Socratic Human Feedback (SoHF): Understanding Socratic Feedback Based Steering Strategies Used by Expert Programmers for Code-generation with LLMs

Anonymous ACL submission

Abstract

Large Language Models (LLMs) are increasingly used for generating code solutions, empowered by features like self-debugging and self-reflection. However, LLMs often struggle with complex programming problems without human guidance. This paper investigates the strategies employed by expert programmers to steer code-generating LLMs toward successful outcomes. Through a study involving experts using natural language to guide GPT-4, Gemini Ultra, and Claude Opus on highly difficult programming challenges, we frame our analysis using the "Socratic Feedback" paradigm for understanding effective steering strategies. By analyzing 30 conversational transcripts across all three models, we map observed feedback strategies to five stages of Socratic Questioning: *Definition, Elenhus, Maieutic, Dialectic, and Counter-factual reasoning*. We find evidence that by employing a combination of different Socratic feedback strategies across multiple turns, programmers successfully guided the models to solve 58% of the problems that the models initially failed to solve on their own.

1 Introduction

The rapid advancements in Large Language Models (LLMs) have revolutionized the field of natural language processing (NLP) and automated code generation. These powerful models, such as GPT-4 (OpenAI, 2023), Claude Opus (Anthropic, 2024) and Gemini Ultra (Gemini Team, 2024) have demonstrated remarkable capabilities in generating code snippets based on natural language prompts, significantly enhancing programmer productivity and transforming daily programming practices. However, despite their impressive performance, LLMs still face challenges when it comes to handling complex coding problems that require a deep understanding of the task (Yeadon et al., 2024), effective problem decomposition, and the nuanced application of algorithms and libraries within specific constraints.

Recent research has shed light on the self-debugging abilities of newer LLMs (Chen et al., 2023b), which enable them to iteratively analyze and refine generated code based on the outcomes of unit tests, mimicking the trial-and-error approach commonly employed by human programmers. While this self-debugging capability has shown promise, it is not without limitations. LLMs may struggle to accurately identify the root cause of code failures or generate effective feedback to guide subsequent code refinements, resulting in modest performance improvements when tackling complex programming tasks, such as certain LeetCode's medium and hard-level problems. However, with certain human feedback during the iterative analysis, we are able to find that we are able to successfully steer models into providing a successful solution. Thus, understanding how humans currently interact with these models and the category of steering strategies that lead to successful steering is essential for future Human-AI model interaction design.

Along this goal in this paper, we present an empirical study that explores how expert programmers can effectively steer SOTA LLMs, such as GPT-4, Gemini Ultra, and Claude Opus, to generate functionally correct code for programming problems that the models initially failed to solve independently. We focus on the Socratic feedback approach, a technique commonly used in argumentation and tutoring, where targeted questions or prompts are used to stimulate critical thinking and guide learners towards formulating their own solutions. This approach mirrors the dynamics of college programming tutoring sessions, with the instructor providing incremental feedback based on the learner's most recent attempt, while the learner engages in multiple rounds of debugging before seeking further guidance.

Our study, involving 8 expert programmers solving 30 problems across three modern LLMs GPT-4, Gemini Ultra, and Claude Opus provided a total of 90 conversational data points. Our study demonstrates that these modern LLMs can successfully solve originally failed competition-level programming problems with just a few rounds of human Socratic feedback. Furthermore, we reveal a set of Socratic feedback techniques employed by programmers to guide the LLM effectively. We also discuss the failed attempts for successful steering and discuss the challenges faced by programmers in steering LLMs for coding task.

1.1 Socratic Questioning

Socratic Questioning (Beversluis, 1974) is a method of inquiry and dialogue that involves asking a series of questions to explore complex ideas, stimulate critical thinking, and guide individuals towards their own understanding of a concept. This approach is based on the belief that knowledge cannot be simply imparted but must be discovered through a process of questioning and self-reflection.

Recent research has shown promising results in applying Socratic Questioning to interact with Large Language Models (LLMs) (Shridhar et al., 2022). For instance, Xu et al. proposed a self-directed Socratic questioning framework that encourages LLMs to recursively decompose complex reasoning problems into solvable sub-problems. Compared to other multi-turn prompting strategies such as few-shot learning or Chain of Thought (CoT) (Wei et al., 2022) prompting, Socratic Questioning offers several advantages. Few-shot learning relies on providing a limited number of examples to the language model to guide its output, while CoT prompting generates intermediate reasoning steps before arriving at the final answer, this leads to accumulation of error. In contrast, Socratic Questioning engages the language model in a dynamic, back-and-forth dialogue, allowing for a more adaptive and targeted exploration of the problem space. This interactive approach enables the model to break down complex problems into smaller, more manageable sub-problems, facilitating a deeper understanding of the task at hand.

Our research aims to address the following question: "What types of Socractic feedback are currently used by expert programmers to resolve errors produced by code-generating LLMs?" We hypothesize that there exist common sequences of steering behaviors, or "steering strategies," employed by programmers to guide LLMs in generating correct and efficient code. By uncovering these strategies, we seek to gain insights into the most effective ways to interact with code-generating LLMs and ultimately improve their performance in solving complex programming problems.

1.2 Categories of Socratic Questions:

Chang et al. (Chang, 2023) investigated the integration of various Socratic methods, such as definition, elenchus, and counterfactual reasoning, to develop effective prompt templates for tasks involving inductive, deductive, and abductive reasoning. However, to the best of our knowledge, no prior work has investigated the application of human Socratic feedback to enhance the code generation capabilities of LLMs. In order to better categories the different types of human feedback, we generalized the different strategies and have mapped them to various categories of questions from Socratic method.

