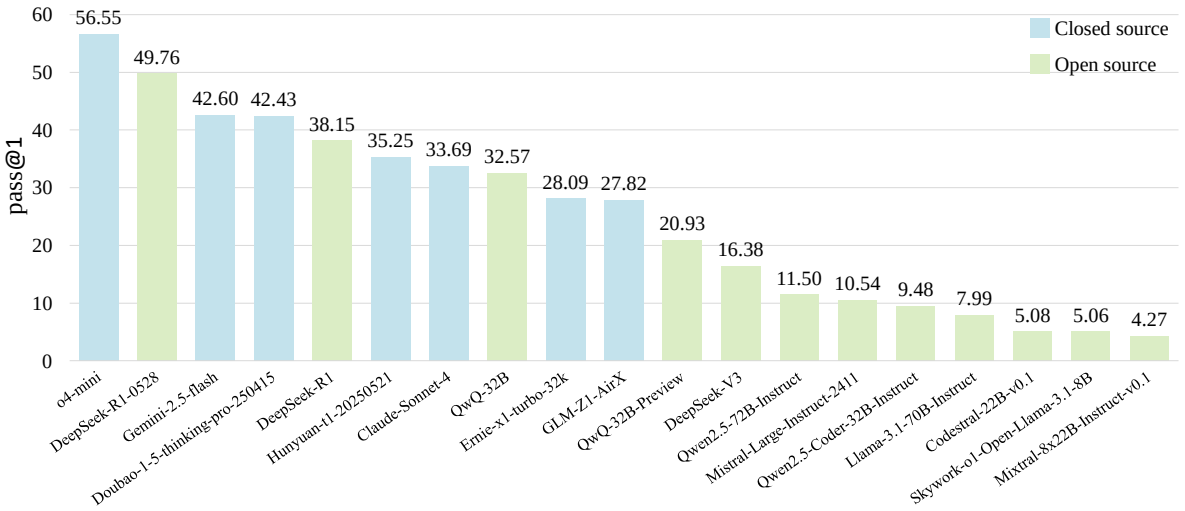


Figure 1: The pass@1 results on ProBench.

sults across different environments, but they also lack support for evaluating special problem types such as interactive problems. Furthermore, existing benchmarks still rely entirely on English problem descriptions, which prevents them from adequately evaluating the coding capabilities of LLMs in non-English scenarios.

To address these challenges, we propose ProBench, a benchmark that incorporates four key characteristics designed to ensure a reliable and comprehensive evaluation. First, ProBench includes problems of sufficient complexity to challenge contemporary LLMs. Second, and most importantly, the correctness of the code generated by LLMs is verified using online competitive programming platforms that host the benchmark problems. These platforms typically provide hundreds of test cases for each problem, which are collectively supervised and maintained by global participants, including various edge cases, as well as a professional and consistent evaluation environment, thereby ensuring unbiased code evaluation. Third, ProBench encompasses problems with natural language descriptions in both English and Chinese, sourced from three popular competitive programming platforms: Codeforces,[1] Luogu,[2] and Nowcoder.[3] Fourth, each problem in ProBench is annotated with comprehensive metadata, including difficulty levels and code evaluation results, facilitating detailed performance analysis and enabling fine-grained evaluation of LLMs. A comparative summary of ProBench and existing benchmarks for evaluating the coding capabilities of LLMs is provided in Table 1.

To construct ProBench, we collect problems and associated metadata, including difficulty levels and algorithmic tags, from online competitive programming platforms. To evaluate the performance of LLMs in coding, we prompt the models to generate solutions for these problems. The generated solutions include intermediate steps for solving the problems as well as the corresponding code. Subsequently, we address the technical challenges inherent in online code evaluation and develop a comprehensive, efficient, and automated online code evaluation pipeline that submits code to the corresponding source online competitive programming platforms to achieve unbiased code evaluation. Finally, we obtain execution results returned by these platforms, such as the number of test cases passed, runtime, and memory usage. The entire evaluation process is illustrated in Figure 2.

We evaluate mainstream LLMs. As shown in Figure 1, experimental results indicate that different LLMs exhibit significant performance differences on ProBench. Reasoning-optimized models outperform their size-matched counterparts, even surpassing larger models. Among these, o4-mini achieves the highest score of 56.55, while DeepSeek-R1-0528 attains the second-best performance, establishing itself as the leading open-source model. Moreover, leveraging the rich metadata of each problem and the granular feedback collected during evaluation, we perform a multidimensional investigation to thoroughly examine the

[1] https://codeforces.com
[2] https://www.luogu.com.cn
[3] https://ac.nowcoder.com

Benchmark	Difficulty Level	#Problems	Language	Unbiased Evaluation	Multiple Sources
APPS (Hendrycks et al., 2021b)	★★	10000	English	✗	✓
CodeContests (Li et al., 2022)	★★★★	165	English	✗	✓
xCodeEval (Khan et al., 2023)	★★	952	English	✗	✗
LeetCode-Hard (Shinn et al., 2024)	★	40	English	✗	✗
TACO (Li et al., 2023b)	★★	1000	English	✗	✓
LiveCodeBench (Jain et al., 2024)	★	511	English	✗	✓
USACO (Shi et al., 2024)	★★	307	English	✗	✗
CodeElo (Quan et al., 2025)	★★★	387	English	✓	✗
ProBench	★★★★★	790	English, Chinese	✓	✓

Table 1: Comparison of the ProBench against previous related benchmarks.

code reasoning capabilities of LLMs. Further analysis points out the decoupling between the basic code generation capability and reasoning capability of LLMs, and based on this insight, it provides valuable guidance for the continuous development of reasoning-oriented language models.

The main contributions of our work can be summarized into three aspects as follows.

- Introduction of ProBench: We propose ProBench, a novel benchmark specifically designed to evaluate the code reasoning capabilities of LLMs. ProBench consists of complex problems sourced from competitive programming platforms and includes both English and Chinese descriptions, making it suitable for multilingual evaluation.

