LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code

Naman Jain[†] King Han[†] Alex Gu[*$] Wen-Ding Li[*‡]

Fanjia Yan[*†] Tianjun Zhang[*†] Sida I. Wang[§]

Armando Solar-Lezama[$] Koushik Sen[†] Ion Stoica[†]

[†] UC Berkeley [$] MIT [‡] Cornell [§] Meta AI

Website: `https://livecodebench.github.io/`

`naman_jain@berkeley.edu`

Abstract

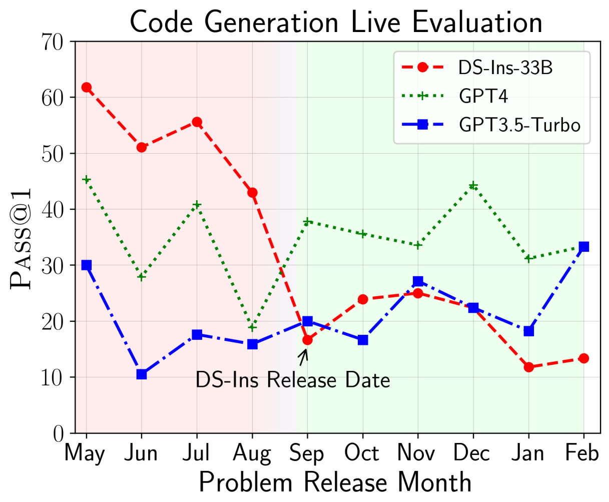
Large Language Models (LLMs) applied to code-related applications have emerged as a prominent domain, receiving significant interest from both the academic and industry communities. However, as new and improved LLMs are developed, prior evaluation benchmarks (e.g., HumanEval, MBPP) are no longer sufficient for understanding and judging their capabilities In this work, we propose LiveCodeBench, a holistic and contamination-free evaluation of LLMs for code that continuously collects *new* problems over time Notably, our benchmark also focuses on broader code-related capabilities, such as self-repair, code execution, and test output prediction, beyond mere code generation. Currently, LiveCodeBench hosts over three hundred high-quality coding problems published between May 2023 and January 2024. We evaluate 16 LLMs on LiveCodeBench and present novel empirical findings not revealed in prior benchmarks. We will release all prompts and model completions for further analysis by the community, as well as a general toolkit for easily adding new scenarios and models.

1 Introduction

Code has emerged to be an important application area for LLMs with a proliferation of various code-specific models (Chen et al., 2021; Li et al., 2022; Zhong et al., 2022; Wang et al., 2021; Austin et al., 2021; Allal et al., 2023; Li et al., 2023b; Roziere et al., 2023; AI, 2023; Luo et al., 2023; Royzen et al., 2023; Wei et al., 2023b; Ridnik et al., 2024) and their applications to various domains and tasks such as program repair (Zheng et al., 2024; Olausson et al., 2023), optimization (Madaan et al., 2023a), test generation (Steenhoek et al., 2023), documentation generation (Luo et al., 2024), tool-usage (Patil et al., 2023; Qin et al., 2024), SQL (Sun et al., 2023), etc. In contrast with these rapid advancements, evaluations have remained rather stagnant, and current benchmarks like HumanEval, MBPP, and APPS may paint a skewed or misleading picture. Firstly, coding is a multi-faceted skill but these benchmarks only focus on natural language-to-code tasks, thus ignoring the broader code-related capabilities. Moreover, these benchmarks suffer from potential contamination or overfitting challenges with benchmark samples present in the training datasets.

Motivated by these shortcomings, we introduce LiveCodeBench, a holistic and contamination-free benchmark for evaluating code capabilities. LiveCodeBench is built on the following principles

1. **Live updates to prevent contamination.** LLMs are trained on massive inscrutable corpora and current benchmarks suffer from the risk of data contamination by being included in those datasets. While prior works have attempted decontamination using both exact and fuzzy matches (Li et al., 2023b,c), it can be non-trivial task (Team, 2024) and evadable using simple strategies like rephrasing the program (Yang et al., 2023). Here, to prevent the risk of problem contamination, we use live updates, that is evaluate models on *new* problems. Particularly, we collect problems from weekly contests on competition platforms and tag them



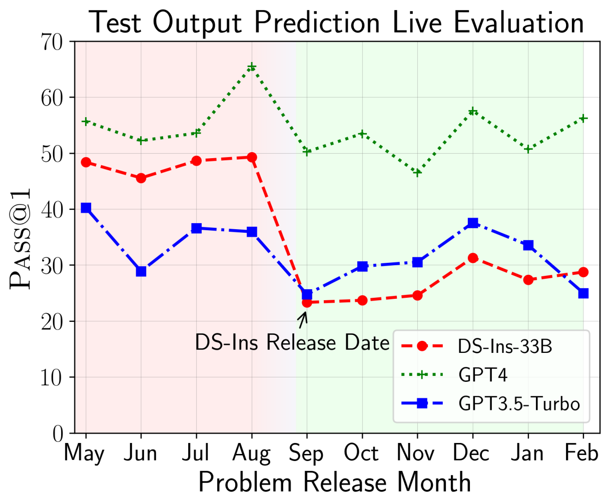


Figure 1: LiveCodeBench comprises problems marked with release dates allowing evaluations over different time windows. For newer models, we can detect and avoid contamination by only evaluating on time-windows after the model's cutoff date. The figures demonstrate the performance of models on code generation and test output prediction LiveCodeBench scenarios with LeetCode problems released across the months between May 2023 and January 2024. Notice that DS performs considerably worse on problems released since September 2023 (its release date!) – indicating potential contamination for the earlier problems. Thus, while performing evaluations, we use the post-September time window (shaded with green) for fairly comparing the two models.

with a *release date*. Next, for newer models, we only consider problems released after the cutoff date for the model to ensure that the model has not *seen* the exact problem in the training dataset. In Figure 1, we find that the performance of the DeepSeek model starkly drops when evaluated on the LeetCode problems released after August 2023. This indicates that DeepSeek is potentially trained on the older LeetCode problems.

2. **Holistic Evaluation.** Current code evaluations primarily focus on natural language to code generation. However, programming is a multi-faceted task and requires a variety of capabilities beyond what code generation measures. In LiveCodeBench, we evaluate code LLMs on three additional scenarios: (a) self-repair - fix the code of a generated program by leveraging the errors during program execution, (b) code execution - evaluate code comprehension by predicting the output of a program, given the source code and input, and (c) test output prediction - evaluate test generation capabilities from the program description, i.e., given the description and an input, generate a test with the correct expected output. Figure 2 (left) depicts performance on the different scenarios considered in LiveCodeBench.

3. **High-quality problems and tests.** High-quality problems and tests are crucial for reliably evaluating and comparing LLMs. However, prior works have revealed deficiencies in existing benchmarks. Liu et al. (2023a) identified insufficient tests and ambiguous problem descriptions in HumanEval and released a fixed *plus*-variant of the benchmark. Similarly, Austin et al. (2021) had to create a sanitized MBPP subset to disambiguate problem descriptions. In LiveCodeBench, we source the problems from reputable competition websites whose quality is already validated by the platform users. In addition, for every problem, we provide a large number of tests (over 58 on average) for meaningful and robust evaluations.

4. **Weighting problem difficulty.** Competition programming is challenging for even the best-

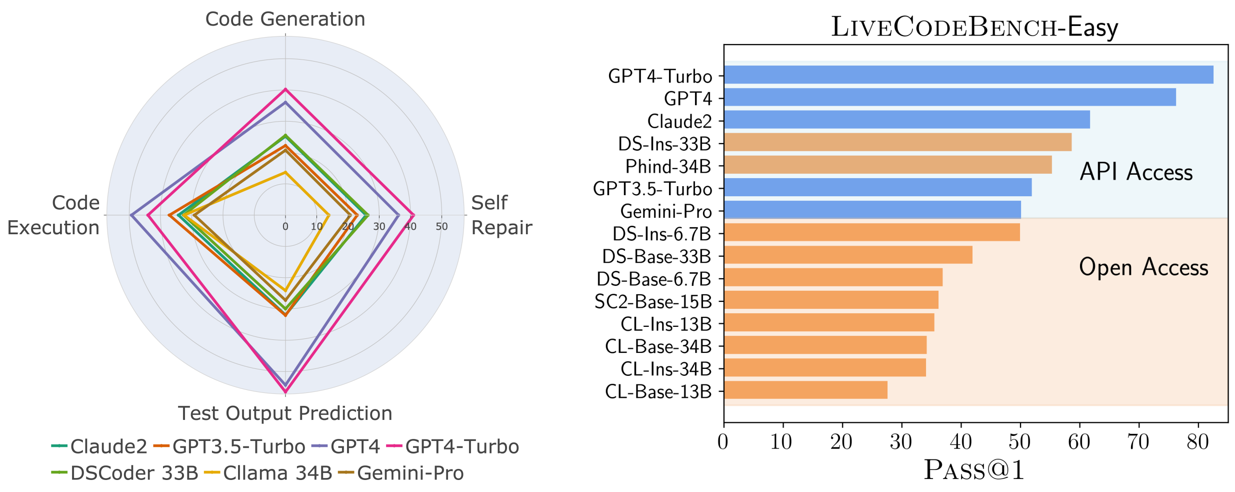


Figure 2: **Left.** We propose evaluating LLMs across scenarios capturing various coding-related capabilities. Specifically, we host four different scenarios, namely code generation, self-repair, code execution, and test output prediction. The figure depicts various model performances across the four scenarios available in LiveCodeBench in a radial plot – highlighting how relative differences across models change across the scenarios. **Right.** Comparison of open access and (closed) API access models on LiveCodeBench-Easy code generation scenario. We find that closed-access models consistently outperform the open models with only strong instruction-tuned variants of 30+B models (specifically DS-Ins-33B and Phind-34B models) crossing the performance gap.

performing LLMs and most of the current SoTA models achieve close to zero performance on those problems. As a result, they can be unsuitable for meaningful comparing today's LLMs because the variance in performances is low. Furthermore, evaluation scores are averaged across problems with different difficulty levels artificially minimizing the differences between models. In contrast, we use problem difficulty ratings (sourced from the competition websites) to ensure balanced problem difficulty distribution and allow granular model comparisons.

With these principles in mind, we build LiveCodeBench, a continuously updated benchmark avoding data contamination. We collect over 300 problems from contests across three competition platforms – LeetCode, AtCoder, and CodeForces occurring from May 2023 to the present (Jan 2024) and use them to construct the different LiveCodeBench scenarios.