- **Definition:** Use of definition to clarify and explain the meaning of key terms and concepts.
- **Elenchus:** This method involves cross-examination, where a series of questions is used to test the consistency and coherence of hypotheses and beliefs. Elenchus aims to test the validity of someone's arguments and to help them refine their thinking and eventually come up with well-supported hypotheses.
- **Maieutics:** This method involves helping individuals bring out the knowledge and understanding they already possess. Maieutics is conducted by asking questions that encourage the person to reflect on their own experience, knowledge, beliefs and to explore alternative perspectives. Maieutics fosters self-discovery, creative writing, and innovation.
- **Counterfactual Reasoning:** This method involves imagining alternative scenarios or "what-if" situations that are contrary to the facts of what actually occurred. It involves modifying prior events and then assessing the consequences of those alternative scenarios.
- **Dialectic:** This method involves exploring opposing view points through dialogue or debate to arrive at a deeper understanding of a subject.

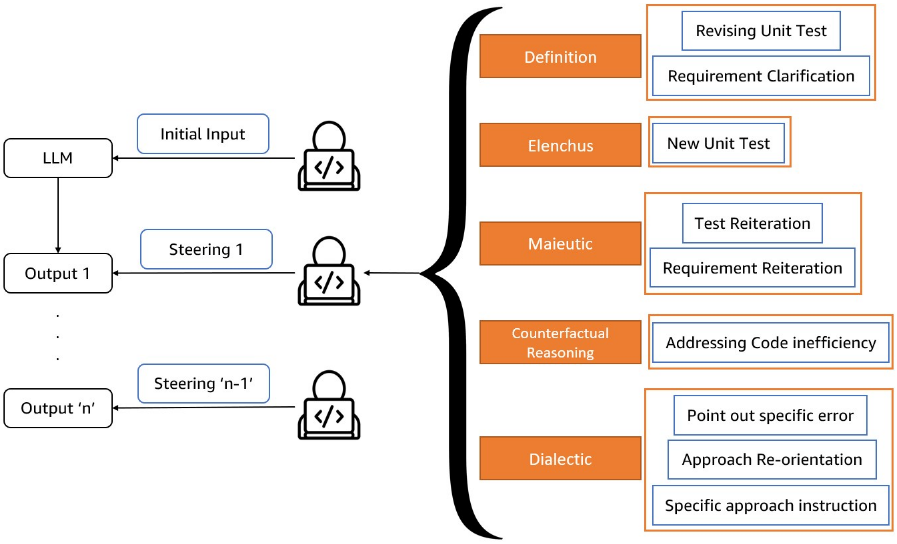


Figure 1: An overview of the study that was conducted to investigate effective steering strategies in code-generation LLMs. Users interact with the LLMs through multi-turn prompts, and various strategies that have been identified are categorized based on Socratic feedback presented on the right.

2 Related Work

Recent studies (OpenAI, 2023; Li et al., 2023b; Rozière et al., 2023; Chen et al., 2021) suggest that incorporating code into training data enables general-purpose LLMs to generate programs from natural language prompts or to complete incomplete code snippets. Alternatively, specialized models like Codex (Chen et al., 2021), AlphaCode (Li et al., 2022), StarCoder (Li et al., 2023b), and Code LLAMA (Rozière et al., 2023) have also been developed or fine-tuned specifically for coding tasks. Though they have achieved SOTA performance on code generation benchmarks (Zheng et al., 2023; Chen et al., 2021), LLMs still exhibit limited performance on medium and hard competition-level programming problems. These complex problems typically require a programmer's adept skills in understanding, planning, and implementing sophisticated reasoning tasks. Furthermore, approaches such as AlphaCode (Li et al., 2022) are impractical for real applications due to the dependency on available unit tests and extreme amounts of computational resources.

To address this limitation, some works used prompt-based techniques to boost LLMs' reasoning for correct code. For example, strategies like the chain-of-thought (Li et al., 2023a) and tree-of-thought (Yao et al., 2023) were employed to prompt models to break down the planning process into manageable intermediate subproblems. Additionally, self-debugging or reflection techniques (Chen et al., 2023b; Shinn et al., 2023; Madaan et al., 2023; Jiang et al., 2023) encouraged models to analyze their own outputs and divide the debugging process into stages of code explanation and self-feedback generation. Then LLMs refined their planning and execution grounded on the insights obtained from their self-generated feedback. Besides stimulating models' self-reflection, some works used human prompts to support the code refinement process. For example, Austin et al. (2021) explored human-model collaborative coding on MBPP dataset. They found that LLMs can improve or correct code based on human feedback, benefiting from human clarification of under-specified prompts and correction of small context errors. However, our focus diverges as we concentrate on tackle competition-level problems, which are notably more complex than those found in the MBPP dataset. Apart from incorporating human feedback as prompts, Chen et al. (2023a) improved CodeGen using imitation learning from human language feedback, where human feedback is used to learn a refinement model that generates modification from human feedback and previous incorrect code.

3 Methodology

To investigate the application of Socratic feedback in steering code-generating Large Language Models (LLMs), we conducted a study involving three state-of-the-art models: GPT-4, Gemini Ultra, and Claude Opus. The study focused on the models' ability to generate Python code solutions for algorithmic and data structure problems sourced from LeetCode, spanning various difficulty levels and topics. We randomly selected 223 problems from LeetCode and filtered them to identify instances where the models were unable to independently generate correct solutions. This filtering process yielded a set of 45 hard problems that the models failed to solve on their first attempt. These problems were chosen as the basis for our study among which 30 were solved due to availability of the programmers. We recruited 8 expert programmers to participate in the study. Each programmer was tasked with steering the three LLMs to solve the selected problems through successive conversational prompting, employing Socratic feedback techniques. The experts were first asked to solve or at least have an understanding on how to solve each problem on their own before starting to steer the model. The final code is considered a success if it passes all test cases provided in the initial problem description and the final solution was accepted by LeetCode. The code was tested and submitted manually to the LeetCode platform.

