Planning-Driven Programming: A Large Language Model Programming Workflow

Chao Lei, Yanchuan Chang, Nir Lipovetzky, Krista A. Ehinger
School of Computing and Information Systems, The University of Melbourne, Australia
`clei1@student.unimelb.edu.au`,
`{yanchuan.chang, nir.lipovetzky, kris.ehinger}@unimelb.edu.au`

Abstract

The strong performance of large language models (LLMs) raises extensive discussion on their application to code generation. Recent research suggests continuous program refinements through visible tests to improve code generation accuracy in LLMs. However, these methods suffer from LLMs' inefficiency and limited reasoning capacity. In this work, we propose an LLM programming workflow (LPW) designed to improve both initial code generation and subsequent refinements within a structured two-phase workflow. Specifically, the solution generation phase formulates a solution plan, which is then verified through visible tests to specify the intended natural language solution. Subsequently, the code implementation phase drafts an initial code according to the solution plan and its verification. If the generated code fails the visible tests, the plan verification serves as the intended solution to consistently inform the refinement process for correcting bugs. Compared to state-of-the-art methods across various existing LLMs, LPW significantly improves the Pass@1 accuracy by up to 16.4% on well-established text-to-code generation benchmarks. LPW also sets new state-of-the-art Pass@1 accuracy, achieving 98.2% on HumanEval, 84.8% on MBPP, 59.3% on LiveCode, 62.6% on APPS, and 34.7% on CodeContests, using GPT-4o as the backbone. Our code is publicly available at: https://github.com/you68681/lpw.

1 Introduction

Code generation, also known as *program synthesis*, studies the automatic construction of a program that satisfies a specified high-level input requirement (Gulwani et al., 2017). Recently, large language models (LLMs) pre-trained on extensive code-related datasets (Brown et al., 2020; Meta, 2024; Li et al., 2023; Roziere et al., 2023; Achiam et al., 2023; Muennighoff et al., 2023) have shown success in code-related tasks, such as code generation from natural language descriptions, also named as text-to-code generation (Chen et al., 2021; Austin et al., 2021; Li et al., 2022), code translation (Pan et al., 2024; Yang et al., 2024), and code completion (Izadi et al., 2024). However, LLM-based code generation remains challenging due to stringent lexical, grammatical, and semantic constraints (Scholak et al., 2021).

Code generation substantially benefits from the empirical insights of human programmers. In practice, human programmers develop high-quality code by consistently rectifying errors through the analysis of test case executions (Huang et al., 2023c; Chen et al., 2023b). Different studies have refined programs based on execution results and LLM-generated information such as code and error explanation (Tang et al., 2023; Shinn et al., 2023; Madaan et al., 2023). Recent work further optimizes refinement (debugging) by performing *rubber duck debugging* processes (Chen et al., 2023b) and leveraging control flow graph information to assist LLMs in locating bugs (Zhong et al., 2024). However, the absence of precise correction instructions in feedback messages results in numerous refinements that deviate from the intended solution. Additionally, refining programs that significantly diverge from the problem description remains an open challenge (Tian and Chen, 2023).

To replicate each phase of program development, several studies (Lin et al., 2024; Qian et al., 2024; Dong et al., 2023b) have employed LLM instances as customized agents, assigning them diverse roles and facilitating their collaboration. Recent work incorporates extra visible tests (Huang et al., 2023a) and solution plans (Islam et al., 2024) generated by designated agents to improve the code refinements in multi-agent collaborations. However, the absence of methodologies for generating reliable visible tests and plans in these studies undermines their credibility, as incorrect visible tests and plans

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 12647–12684
July 27 - August 1, 2025 ©2025 Association for Computational Linguistics

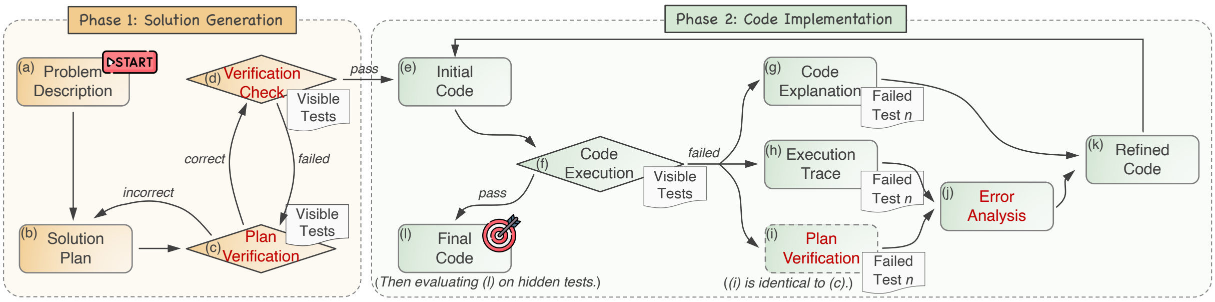


Figure 1: The pipeline of LPW, where the components highlighted in red are exclusive to LPW.

can lead to erroneous codes. Besides, with an increased number of agents, multi-agent collaborations consume significant token resources for communication (Huang et al., 2023a). The detailed related work is discussed in Appendix A.

In this work, we propose LPW, a *large language model programming workflow*, specifically for text-to-code generation, addressing the aforementioned limitations. LPW involves two phases for code generation: the solution generation phase for plan creation and plan verification, and the code implementation phase for initial code development and subsequent refinements. The pipeline of LPW is depicted in Figure 1. LPW leverages various information, including LLM-generated solution plan (Jiang et al., 2023) (block (b)), LLM-generated code explanation (Chen et al., 2023b) (block (g)), and runtime information from program execution (Zhong et al., 2024) (block (h)) to boost the code generation performance, and effectively incorporates them into an end-to-end framework. In LPW, aside from runtime information, all other messages are autonomously generated by LLMs using few-shot prompting, without additional training.

A unique feature of LPW is incorporating plan verification (block (c)) as the natural language intended solution for visible tests. LPW initially produces a solution plan that decomposes a complex problem into several tractable sub-problems (intermediate steps) (block (b)). LPW then verifies the solution plan against visible tests to assess its correctness, known as plan verification. For a visible test, the LLM-generated verification includes a text-based step-by-step analysis to derive the output for each intermediate step and the final output, ensuring that the final output is consistent with the visible test result. Subsequently, each inferred intermediate output is reviewed by LLMs (block (d)) to maintain logical consistency and mitigate hallucination throughout the verification. The plan verification encompasses comprehensive conditions and logical specifications for solving visible tests, eliminating potential misunderstandings before code generation. This is akin to *Test-Driven Development*, where human developers validate the intended solution with test cases (Beck, 2022).

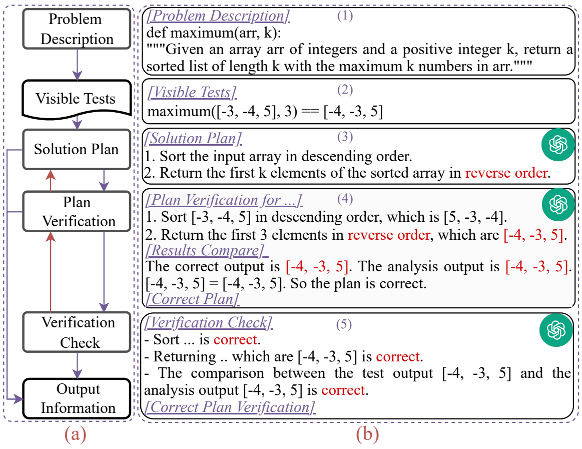
LPW uses the plan and its verification in the initial code development (block (e)) to ensure that the initial code closely aligns with the problem description. Furthermore, LPW incorporates plan verification in the subsequent refinements. By comparing discrepancies between the expected output of each intermediate step, as recorded in the plan verification, against the execution trace on the failed visible test (block (h)), LPW accurately locates bugs, identifies logic flaws in the code implementation, and further generates detailed refinement suggestions, as documented in the error analysis (block (j)). Then, the error analysis when integrated with the code explanation (block (g)) serves as feedback to refine the code, surpassing conventional scalar rewards or deduced error analysis (Chen et al., 2023b; Zhong et al., 2024; Shinn et al., 2023) and thereby improving refinement efficiency and accuracy.

We first evaluate LPW on four text-to-code generation benchmarks: HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and their extended test case variants, HumanEval-ET and MBPP-ET (Dong et al., 2023a). We conduct experiments on the proprietary LLM GPT-3.5 (Achiam et al., 2023), and open-source LLMs, Llama-3 (Meta, 2024) and Phi-3 (Abdin et al., 2024). The Pass@1 accuracy (Chen et al., 2021) is reported. Compared to the state-of-the-art LLM debugger, LDB (Zhong et al., 2024), LPW improves Pass@1 accuracy by 2% to 6.1% across all benchmarks with GPT-3.5 and achieves up to 16.4% improvement on MBPP with Llama-3. When evaluated on additional benchmarks using the advanced GPT-4o (OpenAI, 2024), LPW maintains its advantages, and achieves new state-of-the-art performance across evaluated benchmarks. Notably, on the contamination-free benchmark, LiveCode (Jain et al., 2024), and challenging benchmarks, APPS (Hendrycks et al., 2021) and CodeContests (Li et al., 2022), LPW improves Pass@1 accuracy by around 5%, 10%, and 5%, respectively, compared to LDB. We outline our contributions as follows:

- We introduce an end-to-end large language model programming workflow, LPW, which significantly improves the code generation accuracy over the state-of-the-art methods.
- We derive the intended solution for visible tests, represented as the plan verification. The plan verification clarifies logic specifications required to solve the visible tests for the given problem, thereby increasing the LLMs' confidence during both the initial program generation and subsequent debugging processes.
- We conduct extensive experiments across seven text-to-code generation benchmarks to validate the performance of LPW with various LLM backbones, provide a comprehensive analysis of their performance, token usage, and failure cases, and highlight the existing challenges.

2 Problem Formulation

We follow the problem formulation for text-to-code generation as outlined in Jiang et al. (2023), Chen et al. (2023b), and Zhong et al. (2024). The text-to-code generation problem is formulated as a triple $\mathcal{P} = \langle Q, T_v, T_h \rangle$, where Q represents the natural language problem specifications, and T_v and T_h are sets of visible and hidden tests, each containing input-output pairs $(t^i, t^o) \in T = T_v \cup T_h$. The goal is to leverage the LLM $\mathcal{M}$ to generate a program function f, $\mathcal{M} \rightarrow f$, that maps each input t^i to its output t^o for all pairs in T, i.e., $f(t^i) = t^o$, for $(t^i, t^o) \in T$. Visible tests T_v are accessible during both solution generation and code implementation phases, specifying inputs and expected outputs to guide functionality. In contrast, hidden tests T_h only become accessible for Pass@1 accuracy computation if the generated f passes T_v. In LPW, for all components shown in Figure 1, the problem description Q is, by default, concatenated with task-specific prompts to produce the desired response from LLMs.



[Problem Description] (1)
def maximum(arr, k):
"""Given an array arr of integers and a positive integer k, return a sorted list of length k with the maximum k numbers in arr."""

[Visible Tests] (2)
maximum([-3, -4, 5], 3) == [-4, -3, 5]

[Solution Plan] (3)
1. Sort the input array in descending order.
2. Return the first k elements of the sorted array in reverse order.

[Plan Verification for ...] (4)
1. Sort [-3, -4, 5] in descending order, which is [5, -3, -4].
2. Return the first 3 elements in reverse order, which are [-4, -3, 5].
[Results Compare]
The correct output is [-4, -3, 5]. The analysis output is [-4, -3, 5]. [-4, -3, 5] = [-4, -3, 5]. So the plan is correct.
[Correct Plan]

[Verification Check] (5)
- Sort ... is correct.
- Returning .. which are [-4, -3, 5] is correct.
- The comparison between the test output [-4, -3, 5] and the analysis output [-4, -3, 5] is correct.
[Correct Plan Verification]

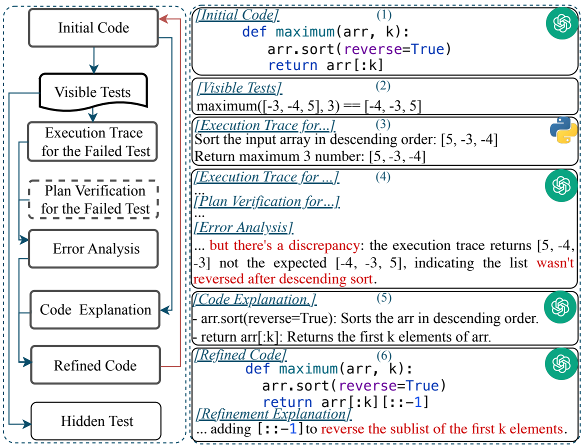
Figure 2: (a): An illustrated workflow of the solution generation phase in LPW. (b): Example message fragments corresponding to each workflow component for a HumanEval problem (*120th*) with the GPT-3.5 backbone. The detailed messages are available in Section 6.

3 Workflow Structure

In this section, we first detail the two phases of LPW separately and then elaborate on the iterative update strategies used in each phase.

Solution Generation. Figure 2 displays the overall workflow of the solution generation phase in LPW (part (a)), with an example programming problem for illustration (part (b)). LPW leverages the self-planning approach introduced by Jiang et al. (2023) to abstract and decompose the problem description Q into a strategic and adaptable plan Π at the start of the solution generation phase.

For a problem in HumanEval described by block (1) in Figure 2, its example solution plan is illustrated at block (3). However, the LLM-generated plan Π may occasionally be incorrect, misguiding subsequent program generation. To avoid this, LPW queries the LLM to verify Π against all visible tests T_v. The LLM-responded plan verification $\mathcal{A}(\Pi, T_v)$ delivers a step-by-step analysis, including all intermediate results and final derived outputs for all visible tests T_v based on Π. For each $t_v \in T_v$, its verification $\mathcal{A}(\Pi, \{t_v\})$ compares the derived output $t_v^{o'}$ with the ground-truth output t_v^o to assess the correctness of Π, as outlined at block (4). If Π is successfully verified on all visible tests, where in $\mathcal{A}(\Pi, T_v)$, $t_v^{o'} = t_v^o, \forall t_v \in T_v$, then the plan verification $\mathcal{A}(\Pi, T_v)$ is reviewed by the LLM again to ensure the accuracy of all intermediate results, since each intermediate step result is used in locating bugs and providing refinement suggestions when compared with the code runtime information



[Initial Code] (1)

```
def maximum(arr, k):
   arr.sort(reverse=True)
   return arr[:k]
```

[Visible Tests] (2)

maximum([-3, -4, 5], 3) == [-4, -3, 5]

[Execution Trace for...] (3)

Sort the input array in descending order: [5, -3, -4]
Return maximum 3 number: [5, -3, -4]

[Execution Trace for ...] (4)

...

[Plan Verification for...]

...

[Error Analysis]

... but there's a discrepancy: the execution trace returns [5, -4, -3] not the expected [-4, -3, 5], indicating the list wasn't reversed after descending sort.

[Code Explanation.] (5)

- arr.sort(reverse=True): Sorts the arr in descending order.
- return arr[:k]: Returns the first k elements of arr.

[Refined Code] (6)

```
def maximum(arr, k):
   arr.sort(reverse=True)
   return arr[:k][::-1]
```

[Refinement Explanation]

... adding `[::-1]` to reverse the sublist of the first k elements.

(a) (b)

Figure 3: (a): An illustrated workflow of the code implementation phase in LPW. (b): Example message fragments extending from Figure 2 and corresponding to each workflow component. See Section 6 for detailes.

on the failed visible test. If all intermediate outputs in $\mathcal{A}(\Pi, T_v)$ are validated as correct by the LLM as shown at block (5), $\mathcal{A}(\Pi, T_v)$ is treated as the intended solution for T_v. The plan Π and its verification $\mathcal{A}(\Pi, T_v)$ serve as the output of the solution generation phase, guiding code development and refinements in the code implementation phase.

Code Implementation. Figure 3 shows the overall workflow of the code implementation phase in LPW (part (a)), using the same problem from Figure 2 as an illustration (part (b)). LPW develops an initial program f by prompting the LLM with the problem description Q (block (1) in Figure 2), along with plan Π and its verification $\mathcal{A}(\Pi, T_v)$ from the solution generation phase. Subsequently, LPW queries the LLM to add *print statements* for each line in f, resulting in f_p, and then executes f_p on visible tests T_v. If f_p successfully solves T_v, LPW validates it on the hidden tests T_h to report Pass@1 accuracy. Otherwise, LPW collects the runtime information on the first failed visible test $\bar{t_v}$, indicating that the implementation in f deviates from the specifications in $\mathcal{A}(\Pi, \{\bar{t_v}\})$. Blocks (1)–(3) in part (b) of Figure 3 depict an initial program f (block (1)) that fails on a visible test $\bar{t_v}$ (block (2)) and its execution trace (block (3)) on $\bar{t_v}$ after adding print statements. We omit f_p from Figure 3 to keep the discussion concise. LPW instructs the LLM to conduct an error analysis by identifying inconsistencies between the intermediate outputs recorded in the execution trace of $\bar{t_v}$ and the expected intermediate outputs documented in the verification $\mathcal{A}(\Pi, \{\bar{t_v}\})$, analyzing causes, and offering refinement suggestions (block (4)). Subsequently, the error analysis and code explanation for f generated by the LLM (block (5)) are concatenated as the prompt to generate the refined program f' (block (6)). The code explanation helps the LLM align the text-based error analysis with the code implementation. LPW replaces f with the refined program f' and revalidates the updated f against the visible tests T_v to assess the need for further refinements.