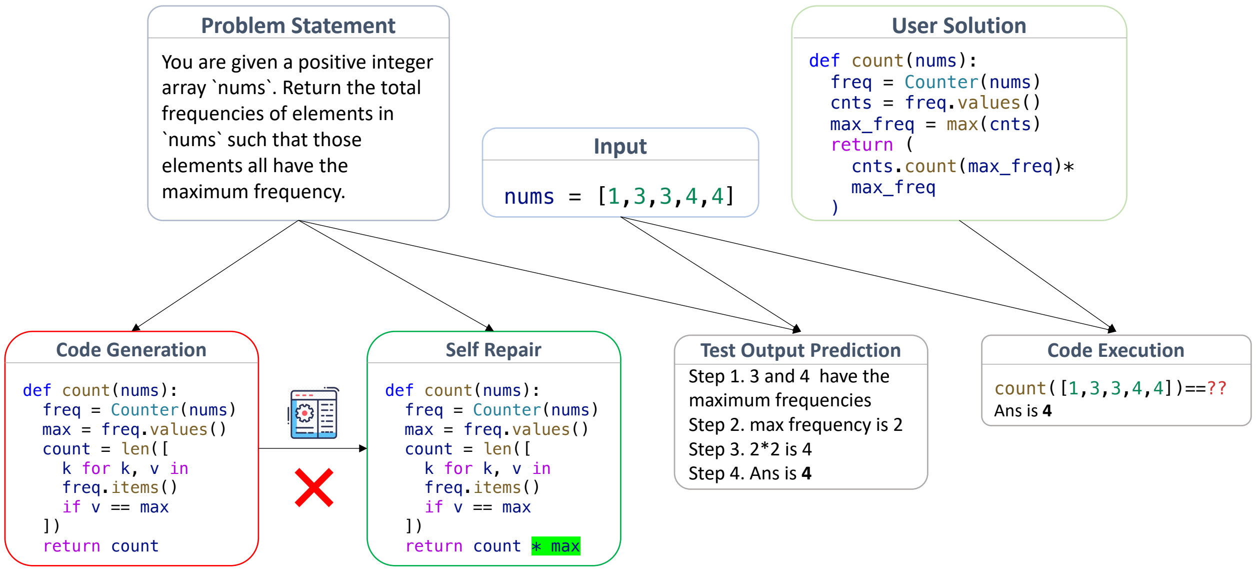
Empirical Findings. We evaluate 8 base models and 16 instruction-tuned models across different LiveCodeBench scenarios. Following, we present the empirical findings from our evaluations (that are not revealed in prior benchmarks).

1. We observe a stark drop in performance of DeepSeek on LeetCode problems released after August 2023 across all scenarios. This highlights likely contamination in the older problems.
2. Our holistic evaluation reveals that model comparison orderings are well correlated across tasks but the relative performance differences do vary. For example, the gap between open and closed models further increases on tasks like self-repair or test output prediction.
3. In comparison to HumanEval, we find that model performances are more distributed on LiveCodeBench (see Figure 5). Particularly, model performances vary between 30-80 on LiveCodeBench-Easy versus 60-80 on HumanEval+. This indicates that models might be overfitting to HumanEval which contains easier problems in comparison to LiveCodeBench.
4. GPT-4 outperform other models with a significant margin in contrast with existing code benchmarks where smaller models achieve performance similar to or better than GPT-4.
5. Among the base models, we find DeepSeek-Base models to be the strongest followed by StarCoder2-Base and CodeLLaMa-Base. These differences diminish after instruction tuning with Phind-34B (based on CL-Base-34B) catching up to DS-Ins-33B.
6. We study the gap between open access and (closed) API models and find that closed models outperform the open models (Figure 2). The open models that close the gap, i.e. DS-Ins-33B and Phind-34B are instruction-tuned variants of large base models (with > 30B parameters).

Concurrent Work. Huang et al. (2023) also evaluate LLMs in a time segmented manner. However, they only focus on CodeForces problems while we combine problems across platforms and additionally propose a holistic evaluation across multiple code-related scenarios. AI (2023) also evaluate DeepSeek on LeetCode problems and mention possibility of problem contamination.

2 Holistic Evaluation

LLMs are evaluated and compared on natural language to code generation tasks. However, this only captures one dimension of code-related capabilities. Indeed, real-world software engineering requires expertise in tasks beyond just *generation* – for instance synthesizing informative test cases, debugging incorrect code, understanding existing code, writing documentation, etc. These tasks are not merely for additional bookkeeping but are crucial parts of the software development process and improve the quality, maintainability, and reliability of the code (Boehm, 2006). This also



You are given a positive integer array `nums`. Return the total frequencies of elements in `nums` such that those elements all have the maximum frequency.

```
nums = [1,3,3,4,4]
```

```
def count(nums):
  freq = Counter(nums)
  cnts = freq.values()
  max_freq = max(cnts)
  return (
    cnts.count(max_freq)*
    max_freq
  )
```

```
def count(nums):
  freq = Counter(nums)
  max = freq.values()
  count = len([
    k for k, v in
    freq.items()
    if v == max
  ])
  return count
```

```
def count(nums):
  freq = Counter(nums)
  max = freq.values()
  count = len([
    k for k, v in
    freq.items()
    if v == max
  ])
  return count * max
```

Step 1. 3 and 4 have the maximum frequencies
Step 2. max frequency is 2
Step 3. 2*2 is 4
Step 4. Ans is **4**

```
count([1,3,3,4,4])==??
```
Ans is **4**

Figure 3: Overview of the different scenarios present in LiveCodeBench. Coding is multi-faceted and we propose evaluating LLMs on a suite of evaluation setups that capture various coding-related capabilities. Specifically, beyond the standard code generation setting, we consider three additional scenarios, namely self-repair, code execution, and a newly introduced test output prediction task.

maps to LLMs and adopting similar workflows enables does allow the models to generate better code. For example, AlphaCodium (Ridnik et al., 2024) built an intricate LLM pipeline for solving competition coding problems. The combined natural language reasoning, test case construction, code generation, and self-repair to solve problems. By combining these individual steps, they achieved significant improvements over a naive code generation baseline, showcasing the importance of these broader capabilities. Motivated by this, in this work we propose evaluating LLMs more holistically on a suite of evaluation setups that capture broader code-related capabilities.

Specifically, we evaluate code LLMs in four scenarios, namely code generation, self-repair, code execution, and test output prediction. Our selection criterion was to pick settings that are useful components in code LLM workflows and in addition, have clear and automated evaluation metrics. Following we describe each of these scenarios in detail.

Code Generation. The code generation scenario is the standard natural language to code generation setup. The model is given a problem statement with natural language description and example input-output examples and is tasked with generating a correct solution. The evaluation is performed via functional correctness using a set of *unseen* test cases comprising input-output pairs. We use the Pass@1 metric measured as the fraction of the problems for which the model was able to generate a program passing all tests. Figure 3 left provides an example of this scenario.

Self Repair. The self-repair scenario follows prior works on testing self-repair capabilities of LLMs (Olausson et al., 2023; Shinn et al., 2023; Chen et al., 2023). Here, the model is given a problem statement from which it generates a candidate program (similar to the single-step code generation scenario above). However, in case of a mistake, the model is additionally provided with error feedback (either the exception message or a failing test case in case of incorrect code generation) and is tasked with generating the fixed solution. Similar to the code generation scenario, the evaluation is performed via functional correctness on the final program, i.e. either the single-step correct generation or the attempted repair. We use the Pass@1 metric to measure the combined model performance after the repair step. Figure 3 mid-left provides an example of this scenario.

Code Execution. The code execution scenario follows the output prediction setup of CRUXEval Gu et al. (2024). The model is provided a program snippet consisting of a function (`f`) along with a test input to the program and is tasked with predicting the output of the program on the input test case. The evaluation is performed via an execution-based correctness metric where the model generation is considered correct if `assert f(input) == generated_output` passes. Figure 3 right provides an example of the code execution scenario.

Test Case Output Prediction. Finally, we introduce a new task aimed at studying natural language reasoning and test generation. Here, the model is given the problem statement along with a test case input and is tasked with generating the expected output for the input. This task follows a setup similar to that in CodeT (Chen et al., 2022), where tests are generated solely from problem statements without the implementation of the function. The main difference is that we provide a fixed set of test inputs for each problem in our dataset, and the models are then prompted to predict the expected output for those specific inputs. This approach allows for a straightforward evaluation of the test generation capabilities by avoiding test input prediction, a hard-to-evaluate task. Figure 3 mid-right provides an example of this scenario.

Finally, we would like to point out that LiveCodeBench also offers an extensible framework to add new scenarios in the future. So other relevant settings like input generation, program summarization, optimization, etc. can be integrated with our setup as they are studied further.

3 Benchmark Curation

We curate our problems from three coding competition websites, namely LeetCode, AtCoder, and CodeForces. These websites periodically host contests containing problems that assess the coding and problem-solving skills of participants. The problems comprise a natural language problem statement along with example input-output examples and the goal is to write a program that passes a set of *hidden* tests. Further, thousands of participants solve these problems thus ensuring that the problems are vetted for ambiguity and correctness.

3.1 Data Collection

We write HTML scrapers for each of the above websites to collect problems and the corresponding metadata. To ensure quality and consistency, we parse mathematical formulas and remove problems with images. We additionally remove problems which are not amenable to be graded by input-output examples such as the ones that accept multiple correct answers or the ones that require constructing data structures. Besides parsing the problem descriptions, we also collect associated ground truth solutions and test cases whenever directly available. Thus for each problem, we collect tuples of natural language problem statement P, test cases T, and ground truth solution S. Finally, we associate the contest date D to mark the release date of each problem and use the collected attributes to construct problems for our four scenarios (detailed in Section 3.3 ahead).

Scrolling through time. As noted, we associate the contest date D for each problem. The release date allows us the measure the performance of LLMs over different time windows by filtering problems based on whether the problem release date lies in the time window (referred to as "scrolling" through time). This is crucial for evaluating and comparing models trained at different times. Specifically, for a new model and the corresponding cutoff date (normalized to the release date if the training cutoff date is not published), we can measure the performance of the model on benchmark problems released after the cutoff date. We have developed a UI that allows comparing models on problems released during different time windows (shown in Figure 7).

