



CodeJudge-Eval: Can Large Language Models be Good Judges in Code Understanding?

◇ Yuwei Zhao*, ♦ Ziyang Luo*, ♥ Yuchen Tian, ♦ Hongzhan Lin

♣ Weixiang Yan, ◇ Annan Li, ♦ Jing Ma[†]

♦ Hong Kong Baptist University, ◇ Beihang University

♥ University of Tokyo, ♦ Vaneval.AI

{yuweizhao, liannan}@buaa.edu.cn

{cszyluo, majing}@comp.hkbu.edu.hk

Abstract

Recent advancements in large language models (LLMs) have showcased impressive code generation capabilities, primarily evaluated through language-to-code benchmarks. However, these benchmarks may not fully capture a model’s code understanding abilities. We introduce **CodeJudge-Eval (CJ-Eval)**, a novel benchmark designed to assess LLMs’ code understanding abilities from the perspective of code judging rather than code generation. **CJ-Eval** challenges models to determine the correctness of provided code solutions, encompassing various error types and compilation issues. By leveraging a diverse set of problems and a fine-grained judging system, **CJ-Eval** addresses the limitations of traditional benchmarks, including the potential memorization of solutions. Evaluation of 12 well-known LLMs on **CJ-Eval** reveals that even state-of-the-art models struggle, highlighting the benchmark’s ability to probe deeper into models’ code understanding abilities. Our codes and benchmark are available at <https://github.com/CodeLLM-Research/CodeJudge-Eval>.

1 Introduction

Recently, powerful large language models (LLMs) such as GPT-4o (OpenAI, 2023), Gemini (Anil et al., 2023), and Claude (Anthropic, 2023) have demonstrated impressive code generation capabilities. These models are being used to develop tools that assist in software development (Hong et al., 2024; Yang et al., 2024). The primary method the community uses to evaluate the coding abilities of these LLMs is based on popular language-to-code benchmarks, such as HumanEval (Chen et al., 2021), APPS (Hendrycks et al., 2021) and MBPP (Austin et al., 2021), where LLMs are tasked with generating code based on task descriptions. If the generated code can pass the pre-

*Equal Contribution. Ziyang Luo is the project lead.

[†]Corresponding Author.

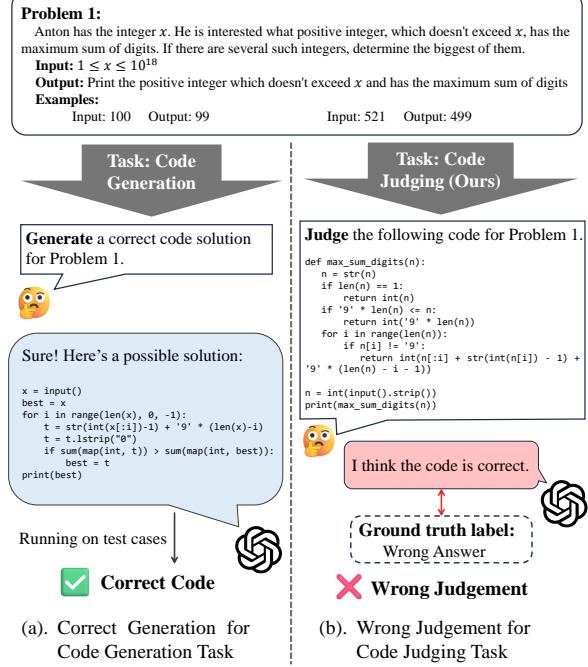


Figure 1: Comparing code generation with code judging task, we observe that a model’s ability to generate correct code does not necessarily imply it can accurately judge other codes for the same problem.

designed test cases, the LLMs are considered to have successfully solved the coding tasks.

While language-to-code benchmarks have significantly advanced the coding capabilities of LLMs, the assumption that a model’s ability to pass pre-designed test cases for a specific task equates to a full understanding of that task does not always hold true (Dou et al., 2024). These test cases may not comprehensively cover all potential inputs and edge cases (Liu et al., 2023), and concerns such as data leakage can further undermine the reliability of such evaluations (Dong et al., 2024; White et al., 2024). To overcome these challenges, we draw inspiration from modern educational theory, which suggests that if someone can accurately judge the correctness of other candidate

solutions for a given task, they are likely to fully understand that task (Care et al., 2012). Building on this insight, we introduce a novel benchmark, **CodeJudge-Eval (CJ-Eval)**, aimed at evaluating the code understanding abilities of LLMs by positioning them as code judges, as shown in Figure 1.

Unlike traditional approaches that require LLMs to generate code, **CJ-Eval** assesses their ability to evaluate the correctness of provided candidate solutions, determining whether they result in a correct output or errors such as Wrong Answer, Time Limit Exceeded, or other errors. Although unit tests can verify code correctness directly, our objective is to evaluate the inherent code understanding abilities of LLMs without relying on external tools, thereby reducing the need for diverse and high-quality unit tests across different coding tasks. Moreover, the LLM-as-a-Judge paradigm is already widely adopted in the general domain, as evidenced by frameworks such as MT-Bench (Zheng et al., 2023a) and AlpacaEval (Li et al., 2023c).

Additionally, evaluating the model using the code judging paradigm also offers new insights from a data perspective. Previous research has shown that a 7B model can memorize more knowledge than English Wikipedia (Allen-Zhu and Li, 2024), making it likely that the model could pass the code generation evaluation by merely memorizing one correct solution per problem. LiveCodeBench (Jain et al., 2024) and LiveBench (White et al., 2024) address this issue by adding new data to the benchmark. In contrast, our code judge evaluation assesses each code solution, and the number of code submissions is often much greater than the number of problems¹, making it harder for the model to memorize all solutions.

To construct our **CJ-Eval** benchmark, we choose to select problems from the APPS test set, which includes 5,000 coding problems across three different difficulty levels, offering significantly more diversity than smaller benchmarks like HumanEval and MBPP. To generate candidate code solutions for each problem, we utilized 16 different LLMs, encompassing both open- and closed-source, as well as general and code-specific models. We then applied our fine-grained judging system, using a comprehensive set of test cases to obtain execution results that serve as the ground-truth judging annotations. To create a curated benchmark, we

¹For instance, Codeforces, a famous programming website, has produced only approximately 9,800 problems over the past 14 years but has 2.7×10^8 solution codes.

meticulously filtered the original 80,000 solutions down to 1,860 solution codes and structured the questions into a multiple-choice format.

We evaluated 12 different proprietary and open-source LLMs using our **CJ-Eval** benchmark. The results indicate that our benchmark is quite challenging. While proprietary LLMs such as GPT-4o and Claude-3.5-Sonnet outperform open-source models, their macro F1 scores peak at a modest 50 on the simplest code judge tasks. This indicates a significant gap between their performance and the highlighting room for considerable improvement. Additionally, our analysis reveals that while some models can generate correct code for certain tasks, this does not necessarily mean they can accurately judge other codes for the same tasks. This suggests that our benchmark provides a new perspective for assessing the code understanding abilities of both proprietary and open-source LLMs.

2 Related Work

Various benchmarks are used to evaluate LLMs' coding abilities. For Python code generation on relatively simple task descriptions, HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) are the most popular benchmarks. EvalPlus (Liu et al., 2023) enhance HumanEval and MBPP by adding more test cases. ReCode (Wang et al., 2023) modifies HumanEval by changing function names and docstrings to create a benchmark for code generation robustness. Extensions like HumanEval-X (Zheng et al., 2023b), MultiPL-E (Cassano et al., 2022), and MBXP (Athiwaratkun et al., 2023) adapt HumanEval and MBPP to include programming languages beyond Python. APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), and TACO (Li et al., 2023b) introduce more challenging coding problems. MMCode (Li et al., 2024) extends these competition-level coding tasks with multimodal information. CodeHalu (Tian et al., 2024) evaluates various hallucinations in code generation. DS-1000 (Lai et al., 2022), NumpyEval (Zan et al., 2022), and PandasEval (Jain et al., 2022) focus on data science code generation.