Iterative Updates. LPW includes two update steps in the solution generation phase to enable *self-correction* as indicated by the red arrows in Figure 2: 1) when the plan verification inferred final output differs from the ground-truth output for a visible test, where $t_v^{o'} \neq t_v^o, \exists t_v \in T_v$ in $\mathcal{A}(\Pi, T_v)$, a revised solution plan Π' is included in the LLM response to substitute the original plan; 2) when the LLM detects any incorrect intermediate values in $\mathcal{A}(\Pi, T_v)$ e.g., contextual inconsistencies, mathematical miscalculations, or logical flaws, LPW prompts the LLM to regenerate the plan verification. These update methods ensure that the solution plan Π and its verification $\mathcal{A}(\Pi, T_v)$ maintain the necessary precision, as well-formed Π and $\mathcal{A}(\Pi, T_v)$ are essential for accurate code generation (Jiang et al., 2023). In the code implementation phase, the code refinement process acts as an update mechanism, replacing the program f with the refined program f' when f fails the visible test T_v as highlighted by the red arrow in Figure 3. Overall, for a problem $\mathcal{P}$, LPW iteratively revises the generated plan Π and its verification $\mathcal{A}(\Pi, T_v)$, in the solution generation phase, until $\mathcal{A}(\Pi, T_v)$ infers correct outputs for all visible tests T_v and no error intermediate outputs are present in $\mathcal{A}(\Pi, T_v)$. Otherwise, LPW reports a failure for $\mathcal{P}$ when reaching the maximum iterations. Similarly, in the code implementation phase, LPW iteratively refines the generated program f if bugs exist. This process continues until a refined f successfully solves T_v, followed by Pass@1 accuracy calculation on hidden tests T_h, or LPW reports a failure for $\mathcal{P}$ upon reaching the maximum iteration limit.

4 Experiments

Benchmarks. We first evaluate LPW on the well-established text-to-code benchmarks HumanEval, MBPP, HumanEval-ET, and MBPP-ET, where the given context outlines the intended functionality of the program to be synthesized. HumanEval-ET and

		HumanEval		HumanEval-ET		MBPP		MBPP-ET	
		Acc ↑	Δ ↑	Acc ↑	Δ ↑	Acc ↑	Δ ↑	Acc ↑	Δ ↑
GPT-3.5	Baseline	74.4 ±0.8	–	66.5 ±1.3	–	67.4 ±0.5	–	52.8 ±0.3	–
	SP	77.4 ±0.8	3.0	69.5 ±0.8	3.0	69.2 ±0.4	1.8	52.4 ±0.2	-0.4
	MapCoder	77.4 ±1.0	3.0	66.5 ±1.0	0.0	72.0 ±0.5	4.6	56.6 ±0.5	3.8
	SD	81.1 ±1.0	6.7	72.0 ±1.0	5.5	71.2 ±0.3	3.8	56.0 ±0.1	3.2
	LDB	82.9 ±1.0	8.5	72.6 ±1.0	6.1	72.4 ±0.3	5.0	55.6 ±0.2	2.8
	LPW (ours)	**89.0** ±0.8	**14.6**	**77.4** ±0.8	**10.9**	**76.0** ±0.2	**8.6**	**57.6** ±0.1	**4.8**
Llama-3	Baseline	73.2 ±1.3	–	61.0 ±1.0	–	44.0 ±1.2	–	35.4 ±1.0	–
	SP	78.0 ±2.0	4.8	65.2 ±1.0	4.2	48.6 ±1.4	4.6	38.4 ±1.4	3.0
	MapCoder	83.5 ±1.3	10.3	73.8 ±0.8	12.8	71.4 ±1.0	27.4	55.6 ±1.0	20.2
	SD	81.7 ±1.3	8.5	68.3 ±0.8	7.3	63.6 ±1.2	19.6	50.0 ±1.3	14.6
	LDB	84.1 ±1.7	10.9	72.0 ±0.8	11.0	57.2 ±1.6	13.2	44.8 ±1.4	9.4
	LPW (ours)	**88.4** ±1.6	**15.2**	**76.2** ±1.3	**15.2**	**73.6** ±1.3	**29.6**	**56.4** ±1.2	**21.0**
Phi-3	Baseline	36.0 ±1.0	–	32.3 ±1.0	–	39.0 ±1.3	–	33.2 ±1.4	–
	SP	40.8 ±1.4	4.8	34.8 ±0.9	2.5	46.4 ±1.4	7.4	37.6 ±1.4	4.4
	MapCoder	–	–	–	–	–	–	–	–
	SD	51.2 ±1.2	15.2	45.7 ±1.0	13.4	45.8 ±1.2	6.8	36.6 ±1.2	3.4
	LDB	65.9 ±1.6	29.9	54.9 ±0.9	22.6	52.4 ±1.6	13.4	42.8 ±1.4	9.6
	LPW (ours)	**76.8** ±1.3	**40.8**	**62.8** ±1.3	**30.5**	**64.0** ±1.2	**25.0**	**48.4** ±1.2	**15.2**

Table 1: Comparisons of Baseline, SP, MapCoder, SD, LDB, and LPW in terms of Pass@1 accuracy (Acc) and improvement (Δ) with respect to Baseline across benchmarks HumanEval, HumanEval-ET, MBPP, and MBPP-ET with LLMs GPT-3.5, Llama-3, and Phi-3. Acc and Δ are measured in percentages. Best results are highlighted in red. The standard deviation (±) is calculated based on three runs and applies to other experiments when reported.

MBPP-ET introduce approximately 100 additional hidden tests, covering numerous edge cases, for each problem in HumanEval and MBPP, thus being regarded as more reliable benchmarks (Dong et al., 2023a). In HumanEval and HumanEval-ET, we treat the test cases described in the task description as visible tests, typically 2-5 per task. For MBPP, we consider its test set that contains 500 problems with 3 hidden tests per problem. We set the first hidden test as the visible test and treat the other two as hidden, consistent with studies (Chen et al., 2023b; Zhong et al., 2024; Ni et al., 2023; Shi et al., 2022). MBPP-ET uses the same set of problems and visible tests for each problem as MBPP.

Experimental Setup. We compare LPW with the representative code generation approaches *Self-Planning* (SP) (Jiang et al., 2023), *MapCoder* (Islam et al., 2024), *Self-Debugging (+Expl)* (SD) (Chen et al., 2023b), and *Large Language Model Debugger* (LDB) (Zhong et al., 2024). SP relies solely on the LLM-generated solution plan to produce the program solution in a single effort without refinements. MapCoder, a multi-agent collaborative approach, generates multiple unverified plans and traverses them to produce and refine code based on the current plan. SD uses a *rubber duck debugging* approach in LLMs, where LLMs are prompted to provide explanations of generated programs as feedback for debugging. LDB, a state-of-the-art LLM debugger, segments generated programs into blocks based on the control flow graph, which facilitates bug detection and the refinement of each program block using runtime information in LLMs. A detailed comparison of different methods is summarized in Appendix B.

We generate a seed program for each problem with the same prompts and parameters introduced by Chen et al. (2023b) for SD and LDB and label the performance of seed programs as Baseline. We note that SD and LDB only perform refinements on the seed program that fails the visible tests. We experiment with various LLMs with different parameter sizes, including GPT-3.5 (turbo-0125, ≥175B), Llama-3 (70B-Instruct), and Phi-3 (14B-Instruct) to evaluate performance and demonstrate that LPW is model-independent. The Pass@1 accuracy is reported. We apply 2-shot prompting in LPW, with a maximum of 12 iterations for both the solution generation and code implementation phases. Similarly, we set the maximum debugging iterations to 12 for SD and LDB. MapCoder generates 3 plans, each with up to 4 refinement iterations. All following experiments adhere to these iteration settings. Appendix C provides additional implementation details. An empirical discussion on the number of iterations for LPW is available in Appendix D.

Results on Various LLMs. Table 1 presents the Pass@1 accuracy for evaluated approaches, along with their respective improvements over Baseline. LPW outperforms all competing methods across all benchmarks and with various LLM backbones. Compared to LDB, LPW improves Pass@1 accu-

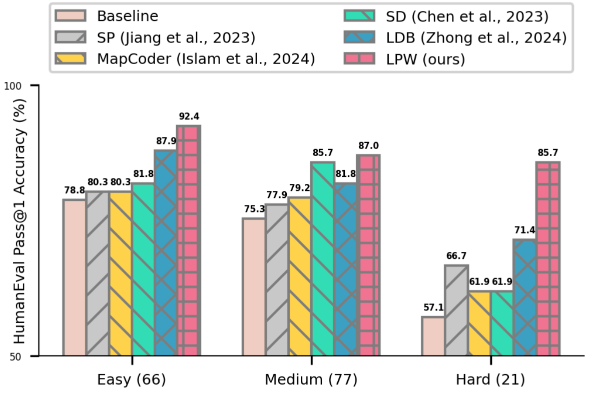


Figure 4: Pass@1 accuracy of Baseline, SP, MapCoder, SD, LDB, and LPW across different difficulty levels, *Easy*, *Medium*, and *Hard* on the HumanEval benchmark when leveraging GPT-3.5 as the LLM backbone.

racy by 6.1%, 4.8% 3.6%, and 2%, on HumanEval, HumanEval-ET, MBPP, and MBPP-ET, respectively, with GPT-3.5 and achieves up to 16.4% improvement on MBPP with Llama-3. These results showcase the effectiveness of the proposed workflow and demonstrate the model-independent benefits of LPW. MapCoder fails on Phi-3 as it requires strict XML-formatted responses, which pose a challenge for Phi-3.

Figure 4 compares the Pass@1 accuracy of competing methods across different difficulty levels, *Easy*, *Medium*, and *Hard*, on the HumanEval benchmark using GPT-3.5. We apply the method described by Zhong et al. (2024) to generate the difficulty annotations in Figure 4 by querying GPT-3.5 with problem descriptions and canonical solutions. LPW displays convincing performance, exceeding 85% accuracy across all difficulty levels. For the *Hard* level, LPW achieves 85.7% accuracy, while competing approaches show a notable degradation, falling below 70%. The failure analysis for LPW with GPT-3.5 is available in Appendix E.

Results on Advanced LLM. To further demonstrate the effectiveness of LPW, we evaluate its performance on the same benchmarks presented in Table 1, along with the contamination-free benchmark, LiveCode, and two competitive benchmarks, APPS and CodeContests, using the advanced LLM GPT-4o as the backbone. LDB is compared due to its second-highest performance in Table 1 using GPT-3.5. We include the experimental results for MapCoder in Appendix F, given its comparatively lower performance. We sample 140 problems from LiveCode, published between November 2023 and September 2024, postdating GPT-4o's cutoff date. For APPS and CodeContests, we use subsets of 139 and 150 problems, respectively. The experiment results are shown in Table 2. Similarly, the performance of the seed programs for LDB is referred to as Baseline. LPW outperforms Baseline and LDB across all benchmarks, and establishes new state-of-the-art Pass@1 accuracy, notably achieving 98.2% on HumanEval. GPT-4o exhibits reduced performance on LiveCode, while LPW reliably outperforms LDB by 5% accuracy. For APPS and CodeContests, LPW surpasses LDB by around 10% and 5% accuracy, highlighting the advantages of LPW in tackling challenging benchmarks. Incorporating the plan verification allows LPW to clarify issues before code generation and efficiently correct bugs overlooked by LLMs. In contrast, LDB shows a negligible improvement of only 0.6% and 1.3% compared to Baseline on HumanEval and CodeContests, underscoring the limitations of debugging with coarse feedback.

LPW fails on only 3 problems in HumanEval with the GPT-4o backbone. We further investigate the reasons behind LPW's failures on the *91st*, *132nd*, and *145th* problems as shown in Figures 15, 16, and 17 in the Appendix, and attempt to generate the correct program solutions. The *91st* problem fails since GPT-4o misinterprets the linguistic distinction between the word and the letter; the *132nd* problem's ambiguous description challenges GPT-4o; and the incomplete description of the *145th* problem leads to failed plan verifications. LPW successfully generates correct program solutions for 2 out of 3 problems, achieving 99.4% Pass@1 accuracy, by incorporating an additional visible test to validate the intended solution for the *91st* problem and providing a comprehensive problem description for the *145th* problem. See Appendix G.1 for a detailed discussion.

Figure 5 compares the Pass@1 accuracy of Baseline, LDB, and LPW across varying difficulty levels, *Easy*, *Medium*, and *Hard* on the LiveCode benchmark using GPT-4o. LPW achieves the highest Pass@1 accuracy across all levels, surpassing LDB by over 5% accuracy in each level. Compared to Baseline, LPW delivers over 15% higher accuracy at the *Easy* and *Medium* levels and 7.5% higher at the *Hard* level. Conversely, LDB performs similarly to Baseline at the *Hard* level, underscoring its limited refinement capability in addressing more complex tasks. The failure analysis for LPW with GPT-4o is available in Appendix G.

Learning from Test. We further investigate the

		HumanEval	HumanEval-ET	MBPP	MBPP-ET	LiveCode	APPS	CodeContests
GPT-4o	Baseline	91.5 ±0.3	81.7 ±0.3	78.4 ±0.4	62.6 ±0.2	45.7 ±0.6	41.7±0.9	28.0 ±0.5
	LDB	92.1 ±0.0	81.7 ±0.0	82.4 ±0.3	65.4 ±0.0	54.3 ±0.3	53.2 ±0.3	29.3 ±0.3
	LPW (ours)	**98.2** ±0.3	**84.8** ±0.3	**84.8** ±0.2	**65.8** ±0.1	**59.3** ±0.6	**62.6** ±0.3	**34.7** ±0.3

Table 2: Pass@1 accuracy, in percentages, for Baseline, LDB, and LPW on HumanEval, HumanEval-ET, MBPP, MBPP-ET, LiveCode, APPS and CodeContests benchmarks when using GPT-4o (2024-05-13) as the backbone.

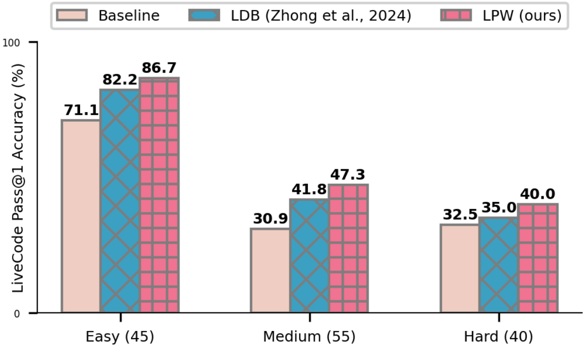


Figure 5: Pass@1 accuracy of Baseline, LDB, and LPW across different difficulty levels, *Easy*, *Medium*, and *Hard*, on the LiveCode benchmark when using GPT-4o as the LLM backbone.

		MBPP-ET ↑	MBPP-ET-3 ↑	Δ ↑
GPT-3.5	MapCoder	56.6 ±0.5	60.6 ±0.2	4.0
	SD	56.0 ±0.1	59.2 ±0.3	3.2
	LDB	55.6 ±0.2	57.6 ±0.2	2.0
	LPW (ours)	**57.6** ±0.1	**62.0** ±0.2	**4.4**

Table 3: The impact on Pass@1 accuracy with additional visible tests using the GPT-3.5 backbone. MBPP-ET-3 includes two more visible tests per problem than MBPP-ET. Δ represents the accuracy improvement on MBPP-ET-3 over MBPP-ET. Pass@1 accuracy and Δ are measured as percentages.

impact of the number of visible tests on MapCoder, SD, LDB, and LPW that use visible tests to refine code. We propose a variant of MBPP-ET, denoted as MBPP-ET-3. In MBPP-ET-3, each problem's visible tests are the three hidden tests from MBPP, while the hidden tests are the extended test cases introduced in MBPP-ET. In other words, each problem in MBPP-ET-3 contains two more visible tests than in MBPP-ET. Results in Table 3 show that LPW achieves the highest Pass@1 accuracy of 62.0% on MBPP-ET-3 and the largest improvement of 4.4% over MBPP-ET when using GPT-3.5. LPW produces the step-by-step solution for each visible test to clarify code development logic and inform subsequent refinements, demonstrating superior efficiency in utilizing visible tests to improve performance among evaluated methods.

Performance Analysis. Figure 6 evaluates the Pass@1 accuracy of LPW when considering different numbers of code implementation iterations on the HumanEval benchmark with GPT-3.5. For SD and LDB, we allocate the same number of debugging iterations. We omit MapCoder due to its distinct refinement strategy. We note that all evaluated approaches start from iteration 0, representing the Pass@1 accuracy before debugging. In Figure 6, Baseline and SP are plotted as straight lines with 74.4% and 77.4% accuracy, respectively, due to no debugging involved. Baseline and SP serve as the control group to illustrate when debugging methods surpass no-debugging methods. SD and LDB refine incorrect programs in Baseline, surpassing SP after two iterations. LPW starts debugging from an initial 79.9% accuracy, higher than the 77.4% for SP, underscoring the importance of plan verification in initial code generation. LPW surpasses the best performance of SD and LDB after only one iteration, demonstrating its efficient code refinement strategy. LPW gradually refines the code and reaches the highest accuracy by the 10th iteration.