- Reliable Evaluation Methodology: ProBench establishes a complete, efficient, and automated online code evaluation pipeline, utilizing official test cases provided by online programming competition platforms. This ensures high coverage, the inclusion of edge cases, as well as a professional and consistent evaluation environment, guaranteeing unbiased evaluation of code correctness.

- Comprehensive and Scalable Design and Analysis: ProBench includes rich problem metadata such as difficulty levels and code evaluation results. This supports fine-grained analysis and facilitates deeper insights into the reasoning abilities of LLMs across different problem categories, and further provides guiding opinions.

2 Related Work

Code Language Models and Reasoning Language Models. Recent years have witnessed emerging LLMs specifically tailored for code-related tasks, such as AlphaCode (Li et al., 2022), CodeLLaMa (Roziere et al., 2023), StarCoder (Li et al., 2023a) and CodeGeeX (Zheng et al., 2023). Trained on extensive open source code repositories, these models demonstrate exceptional performance in code generation benchmarks (Chen et al., 2021b; Austin et al., 2021a; Ni et al., 2024) and code search tasks (Lu et al., 2021; Huang et al., 2021; Khan et al., 2023). Despite significantly enhancing code generation, they exhibit limitations in reasoning capabilities, struggling to independently tackle tasks requiring deep logical reasoning, such as competitive programming and complex engineering requirements. Recent advances in reasoning language models, including OpenAI-o1 (OpenAI, 2024), DeepSeek-R1 (DeepSeek, 2025), and QwQ-32B-Preview (Qwen, 2024), which are trained using reinforcement learning techniques, have demonstrated human-competitive proficiency in handling complex programming challenges. In tasks requiring advanced reasoning skills, such as those encountered in mathematical competitions (Hendrycks et al., 2021c; Cobbe et al., 2021; Rein et al., 2023), these models have demonstrated performance surpassing that of the majority of human participants. This achievement poses significant challenges to conventional evaluation benchmarks, especially in coding.

Competitive Programming Benchmarks. To address the increasingly enhanced reasoning capabilities of models, several benchmarks feature problems drawn from programming competitions, including APPS (Hendrycks et al., 2021b), CodeContests (Li et al., 2022), xCodeEval (Khan et al., 2023), LeetCode-Hard (Shinn et al., 2024), TACO (Li et al., 2023b), LiveCodeBench (Jain et al., 2024), and USACO (Shi et al., 2024). The problems in these benchmarks are exclusively sourced

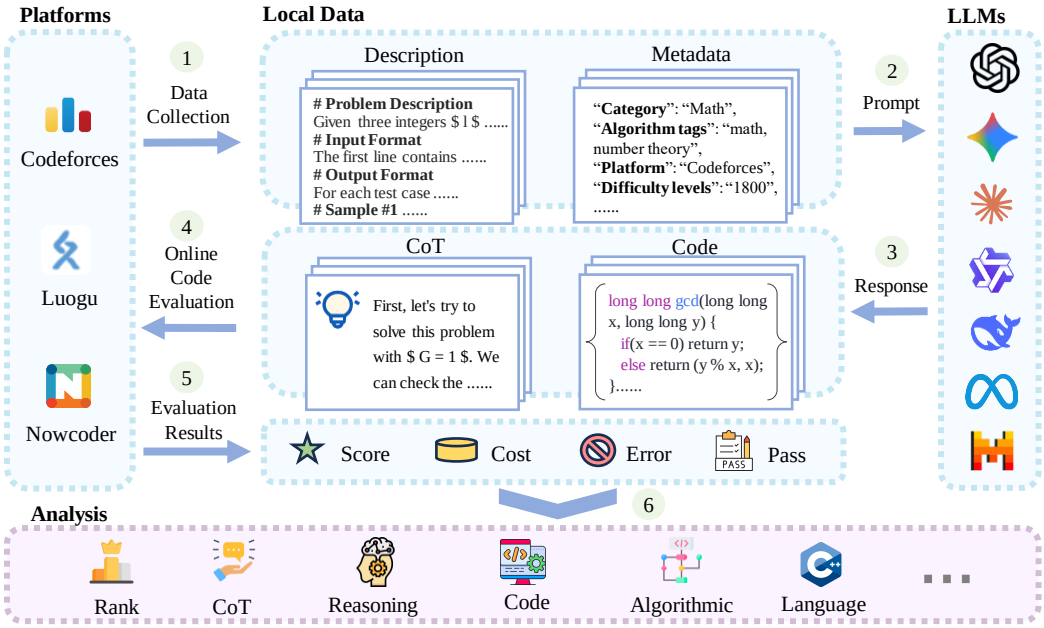


Figure 2: The complete evaluation pipeline of ProBench. From top left to bottom right, the evaluation process consists of the competition platform module, local data management module, LLMs module, and result analysis module. First, data are collected from various competition platforms, including problem descriptions, and relevant metadata. Next, the problem descriptions visible to participants are provided to the LLMs, which then generate problem-solving strategies and corresponding code solutions. These code solutions are subsequently submitted to the respective original platforms through the automated code submission pipeline we built for online evaluation, from which detailed evaluation results are obtained. Finally, analysis of the models' reasoning capabilities is conducted based on the complete locally stored data.

from English-language competition platforms or single platforms (Quan et al., 2025), lacking diversity in both language and content. More critically, the test cases, the most essential component for code evaluation, consist solely of extremely simple examples or cases generated by LLMs that lack rigorous boundary conditions for local offline code evaluation, rather than the test cases provided by the original online competitive programming platforms. These will lead to a high rate of false positives and will fail to accurately reflect model performance. Meanwhile, these benchmarks lack a professional and consistent evaluation environment. They not only fail to accurately measure algorithm efficiency and produce inconsistent evaluation results in different environments but also lack support for evaluating special types of problems such as interactive problems. In contrast, online evaluation mitigates these issues by using internal test cases that are supervised and maintained by global participants on the original problem platforms and the professional and consistent evaluation environment alleviate, providing a more reliable alternative. We discuss online evaluation in detail in Section 3.3, highlighting its advantages as well as the associated technical challenges. Table 1 presents a detailed comparison of our benchmark with existing benchmarks across multiple dimensions.

