Let's Code Step-by-Step: Conversational Question Generation for Coding Assistance

Priscilla Zhu qz2531@columbia.edu
Diya Nair dn2549@barnard.edu
Andrea Lopez apl2171@barnard.edu
Begum Gokmen bg2838@barnard.edu

1 Motivation & Goal

Competitive programming presents significant barriers to entry, especially for students from underrepresented backgrounds who lack the resources to prepare. While plenty of textbooks cover data structures and algorithms, students often rely on solutions from platforms like USACO or Codeforces when they feel stuck on problem-solving. However, seeing complete solutions at once discourages independent thinking - sometimes, a hint is all that's needed. In these cases, coaches play a crucial role by reviewing students' code and asking questions that inspire them to keep thinking without providing direct answers. We want to harness the reasoning ability of language models (LMs) to act as this "coach", prompting students with helpful questions at each step when they're stuck. This would provide AI-powered, personalized assistance for students without a coach, an expensive resource that's hard to access. Our project aims to use chain-of-thought prompting to obtain LLM-generated questions to guide students' algorithmic problem-solving. We believe our pipeline could be adapted to other fields like technical interview preparation or STEM problem-solving, benefiting a broader population in the future. We also aim to address concerns about LLMs in education by promoting guided problem-solving rather than giving away complete answers.

2 Research Hypothesis

We want to investigate: Are LLMs capable of asking helpful guiding questions to help a student solve a coding problem without giving away the answer? We will also explore how different prompts and different models may produce more helpful or less helpful responses.

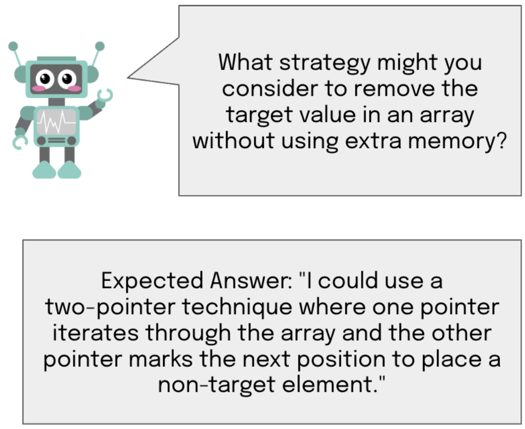


Figure 1: An example of an LLM-generated leading question designed to guide and inspire a student's problem-solving process, and its expected answer.

3 Methods

3.1 Manual Creation of Few-Shot Examples

We manually developed a small set of representative examples illustrating how an effective coding tutor should prompt students with guiding questions. These examples include clearly defined intermediate problem-solving steps and corresponding questions, fostering critical thinking without providing complete solutions outright. See Appendix A and B for a detailed demonstration.

3.2 TACO Dataset for Competitive Programming Problems

We use competition-level algorithmic coding problems from the TACO dataset (Li et al., 2023) as inputs to our LM. TACO is a diverse, large-scale code generation benchmark containing 25,433 coding problems in the training set and 1,000 in the test set. It integrates and deduplicates coding problems and solutions from major code generation benchmarks such as APPS (Hendrycks et al., 2021) and CodeContest (Li et al., 2022) and includes problems from reliable online coding education platforms such as Codeforces, CodeChef, and GeeksforGeeks.

On average, each problem has 58.21 ground truth code solutions and diverse test cases: the training set problems feature 51.6 test cases each, while the test set problems have 202.3 test cases on average. The diversity of solutions in the dataset allows LMs to optimize problem-solving approaches rather than defaulting to a single, "standard" correct solution.

Each entry in the TACO dataset includes challenge problems, solution code in Python, input-output examples, algorithmic skill labels, test cases, and time and memory constraints. The last two are particularly important, as they determine which algorithms are feasible; for example, an $O(n^2)$ algorithm would not satisfy a 1-second time limit if n exceeds 10^9. Additionally, the explicit algorithmic skill labels help the LM stay on track when generating leading questions, ensuring alignment with the specified requirements.