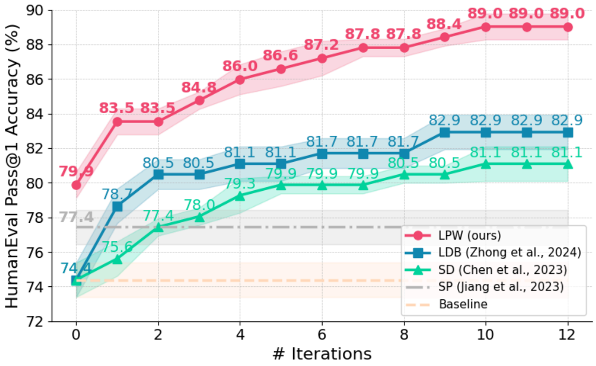


Figure 6: The impact on Pass@1 accuracy with the increased number of code implementation iterations/debugging iterations on the HumanEval benchmark when leveraging GPT-3.5 as the LLM backbone. The shaded area represents the standard deviation.

Ablation Study. Table 4 shows the Pass@1 accuracy of different variants of LPW on the HumanEval and MBPP benchmarks with GPT-3.5. The suffix -V denotes the exclusion of plan verification in both solution generation and code imple-

		HumanEval		MBPP	
		Acc	Δ	Acc	Δ
GPT-3.5	LPW	89.0 ±0.8	–	76.0 ±0.2	–
	LPW-V	86.0 ±0.5	-3.0	73.2 ±0.2	-2.8
	LPW-S	86.0 ±1.0	-3.0	73.0 ±0.3	-3.0
	LPW-C	79.9 ±0.8	-9.1	72.2 ±0.5	-3.8

Table 4: Pass@1 accuracy (Acc) for different variants of LPW with GPT-3.5. Δ denotes the decrease against LPW. Acc and Δ are measured in percentages. See Appendix H for additional ablation study.

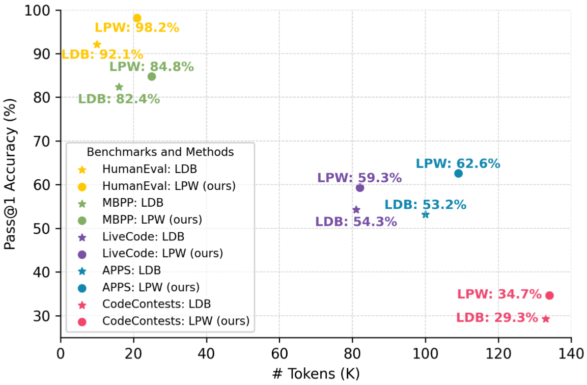


Figure 7: Pass@1 accuracy vs. average token cost per program for LDB and LPW on different benchmarks using GPT-4o as the LLM backbone. K is 10^3. The standard deviation is too small to be visible.

mentation phases; -S stands for the LPW variant that excludes the solution generation phase; while -C represents the removal of the code implementation phase, specifically omitting code refinements. For each problem, LPW-V generates the initial program based on the unverified plan and repairs the program using only code explanation and runtime information. LPW-S refines the seed program from Baseline that fails visible tests, leveraging only code explanation and runtime information. LPW-C generates the program solution based on the plan and its verification without refinements.

The performance decline of LPW-V highlights the significance of plan verification, which serves as the intended solution for visible tests, improving the performance of LLMs in both initial code generation and subsequent refinements. LPW-V considers the unverified plan when drafting initial programs. However, the effect of the unverified plan is limited, as LPW-V shows only slight improvement on MBPP compared to LPW-S, which excludes both the plan and plan verification. This aligns with the results in Table 1, where Self-Planning shows minimal improvement compared to Baseline. The results of LPW-S and LPW-C show that removing either phase in LPW decreases its performance, indicating that both solution generation and code implementation phases are crucial for optimal performance. See Appendix I and J for a discussion on the accuracy of LLM-generated plans, plan verifications and refined programs in LPW.

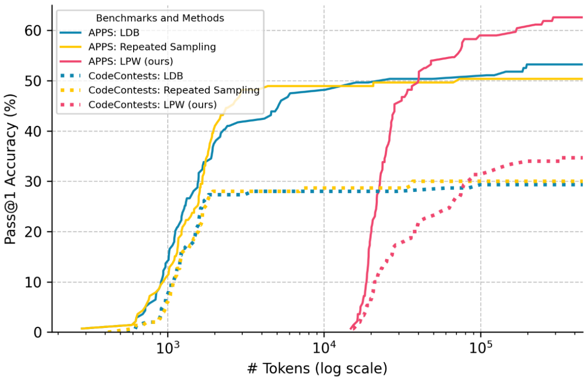
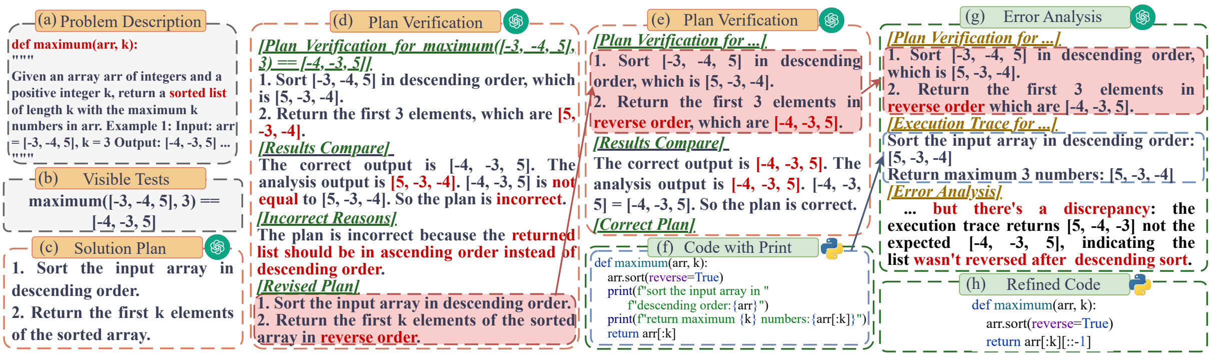


Figure 8: Pass@1 accuracy as a function of token consumption for LDB, Repeated Sampling, and LPW on the APPS and CodeContests benchmarks with GPT-4o. The standard deviation is omitted for better illustration. The same illustration for the LiveCode benchmark is available in Appendix Figure 10.

5 Cost-Performance Analysis

Figure 7 compares Pass@1 accuracy against the average token cost per program for LDB and LPW across five benchmarks using GPT-4o. When analyzing the cost for LDB, we include the tokens used to generate the seed programs, which account for about 2% of its total token consumption. LDB consumes fewer tokens per problem but achieves lower accuracy. When measured by the accuracy-cost ratio, computed as Pass@1 accuracy divided by the total tokens used, LDB shows better ratios on HumanEval and MBPP benchmarks. On LiveCode, APPS, and CodeContests benchmarks, LDB and LPW exhibit similar token usage per problem, while LPW displays notably higher accuracy. As a result, LPW realizes higher accuracy-cost ratios of 0.60% on LiveCode, 0.43% on APPS, and 0.18% on CodeContests per 1000 tokens, compared to LDB, which reports 0.50% on LiveCode, 0.39% on APPS and 0.14% on CodeContests per 1000 tokens. The lower ratios of LDB arise from insufficient refinements, where multiple ineffective debugging iterations consume significant token resources, yet the generated program remains flawed.

Figure 8 illustrates the variation in Pass@1 accuracy with token consumption across different approaches on the APPS and CodeContests bench-



```
def maximum(arr, k):
"""
Given an array arr of integers and a positive integer k, return a sorted list of length k with the maximum k numbers in arr. Example 1: Input: arr = [-3, -4, 5], k = 3 Output: [-4, -3, 5] ...
"""
```

maximum([-3, -4, 5], 3) == [-4, -3, 5]

1. Sort the input array in descending order.
2. Return the first k elements of the sorted array.

[Plan Verification for maximum([-3, -4, 5], 3) == [-4, -3, 5]].
1. Sort [-3, -4, 5] in descending order, which is [5, -3, -4].
2. Return the first 3 elements, which are [5, -3, -4].

[Results Compare]
The correct output is [-4, -3, 5]. The analysis output is [5, -3, -4]. [-4, -3, 5] is not equal to [5, -3, -4]. So the plan is incorrect.

[Incorrect Reasons].
The plan is incorrect because the returned list should be in ascending order instead of descending order.

[Revised Plan]
1. Sort the input array in descending order.
2. Return the first k elements of the sorted array in reverse order.

[Plan Verification for ...].
1. Sort [-3, -4, 5] in descending order, which is [5, -3, -4].
2. Return the first 3 elements in reverse order, which are [-4, -3, 5].

[Results Compare]
The correct output is [-4, -3, 5]. The analysis output is [-4, -3, 5]. [-4, -3, 5] = [-4, -3, 5]. So the plan is correct.

[Correct Plan].

```
def maximum(arr, k):
    arr.sort(reverse=True)
    print(f"sort the input array in "
        f"descending order:{arr}")
    print(f"return maximum {k} numbers:{arr[:k]}")
    return arr[:k]
```

[Plan Verification for ...].
1. Sort [-3, -4, 5] in descending order, which is [5, -3, -4].
2. Return the first 3 elements in reverse order which are [-4, -3, 5].

[Execution Trace for ...].
Sort the input array in descending order: [5, -3, -4]
Return maximum 3 numbers: [5, -3, -4]

[Error Analysis].
... but there's a discrepancy: the execution trace returns [5, -4, -3] not the expected [-4, -3, 5], indicating the list wasn't reversed after descending sort.

```
def maximum(arr, k):
    arr.sort(reverse=True)
    return arr[:k][::-1]
```

Figure 9: A case study of LPW on the *120th* problem in HumanEval, extending from Figures 2 and 3, using GPT-3.5. We omit certain components in Figures 2 and 3, e.g., the plan verification check and the initial code, and present incomplete prompts and responses to save space.

marks using GPT-4o. We introduce Repeated Sampling as an enhanced Baseline. For each problem, it repeatedly samples program solutions from the LLM until either the token consumption exceeds that of LPW, or a solution passes all visible tests and is validated on hidden tests. Repeated Sampling and LDB initially improve accuracy with increased tokens but show negligible improvement after around $10^{3.8}$ tokens on the APPS benchmark and $10^{3.3}$ tokens on the CodeContests benchmark. In contrast, LPW starts with high token consumption for initial plan and verification generation, resulting in a sharp accuracy increase that quickly surpasses Repeated Sampling and LDB after around $10^{4.6}$ tokens on the APPS benchmark and $10^{4.8}$ tokens on the CodeContests benchmark. Repeated Sampling is allocated the same token budget as LPW, while its final accuracy remains lower than LPW on both benchmarks, highlighting the benefits of plan and plan verification in generating high-quality initial code and subsequent refinements. Challenging benchmarks align with LPW usage scenarios, where the precise natural language solution described in the plan and plan verification is essential for logical consistency and understanding non-trivial bugs in the program, particularly when problems involve complex logical reasoning steps.

6 Case Study

Figure 9 illustrates example message fragments from LPW in the *120th* problem of HumanEval using the GPT-3.5 backbone. LPW successfully generates the correct program, while all other approaches fail. This problem requires returning a sorted array with the maximum k numbers. However, in the problem description (block (a)), the unspecified order in the output array introduces ambiguity, confusing other methods. LPW struggles at the initial solution plan (block (c)), while the issue is addressed in the *[Revised Plan]*, during plan verification (block (d)). The visible test (block (b)) delineates the reverse order in the return array after sorting in descending order. The initial code with print statements (block (f)) fails on the visible test since the array is not reversed. Subsequently, its execution trace is compared with the plan verification (block (e)) to identify this bug, as described in the *[Error Analysis]* in block (g). The refined code, which first sorts the array in descending order and then reverses the first k elements into ascending order, successfully solves this problem.

7 Conclusion

We introduce LPW, a large language model programming workflow, designed for text-to-code generation tasks. LPW effectively integrates various advanced code generation techniques within a two-phase development model. A key innovation of LPW is the incorporation of plan verification, which enables LLMs to accurately draft an initial program and effectively correct bugs. We evaluate LPW on well-established text-to-code generation benchmarks across various LLMs. LPW significantly improves code generation accuracy compared to other existing approaches and achieves new state-of-the-art Pass@1 accuracy, with 98.2% on HumanEval, 84.8% on MBPP, 59.3% on LiveCode, 62.6% on APPS, and 34.7% on CodeContests benchmarks using GPT-4o as the backbone. In the future, additional visible tests automatically generated by LLMs (Chen et al., 2023a) can be explored to improve the performance of LPW.

8 Limitations

Similar to other code generation approaches, LPW is constrained by the imperfect reasoning capacity of LLMs. Strengthening the reasoning capacity of LLMs remains an ongoing challenge. While the plan and plan verification have proven valuable across different benchmarks, they require substantial token consumption. In the future, reducing this consumption remains a critical area for improvement. Besides, although the LLM-generated plan and plan verification demonstrate promising accuracy on current tasks, the accuracy of the generated code still lags behind that of the plan and plan verification (Appendix I). Incorporating alternative solution representations (Zelikman et al., 2023) alongside natural language representations may assist LLMs in translating text-based solutions into program solutions more accurately.

9 Ethic Statements

The datasets used in this work are publicly accessible. LPW operates on either open-source LLMs or proprietary LLMs through API access. LPW prioritizes computational efficiency through an optimized workflow. This approach underscores our commitment to responsible and sustainable research practices.

Acknowledgements

Chao Lei is supported by the Melbourne Research Scholarship established by the University of Melbourne. This research was conducted using the University of Melbourne's high-performance computing (HPC) system. We sincerely thank Yubo Wang from the University of Melbourne for assistance with data analysis.

References

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. *Anthropic AI Hub*. Accessed: 2024-07-18.

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*.

Kent Beck. 2022. *Test driven development: By example*. Addison-Wesley Professional.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris 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 *Proceedings of the 34th Advances in Neural Information Processing Systems*, NeurIPS, pages 1877–1901.

Angelica Chen, Jérémy Scheurer, Jon Ander Campos, Tomasz Korbak, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. 2024. Learning from natural language feedback. *Transactions on Machine Learning Research*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023a. Codet: Code generation with generated tests. In *Proceedings of the 11th International Conference on Learning Representations*, ICLR, pages 1–19.

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. 2023b. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. 2023. Chain-of-verification reduces hallucination in large language models. *arXiv preprint arXiv:2309.11495*.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2023a. Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023b. Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages*, 4:1–119.

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. In *Proceedings of the 35th Advances in Neural Information Processing Systems*, NeurIPS.

Samuel Holt, Max Ruiz Luyten, and Mihaela van der Schaar. 2024. L2MAC: Large language model automatic computer for extensive code generation. In *Proceedings of the 12th International Conference on Learning Representations*, ICLR, pages 1–61.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023a. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023b. An empirical study on fine-tuning large language models of code for automated program repair. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 1162–1174.

Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023c. A survey on automated program repair techniques. *arXiv preprint arXiv:2303.18184*.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, ACL, pages 4912–4944.

Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language models for code completion: A practical evaluation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE, pages 1–13.

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

Xue Jiang, Yihong Dong, Lecheng Wang, Fang Zheng, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2023. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, pages 1–28.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Proceedings of the 36th Advances in neural information processing systems*, NeurIPS, pages 22199–22213.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Proceedings of the 36th Advances in Neural Information Processing Systems*, NeurIPS, pages 21314–21328.

Chao Lei, Nir Lipovetzky, and Krista A. Ehinger. 2023. Novelty and lifted helpful actions in generalized planning. In *Proceedings of the 16th International Symposium on Combinatorial Search*, SoCS, pages 148–152.

Chao Lei, Nir Lipovetzky, and Krista A Ehinger. 2024. Generalized planning for the abstraction and reasoning corpus. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence*, AAAI, pages 20168–20175.

Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro Von Werra, and Harm de Vries. 2023. Starcoder: May the source be with you! *Transactions on Machine Learning Research*.

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*, 378:1092–1097.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2024. Let's verify step by step. In *Proceedings of the 12th International Conference on Learning Representations*, ICLR, pages 1–24.

Feng Lin, Dong Jae Kim, et al. 2024. When llm-based code generation meets the software development process. *arXiv preprint arXiv:2403.15852*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. In *Proceedings of the 37th Advances in Neural Information Processing Systems*, NeurIPS, pages 46534–46594.

AI Meta. 2024. Introducing meta llama 3: The most capable openly available llm to date. *Meta AI.* Accessed: 2024-07-18.

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. *arXiv preprint arXiv:2308.07124*.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, ICML, pages 26106–26128.

OpenAI. 2024. Hello gpt-4o. *OpenAI.* Accessed: 2024-07-18.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE, pages 1–13.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, ACL, pages 15174–15186.

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*.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093*.

Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. 2024. Generalized planning as heuristic search: A new planning search-space that leverages pointers over objects. *Artificial Intelligence*, 330:104097.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, EMNLP, pages 3533–3546.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Proceedings of the 37th Advances in Neural Information Processing Systems*, NeurIPS, pages 8634–8652.

Zhi Rui Tam, Cheng-Kuang Wu, Yi-Lin Tsai, Chieh-Yen Lin, Hung-yi Lee, and Yun-Nung Chen. 2024. Let me speak freely? a study on the impact of format restrictions on performance of large language models. *arXiv preprint arXiv:2408.02442*.

Zilu Tang, Mayank Agarwal, Alexander Shypula, Bailin Wang, Derry Wijaya, Jie Chen, and Yoon Kim. 2023. Explain-then-translate: an analysis on improving program translation with self-generated explanations. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1741–1788.

