

Leveraging Lean Feedback for LLM-Based Problem-Solving

Ananya Manas

MSc in Data Science
The University of Bath
2024 - 2025

Leveraging Lean Feedback for LLM-Based Problem-Solving

Submitted by: Ananya Manas

Copyright

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the author unless otherwise specified below, in accordance with the University of Bath's policy on intellectual property (see https://www.bath.ac.uk/publications/university-ordinances/attachments/Ordinances_1_October_2020.pdf).

This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Master of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

The integration of Large Language Models (LLMs) with Interactive Theorem Provers (ITPs) represents a promising direction in mathematical problem-solving. While LLMs can generate plausible reasoning steps, they lack guarantees of correctness, and are prone to hallucinations. In contrast, ITPs such as Lean provide rigorous frameworks for mechanically verifying logical validity. This project investigates the efficacy of combining these systems by leveraging Lean to provide structured feedback on LLM-generated solutions. Specifically, we evaluate how Lean's error messages and verification outcomes can be incorporated into autonomous feedback loops to guide model performance on mathematical tasks. The goal is to assess whether this hybrid approach improves the reliability of LLMs in producing correct, machine-verifiable solutions.

Contents

1	Introduction	1
2	Literature and Technology Survey	4
2.1	Background	4
2.1.1	Large Language Models	4
2.1.2	Training and Fine-tuning LLMs	4
2.1.3	Interactive Theorem Provers	5
2.2	LLMs and Mathematics	5
2.3	Combining LLMs with ITPs	6
2.4	Concluding Remarks	8
3	Design and Methodology	9
3.1	Problem Selection for Testing	9
3.2	The Feedback Loop	10
3.3	Lean Code Generation using LLM	11
3.4	Types of Feedback	12
3.5	Tests and Comparative Analysis	12
3.6	Limitations	14
4	Implementation and Testing	15
4.1	Initialisation	15
4.2	The Problem, the Solution, and the LLM's Output	16
4.2.1	Problem 1	16
4.2.2	Problem 2	16
4.2.3	Problem 3	17
4.3	To the Lean Code and Back	17
4.3.1	Creating the Lean Code Template	17
4.3.2	Interpreting the Lean Output	19
4.3.3	A Note on Prompts	20
4.4	Implementing the Tests	21
4.4.1	Evaluating the Feedback Loop for Problem 1	21
4.4.2	Comparing Feedback Strategies for Problem 2	22
4.4.3	Problem 3: Do Single Iteration Improvements Translate to Feedback Loop	22
4.5	Limitations and Setbacks in Implementation	24
5	Results	26
5.1	Performance on the MATH Dataset	26

5.2	Evaluation of Feedback Strategies	26
5.2.1	Problem 1	26
5.2.2	Problem 2	28
5.2.3	Problem 3	29
5.3	Insights from Results	31
6	Conclusions	33
6.1	Future Work	33
6.2	Closing Remarks	34
	Bibliography	35
A	Solutions Generated By the LLM	39
A.1	Initial Answers for Selected Problems	39
A.2	Answers from Tests on Problem 1	40
A.3	Answers from Tests on Problem 3	43
B	Lean Code Templates	45
B.1	Problem-1	45
B.2	Problem-2	45
B.3	Problem 3	46
C	Prompts Used	47
C.1	Problem 1	47
C.2	Problem 2	52
C.3	Problem 3	56
D	Code	60
D.1	Helper Functions	60
D.2	Asking the LLM to Solve MATH Dataset Problems	61
D.3	Problem 1	62
D.4	Problem 2	70
D.5	Problem 3	75

List of Figures

2.1	Comparison of the performance of different LLMs on MATH and GSM8K datasets (Lai et al., 2024a)	7
3.1	Flowchart Representation of Feedback Loop: a) Initial Feedback Loop; b) Modified Feedback Loop	10
4.1	Feedback Prompts for Problem 2.	23

List of Tables

4.1	Natural Language Error Statements for Problem 2	20
5.1	DeepSeek’s performance on Different Subareas of MATH (Test) Dataset . .	26
5.2	Results of Feedback Loop with Binary Correctness for Problem 1	27
5.3	Results of Different Feedback Strategies for Problem 2	28
5.4	Statistical Comparison of Different Feedback Strategies for Problem 2	29
5.5	Results of Different Feedback Strategies for Problem 3	29
5.6	Statistical Comparison of Different Feedback Strategies for Problem 3	30
5.7	Results of Second Test on Problem-3	30

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Marina De Vos, for her invaluable guidance, constructive feedback, and continuous support throughout the course of this project.

Chapter 1

Introduction

Computers have played a significant role in the advancement of mathematics for more than eighty years. While early machines were primarily used for numerical calculations, advancements in computational power expanded their applications well beyond arithmetic. A major, and at the time controversial, milestone in computer-assisted mathematics came in 1976, when Kenneth Appel and Wolfgang Haken used computers to prove the Four Colour Theorem, a conjecture proposed by Francis Guthrie more than a century ago in 1852. Their approach relied on an exhaustive computer-based check of thousands of reducible configurations, marking one of the first times a mathematical proof depended heavily on computation.