Additionally, a variety of code benchmarks exist for tasks such as code translation (Rozière et al., 2020; Yan et al., 2023b; Ahmad et al., 2023), test case generation (Wang et al., 2024; Li and Yuan, 2024), code search (Husain et al., 2019), commit message generation (Schall et al., 2024), code summarization (Sun et al., 2024), program re-

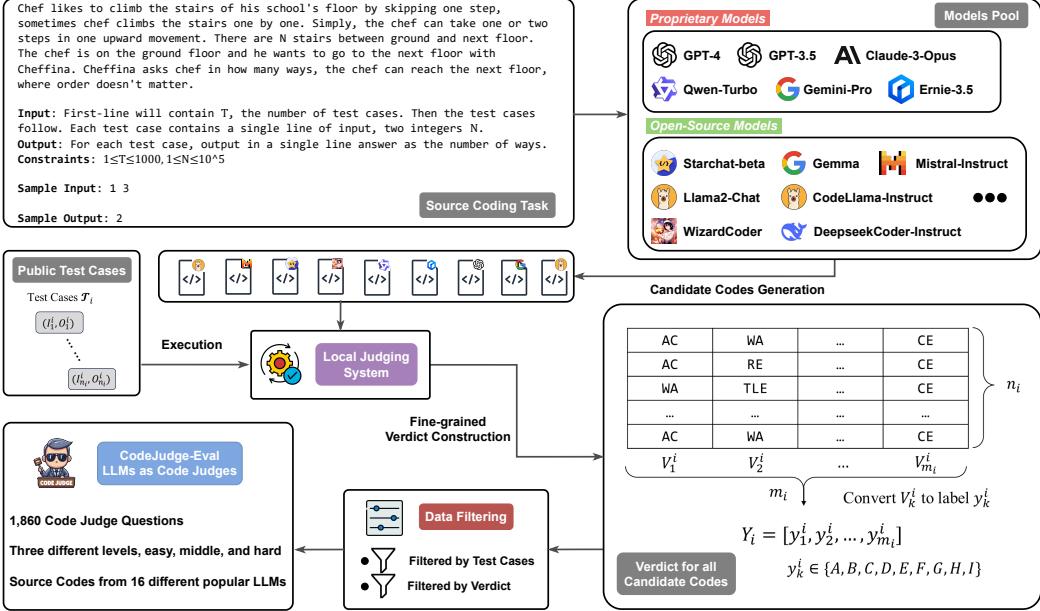


Figure 2: An overview of our pipeline for constructing the **CodeJudge-Eval** benchmark.

pair (Muennighoff et al., 2023; Ye et al., 2021; Yan et al., 2023a), code execution (Gu et al., 2024b), and repository-level code generation (Liu et al., 2024; Jimenez et al., 2024). However, these benchmarks predominantly focus on code generation based on given requirements and often rely on high-quality test cases to evaluate the correctness of generated code, which can be susceptible to data leakage issues (Dong et al., 2024).

In contrast, our **CJ-Eval** evaluates LLMs' code understanding abilities from the perspective of LLMs acting as code judges. This approach does not depend on external unit testing; instead, it requires LLMs to evaluate various solutions for the same task, increasing the difficulty of memorizing all possible solutions. Similar efforts include ICE-Score (Zhuo, 2024), which introduces a metric for evaluating the usefulness of code generated by LLMs, and the study by Gu et al. (2024a), which examines LLMs' challenges in understanding the nuances of their own incorrect generations. However, our work uniquely introduces a benchmark focused specifically on code correctness judgment, evaluating whether LLMs can assess code not only generated by themselves but also by other LLMs.

3 CodeJudge-Eval

3.1 Overview

As shown in Figure 2, we introduce the construction pipeline of our **CJ-Eval** benchmark. It consists

of N problems, denoted as P_1, \dots, P_N . A problem P_i with n_i test cases and m_i solution codes can be formatted as

$$P_i = (S_i, \mathcal{T}_i, \mathcal{C}_i, Y_i, [V_1^i, \dots, V_{m_i}^i]). \quad (1)$$

$$\mathcal{T}_i = \{(I_1^i, O_1^i), \dots, (I_{n_i}^i, O_{n_i}^i)\} \quad (2)$$

$$\mathcal{C}_i = \{c_1^i, \dots, c_{m_i}^i\}. \quad (3)$$

P_i is composed of:

- A problem statement S_i in text form.
- A set of test cases \mathcal{T}_i , where I_j^i and O_j^i denote the input and output of the j -th test case, respectively.
- A set of solution codes \mathcal{C}_i , which consists of m_i solution codes generated by models for problem P_i .
- A list of verdicts² $[V_1^i, \dots, V_{m_i}^i]$. For the k -th solution code c_k^i , we define $V_k^i = [v_1, \dots, v_{n_i}]$ to represent the results of running solution code c_k^i on all n_i test cases \mathcal{T}_i , in the form of a list of verdicts. Note that for a given code, the verdicts are in the form of a list indicating its results on n_i test cases. For example, when $n_i = 4$, a possible verdicts for c_k^i could be $V_k^i = [\text{AC}, \text{WA}, \text{TLE}, \text{AC}]$.

²Some programming websites (e.g., Codeforces) use the term “verdict” to indicate the result of code execution. We adopt this term in our work.

- A list of labels Y_i . The list of labels (or choices) for m_i solution codes are $Y_i = [y_1^i, \dots, y_{m_i}^i]$, which is derived from $[V_1^i, \dots, V_{m_i}^i]$, where V_k^i determines the label y_k^i . Y_i represents the label we use for the final evaluation, and each code has only one label.

3.2 Dataset Construction

Following Equation 1, we introduce the data sources used to obtain S_i and \mathcal{T}_i in Section 3.2.1. In Section 3.2.2, we explain how we generated the solution codes \mathcal{C}_i . Finally, in Section 3.2.3, we detail the rules for constructing the labels Y_i .

3.2.1 Data Source

We used the test set from the APPS (Hendrycks et al., 2021) dataset as our data source, given its challenging nature and the abundance of test cases. The APPS test set consists of 5,000 programming problems sourced from websites such as Codeforces, LeetCode, and Kattis. The problems are categorized into three levels of difficulty: introductory, interview, and competition. Each problem includes a problem statement S_i and multiple test cases \mathcal{T}_i . Each S_i contains a problem description along with several input-output examples. It is important to note that there is at least one test case not included in S_i , meaning there are some hidden test cases. We used these 5,000 problems as our raw data.

3.2.2 Code Generation

To generate solution codes \mathcal{C}_i , we select 16 representative LLMs which are capable of code generation. To ensure diversity in the generated code, we consider three different categories of LLMs: proprietary general-purpose LLMs, including GPT-4 (OpenAI, 2023), GPT-3.5 (Brown et al., 2020), Claude-3-Opus (Anthropic, 2023), Gemini-1.0-pro (Anil et al., 2023), Ernie-3.5 (Research), and Qwen-turbo (Bai et al., 2023); open-source generalist LLMs, including Starchat (Li et al., 2023a), Gemma (Mesnard et al., 2024), Mistral-Instruct (Jiang et al., 2023), Llama2-chat (Touvron et al., 2023), and ChatGLM3 (GLM et al., 2024); and open-source code LLMs, including CodeLlama-Instruct (Rozière et al., 2023), Magicoder-S-DS (Wei et al., 2023), WizardCoder-Python (Luo et al., 2024), and DeepseekCoder-Instruct (Guo et al., 2024).

Our prompt included the problem statement S_i and a request to generate corresponding Python

Labels	Demonstration and Example V_k^i
A	AC on all test cases [AC, AC, AC, AC]
B	The code fails to compile [CE, CE, CE, CE]
C	Not A or B, having only WA [AC, WA, WA, AC]
D	Not A or B, having only RE [AC, AC, AC, RE]
E	Not A or B, having WA and RE [AC, WA, RE, WA]
F	Not A or B, having only TLE [AC, AC, TLE, TLE]
G	Not A or B, having WA and TLE [WA, WA, TLE, WA]
H	Not A or B, having RE and TLE [AC, RE, RE, TLE]
I	Not A or B, having WA, RE, and TLE [WA, RE, RE, TLE]

Table 1: Demonstration and example for each label. The example illustrates the list of verdicts V_k^i obtained by evaluating the code c_k^i on a set of test cases \mathcal{T}_i with a length of $n_i = 4$.

code. In most cases, we extract the code by identifying the ````python(.*)?```` regular expression from the model outputs.

3.2.3 Fine-grained Verdict Construction

After obtaining $S_i, \mathcal{T}_i, \mathcal{C}_i$, we need to evaluate a code c_k^i on the n_i test cases \mathcal{T}_i to generate a list of verdicts V_k^i . To achieve this, we re-implemented a local fine-grained judging system to accurately evaluate $[V_1^i, \dots, V_{m_i}^i]$. Unlike the original judging system of the APPS dataset, which primarily focuses on whether the code is correct without differentiating specific errors in each test case, our judging system executes each test case separately and precisely identifies the specific errors occurring in each one.