Zhao Tian and Junjie Chen. 2023. Test-case-driven programming understanding in large language models for better code generation. *arXiv preprint arXiv:2309.16120*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th Advances in Neural Information Processing Systems*, NeurIPS, pages 24824–24837.

Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A Smith, Mari Ostendorf, and Hannaneh Hajishirzi. 2023. Fine-grained human feedback gives better rewards for language model training. In *Proceedings of the 37th Advances in Neural Information Processing Systems*, NeurIPS, pages 1–26.

Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. In *Proceedings of the 2024 ACM International Conference on the Foundations of Software Engineering*, FSE, pages 1–23.

Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *Proceedings of the 38th International conference on machine learning*, ICML, pages 11941–11952.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. 2023. Parsel: Algorithmic reasoning with language models by composing decompositions. In *Proceedings of the 37th Advances in Neural Information Processing Systems*, NeurIPS, pages 31466–31523.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, and Ed H. Chi. 2023. Least-to-most prompting enables complex reasoning in large language models. In *Proceedings of the 11th International Conference on Learning Representations*, ICLR, pages 1–61.

Appendix

A Related Work

A.1 Program Synthesis

Program synthesis remains an open challenge of generating a program within a target domain-specific language (DSL) from given specifications. One prevalent approach involves searching the large space of possible programs. For example, generalized planning whose solution is formalized as a *planning program* with *pointers* (Segovia-Aguas et al., 2024; Lei et al., 2023) has demonstrated promising results in synthesizing program solutions for abstract visual reasoning tasks (Lei et al., 2024) when the DSL is carefully designed. However, hand-crafted DSLs often suffer from limited generalization capacity, and the huge search space diminishes its effectiveness. Recently, LLMs trained on vast corpora have excelled in natural language processing (NLP) tasks and have been extended to code generation e.g., GPT-series (Achiam et al., 2023; OpenAI, 2024), Llama-series (Meta, 2024; Roziere et al., 2023; Touvron et al., 2023), and Claude-series (Anthropic, 2024). LPW leverages the strengths of LLMs in NLP tasks to generate intended solutions in natural language. These text-based solutions demonstrate high-quality logical reasoning steps and satisfactory accuracy, thereby effectively aiding subsequent code generation.

A.2 Prompting Techniques

To imitate the logical chain in human brain when tackling reasoning tasks, prompting methods direct LLMs to decompose problems into solvable sub-problems (Jiang et al., 2023; Zhou et al., 2023; Lightman et al., 2024; Dhuliawala et al., 2023) and progressively infer the correct answer with intermediate outputs, as exemplified by chain-of-thought prompting (Wei et al., 2022; Kojima et al., 2022). Inspired by these studies, LPW decomposes a text-to-code problem into several sub-problems described by the solution plan and follows the chain-of-thought prompting idea to verify the solution plan against visible tests with step-by-step analysis. The generated plan and its verification provide step-by-step natural language instructions for code generation, supporting LLMs in both the initial code development and subsequent refinements.

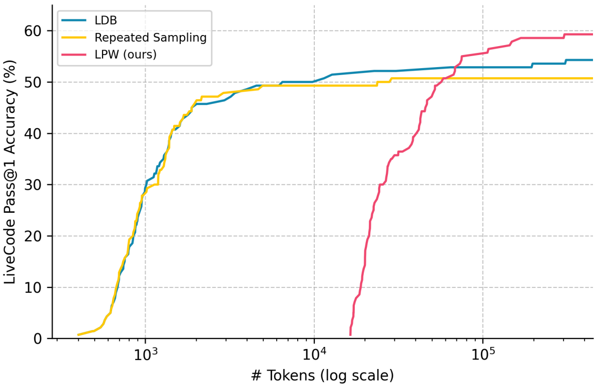


Figure 10: Pass@1 accuracy as a function of token consumption for LDB, Repeated Sampling, and LPW on the LiveCode benchmark with GPT-4o.

A.3 Code Refinement

Accurate program solutions often require iterative refinements due to model limitations (Zhong et al., 2024; Chen et al., 2023b; Shinn et al., 2023). Various interactive approaches have been proposed to optimize debugging performance in LLMs, such as human feedback (Chen et al., 2024; Le et al., 2022; Wu et al., 2023), trained models (Huang et al., 2023b; Le et al., 2022; Yasunaga and Liang, 2021), LLM-generated explanations (Chen et al., 2023b; Madaan et al., 2023; Shinn et al., 2023; Tang et al., 2023), execution results (Zhong et al., 2024; Holt et al., 2024; Tian and Chen, 2023), and multi-agent collaborations (Islam et al., 2024; Huang et al., 2023a; Qian et al., 2024; Dong et al., 2023b). Current state-of-the-art LLM debuggers, such as Self-Debugging and LDB, repair various seed programs to create program solutions. However, they encounter difficulties when the initial code substantially deviates from the original intent. Besides, without safeguarding, the refined code frequently diverges from the problem specifications. In contrast, LPW develops initial code that adheres to the validated intended solution through plan verification, minimizing deviations from the problem description. The plan verification further guides the code refinement, ensuring alignment with the problem specifications.

B Feature Study

Table 5 compares the technical features of LPW with other competing approaches. Different from existing methods, LPW involves plan verification as the intended solution throughout both initial program development and subsequent refinements. Compared to the state-of-the-art method LDB,

	Analysis Before Coding		Coding With Debugging		
	Plan	Plan Verification	Code Explanation	Runtime Information	Plan Verification
SP	✓	✗	✗	✗	✗
MapCoder	✓	✗	✗	✓	✗
SD	✗	✗	✓	✗	✗
LDB	✗	✗	✓	✓	✗
LPW (ours)	✓	✓	✓	✓	✓

Table 5: Features of Self-Planning (SP), MapCoder, Self-Debugging (+Expl) (SD), LDB, and LPW.

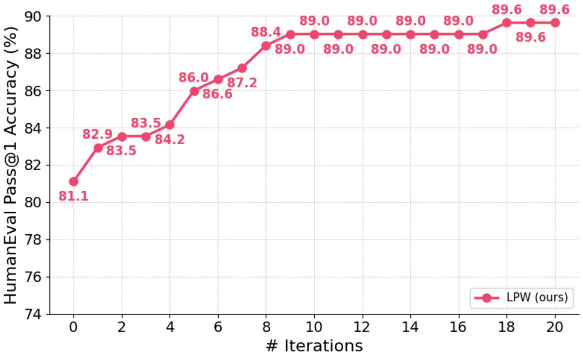


Figure 11: Pass@1 accuracy of LPW on the HumanEval benchmark using GPT-3.5 with 20 iterations in both the solution generation and code implementation phases.

LPW achieves higher initial program generation accuracy by integrating plan verification, which outlines detailed logic for solving each visible test in the given problem. Additionally, LPW refines erroneous code by comparing the execution trace with plan verification on the failed visible test, identifying discrepancies, and modifying the specific program lines responsible for differences. In contrast, LDB relies on LLM inference for code block correctness, which is less reliable without benchmark answers for each block. Besides, LPW introduces features absent in LDB, such as a dedicated plan verification check to ensure logic validity in the plan verification and a code explanation that aligns incorrect code with the natural language refinement suggestion. LPW effectively integrates these techniques with plan verification into an end-to-end framework, achieving state-of-the-art Pass@1 accuracy and demonstrating superiority over existing methods. LDB, however, provides the block-level explanation, overlooking the LLM's comprehension of the overall structure of erroneous code, thereby reducing refinement effectiveness.

C Implementation Details

For all experiments, we configure greedy decoding with a temperature of $T = 0$ in LLMs to ensure reproducibility. LPW utilizes 2-shot prompting, aligned with LDB, which also adopts 2-shot prompting for block-level debugging. Similarly, 2-shot prompting is implemented in Self-Planning to maintain consistency. For Self-Debugging (+Expl), we adhere to its original paper's approach, using 3-shot prompting. Conversely, MapCoder employs 1-shot prompting to ensure that responses conform to the XML format.

In LPW, only essential messages are retained in the context window, excluding the entire conversation. For instance, the context window for initial code generation includes the program description, plan, and plan verification, while omitting prior interactions such as plan generation, verification generation, and verification checks. This facilitates LPW to function within a constrained context window while optimizing token efficiency in the absence of context caching.

D Parameter Study

We conduct an experiment involving 20 iterations for both the solution generation and code implementation phases in LPW. Figure 11 shows the variation in Pass@1 accuracy on the HumanEval benchmark using GPT-3.5. The increased number of iterations in the solution generation phase results in higher initial program accuracy, 81.1%, compared to 79.9% with 12 iterations before refinements. Subsequently, the accuracy steadily increases, reaching the highest value of 89.6% after 18 debugging turns, compared to a maximum of 89.0% with 12 iterations. Overall, a larger number of iterations improves performance in both initial code generation and subsequent refinements. However, the significant token consumption presents

	Missing Conditions	Differ from Intended Solution	No Code	Others
LPW	33.3	5.6	50.0	11.1

Table 6: The percentage of different failure reasons for LPW on the HumanEval benchmark with GPT-3.5 as the backbone. *Missing Conditions* arises from the same issues in the plan and plan verification. *Differ from Intended Solution* indicates the plan and plan verification are manually classified as correct, while the generated code deviates, resulting in failure. *No Code* represents the absence of valid plan and plan verification in the solution generation phase, leading to failure after reaching the maximum number of iterations. *Others* denotes error program solutions caused by various reasons that differ from the previously listed categories.

		HumanEval	MBPP
GPT-3.5	MapCoder	18.9	46.4
	SD	22.6	36.1
	LDB	28.6	37.7
	LPW (ours)	**44.4**	**36.7**

Table 7: The percentage of problems where MapCoder, Self-Debugging (+Expl) (SD), LDB, and LPW generated programs solve the visible tests but fail the hidden tests, out of total failed problems for each method on HumanEval and MBPP, with GPT-3.5 as the backbone.

		HumanEval	MBPP
GPT-3.5	MapCoder	4.3	13
	SD	4.3	10.4
	LDB	4.9	10.4
	LPW (ours)	**4.9**	**8.8**

Table 8: The percentage of problems where MapCoder, Self-Debugging (+Expl) (SD), LDB, and LPW generated programs pass the visible tests but fail the hidden tests, out of a total of 164 problems in HumanEval and 500 problems in MBPP, with GPT-3.5 as the backbone.

challenges for practical applications.

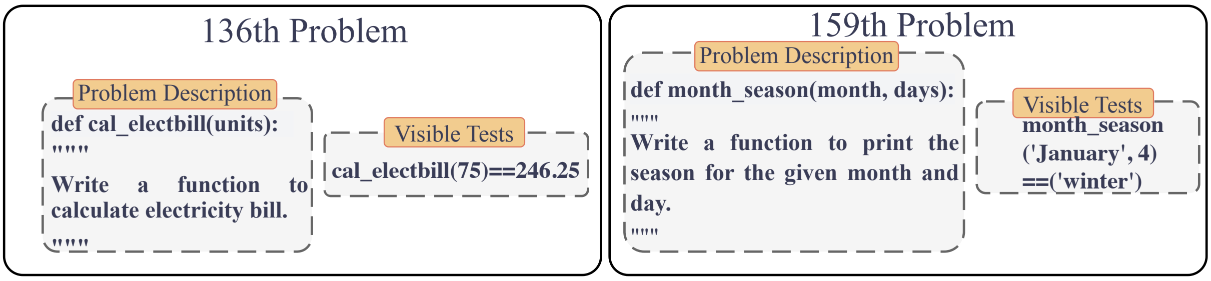
E Analysis of Unsolved Problems for LPW with GPT-3.5

E.1 Failure Reasons

LPW achieves state-of-the-art performance among the evaluated methods and demonstrates superiority over other LLM debuggers. We categorize the failure reasons for LPW on HumanEval with GPT-3.5 into 4 types. Table 6 compares the percentage of different failure reasons out of the total unsolved problems for LPW based on authors' manual review. In LPW, half of the errors result from the *No Code* type, where the generated solution plan fails to be verified on the visible tests, or the resulting verification includes incorrect intermediate outputs in the solution generation phase, leading to failure after reaching the maximum iteration threshold. The second most common reason is *Missing Conditions*, originating from the same issues in the plan and plan verification. For LPW, 5.6% of failures result from the generated program solution differing from the plan and plan verification (*Differ from Intended Solution*). For example, LPW fails in the *91st* problem, where the generated program is unable to solve the hidden tests due to deviations from the plan and plan verification. The plan verification clearly specifies splitting the input string into sentences using delimiters ".", "?" or "!", but the generated code only handles the full stop case and ignores "?" and "!".

E.2 Failure on Hidden Tests

Tables 7 and 8 show the percentage of problems where MapCoder, Self-Debugging (+Expl) (SD), LDB, and LPW generated program solutions pass the visible tests but fail the hidden tests, out of respectively failed problems and the total number of problems in the HumanEval and MBPP benchmarks using GPT-3.5 as the backbone. In Table 7, 44.4% of failures in LPW result from solving the visible tests only on the HumanEval benchmark, since except for the *No Code* category, other reasons discussed in Table 6 could contribute to this issue. In contrast, less than 30% of failures in MapCoder, SD and LDB experience this issue on HumanEval as the larger number of failed problems in these methods. In Table 7, MapCoder demonstrates a higher likelihood of passing only the visible tests, while all other evaluated approaches show similar percentages on the MBPP benchmark, with the remaining failures arising from different reasons. We note that all methods tend to address visible tests only on the same set of problems in the HumanEval benchmark, resulting in the similar percentage out of the total number of problems, as shown in Table 8. In contrast, on the MBPP benchmark, MapCoder exhibits the highest rate of passing only the visible tests out of the total number of problems, consistent with the result in Table 7, while LPW demonstrates



```
def cal_electbill(units):
"""
Write a function to calculate electricity bill.
"""
```

```
cal_electbill(75)==246.25
```

```
def month_season(month, days):
"""
Write a function to print the season for the given month and day.
"""
```

```
month_season('January', 4)==('winter')
```

Figure 12: Example problems in MBPP.

the lowest rate, as shown in Table 8. Meanwhile, all methods are prone to addressing visible tests only on MBPP rather than on HumanEval as indicated in Table 8. Compared to the detailed problem descriptions in HumanEval, the problem descriptions in MBPP are concise but lack clarity. For example, Figure 12 illustrates two problems in MBPP where LPW generated solutions are tailored to the visible tests but deviate significantly from the canonical solution.

F Experimental Results for MapCoder with GPT-4o as the Backbone

Table 9 includes the Pass@1 accuracy for MapCoder when using GPT-4o as the backbone across various benchmarks. MapCoder generally underperforms compared to LDB and lags behind LPW across all benchmarks, consistent with the results in Table 1 using GPT-3.5 as the backbone. This aligns with recent findings (Tam et al., 2024), which suggest that structured generation constraints, such as XML, may hinder LLM reasoning performance.

G Analysis of Unsolved Problems for LPW with GPT-4o

G.1 HumanEval

G.1.1 The *91st* Problem

Figure 15 illustrates the *91st* problem in HumanEval, where the GPT-4o generated code (part (c)) contains an incorrect condition. The code verifies if the sentence starts with the letter “I”, which is inconsistent with the problem description (part (a)) that requires the sentence to start with the word “I”. The provided visible tests (part (b)) fail to clarify the correct condition, resulting in the error program passing the visible tests only. Inspired by the superior learning-from-test capacity discussed earlier, we convert a failed hidden test into a visible test, highlighted in red in part (d). Consequently, GPT-4o successfully generates the correct program, as shown in part (e).

G.1.2 The *145th* Problem

Figure 17 displays the *145th* problem, where the incomplete problem description (part (a)) results in incorrect plan verification on visible tests (part (b)), leading to a failure after reaching the iteration threshold. The problem description requires returning a list sorted by the sum of digits but omits the specification regarding the sign of negative numbers. This omission confuses GPT-4o, resulting in consistently incorrect sorting when verifying the solution plan on the first visible test. We refine the problem description with a detailed explanation on handling both positive and negative numbers (part (c)), leading to the correct program solution, as shown in part (d).

G.1.3 The *132nd* Problem

Figure 16 illustrates the *132nd* problem, where ambiguities in the problem description (part (a)) challenge GPT-4o. The problem description lacks clarity on “a valid subsequence of brackets” and fails to specify the meaning of “one bracket in the subsequence is nested”. We deduce the intended problem description by prompting GPT-4o with a canonical solution (part (d)). However, the LLM-generated description remains unclear and results in various erroneous programs. Furthermore, adding typically failed hidden tests to the visible tests (part (b)) is also ineffective in clarifying the correct logic. We acknowledge that a clearer description might help generate the correct program. However, some problems are inherently difficult to describe accurately in natural language without careful organization, posing challenges for LLMs.

G.2 LiveCode

LiveCode, a contamination-free dataset, serves as a reliable benchmark for evaluating code genera-

		HumanEval	HumanEval-ET	MBPP	MBPP-ET	LiveCode	APPS	CodeContests
GPT-4o	Baseline	91.5	81.7	78.4	62.6	45.7	41.7	28.0
	MapCoder	92.7	82.3	81.2	64.6	52.9	51.8	28.7
	LDB	92.1	81.7	82.4	65.4	54.3	53.2	29.3
	LPW (ours)	**98.2**	**84.8**	**84.8**	**65.8**	**59.3**	**62.6**	**34.7**

Table 9: Extended Table 2 to include Pass@1 accuracy (in percentages) for MapCoder across the HumanEval, HumanEval-ET, MBPP, MBPP-ET, LiveCode, APPS, and CodeContests benchmarks, using GPT-4o as the backbone.