3 ProBench

We elaborate ProBench with a detailed account of its data collection, metadata integration, and online evaluation process. Figure 2 illustrates the complete evaluation pipeline of ProBench.

3.1 Data Collection

To enhance problem diversity and collect a large number of problems while minimizing the associated collection costs, we aggregate all publicly available problems from contests held between July and December 2024 across three online competitive programming platforms: Codeforces, Luogu, and Nowcoder. While several existing benchmarks (Quan et al., 2025) primarily source problems from Codeforces, this focus limits the diversity of their problem sets. In contrast, ProBench incorporates problems from two additional major platforms, Luogu and Nowcoder, both of which provide natural language problem descriptions in Chinese. This

Platform	Language	#Problems	Difficulty			Algorithm Tags						
			Easy	Medium	Hard	String	Basic	Math	Graph	Data Structures	Search	Dynamic Programming
Codeforces	English	446	173	139	110	31	389	235	59	152	128	128
Luogu	Chinese	63	30	23	10	3	31	16	5	11	7	9
Nowcoder	Chinese	281	102	40	12	7	70	50	13	27	13	16
Total	-	790	305	202	132	41	490	301	77	190	148	153

Table 2: Statistics of ProBench.

inclusion enhances the linguistic and topical diversity of the benchmark and enables the evaluation of LLMs' coding capabilities in Chinese.

3.2 Metadata Integration

Problem metadata varies significantly across different programming platforms. Each platform adopts its own criteria for evaluating difficulty and categorizing algorithmic tags. To facilitate systematic analysis, we perform unified integration of problem difficulty levels and algorithm tags from heterogeneous sources to establish consistent representations, leveraging the extensive experience of seasoned competitive programming experts (Dinshey, 2023). Detailed statistics of ProBench are presented in Table 2.

We systematically consolidate and, drawing on the experience of seasoned competitors, standardize difficulty descriptions from multiple platforms into three unified tiers : Easy, Medium, and Hard, corresponding to ICPC award criteria. A comprehensive cross-platform difficulty mapping is provided in Appendix B. In addition, solving programming problems requires not only profound thinking and reasoning skills but also mastery of relevant data structures and algorithms, collectively termed as algorithm tags. Given the heterogeneous taxonomy across platforms, we normalize these tags into seven knowledge categories. This categorization scheme considers both the cognitive requirements of different algorithms and enables precise analysis of models' logical reasoning

. Specific classification criteria and implementation details are documented in Appendix B.

3.3 Online Evaluation Mechanism

Competitive programming evaluation is highly dependent on complex and specialized test cases. Taking tree insertion, search, and deletion operations as an example, if the test cases contain only 100 simple operations, a traversal algorithm with $O(n)$ time complexity would suffice. However, when the number of operations reaches the order of 10^5 with a random distribution, the traversal algorithm will fail evaluation due to severe timeout issues, necessitating the implementation of complex data structures such as treap to achieve $O(\log n)$ time complexity. It is important to note boundary conditions: if the input operations exhibit specific patterns rather than a random distribution, the treap degrades to a linked structure, causing the time complexity to deteriorate to $O(n)$. In such cases, more optimized data structures, such as red-black trees or splay trees, must be introduced, which maintain the structural properties of the tree through balancing mechanisms, thereby ensuring stable $O(\log n)$ time complexity for operations. This demonstrates that comprehensive and rigorous test cases directly determine the accuracy of code evaluation, imposing extremely high demands on the professional expertise of problem setters.

Existing competitive programming benchmarks suffer from a scarcity of test cases. Even experienced problem setters find it difficult to exhaustively cover all boundary conditions. Consequently, competitive platforms such as Codeforces establish hacking phases after contests, allowing participants to review others' submitted code and construct boundary test cases to challenge seemingly correct solutions. Through this mechanism, the platform's test cases are effectively maintained and refined collectively by global participants. Precisely because test cases constitute an exceptionally valuable resource, platforms typically treat them as proprietary assets under strict confidentiality and do not make them publicly available. This leads to the situation where developers of existing competitive programming benchmarks must construct test cases independently, such as by obtaining only publicly available simple examples or generating cases through AI. However, these test cases often exhibit significant limitations in quantity, complexity, and boundary coverage, subsequently leading to false positive results in code evaluation.

Existing competitive programming benchmarks lack professional and consistent evaluation environments. Competitive programming

Model	Size	Reasoner	pass@				pass@1 for			Language	
			1	2	4	8	Easy	Medium	Hard	English	Chinese
o4-mini	🔒	✓	56.55	64.78	71.49	77.11	80.41	48.52	5.07	45.60	68.94
DS-R1-0528	37/671B	✓	49.76	56.94	62.32	65.93	72.05	39.62	1.83	43.61	58.34
Gemini-2.5-flash	🔒	✓	42.60	48.81	53.86	58.10	70.46	21.64	2.03	33.19	53.25
Doubao	🔒	✓	42.43	47.83	52.67	56.99	65.45	25.73	0.13	35.23	51.82
DS-R1	37/671B	✓	38.15	43.71	48.23	52.03	61.48	18.63	0.66	31.42	46.88
Hunyuan-t1	🔒	✓	35.25	40.99	45.43	49.25	57.65	14.61	0.00	28.84	43.51
Claude-Sonnet-4	🔒	✓	33.69	38.91	43.50	47.35	60.90	7.87	0.00	27.36	40.87
QwQ-32B	32B	✓	32.57	38.31	42.82	46.07	53.68	11.94	0.00	27.35	39.46
Ernie-x1-turbo	🔒	✓	28.09	33.72	38.71	42.69	47.41	8.75	0.00	23.39	34.25
GLM-Z1-AirX	🔒	✓	27.82	34.40	39.58	42.89	47.41	7.92	0.00	21.80	35.35
QwQ-32B-Preview	32B	✓	20.93	26.43	31.35	36.08	40.66	2.62	0.00	18.93	23.80
DS-V3	37/671B	✗	16.38	20.24	23.67	26.58	31.76	0.80	0.00	12.39	21.55
Qwen2.5-72B	72B	✗	11.50	14.39	16.97	19.24	23.24	0.37	0.00	8.66	15.19
Mistral-Large	123B	✗	10.54	13.87	17.26	20.89	20.82	0.37	0.00	8.07	13.74
Qwen2.5-Coder	32B	✗	9.48	12.73	15.80	18.48	17.91	0.56	0.00	5.41	14.75
Llama-3.1-70B	70B	✗	7.99	10.15	12.50	15.06	16.23	0.06	0.00	5.80	10.83
Codestral-v0.1	22B	✗	5.08	7.08	9.36	11.65	10.70	0.00	0.00	3.59	7.01
Skywork-o1	8B	✓	5.06	6.80	8.48	10.13	10.53	0.00	0.00	3.53	7.05
Mixtral-8x22B	22/176B	✗	4.27	5.85	7.49	9.11	8.61	0.00	0.00	2.83	6.14