Programmers were provided with a prompt template that addressed key aspects of the problem, including the problem description, function signature, test cases, and constraints. They were also given a digital document containing task instructions and sample prompt templates to guide their interactions with the LLMs. Programmers engaged in an iterative prompting process, providing Socratic feedback to the LLMs based on the generated code's performance. They were instructed to continue the prompting process for a maximum of 10 iterations or until a correct solution was generated, whichever occurred first. The collected conversational data was analyzed to identify and categorize the various strategies employed by the programmers. These strategies were then mapped to corresponding Socratic feedback themes.

4 Steering strategies

After analyzing all 90 sub-tasks (3 models across 30 problems), we have identified various strategies employed by users in their interactions with the model. Figure 8 presents a snippet of the conversation with the model, and we will elucidate these different strategies using samples from these conversation snippets.

(A) ***Test Reiteration:*** When the provided code fails a unit test, users prompt the model by reiterating that one or more unit tests have not passed. In Figure 8A, users prompt the model about the failure of a specific unit test, successfully steering the model by reiterating the code.

(B) ***New Test Definition:*** If the model's provided code is partially or fully correct but less optimal solution, users refine it by introducing new unit test samples. In Figure 8B, the code produces the correct output but lacks optimization. Users, therefore, provide additional test cases, prompting the model to consider a different strategy and leading to successful steering.

(C) ***Revising Unit Test:*** Some users modify unit test conditions to add more constraints for the model to consider. In Figure 8C, the initial code does not yield the correct output, but by revising the unit test conditions, the user successfully steers the model output.

(D) ***Pointing out Specific Programmatic Error:*** Users identify specific errors by specifying the location and nature of the error in the output. For instance, in Figure 8D, the user points out an "index out of bound error" on line 17, successfully steering the model to fix the issue.

(E) ***Addressing Code Inefficiency:*** Users enhance program efficiency by requesting an alternative approach from the model. In Figure 8E, the user asks for an alternative approach due to the code's slow computation, leading to a more efficient solution.

(F) ***Requirement Reiteration:*** Similar to test case reiteration, users emphasize specific constraints if the model initially overlooks them. Figure 8F illustrates a successful code steer where the user reiterates the need for a particular requirement to be satisfied.

(G) ***Requirement Clarification:*** Users clarify requirements, as seen in Figure 8G, where the user specifies the range of an index that was unclear initially.

(H) ***Approach Re-orientation:*** Users reorient the model by suggesting an approach not considered previously. Figure 8H exemplifies a user providing a hint on a potential approach, leading to a successful code steering.

Figure 2: Examples of code steering strategies with model **(Left) New Test Definition (B); (Center) Pointing out Specific Programmatic Error (D); (Right) Requirement Clarification (G);** The modified portion of the code, crucial for achieving successful steering, is highlighted by the red boxes.

(I) ***Specific Approach Instruction:*** Finally, users provide a specific code block or instructions on how to solve a problem. In Figure 8I, the user offers a specific implementation approach along with a code block for an erroneous function, and the model successfully incorporates this input to solve the problem.

4.1 Aligning the Socratic Method to Human Feedback Strategies:

Although the Socratic method encompasses various question categories, not all were pertinent or observed in the empirical study. Figure 1 provides an overview of the identified categories, mapping them to general strategies observed, and includes a list of corresponding sample IDs exhibiting the associated strategy in the data.

The Socratic questioning method labeled "Definition" pertains to clarification, which could involve elucidating testing conditions, as seen in the "revising unit test" strategy, or specifying requirements. Elenchus involves cross-examining results to assess the consistency and coherence of arguments, essentially employing logical refutation, such as providing a testing condition (Strategy: New Unit test) to logically evaluate whether the condition satisfies the result.

Maieutic is a technique wherein ideas are tested to elicit existing knowledge and understanding positively. This mirrors how some test cases and requirements/constraints are reiterated to reveal the system's inherent knowledge. Counterfactual reasoning, involving the exploration of alternative perspectives, can be observed as users consider alternative options to enhance code efficiency.

Dialectic questioning is a systematic reasoning method that places opposed or contradictory ideas side by side, seeking to resolve their conflict. This is akin to a user pinpointing a specific error location or approach in a solution, where conflicting ideas between the previous prompt response and the user's input prompt overlap, leading to a successful resolution.

4.2 Multi-turn code steering:

Most interactions with the model involve multi-turn prompts, employing a sequence of inputs to guide the model towards a successful outcome. To illustrate this process, we examine a full specific example in Figure 3.

The initial prompt (Figure 3-1) presents a challenging programming problem categorized as "Hard" on LeetCode. The user's initial input comprehensively outlines the problem statement, provides examples, emphasizes constraints, and provides unit tests for validation. The user then instructs the model to articulate its understanding, outline a planned approach, and proceed to implement and test the code. However, the initial model