As the field of computational mathematics matured, the focus expanded from numerical verifications to symbolic and algebraic manipulation, giving rise to computer algebra systems (CAS) in the 1980s and 1990s. Software such as MATLAB, Mathematica, and Maple played a pivotal role in computational mathematics, enabling users to perform symbolic algebra, numerical analysis, and data visualisation with remarkable ease. Consequently, CAS became indispensable tools in both academic research and industrial applications, facilitating testing of conjectures, discovery of counterexamples, and exploration of previously intractable problems. Thus, CAS bridged the gap between manual calculation and computer-aided mathematics. However, despite their utility, CAS lacked the capacity to tackle mathematical problems with complete certainty.

This limitation was addressed with the emergence of Interactive Theorem Provers (ITPs), which enabled the formal verification of mathematical steps. Unlike CAS, which focus on symbolic computation, ITPs such as Lean, Coq, and Isabelle allow for the construction of mathematical statements that are checked line-by-line by a formal logic engine. These systems guarantee complete correctness, as each step must adhere to strict logical rules. ITPs have been used to formalise complex mathematical results, especially in areas such as modern algebra, number theory, and analysis. However, their use requires knowledge of formal languages and proof strategies, severely limiting their accessibility. Nonetheless, ITPs marked a major leap toward formalised, machine-verifiable mathematics, providing the foundations for the next generation of formal verification tools.

Recently, the emergence and growth of Large Language Models (LLMs) has initiated a new paradigm for computer-assisted mathematics. Unlike previous tools that primarily focus on computation or symbolic manipulation, LLMs offer the ability to interpret and generate natural language, allowing them to suggest mathematical strategies, execute routine steps, and even

translate between informal and formal mathematical language. A major limitation of LLMs is their unreliability in producing correct mathematics, as they are often prone to hallucinations. However, when paired with ITPs, LLMs can receive structured, formal feedback on each generated step, enabling them to iteratively refine their output, and converge towards fully verified and correct solutions.

This project explores the efficacy of this integration, and evaluates how effectively LLMs can solve higher-level mathematical problems when guided by feedback received from ITPs. By analysing performance on a variety of problems, ranging from straightforward exercises to more abstract problems, the project evaluates their potential in guided mathematical problem-solving. The aim is to develop systems that can iteratively refine their outputs, thus improving correctness and reliability in machine-assisted mathematical reasoning.

To this end, a methodology is developed in which Lean is employed as a feedback mechanism to guide LLMs through verification outcomes and structured error messages. The investigation focuses on the extent to which autonomous feedback loops enable LLMs to iteratively refine their solutions and produce machine-verifiable results. Performance is evaluated across a range of representative mathematical problems, allowing both the potential and limitations of this hybrid approach to be assessed.

The main objectives of the project are as follows:

1. To design autonomous feedback loops that involve the LLM generating a solution and corresponding Lean code, and utilising error feedback to refine the solution until correctness is achieved.
2. To evaluate different feedback modalities, including binary correctness indicators and detailed error messages.
3. To measure the effectiveness of Lean-guided feedback in improving the logical validity of LLM-generated solutions.
4. To analyse limitations and challenges of integrating LLMs with theorem provers for mathematical problem-solving.

The remainder of the dissertation is organised as follows:

Chapter 2: Literature and Technology Survey. This chapter provides the background on LLMs and Lean, outlining their evolution and relevance to the project. It highlights limitations of LLMs in mathematical reasoning, and surveys recent work on integrating them with ITPs. Insights from this chapter inform the experimental methodology.

Chapter 3: Design and Methodology. This chapter outlines experimental design for evaluating the feedback loops, covering problem selection, loop structure, feedback strategies, and statistical framework for comparison. Key modifications and refinements introduced during the implementation stage are highlighted, emphasising the experimental and iterative nature of the project.

Chapter 4: Implementation and Testing. This chapter presents the implementation of the experimental framework for assessing the the feedback loop across different strategies. For each problem, design of the Lean code template, interpretation of Lean output, and execution of the feedback loop are discussed. The chapter concludes with a discussion of the limitations encountered during implementation.

Chapter 5: Results. This chapter presents the experimental findings, starting with the performance of the LLM on the MATH dataset. Thereafter, the results of the autonomous feedback loop are analysed by comparing different feedback strategies. Statistical metrics are employed to provide objective insight into the effectiveness of each strategy. Key observations, trends, and limitations identified from the experiments are discussed.

Chapter 6: Conclusion. This chapter summarises the findings of the project and reflects on the integration of LLMs with Lean for autonomous mathematical problem-solving. Potential areas for future investigation are also discussed.

Chapter 2

Literature and Technology Survey

Recent work has explored the use of Large Language Models in mathematical reasoning, but their lack of formal guarantees limits their reliability. While they can produce plausible solution steps, these are often flawed or incomplete. To address this, one of the approaches is to integrate LLMs with Interactive Theorem Provers such as Lean, which provide step-by-step feedback on correctness. This combination leverages the generative strengths of LLMs while enforcing the strict logical standards required for formal proofs. In this chapter, we survey key developments in the field, highlighting both foundational tools and recent advances.

