Consistent Coding Problem Synthesis with Reflective Analysis

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have shown great promise in educational applications, e.g., generating coding exercises for programming instruction. However, two major challenges remain in automatic coding exercise synthesis: (1) the generated solution code often fails to pass all test cases, and (2) there is no automatic metric to assess the conceptual relevance or pedagogical quality of the synthesized problems. In this paper, we present a three-stage framework for educational coding problem synthesis. First, we perform Chain-of-Thought-based Reflective Analysis, incorporating Error Analysis and Concept Analysis, to improve the pedagogical quality of generation. Second, we introduce an iterative code refinement to ensure the generated solution passes a Code Check. Third, we propose a Concept Check procedure to automatically evaluate the conceptual alignment between the input and the synthesized problem. Experiments show that our methods significantly improve both code and concept consistency, providing a reliable pipeline for automatic coding problem synthesis.

1 Introduction

Large Language Models (LLMs) demonstrate significant potential in educational applications. For instance, they can perform teaching-assistant-like question answering (Hicke et al., 2023) and provide multi-modal feedback on diagrams (Li et al., 2024a; Jurenka et al., 2024). Prior research has explored automatic distractor generation for multiple-choice questions in math and coding (Scarlatos et al., 2024; Lee et al., 2025), while language learning studies have personalized exercises using knowledge states and difficulty levels (Cui and Sachan, 2023).

Given LLMs' proficiency in solving programming and math problems (Chen et al., 2021; Jain et al., 2024; Li et al., 2024b), synthesizing high-quality, adaptive coding problems emerges as a valuable yet under-explored area (Chen et al., 2024; Fan et al., 2023). This task involves generating problem statements, solution code, and test cases (Frankford et al., 2024), with two critical challenges: code consistency and concept consistency.

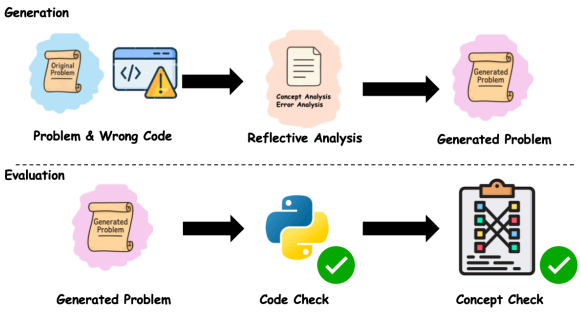

Figure 1: Our Generation-Evaluation framework

During generation, ensuring consistency between the solution code and its associated test cases is critical. If the reference solution fails to pass all provided test cases, revisions to either the code or the test cases are required. Prior work has shown that the pass rate declines as problem difficulty increases (TA et al., 2023). New benchmarks like TestCase-Eval also highlight the significance of this code consistency problem (Yang et al., 2025b). Therefore, we define **code consistency** as: solution code passing all test cases.

Beyond code consistency, quality evaluation of generated problems presents another major bottleneck. Existing quality assessments rely on costly expert annotations (Sarsa et al., 2022), which are costly and infeasible at scale. Moreover, such evaluations offer only coarse-grained judgments and are insensitive to learner skill levels. ExGen introduced the difficulty level as a criterion for the synthesized problem (TA et al., 2023). Therefore, we focus on **concept consistency** to evaluate the quality of synthesized coding problems.

In this paper, we propose a three-step method to improve consistency and pedagogical quality in coding problem generation (Figure 2). In the first