Platform	Total Count	#Easy	#Medium	#Hard	Average Tests
LCB (May-Jan)	349	122	148	79	61.9
LCB (Sep-Jan)	223	80	91	52	61.9
AtCoder	171	62	61	48	28.2
LeetCode	169	56	85	28	96.9
CodeForces	9	4	2	3	44.3
LCB-Easy	122	122	0	0	53.1
LCB-Medium	148	0	148	0	69.9
LCB-Hard	79	0	0	79	60.3

Table 1: The statistics of problems collected in LiveCodeBench (LCB). We present the number of problems, their difficulty distributions and the average number of tests per problem. We present the results on the following subsets of LiveCodeBench (used throughtout this manuscript) - (a) problems in the May-Jan and Sep-Jan time-windows, (b) problems sourced from the three platforms, and (c) problems in the LCB-Easy LCB-Medium, and LCB-Hard subsets.

Test collection. Tests are crucial for evaluating the correctness of the generated programs and are used in all four scenarios. We collect public-facing tests whenever available and use them for the benchmark. Otherwise, following Liu et al. (2023b), we use LLMs (here GPT-4-Turbo) to generate tests for the problems. Notably, instead of generating inputs directly using LLM, we construct generators that sample inputs based on the problem specifications using few-shot prompting. Details and examples of such input generators can be found in Section A.2.

Problem difficulty. Competition programming has remained challenging for LLMs with GPT-4 achieving an average ELO of 392 on CodeForces (bottom 5 percentile! OpenAI (2023)). This makes comparing LLMs hard as performance variation across models is low. In LiveCodeBench, we collect problems with diverse difficulties (labeled in competition platforms), removing problems rated as very difficult. Further, we use these ratings to classify problems as Easy, Medium, and Hard for more granular model comparisions.

3.2 Platform Specific Curation

We describe the curation process for each platform.

LeetCode. We collect problems from all weekly and biweekly contests on LeetCode happening since May'23. For each problem, we collect the problems, public tests, and user solutions. The platform additionally provides a difficulty label for each problem which we use to tag the problems as Easy, Medium, and Hard. Since LeetCode provides a starter code for each problem, we additionally collect it and provide it to the LLM for the STDIN format. Since the hidden tests are not directly available, we use our generator-based test input generation approach (Section A.2).

AtCoder. We collect problems from the `abc` (beginner round) contests on AtCoder happening after April'23. We deliberately avoid the more challenging `arc` and `agc` contests which cater to more advanced Olympiad (preparing) participants. The problems are tagged with numeric difficulty ratings and we further remove `abc` problems with rating more than 500. The difficulty ratings are also used to tag the problems as Easy, Medium, and Hard. Specifically, we use the rating brackets $[0-200)$, $[200-400)$, and $[400-500]$ to perform the classification. AtCoder provides public and hidden tests for each problem which we directly use in the benchmark.

CodeForces. We collect problems from the Division 3 and Division 4 contests on CodeForces. Notably, we find that even with this filter, the problems are harder than the other two platforms.

CodeForces also provides difficulty ratings for the problems which we use to tag the problems as Easy, Medium, and Hard using the rating brackets $\{800\}$, $(800 - 1000]$, and $(1000 - 1300]$ respectively. Due to the higher difficulty, we only consider a small fraction of problems from CodeForces and semi-automatically construct test case generators since they don't provide complete tests. Table 1 provides various statistics about the problems collected in LiveCodeBench.

3.3 Scenario-specific benchmark construction

Code Generation and Self-Repair. We used the natural language problem statement as the problem statement for these scenarios. For LeetCode, as noted above, an additional starter code is provided for the functional input format. For AtCoder and CodeForces problems, we use the standard input format (similar to Hendrycks et al. (2021)). The collected or generated tests are next for evaluating the correctness of the generated programs. Our final dataset consists of 349 problem instances across the three platforms.

Code Execution. We draw inspiration from the benchmark creation procedure of CRUXEval. First, we collect a large pool of $\sim$ 2000 *correct, human-submitted solutions* from the LeetCode subset. However, many of these programs have multiple nested loops, complex numerical computations, and a large number of execution steps. Therefore, we apply compile-time and run-time filters to ensure samples are reasonable, which was double-checked with a manual inspection. More details on the filtering criteria and statistics of the dataset can be found in Appendix A.3. Our final dataset consists of 479 samples from 85 problems[1].

Test Case Output Prediction. We take the natural language problem statement from the LeetCode platform and use the example test inputs to construct our test case output prediction dataset. Since the example test inputs in the problems are reasonable test cases for humans to reason through, they also serve as ideal test inputs for LLMs to work out. Our final dataset consists of 411 problem instances from a total of 170 LeetCode problems.

4 Experiment Setup

We describe the experimental setup here. First, common setup across the scenarios is provided followed by scenario-specific setups in Section 4.1.

Models. We evaluate 24 models across various sizes ranging from 1.3B vs 34B, base models vs instruct models, and open models vs closed models. Since we consider zero-shot evaluations for some scenarios, we only use instruction-tuned models. Our experiments includes models from different classes GPTs (GPT-3.5-turbo, GPT-4, GPT-4-Turbo), Claudes (Claude-Ins-1, Claude-2), Gemini-Pro among closed-access and DeepSeeks (DS-{1.3, 6.7, 33}B, DS-{1.3, 6.7, 33}B), CodeLLaMas (CL-Ins-{7, 13, 34}B, CL-Base-{7, 13, 34}B), StarCoder2 (SC2-Base-{7,15}B) among open. Additionally, we also include fine-tuned models WizardCoder (WC-34B) and Phind-34B from CL-Base-34B, and MagiCoders (MC-6.7, 7B) from CL-Base-7B and DS-Ins-6.7B. See Appendix C.1 for their details and estimated cutoff or release dates.

Evaluation Metrics. We use Pass@1 (Kulal et al., 2019; Chen et al., 2021) for our evaluations. Specifically, we generate 10 candidate answers for each problem (temperature 0.2) and calculate the correct fraction. For code generation and self-repair scenarios, we use tests to check program correctness. For code execution and test output prediction scenarios, we parse the generated response to extract the answer and use object equivalence to determine the correctness.

[1]Code execution scenario currently uses problems between May-Dec.

4.1 Scenario specific setup

The setup for different scenarios is presented below. Note that the base models are only used in the code generation scenario since they do not easily follow the format for the other scenarios.

Code Generation. We use zero-shot prompting for the instruction-tuned models and one-shot prompting for the fine-tuned models. We use a constant one-shot example for the base models, with a separate example for the problems accepting stdin input and functional output. For the zero-shot prompt, following Hendrycks et al. (2021), we add appropriate instructions to generate solutions in functional or stdin format. Section C.2 shows the high-level zero-shot prompt used.

Self Repair. Similar to prior work Olausson et al. (2023), we use the programs generated during the code generation scenario along with the corresponding error feedback to build the zero-shot prompt for the self-repair scenario. The type of error feedback includes syntax errors, runtime errors, wrong answers, and time-limit errors, as applicable. Section C.3 provides the pseudo-code for computing the error feedback and the corresponding prompt.

Code Execution. We use few-shot prompting for code execution with and without chain-of-thought prompting (COT). Particularly, we use a 2-shot prompt without COT and a 1-shot prompt with COT with manually detailed steps. The prompts are detailed in Section C.4.

Test Output Prediction. We use a zero-shot prompt querying the model to complete assertions given the problem, function signature, and test input. We provide the prompt in Section C.5.

5 Results

We first describe how LiveCodeBench helps detect and avoid benchmark contamination in Section 5.1. Next, we present the findings from our evaluations on LiveCodeBench in Section 5.2.

5.1 Avoiding Contamination

LiveCodeBench comprises of problems released since May 2023. However, DeepSeek models were released in Sep 2023 and might be already trained on some of the problems in our benchmark. Since we attribute problem release dates, we can measure the performance of the model on the benchmark on problems released after the cutoff date, estimating the performance of the model on uncontaminated problems. Figure 1 shows the performance of DS-Ins-33B on LiveCodeBench code generation problems released over between May 2023 and Jan 2024. We notice a stark drop in performance after Aug. 2023, hinting that the DeepSeek model might indeed be contaminated. This trend is consistent across other LiveCodeBench scenarios (Figure 8 in appendix). Concurrently, AI (2023) (Section 4.1 last para) also acknowledges possible data contamination in LeetCode – "*achieved higher scores in the LeetCode Contest held in July and August*".

5.2 Model Performance Findings

We evaluate 16 models in our study ranging from closed access to open access (with their various fine-tuned variants) on LiveCodeBench scenarios. To overcome contamination issues in DeepSeek models, we only consider problems released since Sep 2023 for all experiments. Figure 4 shows the performance of a subset of models on all four scenarios. We highlight our key findings below.

Holistic Evaluations. We evaluate the models across the four scenarios currently available in LiveCodeBench. Figure 2 shows the performance of models on all scenarios jointly along the axes of the polar chart. First, we observe that the relative order of models remains mostly consistent across the scenarios. This is also supported by high correlations between Pass@1 metric across