		LiveCode
GPT-4o	LDB	23.4
	LPW (ours)	**31.6**

Table 10: The percentage of problems where LDB and LPW generated programs solve the visible tests but fail the hidden tests, out of total failed problems for each method in LiveCode, with GPT-4o as the backbone.

		LiveCode
GPT-4o	LDB	10.7
	LPW (ours)	**12.9**

Table 11: The percentage of problems where LDB and LPW generated programs pass the visible tests but fail the hidden tests, out of a total of 140 problems in LiveCode, with GPT-4o as the backbone.

tion performance. It presents challenges for the advanced LLM GPT-4o, with all competing approaches showing performance limitations. We note that LPW surpasses Baseline and LDB by approximately 15% and 5% in Pass@1 accuracy on the LiveCode benchmark, respectively, as shown in Table 2, underscoring its reliability.

G.2.1 Failure on Hidden Tests

Tables 10 and 11 show the percentage of problems where the program solutions generated by LDB and LPW pass only the visible tests but fail the hidden tests out of the total number of failed problems and the total number of problems, respectively, on the LiveCode benchmark using GPT-4o. For LPW, 31.6% of failures occur when only the visible tests are solved, while for LDB, this percentage is 23.4%, as shown in Table 10. In Table 11, both LDB and LPW generate a similar proportion of solutions that pass only the visible tests. However, LPW tends to solve only the visible tests in 12.9% of problems, compared to 10.7% for LDB.

		APPS	CodeContests
GPT-4o	LDB	23.1	27.4
	LPW (ours)	**23.1**	**29.6**

Table 12: The percentage of problems where LDB and LPW generated programs solve the visible tests but fail the hidden tests, out of total failed problems for each method in APPS and CodeContests, with GPT-4o as the backbone.

		APPS	CodeContests
GPT-4o	LDB	10.8	19.3
	LPW (ours)	**8.6**	**19.3**

Table 13: The percentage of problems where LDB and LPW generated programs pass the visible tests but fail the hidden tests, out of a total of 139 problems in APPS and 150 problems in CodeContests, with GPT-4o as the backbone.

G.3 APPS and CodeContests

APPS and CodeContests are unstructured benchmarks where visible tests are intermingled with the problem statements and function signatures are excluded. To align input data structure across benchmarks, we instruct GPT-4o to derive the optimal function signature and identify visible tests for each problem in APPS and CodeContests prior to conducting experiments. Example structured problems from APPS and CodeContests are illustrated in Figures 18 and 19. In Table 2, LPW demonstrates significant improvements on APPS and CodeContests, exceeding around 10% and 5% Pass@1 accuracy, respectively, compared to LDB with GPT-4o. However, in contrast to the performance on the HumanEval and MBPP benchmarks, where LPW achieves 98.2% and 84.8% Pass@1 accuracy, the 62.6% accuracy on APPS and 34.7% accuracy on CodeContests indicate that even for the advanced LLM GPT-4o, code generation remains challenging when addressing complicated programming problems, such as those encountered in collegiate programming competitions like IOI

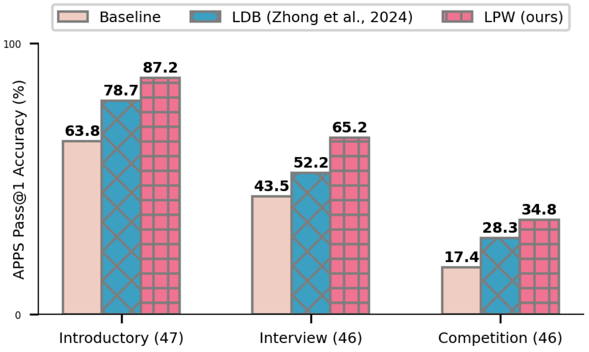


Figure 13: Pass@1 accuracy of Baseline, LDB, and LPW across different difficulty levels, *Introductory*, *Interview*, and *Competition*, on the APPS benchmark when using GPT-4o as the LLM backbone.

and ACM (Hendrycks et al., 2021).

G.3.1 Performance Across Different Difficulty Levels

Figure 13 compares the Pass@1 accuracy of Baseline, LDB, and LPW across different difficulty levels, *Introductory*, *Interview*, and *Competition*, on the APPS benchmark using GPT-4o. LPW consistently dominates in Pass@1 accuracy across all difficulty levels. LPW shows strong performance on the relatively easier levels, i.e., *Introductory* and *Interview*, surpassing LDB by around 9% and 13% accuracy, respectively, and outperforming Baseline by over 20% accuracy. For the problems belonging to the most challenging level, *Competition*, LPW achieves 34.8% accuracy, compared to 28.3% for LDB and 17.4% for Baseline. However, all approaches experience a substantial decrease at the *Competition* level, emphasizing the necessity for further improvements.

G.3.2 Failure on Hidden Tests

Tables 12 and 13 present the percentage of problems where the generated program solutions from LDB and LPW solve visible tests but fail hidden tests out of the total failed problems and the total number of problems, respectively, on the APPS and CodeContests benchmarks using GPT-4o as the backbone. In Table 12, 23.1% of failures result from passing only the visible tests on the APPS benchmark, with this percentage increasing to around 30% on CodeContests for both LDB and LPW. In Table 13, LDB and LPW display similar percentages of solving visible tests only on each benchmark, ranging from around 10% on APPS to 19% on CodeContests. Compared to the results in Table 8, where LDB and LPW address only visible tests in around 5% of problems on the HumanEval benchmark, LDB and LPW exhibit weaker performance on the more challenging APPS and CodeContests benchmarks. This is particularly evident on CodeContests, where the percentage is roughly twice as high as APPS for both LDB and LPW. In APPS and CodeContests, each problem averages approximately 2 visible tests, while CodeContests includes more comprehensive hidden tests, averaging about 23 per problem compared to only around 5 per problem in APPS, increasing the likelihood of solving only the visible tests.

		HumanEval		MBPP	
		Acc	Δ	Acc	Δ
GPT-3.5	LPW	89.0	–	76.0	–
	LPW-E	87.8	-1.2	75.6	-0.4

Table 14: Pass@1 accuracy (Acc) for the variant of LPW with the GPT-3.5 backbone. The suffix -E stands for the exclusion of code explanation in the code implementation phase. Other metrics remain consistent with those in Table 4.

H Additional Ablation Study

Table 14 shows the performance of the variant of LPW on the HumanEval and MBPP benchmarks using GPT-3.5 as the LLM backbone. The suffix -E denotes removing the code explanation when generating the refined program in the code implementation phase. The code explanation facilitates LLMs in aligning text-based error analysis with code implementation when locating and refining incorrect program lines. LPW-E demonstrates a decrease in Pass@1 accuracy on both the HumanEval and MBPP benchmarks.

I Accuracy of Plans, Plan Verifications, and Programs in LPW with GPT-3.5

I.1 Plans and Plan Verifications

LLMs show compelling performance when generating solution plans and plan verifications. In LPW, the solution generation phase requires an average of only 1.8 iterations to produce available plan and its verification on the HumanEval benchmark using GPT-3.5, significantly below the predefined limit of 12 iterations.

We manually investigate the accuracy of solution plans and plan verifications generated by GPT-3.5 on the HumanEval benchmark, and the results are presented in Table 15. Overall, GPT-3.5 generates

	Plan and Plan Verification	Correct Plan	Correct Plan Verification
LPW	94.5	92.7	92.7

Table 15: Percentage of problems where the LLM successfully generates the valid plans and plan verifications in the solution generation phase (first column); percentage of problems where the LLM-generated plans are manually classified as correct (middle column), considering no plan cases; and percentage of problems where the LLM-generated plan verifications are manually classified as correct (last column), considering no plan verification cases. All percentages are reported using GPT-3.5 as the backbone on the HumanEval benchmark, with a total of 164 problems.

	Wrong Code ← Wrong Plan Verification	Correct Code ← Correct Plan Verification
LPW	100	96.1

Table 16: The relationship between LLM-generated code solutions and plan verifications on the HumanEval benchmark with GPT-3.5. The first column shows the percentage of problems where the LLM generates incorrect code solutions when plan verifications are incorrect; the second column shows the percentage of problems where correct code solutions are generated when plan verifications are correct.

the correct solution plans and plan verifications in natural language for majority of problems. In LPW, GPT-3.5 successfully produces plans and plan verifications for 94.5% of the problems. GPT-3.5 generates the correct plans for 92.7% of the problems and achieves the same accuracy for plan verifications. A common issue in the LLM-generated plan is the omission of certain conditions. For example, solution plan frequently overlooks uppercase situations and negative numbers. We note that the LLM-generated plan verification closely adheres to the solution plan. When the plan is accurate, the verification process strictly follows the plan logic, resulting in a correct analysis. Conversely, if the plan contains logical errors or omits edge cases, the verification process replicates these mistakes. Specifically, for LPW, all correct plans lead to accurate plan verifications, and vice versa.

I.2 Plan Verifications and Programs

We further manually explore the relationship between plan verification and program solution on the HumanEval benchmark with GPT-3.5. Table 16 evaluates the conditional probabilities between wrong code and wrong plan verification, as well as between correct code and correct plan verification. Typically, in LPW, accurate plan verifications significantly contribute to correct program solutions, whereas incorrect plan verifications inevitably result in errors. LPW generates program solutions based on plans and plan verifications. Therefore, any accurate descriptions or mistakes, including missed conditions, in the plan and plan verification are propagated to the code. When plan verifications are accurate, 96.1% of program solutions are correct in LPW. The remaining incorrect instances arise from unclear condition statements in plan verification that fail to account for hidden tests, leading to erroneous program solutions.

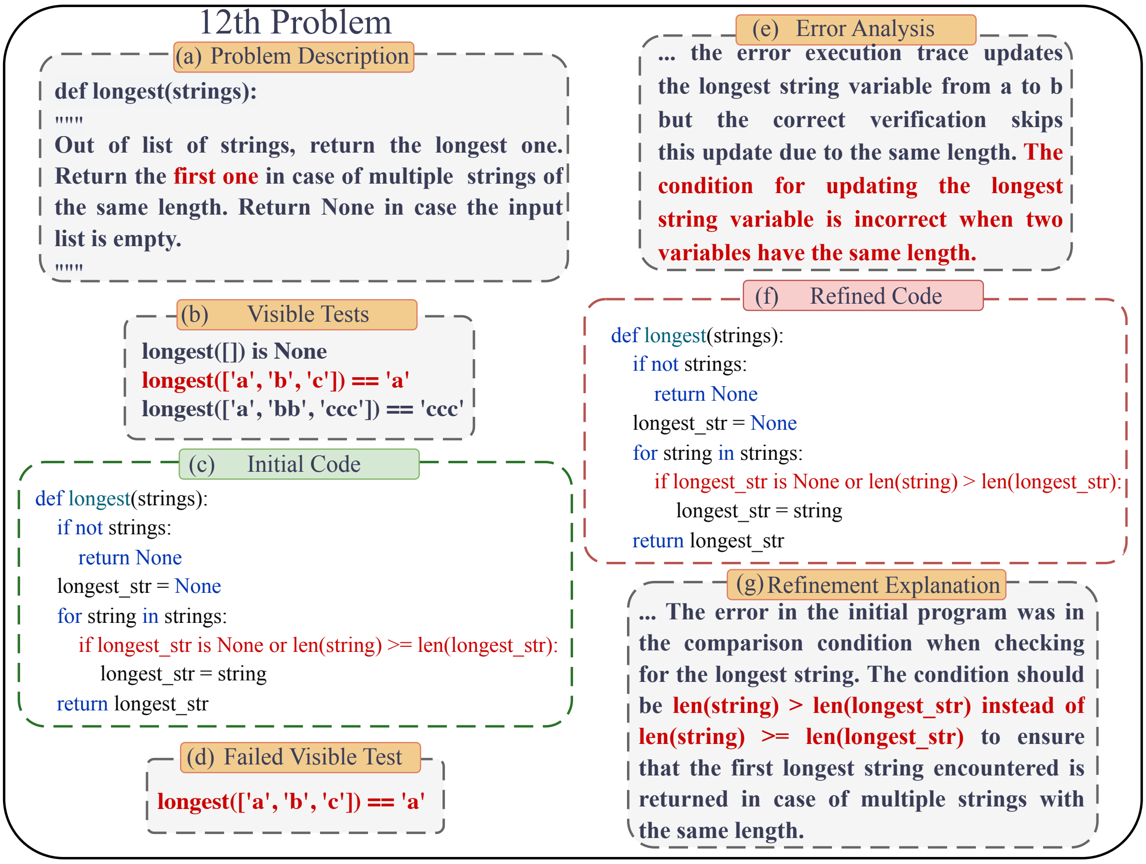
The results from Tables 15 and 16 highlight the impressive capability of LLMs in tackling text-to-code generation tasks when outputs are represented in natural language. Plan and plan verification generation accuracy is typically higher than code generation accuracy, underscoring the rationale behind LPW, which produces the high-quality program solution by leveraging plan and plan verification. It is worth exploring methods to help LLMs overcome the challenges of translating natural language solutions into programs, given the strict lexical, grammatical, and semantic constraints. Integrating alternative solution representations (Zelikman et al., 2023) alongside natural language representations could offer a promising approach.

J Refinement Consistency in LPW

LPW allows multiple rounds of debugging to refine code based on error analysis, generated by comparing the code execution trace with plan verification on the failed visible test. LPW queries LLMs to generate refined code accompanied by a refinement explanation, detailing the modifications implemented to address the errors identified in the error analysis. For instance, Figures 14 and 9 illustrate two HumanEval problems where LPW successfully generates the correct program through refinements informed by the error analysis using the GPT-3.5 backbone. We note that in LPW, if the refined code is irrelevant to the error analysis, the entire debugging process degrades to a simple sampling approach, contradicting our original intent. As a result, we manually evaluate the debugging consistency among the generated error analysis (part (e)), the refined code (part (f)), and the refinement explanation (part (g)), as exampled in Figure 14. LPW demonstrates excellent consistency between error analysis and the refined code, where only one refined code deviates from the error analysis yet still produces the correct solution, across all problems solved through debugging. This result validates the effectiveness of the debugging steps in the code implementation phase for LPW, where the meaningful error analysis enables LLMs to produce the correct program with precise refinements.

J.1 Prompts for LPW

We provide the LLM prompts used in LPW in Prompts 1 to 8. For conciseness, we only include one example in each prompt. Full prompts can be found in our released code.



12th Problem

(a) Problem Description

```
def longest(strings):
"""
Out of list of strings, return the longest one. Return the first one in case of multiple strings of the same length. Return None in case the input list is empty.
"""
```

(b) Visible Tests

```
longest([]) is None
longest(['a', 'b', 'c']) == 'a'
longest(['a', 'bb', 'ccc']) == 'ccc'
```

(c) Initial Code

```
def longest(strings):
    if not strings:
        return None
    longest_str = None
    for string in strings:
        if longest_str is None or len(string) >= len(longest_str):
            longest_str = string
    return longest_str
```

(d) Failed Visible Test

```
longest(['a', 'b', 'c']) == 'a'
```

(e) Error Analysis

... the error execution trace updates the longest string variable from a to b but the correct verification skips this update due to the same length. The condition for updating the longest string variable is incorrect when two variables have the same length.

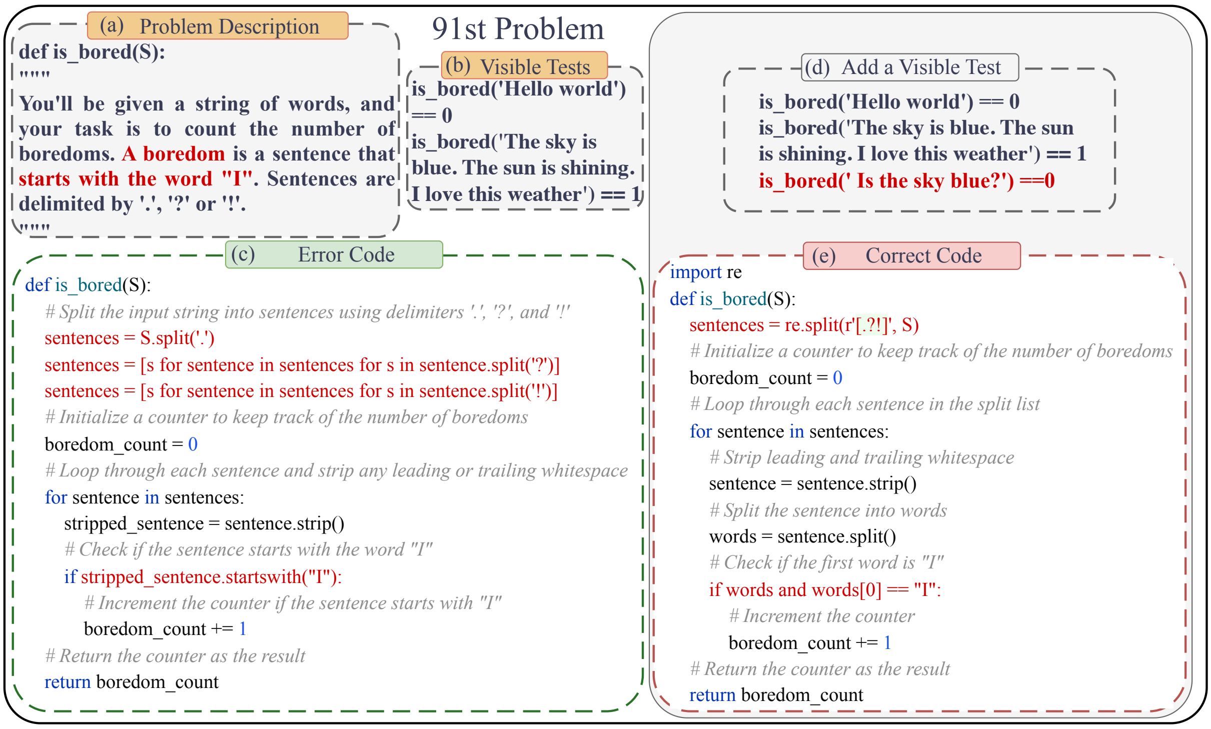
(f) Refined Code

```
def longest(strings):
    if not strings:
        return None
    longest_str = None
    for string in strings:
        if longest_str is None or len(string) > len(longest_str):
            longest_str = string
    return longest_str
```

(g) Refinement Explanation

... The error in the initial program was in the comparison condition when checking for the longest string. The condition should be len(string) > len(longest_str) instead of len(string) >= len(longest_str) to ensure that the first longest string encountered is returned in case of multiple strings with the same length.