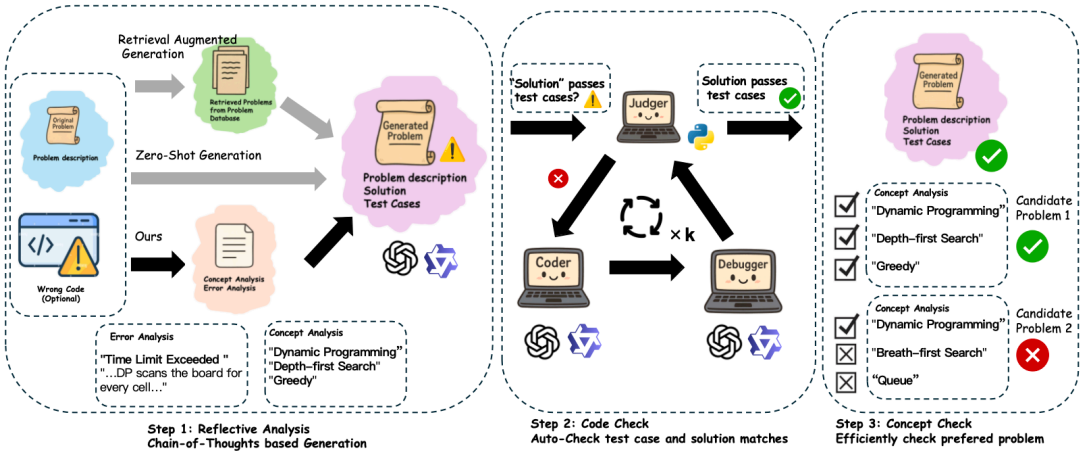

Figure 2: Three-stages pipeline for consistent problem synthesis. Reflective analysis synthesizes the problem by analyzing the concept or the error code before generation. Code Check requires repeatedly refining the code to achieve consistency. Concept Check requires the generated problem to conceptually match the input problem.

step, Reflective Analysis is applied to guide the LLM toward more accurate and instructional generation. Reflective Analysis aims to prompt the LLM with both Concept Analysis and Error Analysis tasks, encouraging deeper reasoning over the input problem before generating new exercises. In the second step, Code Check is performed to refine the inconsistency between the generated code and the test cases. We adapt a multi-agent framework to iteratively improve the correctness of synthesized code. In the final step, we introduce a Concept Matching metric to evaluate the similarity in coding concept between the input problem and the synthesized problem.

Our experimental results show that the proposed framework improves both concept consistency and code consistency. The Concept Matching of the synthesized problem increased by around 20% - 30% in Precision and Recall. The Pass All Rate of the synthesized solution code and test cases increases by 20%–30% compared to baselines, with enhanced conceptual consistency. This result accords with the code similarity increase, demonstrating the effectiveness of our metrics.

2 Methods

2.1 Concept Consistency

We propose Chain-of-Thought-based Reflective Analysis to enhance the concept consistency of the generated problems. As shown in Figure 2, Reflective Analysis consists of two types of analyses:

Concept Analysis. Given an input problem, we prompt the LLM to identify a list of high-level algorithmic concepts (e.g., “dynamic programming”, “depth-first search”, “greedy”). For each concept, the model is further instructed to explain how it could be applied to solve the problem. A new problem is then generated using this concept-aware reasoning as context, promoting conceptual alignment between the original and synthesized tasks.

Error Analysis. To identify unmastered concepts, we utilize erroneous code submissions as auxiliary signals. Given an input problem and an incorrect solution, the LLM is prompted to analyze the underlying cause of the error. The resulting analysis is then used to guide problem generation, encouraging the model to focus on algorithmic reasoning rather than surface-level text similarity. Further details and prompt examples for both analyses are provided in Appendix A.

To evaluate the concept consistency between the input problem and the generated problem, we propose the *Concept Matching* metric, which compares the sets of concepts extracted from both problems and quantitatively measures their alignment. The detailed formulation is introduced in Section 3.3, and we discuss the effectiveness of this metric in Section 4.1.

2.2 Code Consistency

Prior studies have investigated code-test consistency as a proxy for problem validity, primarily by reporting the percentage of problems where the sample solution passes all provided test cases (Sarsa et al., 2022; TA et al., 2023). However, these works typically stop at evaluation and do not propose methods for repairing inconsistent problems.

Iterative and self-reflective generation is commonly used in generation tasks. In the domain of code generation, multi-round refinement is also widely studied. CodeTree Li et al., 2024c proposes a tree search strategy that models the trajectory of reasoning, solving, and debugging. Socratic Human Feedback (SoHF) (Chidambaram et al., 2024) leverages Socratic questioning to elicit informative human feedback during multi-turn generation. MapCoder (Islam et al., 2024) and CodeSim (Islam et al., 2025) introduce agents with different tasks to enhance code generation.

We designed a multi-agent system that enables iterative refinement of codes and test cases, as shown in Figure 2, step 2. In this system, we have three agents: a Coder agent (LLM agent), a Judger (Python script running in local environment), and a Debugger (LLM agent).