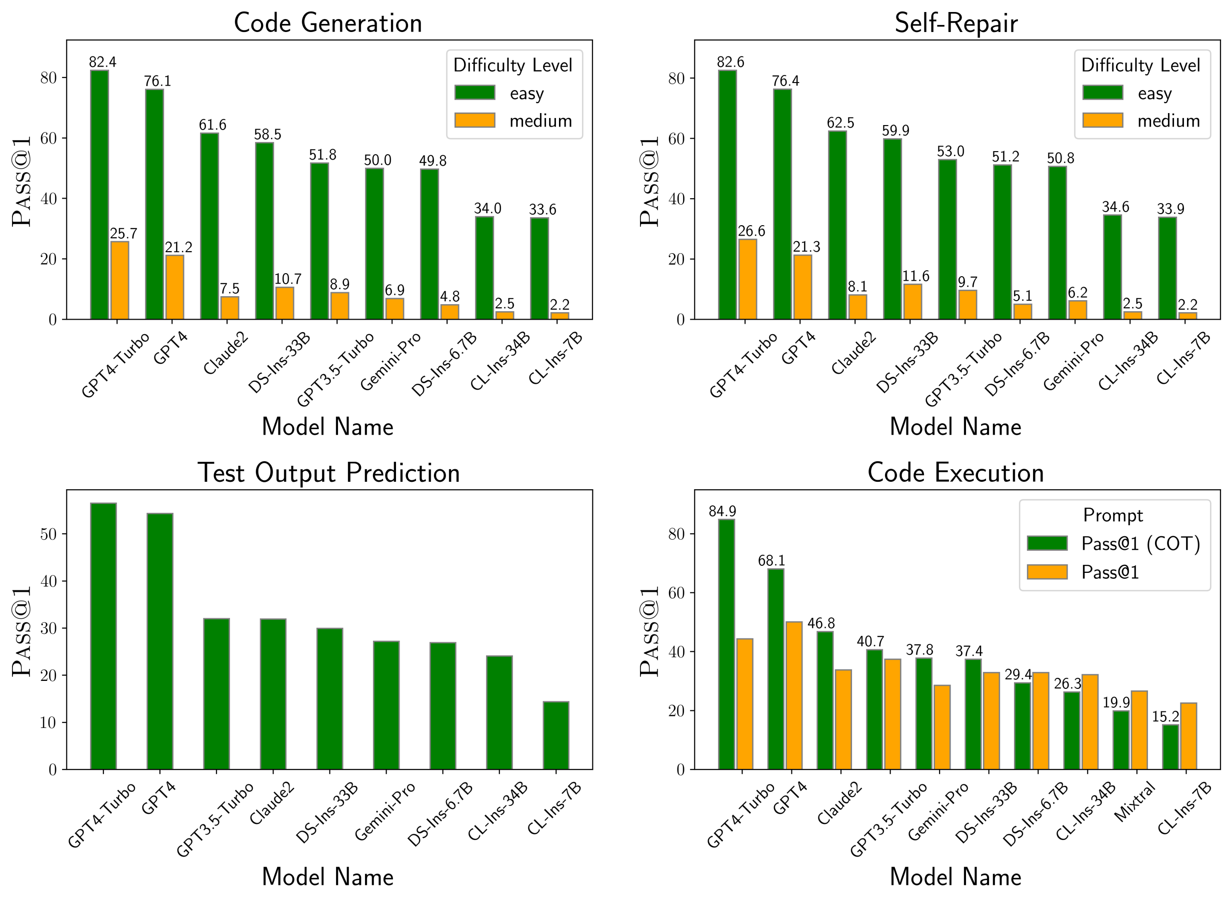


Figure 4: Model performances across the four scenarios available in LiveCodeBench. The top-left and top-right plots depict Pass@1 of models on easy and medium splits across the code generation and self-repair scenarios respectively (results on hard subset deferred to the Appendix). The bottom-left and bottom-right plots depict Pass@1 of models across the test output prediction and code execution scenarios respectively.

the scenarios – over 0.88 across all pairs. Interestingly, correlations are stronger between related tasks – code generation and self-repair scenarios with 0.98 correlation and code execution and test output prediction with 0.95.

However, even though well correlated, the relative differences in performances do change across the scenarios. For example, GPT-4 (closed access models in general) perform even better on test output prediction tasks. Similarly, the difference between GPT-4-Turbo and GPT-4 becomes even more pronounced in the self-repair and code execution scenarios highlighting the benefit of holistic evals.

Comparision to HumanEval. We next compare how performances translate between LiveCodeBench and HumanEval, the primary benchmark used for evaluating coding capabilities. Figure 5 shows a scatter plot of Pass@1 on HumanEval versus LiveCodeBench code generation scenario, filtered on the easy subset. We only find a moderate correlation (0.81) with much larger performance variations on LiveCodeBench (from 29.2 to 82.3) while HumanEval shows a more consistent range (59.8 to 81.3). Additionally, all the points lie above and to the left of the diagonal connecting the origin and GPT-4-Turbo marker. This means that models that perform well on HumanEval are often weaker on LiveCodeBench. This highlights two key findings – (a.)

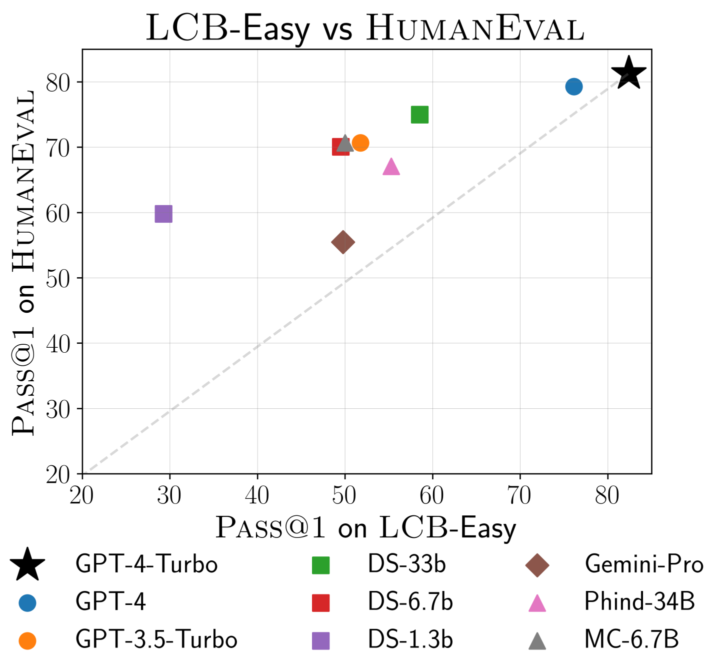


Figure 5: Scatter plot comparing Pass@1 of models on HumanEval versus easy subset of LiveCodeBench code generation scenario. We find larger variation in LiveCodeBench performances (~ 50 points difference) versus HumanEval with only ~ 20 points difference thus allowing better discrimination between models. Further, all the points lie to the left of the diagonal between origin and GPT-4-Turbo highlighting that models performing well on HumanEval are often weaker on LiveCodeBench. This indicates potential overfitting on HumanEval starkly observed for DS-Ins-1.3B which achieves Pass@60 on HumanEval but only achieves Pass@30 on LCB-Easy subset.

LiveCodeBench problems allow better discrimination between models and (b.) models that perform well on HumanEval might be overfitting on the benchmark and their performances do not translate well beyond such problems.

Indeed HumanEval is an easier benchmark with small and isolated programming problems and thus easier to overfit. For instance, DS-Ins-1.3B which achieves 59.8% Pass@1 on HumanEval only 29.2% Pass@1 on LiveCodeBench easy subset. GPT-4 model in contrast achieves close to 80% Pass@1 on both HumanEval and LiveCodeBench. Finally, we wish to highlight that LiveCodeBench-easy provides a good discriminatory evaluation set for code LLMs escaping contamination and overfitting.

Vacuum beyond GPT-4. One distinct observation from our evaluations is the large gap between GPT-4 and other models. This gap is only amplified by GPT-4-Turbo which further improves over GPT-4 across all tasks (only with COT for code execution). We analyze GPT-4-Turbo generated code samples and find that the new model generates more readable code. Specifically, the code consists of more inline natural language comments that reason about code generation.We verify this quantitatively and find GPT-4-Turbo generated uses 1.4× more tokens than GPT-4, primarily due to these comments.

Comparing open models. We also compare various fine-tuned variants of the DeepSeek and CodeLLaMa base models across different model sizes. Good instruction-tuned variants of these models (DS-Ins-33B and Phind-34B) can come close to or outperform various closed models. Thus, a combination of strong base models and good fine-tuning datasets allows building good code LLMs.

6 Related Work

Language Models for Code Generation. Starting with Codex (Chen et al., 2021), there are over a dozen code LLMs. These include CodeT5 (Wang et al., 2021, 2023), CodeGen (Nijkamp et al., 2022), SantaCoder (Allal et al., 2023), StarCoder (Li et al., 2023b), AlphaCode (Li et al., 2022), InCoder (Fried et al., 2022), and CodeGeeX (Zheng et al., 2023). As of January 2024, DeepSeek-Coder (Bi et al., 2024) and Code Llama (Roziere et al., 2023) are the two most popular open models. Many downstream models resulted from fine-tuning them on synthetically generated data, such as WizardCoder (Luo et al., 2023), Phi (Gunasekar et al., 2023), Magicoder (Wei et al., 2023b), and Phind.

Code Generation Benchmarks. Many benchmarks have been proposed to compare and evaluate these models. These primarily focus on natural language to Python code generation: HumanEval (Chen et al., 2021), HumanEval+ (Liu et al., 2023b), APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), MBPP (Austin et al., 2021), L2CEval (Ni et al., 2023). Their variants have been proposed to cover more languages, (Wang et al., 2022a; Zheng et al., 2023; Cassano et al., 2022; Athiwaratkun et al., 2022).

A few benchmarks specifically measure competitive programming, such as APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), xCodeEval (Khan et al., 2023), and LeetCode-Hard (Shinn et al., 2023). Methods such as AlphaCode (Li et al., 2022), AlphaCode 2(Gemini Team et al., 2023), ALGO (Zhang et al., 2023d), Parsel (Zelikman et al., 2022), code cleaning (Jain et al., 2023), code explainations (Li et al., 2023a), analogical reasoning (Yasunaga et al., 2023), and AlphaCodium (Ridnik et al., 2024) have been pushing the boundaries of what is possible with LLMs in this domain.

The biggest differentiating factor between LiveCodeBench and these benchmarks is that our benchmark is *continuously updated* and contains *more tasks* such as code repair, code execution, and test output prediction capturing more facets of code reasoning and code generation.

There are also benchmarks to evaluate code generation in data science applications, such as DS-1000 (Lai et al., 2023), ARCADE (Yin et al., 2022), NumpyEval (Zhang et al., 2023b), and PandasEval (Jain et al., 2022). Going one step further, some benchmarks also measure ability to use API's or perform more general software engineering tasks, such as JuICe (Agashe et al., 2019), APIBench (Patil et al., 2023), RepoBench (Liu et al., 2023c), ODEX (Wang et al., 2022b), SWE-Bench (Jimenez et al., 2023), GoogleCodeRepo (Shrivastava et al., 2023), RepoEval (Zhang et al., 2023a), and Cocomic-Data (Ding et al., 2022).