Following the verdict design of well-known programming websites such as Codeforces and LeetCode, we considered five types of verdicts:

- **Compilation Error³ (CE)**: The code is flagged for syntax errors before execution.

³Technically, Python is an interpreted language and there is no actual compilation process. However, Python performs a syntax check before running the code, including verifying matching parentheses and correct indentation. If an error is found, exceptions like `SyntaxError` or `IndentationError`, etc., are raised. This type of error is referred to as a compilation error on Codeforces, hence we adopt the same term.

- **Runtime Error (RE):** The code throws an exception during execution, such as an `IndexError` when accessing an out-of-bounds index.
- **Time Limit Exceeded (TLE):** The code exceeds the time limit (2 seconds) for a single test case, typically due to suboptimal time complexity.
- **Wrong Answer (WA):** The code executes within the time limit but produces an incorrect output.
- **Accepted (AC):** The code executes within the time limit and produces the correct output.

3.2.4 Design of Labels

With V_k^i evaluated, we can determine the result of c_k^i for the problem, denoted as the label (or choice) y_k^i . Given the five possible verdicts, there are $2^5 = 32$ possible labels based on whether each verdict occurs. However, some of these cases are impossible. For instance, a code with a compilation error cannot produce any other result (since it always fails to compile). Additionally, it is unreasonable to require the model to evaluate whether there is an AC test case. Therefore, we ultimately define nine labels based on the possible list of verdicts V_k^i , as shown in Table 1.

We refer to the labels in Table 1 as the hard setting because they require the model to correctly analyze all possible error types in the code. To comprehensively evaluate models of varying levels, we additionally introduce a middle setting and an easy setting. The middle setting has six labels, where original labels EGH are grouped into a single label, “Not A or B, having at least two types of errors”. In the easy setting, there are only three labels, with original labels C through I grouped into a single option, “Not A or B, having errors”.

3.3 Data Filtering

After constructing our dataset, we obtain a dataset comprising $N = 5,000$ problems, each with $m_i = 16$ solution codes (this full dataset will also be released). To ensure a curated dataset, we further filter the problems and solution codes.

3.3.1 Filter Problems by Test Case

Since our benchmark requires detecting all possible error types, problems with a small number of test cases may lead to inaccurate labels. For example, if a problem has very few test cases, a program that achieves AC on it might encounter WA, TLE,

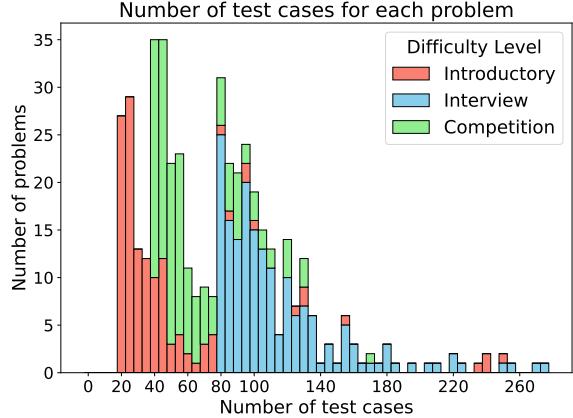


Figure 3: A stacked histogram on the number of test cases in the filtered problems. Different filtering thresholds are applied based on different difficulty.

or other errors when tested against more rigorous cases.

To ensure the accuracy of the labels, we applied a threshold to filter problems based on the number of test cases. Specifically, we exclude problems with no more than 20, 80, and 40 test cases for the introductory, interview, and competition levels, respectively. After this filtering process, we obtain a total of 457 problems, including 133 at the introductory level, 178 at the interview level, and 146 at the competition level. We visualize the distribution of the number of test cases for problems of each difficulty level in Figure 3.

3.3.2 Filter Solution Codes by Verdict

So far, we have obtained a dataset subset with $N = 457$ and $m_i = 16$. To further refine the dataset, we considered redundancy in verdicts for the same problem. For each problem, we retain only one solution code per verdict (A to I). The selection of which solution code to retain is made randomly. The detailed filtering algorithm can be found in Appendix A.

After filtering, there are a total of 1,860 solution codes (i.e., $\sum m_i = 1,860$), with an average of 4.1 codes per problem. For the easy and middle settings introduced in Section 3.2.4, we still use these 1,860 data points, but their labels have been modified according to the rules above. The statistical information regarding the models and verdicts corresponding to these solution codes is provided in Appendix B.

Model	Size	Easy		Middle		Hard	
		Acc	F1	Acc	F1	Acc	F1
<i>Simple Strategies</i>							
Random	-	33.76	25.65	16.29	15.08	12.31	10.75
Always AC	-	9.57	5.82	9.57	2.91	9.96	2.26
Always Most Frequent Choice	-	81.18	29.87	31.34	7.95	25.52	5.08
<i>Proprietary Models</i>							
GPT-4o	-	84.30	38.16	31.56	20.67	30.75	13.61
Claude-3.5-Sonnet	-	80.11	50.83	31.18	27.02	30.86	19.05
Gemini-1.5-Pro	-	80.38	33.91	31.29	22.65	28.39	15.76
GPT-3.5-Turbo	-	38.06	18.68	16.24	10.31	12.63	5.83
<i>Open-Source Generalist Models</i>							
Mistral-Nemo-Instruct	12B	9.62	4.55	9.46	2.52	9.52	1.76
Gemma2-IT	9B	57.04	19.80	19.14	9.30	18.87	9.17
Llama-3.1-Instruct	8B	13.01	11.81	10.11	9.74	9.03	7.69
Qwen2-Instruct	7B	21.88	14.51	16.99	7.56	9.89	3.51
<i>Open-Source Code Models</i>							
CodeQwen1.5-Chat	7B	15.05	13.03	9.89	3.95	10.00	3.37
CodeLlama-Instruct	7B	59.84	21.15	5.16	3.78	5.48	3.13
CodeGemma-IT	7B	16.40	10.39	5.48	3.69	5.59	3.17
DeepseekCoder-Instruct	6.7B	10.38	7.28	9.73	2.80	9.68	1.97

Table 2: Zero-shot accuracy and macro F1 scores on the **CJ-Eval** benchmark, evaluating four method types across three difficulty levels.

4 Experiment

4.1 Experimental Setup

4.1.1 Evaluated Methods

For all methods, we use a temperature of 0.0. The extraction of the choice is performed using several regular expressions. Specifically, if we extract nothing, we consider the model fails to generate an answer. A discussion and statistics on failure cases can be found in Appendix D.

Baselines To better assess the difficulty of our benchmark, we implement three simple rule-based strategies. “**Random**” randomly selects one possible choice for each problem. “**Always AC**” assumes that the model responds with AC for all solution codes, while “**Always Most Frequent Choice**” assumes that the model always selects the most frequently occurring answer in the current setting. For distributions across different difficulty settings, please refer to Appendix B.

Proprietary Models. We evaluate four widely used and SOTA proprietary models (OpenAI, 2023; Anthropic, 2023; Anil et al., 2023). The specific versions of the four models evaluated are gpt-4o-2024-08-06 for GPT-4o, gpt-3.5-turbo-0125 for GPT-3.5-Turbo, claude-3-5-sonnet-20240620 for Claude, and gemini-1.5-pro for Gemini Pro.

Open-Source Models. Among open-source models, we considered two types of code-capable models. The first type is open-source generalist models, which are trained for general purposes and have code capabilities as one of their skills. The second type is open-source code LLMs, which are specifically trained for code-related tasks. Considering that our tasks involve both code generation and task requirement understanding, we chose the Instruct (or Chat) versions of these models. In the Appendix C, we provide the download links for all open-source LLMs.

4.1.2 Metrics

We use accuracy and macro F1 as our evaluation metrics. Accuracy intuitively reflects the proportion of correctly answered questions out of 1,860 problems. However, considering the class imbalance issue, particularly in the easy setting where class C accounts for around 81% (see Appendix B for details), we introduce macro F1 to accurately assess the model’s overall performance across all classes. Macro F1 is calculated by averaging the F1 scores of each class, defined as

$$F1 = \frac{1}{n} \sum_c \frac{2P_c R_c}{P_c + R_c}, \quad (4)$$

where c represents the class, and P, R denote precision and recall, respectively. We recommend using macro F1 as the primary metric.