After the code is generated, the Judger executes the code through test cases in a local environment and verifies the results. If the code does not match the test cases, the Debugger then provides revision feedback to the Coder to re-generate a new solution code. In case of the test cases not being consistent with the problem, when we found the sample code failed on $1-2$ same test cases every time, we will consider the test cases being wrong and prompt the LLM to re-generate these test cases. The iteration may loop multiple times, and we use Code Check *Pass@k* to represent a k-times iteration in table 1.

3 Experiments

3.1 Experimental Setup

We conduct experiments to evaluate the effectiveness of Error Analysis, Concept Analysis, and code refinement in improving the quality of coding problem synthesis. Our focus is on assessing whether our method improves the concept and code consistency of generated problems.

Baseline In all scenarios, we incorporate one-shot prompting as a baseline. Our prompts are shown in appendix G. For the scenario w/o error code submission, we also use retrieval augmented generation (RAG) (Wang et al., 2025) as a baseline. In RAG, we use the input problem to search for a similar problem in the dataset, with cosine-similarity by its vector embedding, and we prompt the LLM with the retrieved problem. For the embedding model, we used an open-source model in Huggingface (E): thenlper/gte-large.

Model Choice We conduct our experiment on both a closed-source model, OpenAI's GPT-4.1 (version released on April 14, 2025)—-and an open-source model Qwen3-235B (Yang et al., 2025a).

3.2 Dataset

CodeChef we use a dataset of 26,663 code submissions to 352 programming problems from CodeChef. We sample 999 incorrect submissions, proportionally distributed across three major error types: Wrong Answer, Time Limit Exceeded, and Runtime Error. To ensure broader coverage, we cap the number of submissions per problem at five, resulting in 279 unique problems and an average of 3.58 submissions per problem. These examples provide diverse failure cases for error analysis. Full statistics are shown in Table 2. In the experiment, we used the first 100 problems with submissions.

3.3 Metrics

Concept Matching To evaluate concept consistency, we define a new metric: Concept Matching. We take the concepts for the input problem as the ground truth, and take the concepts for the synthesized problem as the prediction. We use the precision and recall for the appearance indicators of the method keywords as our metric.

Code Check To evaluate code consistency, we adopt the percentage of synthesized problems whose solution code passed all their test cases. The solution code and the test cases are synthesized separately; therefore, the Pass All metric reflects how the synthesis process is consistent with itself. Here, the Pass@k means the percentage of passed problems with k iterations of code refinement.

Code Similarity We use the code similarity between the solution code from the input problem and the synthesized problem. This serves as a quality that measures whether the synthesized problem tests the same set of knowledge as the input problem. We use an embedding-based code similarity model from Huggingface (E): intfloat/e5-mistral-7b-instruct (Wang et al., 2023, 2022).

4 Results

We report results across three metrics: Concept Matching, Code Check, and Code Similarity. We examine the impact of Error Analysis (**EA**), Concept Analysis (**CA**), and their combination (**EA+CA**).

Method	Concept Matching ↑ Precision	Recall	Code Check ↑ Pass@0/Pass@3	Code Similarity ↑ Pass@0/Pass@3
GPT-4.1				
Direct Prompt	0.29	0.30	8%/23%	0.79/0.82
RAG	0.35	0.34	12%/33%	0.78/0.83
Error Code	0.54	0.53	15%/25%	0.80/0.84
Error Code + **EA**	0.65	0.66	14%/29%	**0.84**/0.85
Error Code + **CA**	**0.77**	**0.75**	**18%**/32%	0.79/0.83
Error Code + **EA+CA**	0.67	0.71	17%/**35%**	0.83/**0.87**
Qwen3				
Direct Prompt	0.36	0.38	15%/23%	0.77/0.78
RAG	0.42	0.49	16%/22%	0.76/0.77
Error Code	0.42	0.42	14%/25%	0.82/0.82
Error Code + **EA**	**0.68**	**0.67**	**25%/30%**	**0.86/0.85**
Error Code + **CA**	0.60	0.69	20%/28%	0.78/0.79
Error Code + **EA+CA**	0.63	0.60	22%/28%	0.84/0.83

Table 1: Main results on GPT-4.1 and Qwen3. We compare three types of inputs: raw prompts, RAG, and error code. Our proposed reasoning modules, EA (Error Analysis) and CA (Concept Analysis), are applied on top of error code inputs and yield consistent improvements across all metrics.