2.1 Background

2.1.1 Large Language Models

A Large Language Model (LLM) is a natural language processing model designed to process and generate human language. LLMs are trained on a vast amount of text datasets, and use deep neural networks to learn the statistical patterns between words. The underlying structure is the transformer, introduced by Vaswani et al. (2017). The transformer architecture replaced recurrent neural networks with self-attention mechanisms, allowing the efficient capturing of long-range dependencies, helping the models predict the next word in the sequence. By stacking multiple layers of self-attention and feed-forward networks, transformers enabled models to scale effectively with more data and larger parameter counts, leading to the dramatic advances seen in modern LLMs.

2.1.2 Training and Fine-tuning LLMs

Fine-tuning LLMs is a common approach for adapting pre-trained models to specific tasks by further training them on task-specific datasets. Full fine-tuning requires updating all model weights, making it computationally and memory intensive, particularly for very large models. On the other hand, parameter-efficient fine-tuning techniques (PEFTs), such as LoRA (Hu et al., 2021), adapters (Houlsby et al., 2019), prefix-tuning (Li and Liang, 2021), and prompt-tuning (Lester, Al-Rfou and Constant, 2021), update only a small subset of parameters, providing an efficient alternative to full fine-tuning.

Few-shot prompting (Brown et al., 2020) offers another approach, leveraging the pre-trained model directly without modifying its parameters. In few-shot prompting, the model is provided

with a small number of examples at inference time to illustrate the task, enabling it to generalise to new instances based on observed patterns. This approach avoids the computational cost of fine-tuning, and is particularly effective when task-specific datasets are limited. A major limitation of few-shot prompting is that performance can be highly sensitive to the choice, number, and quality of examples, and the model may struggle to generalise to tasks that differ significantly from the prompt demonstrations.

2.1.3 Interactive Theorem Provers

Interactive Theorem Provers (ITPs) are software tools that assist users in developing formal mathematical proofs by providing both guidance and rigorous verification. Their core functionality lies in enabling users to construct proofs incrementally, ensuring that each step adheres strictly to a formal logical system. Many ITPs also support proof tactics, which are higher-level strategies that automate common proof steps. Notable examples of ITPs include Coq (Barras et al., 1997), Isabelle/HOL (Nipkow, Wenzel and Paulson, 2002), and Lean (de Moura et al., 2018). ITPs have become increasingly prominent in both academic and industrial settings due to their ability to ensure correctness in software verification, protocol design, and pure mathematics. A landmark achievement highlighting the capabilities of ITPs is the complete formalisation of the Feit–Thompson theorem in group theory using Coq (Gonthier et al., 2013), illustrating the feasibility of formalising complex mathematical arguments.

Among modern ITPs, Lean has gained substantial popularity for its powerful tactic framework, clean syntax, and strong community-driven development. One of Lean’s key strengths lies in its integration with Mathlib (the Lean mathematical library, 2019), a rapidly growing library of formalised mathematics. Mathlib encompasses a broad range of topics including algebra, real analysis, topology, and number theory. As of early 2025, the library contains over 100,000 definitions and more than twice as many theorems, all verified within Lean’s logical framework. Lean also offers excellent tooling through its Visual Studio Code extension and Lean Copilot (Song, Yang and Anandkumar, 2025), which provide an interactive and incremental workflow for proof development.

These features make Lean particularly appealing for projects that combine formal verification with AI systems such as Large Language Models (LLMs), which can assist in suggesting or refining proof strategies while relying on Lean for correctness validation. Its combination of a robust logical framework, extensive libraries, and advanced tooling positions Lean as the preferred ITP for projects that aim to leverage machine learning for formal mathematical reasoning.

2.2 LLMs and Mathematics

Despite their ability to generate human-like text and solve certain computational problems, LLMs are prone to “hallucinations” (Huang et al., 2025), especially in domains requiring precise reasoning. Several studies, such as Boye and Moell (2025) and Williams and Huckle (2024), have illustrated the models’ struggles with even simple logical and mathematical word problems. These errors arise from their reliance on statistical patterns in language rather than true logical deduction or symbolic manipulation. This limitation is particularly pronounced in tasks requiring formal reasoning, such as mathematical problem-solving and theorem-proving. Recent benchmarks, such as MATH (Hendrycks et al., 2021) and GSM8K (Cobbe et al.,

2021), have been developed to rigorously evaluate the mathematical reasoning abilities of LLMs. Results consistently show that even state-of-the-art models demonstrate significant performance gaps when confronted with problems that require multi-step logical inference or formal proof construction (Figure 2.1).

A major challenge in applying LLMs to mathematical reasoning is their limited understanding of the formal structures that underpin mathematics. Unlike mathematicians, who reason within rigorous logical frameworks, LLMs typically generate sequences of tokens that imitate the appearance of logical reasoning without ensuring formal validity. This shortcoming is particularly evident in theorem-proving, where verifying mathematical statements demands structured, step-by-step reasoning. Compared to conventional NLP tasks, theorem-proving requires precise symbolic manipulation and long-range logical consistency. To this end, recent research has explored different methods to increase the reasoning abilities and theorem-proving capabilities of LLMs. These include chain-of-thought prompting (Wei et al., 2022a), reinforcement learning (Dong, Mahankali and Ma, 2024), supervised fine tuning (Dong et al., 2024), Step-DPO (Lai et al., 2024b), and neural theorem-proving (Wang et al., 2023). However, there still remains a large gap between achieving automated theorem-proving through LLMs.