Table 3: Evaluation results of the examined models on ProBench, sorted by pass@1. For detailed model information, refer to the Table 6.

requires not only verification of output correctness but also rigorous examination of algorithmic time complexity, space complexity, and the ability to handle special inputs, which depends on consistent compilation and execution environments, as well as strict resource monitoring mechanisms. However, existing offline local code evaluation benchmarks typically employ simplified evaluation approaches, which not only fail to accurately measure algorithmic efficiency but also produce inconsistent evaluation results across different environments. More critically, existing benchmarks cannot support the evaluation requirements of special problem types. For instance, interactive problems require the evaluation system to be equipped with an interactor program that dynamically verifies code correctness through multiple rounds of real-time interaction. The absence of a professional and consistent evaluation infrastructure makes it difficult to authentically reflect model performance in actual competitive scenarios, severely compromising the objectivity and reproducibility of the evaluation process.

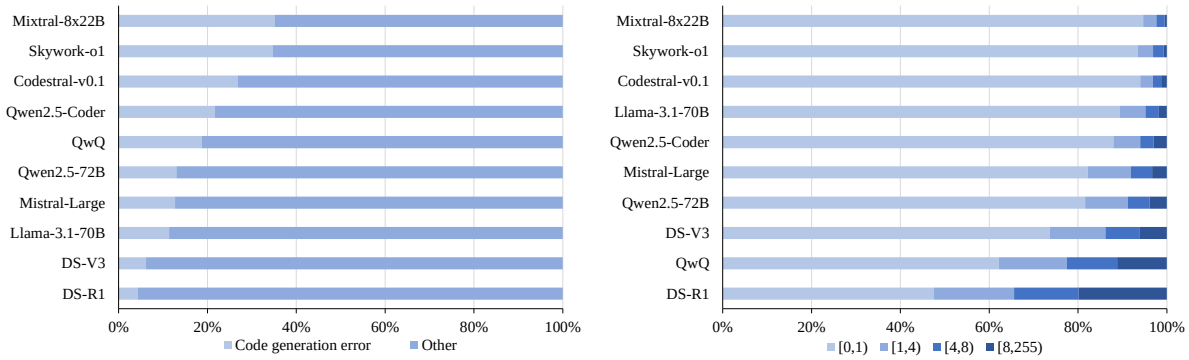
Online evaluation mechanism. To address the aforementioned limitations, we propose an online evaluation mechanism, whereby model-generated code is submitted to the original competitive platform of the problem and detailed evaluation results are obtained. Specifically, under strict compliance with platform usage agreements, we have constructed a complete, efficient, and automated online code evaluation pipeline by developing automated submission tools and purchasing relevant services. By leveraging the complex and specialized test cases within the platform, as well as the professional and consistent evaluation system environment, we ensure unbiased evaluation of code correctness, thereby enabling accurate evaluation of actual model performance.

4 Experiments and Analysis

We conducted a series of experiments to evaluate recent advanced LLMs on ProBench. Moreover, to analyze the reasoning capabilities of LLMs, we combined the metadata of the problems with the code evaluation results and conducted detailed discussions from different perspectives. In addition, we provide a more extensive statistical analysis in Appendix C.

4.1 Settings

Models. We evaluated open source LLMs from the DeepSeek, Qwen, Mistral, and Llama series, as well as closed source LLMs such as OpenAI, Gemini, and Claude. These include reasoning-optimized models, code-specialized models, and instruction-tuned models. The open source models cover a parameter scale ranging from 8B to



(a) Distribution of error types.

(b) Statistics of failed test cases.

Figure 3: (a) The proportion of error types reflecting insufficient basic code generation capabilities. (b) The number of passed test cases, counted in intervals. For example, the interval [1,4) represents the sum of the numbers of problems where 1, 2, or 3 test examples are passed.

671B. We provide detailed model information in Appendix D.

Evaluation Metrics. Unlike other benchmarks (Hendrycks et al., 2021b; Shi et al., 2024; Jain et al., 2024) that predominantly use Python as the target language for LLM-generated code, we adopted C++ as the primary evaluation language given its prevalence in competitive programming. To accommodate different description languages, we employed both English and Chinese prompts. For model evaluation, we utilized the established pass@k metric (Kulal et al., 2019; Chen et al., 2021b) by generating 8 candidate solutions per problem. We provide detailed prompt templates and experimental parameter settings in Appendix E.

Prompt. To facilitate more granular analysis of model performance, we preserve key metadata for each problem, including difficulty level, algorithm tags, and so on. However, in the prompts provided to LLMs, we retain only the information consistent with real-world competition scenarios (see problem descriptions in Appendix A), explicitly excluding any external cues, such as difficulty ratings or algorithmic tags, that might influence model reasoning.

4.2 Main Results

Table 3 presents the main results of the evaluation of the models examined. It can be observed that as the number of samples increases, the accuracy of all models exhibits an upward trend. o4-mini leads by a wide margin with a score of 56.55, while DeepSeek-R1-0528 ranks first among open source models and second overall with a score of 49.76. Notably, reasoning-optimized models significantly outperform non-reasoning models. For example, the reasoning-optimized model QwQ-32B, with merely 32B parameters, outperforms DeepSeek-V3 despite the latter's substantially larger parameter scale. Furthermore, the 8B-sized Skywork-o1-Open-Llama-3.1-8B demonstrates superior performance compared to the Mixtral-8x22B-Instruct-v0.1, while maintaining comparable performance with the Codestral-22B-v0.1.