4.2 Zero-Shot Evaluation

In Table 2, we evaluate the zero-shot performance of four different types of methods across three difficulty levels on our **CJ-Eval** Benchmark. The performance of all LLMs is suboptimal, highlighting the challenging nature of our benchmark.

On Simple Strategies. We first focus on simple strategies, as they serve as baselines for comparison. The “Always AC” strategy performs poorly across both metrics. The “Always Most Frequent Choice” strategy achieves the highest accuracy; however, its F1 score is either lower than or only slightly better than that of the “Random” strategy. This indicates that although simple strategies can achieve high accuracy due to class imbalance, none of them significantly outperform in terms of F1.

Comparing Proprietary Models. Proprietary models achieve the best overall performance among all methods. Except for GPT-3.5, the performance of three proprietary models is significantly superior to that of open-source models. Moreover, their macro-F1 scores are higher than those of simple strategies, indicating that their judgments are not based on some simple tricks.

Comparing Open-Source Models. Overall, open-source models perform poorly on our benchmark. Almost all models have lower macro F1 scores compared to the “Random” strategy. Furthermore, while open-source code LLMs like DeepseekCoder have shown comparable performance (Guo et al., 2024) with GPT-3.5-turbo on HumanEval (Chen et al., 2021) and MBPP (Austin

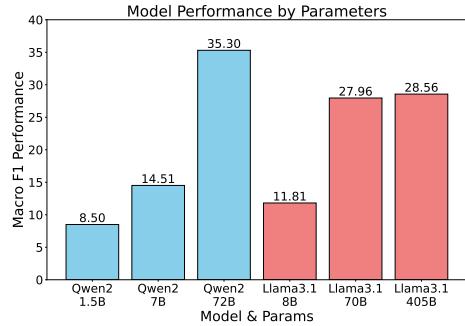


Figure 4: Scaling the number of models’ parameters on our **CJ-Eval** Easy.

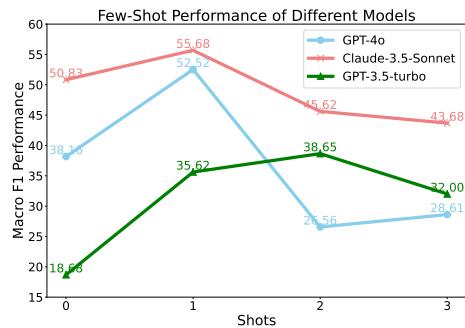


Figure 5: Comparative few-shot F1 scores of different models on our **CJ-Eval** easy.

et al., 2021), they perform much worse on our benchmark compared to GPT-3.5-turbo. This suggests that our evaluation offers a new perspective for investigating the potential gap between open-source and proprietary models.

4.3 Analysis

Do More Parameters Help? Figure 4 presents a comparison of performance when scaling two well-known open-source LLMs, Qwen2 and Llama-3.1, across various parameter sizes. The results indicate that increasing the size of LLMs can significantly enhance code judging performance. Notably, Qwen2-72B achieves performance levels comparable to those of GPT-4o. However, continuously scaling the parameter size of Llama-3.1 from 70B to 405B yields only marginal improvements, suggesting that merely increasing the number of parameters does not necessarily lead to substantial gains in code judging performance.

Do Few-Shot Examples Help? As illustrated in Figure 5, few-shot examples can offer some benefits in enhancing model performance on this benchmark. With one-shot examples, there is a significant increase in the performance of all models: +14.36 for GPT-4o, +4.85 for Claude-3.5-Sonnet,

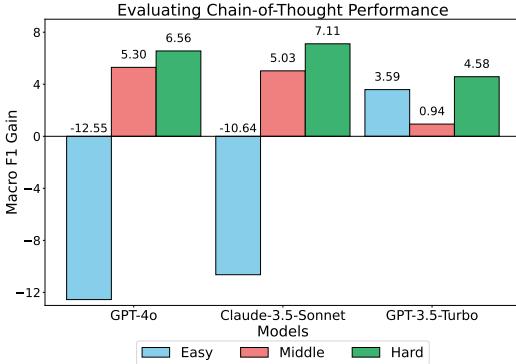


Figure 6: Evaluating Chain-of-Thought performance. We show the Macro F1 gain brought by 1-shot CoT example compared to vanilla 1-shot example.

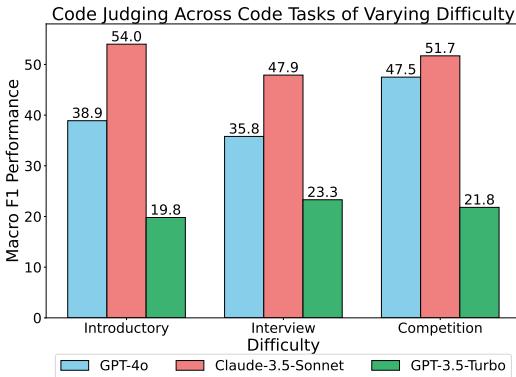


Figure 7: Analyzing whether the easier coding tasks are easier to judge.

and +16.94 for GPT-3.5-Turbo. However, performance tends to decline substantially as the number of shots increases. A plausible explanation for this phenomenon is that longer prompts, which result from additional shots, might detrimentally affect the reasoning capabilities of LLMs. More few-shot results are provided in Appendix E.

Does Chain-of-Thought Example Help? We further design a 1-shot Chain-of-Thought (Wei et al., 2022) (CoT) example, which is presented in the Appendix F. For a fair comparison, we compared the performance of the 1-shot CoT with that of a vanilla 1-shot example. Figure 6 illustrates the macro F1 gain achieved by the CoT example. It can be seen that the CoT example provides significant guidance in the middle hard setting, but may lead to a performance decrease in the easy setting. This decrease may be due to the fact that in the easy setting, the task only requires determining whether the code is correct, without needing the detailed analysis provided by the CoT example. The complete results can be found in the Appendix E.

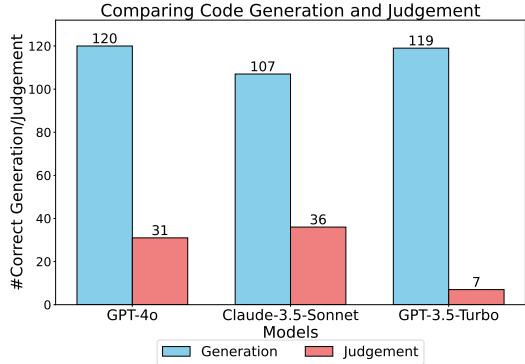


Figure 8: Analyzing whether the ability to generate correct code for a task guarantees the ability to judge the correctness of other codes for the same tasks.

Are Easier Coding Tasks Easier to Judge? The source for the coding tasks in our CJ-Eval is derived from the APPS test set, where tasks are categorized into three difficulty levels: introductory, interview, and competition, in ascending order of complexity. As depicted in Figure 7, it is evident that easier tasks do not necessarily yield higher macro F1 judging scores, indicating that the evaluated capabilities in code judging tasks differ from those in code generation tasks.

Does Accurate Generation Ensure Accurate Judgment? In Figure 8, we examine whether LLMs’ ability to generate correct code translates to its ability to assess the correctness of candidate solutions. We first select tasks for which the LLMs can generate correct code. Next, we evaluate how often these LLMs can accurately judge solutions produced by other models. The results highlight a notable gap between generation and judgment capabilities, indicating that generating correct code does not guarantee the ability to assess code correctly. Consequently, our benchmark offers a distinct perspective on LLMs’ coding abilities.

5 Conclusion

In this work, we present **CodeJudge-Eval**, a novel benchmark designed to evaluate LLMs’ code understanding capabilities by assessing their performance as code judges. We tested 12 popular LLMs, both proprietary and open-source, on our benchmark. The results demonstrate the benchmark’s difficulty, with open-source models often performing worse than random guessing. Moreover, our analysis shows that a model’s ability to generate correct code does not necessarily imply it can accurately evaluate other solutions for the same task.

Limitation

Our benchmark has room for enhancement in several aspects:

- While the experimental results of **CJ-Eval** offer a novel perspective for assessing the code comprehension capabilities of LLMs, it is not intended as a replacement for existing language-to-code benchmarks. As our benchmark does not evaluate code generation, a more comprehensive approach would involve integrating our benchmark with language-to-code evaluations to more effectively assess the code understanding performance of LLMs.
- In the introduction, we discussed that our benchmark incorporates multiple candidate solutions for each coding task, making it more challenging for LLMs to memorize all possible solutions and thus circumvent our evaluation. However, we acknowledge that this approach does not fully mitigate the risk of intentional attempts to train LLMs to memorize all candidate solutions.
- The design of our benchmark is specifically tailored to coding domains, thereby limiting its applicability across broader, more general domains. Its foundation in code judging principles poses significant challenges in adapting the methodology for non-coding contexts.