Finally, there are a variety of benchmarks for other tasks, such as code translation (Roziere et al., 2020; Zhu et al., 2022; Ahmad et al., 2021), test case generation (Tufano et al., 2022; Watson et al., 2020), code search (Husain et al., 2019), type prediction (Mir et al., 2022; Wei et al., 2023a; Malik et al., 2019), commit message generation (Liu et al., 2020), code summarization (LeClair et al., 2019; Iyer et al., 2016; Barone and Sennrich, 2017; Hasan et al., 2021; Alon et al., 2018), code security (Liguori et al., 2022; Pearce et al., 2022; Tony et al., 2023), program repair (Jiang et al., 2023; Xia et al., 2022; Tufano et al., 2019; Haque et al., 2022; Jin et al., 2023; Gupta et al., 2017; Berabi et al., 2021), performance optimization (Garg et al., 2022; Madaan et al., 2023a), and so on.

Code Repair. (Chen et al., 2023; Olausson et al., 2023; Madaan et al., 2023b; Peng et al., 2023; Zhang et al., 2023c) investigate self-repair, using error messages as feedback for models to improve inspiring our code repair scenario.

Code Execution. Code execution was first studied in Austin et al. (2021); Nye et al. (2021) LiveCodeBench's execution scenario is particularly inspired by CRUXEval (Gu et al., 2024), a recent benchmark measuring the reasoning and execution abilities of code LLMs. We differ from CRUXEval in that our benchmark is live, and our functions are more complex and human-produced (unlike Code Llama generations in CRUXEval).

Test Generation. Test generation using LLMs has been explored in Yuan et al. (2023); Schäfer et al. (2024). Furthermore, Chen et al. (2022) demonstrated that LLMs can assist in generating test case inputs/outputs for competitive programming problems, thereby improving the accuracy of the generated code, thus inspiring our test generation scenario. However, LiveCodeBench's test generation scenario is unique in that it decouples the test inputs and outputs allowing more proper evaluations.

Contamination: Data contamination and test-case leakage have received considerable attention Oren et al. (2024); Golchin and Surdeanu (2023); Weller et al. (2023); Roberts et al. (2024) as LLMs might be getting trained on benchmarks. Sainz et al. (2023) demonstrated contamination by simply prompting the model highlighting contamination. Some detection methods have also been built to avoid these cases Shi et al. (2023); Zhou et al. (2023). Our work avoids contamination by live updates.

7 Conclusion

In this work, we propose LiveCodeBench, a new benchmark for evaluating LLMs for code. Our benchmark mitigates contamination issues in existing benchmarks by introducing live evaluations and emphasizing scenarios beyond code generation to account for the broader coding abilities of LLMs. LiveCodeBench is an extensible framework, that will keep on updating with new problems, scenarios, and models. Our evaluations reveal novel findings such as contamination detection and potential overfitting on HumanEval. We hope LiveCodeBench with serve to advance understanding of current code LLMs and also guide future research in this area through our findings.

Limitations

Live Evaluation Set Size. LiveCodeBench currently hosts over 300 problems for all scenarios starting from problems released between May and January. To evaluate models released after May, we only perform evaluations on problems released after the model cutoff date. Thus for very recent models, we might only be able to use very recent problems, potentially leading to a small evaluation set. We highlight the following two solutions. First, for some models the actual cutoff date might be earlier than the release date allowing us to use the older problems, increasing the size of the evaluation set size. Second, going forward, we will collect problems for other competition platforms, allowing collecting larger amount of recent problems. In addition, we can also supplement this with an *unreleased* private test set constructed specifically for evaluating the models with similar format to current benchmark problems, preventing contamination altogether while also providing public access to *new* problems released on the platforms.

Single Programming Language. LiveCodeBench currently only focuses on Python which might not provide enough signal about model capabilities in other languages. However, since we collected problem statements and serialized tests, adding new programming languages would be straightforward once appropriate evaluation engines are used.

Robustness to Prompts. Recent works have identified huge performance variances that can be caused due to insufficient prompt. Here, we either do not tune prompts across models or make minor adjustments based on the system prompts and delimiter tokens. This can lead to performance variance in our results. Our findings and model comparison orders generalize across LiveCodeBench scenarios and mostly match the performance trends observed on HumanEval making this a less prominient issue. We do emphasize that $1-2\%$ performance differences can be attributed to variance from problem evaluation window or prompt used and **recommend exercising such variance into account when comparing models** using LiveCodeBench.

This issue can be particularly observed open models on the code execution scenario with COT prompting. Interestingly, often the open models perform even worse in comparsion to the direct code execution baseline. Note that we used same prompts for the closed models all of which show noticable improvement from COT. While the used prompts might be sub-optimal, this highlights how open-models perform worse against the closed models at performing chain-of-thought.

Acknowledgements

This work was supported in part by NSF grants CCF-1900968, CCF-1908870 and by SKY Lab industrial sponsors and affiliates Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Nexla, Samsung SDS, Uber, and VMware. A. Gu is supported by the National Science Foundation (NSF) Graduate Research Fellowship under Grant No. 2141064. A. Solar-Lezama is supported by the National Science Foundation (NSF) and Intel Corporation through NSF Grant CCF:2217064. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position of the sponsors.
Finally, we thank Federico Cassano, Horace He, Aman Madaan, Manish Shetty for helpful feedback at various stages of the work.

References

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. Juice: A large scale distantly supervised dataset for open domain context-based code generation. *arXiv preprint arXiv:1910.02216*.

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021. Avatar: A parallel corpus for java-python program translation. *arXiv preprint arXiv:2108.11590*.

DeepSeek AI. 2023. Deepseek coder: Let the code write itself. `https://github.com/deepseek-ai/DeepSeek-Coder`.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*.

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

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275.*

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, pages 780–791. PMLR.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954.*

Barry Boehm. 2006. A view of 20th and 21st century software engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 12–29, New York, NY, USA. Association for Computing Machinery.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227.*

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397.*

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374.*

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128.*

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007.*

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *preprint arXiv:2204.05999.*

Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B Clement, Neel Sundaresan, and Chen Wu. 2022. Deepperf: A deep learning-based approach for improving software performance. *arXiv preprint arXiv:2206.13619.*

A Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805.*

Shahriar Golchin and Mihai Surdeanu. 2023. Time travel in llms: Tracing data contamination in large language models. *arXiv preprint arXiv:2308.08493.*

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065.*

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31.

Md Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2022. Fixeval: Execution-based evaluation of program fixes for competitive programming problems.

Masum Hasan, Tanveer Muttaqueen, Abdullah Al Ishtiaq, Kazi Sajeed Mehrab, Md Mahim Anjum Haque, Tahmid Hasan, Wasi Uddin Ahmad, Anindya Iqbal, and Rifat Shahriyar. 2021. Codesc: A large code-description parallel dataset. *arXiv preprint arXiv:2105.14220*.

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

Yiming Huang, Zhenghao Lin, Xiao Liu, Yeyun Gong, Shuai Lu, Fangyu Lei, Yaobo Liang, Yelong Shen, Chen Lin, Nan Duan, et al. 2023. Competition-level problems are effective llm evaluators. *arXiv preprint arXiv:2312.02143*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *54th Annual Meeting of the Association for Computational Linguistics 2016*, pages 2073–2083. Association for Computational Linguistics.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231.

Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E Gonzalez, Koushik Sen, and Ion Stoica. 2023. Llm-assisted code cleaning for training accurate code generators. *arXiv preprint arXiv:2311.14904*.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020*.

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

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. *arXiv preprint arXiv:2303.07263*.

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.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806. IEEE.

Jierui Li, Szymon Tworkowski, Yingying Wu, and Raymond Mooney. 2023a. Explaining competitive-level programming solutions using llms. *arXiv preprint arXiv:2307.05337*.

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

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023c. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463*.

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

Pietro Liguori, Erfan Al-Hossami, Domenico Cotroneo, Roberto Natella, Bojan Cukic, and Samira Shaikh. 2022. Can we generate shellcodes via natural language? an empirical study. *Automated Software Engineering*, 29(1):30.

Hanmeng Liu, Ruoxi Ning, Zhiyang Teng, Jian Liu, Qiji Zhou, and Yue Zhang. 2023a. Evaluating the logical reasoning ability of chatgpt and gpt-4. *arXiv preprint arXiv:2304.03439*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*.

Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. 2020. Atom: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering*, 48(5):1800–1817.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023c. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, et al. 2024. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. 2023a. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867.*

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023b. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651.*

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. Nl2type: inferring javascript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 304–315. IEEE.

Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2241–2252.

Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, et al. 2023. L2ceval: Evaluating language-to-code generation capabilities of large language models. *arXiv preprint arXiv:2309.17446.*

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations.*

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114.*

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying gpt self-repair for code generation. *arXiv preprint arXiv:2306.09896.*

R OpenAI. 2023. Gpt-4 technical report. arxiv 2303.08774. *View in Article.*

Yonatan Oren, Nicole Meister, Niladri Chatterji, Faisal Ladhak, and Tatsunori B Hashimoto. 2024. Proving test set contamination for black-box language models. In *The Twelfth International Conference on Learning Representations.*

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334.*

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE.

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, and Jianfeng Gao. 2023. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813.*

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark

Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. 2024. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*.

Manley Roberts, Himanshu Thakur, Christine Herlihy, Colin White, and Samuel Dooley. 2024. To the cutoff... and beyond? a longitudinal perspective on LLM data contamination. In *The Twelfth International Conference on Learning Representations*.

Michael Royzen, Justin Wei, and Russell Coleman. 2023. Phind.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611.

Oscar Sainz, Jon Ander Campos, Iker García-Ferrero, Julen Etxaniz, and Eneko Agirre. 2023. Did chatgpt cheat on your test?

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.

Weijia Shi, Anirudh Ajith, Mengzhou Xia, Yangsibo Huang, Daogao Liu, Terra Blevins, Danqi Chen, and Luke Zettlemoyer. 2023. Detecting pretraining data from large language models. *arXiv preprint arXiv:2310.16789*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pages 31693–31715. PMLR.

Benjamin Steenhoek, Michele Tufano, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Reinforcement learning from automatic feedback for high-quality unit test generation. *arXiv preprint arXiv:2310.02368*.

Ruoxi Sun, Sercan O Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. 2023. Sql-palm: Improved large language modeladaptation for text-to-sql. *arXiv preprint arXiv:2306.00739*.

Gemini Team. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context.

Catherine Tony, Markus Mutas, Nicolás E Díaz Ferreyra, and Riccardo Scandariato. 2023. Llmseceval: A dataset of natural language prompts for security evaluations. *arXiv preprint arXiv:2303.09384*.

Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. 2022. Methods2test: A dataset of focal methods mapped to test cases. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 299–303.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. 2022a. Recode: Robustness evaluation of code generation models. *arXiv preprint arXiv:2212.10264*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022b. Execution-based evaluation for open-domain code generation. *arXiv preprint arXiv:2212.10481*.

Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1398–1409.

Jiayi Wei, Greg Durrett, and Isil Dillig. 2023a. Typet5: Seq2seq type inference using static analysis. *arXiv preprint arXiv:2303.09564*.

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

Orion Weller, Marc Marone, Nathaniel Weir, Dawn Lawrie, Daniel Khashabi, and Benjamin Van Durme. 2023. " according to..." prompting language models improves quoting from pre-training data. *arXiv preprint arXiv:2305.13252*.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical program repair in the era of large pre-trained language models. *arXiv preprint arXiv:2210.14179*.

Shuo Yang, Wei-Lin Chiang, Lianmin Zheng, Joseph E. Gonzalez, and Ion Stoica. 2023. Rethinking benchmark and contamination for language models with rephrased samples.

Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H Chi, and Denny Zhou. 2023. Large language models as analogical reasoners. *arXiv preprint arXiv:2310.01714*.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, et al. 2022. Natural language to code generation in interactive data science notebooks. *arXiv preprint arXiv:2212.09248*.

Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D Goodman, and Nick Haber. 2022. Parsel: A unified natural language framework for algorithmic reasoning. *arXiv preprint arXiv:2212.10561*.

Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*.

Kechi Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023b. Toolcoder: Teach code generation models to use apis with search tools. *arXiv preprint arXiv:2305.04032*.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023c. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787, Toronto, Canada. Association for Computational Linguistics.

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

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

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *https://arxiv.org/abs/2402.14658*.

Maosheng Zhong, Gen Liu, Hongwei Li, Jiangling Kuang, Jinshan Zeng, and Mingwen Wang. 2022. Codegen-test: An automatic code generation model integrating program test information. *arXiv preprint arXiv:2202.07612*.

Kun Zhou, Yutao Zhu, Zhipeng Chen, Wentong Chen, Wayne Xin Zhao, Xu Chen, Yankai Lin, Ji-Rong Wen, and Jiawei Han. 2023. Don't make your llm an evaluation benchmark cheater. *arXiv preprint arXiv:2311.01964*.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*.

A Dataset

A.1 License

Similar to Hendrycks et al. (2021), we scrape only the problem statements, ground-truth solutions, and test cases from coding challenge websites – LeetCode, AtCoder, and CodeForces. Further, we only scrape publicly visible portions of websites, avoiding any data collection that might be pay-walled or require interaction with the website. We would like to emphasize that we use the collected problems for academic purposes only and in addition do not train on the collected problems. Some of the portions scraped may be copyrighted and take the necessary measures before releasing our benchmark.

A.2 Generator Based Test Generation

We use GPT-4-Turbo to construct input generators. The following prompts provide one-shot prompt templates used for synthesizing random and adversarial input generators. These generations are executed to construct inputs which validated on the collected correct programs.

A.3 Code Execution

The code execution split of LiveCodeBench consists of 479 samples from 85 distinct problems. To encourage diversity in our benchmark while keeping our benchmark small and usable, we place a limit of six samples for each given problem. These sample programs and corresponding test cases are chosen uniformly at random from all those passing the filter.

Filtering Criteria: The specific filtering criteria are as follows:

- Compile time: length of code is between 100 and 500 characters, no syntax errors, all necessary imports are included
- Runtime: no floating point operations, true division, exp, other integer operations must have at least one argument ≤ 3, string and list operations must have at least one argument with length ≤ 3, finish running in 2 seconds, "reasonable" number of steps (roughly, under 1000 Python bytecode operations).

We give two examples of two programs that are filtered out in the Listings below. Our final benchmark consists of 479 samples from 85 problems, but will increase in size due to its live nature.

Dataset Statistics: As mentioned, we filter for codes between 100 and 500 characters, as well as below 1000 steps. The statistics for programs in our dataset, from this lens, are shown in Fig. 6.

```
You are an expert python competitive programmar and your goal is to construct input-
    ↪ generators for testing programming contest problems. You will write relevant
    ↪ generators and finally construct ‘construct_inputs‘ function that returns a list
    ↪  of diverse inputs sampled from the generator. Remember to strictly follow the
    ↪ instructions and constraints present in the problem statement

<USER>
{EXAMPLE_PROBLEM}

Construct a random input generator.
</USER>

<ASSISTANT>
import numpy as np
def random_input_generator(weight_min, weight_max, size_min, size_max):
    weights_size = np.random.randint(size_min, size_max+1)
    weights = np.random.randint(weight_min, weight_max, size=weights_size).tolist()
    k = np.random.randint(1, len(weights)+1)
    return weights, k

def construct_inputs():
    inputs_list = []
    ## small inputs
    for i in range(15):
        inputs_list.append(random_input_generator(1, 10**3, 1, 10))
    ## medium inputs
    for i in range(15):
        inputs_list.append(random_input_generator(1, 10**6, 1, 10**3))
    ## large inputs
    for i in range(15):
        inputs_list.append(random_input_generator(1, 10**9, 1, 10**5))
    return inputs_list
</ASSISTANT>

<USER>
{PROBLEM}

Construct a random input generator. Use the format used in the above example by
    ↪ returning a single function that builds diverse inputs named ‘construct_inputs‘
</USER>
```

Prompt for random input generation

```
You are an expert python competitive programmar and your goal is to construct input-
    ↪ generators for testing programming contest problems. You will write relevant
    ↪ generators and finally construct ‘construct_inputs‘ function that returns a list
    ↪  of diverse inputs sampled from the generator. Remember to strictly follow the
    ↪ instructions and constraints present in the problem statement

<USER>
{EXAMPLE_PROBLEM}

Construct an adversarial input generator.
</USER>

<ASSISTANT>
import numpy as np
## case 1 - alternating large and small weights
def generate_adversarial_inputs_1(weight_size, max_weight, k):
    weights = [1 if i%2==0 else max_weight for i in range(weight_size)]
    return weights, k

## case 2 - equal_weights
def adversarial_input_generator_2(weight_size, max_weight, k):
    weights = [max_weight for _ in range(weight_size)]
    return weights, k

# Case 3 - Large weights at the ends
def adversarial_input_generator_3(weight_size, max_weight, k):
    pass ## truncatede
def construct_inputs():
    inputs_list = []

    weight_sizes = [10, 1000, 100000]
    max_weights = [10**3, 10**6, 10**9]

    for weight_size in weight_sizes:
        for max_weight in max_weights:
            ks = [1, 2, 5, weight_size//2, weight_size-1, weight_size]
            for k in ks:
                inputs_list.append(generate_adversarial_inputs_1(weight_size,
    ↪ max_weight, k))
                # truncated
    return inputs_list
</ASSISTANT>

<USER>
{PROBLEM}

Construct an adversarial input generator. Use the format used in the above example by
    ↪ returning a single function that builds diverse inputs named ‘construct_inputs‘
</USER>
```

Prompt for adversarial input generation

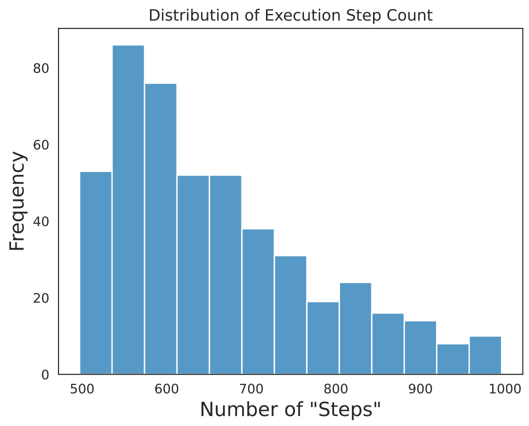
```
def check(x, t):
    if x == '':
        return t == 0
    if t < 0:
        return False
    for i in range(len(x)):
        if check(x[:i], t - int(x[i:])):
            return True
    return False

@cache
def punishmentNumber(n: int) -> int:
    if n == 0:
        return 0
    ans = punishmentNumber(n-1)
    if check(str(n * n), n):
        ans += n * n
    return ans
assert punishmentNumber(n = 37) == 1478
```

Program filtered because of multiplication

```
dp = [True for _ in range(int(1e6 + 5))]
MAXN = int(1e6 + 5)
p = []
dp[0] = False
dp[1] = False
for i in range(2, MAXN):
    if not dp[i]: continue
    p.append(i)
    for j in range(2 * i, MAXN, i):
        dp[j] = False
def findPrimePairs(n: int) -> List[List[int]]:
    res = []
    for i in range(1, n):
        if n % 2 == 1 and i > n//2: break
        if n % 2 == 0 and i > n//2: break
        if dp[i] and dp[n - i]:
            res.append([i, n - i])
    return res
assert findPrimePairs(n = 2) == []
```

Program filtered because of control flow



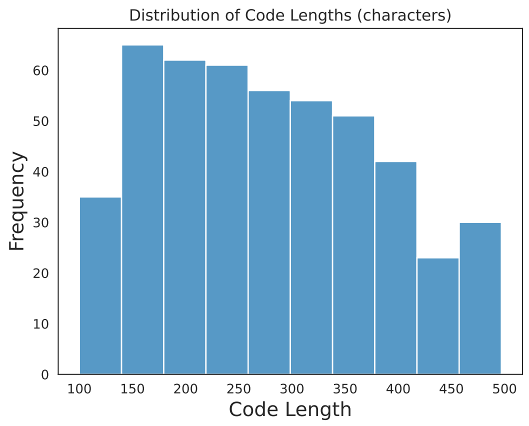
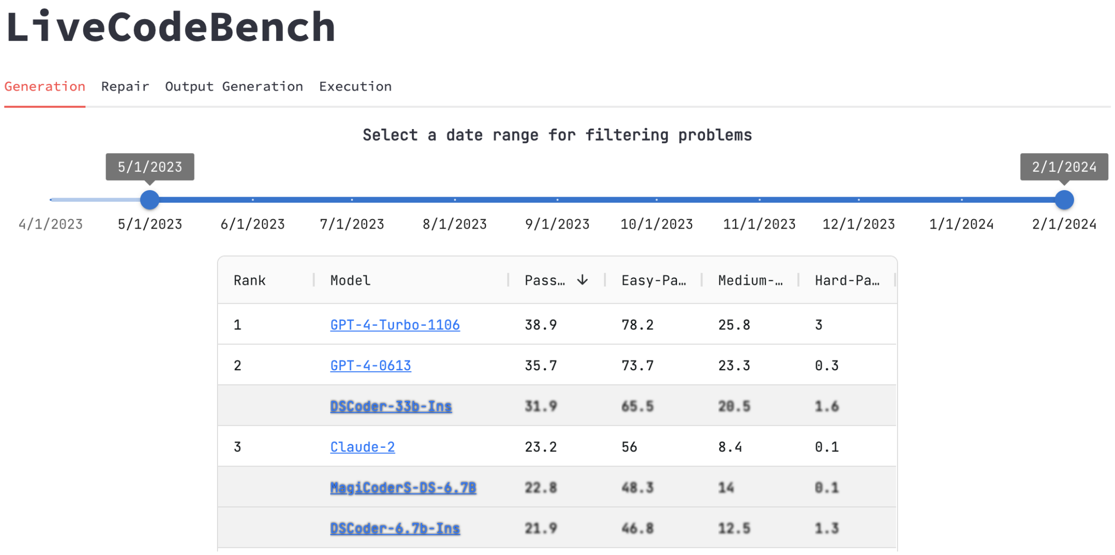