Difficulty. As evidenced by the results presented in Table 3, model scores exhibit a marked decline with increasing problem difficulty. At the easy level, the scores of LLMs vary significantly, ranging from 8.61 to 80.41, and the scores decrease significantly at the medium and hard levels. This observation indicates that our benchmark can not only effectively distinguish differences in the reasoning capacities of models but also has sufficient difficulty to meet the future evaluation needs of reasoning-optimized models.

Multilingual Competence. Further analysis of language-specific performance, after controlling for variations in problem difficulty, reveals problem horizontal comparison of model performance across Chinese and English contexts. The results demonstrate robust bilingual support across all evaluated models. Moreover, the higher scores in Chinese compared to English are consistent with the distribution of easy-difficulty questions across different languages shown in Table 2, where easy questions account for 41% of English questions and 61% of Chinese questions.

4.3 Decoupling of Generation Capability and Reasoning Capability

Basic code generation capability exhibits a positive correlation with model scale. In the competitive platform, code that fails to pass returns detailed

error types, such as "Compilation_Error" and "Wrong_Answer". We computed the proportion of error types reflecting insufficient basic code generation capability, such as "Compilation_Error", indicating that the code cannot pass static compiler checks, suggesting inadequate mastery of syntactic rules or issues such as inconsistent variable naming, for each model. The complete error type classification is provided in Appendix F. As illustrated in Figure 3a, a larger model scale corresponds to lower code error rates, demonstrating that basic code generation capability exhibits a significant positive correlation with model scale. This indicates that large-scale models can more thoroughly learn the syntactic rules and code structural patterns of programming languages during the pre-training phase, thereby generating syntactically correct and structurally complete code.

Decoupling of reasoning capability. As describe in Section 3.3, the evaluation of competitive programming relies heavily on complex and specialized test cases. During evaluation, competition platforms execute test cases sequentially in order from simple to complex, verifying the correctness of code output until encountering an erroneous output or achieving complete passage, and return the number of passed test points after evaluation. Consequently, a higher number of passed test points indicates stronger robustness of the code. Figure 3b presents statistics on the number of test points passed by each model. Combined with Table 3, it can be observed that code robustness exhibits a clear positive correlation with model reasoning capability, implying that models with stronger reasoning capability can engage in deeper and more comprehensive deliberation. Synthesizing the results from Figure 3a, we can discern a certain decoupling relationship between models' basic code generation capability and reasoning capability: the former depends primarily on model scale, whereas the latter relies more heavily on model architecture design and training strategies.

Based on the above findings, we can provide the following instructive recommendations for model training and optimization: (1) *Staged training strategy*: In response to the decoupling characteristics of basic code generation capability and reasoning capability, a staged training strategy can be adopted. During the pre-training phase, priority should be given to utilizing large-scale code corpora to reinforce the model's syntactic mastery and code structure comprehension. In the subsequent fine-tuning phase, emphasis should be placed on introducing tasks requiring deep reasoning, such as algorithmic competitions, to enhance the model's logical reasoning capabilities. (2) *Differentiated model selection*: For resource-constrained application scenarios, if tasks primarily involve syntactically correct code generation (e.g., code completion and simple script writing), models with larger scales but moderate reasoning capability can be selected to avoid resource consumption from long chains of thought. For scenarios requiring the solution of complex algorithmic problems, models with strong reasoning capability should be prioritized, even if their scales are relatively smaller. (3) *Targeted data augmentation*: Based on the performance of the model on different types of errors, targeted data augmentation strategies can be designed. For example, for models with high compilation error rates, training strategies that incorporate syntax correction and code repair tasks can be augmented. Such refined training methodologies can more efficiently enhance model capabilities along specific dimensions, avoiding increased computational costs from blindly expanding model scale.

5 Conclusion

We introduce ProBench, a challenging benchmark designed to conduct an unbiased evaluation of LLMs' competitive programming reasoning and coding capabilities in both English and Chinese contexts. ProBench integrates complex problems from competitive programming platforms and submits the code generated by LLMs to these platforms for evaluation through a complete, efficient, and automated online code evaluation pipeline that we have constructed. This approach ensures high coverage, inclusion of edge cases, and a professional and consistent evaluation environment, thereby guaranteeing unbiased evaluation of code correctness and model performance. Furthermore, ProBench provides rich metadata, including problem difficulty levels and code evaluation results, to facilitate comprehensive and extensible analysis. Experimental results demonstrate that different LLMs exhibit significant performance differences on ProBench, and their pass@1 scores are only at a medium to low level. Further analysis reveals a decoupling phenomenon between LLMs' basic code generation capabilities and reasoning capabilities, offering valuable guidance for the continued development of reasoning-oriented language models.

Limitations

We focus on evaluating the code reasoning capabilities of LLMs in a one-shot solving scenario. While this non-interactive setting differs from the real-world environment where human participants can perform code self-testing, it enables the establishment of a more equitable evaluation benchmark, wherein all models compete under identical constraints and the evaluation results possess complete reproducibility. This design choice allows us to isolate and quantify the models' core code reasoning capability. Interactive scenarios requiring iterative debugging can be explored as an independent research direction.

Ethical Statement

We sincerely thank the Codeforces, Luogu, and Nowcoder platforms for their excellent infrastructure, which has provided strong support for our research. For the Luogu website that offers relevant services, we obtained the usage authorization by purchasing its services.[4] For websites that do not provide such services, throughout the entire experimental process, we strictly adhered to the terms of use of Codeforces[5] and Nowcoder,[6] ensuring that all experiments were solely for academic purposes and used within a limited scope to avoid affecting the normal operation of the website. Furthermore, in accordance with the Codeforces Rule Restricting the Use of AI,[7] it is designed only for virtual participation and not for in-contest testing.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021a. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé 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. *CoRR*, abs/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, and 1 others. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

DeepSeek. 2025. Deepseek-r1 release.