3.3 Dataset Generation via Gemini API

We will use the few-shot examples to perform chain-of-thought prompting through Gemini API. Specifically, we'll input competitive programming problems from the TACO dataset along with their solutions, instructing the model to decompose each solution into discrete, logical intermediate steps. Following this decomposition, we'll prompt GPT to generate insightful guiding questions at each identified step, explicitly pairing each question with its expected answer. This approach ensures coherence between problem-solving steps and instructional prompts.

3.4 Dataset Structure

Each entry in the dataset will include:

- Competitive programming problem statement
- Ground truth solution
- LLM-generated intermediate CoT step breakdown
- Guiding questions generated by LLM
- Expected answers to the guiding questions

3.5 Training an LLM for Guided Questioning

Using the generated dataset, we will train an open-source language model (e.g., DeepSeek, Ministral, etc.) specifically for the task of prompting guiding questions in competitive programming. The trained model will be optimized to reliably produce thoughtful and relevant guiding questions rather than complete code solutions.

3.6 Models

We plan to use open-source models from the Gemini family to perform leading question generations. Gemini 2.0 Flash is a state-of-the-art AI system engineered for rapid, real-time inference through innovative acceleration techniques and optimized model architectures. The model's free API also makes it accessible for research. We also want to explore the advanced reasoning abilities of DeepSeek LLMs for code understanding and CoT generation. DeepSeek's DeepThink model complements Gemini by employing advanced retrieval algorithms and multi-layered reasoning to deliver deep, context-rich insights for complex queries. Together, these platforms represent a new frontier in artificial intelligence—harmonizing high-speed performance with nuanced, in-depth analysis to enhance interactive and decision-making experiences.

4 Previous Research

4.1 CoT Prompting & Instruction Fine Tune

Recent work in large language models has explored how prompting techniques can be harnessed to enhance internal reasoning processes. In "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models" (Wei et al., 2022), the authors introduce a novel approach that encourages models to generate intermediate reasoning steps before arriving at a final answer. This method has been shown to significantly boost performance on complex multi-step tasks including arithmetic, commonsense, and symbolic reasoning. It decomposes problems into a sequence of manageable steps. Extensive experimentation on different benchmarks reveals that chain-of-thought prompting improves answer accuracy and increase transparency of the model's decision-making process.

4.2 LLM in Competitive Programming and Code Generation

LLMs have demonstrated promising capabilities in solving competition-level coding problems. Dumitran et al. evaluated the performance of several chat/instruct-type LLMs, including GPT-4 1106, Gemini 1.0 Pro, and DeepSeek Coder, on competitive programming problems from the Romanian Informatics Olympiad at the county level (Dumitran et al., 2024). They collected 304 challenge problems from 2002 to 2023, sourcing data from benchmarks such as TACO (Li et al., 2023) and RoCode (Cosma et al., 2024), the latter of which uniquely features problems written in Romanian.

GPT-4 achieved the highest performance, scoring 74.2% overall, while the second-best model, CodeStral 22B C++, achieved only 29.6%. The model we plan to use, DeepSeek Coder, attained 17.3% accuracy, indicating a reasonable level of competence in code generation and potential for code understanding. Notably, GPT-4 successfully solved 100% of the problems designed for 5th-grade students.

The authors highlighted GPT-4's potential in personalized learning, emphasizing its ability to identify students' strengths and weaknesses, provide instant feedback on problem-solving, and offer step-by-step explanations. Our project builds upon this educational potential by taking a novel approach—generating conversational leading questions to guide learning.

In 2022, researchers at DeepMind introduced AlphaCode, a code-generation system designed to solve competitive programming problems that require a deep understanding of algorithms and complex natural language instructions (Li et al., 2022). AlphaCode was pre-trained on 715.1 GB of open-source, filtered, and deduplicated code from GitHub, and fine-tuned on CodeContests—a dataset of competitive programming problems primarily scraped from Codeforces—created to help the model specialize in the competitive programming domain. The CodeContests training set contains 13,328 problems, with an average of 493.4 C++ solutions and 281.1 Python solutions per problem. AlphaCode achieved an average ranking in the top 54.3% of participants in programming competitions with more than 5,000 competitors, showing LLMs's strength in code generation and tackling competition-level problems.