Efforts to bridge this gap forms an active area of research in deep learning. Another method to overcome this limitation includes integrating LLMs with Interactive Theorem Provers, which will be discussed in the next section. Nevertheless, truly mastering mathematical proof generation remains an open challenge for current models.

2.3 Combining LLMs with ITPs

The integration of LLMs with ITPs has been explored in two primary directions: first, incorporating LLMs into proof assistants to enhance proof automation, and second, leveraging feedback from ITPs to guide LLM-generated proofs toward valid formal conclusions.

In the first direction, LLMs are used within proof assistants to support or automate proof construction. Notable examples include CoqPilot (Kozyrev et al., 2024), a Visual Studio Code extension that assists users by filling in proof holes using LLM-generated suggestions, and Lean Copilot (Song, Yang and Anandkumar, 2025), which integrates LLMs with the Lean proof assistant to provide proof-generation support and feedback.

In the second direction, research has focused on leveraging feedback from interactive theorem provers (ITPs) to improve the reliability of LLM-generated proofs. A crucial first step is translating natural language mathematics into formal syntax. Wu et al. (2022) demonstrated that LLMs such as Codex (Chen et al., 2021) can successfully formalise 25.3% of competition-level math problems into Isabelle/HOL using only few-shot prompting (Brown et al., 2020). For the Lean proof assistant, Azerbayev et al. (2023) introduced ProofNet, a dataset containing 371 undergraduate-level problems wherein each entry includes a natural language theorem statement and proof, along with a corresponding formal statement in Lean syntax. They further evaluated autoformalisation performance using few-shot learning, and proposed two enhancement techniques: prompt retrieval and distilled backtranslation.

Expanding on the challenges of LLM-based proof generation, Lu, Delaware and Zhang (2024) conducted a study analysing the types of errors LLMs (specifically GPT-3.5) make during theorem-proving. They introduced PALM (Proof Automation with Language Models), a system that generates initial proofs using LLMs and refines them through a combination of Coq tactics

Model	size	general	open	MATH (%)	GSM8K (%)
GPT-3.5-Turbo	-	✓	✗	42.5	92.0
Gemini-1.5-Pro (Feb) (Reid et al., 2024)	-	✓	✗	58.5	91.7
Gemini-1.5-Pro (May) (Reid et al., 2024)	-	✓	✗	67.7	90.8
Claude-3-Opus	-	✓	✗	60.1	95.0
GPT-4-1106 (Achiam et al., 2023)	-	✓	✗	64.3	91.4
GPT-4-Turbo-0409 (Achiam et al., 2023)	-	✓	✗	73.4	93.7
GPT-4o-0513	-	✓	✗	76.6	95.8
Llama-3-8B-Instruct (Touvron et al., 2023)	8B	✓	✓	30.0	79.6
Qwen2-7B-Instruct (Bai et al., 2023)	7B	✓	✓	49.6	82.3
Llama-3-70B-Instruct (Touvron et al., 2023)	70B	✓	✓	50.4	93.0
DeepSeek-Coder-V2-Instruct (Zhu et al., 2024)	236B	✗	✓	75.7	94.9
Code-Llama-7B (Roziere et al., 2023)	7B	✗	✓	13.0	25.2
MAMooTH-CoT (Yue et al., 2023)	7B	✗	✓	10.4	50.5
WizardMath (Luo et al., 2023)	7B	✗	✓	10.7	54.9
MetaMath (Yu et al., 2023)	7B	✗	✓	19.8	66.5
MetaMath-Mistral-7B (Yu et al., 2023)	7B	✗	✓	28.2	77.7
MathScale-Mistral Tang et al. (2024)	7B	✗	✓	35.2	74.8
InternLM-Math-7B (Ying et al., 2024)	7B	✗	✓	34.6	78.1
Xwin-Math-Mistral-7B (Li et al., 2024)	7B	✗	✓	43.7	89.2
MAMmoTH2-7B-Plus (Yue et al., 2024)	7B	✗	✓	45.0	84.7
MathGenieLM-Mistral (Lu et al., 2024)	7B	✗	✓	45.1	80.5
InternLM-Math-20B (Ying et al., 2024)	20B	✗	✓	37.7	82.6
MathGenieLM-InternLM2 (Lu et al., 2024)	20B	✗	✓	55.7	87.7
DeepSeekMath-Instruct (Shao et al., 2024)	7B	✗	✓	46.8	82.9
DeepSeekMath-RL (Shao et al., 2024)	7B	✗	✓	51.7	88.2
DeepSeekMath-RL + Step-DPO	7B	✗	✓	53.2 (+1.5)	88.7 (+0.5)
DeepSeekMath-Base-SFT [†]	7B	✗	✓	52.9	86.7
DeepSeekMath-Base-SFT + Step-DPO	7B	✗	✓	54.1 (+1.2)	86.7 (+0.0)
Qwen2-7B-SFT [†]	7B	✗	✓	54.8	88.2
Qwen2-7B-SFT + Step-DPO	7B	✗	✓	55.8 (+1.0)	88.5 (+0.3)