Figure 6: Distribution of code lengths and number of execution steps

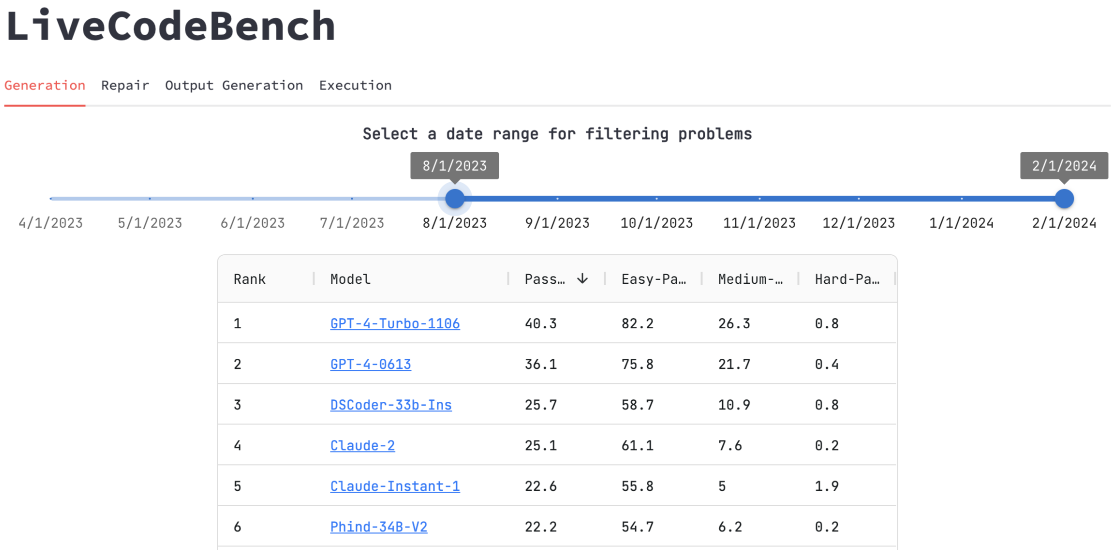
B UI



Select a date range for filtering problems

5/1/2023 – 2/1/2024

Rank	Model	Pass… ↓	Easy-Pa…	Medium-…	Hard-Pa…
1	GPT-4-Turbo-1106	38.9	78.2	25.8	3
2	GPT-4-0613	35.7	73.7	23.3	0.3
	DSCoder-33b-Ins	31.9	65.5	20.5	1.6
3	Claude-2	23.2	56	8.4	0.1
	MagiCoderS-DS-6.7B	22.8	48.3	14	0.1
	DSCoder-6.7b-Ins	21.9	46.8	12.5	1.3



Select a date range for filtering problems

8/1/2023 – 2/1/2024

Rank	Model	Pass… ↓	Easy-Pa…	Medium-…	Hard-Pa…
1	GPT-4-Turbo-1106	40.3	82.2	26.3	0.8
2	GPT-4-0613	36.1	75.8	21.7	0.4
3	DSCoder-33b-Ins	25.7	58.7	10.9	0.8
4	Claude-2	25.1	61.1	7.6	0.2
5	Claude-Instant-1	22.6	55.8	5	1.9
6	Phind-34B-V2	22.2	54.7	6.2	0.2

Figure 7: UI of LiveCodeBench showing two views – May-Jan and Sep-Jan. The contaminated models are blurred and the performance difference is visible across the two views. The scroller on the top allows selecting different periods of time highlighting the live nature of the benchmark.

C Experimental Setup

C.1 Models

We describe the details of models considered in our study in Table 2.

Model ID	**Short Name**	**Approximate Cutoff Date**	**Link**
deepseek-ai/deepseek-coder-33b-instruct	DSCoder-33b-Ins	2023-07-30	deepseek-coder-33b-instruct
deepseek-ai/deepseek-coder-6.7b-instruct	DSCoder-6.7b-Ins	2023-07-30	deepseek-coder-6.7b-instruct
deepseek-ai/deepseek-coder-1.3b-instruct	DSCoder-1.3b-Ins	2023-07-30	deepseek-coder-1.3b-instruct
codellama/CodeLlama-34b-Instruct-hf	Cllama-34b-Ins	2023-01-01	CodeLlama-34b-Instruct-hf
codellama/CodeLlama-13b-Instruct-hf	Cllama-13b-Ins	2023-01-01	CodeLlama-13b-Instruct-hf
codellama/CodeLlama-7b-Instruct-hf	Cllama-7b-Ins	2023-01-01	CodeLlama-7b-Instruct-hf
WizardLM/WizardCoder-Python-34B	WCoder-34B-V1	2023-01-01	WizardCoder-Python-34B-V1.0
Phind/Phind-CodeLlama-34B-v2	Phind-34B-V2	2023-01-01	Phind-CodeLlama-34B-v2
mistralai/Mixtral-8x7B-v0.1	Mixtral-8x7B-0.1	2023-04-01	Mixtral-8x7B-v0.1
gpt-3.5-turbo-0301	GPT-3.5-Turbo-0301	2021-10-01	OpenAI
gpt-3.5-turbo-1106	GPT-3.5-Turbo-1106	2021-10-01	OpenAI
gpt-3.5-turbo-0613	GPT-3.5-Turbo-0613	2021-10-01	OpenAI
gpt-4-0314	GPT-4-0314	2021-10-01	OpenAI
gpt-4-0613	GPT-4-0613	2021-10-01	OpenAI
gpt-4-1106-preview	GPT-4-Turbo-1106	2023-04-30	OpenAI
claude-2	Claude-2	2022-12-31	claude-2
claude-instant-1	Claude-Instant-1	2022-12-31	introducing-claude
gemini-pro	Google-Gemini-Pro	2023-05-01	gemini-api-developers-cloud
ise-uiuc/Magicoder-S-DS-6.7B	MagiCoderS-DS-6.7B	2023-07-30	Magicoder-S-DS-6.7B
ise-uiuc/Magicoder-S-CL-7B	MagiCoderS-CL-7B	2023-01-01	Magicoder-S-CL-7B

Table 2: Language Models Overview

C.2 Code Generation

Below we provide the prompt format (with appropriate variants adding special tokens accommodating each instruct-tuned model) used for this scenario.

~~~
You are an expert Python programmer. You will be given a question (problem
    ↪ specification) and will generate a correct Python program that matches the
    ↪ specification and passes all tests. You will NOT return anything except for the
    ↪ program

### Question:\n{question.question_content}


{ if question.starter_code }
 ### Format: {PromptConstants.FORMATTING_MESSAGE}

```python
{question.starter_code}
~~~
{ else }
### Format: {PromptConstants.FORMATTING_WITHOUT_STARTER_MESSAGE}

```python
# YOUR CODE HERE
```
{ endif }

### Answer: (use the provided format with backticks)
```

Code Generation Prompt

C.3 Self Repair

Below we provide the prompt format (with appropriate variants adding special tokens accommodating each instruct-tuned model) used for this scenario.

C.4 Code Execution

Below we provide the prompts for code execution with and without CoT. The prompts are modified versions of those from (Gu et al., 2024) to fit the format of the samples in our benchmark.

C.5 Test Output Prediction

Below we provide the prompt format (with appropriate variants adding special tokens accommodating each instruct-tuned model) used for this scenario.
```

```
{if check_result.result_status is "Wrong Answer"}
The above code is incorrect and does not pass the testcase.
Input: {wrong_testcase_input}
Output: {wrong_testcase_output}
Expected: {wrong_testcase_expected}


{elif check_result.result_status is "Time Limit Exceeded"}
The above code is incorrect and exceeds the time limit.
Input: {wrong_testcase_input}


{elif check_result.result_status is "Runtime Error"}
The above code is incorrect and has a runtime error.
Input: {wrong_testcase_input}
Error Message: {wrong_testcase_error_message}

{endif}
```

Self Repair Error Feedback Pseudocode

```
You are a helpful programming assistant and an expert Python programmer. You are
    ↪ helping a user write a program to solve a problem. The user has written some
    ↪ code, but it has some errors and is not passing the tests. You will help the
    ↪ user by first giving a concise (at most 2-3 sentences) textual explanation of
    ↪ what is wrong with the code. After you have pointed out what is wrong with the
    ↪ code, you will then generate a fixed version of the program. You must put the
    ↪ entired fixed program within code delimiters only for once.

### Question:\n{question.question_content}

### Answer: ‘‘‘python
{code.code_to_be_corrected}
‘‘‘

### Format: {PromptConstants.FORMATTING_CHECK_ERROR_MESSAGE}

### Answer: (use the provided format with backticks)
```

Self-Repair Prompt

```
You are given a Python function and an assertion containing an input to the function.
    ↪ Complete the assertion with a literal (no unsimplified expressions, no function
    ↪ calls) containing the output when executing the provided code on the given input
    ↪ , even if the function is incorrect or incomplete. Do NOT output any extra
    ↪ information. Provide the full assertion with the correct output in [ANSWER] and
    ↪ [/ANSWER] tags, following the examples.