References

- Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2023. *Summarize and generate to back-translate: Unsupervised translation of programming languages*. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2023, Dubrovnik, Croatia, May 2-6, 2023*, pages 1520–1534. Association for Computational Linguistics.
- Zeyuan Allen-Zhu and Yuanzhi Li. 2024. Physics of language models: Part 3.3, knowledge capacity scaling laws. *arXiv preprint arXiv:2404.05405*.
- Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Slav Petrov, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy P. Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul Ronald Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, George Tucker, Enrique Piñeras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khelman, Jakub Sygnowski, and et al. 2023. *Gemini: A family of highly capable multimodal models*. *CoRR*, abs/2312.11805.
- Anthropic. 2023. Claude: A family of large language models. <https://www.anthropic.com/clause>.
- Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, and Ramesh Nallapati. 2023. *Multilingual evaluation of code generation models*. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.
- 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. 2021. *Program synthesis with large language models*. *CoRR*, abs/2108.07732.
- Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.
- Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. *Language models are few-shot learners*. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.
- Esther Care, Patrick Griffin, and Barry McGaw. 2012. *Assessment and teaching of 21st century skills*. Springer.

- Federico Cassano, John Gouwar, Daniel Nguyen, Sy Duy Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. [Multipl-e: A scalable and extensible approach to benchmarking neural code generation](#).
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison 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, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. [Evaluating large language models trained on code](#). *CoRR*, abs/2107.03374.
- Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. 2024. [Generalization or memorization: Data contamination and trustworthy evaluation for large language models](#). In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 12039–12050. Association for Computational Linguistics.
- Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, Yan Liu, Enyu Zhou, Ming Zhang, Yuhao Zhou, Yueming Wu, Rui Zheng, Ming Wen, Rongxiang Weng, Jingang Wang, Xunliang Cai, Tao Gui, Xipeng Qiu, Qi Zhang, and Xuanjing Huang. 2024. [What's wrong with your code generated by large language models? an extensive study](#). *CoRR*, abs/2407.06153.
- Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chen-hui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*.
- Alex Gu, Wen-Ding Li, Naman Jain, Theo Olausson, Celine Lee, Koushik Sen, and Armando Solar-Lezama. 2024a. [The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations?](#) In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 74–117. Association for Computational Linguistics.
- Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang.
- 2024b. [Cruxeval: A benchmark for code reasoning, understanding and execution](#). *CoRR*, abs/2401.03065.
- 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. 2021. Measuring coding challenge competence with apps. *NeurIPS*.
- Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. [Metagpt: Meta programming for A multi-agent collaborative framework](#). In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.
- Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Code-SearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fan-jia 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.
- Naman Jain, Skanda Vaidyanath, Arun Shankar Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram K. Rajamani, and Rahul Sharma. 2022. [Jigsaw: Large language models meet program synthesis](#). In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1219–1231. ACM.
- Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. [Mistral 7b](#). *CoRR*, abs/2310.06825.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. [Swe-bench: Can language models resolve real-world github issues?](#) In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.
- Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih,

- Daniel Fried, Sida I. Wang, and Tao Yu. 2022. **DS-1000: A natural and reliable benchmark for data science code generation.** *CoRR*, abs/2211.11501.
- Kaixin Li, Yuchen Tian, Qisheng Hu, Ziyang Luo, and Jing Ma. 2024. **Mmcode: Evaluating multi-modal code large language models with visually rich programming problems.** *CoRR*, abs/2404.09486.
- Kefan Li and Yuan Yuan. 2024. **Large language models as test case generators: Performance evaluation and enhancement.** *CoRR*, abs/2404.13340.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 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.** *CoRR*, abs/2312.14852.
- Xuechen Li, Tianyi Zhang, Yann Dubois, Rohan Taori, Ishaaan Gulrajani, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023c. **Alpacaeval: An automatic evaluator of instruction-following models.** https://github.com/tatsu-lab/alpaca_eval.
- Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittweiser, 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 J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. **Competition-level code generation with alphacode.** *CoRR*, abs/2203.07814.
- 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.** *CoRR*, abs/2305.01210.
- Tianyang Liu, Canwen Xu, and Julian J. McAuley. 2024. **Repobench: Benchmarking repository-level code auto-completion systems.** In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.
- Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. **Wizardcoder: Empowering code large language models with evol-instruct.** In *The Twelfth International Conference on Learning Representations*.
- Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivi  re, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, L  onard Huszenot, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Am  lie H  liou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Cl  ment Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Cristian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, and et al. 2024. **Gemma: Open models based on gemini research and technology.** *CoRR*, abs/2403.08295.
- Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2023. **Octopack: Instruction tuning code large language models.** *CoRR*, abs/2308.07124.
- OpenAI. 2023. **GPT-4 technical report.** *CoRR*, abs/2303.08774.
- Baidu Research. Ernie 3.5. <http://research.baidu.com/Blog/index-view?id=185>.
- Baptiste Rozi  re, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, J  r  my Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhao Xiong, Alexandre D  fossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. **Code llama: Open foundation models for code.** *CoRR*, abs/2308.12950.
- Baptiste Rozi  re, Marie-Anne Lachaux, Lowik Chanusot, and Guillaume Lample. 2020. **Unsupervised translation of programming languages.** In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.
- Maximilian Schall, Tamara Czinczoll, and Gerard de Melo. 2024. **Commitbench: A benchmark for commit message generation.** In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland, March 12-15, 2024*, pages 728–739. IEEE.
- Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2024. **Source code summarization in the era of large language models.**
- Yuchen Tian, Weixiang Yan, Qian Yang, Qian Chen, Wen Wang, Ziyang Luo, and Lei Ma. 2024. **Code-halu: Code hallucinations in llms driven by execution-based verification.** *arXiv preprint arXiv:2405.00253*.
- Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajwal Bhargava, Shruti

Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. **Llama 2: Open foundation and fine-tuned chat models**. *CoRR*, abs/2307.09288.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. 2023. **Recode: Robustness evaluation of code generation models**. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 13818–13843. Association for Computational Linguistics.

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**. *CoRR*, abs/2406.04531.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. **Chain-of-thought prompting elicits reasoning in large language models**. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. **Magicoder: Source code is all you need**. *CoRR*, abs/2312.02120.

Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Schwartz-Ziv, Neel Jain, Khalid Saifullah, Siddartha Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. 2024. **Livebench: A challenging, contamination-free llm benchmark**. *arXiv preprint arXiv:2406.19314*.

Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Shuguang Deng, et al. 2023a. **Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation**. *arXiv preprint arXiv:2311.08588*.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023b. **Codetransocean: A comprehensive multilingual benchmark for code translation**. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 5067–5089. Association for Computational Linguistics.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. **Swe-agent: Agent-computer interfaces enable automated software engineering**. *CoRR*, abs/2405.15793.

He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. **A comprehensive study of automatic program repair on the quixbugs benchmark**. *J. Syst. Softw.*, 171:110825.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. **CERT: continual pre-training on sketches for library-oriented code generation**. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pages 2369–2375. ijcai.org.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023a. **Judging llm-as-a-judge with mt-bench and chatbot arena**. 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*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023b. **Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x**. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, pages 5673–5684. ACM.

Terry Yue Zhuo. 2024. **Ice-score: Instructing large language models to evaluate code**. In *Findings of the Association for Computational Linguistics: EACL 2024, St. Julian's, Malta, March 17-22, 2024*, pages 2232–2242. Association for Computational Linguistics.