Figure 14: The *12th* problem in HumanEval, where LPW with GPT-3.5 generated initial code (part (c)) is unable to solve one of the visible tests (part (d)). The refined code (part (f)) successfully solves both visible and hidden tests based on the error analysis (part (e)). The modification in the refined code aligns with the error analysis, as evidenced by the refinement explanation (part (g)).



```
def is_bored(S):
"""
You'll be given a string of words, and your task is to count the number of boredoms. A boredom is a sentence that starts with the word "I". Sentences are delimited by '.', '?' or '!'.
"""
```

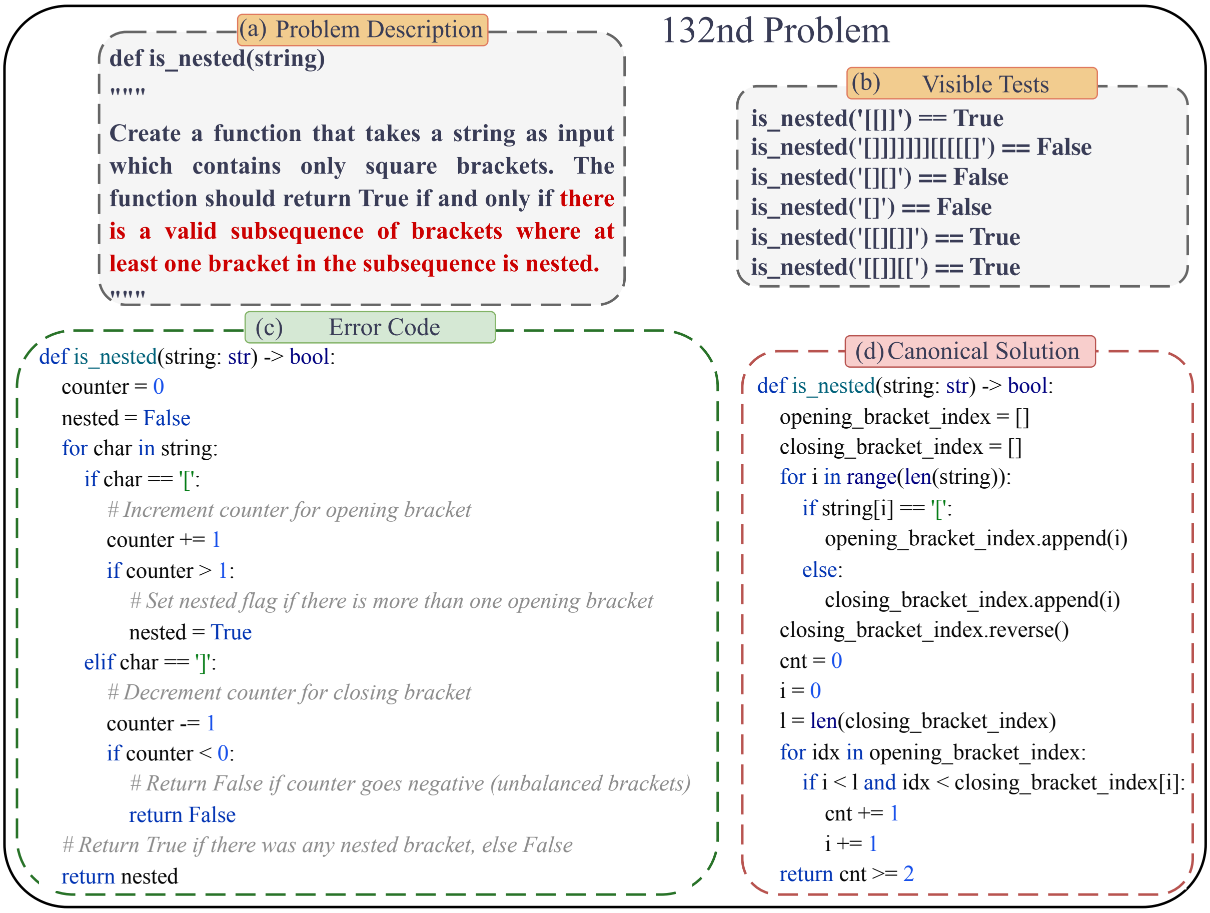
```
is_bored('Hello world') == 0
is_bored('The sky is blue. The sun is shining. I love this weather') == 1
```

```
def is_bored(S):
    # Split the input string into sentences using delimiters '.', '?', and '!'
    sentences = S.split('.')
    sentences = [s for sentence in sentences for s in sentence.split('?')]
    sentences = [s for sentence in sentences for s in sentence.split('!')]
    # Initialize a counter to keep track of the number of boredoms
    boredom_count = 0
    # Loop through each sentence and strip any leading or trailing whitespace
    for sentence in sentences:
        stripped_sentence = sentence.strip()
        # Check if the sentence starts with the word "I"
        if stripped_sentence.startswith("I"):
            # Increment the counter if the sentence starts with "I"
            boredom_count += 1
    # Return the counter as the result
    return boredom_count
```

```
is_bored('Hello world') == 0
is_bored('The sky is blue. The sun is shining. I love this weather') == 1
is_bored(' Is the sky blue?') ==0
```

```
import re
def is_bored(S):
    sentences = re.split(r'[.?!]', S)
    # Initialize a counter to keep track of the number of boredoms
    boredom_count = 0
    # Loop through each sentence in the split list
    for sentence in sentences:
        # Strip leading and trailing whitespace
        sentence = sentence.strip()
        # Split the sentence into words
        words = sentence.split()
        # Check if the first word is "I"
        if words and words[0] == "I":
            # Increment the counter
            boredom_count += 1
    # Return the counter as the result
    return boredom_count
```

Figure 15: The problem description (part (a)) and visible tests (part (b)) of the *91st* problem in HumanEval, where GPT-4o generated code (part (c)) addresses the visible tests but fails the hidden tests. However, after converting a failed hidden test to a visible test (part (d)), GPT-4o successfully generates the correct program (part (e)).



132nd Problem

(a) Problem Description

```
def is_nested(string)
"""
Create a function that takes a string as input which contains only square brackets. The function should return True if and only if there is a valid subsequence of brackets where at least one bracket in the subsequence is nested.
"""
```

(b) Visible Tests

```
is_nested('[[]]') == True
is_nested('[]]]]]]][[[[[]') == False
is_nested('[][]') == False
is_nested('[]') == False
is_nested('[[][]]') == True
is_nested('[[]][[') == True
```

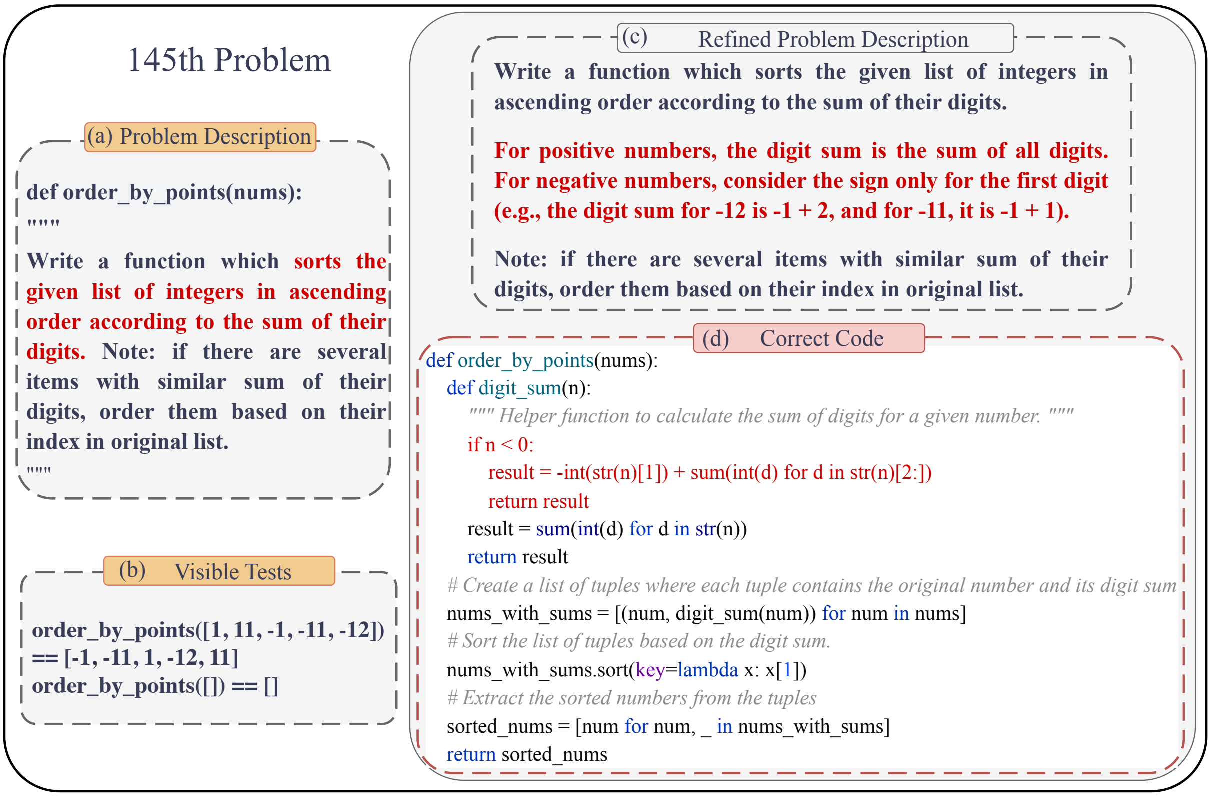
(c) Error Code

```
def is_nested(string: str) -> bool:
    counter = 0
    nested = False
    for char in string:
        if char == '[':
            # Increment counter for opening bracket
            counter += 1
            if counter > 1:
                # Set nested flag if there is more than one opening bracket
                nested = True
        elif char == ']':
            # Decrement counter for closing bracket
            counter -= 1
            if counter < 0:
                # Return False if counter goes negative (unbalanced brackets)
                return False
    # Return True if there was any nested bracket, else False
    return nested
```

(d) Canonical Solution

```
def is_nested(string: str) -> bool:
    opening_bracket_index = []
    closing_bracket_index = []
    for i in range(len(string)):
        if string[i] == '[':
            opening_bracket_index.append(i)
        else:
            closing_bracket_index.append(i)
    closing_bracket_index.reverse()
    cnt = 0
    i = 0
    l = len(closing_bracket_index)
    for idx in opening_bracket_index:
        if i < l and idx < closing_bracket_index[i]:
            cnt += 1
            i += 1
    return cnt >= 2
```

Figure 16: The problem description (part (a)) and visible tests (part (b)) of the *132nd* problem in HumanEval, where the GPT-4o generated error code (part (c)) passes the visible tests yet fails the hidden tests. GPT-4o consistently generates incorrect programs despite providing additional visible tests or refining the problem description.



145th Problem

(a) Problem Description

```
def order_by_points(nums):
"""
Write a function which sorts the given list of integers in ascending order according to the sum of their digits. Note: if there are several items with similar sum of their digits, order them based on their index in original list.
"""
```

(b) Visible Tests

```
order_by_points([1, 11, -1, -11, -12]) == [-1, -11, 1, -12, 11]
order_by_points([]) == []
```

(c) Refined Problem Description

Write a function which sorts the given list of integers in ascending order according to the sum of their digits.

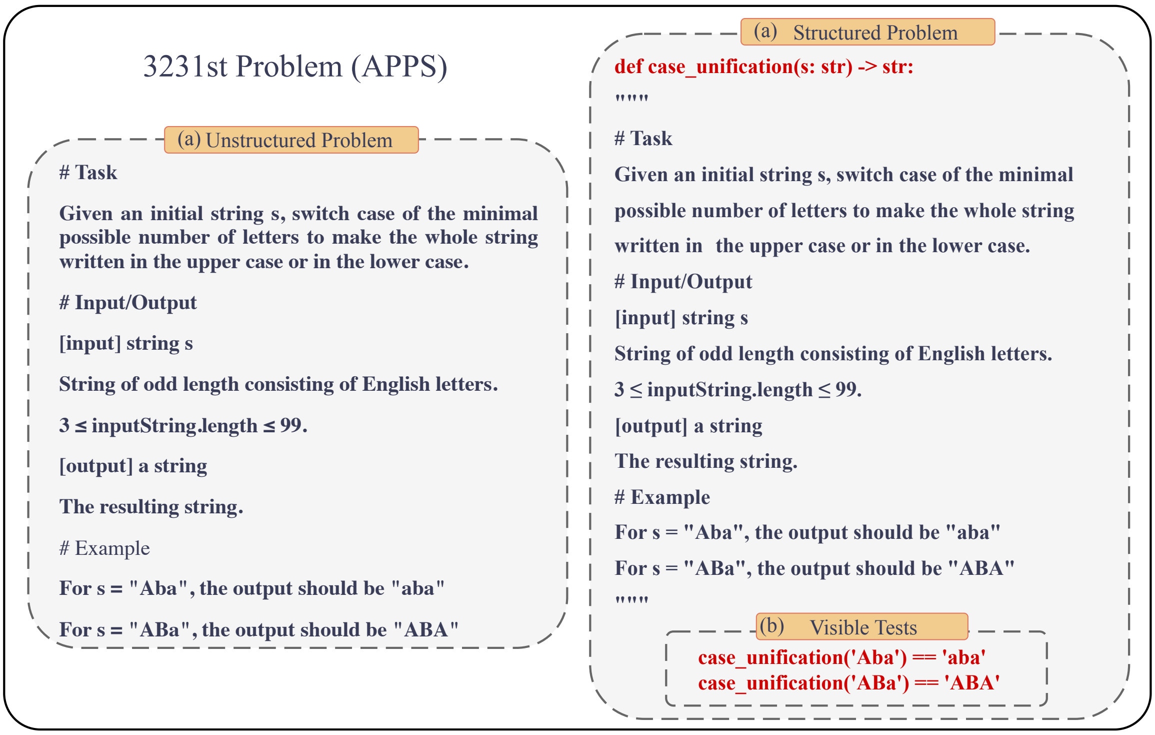
For positive numbers, the digit sum is the sum of all digits. For negative numbers, consider the sign only for the first digit (e.g., the digit sum for -12 is -1 + 2, and for -11, it is -1 + 1).

Note: if there are several items with similar sum of their digits, order them based on their index in original list.

(d) Correct Code

```python
def order_by_points(nums):
  def digit_sum(n):
    """ Helper function to calculate the sum of digits for a given number. """
    if n < 0:
      result = -int(str(n)[1]) + sum(int(d) for d in str(n)[2:])
      return result
    result = sum(int(d) for d in str(n))
    return result
  # Create a list of tuples where each tuple contains the original number and its digit sum
  nums_with_sums = [(num, digit_sum(num)) for num in nums]
  # Sort the list of tuples based on the digit sum.
  nums_with_sums.sort(key=lambda x: x[1])
  # Extract the sorted numbers from the tuples
  sorted_nums = [num for num, _ in nums_with_sums]
  return sorted_nums
```

Figure 17: The problem description (part (a)) and visible tests (part (b)) of the *145th* problem in HumanEval where GPT-4o fails to respond with a valid plan verification, resulting in failure. However, after refining the problem description (part (c)), GPT-4o successfully generates the correct program (part (d)).



3231st Problem (APPS)

(a) Unstructured Problem

# Task

Given an initial string s, switch case of the minimal possible number of letters to make the whole string written in the upper case or in the lower case.

# Input/Output

[input] string s

String of odd length consisting of English letters.

3 ≤ inputString.length ≤ 99.

[output] a string

The resulting string.