4.1 Concept Matching

As shown in Table 1, adding EA and/or CA increases both precision and recall as compared to baseline in both models. For Precision, GPT-4.1 achieved 0.54 with Error Code as context, but increased by 42.5% with Concept Analysis. Similarly, Qwen3 achieved at most 61% improvement in Precision utilizing Error Analysis. Recall shows a similar trend to Precision. This demonstrates the effectiveness of our Reflective Analysis in improving concept consistency.

4.2 Code Check

As shown in Table 1, multi-round code refinement generally improves the Pass All rate by 5%-15%. For GPT4.1, the Pass All rate increased by at most 20% with EA, CA, and 3-round code refinement. For Qwen3, adding EA and/or CA can also improve the Pass All rate by around 5%. Interestingly, inputting EA only surpasses inputting EA and CA together in Qwen3, which might be caused by generation stability in handling long context. Code Check result also accords roughly with the code similarity. Therefore, our pipeline can effectively improve the code consistency in generation.

5 Conclusion

We study the consistency of coding problem synthesis with LLMs at the generation and evaluation stages. In the generation stage, we propose Reflective Analysis methods to augment the concept consistency between a reference problem and its generated counterpart, based on the analysis of concepts and error code. We further introduce a multi-agent framework to improve the code consistency within the synthesized code and test cases. In the evaluation stage, we conduct Code Check and Concept Check to automatically evaluate the code consistency and the concept consistency. We conduct experiments on a proprietary and an open-source LLM with case analysis. The results demonstrated that our methods improve the generation consistency and evaluation efficiency of coding problem synthesis.

Limitations

Test Case generalization Code generation is currently explored more than test case generation. We consider revising the code in a loop with the assumption that the test case is correct. If the code can not pass all test cases, this can also indicate that certain test cases are incompatible. In some cases, both the test cases and the code may not correctly describe the problem, making it hard to diagnose.

LLM restriction Due to the use of LLM, we can not restrict the method name with a pre-defined category set. This might include some unwanted method names that might influence the accuracy of Concept Matching.

Efficiency consideration for Multi-agent In our paper, we constructed a multi-agent system to revise the code in a loop. On one hand, this might be inefficient in time for larger-scale tasks. On the other hand, to maintain the complete history of the multi-agent generation might be costly in terms of token length.

Ethics Statement

We release our code under the MIT license. The experiments use the intfloat/e5-mistral-7b-instruct model, available on Hugging Face under a CreativeML license. The benchmark data is based on the Codeforces problem set, which is publicly accessible for research and educational use. We do not observe significant ethical issues induced by our methods.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Mingda Chen, Xilun Chen, and Wen tau Yih. 2024. Few-shot data synthesis for open domain multi-hop question answering.

Subramanian Chidambaram, Li Erran Li, Min Bai, Xiaopeng Li, Kaixiang Lin, Xiong Zhou, and Alex C. Williams. 2024. Socratic human feedback (SoHF): Expert steering strategies for LLM code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 15491–15502, Miami, Florida, USA. Association for Computational Linguistics.

Peng Cui and Mrinmaya Sachan. 2023. Adaptive and personalized exercise generation for online language learning.

Aysa Xuemo Fan, Ranran Haoran Zhang, Luc Paquette, and Rui Zhang. 2023. Exploring the potential of large language models in generating code-tracing questions for introductory programming courses.

Eduard Frankford, Ingo Höhn, Clemens Sauerwein, and Ruth Breu. 2024. A survey study on the state of the art of programming exercise generation using large language models.

Yann Hicke, Anmol Agarwal, Qianou Ma, and Paul Denny. 2023. Ai-ta: Towards an intelligent question-answer teaching assistant using open-source llms. Accessed: 06 May 2025.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2025. Codesim: Multi-agent code generation and problem solving through simulation-driven planning and debugging.

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.

Irina Jurenka, Markus Kunesch, Kevin R McKee, Daniel Gillick, Shaojian Zhu, Sara Wiltberger, Shubham Milind Phal, Katherine Hermann, Daniel Kasenberg, Avishkar Bhoopchand, et al. 2024. Towards responsible development of generative ai for education: An evaluation-driven approach. *arXiv preprint arXiv:2407.12687*.