4.3 LLM As Teaching Assistant / Personalized Learning Agents

Recent work has explored the use of LLMs in educational settings, particularly in programming education. The TAMIGO system (et. al., 2024) demonstrated how LLMs can support teaching assistants by generating viva questions, evaluating student answers, and providing feedback on code submissions in an advanced computing course. Their findings show that LLMs are effective at generating relevant and context-aware questions and feedback, although limitations such as occasional hallucinations and misalignment with grading rubrics remain.

While TAMIGO focuses on creating assignments for students and grading their answers, our work focuses on generating guiding questions to assist student learning during the problem-solving process. Unlike TAMIGO, which targets teaching assistant workflows, our approach focuses on using question generation as a tutoring strategy to help students solve coding problems without giving away the answers—an area that remains underexplored in existing research.

LLMs have also been studied in the context of supporting large-scale online courses. The AI-TA system (Hicke et al., 2023) presents a novel approach that leverages open-source LLMs from the LLaMA-2 family along with augmentation techniques such as Retrieval Augmented Generation (RAG), Supervised Fine-Tuning (SFT), and Direct Preference Optimization (DPO) to tackle the challenges of scalable question-answering in computing courses. Their approach demonstrates significant improvements in answer quality through extensive experimentation on a Piazza dataset, highlighting the effectiveness of context-aware, guided responses in reducing the human effort required for managing student queries. Unlike traditional systems that provide complete solutions, AI-TA focuses on generating guiding questions to help students navigate complex problem-solving processes without directly revealing the answers—thereby fostering independent learning and critical thinking.

4.4 Hint Generation in Intelligent Tutoring Systems

Recent work in the field has increasingly focused on the role of hints as a critical component for scaffolding student learning. Jangra et al. (Jangra et al., 2024) provide a comprehensive review of automatic hint generation research, bridging insights from education, cognitive science, and natural language processing. Their survey not only formalizes the hint generation task but also outlines a roadmap for future research, highlighting challenges such as personalization, multi-modal integration, and ethical considerations. Unlike traditional systems that often provide direct answers, their work emphasizes the importance of generating hints that effectively guide learners by linking new information to prior knowledge—an approach that holds promise for creating more adaptive and supportive tutoring environments.

5 Preliminary Results

We tested GPT o3-mini, Deep Seek Deep Think, Gemini 2.0 Flash, and Claude 3.7 Sonnet using the following prompt and coding problem:

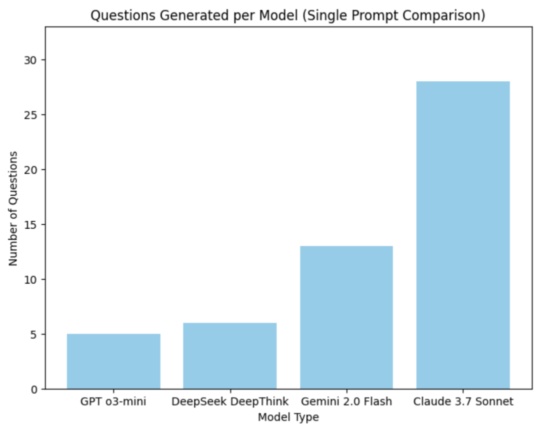
5.1 Prompt

> "For this question, we want to build an AI assistant that only asks directive questions and never gives the answer. If a student were solving this problem, what issues might they run into that the AI agent can help by asking questions? This should be an interactive Q&A based on what the student may be struggling with."

5.2 Coding Problem

> Given an integer array `nums` and an integer `val`, remove all occurrences of `val` in `nums` in-place. The order of the elements may be changed. Then return the number of elements in `nums` which are not equal to `val`.