Figure 2.1: Comparison of the performance of different LLMs on MATH and GSM8K datasets (Lai et al., 2024a)

and a backtracking procedure to correct errors. Similarly, First et al. (2023) proposed Baldur, a proof automation method for Isabelle/HOL that uses LLMs to generate entire proofs at once, followed by a dedicated repair model to fix inaccuracies. For Lean, Yang et al. (2023) introduced LeanDojo, which "turns Lean into a Gym-like interactive environment," through which models can observe proof states, propose tactics, and receive structured feedback from Lean. These studies demonstrate notable improvements, with both PALM and Baldur achieving strong performance. Additional studies such as Karatarakis (2024) and Jiang et al. (2023) have also explored the effectiveness of LLMs in autoformalising and theorem-proving using Lean. Collectively, they highlight the growing effectiveness of combining LLMs with ITPs, emphasising that structured feedback and repair mechanisms are key to improving proof accuracy and automation.

2.4 Concluding Remarks

Recent literature suggests that combining Large Language Models (LLMs) with feedback from Interactive Theorem Provers (ITPs) offers a promising pathway toward the automation of mathematical problem-solving. This integration leverages the generative capabilities of LLMs alongside the logical rigor enforced by ITPs, allowing models to refine solution attempts based on structured feedback iteratively. From the survey of related work and tools, we conclude that Lean should be the choice of ITP because of its active development, comprehensive Mathlib library, and well-documented formal language. Additionally, few-shot prompting can be applied for the generation of Lean code; if this proves insufficient, fine-tuning on Lean syntax using datasets such as ProofNet and MiniF2F (Zheng, Han and Polu, 2021) may be considered. In such cases, parameter-efficient fine-tuning techniques (PEFTs) like LoRA are advantageous because of their computational efficiency and ability to adapt large models with limited resources.

A key distinction between the reviewed approaches and our project lies in the granularity of model interaction with the ITP. While frameworks such as PALM and Baldur focus on whole-proof generation followed by post-hoc repair mechanisms, our project aims at step-by-step interaction with Lean. This fine-grained feedback facilitates iterative improvement and aligns more closely with the formal nature of mathematical reasoning.

Chapter 3

Design and Methodology

This chapter presents the methodology for designing and conducting experiments to evaluate the effectiveness of feedback loops in mathematical problem-solving. Being an experimental project, the original methodology required iterative refinement as limitations became apparent; these are also discussed, along with the modifications made to address them.

3.1 Problem Selection for Testing

To evaluate the effectiveness of the feedback loop, problems were sampled from two benchmark datasets as sources of mathematical problems:

1. **GSM8K dataset (Cobbe et al., 2021)**: This dataset contains 8,500 grade-school level mathematical word problems. Solving these problems requires translating the natural language description into equations, and solving them. In this context, Lean verifier can be employed to check whether the LLM correctly solves the equation. This dataset serves as a foundation for testing simpler problems.
2. **MATH dataset (Hendrycks et al., 2021)**: This dataset comprises 12,500 competition-style problems spanning seven mathematical domains. It is widely used to evaluate the mathematical capabilities of LLMs, where performance has remained consistently low. This makes the dataset particularly well-suited for identifying problems for application of the feedback loop. Furthermore, the dataset’s theoretical orientation makes it appropriate for the development of a formal verification mechanism.

For the feedback experiments, problems from the *test* split of each dataset were selected where the LLM failed to produce a correct solution when prompted with Chain-of-Thought (CoT) reasoning (Wei et al. (2022b)). CoT prompting guides the model to solve problems step-by-step, articulating its reasoning at each step. Prior studies have shown that this approach substantially improves the LLM’s performance on mathematical tasks, including the GSM8K dataset. Moreover, the intermediate steps were crucial, as they revealed logical and mathematical errors, and provided a basis for designing Lean code to verify the solution.