Yooseop Lee, Suin Kim, and Yohan Jo. 2025. Generating plausible distractors for multiple-choice questions via student choice prediction.

Hai Li, Chenglu Li, Wanli Xing, Sami Baral, and Neil Heffernan. 2024a. Automated feedback for student math responses based on multi-modality and fine-tuning. In *Proceedings of the 14th Learning Analytics and Knowledge Conference*, LAK '24, page 763–770, New York, NY, USA. Association for Computing Machinery.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, Yongbin Li, Bin Gu, and Mengfei Yang. 2024b. DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3603–3614, Bangkok, Thailand. Association for Computational Linguistics.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024c. Codetree: Agent-guided tree search for code generation with large language models.

Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, pages 27–43. ACM.

Alexander Scarlatos, Wanyong Feng, Digory Smith, Simon Woodhead, and Andrew Lan. 2024. Improving automated distractor generation for math multiple-choice questions with overgenerate-and-rank.

Nguyen Binh Duong TA, Hua Gia Phuc Nguyen, and Swapna Gottipati. 2023. Exgen: Ready-to-use exercise generation in introductory programming courses. In *Proceedings of the International Conference on Computers in Education (ICCE)*. Accessed May 2025.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. 2022. Text embeddings by weakly-supervised contrastive pre-training. *arXiv preprint arXiv:2212.03533*.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023. Improving text embeddings with large language models. *arXiv preprint arXiv:2401.00368*.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. CodeRAG-bench: Can retrieval augment code generation? In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3199–3214, Albuquerque, New Mexico. Association for Computational Linguistics.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025a. Qwen3 technical report.

Zheyuan Yang, Zexi Kuang, Xue Xia, and Yilun Zhao. 2025b. Can llms generate high-quality test cases for algorithm problems? testcase-eval: A systematic evaluation of fault coverage and exposure.

Appendix

A Chain of Thought Structure

Listing 1: Concept Analysis

```
{
"methods": [
    {
        "method": "Dynamic Programming",
        "instruction": "For each cell,
            track the maximum number of
            pawns the knight can capture
             if it reaches that cell,
            updating only when moving
            rightward due to the knight'
            s move restrictions."
    },
    {
        "method": "Array",
        "instruction": "Use a 2D array
            to maintain state for each
            cell (number of pawns
            captured if the knight
            reaches that cell),
            iterating through allowed
            knight moves efficiently."
    },
    {
        "method": "Depth-first Search",
        "instruction": "Optionally, try
            all possible knight paths
            recursively, updating a
            maximum as paths are
            explored (not ideal for the
            largest cases but useful for
             understanding)."
    },
    {
        "method": "Greedy",
        "instruction": "Although greedy
            is sub-optimal here, you
            could try always picking the
             locally best pawn, but this
             could miss the optimal
            globally. Useful for
            comparison or quick
            heuristics."
    }
]
},
```

Listing 2: Error Analysis

```
{
"wrong_submission_code": "...",
"error_type": "time limit exceeded",
"mistake_analysis": [
    "First, the student defines three
        solution methods: brute-force
        recursion (alg), memoized
        recursion (alg2), and a DP
        approach (algdp).",
    "Then, in main(), only algdp (the DP
         approach) is used to process
        each test case, but this DP
        scans the board for every cell
        from the last column to the
        first, recalculating the maximal
         moves inefficiently for every
        cell, not just the relevant
        knight's path.",
    "However, the DP does not use proper
         boundary checks and can
        potentially revisit many cells
        needlessly due to non-optimal
        transitions and redundant
        recomputation on large boards (N
         up to 1000), which causes
        excessive execution time.",
    "Therefore, for large N, this
        results in a TLE as the per-cell
         computations are not minimized
        or restricted to only reachable
        cells from the knight's actual
        position."
        ],
    "misconception_breakdown": "
        Initially, the student tries to
        directly fill the DP table for
        all board cells, assuming it
        models the optimal paths.
        However, they overlook the
        importance of restricting DP
        computation to only the cells
        reachable from the knight\u2019s
         actual position and do not
        optimize transitions to avoid
        redundant work. Thus, they
        misunderstand how to efficiently
         model and restrict state space
        for DP in large constraint
        problems."
    },
}
```

B Dataset Detail

This is the dataset statistics processed from the CodeChef dataset [1].