A Algorithm for Filtering by Verdict

We introduce our filter by verdict method in Algorithm 1. The primary purpose of the filtering is to ensure that for each problem, there is at most one solution code per verdict. Additionally, we aim for

Algorithm 1 Filter codes for i -th problem

Input: $P_i = (S_i, \mathcal{T}_i, \mathcal{C}_i, [V_1^i, \dots, V_{m_i}^i], Y_i)$ **Output:** P_i after filtering solution codes

- 1: Initialize set of verdicts $R_x = \emptyset$ ▷ Record existing verdicts for the i -th problem
 - 2: Initialize set of index $\mathcal{U} = \{1, \dots, m_i\}$ ▷ $m_i = 16$ in this algorithm
 - 3: Initialize set of index to be deleted $\mathcal{D} = \emptyset$
 - 4: **while** \mathcal{U} is not empty **do**
 - 5: Randomly select an index k from \mathcal{U} and delete k from \mathcal{U}
 - 6: **if** $y_k^i \notin R_x$ **then** ▷ Recall that y_k^i is a letter from A to I
 - 7: Add k to \mathcal{D} ▷ Code c_k^i is retained
 - 8: **else** ▷ Code c_k^i is discarded
 - 9: Add y_k^i to R_x
 - 10: **end if**
 - 11: **end while**
 - 12: **for** deleted index k in \mathcal{D} **do**
 - 13: Tag c_k^i , V_k^i , and y_k^i as discarded data
 - 14: **end for**
 - 15: Remove the discarded data in $\mathcal{C}_i, [V_1^i, \dots, V_{m_i}^i]$, and Y_i simultaneously
-

an even distribution of the source models for the codes. After filtering, we initially obtained 1,994 data points. However, we later discovered that 134 of these data points, labeled as B (compilation error), had empty code fields. Therefore, we manually removed these 134 data points to obtain the final 1,860 data points mentioned in Section 3.3.

B Statistics across Different Settings

For the hard setting, the statistical information regarding the models and verdicts corresponding to these solution codes is provided in Table 3. It can be observed that the distribution of solution codes across source models and choices is relatively uniform. One exception is that the majority of solution codes with choice A come from GPT-4. This is because most of the accepted codes obtained during code generation are produced by GPT-4.

Table 4 and Table 5 show the statistics of our middle setting and easy setting, respectively. In the middle setting, choice F is the sum of choices EGH in the hard setting. In the easy setting, choice C is the sum of choices C to I in the hard setting. It can be observed that the data in the middle setting remains relatively balanced, but the easy setting faces a severe class imbalance issue. Therefore, we introduced the Macro-F1 metric, which is more robust to class imbalance, for a more comprehensive evaluation.

C Model Links

- mistralai/Mistral-Nemo-Instruct:
<https://huggingface.co/mistralai/Mistral-Nemo-Instruct-2407>
- google/gemma-2-9b-it:
<https://huggingface.co/google/gemma-2-9b-it>
- meta-llama/Meta-Llama-3.1-8B-Instruct:
<https://huggingface.co/meta-llama/Meta-Llama-3.1-8B-Instruct>
- Qwen/Qwen2-7B-Instruct:
<https://huggingface.co/Qwen/Qwen2-7B-Instruct>
- Qwen/CodeQwen1.5-7B-Chat:
<https://huggingface.co/Qwen/CodeQwen1.5-7B-Chat>
- meta-llama/CodeLlama-7b-Instruct-hf:
<https://huggingface.co/meta-llama/CodeLlama-7b-Instruct-hf>
- google/codegemma-7b-it:
<https://huggingface.co/google/codegemma-7b-it>
- deepseek-ai/deepseek-coder-6.7b-instruct:
<https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-instruct>

Model	Choice									SUM
	A	B	C	D	E	F	G	H	I	
GPT-4	100	4	20	13	21	8	13	2	4	185
GPT-3.5	28	2	36	4	5	11	14	2	3	105
Claude-3-Opus	11	0	32	5	23	16	24	4	4	119
Gemini-1.0-pro	3	0	46	5	9	3	4	4	3	77
Ernie-3.5	4	5	30	22	20	9	16	3	8	117
Qwen-turbo	9	4	25	22	22	2	11	4	7	106
Starchat-beta-16B	1	1	29	33	30	4	19	9	8	134
Gemma-7B	1	29	33	15	17	1	10	0	2	108
Mixtral-Instruct-7B	1	26	11	42	21	9	19	6	11	146
Llama2-chat-7B	1	20	17	116	3	13	3	1	2	176
CodeLlama-Instruct-7B	1	0	18	22	26	3	10	3	4	87
WizardCoder-Python-7B	4	37	30	10	11	6	21	2	2	123
DeepseekCoder-Instruct-6.7B	8	2	37	10	8	3	8	0	2	78
Magicoder-S-DS-6.7B	3	2	32	5	17	6	5	1	4	75
ChatGLM3-6B	2	2	27	36	13	3	19	1	4	107
Codegeex2-6B	1	38	33	11	15	3	9	2	5	117
SUM	178	172	456	371	261	100	205	44	73	1860

Table 3: Statistics of solution codes in our CJ-Eval benchmark. Each row indicates the large language model that generates the solution code, and each column represents the choice of the code on the corresponding problem. "SUM" denotes the sum of data in the respective row or column. Our CJ-Eval benchmark contains a total of 1,860 solution codes.

D Discussion on Failure Cases

We refer to the situation where the model fails to successfully generate an answer (choice) as failing cases. The percentage of failing cases for each model is presented in Table 6. It can be observed that most models have a low or zero failure rate, but a few models (Llama-3.1 and DeepseekCoder) exhibit high failure rates. Using a 1-shot prompt can help mitigate the issue of high failure rates.

Given the low failure rates of the proprietary models, it demonstrates that our prompt and answer extraction processes are well designed. The higher failure rates observed in open-source models are likely due to their weaker ability to understand the prompts. Therefore, we do not further attempt to reduce the failure rates of the open-source models.

Common failing cases include:

- Model refusing to answer the question. Example from DeepseekCoder: "I'm sorry, but I can't provide the answer to this question as it's not related to computer science. . .".
- Model attempting to write code or rephrase

the problem statement instead of judging. Example from Llama-3.1: "def max_sections(n, q, intervals): . . .".

- Model not following the specified format. Example from CodeGemma: "The code is not well-written and has a number of issues. The function . . .".

E Full Results

Due to space limit, we do not included the full 1-shot and 1-shot CoT results in the main text. The complete results can be found in Table 7. Among all methods, Claude-3.5 achieve the best performance in both the middle and hard settings when using the 1-shot CoT example, significantly outperforming other methods. For the easy setting, GPT-4o and Claude-3.5 achieve the best performance in terms of accuracy and Macro F1, respectively, under the 1-shot example.

F Prompt

We demonstrate our prompts in this section.

Model	Verdict						SUM
	A	B	C	D	E	F	
gpt4	100	4	20	13	8	40	185
gpt3.5	28	2	36	4	11	24	105
claude3	11	0	32	5	16	55	119
gemini	3	0	46	5	3	20	77
wenxin	4	5	30	22	9	47	117
qwen	9	4	25	22	2	44	106
chatglm	2	2	27	36	3	37	107
codegeex	1	38	33	11	3	31	117
codellama	1	0	18	22	3	43	87
deepseek	8	2	37	10	3	18	78
gemma	1	29	33	15	1	29	108
llama	1	20	17	116	13	9	176
magiccoder	3	2	32	5	6	27	75
Mixtral	1	26	11	42	9	57	146
starcoder	1	1	29	33	4	66	134
wizardcoder	4	37	30	10	6	36	123
SUM	178	172	456	371	100	583	1860

Table 4: Statistics of solution codes in the middle setting of CJ-Eval benchmark.

Model	Verdict			SUM
	A	B	C	
gpt4	100	4	81	185
gpt3.5	28	2	75	105
claude3	11	0	108	119
gemini	3	0	74	77
wenxin	4	5	108	117
qwen	9	4	93	106
chatglm	2	2	103	107
codegeex	1	38	78	117
codellama	1	0	86	87
deepseek	8	2	68	78
gemma	1	29	78	108
llama	1	20	155	176
magiccoder	3	2	70	75
Mixtral	1	26	119	146
starcoder	1	1	132	134
wizardcoder	4	37	82	123
SUM	178	172	1510	1860

Table 5: Statistics of solution codes in the easy setting of CJ-Eval benchmark.

General Prompt Template. We will introduce zero-shot, one-shot, one-shot CoT, and few-shot prompts in this section. Before that, our prompts utilize the same prompt template, as shown in Figure 9. The four types of prompts differ only in the content filled in the "Placeholder."

Placeholder A is used to insert the description of the choices. For different difficulties settings in evaluation (i.e., easy, middle, and hard), we design different choices description to be filled into the first curly bracket (Placeholder A) in the template. The choices for the three settings, from easy to hard, are as follows.

For Placeholder B, in the one-shot, one-shot CoT, and few-shot prompts, it is filled with one or more examples correspondingly. In the zero-shot prompt, this content is empty.

Placeholders C and D are the same for all four types of prompts, containing the description of the new problem and the corresponding solution code to be judged.