[PYTHON]
def repeatNumber(number : int) -> int:
    return number
assert repeatNumber(number = 17) == ??
[/PYTHON]
[ANSWER]
assert repeatNumber(number = 17) == 17
[/ANSWER]

[PYTHON]
def addCharacterA(string : str) -> str:
    return string + "a"
assert addCharacterA(string = "x9j") == ??
[/PYTHON]
[ANSWER]
assert addCharacterA(string = "x9j") == "x9ja"
[/ANSWER]

[PYTHON]
{code}
assert {input} == ??
[/PYTHON]
[ANSWER]
```

Code Execution Prompt

```
You are given a Python function and an assertion containing an input to the function.
    ↪ Complete the assertion with a literal (no unsimplified expressions, no function
    ↪ calls) containing the output when executing the provided code on the given input
    ↪ , even if the function is incorrect or incomplete. Do NOT output any extra
    ↪ information. Execute the program step by step before arriving at an answer, and
    ↪ provide the full assertion with the correct output in [ANSWER] and [/ANSWER]
    ↪ tags, following the examples.

[PYTHON]
def performOperation(s):
    s = s + s
    return "b" + s + "a"
assert performOperation(s = "hi") == ??
[/PYTHON]
[THOUGHT]
Let's execute the code step by step:

1. The function performOperation is defined, which takes a single argument s.
2. The function is called with the argument "hi", so within the function, s is
    ↪ initially "hi".
3. Inside the function, s is concatenated with itself, so s becomes "hihi".
4. The function then returns a new string that starts with "b", followed by the value
    ↪ of s (which is now "hihi"), and ends with "a".
5. The return value of the function is therefore "bhihia".
[/THOUGHT]
[ANSWER]
assert performOperation(s = "hi") == "bhihia"
[/ANSWER]

[PYTHON]
{code}
assert {input} == ??
[/PYTHON]
[THOUGHT]
```
Code Execution Prompt with CoT

```
### Instruction: You are a helpful programming assistant and an expert Python
    ↪ programmer. You are helping a user to write a test case to help to check the
    ↪ correctness of the function. The user has written a input for the testcase. You
    ↪ will calculate the output of the testcase and write the whole assertion
    ↪ statement in the markdown code block with the correct output.

Problem:
{problem_statement}

Function:
‘‘‘
{function_signature}
‘‘‘
Please complete the following test case:

‘‘‘
assert {function_name}({testcase_input}) == # TODO
‘‘‘
### Response:
```

Test Output Prediction Prompt

D Results

D.1 Contamination

Figure 8 demonstrates contamination in DeepSeek in self repair and test output prediction scenarios.

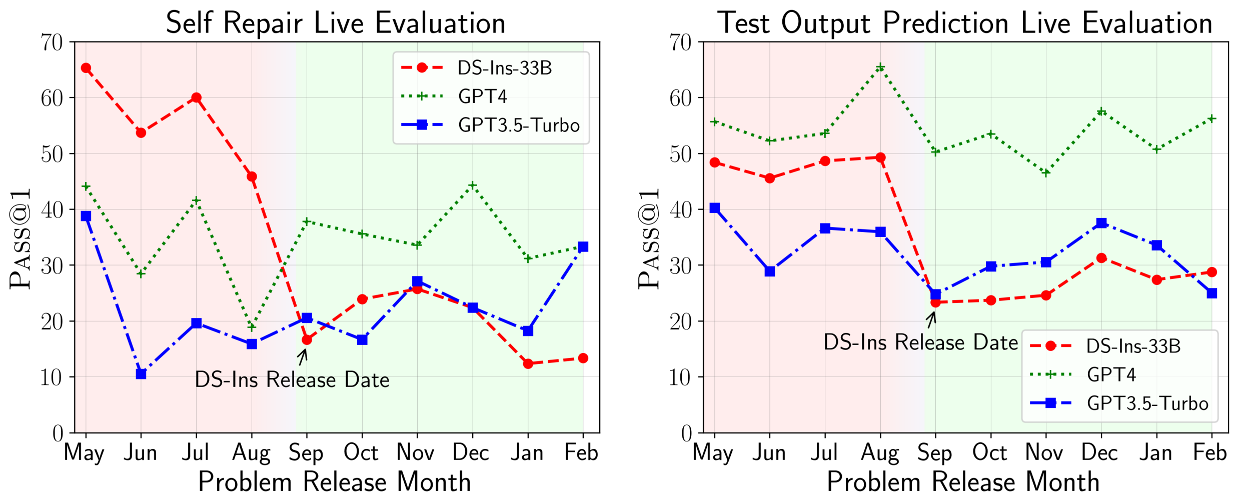


Figure 8: Contamination in DS models across self-repair and testgen scenarios over time

D.2 All Results

Below we provide the tables comprising of results across different LiveCodeBench scenarios.

Table 3: Code Generation Performances

Model Name	Easy	Medium	Hard	Total
Claude-2	61.6	7.5	0.2	23.1
Claude-Instant-1	56.4	4.9	1.9	21.1
Cllama-13b-Ins	35.4	2.3	0.0	12.6
Cllama-34b-Ins	34.0	2.5	1.9	12.8
Cllama-7b-Ins	33.6	2.2	0.0	11.9
DSCoder-1.3b-Ins	29.2	1.8	0.0	10.3
DSCoder-33b-Ins	58.5	10.7	0.8	23.3
DSCoder-6.7b-Ins	49.8	4.8	1.2	18.6
GPT-3.5-Turbo-0301	51.8	8.9	0.2	20.3
GPT-4-0613	76.1	21.2	0.4	32.6
GPT-4-Turbo-1106	82.4	25.7	0.8	36.3
Google-Gemini-Pro	50.4	5.8	0.2	18.8
MagiCoderS-CL-7B	33.2	2.0	0.0	11.7
MagiCoderS-DS-6.7B	49.5	7.8	0.0	19.1
Phind-34B-V2	55.2	6.0	0.2	20.5
WCoder-34B-V1	46.1	4.2	0.2	16.8

Table 4: Test Output Prediction Performances

Model Name	Pass@1
Claude-2	31.9
Claude-Instant-1	24.4
Cllama-13b-Ins	22.8
Cllama-34b-Ins	24.1
Cllama-7b-Ins	14.4
DSCoder-1.3b-Ins	14.7
DSCoder-33b-Ins	30.0
DSCoder-6.7b-Ins	26.9
GPT-3.5-Turbo-0301	32.0
GPT-4-0613	54.3
GPT-4-Turbo-1106	56.5
Google-Gemini-Pro	27.2
MagiCoderS-CL-7B	21.6
MagiCoderS-DS-6.7B	27.5
Phind-34B-V2	27.6
WCoder-34B-V1	23.1

Table 5: Code Execution Performances

Model Name	Pass@1	Pass@1 (COT)
Claude-2	33.8	46.8
Claude-Instant-1	26.8	41.6
Cllama-13b-Ins	26.7	14.9
Cllama-34b-Ins	32.2	26.3
Cllama-7b-Ins	22.6	15.2
CodeTulu-34b	32.6	37.4
DSCoder-1.3b-Base	24.4	18.0
DSCoder-1.3b-Ins	21.8	19.9
DSCoder-33b-Base	35.8	35.0
DSCoder-33b-Ins	32.9	37.4
DSCoder-6.7b-Base	31.1	27.8
DSCoder-6.7b-Ins	32.9	29.4
GPT-3.5-Turbo-0301	37.4	40.7
GPT-3.5-Turbo-0613	38.3	40.1
GPT-4-0613	50.1	68.1
GPT-4-Turbo-1106	44.4	84.9
Google-Gemini-Pro	28.6	37.8
Mixtral-8x7B-0.1	26.7	19.9

E Qualitative Examples

E.1 Code Execution

We show 5 examples from the code execution task that GPT-4 (`gpt-4-1106-preview`) still struggles to execute, even with CoT.

```python
def countWays(nums: List[int]) -> int:
    nums.sort()
    n = len(nums)
    ans = 0
    for i in range(n + 1):
        if i and nums[i-1] >= i: continue
        if i < n and nums[i] <= i: continue
        ans += 1
    return ans
assert countWays(nums = [6, 0, 3, 3, 6, 7, 2, 7]) == 3
# GPT-4 + CoT Outputs: 1, 2, 4, 5
```

Program filtered because of multiplication

```python
def minimumCoins(prices: List[int]) -> int:

    @cache
    def dfs(i, free_until):
        if i >= len(prices):
            return 0

        res = prices[i] + dfs(i + 1, min(len(prices) - 1, i + i + 1))

        if free_until >= i:
            res = min(res, dfs(i + 1, free_until))

        return res

    dfs.cache_clear()
    return dfs(0, -1)
assert minimumCoins(prices = [3, 1, 2]) == 4
# GPT-4 + CoT Outputs: 1, 3, 5, 6
```

Program filtered because of multiplication

```python
def sortVowels(s: str) -> str:
    q = deque(sorted((ch for ch in s if vowel(ch))))
    res = []
    for ch in s:
        if vowel(ch):
            res.append(q.popleft())
        else:
            res.append(ch)
    return ''.join(res)
assert sortVowels(s = 'lEetcOde') == 'lEOtcede'
# GPT-4 + CoT Outputs: "leetecode", "lEetecOde", "leetcede", "leetcEde", "leetcOde"
```

Program filtered because of multiplication

```python
def relocateMarbles(nums: List[int], moveFrom: List[int], moveTo: List[int]) -> List[
    ↪ int]:

    nums = sorted(list(set(nums)))
    dd = {}
    for item in nums:
        dd[item] = 1
    for a,b in zip(moveFrom, moveTo):
        del dd[a]
        dd[b] = 1
    ll = dd.keys()
    return sorted(ll)
assert relocateMarbles(nums = [1, 6, 7, 8], moveFrom = [1, 7, 2], moveTo = [2, 9, 5])
    ↪ == [5, 6, 8, 9]
# GPT-4 + CoT Outputs: [2, 6, 8, 9], [2, 5, 6, 8, 9], KeyError
```

Program filtered because of multiplication

```python
def minimumSum(nums: List[int]) -> int:
    left, right, ans = [inf], [inf], inf
    for num in nums:
        left.append(min(left[-1], num))
    for num in nums[::-1]:
        right.append(min(right[-1], num))
    right.reverse()
    for i, num in enumerate(nums):
        if left[i] < num and right[i + 1] < num:
            ans = min(ans, num + left[i] + right[i + 1])
    return ans if ans < inf else -1
assert minimumSum(nums = [6, 5, 4, 3, 4, 5]) == -1
# GPT-4 + CoT Outputs: 10, 11, 12
```

Program filtered because of multiplication