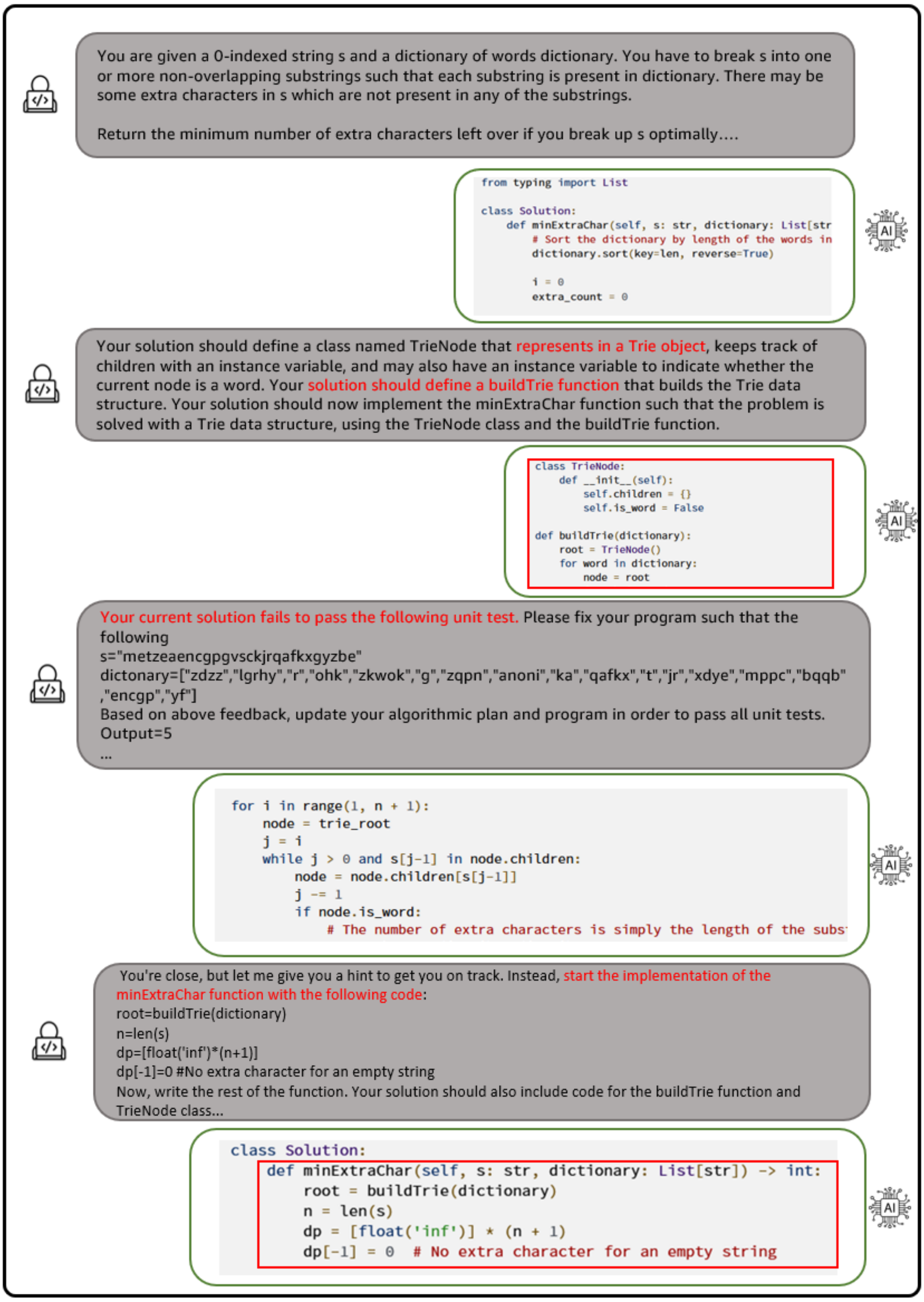


Figure 3: Example of successfully implemented multi-turn code steering.

response proves incorrect, lacking the appropriate solving approach.

In the user's first attempt to guide the model (Figure 3-2), they rectify the situation by offering the correct approach and reorienting the model toward the proper direction. Specifically, the user suggests using a specific data structure, such as the "Trie" data structure. The model incorporates this suggestion, updating its solution accordingly (highlighted in red in Figure 3-2). Although the revised output still fails certain unit tests, the user iterates on the failed test and prompts the model to address the issue by modifying its solution.

In this iteration, the model correctly identifies the problem with its approach, acknowledging it in the observation presented within its response plan. Furthermore, the model correctly recognizes that the appropriate approach is dynamic programming, proceeding to update its solution. However, this modified program still falls short due to an implementation error. In the user's third attempt to guide the model (Figure 3-4), they pinpoint the implementation issues and provide a code block to rectify them. The final response in Figure 3-4 indicates that this intervention successfully resolves all issues. The model incorporates the user-provided code block into its final implementation, resulting in a concise and clean solution.

5 Results & Discussion

A total of 90 conversations were recorded across the three models: GPT-4, Gemini Ultra, and Claude. The conversations comprised a total of 617 turns, during which programmers employed various steering strategies to guide the LLMs towards correct solutions. Among these conversations, 53 (58%) led to successful code generation after steering, with the model producing a solution that was accepted by LeetCode. GPT-4 had the highest success rate, with 26 out of 30 conversations resulting in correct solutions, followed by Claude Opus and Gemini Ultra with 15 and 12 respectively.

As shown in figure 5, the most commonly used strategy was "Point Out Specific Error," which was applied in 22% of the turns (136 out of 617). This strategy involved programmers identifying and highlighting specific errors in the code generated by the LLMs, prompting the models to rectify those issues. The second most frequently employed strategy was "Specific Approach Instruction," used in 18% of the turns (112 out of 617). In this approach, programmers provided the LLMs with specific guidelines, algorithms, or techniques to solve the problem at hand. By offering targeted guidance, programmers aimed to steer the models towards more efficient and effective solutions. Interestingly, "Revising Unit Test" and "Requirement Reiteration" were the least preferred strategies among the programmers, applied in only 4% and 5% of the turns, respectively. This suggests that programmers found it more effective to directly address the code generated by the LLMs, rather than modifying the test cases or restating the problem requirements. Other strategies employed by the programmers included "New Unit Test," used in 8% of the turns, and "Requirement Clarification," used in 13% of the turns. "New Unit Test" involved providing additional test cases to help the LLMs understand the problem better and cover edge cases, while "Requirement Clarification" focused on explaining the problem statement or constraints more clearly to the models. "Address Code Inefficiency" and "Test Re-iteration" were used in 9% and 13% of the turns, respectively. The former strategy aimed at guiding the LLMs to generate more efficient and optimized code, while the latter involved re-running the test cases to validate the correctness of the generated solutions.