# Example

For s = "Aba", the output should be "aba"

For s = "ABa", the output should be "ABA"

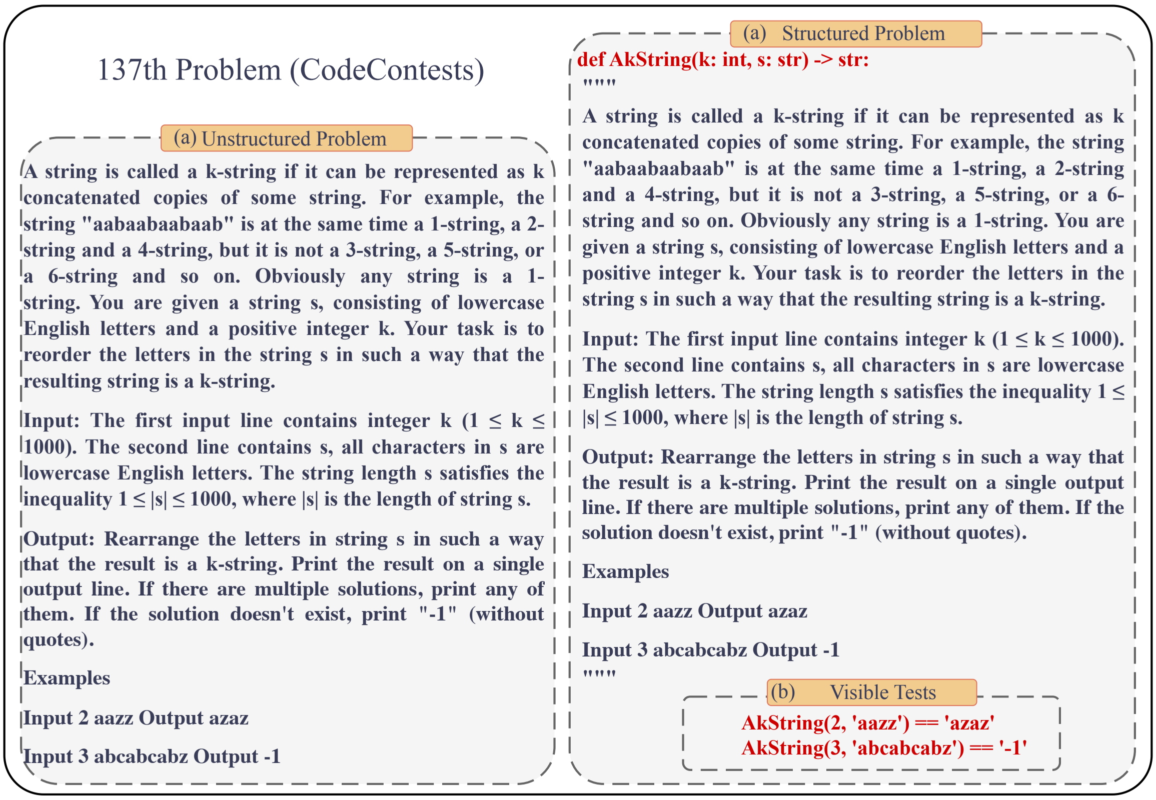
(a) Structured Problem

```
def case_unification(s: str) -> str:
"""
# Task
Given an initial string s, switch case of the minimal
possible number of letters to make the whole string
written in  the upper case or in the lower case.
# Input/Output
[input] string s
String of odd length consisting of English letters.
3 ≤ inputString.length ≤ 99.
[output] a string
The resulting string.
# Example
For s = "Aba", the output should be "aba"
For s = "ABa", the output should be "ABA"
"""
```

(b) Visible Tests

```
case_unification('Aba') == 'aba'
case_unification('ABa') == 'ABA'
```

Figure 18: An example structured APPS problem with a function signature and visible tests, generated by instructing GPT-4o with the unstructured problem description.



137th Problem (CodeContests)

(a) Unstructured Problem

A string is called a k-string if it can be represented as k concatenated copies of some string. For example, the string "aabaabaabaab" is at the same time a 1-string, a 2-string and a 4-string, but it is not a 3-string, a 5-string, or a 6-string and so on. Obviously any string is a 1-string. You are given a string s, consisting of lowercase English letters and a positive integer k. Your task is to reorder the letters in the string s in such a way that the resulting string is a k-string.

Input: The first input line contains integer k (1 ≤ k ≤ 1000). The second line contains s, all characters in s are lowercase English letters. The string length s satisfies the inequality 1 ≤ |s| ≤ 1000, where |s| is the length of string s.

Output: Rearrange the letters in string s in such a way that the result is a k-string. Print the result on a single output line. If there are multiple solutions, print any of them. If the solution doesn't exist, print "-1" (without quotes).

Examples

Input 2 aazz Output azaz

Input 3 abcabcabz Output -1

(a) Structured Problem

```
def AkString(k: int, s: str) -> str:
"""
A string is called a k-string if it can be represented as k concatenated copies of some string. For example, the string "aabaabaabaab" is at the same time a 1-string, a 2-string and a 4-string, but it is not a 3-string, a 5-string, or a 6-string and so on. Obviously any string is a 1-string. You are given a string s, consisting of lowercase English letters and a positive integer k. Your task is to reorder the letters in the string s in such a way that the resulting string is a k-string.

Input: The first input line contains integer k (1 ≤ k ≤ 1000). The second line contains s, all characters in s are lowercase English letters. The string length s satisfies the inequality 1 ≤ |s| ≤ 1000, where |s| is the length of string s.

Output: Rearrange the letters in string s in such a way that the result is a k-string. Print the result on a single output line. If there are multiple solutions, print any of them. If the solution doesn't exist, print "-1" (without quotes).

Examples

Input 2 aazz Output azaz

Input 3 abcabcabz Output -1
"""
```

(b) Visible Tests

```
AkString(2, 'aazz') == 'azaz'
AkString(3, 'abcabcabz') == '-1'
```

Figure 19: An example structured CodeContests problem with a function signature and visible tests, generated by instructing GPT-4o with the unstructured problem description.

Listing 1: Prompt for plan generation

```
====================================== System ======================================

You are a Python writing assistant that responds with a step-by-step thought process (
IN ENGLISH) to solve Python coding problems.

======================================= User =======================================

You will be provided with a series of examples, where each example begins with [Start
Example] and ends with [End Example]. In each example, you will be presented with a
Python coding problem, starting with [Example Problem Description], which includes the
function signature and its accompanying docstring. You will then provide a reasonable
solution plan, starting with [Example Start Plan] and ending with [Example End Plan],
to solve the given problem.

[Start Example]
[Example Problem Description]
def encrypt(s):
    ”””
    Create a function encrypt that takes a string as an argument and returns a string
    encrypted with the alphabet being rotated. The alphabet should be rotated in a
    manner such that the letters shift down by two multiplied to two places.
    ”””

[Example Start Plan]
Create an alphabet, biased by two places multiplied by two.
Loop through the input, find the letter biased by the alphabet.
Return the result.
[Example End Plan]
[End Example]

... Authors' notes: We omit another example for conciseness. The full prompt can be
found in our released code. ...

Lastly, you will be given a Python writing problem, beginning with [Problem Description
], which includes the function signature, its docstring, and any potential constraints.
 The phrase ”Let's think step by step” will signal the start of the plan. Your task is
to create a solution plan, starting with [Start Plan] and ending with [End Plan].
```

Listing 2: Prompt for plan verification

```
====================================== System ======================================

You are a logical reasoner tasked with performing a step-by-step analysis to derive the
 correct solution to the given problem based on the provided plan.

======================================= User =======================================

You will be presented with several plan verification examples, each starting with [
Start Example] and ending with [End Example]. In each example, you will be given a
Python writing problem, starting with [Example Problem Description], followed by the
solution plan starting with [Example Solution Plan], and its verification process
beginning with [Example Plan Verification for X] for a test case X, starting with [
Example Test Cases]. During the verification process, intermediate variables that need
to be recorded are clearly identified at the outset, starting with [Record Analysis].
Whenever the value of a recorded intermediate variable is updated, the new result is
clearly displayed, beginning with [Record]. After the verification, the derived result
is compared to the correct test result, starting with [Results Compare]. If the derived
 result matches the test result, the output will be [Correct Plan]. If the derived
result differs from the test result, the output will be [Incorrect Plan], followed by
the reasons for the discrepancy, starting with [Incorrect Reasons], and the revised
correct solution plan, beginning with [Start Revised Solution Plan] and ending with [
End Revised Solution Plan].

[Start Example]
[Example Problem Description]
def prime_number(n: int):
    ”””
    In range 0 to 100, returns n-th number that is a prime.
    """

[Example Solution Plan]
1. Iterate number through 0 to 100.
2. Check each number, if it's prime.
3. Keep track of the count of prime numbers found.
4. Stop when we find the n-th prime number.
5. Return the nth prime number.
[Example Test Cases]
assert prime_number(3)==5
[Example Plan Verification for assert prime_number(2)==3]
[Record Analysis]
The return value is the nth prime number, so all nth prime numbers need to be clearly
recorded!
1. Call the function prime_number(2).
2. According to line 1 in solution plan, Iterate number through 0 to 100.
3. According to line 2 in solution plan, Check if 0 is prime. It's not.
4. Move to next number 1.
5. According to line 2 in solution plan, Check if 1 is prime. It's not.
6. Move to next number 2.
7. According to line 2 in solution plan, Check if 2 is prime. It is a prime.
8. According to line 3 in solution plan, the count of prime numbers is 1.
[Record]: 1st prime number is 2
9. Move to next number 3.
10. According to line 2 in solution plan, Check if 3 is prime. It is a prime.
11. According to line 3 in solution plan, the count of prime numbers is 2.
[Record]: 2nd prime number is 3
12. According to line 4 in solution plan, Stop when we find the 2nd prime number.
13. According to line 5 in solution plan, Return the 2nd prime number, which is 3.
[Results Compare]
The test correct output is 3. The logic analysis output is 3. 3=3. Thus, the plan is
verified to correctly handle all test cases.
[Correct Plan]
[End Example]

... Authors' notes: We omit another example for conciseness. The full prompt can be
found in our released code. ...

Finally, you will be given a problem description, beginning with [Problem Description],
 along with your generated solution plan, starting with [Solution Plan], to solve the [
Problem Description], and multiple test cases starting with [Test Cases]. The phrase "
Let's verify the plan" will indicate the beginning of the verification process,
followed by your verification steps to confirm whether your generated plan can pass all
 test cases.

For each test case, the verification must include [Record Analysis] to track the
intermediate variables at the beginning. If any intermediate variable value is updated
during the reasoning process, the updated value should be clearly displayed, starting
with [Record]. Please include [Results Compare] to assess the derived outcome against
the correct test output. If the derived result matches the test result, output [Correct
 Plan] and proceed to the next test case. If the derived result does not match the test
 result, output [Incorrect Plan], followed by the reasons for the discrepancy, starting
 with [Incorrect Reasons]. Finally, provide the revised solution plan, starting with [
Start Revised Solution Plan] and ending with [End Revised Solution Plan], to complete
the process.
```

Listing 3: Prompt for plan verification check

```
======================================= System ========================================

You are a logical reasoner. Your goal is to identify any incorrect logic within the
logic verification process.

======================================== User =========================================

You will be given several examples demonstrating how to evaluate a logic verification
process. Each example will begin with [Start Example] and end with [End Example]. In
each example, you will find the following:

[Example Problem Description] outlining the Python writing problem;
```

```
[Example Solution Plan] describing the approach to solve the problem;

[Example Plan Verification for X], applying the solution plan to a specific test case X.
 In this process, the intermediate variables to be tracked are analyzed at the start,
marked by [Record Analysis]. Whenever the value of a recorded intermediate variable is
updated, its new value is displayed starting with [Record]. The [Results Compare]
section compares the verification derived result with the correct test output;

[Example Verification Check for X], this section evaluates, step by step, whether the
logic verification process for test case X is correct.

If the verification is correct, the output will be [Correct Plan Verification], and
please proceed to the next example. If the verification is incorrect, explanation
should be provided and [Incorrect Plan Verification] will be the output to conclude the
 evaluation.

[Start Example]
[Example Problem Description]
def addOne(message: str):
    """
    You are given a large integer represented as an integer array digits, where each
    digits[i] is the ith digit of the integer. The digits are ordered from most
    significant to least significant in left-to-right order. The large integer does not
    contain any leading 0's. Increment the large integer by one and return the resulting
     array of digits.
    """

[Example Solution Plan]
1. Convert the list of digits into a number.
2. Increment the number by one.
3. Convert the incremented number back into a list of digits and return it.

[Example Plan Verification for assert addOne([1,2,3])==[1,2,4]]
[Record analysis]
The return value is the incremental resulting array of digits, so the incremental
resulting array of digits needs to be clearly recorded!

According to line 1 in solution plan, convert [1,2,3] to the number 123.
According to line 2 in solution plan, Increment 123 by one to get 124.
According to line 3 in solution plan, convert 124 back into the list [1,2,4]
[Record]: incremental resulting array is [1,2,4]
According to line 3 in solution plan return incremental resulting array [1,2,4].

[Results Compare]
The test correct output is [1,2,4]. The logic analysis output is [1,2,4].
[1,2,4]=[1,2,4]. So the plan is verified to correctly handle all test cases.

[Correct Plan]

[Example Verification Check for assert ddOne([1,2,3])==[1,2,4]]:
"Convert [1,2,3] to the number 123" is correct!
"Increment 123 by one to get 124" is correct! since 123+1=124
"Convert 124 back into the list [1,2,4]" is correct!
"return incremental resulting array [1,2,4]" is correct!

In [Results Compare] "The test correct output = [1,2,4]" is correct! "The logic
analysis output = [1,2,4]" is correct! The results comparison "[1,2,4]=[1,2,4]" is
correct!

All analysis steps are correct!

[Correct Plan Verification]

[Example Plan Verification for assert addOne([-1,2])==[-1,1]]
[Record analysis]
The return value is the incremental resulting array of digits, so the incremental
resulting array of digits needs to be clearly recorded!
According to line 1 in solution plan, convert [-1,2] to the number 12.
According to line 2 in solution plan, Increment 12 by one to get 13.
According to line 3 in solution plan, convert 13 back into the list [1,3]
[Record]: incremental resulting array is [1,3]

According to line 3 in solution plan return incremental resulting array [1,3].

[Results Compare]
The test correct output is [-1,1]. The logic analysis output is [-1,1]. [-1,1]=[-1,1].
So the plan is verified to correctly handle all test cases.
[Correct Plan]

[Example Verification Check for assert addOne([-1,2])==[-1,1]]:

”Convert [-1,2] to the number 12” is incorrect. The analysis doesn't correctly
interpret the -1 and assumes all values are positive, the sequence -1, 2 should form
-12.
”Increment 12 by one to get 13” is correct, but as established, the initial conversion
should not yield 12.
”Convert 13 back into the list [1,3]” is correct!
”Return incremental resulting array [1,3]” is correct!

In [Results Compare] ”The test correct output = [-1,1]” is correct! ”The logic analysis
 output = [-1,1]” is incorrect! The logic analysis result is [1,3] mentioned in the
verification ”return incremental resulting array [1,3]”. The results comparsion
”[-1,1]=[-1,1]” is incorrect! The logic analysis result is [1,3] and [-1,1] is not
equal [1,3].

The logic verification process for addOne([-1,2])==[-1,1] is incorrect. The analysis
doesn't correctly interpret the -1 and assumes all values are positive, the sequence -1,
 2 should form -12. The logic analysis output = [-1,1] is incorrect! It is [1,3]. The
results comparison is incorrect since [-1,1] is not equal [1,3].

[Incorrect Plan Verification]

[End Example]

... Authors' notes: We omit another example for conciseness. The full prompt can be
found in our released code. ...

Finally, you will be given a problem description, beginning with [Problem Description],
 followed by your generated solution plan, starting with [Solution Plan], to address
the [Problem Description]. You will then work through multiple Plan Verification, each
starting with [Plan Verification for X], where X represents a test case. At the start
of the verification process, [Record Analysis] examines the intermediate variables that
 should be tracked. During the logic verification, the tag [Record] indicates any
updates to the values of the recorded intermediate variables. The [Results Compare]
section documents the comparison between the verification derived result and the
expected test output.

The phrase ”Let's evaluate the verification” will indicate the start of the evaluation
for each verification process. This will be followed by your step-by-step verification
check to assess whether each intermediate output in the verification process is correct,
 starting with [Verification Check for X], as shown in the examples. If all
intermediate results in the verification process are correct, the output will be [
Correct Plan Verification], and you will proceed to the next verification. If the
verification process is incorrect, an explanation should be provided, and [Incorrect
Plan Verification] will be output to conclude the evaluation.
```