5.3 Questions generated per model



We prompt four different LLMs with the same input: the prompt, a coding problem, and its ground-truth solution. Our goal is to evaluate each model's ability to break down the problem by measuring how many leading questions it generates. We hypothesize that there is a positive correlation between the number of questions generated and their quality—that is, whether the questions are detailed and specific enough to guide students toward the next step in solving the problem. Claude performs the best on this task, with Gemini coming in second. For GPT-3.5-mini and DeepThink—two models known for strong reasoning capabilities—we believe that more refined prompting could help elicit more detailed and effective breakdown questions. Overall, this experiment demonstrates Gemini's potential to serve as the model behind our conversational question-generation agent.

6 Results

We ran our custom code utilizing the Gemini API to automatically generate a dataset from the TACO coding problems. The code was designed to iterate over each sample from the TACO dataset and use a prompt containing few-shot examples. The Gemini API then returned a JSON object for each sample containing the following keys:

- **Coding Problem**
- **Ground Truth Solution**
- **LLM CoT Steps Breakdown**
- **LLM Questions**
- **Expected Answers to LLM Questions**

The complete custom dataset consists of one JSON object per coding problem. A sample entry from the generated dataset is shown here: https://docs.google.com/document/d/1QnsdzE4MAqc5–j55PsyeFDWWr8C3O3IPb7PY4VbBHM/edit?usp=sharing.

This sample demonstrates that each dataset entry includes not only the problem statement and ground truth solution but also a detailed chain-of-thought breakdown, targeted questions for further model analysis, and the expected answers. These elements collectively provide a comprehensive resource for evaluating the performance of language models on coding problems.

Contributions

- **Priscilla**: Dataset generation pipeline, motivation, TACO description, TACO code, overview of LLMs in competitive programming and code generation, prompt and example generation
- **Begum**: Methodology description, dataset structure, Gemini API implementation,

prompt and example generation, hint generation paper

- **Andrea**: Research hypothesis, prompt and example generation, summaries of papers on LLMs as teaching assistants
- **Diya**: Chain-of-thought (CoT) prompting paper, comparative model analysis, model descriptions, prompt and example generation

References

Adrian Cosma, Bogdan Iordache, and Paolo Rosso. 2024. Rocode: A dataset for measuring code intelligence from problem definitions in romanian. *arXiv preprint arXiv:2402.13222*.

Adrian Marius Dumitran, Adrian Cătălin Badea, and Ștefan Gabriel Muscalu. 2024. Evaluation of large language models in solving romanian olympiad-level competitive programming problems. *arXiv preprint arXiv:2409.09054*.

Anishka et. al. 2024. Tamigo: Empowering teaching assistants using llm-assisted viva and code assessment in an advanced computing class.

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. *arXiv preprint arXiv:2105.09938*.

Yann Hicke, Anmol Agarwal, Qianou Ma, and Paul Denny. 2023. Ai-ta: Towards an intelligent question-answer teaching assistant using open-source llms. *arXiv preprint arXiv:2311.02775*.

Anubhav Jangra, Jamshid Mozafari, Adam Jatowt, and Smaranda Muresan. 2024. Navigating the landscape of hint generation research: From the past to the future. *arXiv preprint arXiv:2404.04728*.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

Jason Wei, Xi Wang, David Schuurmans, Maarten Bosma, Eric H. Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

A Example Prompts

- Here is a coding problem and its solution. Please ask me questions to help me work through the problem step by step, without giving away the answer. Make sure to only give guiding questions rather than hints or solutions. Make sure that this is conversational.
- This is a coding problem and here's its solution. I don't want you to give me answers. Please break down the solution into steps, and for each step, ask me a question that could help me think and get to the answer! Don't tell me the steps, just tell me the questions!
- For this question, we want to build an AI assistant that only asks directive questions and never gives the answer. If a student were solving this problem, what issues might they run into that the AI agent can help by asking questions? This should be an interactive Q&A based on what the student may be struggling with.