The results of our study demonstrate the effectiveness of Socratic feedback in enabling expert programmers to steer code-generating Large Language Models (LLMs) towards correct solutions for complex programming problems. By employing a combination of strategies, with a focus on pointing out specific errors and providing targeted guidance, programmers successfully guided the models to overcome initial failures and generate code that met the required specifications. GPT-4 exhibited a higher success rate when prompted to self-debugging and self-reflection, while other models required more human feedback. This difference may be attributed to GPT-4's ability to run its own code, while Opus clearly states its current inability to execute its generated code.

An essential aspect of successful steering identified is the ability to identify the specific programming stage at which the model is struggling. Participants who provided relevant feedback to help the model overcome hurdles at different stages, such as understanding, planning, implementation, and testing, were more likely to achieve successful outcomes. Clear communication about misunderstandings or overlooked details proved to be crucial in guiding the LLMs towards the correct solution. In one example, clarifying a misunderstood problem condition led to successful steering, while in another case, overlooking a crucial detail resulted in a failed discourse. This finding emphasizes the importance of programmers being attentive to the specific challenges faced by the LLMs at each stage of the problem-solving process and providing targeted feedback to address those issues. It is unclear however, if novice programmers will have the same level of success similar to that of the experts in this study.

Figure 4: Steering success rates for the problems through steering after initially failed by the code-generating Large Language Models (LLMs). Green indicates the number of problems successfully steered, while red bars represent the number of problems that remained unsolved after 10 interactions.

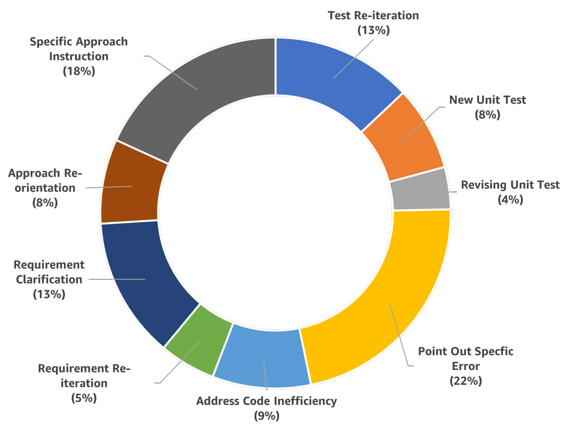


Figure 5: Percentage breakdown of the steering strategies employed by expert programmers to guide code-generating Large Language Models (LLMs)

6 Limitation & Future Work

Our study presents an initial investigation into the application of Socratic feedback in steering code-generating Large Language Models (LLMs). To establish an upper bound on the feasibility of LLM interaction, we focused our data collection on expert programmers. While the observed strategies demonstrated success across three different models, suggesting their generalizability in improving the models' problem-solving abilities through human steering, some strategies, such as "Point out specific error" or "Approach Re-orientation," may only be feasible for experts. Future research could conduct a longitudinal study involving novice users to determine if they can effectively employ the same strategies identified by experts and if their productivity improves with the understanding and application of Socratic feedback in their prompting techniques.

We acknowledge the limitations in the quantity of data points gathered for making larger generalized claims. However, this preliminary work provides valuable insights that can be expanded upon through more extensive data collection efforts in the future. One potential direction is to create a mapping between the different types of errors in the model feedback and the programmers' chosen strategies for steering the models. Such a mapping would be instrumental in designing future Human-LLM interfaces, enabling the model to recommend follow-up prompts, ask clarifying questions, or provide prompt templates that align with the Socratic feedback paradigm.

The findings from this study pave the way for future research to explore the dynamics of steering language models in code generation tasks further, enhancing our understanding of user challenges and optimizing human-AI collaboration. While our study participants employed various strategies, there is potential for developing and evaluating more sophisticated steering techniques. Future work could investigate the integration of machine learning or reinforcement learning approaches to dynamically adapt steering strategies based on the model's responsiveness and the evolving context of the conversation.

7 Conclusion

In this paper, we conducted an empirical study to investigate the use of Socratic feedback by expert programmers in steering code-generating Large Language Models (LLMs) to solve complex programming problems. By examining the interactions between programmers and three state-of-the-art LLMs - GPT-4, Gemini Ultra, and Claude Opus - we identified common strategies and feedback techniques employed to guide the models towards generating correct and efficient solutions. Our findings demonstrate that Socratic feedback plays a crucial role in enabling programmers to effectively steer LLMs when the models are unable to independently generate correct solutions. Our findings contribute to the growing body of research on human-AI interaction and provide valuable insights for the development of more effective collaboration techniques.

Acknowledgments

Blinded for double blinded reviews

References

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku.

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*.

John Beversluis. 1974. Socratic definition. *American Philosophical Quarterly*, 11(4):331–336.

Edward Y Chang. 2023. Prompting large language models with the socratic method. In *2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0351–0360. IEEE.

Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. 2023a. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749*.

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*.

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

Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. Self-evolve: A code evolution framework via large language models. *Preprint*, arXiv:2306.02907.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023a. Structured chain-of-thought prompting for code generation. *Preprint*, arXiv:2305.06599.

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

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. *Preprint*, arXiv:2303.17651.

OpenAI. 2023. Gpt-4 technical report. *ArXiv*, abs/2303.08774.

Baptiste Rozière, 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*.

Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*.

Kumar Shridhar, Jakub Macina, Mennatallah El-Assady, Tanmay Sinha, Manu Kapur, and Mrinmaya Sachan. 2022. Automatic generation of socratic subquestions for teaching math word problems. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4136–4149, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

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. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*.