Listing 4: Prompt for initial code generation

```
======================================= System ======================================

You are a Python writing assistant that only responds with Python programs to solve a
Python writing problem.

======================================== User ======================================

You will receive several examples, each structured as follows, beginning with [Start
Example] and ending with [End Example]. Within each example, you will find a Python
programming problem starting with [Example Problem Description] and a solution plan
starting with [Example Solution Plan]. Additionally, you will receive plan
```

```
verifications for specific test cases. For each test case X, the plan verification is
labeled as [Example Plan Verification for X], providing a detailed logical breakdown
and variable value updates, which are recorded starting with [Record]. Following the
verification, you will encounter the example-generated program starting with [Example
Generated Program]. The program, marked from [Start Program] to [End Program], is
generated based on the solution plan and plan verification, ensuring that the program's
 execution aligns with the plan verification when test case X is used as input.

[Start Example]

[Example Problem Description]
from typing import List
def get_closest_transition_character(word):
    """
    You are given a word. Your task is to find the closest transition character from the
     right side of the word(case sensitive). The transition character is lowercase and
    the character after it is uppercase. If no such lowercase character is found, return
     an empty string.
    >>> get_closest_transition_character(”eAsy”) == ”s”
    """

[Example Solution Plan]
1. Reverse iterate through the characters of the word starting from the last character
from the right.
2. For each character, check if the current character is lowercase and if the character
 after it is uppercase.
3. If step 2 is satisfied,
4. return the current vowel character.
5. If no such vowel is found, return an empty string.

[Example Plan Verification for assert get_closest_transition_character(”eAsy”)==”s”]

[Record analysis]
The return value is the closest transition character, so the closest transition
character should be recorded!

1. Call the function get_closest_vowel(”eAsy”).
2. According to line 1 in the solution plan, reverse iterate the word, from the last
character to the first character, so the last character is ”y”
3. According to line 2 in the solution plan, ”y” is a lowercase but the character after
 ”y” is ”s” and ”s” is a lowercase.
4. Move to the next character based on the reverse iterate, so the character is ”s”.
5. According to line 2 in the solution plan, ”s” is a lowercase and the character after
 's' is 'A', and 'A' is uppercase.
6. According to line 3 in the solution plan, step 2 is satisfied,
7. [Record]: the closest transition character 's'
8. According to line 4 in the solution plan, return the current lowercase character 's'


[Example Generated Program]
[Start Program]

from typing import List
def get_closest_transition_character(word):
    """ You are given a word. Your task is to find the closest transition character from
     the right side of the word(case sensitive). The transition character is lowercase
    and the character after it is uppercase.
    >>> get_closest_transition_character(”eAsy”) == ”s”
    """
    # reverse iterate the word
    for i in range (len(word)-1,-1,-1):
        current_character=word[i]
        if current_character.islower():
            if i!=0:
                after_character=word[i-1]
                if after_character.isupper():
                    return current_character
    return ””

[End Program]
```

```
[End Example]

... Authors' notes: We omit another example for conciseness. The full prompt can be
found in our released code. ...

Finally, you will be provided with a Python writing problem, starting with [Problem
Description]. A solution plan will follow, beginning with [Solution Plan]. Next, you
will receive several plan verifications. For each test case X, the plan verification,
starting with [Plan Verification for X] provides detailed logical reasoning steps to
solve it.

Once the plan verification is provided, the ”Let's generate the program” flag indicates
 the start of Python program generation. You will then need to generate the Python
program solution for the problem. The plan verification serves as a constraint during
program generation. It is essential to ensure that the execution of the generated
program remains consistent with [Plan Verification for X] when using test case X as
input. Additionally, the generated program should incorporate all conditions noted in [
Plan Verification for X] to solve test case X. Please ONLY output the generated Python
program, starting with [Start Program] and ending with [End Program].
```

Listing 5: Prompt for print statement

```
======================================= System ========================================

You are a Python writing assistant that only responds with Python programs with PRINT
statements.

======================================== User =========================================

You'll be provided with several examples structured as follows, beginning with [Start
Example] and ending with [End Example]. In each example, you will be given a sample
Python program, starting with [Example Python Program]. You will also receive several
plan verifications for specific test cases. For a test case X, its plan verification,
starting with [Example Plan Verification for X], includes a worded description of the
logic used to solve test case X. During the verification, the intermediate variable
that needs to be tracked is clearly identified, starting with [Record Analysis] at the
beginning, and any updates to its value are recorded, starting with [Record].

Following this, you will be shown a Python program that includes detailed print
statements, starting with [Example Python Program with Print Statements]. These print
statements illustrate how the values of the intermediate variables (described in the
plan verification) are modified during program execution, as well as how other
variables in the program change. These examples will guide you on where and how to add
print statements in your Python program.

[Start Example]

[Example Python Program]
from typing import List
def get_closest_transition_character(word):
    ””” You are given a word. Your task is to find the closest transition character from
     the right side of the word(case sensitive). The transition character is lowercase
    and the character after it is uppercase.
    >>> get_closest_transition_character(”eAsy”) == ”s”
    ”””
    for i in range (len(word)-1,-1,-1):
        current_character=word[i]
        if current_character.islower():
            if i!=0:
                after_character=word[i-1]
                if after_character.isupper():
                    return current_character
    return ””

[Example Plan Verification for assert get_closest_transition_character(”eAsy”)==”s”]
[Record analysis]
The return value is the closest transition character, so the closest transition
character should be recorded!

1. Call the function get_closest_vowel(”eAsy”).
```

```
2. According to line 1 in the solution plan, reverse iterate the word, from the last
character to the first character, so the last character is ”y”
3. According to line 2 in the solution plan, ”y” is a lowercase but the character after
 ”y” is ”s” and ”s” is a lowercase.
4. Move to the next character based on the reverse iterate, so the character is ”s”.
5. According to line 2 in the solution plan, ”s” is a lowercase and the character after
 's' is 'A', and 'A' is uppercase.
6. According to line 3 in the solution plan, step 2 is satisfied,
7. [Record]: the closest transition character 's'
8. According to line 4 in the solution plan, return the current lowercase character 's'

[Example Python Program with Print Statements]
from typing import List
def get_closest_transition_character(word):
    ””” You are given a word. Your task is to find the closest transition character from
     the right side of the word(case sensitive). The transition character is lowercase
    and the character after it is uppercase.
    >>> get_closest_transition_character(”eAsy”) == ”s”
    ”””

    print(f”Reverse iterate the word {word}”)
    for i in range (len(word)-1,-1,-1):
        current_character=word[i]
        print(f”current character at index {i} is {word[i]}”)
        if current_character.islower():
            print(f”current character {word[i]} is lowercase”)
            if i!=0:
                print(f”There is a character after {word[i]}”)
                after_character=word[i-1]
                print(f”character after {word[i]} is {word[i-1]}”)
                if after_character.isupper():
                    print(f”character is {word[i-1]} is uppercase”)
                    print(f”[Record]: the closest transition character {word[i]}”)
                    print(f”Return the closest transition character {word[i]}”)
                    return current_character

    print(f”no such lowercase character is found, return an empty string”)
    return ””
[End Example]

... Authors' notes: We omit another example for conciseness. The full prompt can be
found in our released code. ...

Finally, you will be provided with a Python program, starting with [Python Program],
along with several plan verifications for specific test cases. For each test case X,
the plan verification, starting with [Plan Verification for X], includes a detailed
description of the logic used to solve test case X. In the plan verification, the
intermediate variables to be tracked are clearly analyzed at the beginning, starting
with [Record Analysis], and any updates to these variable values are recorded, starting
 with [Record].

The phrase ”Let's add print statements” signals the start of the process to incorporate
 print statements into the provided Python program. Your task is to add print
statements that track how the variables in the program change. Ensure that the
intermediate variable values (as outlined in the plan verification) are printed using
these statements. Output your program with print statements, starting with [Start
Program] and ending with [End Program].
```

Listing 6: Prompt for code explanation

```
======================================= System =======================================

You are a Python interpreter tasked with providing an explanation for each line of the
Python program.

======================================== User ========================================

You will be given a Python program, and your task is to generate a word-by-word
explanation describing the effect of each line in the program. You will be provided
with several examples, each beginning with [Start Example] and ending with [End Example
]. In each example, you will receive a Python programming problem, starting with [
Example Problem Description], along with an example Python program, marked as [Example
Python Program], which is generated to solve the given problem. Additionally, you will
be provided with an explanation of each line in the example Python program, starting
with [Example Explanation For Each Line].

[Start Example]
[Example Problem Description]
def encode(message):
    """
    Write a function that takes a message, and encodes in such a way that replaces all
    letters in the message with the letter that appears 1 place after of that letter in
    the english alphabet and then delete letter is a vowel.
    The last letter z is execluded in the message. Assume the input only containing
    letters.
    """

[Example Python Program]
def encode(message):
    encoded_message = ""
    for char in message:
        next_char = chr(ord(char) + 1)
        if next_char not in "aeiouAEIOU":
            encoded_message += next_char
    return encoded_message

[Example Explanation For Each Line]
The Python function encode is designed to manipulate a given string (referred to as a
message) by replacing the current letter with the letter that appears 1 place after it
in the English alphabet and then skipping the letter if it is a vowel:

Function Definition (def encode(message):): Defines a function named encode that
accepts one parameter, message. This parameter is intended to be a string that will be
processed to create an encoded message.

Initialize Encoded Message (encoded_message = ""): Initializes a variable
encoded_message as an empty string. This variable will store the encoded version of the
 message as characters are processed and approved for inclusion.

For Loop (for char in message:): Iterates over each character in the message. Each
character is processed individually.

Calculate Next Character (replacechar = chr(ord(char) + 1)): For each character in the
message, this line calculates its next character that appears 1 place after it in the
English alphabet. It converts the character to its ASCII value with ord(char),
increments this value by 1, and then converts it back to a character with chr().

Check if the resulting character is a Vowel (if replacechar in "aeiouAEIOU":): Check if
 the resulting character (replacechar) after incrementation is a vowel (either
uppercase or lowercase is checked here). If it is a vowel, the continue statement is
executed.

Add Character to Encoded Message (else: encoded_message += replacechar): If replacechar
 is not a vowel, it is appended to encoded_message. This builds up the final encoded
string with the modified characters.

Return Encoded Message (return encoded_message): After processing all characters in the
 original message, the function returns the fully encoded string which consists of all
non-vowel characters that are the successors of the original characters in the ASCII
sequence.
[End Example]

... Authors' notes: We omit another example for conciseness. The full prompt can be
found in our released code. ...

Finally, you will be presented with a problem description, starting with [Problem
Description], and your generated Python program, starting with [Python Program], which
is meant to solve the [Problem Description]. After this, the "Let's generate the
explanation" flag will signal the beginning of the explanation phase. Your task is to
generate a word-by-word explanation for each line in the Python program, following the
```

```
format shown in the previous examples. Please skip the explanation for any line that is
 a print statement. Output your explanation starting with [Start Explanation] and
ending with [End Explanation].
```

Listing 7: Prompt for error analysis

```
======================================= System ========================================

You are a logical reasoner. You will be provided with two logical reasoning processes:
[Plan Verification] and [Error Execution Trace]. Your task is to identify any errors in
 the [Error Execution Trace] by comparing it with the [Plan Verification].

======================================= System ========================================

You will be provided with several examples, each starting with [Start Example] and
ending with [End Example]. In each example, you will receive a Python programming
problem, starting with [Example Problem Description], along with an example of an
incorrect Python program, marked as [Example Error Program], generated for that problem.
 You will also be provided with a detailed execution trace of the error program on the
failed test case X, labeled as [Example Error Execution Trace for X], including the
intermediate variable values.

Additionally, you will be provided with an example of the correct logical reasoning
process, labeled as [Example Plan Verification for X]. This process outlines the
necessary steps to solve test case X accurately, including condition checks and
recording intermediate variable updates, starting with [Record]. Next, [Example
Discrepancy Analysis] provides a comparison between the Example Plan Verification and
the Example Error Execution Trace, highlighting output differences and identifying
where the Error Execution Trace deviates from correctness. Finally, [Example Error
Analysis] summarizes the errors identified in the [Example Discrepancy Analysis] and
proposes solutions to correct them.

[Start Example]
[Example Problem Description]
def is_palindrome(num):
    """
    check if a given integer is a palindrome.
    """

[Example Error Program]
def is_palindrome(num):
    num_str = str(abs(num))
    return num_str == num_str[::-1]

[Example Error Execution Trace for assert is_palindrome(-121)==False]
1. Convert the integer -121 to the string "121"
2. The integer string "121" is equal to the reversed string "121", the result is True
3. Return True

[Example Plan Verification for assert is_palindrome(-121)==False]
[Record analysis]
The return value is the checking result about a given integer is a palindrome, so the
checking result should be clearly recorded!

1. Call the function is_palindrome(-121).
2. change integer to string, it is "-121"
3. check whether the string "-121" is equal to its reversed string "121-", the checking
 result is False
4. [Record]: checking result = False
5. Return checking result False


[Example Discrepancy Analysis]
In the plan verification, the recorded value is the checking result:

Let's trace the "checking result" value in the plan verification when it is first-time
recorded (SKIP INITIALIZATION).

In the plan verification, the value of checking result is first-time recorded in Line 4
 after executing lines:
```

```
1. Call the function is_palindrome(-121).
2. change to integer to the string, it is "-121"
3. check whether the string "-121" is equal to its reversed string "121-", the checking
 result is False
4. [Record]: checking result = False

In the plan verification, the first-time update changes the checking result value to
False.

Let's trace the "checking result" value in the Error Execution Trace.
In Error Execution Trace, the value of checking result is first-time recorded in Line 2
 after executing lines
1. Convert the integer -121 to the string "121"
2. The integer string "121" is equal to the reversed string "121", the result is True

In Error Execution Trace, the first-time update changes the checking result value to
True.

The checking result value in the plan verification and Error Execution Trace are NOT
the same, due to False NOT equaling True when the checking result value is first
updated.

Let's carefully analyse the reason with step-by-step thinking:
In lines 1-4 in the plan verification, the integer -121 is first converted to the
string "-121". Then "-121" is compared with its reversed string "121-". "-121" is NOT
equaling "121-" so the result is False

In lines 1-2 in Error Execution Trace, the integer -121 is first converted to the
string "121". This is different from the plan verification where converting -121 to
string is "-121" rather than "121". Then "121" is compared with its reversed string
"121". "121" is equaling "121" so the result is True.

[Example Error Analysis]
The error execution trace incorrectly converts the negative integer to its negative
integer string. The negative signal is missed. For example, negative integer -121
should be converted to string "-121" but not "121. To fix this error, the negative
number must be considered and its negative sign should be contained when converted to
string. Such as negative integer -121 should be converted to string "-121".

[End Example]

... Authors' notes: We omit another example for conciseness. The full prompt can be
found in our released code. ...

Finally, you will be presented with a problem description, starting with [Problem
Description], along with your generated error program, starting with [Error Program],
which attempts to solve the [Problem Description]. You will also receive a detailed
execution trace, including intermediate variable values, for the failed test case X,
starting with [Error Execution Trace for X]. This trace is generated by the error
program. Additionally, you will be provided with a correct logical reasoning process,
labeled as [Plan Verification for X], which outlines the necessary steps to solve test
case X accurately, including condition checks and recording intermediate variable
updates, starting with [Record].

Following this, the "Let's do analysis" flag will indicate the start of the analysis
phase. Your task is to analyze where the [Error Execution Trace for X] deviates from
the [Plan Verification for X], as demonstrated in the examples. This analysis should be
 output starting with [Discrepancy Analysis]. Finally, you should provide a summary of
the errors identified in the [Discrepancy Analysis], including the reasons for these
mistakes (IN ENGLISH) and suggestions on how to correct them, starting with [Error
Analysis].
```

Listing 8: Prompt for code refinement

```
====================================== System ======================================

You are a Python program fixer. You need to correct an error Python program based on
the provided information.

======================================= User =======================================
You will receive several examples, each structured as follows, starting with [Start
Example] and ending with [End Example]. Within each example, you will find a Python
programming problem, beginning with [Example Problem Description], followed by an error
 program provided under [Example Error Program] for the given problem. You will then
receive an explanation for the error program, including a line-by-line explanation
starting with [Example Error Program Explanation].

Additionally, an error analysis will be provided, starting with [Example Error Analysis
], describing the issues in the error program and offering suggestions for refinement.
You will then be provided with the refined Python program under [Example Refined
Program], based on the error analysis. Following that, a refinement explanation,
starting with [Example Refinement Explanation], will be given to show which lines of
the program were changed and explain the reasons for those changes.

[Start Example]

[Example Problem Description]
def is_palindrome(num):
    """
    check if a given integer is a palindrome.
    """

[Example Error Program]
def is_palindrome(num):
    num_str = str(abs(num))
    return num_str == num_str[::-1]

[Example Error Program Explanation]
Function Definition (def is_palindrome(num):): This line defines a function named
is_palindrome that takes one parameter, num. This parameter is expected to be an
integer.

Convert Number to Absolute String (num_str = str(abs(num))): A variable num_str is
initialized with the absolute value of num converted to a string. The abs() function
removes the sign from num if it's negative, ensuring the palindrome check is based
solely on the digits.

Check Palindrome and Return (return num_str == num_str[::-1]): This line checks if the
string representation of num_str is the same forwards and backwards. It uses the
slicing technique [::-1] to reverse the string. If num_str is equal to its reversed
version, the function returns True, indicating the number is a palindrome. Otherwise,
it returns False.

[Example Error Analysis]
The error execution trace incorrectly converts the negative integer to its negative
integer string. The negative signal is missed. For example, negative integer -121
should be converted to string "-121" but not "121. To fix this error, the negative
number must be considered and its negative sign should be contained when converted to
string.


[Example Refined Program]
def is_palindrome(num):
    num_str = str(num)
    return num_str == num_str[::-1]

[Example Refinement Explanation]
Program line (num_str = str(abs(num))) is changed to (str(num)) to convert the negative
 integer to its negative integer string by deleting the abs function to keep the
negative representation as mentioned in the the error analysis. (str(num)) can
correctly convert negative integer -121 to string "-121".

[End Example]

... Authors' notes: We omit another example for conciseness. The full prompt can be
found in our released code. ...

You will be presented with a Python writing problem, starting with [Problem Description
]. The error program will be provided under [Error Program], followed by an explanation
```

of each line, starting with [Error Program Explanation]. You will then receive an error analysis, starting with [Error Analysis], which describes the issues in the error program and provides refinement suggestions.

The repair process will begin with the phrase "Let's correct the program." Based on the error analysis, generate the refined program. Output your refined program, starting with [Start Refined Program] and ending with [End Refined Program], ensuring that ONLY the Python code is included between these markers. Finally, provide a refinement explanation, starting with [Refinement Explanation], detailing how the program was modified to align with the error analysis.