Dinshey. 2023. Difficulty levels of programming contests.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021a. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021b. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021c. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(8):220:1–220:79.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20,000+ web queries for code search and question answering. *arXiv preprint arXiv:2105.13239*.

[4] https://docs.lgapi.cn/open/
[5] https://codeforces.com/terms
[6] https://www.nowcoder.com/html/disclaimer
[7] https://codeforces.com/blog/entry/133941

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, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

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

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, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024a. A survey on large language models for code generation. *CoRR*, abs/2406.00515.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2024b. Ledex: Training llms to better self-debug and explain code. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

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

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023a. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

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, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, and 1 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, and 1 others. 2024. L2ceval: Evaluating language-to-code generation capabilities of large language models. *Transactions of the Association for Computational Linguistics*, 12:1311–1329.

OpenAI. 2024. Openai o1-mini.

Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, and 1 others. 2025. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings. *arXiv preprint arXiv:2501.01257*.

Qwen. 2024. Qwq: Reflect deeply on the boundaries of the unknown.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. 2023. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, and 6 others. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. 2024. Can language models solve olympiad programming? *arXiv preprint arXiv:2404.10952*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, and 1 others. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

A Problem Description

Figure 4 illustrates a typical task from the basic code generation benchmark HumanEval, including the problem description, the reference solution, and the complete test cases. These tasks directly prompt LLMs to generate functions with specific functionality. Both the solutions and test cases are relatively simple. Figure 5 presents the description of a competitive programming problem 2043/D from Codeforces. Figure 6 shows the reasoning process for solving this problem, while Figure 7 provides the corresponding implementation code. The input-output examples given in the problem description are only simple sample cases intended for preliminary validation by contestants. More complex test cases (with parameters such as l, r, G on the order of 10^{18}) are hidden within the online judge system. It is evident that, unlike HumanEval tasks, this type of problem does not explicitly require LLMs to produce code directly. Instead, it necessitates complex reasoning, selection of appropriate algorithmic strategies, and ultimately the generation of fully executable code capable of passing stringent test cases under tight time and memory constraints.

B Metadata Integration Details

Table 4 shows the complete difficulty classification of the problems. The ones from Luogu are originally in Chinese on the original website, and we have translated them into English for display. The classification is based on award tiers corresponding to ICPC, though it does not constitute a rigorous standard in practical applications.

Table 5 presents a classification of complete algorithm labels, with each category summarizing the complete algorithm labels from three websites. For Chinese labels, we have translated them into English.

C Additional Statistical Analysis

In this section, we present some additional statistical analyses.

C.1 Length of Responses

We counted the length of responses from each model (including CoT). To ensure consistency in comparison, we used the cl100k_base encoding in tiktoken to calculate the average length. As shown in Figure 8, the length of reasoning-optimized models is significantly longer than that of non-reasoning models. Moreover, for reasoning-optimized models, the length of failed problems at medium and hard difficulty levels is significantly longer than that of successfully solved problems at the easy difficulty level.

C.2 Algorithmic Tags

We compiled statistics on the performance of various models across different data structures and algorithms at the easy difficulty level. As shown in Figure 9a, most models achieve high scores in more basic data structures and algorithms, such as those tagged with String, Basic, and Math, while scoring lower in more advanced ones, such as Graph, Data Structures, Search, and Dynamic Programming.

C.3 Code Language

In addition to C++, we selected five models to generate code in Java and Python, commonly used languages among programming contestants, to investigate their code generation capabilities across different programming languages. Figure 9b demonstrates that evaluation results across the three programming languages remain largely consistent, fluctuating within approximately 2%. This suggests that the reasoning capabilities of the models are largely independent of the programming languages used, with similar learning capacities observed across the three programming languages.

D Model Details

During preliminary analysis, we observed that non-reasoning models with fewer parameters exhibit limited reasoning capabilities, often failing to complete even elementary tasks. To maintain evaluation efficiency and prevent the inclusion of underperforming models in the competition platform, we consequently excluded non-reasoning models with below 14B parameters from our formal evaluation. Table 6 presents the details of the models we evaluated, including the model link, number of parameters, type, and release date.

E Additional Evaluation Metrics

We use the prompts of Figure 10 for generating model responses. The Chinese question descriptions are translated into English and then used accordingly.

For evaluation metrics, while explicit pass@k results were reported where applicable, all other evaluations defaulted to pass@1. During code generation, we maintained each model's default hyperparameters (including temperature, top_p, and top_k) while setting max_tokens to the model's default context length minus the prompt length.

F Error Category

We classify errors on Codeforces and Nowcoder as follows: "COMPILATION_ERROR", "RUNTIME_ERROR", "IDLENESS_LIMIT_EXCEEDED", "Execution error", "Segmentation fault", and "Floating point error" are categorized as code generation errors.

prompt:

```
from typing import List


def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each
other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

canonical_solution:

```
sorted_numbers = sorted(numbers)
    for i in range(len(sorted_numbers) - 1):
        if sorted_numbers[i + 1] - sorted_numbers[i] < threshold:
            return True
    return False
```

test:

```
def check(candidate):
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
    assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False
```

Figure 4: Problem of HumanEval.

Platform	Easy	Medium	Hard
Codeforces	$[800, 1500]$	$(1500, 2400]$	$(2400, 3500)$
Luogu	Beginner, Basic- Basic/Intermediate-	Intermediate+/Advanced Advanced+/Provincial-	Provincial/NOI- NOI/NOI+/CTSC
Nowcoder	$[0, 1500]$	$(1500, 2300]$	$(2300, \infty)$
ICPC award tiers	Regional bronze and silver	Regional gold Eligibility for final	Winning the final

Table 4: Difficulty levels of different platforms.

D. Problem about GCD

time limit per test: 1 second
memory limit per test: 256 megabytes

Given three integers l, r, and G, find two integers A and B ($l \leq A \leq B \leq r$) such that their greatest common divisor (GCD) equals G and the distance $|A - B|$ is maximized.

If there are multiple such pairs, choose the one where A is minimized. If no such pairs exist, output "`-1 -1`".

Input

The first line contains a single integer t ($1 \leq t \leq 10^3$) — the number of test cases. Then, t test cases follow.

Each test case consists of a single line containing three integers l, r, G ($1 \leq l \leq r \leq 10^{18}$; $1 \leq G \leq 10^{18}$) — the range boundaries and the required GCD.

Output

For each test case, output two integers A and B — the solution to the problem, or "`-1 -1`" if no such pair exists.

Example

input Copy