Will Yeadon, Alex Peach, and Craig P. Testrow. 2024. A comparison of human, gpt-3.5, and gpt-4 performance in a university-level coding course. *Preprint*, arXiv:2403.16977.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, page 5673–5684, New York, NY, USA. Association for Computing Machinery.

A Example Appendix

A.1 Initial prompt Template provided:

You are given a function signature and description the programming tasks. Complete the function body that pass all the unit tests. Task description:

<Paste the problem task description here: include examples and constraints if available >

Your task:

First, describe your plan for solving this problem in natural language and then your implementation with a explanation of your code.

Take the following three stage approach in solving the problem:

1. Understand the problem. Ask any clarifying questions if you do not understand the problem well.

2. Please clearly describe how your would approach this problem.

3. When you describe your plan, please clarify what specific steps the algorithm includes and how you would implement them.

4. Provide your implementation code of your solution to the problem. Do not move to the next stage if you can't do the previous stage.

Then implement your plan in Python to solve this problem and make sure your algorithm passes all the given unit tests and comply with given constraints

A.2 Examples of Steering strategies

Figure 6: Examples of code steering strategies with model **(Left) Requirement Reiteration (F); (Right) Addressing Code Inefficiency (E);** The modified portion of the code, crucial for achieving successful steering, is highlighted by the red boxes.

Figure 7: Examples of code steering strategies with model **(Left) Test Reiteration (A); (Center) Requirement Clarification (G); (Right) Revising Unit Test (C).** The modified portion of the code, crucial for achieving successful steering, is highlighted by the red boxes.