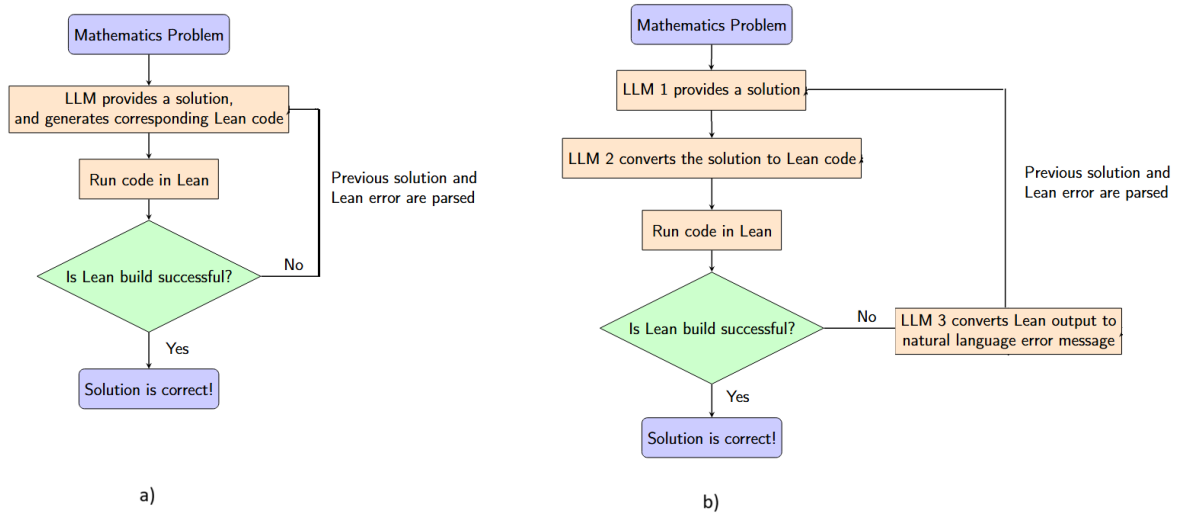


Figure 3.1: Flowchart Representation of Feedback Loop: a) Initial Feedback Loop; b) Modified Feedback Loop

3.2 The Feedback Loop

The initially proposed feedback loop involved prompting a single LLM to both solve the problem and generate corresponding Lean code for verification (Figure 3.1 a). However, this design had a significant limitation: producing Lean code requires familiarity with Lean syntax, which a generic LLM does not possess. If the LLM were exposed to Lean code, it would no longer represent a generic model, defeating the purpose of testing whether feedback alone improves performance. Additionally, training methods such as few-shot prompting could also influence the solution generation, making it difficult to isolate the effect of feedback.

To address this limitation, a crucial modification was introduced: the feedback loop now separates the tasks of problem-solving and Lean code generation. By isolating these steps, the feedback loop retains its independence, with Lean execution continuing to function as an objective, model-agnostic verifier. Furthermore, to provide meaningful guidance to the solver, the output of Lean is translated into natural language feedback rather than raw error messages. This is achieved through a dedicated interpreter LLM. The modified feedback loop structure is represented by Figure 3.1 b).

Consequently, the modified feedback loop employs three specialised LLMs:

1. Solver M_s : Generates solutions to the problem. This is the LLM on which various feedback strategies are evaluated.
2. Translator M_t : Converts the natural language solution produced by M_s into Lean code. This LLM is trained on Lean scripts to ensure syntactic correctness and compatibility with Lean’s verification system.
3. Interpreter M_i : Translates Lean error messages into clear natural language feedback, which is then fed back to M_s to guide the generation of revised solutions.

This modular structure also provides flexibility for experimentation, allowing variations in

prompts for each LLM role to be studied independently without impacting the others.

The complete algorithm for the pipeline is given below:

Algorithm 1 Feedback Loop

Require: Problem statement P , solver LLM M_s , translator LLM M_t , interpreter LLM M_i , maximum attempts N

Ensure: Verified solution A or failure after N attempts

```

1: Initial Solution:
2: Query  $M_s$  with  $P$  to generate candidate solution  $A_0$ 
3: Provide  $A_0$  to translator  $M_t$  to generate Lean code  $C_0$ 
4: Compile and execute  $C_0$  in Lean
5: if verification succeeds then
6:   return  $A_0$ 
7: end if
8: Capture Lean error message  $E_0$ 
9: Query interpreter  $M_i$  with  $(C_0, E_0)$  to obtain natural language feedback  $F_0$ 
  Feedback Loop:
10: for  $i \leftarrow 1$  to  $N$  do
11:   Provide  $(P, A_{i-1}, F_{i-1})$  to solver  $M_s$  to generate revised solution  $A_i$ 
12:   Provide  $A_i$  to translator  $M_t$  to generate Lean code  $C_i$ 
13:   Compile and execute  $C_i$  in Lean
14:   if verification succeeds then
15:     return  $A_i$ 
16:   end if
17:   Capture Lean error message  $E_i$ 
18:   Query interpreter  $M_i$  with  $(C_i, E_i)$  to obtain natural language feedback  $F_i$ 
19: end for
20: return "No verified solution found"

```

3.3 Lean Code Generation using LLM

One of the central components of the pipeline is the generation of correct Lean code using an LLM. Ideally, this would be best achieved with a model trained or fine-tuned on a large corpus of Lean scripts. However, Lean is a highly specialised system, and the availability of such large-scale, high-quality corpora is limited. This scarcity constrains the feasibility of extensive fine-tuning. Given these constraints, the project employs few-shot prompting (Brown et al., 2020), where the model is provided with a set of example inputs and outputs at inference time. This approach does not require additional training, helps the LLM adapt to specific templates and output formats, and improves syntactic correctness while reducing compilation errors. Thus, few-shot prompting provides a lightweight, flexible, and practical alternative to fine-tuning.

It is important to note that the few-shot examples for Lean code generation are used exclusively by the translator LLM. The solver is unaffected by these demonstrations, ensuring that the reasoning process remains unbiased. The examples serve as scaffolding, aligning the translator's output with Lean syntax in a reliable and verifiable manner.

A limitation of this approach is that the translation quality depends on carefully selected

few-shot examples that enable the LLM to remain robust to changes in the input structure. Additionally, the choice of LLM is restricted to those that can generate correct Lean code using only a few demonstrations. Thus, few-shot prompting is not a definitive substitute for domain-specific training.