CodeChef Dataset Statistics	Count
Total submissions	26,663
Total problems	352
Sampled incorrect submissions	999
– Wrong Answer (WA)	463
– Time Limit Exceeded (TLE)	287
– Runtime Error (NZEC)	249
Unique problems (after sampling cap)	279
Avg. submissions per problem	3.58
Max submissions per problem	5

Table 2: CodeChef dataset statistics used for training and evaluating error-aware Chain-of-Thought (CoT) generation.

C Executability of generated problem

Method	Successfully generated	Code Runs	Pass ≥ 1	Failed only 1	Pass All
GPT4.1 + Direct Prompt	89	95.5%	44.9%	11.2%	9.0%
GPT4.1 + RAG	87	87.4%	54.0%	17.2%	13.8%
GPT4.1 + Error Code	97	96.9%	70.1%	19.6%	16.2%
GPT4.1 + EA	89	96.6%	59.6%	10.1%	15.7%
GPT4.1 + CA	83	96.4%	72.3%	20.5%	21.7%
GPT4.1 + CA + EA	89	95.5%	65.2%	16.9%	19.1%

Table 3: Executability of GPT4.1 generation without refinement iteration. Percentages are calculated over successfully generated samples, thus higher than the main table (Pass All / 100).

Method	Successfully generated	Code Runs	Pass ≥ 1	Failed only 1	Pass All
Qwen3 + Direct Prompt	90	91.1%	67.8%	16.7%	17.2%
Qwen3 + RAG	92	88.1%	45.7%	14.1%	17.4%
Qwen3 + Error Code	95	89.5%	62.1%	28.4%	14.7%
Qwen3 + EA	99	85.8%	57.6%	18.2%	25.3%
Qwen3 + CA	93	93.5%	67.7%	19.4%	22.2%
Qwen3 + CA + EA	97	85.6%	58.8%	15.5%	22.7%

Table 4: Evaluation results of Qwen3-235B generation without refinement iteration. Percentages are calculated over successfully generated samples, thus higher than the main table (Pass All/ 100).

[1] Data source: https://www.kaggle.com/datasets/arjoonn/codechef-competitive-programming

D Case Analysis

Input	Generated Output	Concept Analysis
Input Problem	...on a 2D grid, ...determine the number of integer-coordinate...**Manhattan distance** to all restaurants is **minimized**	**Math**: The sum of Manhattan distances is minimized at the median of coordinates. **Sort**: Sorting is required to identify medians, especially for even N (to determine the valid coordinate range). **Array**: Use arrays to store x and y coordinates independently and operate on them efficiently.
Problem Only	Maintain a stack ... When query, remove the highest beauty score closest to the top	**Stack**: Simulate marble insertion/removal with LIFO order. **Heap**: Use a max-heap to track the highest beauty marble. **Array**: Maintain the order of insertions to determine how many marbles lie above the selected one. **Hash Table**: Use for deletion and verification in the heap.
Concept Analysis	... **minimize the total Manhattan distance**	**Sort**: X and Y coordinates are sorted to find medians. **Math**: Manhattan distance minimization leads to median selection; multiple medians may be valid. **Array**: Coordinate arrays are used to support median detection and counting.

Table 5: Examples of generated answers by the model.