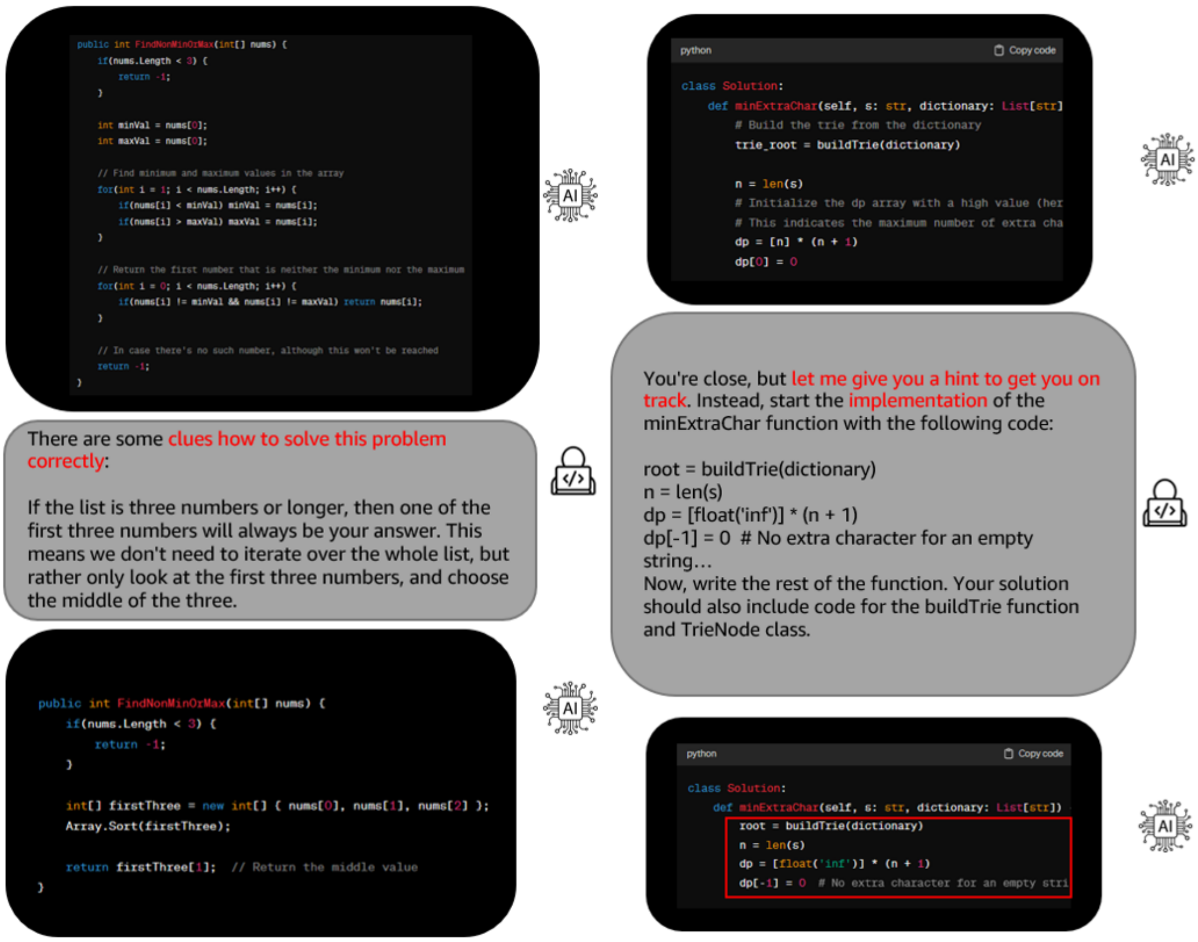
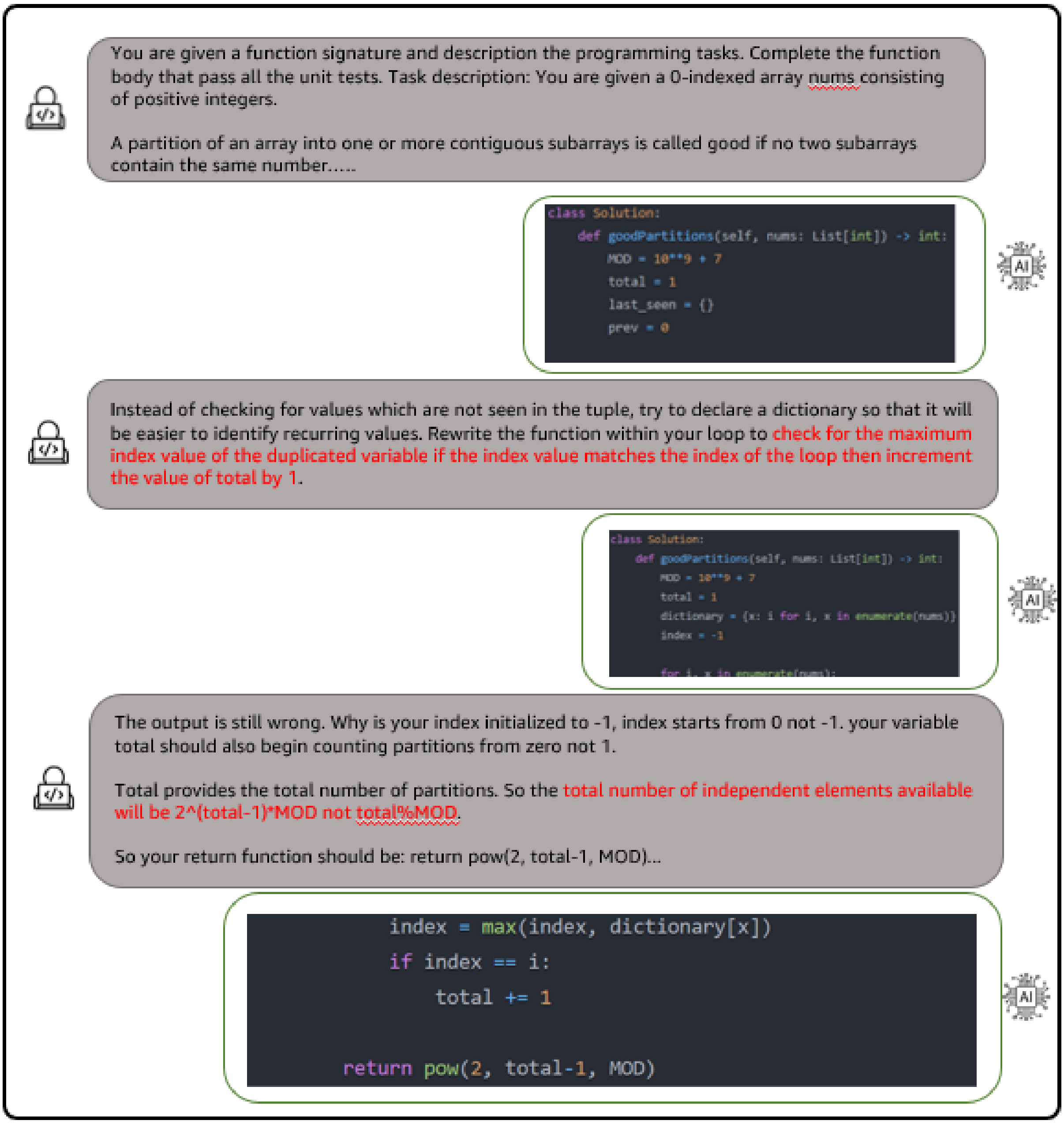
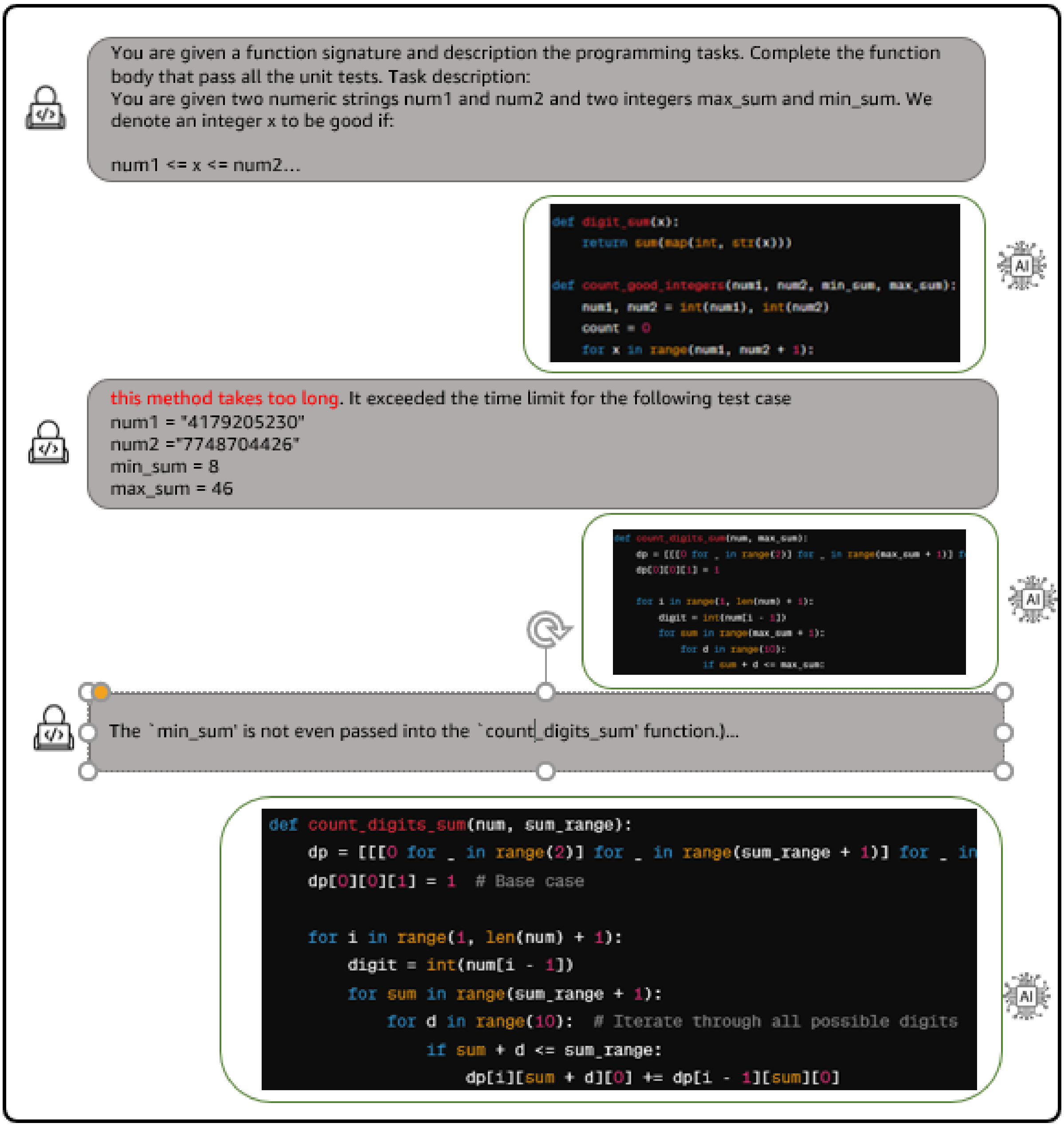