B Example Conversation

Coding Problem:

Given an integer array nums and an integer val, remove all occurrences of val in nums in-place. The order of the elements may be changed. Then return the number of elements in nums which are not equal to val.

Ground Truth Solution:

```
class Solution:
    def removeElement(self, nums: List[int], val: int) -> int:
        k = 0  # Pointer to track valid elements
        for i in range(len(nums)):
            if nums[i] != val:
                nums[k] = nums[i]  # Move valid element forward
                k += 1
        return k  # New length of array
```

LLM CoT Steps Breakdown:

- Understanding Problem Constraints: Identify that the array must be modified in place. Recognize that the order of elements can be changed. Understand the need to return the count of elements that are not equal to the given value.
- Identifying Potential Pitfalls: Consider edge cases such as an empty array or an array where all elements are the target value. Acknowledge issues related to pointer/index management during in-place modifications.
- Selecting an Algorithmic Strategy: Think about how a two-pointer approach might help iterate and reorganize the array. Determine how to count non-target elements as you modify the array.
- Implementing In-Place Modification: Explore how to overwrite or swap elements without additional storage. Focus on ensuring that each valid element is moved to the correct position.

LLM Questions/Expected Answers:

- LLM: "What does it mean to modify the array in place, and why is this important for the problem?"
- Expected Answer: "It means updating the existing array without allocating extra space for another array, which is crucial for meeting the problem's in-place requirement."
- LLM: "How would you handle edge cases, such as when the array is empty or when every element is the target value?"

- Expected Answer: "If the array is empty, there's nothing to remove, so the result should be 0. Similarly, if every element equals the target, all should be removed, resulting in a count of 0."
- LLM: "What strategy might you consider to remove the target value without using extra memory?"
- Expected Answer: "I could use a two-pointer technique where one pointer iterates through the array and the other pointer marks the next position to place a non-target element."
- LLM: "How can you keep track of the number of non-target elements as you traverse the array?"
- Expected Answer: "I can use the second pointer as a counter, incrementing it each time a non-target element is placed into the array."
- LLM: "Given that the order of elements can be changed, how might this simplify your solution?"
- Expected Answer: "Since order doesn't matter, I can swap target elements with the last element or overwrite them, which may reduce the number of moves needed and simplify the algorithm."

C Imagined Scenario of a Student Asking Questions

- **User:** I don't know where to start. How do I figure out the most frequent elements?
- **LLM:** If you were given a list of numbers, how would you figure out the k most frequent elements, without coding?
- **User:** I would count each number and keep track of the count for each number. Then, I would look at all the counts, find the k highest counts, and those would be the k most frequent numbers.
- **LLM:** In code, what data structure can you use to keep track of the count for each number? (What's the best way to store key-value pairs?)
- **User:** A hashmap.
- **LLM:** How do you populate the hashmap with numbers and their counts?
- **User:** I iterate through the array. For each number, I increment the count.
- **LLM:** If the number doesn't already exist in the hashmap, what should you do?
- **User:** I will create a new key in the hashmap for the number and set its count to 1.
- **LLM:** You just populated the hashmap. What's the next step?
- **User:** I need to find the most frequent ones.
- **LLM:** How do you determine which numbers are the most frequent?
- **User:** I sort the key-value pairs based on the counts.
- **User:** I don't know.
- **LLM:** How do you find the most frequent element?
- **User:** I can find the element with the maximum count.
- **LLM:** How can we apply that concept to find the k most frequent elements?
- **User:** I can find the maximum, remove it, and find the maximum again, repeating this process k times.

- **LLM:** How can you sort the key-value pairs? Do you need to convert the hashmap into a new data structure?
- **User:** Yes, I can't sort a hashmap directly, so I need to convert it to an array of key-value pairs in order to sort it.
- **LLM:** Should the array be sorted by keys or by values?
- **User:** By values, because we want the highest counts.
- **LLM:** Now that you have your sorted array, how can you get your final answer of the k most frequent elements?
- **User:** I sorted the array from highest to lowest count, so I can take the first k pairs and return a list of their keys.