Input	Generated Output	Concept Analysis
Input Problem	find M positive integers A1, A2, A3, ..., AM, so that A1*A2*A3*...*AM would be **equal to N**. You should **maximize the number M.**	**Math**: "Find the prime factorization of N. To maximize M, represent N as a product with as many numbers >1 as possible, i.e., break N down into as many factors >1 as possible. **Greedy**: At each step, select the smallest prime factor repeatedly to split N into the maximum number of factors. **Divide and Conquer**: Recursively split N into smaller divisors, always trying to break it down into more factors, but this overlaps with recursive/greedy splitting by the smallest factor. **Recursion**: Recursively divide N by its smallest divisor >1 as long as possible, outputting the path of division as the sequence of factors.
Error Type: Time Limit Exceeded Mistake Analysis: ... trial division to factor each number N...for very large N (up to 10**1000), trial division becomes extremely slow.	write N as a sum $A1 + A2 + ... + AM (M \geq 1, each A_i \geq 1)$ so that A1*A2*...*AM is as large as possible. You must output both this maximum product and the values of A1, ..., AM that achieve it.	**Greedy**: "Repeatedly subtract 3 from N, using as many 3s as possible, and use 2s only when N mod 3 == 1 or 2; this maximizes the product. **Math**: "Notice that splitting N into as many 3s as possible (and possibly one or two 2s) gives the maximum product, based on properties of numbers and exponentials. **Dynamic Programming**:(Alternative for smaller N) Try all partitions recursively and memorize the maximum product for each sum, but this is less efficient for large N.
Error Type: Wrong Answer Mistake Analysis: ...leaving potentially large prime factors unprocessed if n > 10,000...	Your task is to split the integer N into the sum of M positive integers B1, B2, ..., BM, ... so that B1 + B2 + ... + BM **= N**. In addition, you should **maximize the possible value of M**.	**Greedy**: "Assign as many 2s as possible to the sum, then possibly one larger integer to account for the remainder. This guarantees maximizing M since 2 is the smallest allowed value." **Math**: "Determine how many times 2 fits into N (i.e., floor(N/2)) and adjust the last element if N is odd, ensuring all elements are greater than 1 and the sum matches N." **Array**: "Construct an array of 2's of length M, setting the last value to account for any leftover (e.g., 3 if N is odd), ensuring all values are >1."

Table 6: Examples of generated answers by the model.

E Embedding Model Source

The embedding model used in RAG is: https://huggingface.co/thenlper/gte-large. The embedding model in code similarity is intfloat/e5-mistral-7b-instruct.

F Model Size and Budget

We used the OpenAI API for GPT-4.1 from https://platform.openai.com/, the cost to finish the table result is around $31.44 USD. We used the TogetherAI API for Qwen3 235B from https://api.together.ai/, the cost to finish the table result is around $12.2 USD.

G Prompt

This list is the prompt for our problem generation task. The placeholder "[PLACEHOLDER FOR INPUT DESCRIPTION]" will be replaced by the text to describe the input format in different tasks.

H Ablation Study

To determine whether the concept analysis method is needed in the scenario with error code submission, we conduct an experiment that which the LLM is prompted with both concept analysis and error analysis.

Listing 3: Problem Generation Prompt

```
## YOU MUST FOLLOW THESE RULES:
## ## Notice: There might be an existing
     problem in Leetcode. Never present
    an existing problem; try to generate
     a new one
## Generate one similar CODING EXERCISE
    problem with the following. Notice
    that you have:

[PLACEHOLDER FOR INPUT DESCRIPTION]

## MUST INCLUDE:
    - Problem Statement
    - More than 3 test cases
    ## Testcase in this format:
    **Input:** `1 - 2 - 3`
    **Output:** `-4`
    - Sample Python solution:
    ## Code must pass all Test cases
--------Here is one example --------
{
    "problem_statement": "Chef is
        fascinated by triangular numbers
        , defined by the formula T(n) =
        n*(n+1)/2. He wants to challenge
         you to determine, given a very
        large integer K, if K is a
        triangular number.\n\nInput\n\
        nThe first line of the input
        contains an integer T, the
        number of test cases. Each of
        the next T lines contains a
        single integer K (given as a
        string to allow up to 1000
        digits).",
    "test_cases": [
        {
            "inputs": "3\n6\n7\n28",
            "outputs": "YES\nNO\nYES"
        },
        {
            "inputs": "2\n0\n500500",
            "outputs": "YES\nYES"
        }
    ],
    "code": "import sys\nimport math\n\
        ndef is_perfect_square(n):\n
        if n < 0:\n        return False\
        n    x = int(n ** 0.5)\n
        return x * x == n\n\ndef solve()
        :\n    T = int(sys.stdin.
        readline())\n    for _ in range(
        T):\n        Kstr = sys.stdin.
        readline().strip()\n        K =
        int(Kstr)\n        s = 8*K + 1\n
                sqrt_s = int(s ** 0.5)\n
                if sqrt_s * sqrt_s != s:
        \n            print(\"NO\")\n
                continue\n        if
        (sqrt_s - 1) % 2 == 0:\n
                print(\"YES\")\n
            else:\n            print
        (\"NO\")\n\nif __name__ == \"
        __main__\":\n    solve()\n"
}

--------Here are input --------
```