```
4
4 8 2
4 8 3
4 8 4
5 7 6
```

output Copy

```
4 6
-1 -1
4 8
6 6
```

Figure 5: Problem description of 2043/D.

2043D - Problem about GCD

First, let's try to solve this problem with $G = 1$. We can check the pair (l, r). If its greatest common divisor is not 1, then we should check $(l, r-1)$ and $(l+1, r)$, i. e. the pairs on the distance $(r - l - 1)$. If these don't work, we can check $(l, r-2)$, $(l+1, r-1)$ and $(l+2, r)$, and so on, and the answer will be located fast enough (more about that in the third paragraph of the editorial). So, we get a solution in $O(K^2 \log A)$ per test case, where A is the bound on the integers on the input, and K is the decrease in distance we had to make to find this pair (i. e. if the answer has distance $|A - B|$, then $K = |r - l| - |A - B|$).

What to do if $G \neq 1$? Almost the same, but first, we need to ensure that the first pair we try has both integers divisible by G. So, if $l \bmod G \neq 0$, let's shift it to the next closest integer which is divisible by G; and if $r \bmod G \neq 0$, let's subtract $r \bmod G$ from r to make it divisible. Then, we make the same process, but instead of trying pairs like $(l, r-1)$, $(l+1, r)$, $(l, r-2)$ and so on, we try $(l, r-G)$, $(l+G, r)$, $(l, r-2G)$, and so on.

Okay, now let's talk about why this works fast, i. e. why this K in the complexity formula is not that big. All the following paragraphs will assume $G = 1$, but we can use the same reasoning with $G > 1$ if we divide everything by G.

Intuitively, we can think about it in terms of prime gaps: as soon as $r - K$ becomes a prime number, we get our result. Average gap between two primes is about lnA, but there can be pretty big gaps, more than 1000. If you're bold and brave, you can stop here and submit, but let's find a better bound.

Instead of thinking about the gap between two primes, let's think about the gap between two numbers which are coprime with l. Let's assume that l is the product of several first prime numbers (if it is not, integers which are coprime with l will appear even more often). $\frac{1}{2}$ of all integers are not divisible by 2; $\frac{2}{3}$ of them are not divisible by 3; $\frac{4}{5}$ of them are not divisible by 5, and so on. If we repeat this process until the product of primes we considered becomes too large, we can get that, on average, 1 in 7 or 8 integers is coprime with l. This is a better bound, but it still uses "average" gaps. However, this should be enough to try to submit the solution.

Okay, now let's show a rigorous proof (which you really shouldn't need during the contest) of some reasonable bound. We can prove that if you consider all possible pairs of integers from intervals $[l, l+30)$ and $(r-30, r]$, you will find at least one coprime pair.

There are 30 integers in each interval, so there are 900 pairs to consider. Suppose in some of them, both integers are divisible by 2. There will be at most 225 such pairs, so we are left with 675 pairs.

Suppose in some of the remaining pairs, both integers are divisible by 3. There will be at most 10 integers divisible by 3 in each segment, so at most 100 pairs. We are left with 575 pairs.

Suppose in some of the remaining pairs, both integers are divisible by 5. There will be at most 6 integers divisible by 5 in each segment, so at most 36 pairs. We are left with 539 pairs.

If we repeat this until some prime number like 37, we will still have to "fix" more than 450 pairs, so at least one number has 15 or more pairs to "fix". This should mean that it is divisible by at least 15 primes which are greater than 37, and it means it's greater than 10^{18}. So, in every pair of intervals $[l, l+30)$ and $(r-30, r]$ such that the numbers are not greater than 10^{18}, there will be at least one coprime pair, and this proves that $K \leq 60$. In practice, it is much lower since, in our proof, we didn't consider the fact that a lot of pairs will have more than 1 prime which they are divisible by; if we take this into account (or repeat the same process until primes become much greater), we can prove tighter bounds, for example, $K = 40$.

Figure 6: Tutorial of 2043/D.

```
#include<bits/stdc++.h>

using namespace std;

long long gcd(long long x, long long y)
{
    if(x == 0) return y;
    else return gcd(y % x, x);
}

void solve()
{
    long long l, r, g;
    scanf("%lld %lld %lld", &l, &r, &g);
    long long L = l + (l % g == 0 ? 0 : g - (l % g));
    long long R = r - r % g;
    for(int i = 0; i <= (R - L) / g; i++)
        for(int j = 0; j <= i; j++)
            if(gcd(L + j * g, R - (i - j) * g) == g)
            {
                printf("%lld %lld\n", L + j * g, R - (i - j) * g);
                return;
            }
    puts("-1 -1");
}