3.4 Types of Feedback

Different types of strategies were designed to test the effect of feedback on the model's ability to improve its solutions. Initially, two feedback strategies were developed: binary correctness, where the LLM receives its previous solution along with a simple indication of its incorrectness; and error feedback, where the LLM is provided with information on errors in its previous attempt. During implementation, it became apparent that the original design created a potential bias: the feedback strategies were allowed multiple iterations, whereas the initial prompt was evaluated on a single attempt. To address this, a "no feedback" baseline was introduced. Whether the absence of feedback counts as feedback is a philosophical question; in this project, it functions as a fair baseline for comparison. Additionally, a fourth strategy was introduced, which provided only the error statement from the initial attempt, without including the full solution. Accordingly, the conducted experiments evaluate four feedback strategies in total:

1. **No Feedback:** The solver was given the same initial prompt, without any information about its prior attempt. This serves as a baseline.
2. **Binary Correctness with Previous Solution:** The LLM was provided with its full prior solution in natural language, along with a message indicating that the solution was incorrect.
3. **Error Feedback without Previous Solution:** The LLM is provided only with the error message from its previous solution, without access to the full prior attempt.
4. **Error Feedback with Previous Solution:** The LLM is provided with both its previous solution and the corresponding error feedback, thus indicating what and where the errors occurred.

3.5 Tests and Comparative Analysis

For mathematical problems that the LLM failed to solve during initial testing, three distinct categories were identified:

1. **Hard problems:** the LLM consistently generates a complete solution, but it is almost always incorrect, with accuracy below 10%.
2. **Easier problems:** the LLM alternates between correct and incorrect solutions, achieving moderate accuracy between 10% and 30%.
3. **No Final Answer:** the LLM is unable to arrive at a final answer, looping indefinitely in its reasoning.

For problems of the first type, the primary test of a feedback loop is whether it can lead the LLM to a correct, Lean-verified solution before the maximum number of attempts is reached. Here, the comparative measure is the number of iterations required.

By contrast, for problems of the second type, given enough iterations, even the no-feedback baseline can eventually produce a correct solution, making a “run-until-correct” evaluation less informative. To assess these cases, the influence of each feedback strategy within a single iteration of the loop was measured, thus focusing on the immediate effect of feedback. Here, it is assumed that improvements observed within a single iteration serve as a proxy for faster convergence in multi-iteration scenarios.

For this second category, a two-proportion z-test was employed to compare success rates across feedback strategies. This test evaluates whether observed differences between strategies are statistically significant. The stochastic nature of the LLM’s solution generation ensures independence across trials, and selecting a sufficiently large sample size supports the robustness of the conclusions. The complete framework for the test is explained below:

1. **Sample Proportions:** Let p_A and p_B be the sample proportions of feedback strategies A and B , respectively, that is, $p_A = \frac{x_A}{n_A}$ and $p_B = \frac{x_B}{n_B}$, where:

- x_A, x_B are the number of successful solutions under strategies A and B ;
- n_A, n_B are the total number of trials for strategies A and B .

2. **Hypotheses** The test evaluates whether the difference in success rates is statistically significant:

Null Hypothesis (H_0): The probability of success is the same for both strategies, that is,

$$p_A = p_B.$$

Alternative Hypothesis (H_1): The probability of success differs between the strategies, that is,

$$p_A \neq p_B.$$

3. **Pooled proportion:** The pooled proportion \hat{p} represents the combined probability of success across both strategies A and B .

$$\hat{p} = \frac{x_A + x_B}{n_A + n_B}.$$

4. **Standard error:** The standard error quantifies the expected variability of the difference between sample proportions.

$$SE = \sqrt{\hat{p}(1 - \hat{p}) \left(\frac{1}{n_A} + \frac{1}{n_B} \right)},$$

5. **z-statistic:** The z-statistic measures how far apart the two sample proportions are in units of the standard error.

$$z = \frac{p_A - p_B}{SE}.$$

For a two-tailed test at a significance level of $\alpha = 0.05$, the difference is considered statistically significant if $|z| \geq 1.96$.

Concerning the third type of problem, where the LLM fails to produce any valid final solution, no actionable feedback can currently be generated by the proposed loop. Few-shot prompting is also unlikely to resolve these cases. Nevertheless, studying such problems represents an interesting avenue for future research, as identifying logical inconsistencies could potentially break the loop and enable meaningful feedback.

3.6 Limitations

In this project, feedback was restricted to the immediately preceding problem attempt. While including multiple previous solutions could potentially provide more context for the LLM, doing so risks overloading the model with input, which may reduce its performance. Nevertheless, determining the optimal scope and selection of feedback must be investigated.

Chapter 4

Implementation and Testing

4.1 Initialisation

For implementing the feedback loop, three LLMs were considered: LLaMa 7B (via Ollama), GPT4o (via API call), and DeepSeek-chat (via API call). Initial experiments with LLaMa indicated that it was unable to generate correct Lean code through few-shot prompts. Furthermore, the GSM8K problems involved basic arithmetic, where correctness depends on the model's language comprehension than on mathematical verification. In these cases, Lean effectively functions as a calculator. Consequently, all experimentation was conducted on problems from the MATH dataset using DeepSeek-chat, selected due to its lower API cost and reasonable performance on mathematical and code-generation tasks. Henceforth, all references to "LLM" refer to DeepSeek-chat.