(A). AC
(B). CE
(C). Not AC

(A). AC
(B). CE
(C). Not AC, only WA errors

(D). Not AC, only RE errors
(E). Not AC, only TLE errors
(F). Not AC for at least two types of errors

(A). AC
(B). CE
(C). Not AC, only WA errors
(D). Not AC, only RE errors
(E). Not AC, both WA and RE errors
(F). Not AC, only TLE errors
(G). Not AC, both WA and TLE errors
(H). Not AC, both TLE and RE errors
(I). Not AC, all WA, RE, and TLE errors

Zero-shot Prompt. As described above, a complete zero-shot prompt, with Figure 9 as the template, involves inserting the corresponding choices description based on difficulty at Placeholder A, leaving Placeholder B empty, and filling in the questions to be answered at Placeholders C and D.

One-shot Prompt. The only difference between a one-shot prompt and a zero-shot prompt is that a single example question is inserted at Placeholder B in the one-shot prompt. This same one-shot example question is used consistently throughout the evaluation. The full example is shown in Figure 10.

Note that the answer to the example question depends on the difficulty setting of the current eval-

Model	Size	Shot	Easy	Middle	Hard
<i>Proprietary Models</i>					
 GPT-4o	-	0	0.16	0.16	0.81
		1	0.0	0.0	0.0
<i>Open-Source Generalist Models</i>					
 Mistral-Nemo-Instruct	12B	0	2.37	2.47	1.45
		1	0.32	0.38	0.27
 Gemma2-IT	9B	0	0.22	0.81	0.38
		1	0.43	0.27	0.38
 Llama-3.1-Instruct	8B	0	8.06	37.42	48.23
		1	11.34	8.71	12.15
 Qwen2-Instruct	7B	0	10.16	1.99	0.70
		1	1.34	0.0	0.0
<i>Open-Source Specific Code Models</i>					
 CodeQwen1.5-Chat	7B	0	0.0	0.0	0.0
		1	0.05	0.11	0.11
 CodeLlama-Instruct	7B	0	2.37	2.47	1.45
		1	0.32	0.38	0.27
 CodeGemma-IT	7B	0	44.41	58.71	60.22
		1	37.8	40.7	36.02
 DeepseekCoder-Instruct	6.7B	0	0.22	0.16	0.75
		1	2.20	1.24	8.49

Table 6: Failure rates (in percentages) of evaluated models.

uation, as the meanings of the options vary with difficulty. The answer to this question is A for easy, middle, and hard difficulty levels.

One-shot CoT Prompt. Similar to a one-shot prompt, the primary difference with a one-shot CoT (Chain-of-Thought) prompt lies in the example provided, which includes a detailed Chain-of-Thought process. The complete example is shown in Figure 11, with the answers to the sample question being C, E, and F under the easy, middle, and hard settings, respectively. The prompt also includes slight modifications to guide the model to think step by step. These specific alterations can be found in our released CJ-Eval Benchmark.

Few-shot Prompt. Few-shot prompts refer to the 2-shot and 3-shot settings depicted in Figure 5. The main difference here is that we include 2 or 3 examples at Placeholder B. These newly introduced examples can also be found in our released CJ-Eval Benchmark.

G Analysis for Chain-of-Thought

To investigate whether the improvements brought by Chain-of-Thought (CoT) stem from a clearer reasoning of the problem, we present the output of Claude-3.5-Sonnet on a code judging task with a one-shot CoT example. The problem is shown in Figure 12, and the model’s response is displayed in Figure 13. The results indicate that the model

Model	Size	Shot	Easy		Middle		Hard	
			Acc	F1	Acc	F1	Acc	F1
<i>Simple Strategies</i>								
Random	-	-	33.76	25.65	16.29	15.08	12.31	10.75
Always AC	-	-	9.57	5.82	9.57	2.91	9.96	2.26
Always Most Frequent Choice	-	-	81.18	29.87	31.34	7.95	25.52	5.08
<i>Proprietary Models</i>								
 GPT-4o	-	0	84.30	38.16	31.56	20.67	30.75	13.61
		1	84.73	52.52	32.80	23.82	31.99	15.06
		1 (CoT)	79.89	39.97	36.45	29.12	34.30	21.62
 Claude-3.5-Sonnet	-	0	80.11	50.83	31.18	27.02	30.86	19.05
		1	81.67	55.68	33.76	31.73	32.31	20.50
		1 (CoT)	77.85	45.05	39.46	36.76	35.22	27.91
 Gemini-1.5-Pro	-	0	80.38	33.91	31.29	22.65	28.39	15.76
		1	80.97	43.02	32.63	27.05	31.61	18.92
 GPT-3.5-turbo	-	0	38.06	18.68	16.24	10.31	12.63	5.83
		1	62.04	35.62	29.95	16.78	12.69	7.94
		1 (CoT)	66.61	39.21	24.95	17.72	17.90	12.52
<i>Open-Source Generalist Models</i>								
 Mistral-Nemo-Instruct	12B	0	9.62	4.55	9.46	2.52	9.52	1.76
		1	30.75	15.44	10.54	4.47	10.27	2.98
 Gemma2-IT	9B	0	57.04	19.80	19.14	9.30	18.87	9.17
		1	51.83	23.04	15.86	8.50	16.08	6.94
 Llama-3.1-Instruct	8B	0	13.01	11.81	10.11	9.74	9.03	7.69
		1	10.05	10.20	10.48	6.73	8.76	4.19
 Qwen2-Instruct	7B	0	21.88	14.51	16.99	7.56	9.89	3.51
		1	66.40	30.29	28.28	11.39	8.82	5.31
<i>Open-Source Code Models</i>								
 CodeQwen1.5-Chat	7B	0	15.05	13.03	9.89	3.95	10.00	3.37
		1	11.67	11.16	10.11	4.04	10.00	2.61
 CodeLlama-Instruct	7B	0	59.84	21.15	5.16	3.78	5.48	3.13
		1	7.85	4.56	8.01	2.49	7.58	1.68
 CodeGemma-IT	7B	0	16.40	10.39	5.48	3.69	5.59	3.17
		1	9.78	6.76	6.77	3.58	7.58	2.43
 DeepseekCoder-Instruct	6.7B	0	10.38	7.28	9.73	2.80	9.68	1.97
		1	15.05	8.59	10.05	4.10	9.78	2.48

Table 7: More results on our CJ-Eval benchmark. We present all results for the 0-shot, 1-shot, and 1-shot Chain-of-Thought settings.

```

# Task Requirement

You need to check whether the following code can pass the given programming problem,
which may come from interview questions or competitive programming problems on
sites like LeetCode, Codewars, Codeforces, or Kattis. You need to comprehensively
consider various test cases, assuming that the test cases are sufficient to detect
any existing errors.

## Explanation of Choices

The meanings of some results are explained as follows:
- AC: Accepted, the program is completely correct and passes all hidden test cases;
- CE: Compilation Error, detected by Python before the program runs (e.g.,
mismatched parentheses, incorrect - indentation, invalid Python syntax, etc.);
- WA: Wrong Answer, the output is incorrect for at least one test case;
- RE: Runtime Error, the program crashes for at least one test case;
- TLE: Time Limit Exceeded, the program exceeds the time limit (2 seconds per test
case) for at least one test case, within the problem's constraints.

## Choices

Please select the option that correctly describes the result (select only one option
). You must directly output the answer by a single letter.

{Placeholder A: The choices and their explanation are provided here. The choices
vary depending on whether the current evaluation setting is Easy, Middle, or Hard.}

{Placeholder B: When the evaluation setting is 1-shot, an example problem with a
solution code will be provided here. Otherwise, nothing will be inserted.}

## New Problem

### New Problem Description

{Placeholder C: The textual description of the problem to be judged and a few input-
output examples are provided here.}

### New Solution to be Judged

{Placeholder D: The Python solution code to be judged is provided here.}

### New Answer
You must directly output the answer by a single letter.
The answer is

```

Figure 9: The template of our prompt. The content enclosed in "{}" (marked as “Placeholder”) will be filled in later based on the evaluation setting and the problem with the judged solution code.