Figure 8: Examples of code steering strategies with model **(Left) Approach Re-orientation (H); (Right) Specific Approach Instruction (I).** The modified portion of the code, crucial for achieving successful steering, is highlighted by the red boxes.


You are given a function signature and description the programming tasks. Complete the function body that pass all the unit tests. Task description: You are given a 0-indexed array nums consisting of positive integers.
A partition of an array into one or more contiguous subarrays is called good if no two subarrays contain the same number.....
class Solution:
def goodPartitions(self, nums: List[int]) -> int:
MOD = 10**9 + 7
total = 1
last_seen = {}
prev = 0
Instead of checking for values which are not seen in the tuple, try to declare a dictionary so that it will be easier to identify recurring values. Rewrite the function within your loop to check for the maximum index value of the duplicated variable if the index value matches the index of the loop then increment the value of total by 1.
class Solution:
def goodPartitions(self, nums: List[int]) -> int:
MOD = 10**9 + 7
total = 1
dictionary = {x: i for i, x in enumerate(nums)}
index = -1
The output is still wrong. Why is your index initialized to -1, index starts from 0 not -1. your variable total should also begin counting partitions from zero not 1.
Total provides the total number of partitions. So the total number of independent elements available will be 2^(total-1)*MOD not total%MOD.
So your return function should be: return pow(2, total-1, MOD)...
index = max(index, dictionary[x])
if index == i:
total += 1
return pow(2, total-1, MOD)
AI


You are given a function signature and description the programming tasks. Complete the function body that pass all the unit tests. Task description:
You are given two numeric strings num1 and num2 and two integers max_sum and min_sum. We denote an integer x to be good if:
num1 <= x <= num2...
def digit_sum(x):
return sum(map(int, str(x)))
def count_good_integers(num1, num2, min_sum, max_sum):
num1, num2 = int(num1), int(num2)
count = 0
for x in range(num1, num2 + 1):
this method takes too long. It exceeded the time limit for the following test case
num1 = "4179205230"
num2 ="7748704426"
min_sum = 8
max_sum = 46
def count_digits_sum(num, max_sum):
dp = [[[0 for _ in range(2)] for _ in range(max_sum + 1)] f
dp[0][0][1] = 1
for i in range(1, len(num) + 1):
digit = int(num[i - 1])
for sum in range(max_sum + 1):
for d in range(10):
if sum + d <= max_sum:
The `min_sum' is not even passed into the `count_digits_sum' function.)...
def count_digits_sum(num, sum_range):
dp = [[[0 for _ in range(2)] for _ in range(sum_range + 1)] for _ in
dp[0][0][1] = 1 # Base case
for i in range(1, len(num) + 1):
digit = int(num[i - 1])
for sum in range(sum_range + 1):
for d in range(10): # Iterate through all possible digits
if sum + d <= sum_range:
dp[i][sum + d][0] += dp[i - 1][sum][0]
AI
AI
AI

You are given a function signature and description the programming tasks. Complete the function body that pass all the unit tests. Task description:
Given a m * n matrix seats that represent seats distributions in a classroom. If a seat is broken, it is denoted by '#' character otherwise it is denoted by a '.' character……
def maxStudents(seats):
m, n = len(seats), len(seats[0])
max_students = 0
for i in range(m):
for j in range(n):
if seats[i][j] == '.': # only consider open seats
check adjacent seats
Your current code does not consider actual placement of students, which changes the state of the seat map
def canPlace(i, j):
if seats[i][j] == '#':
return False
for x, y in [(i-1, j-1), (i-1, j+1), (i, j-1), (i, j+1)]:
if 0 <= x < m and 0 <= y < n and seats[x][y] == '.':
return False
return True
The backtracking algorithm is far too slow....
def maxStudents(seats):
m, n = len(seats), len(seats[0])
dp = [[0] * (n+1) for _ in range(m+1)]
for j in range(n-1, -1, -1):
...
This dynamic programming approach in the j index doesn't make sense. You are simultaneously looking at j-1 and j+1 at every step, but j+1 has not been computed yet when you are at j....
def maxStudents(seats):
m, n = len(seats), len(seats[0])
def countBits(num):
count = 0
while num:
count += num & 1
num >>= 1
return count
AI