int main()
{
    int t;
    scanf("%d", &t);
    for(int i = 0; i < t; i++) solve();
}
```

Figure 7: Corresponding implementation code of 2043/D.

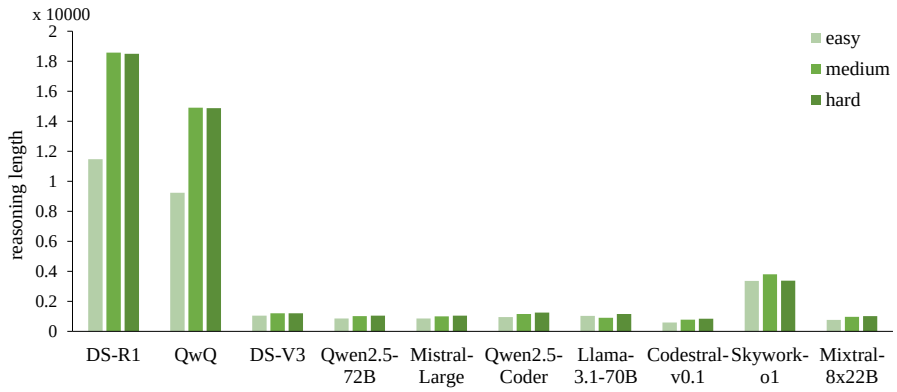
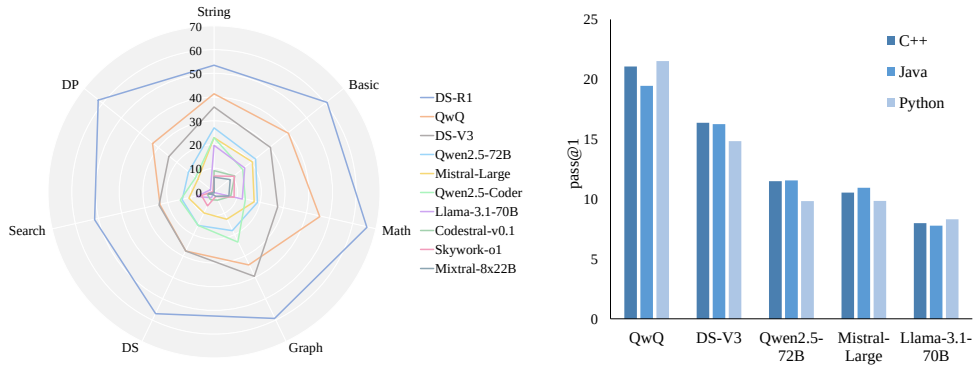


Figure 8: Presents the responses length.



(a) Performance of algorithmic tags.

(b) Capability of programming languages.

Figure 9: (a) Presents the performance across different data structures and algorithms. (b) The model's performance to generate different programming languages.

Category	Algorithm tags
Basic	greedy, implementation, brute force, constructive algorithms, sortings, two pointers, divide and conquer, bitmasks, simulation, construction, enumeration, recursion, two-pointer, thinking, violence, divide and conquer, base conversion, Ad-hoc, classification discussion, bitwise operations, randomization, discretization, prefix sums, differences
Search	dfs and similar, meet-in-the-middle, binary search, ternary search, search, breadth-first search BFS, depth-first search DFS, breadth-first search(BFS)
String	strings, string suffix structures, expression parsing, trie trees, kmp and extended kmp, suffix arrays(SA), string hash
Dynamic Programming	dynamic programming, dynamic programming DP, knapsack DP, tree DP, dp, state compression enumeration, knapsack problem, interval dp
Data Structures	data structures, trees, dsu, hashing, STL, segment trees, scanlines, tree theory, monotonic stacks, violent data structures, union-find, queues, heaps, data structures, trees, stacks, binary indexed trees, ST tables, doubling, chairman trees, hash, blocking, balanced trees, tree of trees, RMQ, square root decomposition
Graph	graphs, shortest paths, graph matchings, flows, 2-sat, graph theory, connected components, topological sorting, graph modeling, Tarjan, biconnected components, square trees, spanning trees, connectivity, shortest paths, network flows, tree decomposition
Math	math, number theory, combinatorics, geometry, matrices, probabilities, fft, Chinese remainder theorem, games, number theory, combinatorics, harmonic series, indeterminate equations, pigeonhole principle, linear algebra, computational geometry, probability expectation, prime factorization, gcd and exgcd, mobius inversion, linear basis, inclusion-exclusion principle and pigeonhole principle, sieve method, quadrilateral inequality, mathematics, matrix multiplication, game theory, polynomials, schedules

Table 5: Algorithm categories and their topics.

Short Name	Model	Parameter	Type
o4-mini	OpenAI/o4-mini	🔒	Reasoner
Gemini-2.5-flash	Google/Gemini-2.5-flash	🔒	Reasoner
Claude-Sonnet-4	Anthropic/Claude-Sonnet-4	🔒	Reasoner
Doubao	ByteDance/Doubao-1-5-thinking-pro-250415	🔒	Reasoner
Hunyuan-t1	Tencent/Hunyuan-t1-20250521	🔒	Reasoner
Ernie-x1-turbo	Baidu/Ernie-x1-turbo-32k	🔒	Reasoner
GLM-Z1-AirX	ChatGLM/GLM-Z1-AirX	🔒	Reasoner
DS-V3	deepseek-ai/DeepSeek-V3	37/671B	Instruct
DS-R1	deepseek-ai/DeepSeek-R1	37/671B	Reasoner
DS-R1-0528	deepseek-ai/DeepSeek-R1-0528	37/671B	Reasoner
QwQ-32B-Preview	Qwen/QwQ-32B-Preview	32B	Reasoner
Qwen2.5-Coder	Qwen/Qwen2.5-Coder-32B-Instruct	32B	Coder
Qwen2.5-72B	Qwen/Qwen2.5-72B-Instruct	72B	Instruct
Mixtral-8x22B	mistralai/Mixtral-8x22B-Instruct-v0.1	22/176B	Instruct
Codestral-v0.1	mistralai/Codestral-22B-v0.1	22B	Coder
Mistral-Large	mistralai/Mistral-Large-Instruct-2411	123B	Instruct
Llama-3.1-70B	meta-llama/Llama-3.1-70B-Instruct	70B	Instruct
Skywork-o1	Skywork/Skywork-o1-Open-Llama-3.1-8B	8B	Reasoner

Table 6: Model details.

```
messages = [
    {
        "role": "system",
        "content": "You are a highly skilled competitive
           programming expert, adept at analyzing complex problems
           and designing efficient solutions. Your task is to solve
            the following programming challenge. Always maintain
           clear logic and think step by step."
    },
    {
        "role": "user",
        "content": """
            Your task is to carefully read the following problem
               description, analyze the problem step by step, and
               clearly explain your thought process. Finally, write
                the solution in {lang} and ensure the code is
               correct and readable. Wrap your code in the
               following format:

            ```{lang_type}
            {lang} solution code
            ```

            Here is the problem description:

            {description}
            '''
        """
    }
]
```

Figure 10: Prompts for generating model responses.