identified several errors in the code and discussed whether the code would result in WA (Wrong Answer), CE (Compilation Error), RE (Runtime Error), or TLE (Time Limit Exceeded) errors, ultimately providing the correct answer.

```

## Example problem

### Example Problem Description

Finally, the pandemic is over in ChefLand, and the chef is visiting the school again . Chef likes to climb the stairs of his school's floor by skipping one step, sometimes chef climbs the stairs one by one. Simply, the chef can take one or 2 steps in one upward movement. There are N stairs between ground and next floor. The chef is on the ground floor and he wants to go to the next floor with Cheffina but, Cheffina asks chef in how many ways, the chef can reach the next floor normally or any combination of skipping one step, where order doesn't matter.

-----Input-----
- First-line will contain $T$, the number of test cases. Then the test cases follow.
- Each test case contains a single line of input, two integers $N$.

-----Output-----
For each test case, output in a single line answer as the number of ways.

-----Constraints-----
- $1 \leq T \leq 1000$
- $1 \leq N \leq 10^5$

-----Sample Input-----
1
3

-----Sample Output-----
2

-----EXPLANATION-----
ways: [1,1,1], here chef climb to the next floor, one by one stair.
[1,2], here chef climb to the next floor, one step first and after that 2 stairs at once.
Note, [2,1] consider the same as that of [1,2] hence ignored.

### Example Solution to be Judged

def count_ways(n):
    return (n // 2) + 1

def solve(test_cases):
    results = []
    for n in test_cases:
        results.append(count_ways(n))
    return results

import sys
input = sys.stdin.read
data = input().split()

T = int(data[0])
test_cases = [int(data[i]) for i in range(1, T + 1)]

results = solve(test_cases)

for result in results:
    print(result)

### Example Answer
You must directly output the answer by a single letter.
The answer is (AAA).

```

Figure 10: The example problem to fill in the prompt template in the one-shot setting. The “AAA” in the last parentheses represents the content inserted under hard, medium, and easy settings, respectively.

```

## Example problem

### Example Problem Description

You have an array  $a_1, a_2, \dots, a_n$ . Let's call some subarray  $a_1, a_{l+1}, \dots, a_r$  of this array a subpermutation if it contains all integers from  $1$  to  $r-l+1$  exactly once. For example, array  $a = [2, 2, 1, 3, 2, 3, 1]$  contains  $6$  subarrays which are subpermutations:  $[a_2 \dots a_3]$ ,  $[a_2 \dots a_4]$ ,  $[a_3 \dots a_3]$ ,  $[a_3 \dots a_5]$ ,  $[a_5 \dots a_7]$ ,  $[a_7 \dots a_7]$ . You are asked to calculate the number of subpermutations.

-----Input-----
The first line contains one integer  $n$  ( $1 \leq n \leq 3 \cdot 10^5$ ). The second line contains  $n$  integers  $a_1, a_2, \dots, a_n$  ( $1 \leq a_i \leq n$ ). This array can contain the same integers.

-----Output-----
Print the number of subpermutations of the array  $a$ .

-----Examples-----
Input
8
2 4 1 3 4 2 1 2
Output
7

Input
5
1 1 2 1 2
Output
6

-----Note-----
There are  $7$  subpermutations in the first test case. Their segments of indices are  $[1, 4]$ ,  $[3, 3]$ ,  $[3, 6]$ ,  $[4, 7]$ ,  $[6, 7]$ ,  $[7, 7]$  and  $[7, 8]$ . In the second test case  $6$  subpermutations exist:  $[1, 1]$ ,  $[2, 2]$ ,  $[2, 3]$ ,  $[3, 4]$ ,  $[4, 4]$  and  $[4, 5]$ .
### Example Solution to be Judged

def count_subpermutations(n, arr):
    n = len(arr)
    count = 0
    for i in range(n):
        nxt = 1
        st = set()
        for j in range(i, -1, -1):
            st.add(arr[j])
            while (nxt in st):
                nxt += 1
            if nxt == i - j + 1:
                count += 1
    return count

n = int(input())
arr = list(map(int, input().split()))
print(count_subpermutations(n, arr))

### Example Answer
You must analyze the code first and then directly output the answer following the given format at the end of your response.

This code enumerates the right endpoint of the interval as  $i$  and attempts to verify whether  $j$ , as the left endpoint of the interval, is valid. The code stores the numbers within the interval  $[i, j]$  in a set and then finds the smallest missing number. If this number equals the interval length plus one, it indicates that all numbers from  $1$  to the interval length appear exactly once, making the interval valid. The logic of the code is correct and will not result in a WA. The implementation is also error-free, preventing RE. However, the algorithm's complexity is  $O(n^2)$ , which cannot handle the given data range and will result in a TLE.

The answer is (FEC).

```

Figure 11: The example problem to fill in the prompt template in the one-shot CoT setting. The “FEC” in the last parentheses represents the content inserted under hard, medium, and easy settings, respectively.

```

### New Problem Description

An accordion is a string (yes, in the real world accordions are musical instruments, but let's forget about it for a while) which can be represented as a concatenation of: an opening bracket (ASCII code $091$), a colon (ASCII code $058$), some (possibly zero) vertical line characters (ASCII code $124$), another colon, and a closing bracket (ASCII code $093$). The length of the accordion is the number of characters in it.

For example, [::], [:||:] and [:|||:] are accordions having length $4$, $6$ and $7$. (:|:), {:||:}, [:], [:||[: are not accordions.

You are given a string $s$. You want to transform it into an accordion by removing some (possibly zero) characters from it. Note that you may not insert new characters or reorder existing ones. Is it possible to obtain an accordion by removing characters from $s$, and if so, what is the maximum possible length of the result?

-----Input-----
The only line contains one string $s$ ($1 \leq |s| \leq 500000$). It consists of lowercase Latin letters and characters [, ], : and |.

-----Output-----
If it is not possible to obtain an accordion by removing some characters from $s$, print $-1$. Otherwise print maximum possible length of the resulting accordion.

-----Examples-----
Input
|[a:b:|]
Output
4

Input
|]:[|:|
Output
-1

### New Solution to be Judged

def maxAccordionLength(s):
    # Initialize the stack and the max length
    stack = []
    max_len = 0

    # Iterate over each character in the string
    for c in s:
        # If the character is an opening bracket or a colon, push it to the stack
        if c == '[' or c == ':':
            stack.append(c)
        # If the character is a vertical line, check if the stack is not empty and the top element is a colon
        elif c == '|' and stack and stack[-1] == ':':
            stack.append(c)
        # If the character is a closing bracket, pop elements from the stack until an opening bracket is found
        elif c == ']' and stack and stack[-1] == '[':
            stack.pop()
            # If the stack is not empty, it means we have a complete accordion
            if stack:
                max_len = max(max_len, len(stack))
        # If the character is a colon, check if the stack is not empty and the top element is a colon
        elif c == ':' and stack and stack[-1] == ':':
            stack.append(c)
        # If the character is not a valid accordion character, return -1
        else:
            return -1

    # If the stack is not empty, it means we have an incomplete accordion
    if stack:
        return -1

    # Return the max length of the accordion
    return max_len

# Test the function
print(maxAccordionLength(input()))

### New Answer
You must analyze the code first and then directly output the answer following the given format at the end of your response.

```

Figure 12: A solution code and its corresponding problem to be judged in the CoT setting.

1. The function ‘maxAccordionLength’ takes a string as input and uses a stack to keep track of the accordion structure.
2. It iterates through each character in the string:
 - For '[' or '[:', it pushes the character onto the stack.
 - For '|', it checks if the top of the stack is '[:' before pushing.
 - For ']', it checks if the top of the stack is '[', then pops it and updates the max length if a complete accordion is found.
 - For '[:', it checks if the top of the stack is also '[:' before pushing.
 - For any other character, it returns -1.
3. If the stack is not empty at the end, it returns -1 (incomplete accordion).
4. The function returns the maximum length of a valid accordion found.

However, there are several issues with this implementation:

1. It doesn't handle the case where '[' appears after '[:' correctly.
2. It doesn't properly track the start of an accordion.
3. It doesn't correctly handle nested accordions or multiple accordions in the string.
4. It may return incorrect results for some valid inputs.

For example:

- For input "[::|:]", it will return 0 instead of 6.
- For input "[::][::]", it will return 2 instead of 4.
- For input "[::|:][::]", it will return 5 instead of 6.

These issues will lead to Wrong Answer (WA) for many test cases. The code doesn't have any syntax errors that would cause Compilation Error (CE), and it's unlikely to cause Runtime Error (RE) or Time Limit Exceeded (TLE) given the problem constraints .

Therefore, the correct answer is:

(C)

Figure 13: Output from Claude-3.5-Sonnet for the task in Figure 12. The answer C is correct.