The first step involved evaluating the performance of the LLM on the test splits of the MATH dataset. To guide the model's output, the following prompt was used:

Given a math word problem, return only the required steps/formula, only required explanation, and the answer. Only main explanation, reduce the number steps. No extra text.

This prompt was designed to minimise irrelevant content, thereby reducing token usage and focusing the output on essential reasoning steps. Appendix D.2 contains the code used to query the LLM for solutions to the the MATH test dataset.

After obtaining the results, problems for which the LLM produced incorrect answers were identified. Although an automated answer comparison mechanism was implemented to flag discrepancies, several responses were erroneously marked as incorrect due to representational differences with the expected answers in the dataset. Common examples of such discrepancies include:

1. Alternative numeric representations (53332.8 vs. $\frac{266664}{5}$).
2. Variation in the ordering of terms $((2x+3)(2x+1)(2x-1)$ vs. $(2x+3)(2x-1)(2x+1)$).
3. Presence or absence of units (0.5 vs $\$0.50$).
4. Formatting differences in LaTeX syntax ($\dfrac{\sqrt{3}}{2}$ vs. $\frac{\sqrt{3}}{2}$).

As a result, each flagged answer required manual inspection to determine whether the discrepancy was substantive or merely a notational variation. It is important to note that the evaluation was based solely on final answers. Consequently, there may exist cases where the LLM's reasoning was flawed, yet the final answer coincidentally matched the correct value. In such instances, the solution was marked as correct despite the logic being invalid. This limitation reinforces the importance of using Lean-based verification in the project. The final evaluation metrics, presented in Table 5.1, reflect the LLM's performance on the MATH dataset after accounting for such equivalences. It should be noted that these tests were performed using DeepSeek-V3, prior to the release of DeepSeek-V3.1.

4.2 The Problem, the Solution, and the LLM's Output

To evaluate the effectiveness of the feedback loop, a set of MATH dataset problems that could be fully verified in Lean was selected. These selected problems are provided below. For each problem, both the correct solution and the LLM's attempt are presented. The verbatim LLM response for each problem is provided in Appendix A.1.

4.2.1 Problem 1

The first problem involves the coefficients of polynomials, and prime numbers:

Suppose the roots of the polynomial $x^2 - mx + n$ are positive prime integers (not necessarily distinct). Given that $m < 20$, how many possible values of n are there?

For a quadratic polynomial, the sum of the roots equals the coefficient of x , and the product equals the constant term. Hence, the problem reduces to enumerating all prime pairs with sum less than 20, and computing their products. This yields 18 valid prime pairs and, correspondingly, 18 distinct values of n .

The LLM's solution followed the correct approach; however, it omitted the pair (5, 13), resulting in the incorrect answer of 17.

4.2.2 Problem 2

Problem 2 is a combinatorial sum problem:

The digits 1, 2, 3, 4, 5, 6, 7, and 9 are used to form four two-digit prime numbers, with each digit used exactly once. What is the sum of these four primes?

Mathematically, every two-digit prime must end with 1, 3, 7 or 9, restricting the tens digit to 2, 4, 5, and 6. The sum of the four primes is, therefore,

$$10(2 + 4 + 5 + 6) + (1 + 3 + 7 + 9) = 190.$$

Alternatively, the problem can be brute-forced by listing all admissible sets of four primes numbers, and computing their sums. All such valid sets sum to 190.

The LLM attempted the brute-force approach. However, its final list of primes was incorrect, repeating the digit 1, and omitting 5. Repeated runs demonstrated that the LLM consistently produced solutions by listing the primes.

4.2.3 Problem 3

Problem 3 concerns with the root of a polynomial:

If $3 + \sqrt{5}$ is a root of the equation $[x^2 - 3x + b = 0, \backslash]$
compute b .

The problem can be solved directly by substituting $x = 3 + \sqrt{5}$ in the equation, resulting in $b = -5 - 3\sqrt{5}$. However, the LLM incorrectly applied the conjugate root theorem, assuming that assuming that $3 - \sqrt{5}$ is also a root. This led it to produce the wrong solution $b = 4$.

The selected problems highlight three distinct types of errors in the LLM's mathematical reasoning. In Problem 1, the model omitted a valid prime pair, undercounting possible values. Problem 2 revealed difficulties with combinatorial reasoning, as the LLM repeatedly generated invalid prime sets with repeated or missing digits. Problem 3 demonstrates the misapplication of mathematical theorems, leading to an incorrect solution. Together, these problems were chosen to evaluate whether the feedback framework can detect and correct errors across different levels of mathematical reasoning.

4.3 To the Lean Code and Back

Once the LLM produces a solution, it must be translated into Lean code for formal verification. Importantly, the generated Lean code functions purely as a verifier, evaluating only the reasoning steps proposed by the LLM. This keeps the verification process independent and objective.

Equally crucial is the reverse step: converting Lean's output back into a natural-language error statement. This interpretation makes the verification results accessible to the solver LLM.

4.3.1 Creating the Lean Code Template

For each problem, a dedicated template defines the expected structure of the Lean script, guiding the LLM to formalise each reasoning step. These templates ensure consistency across problems, and provide a framework for translating intermediate reasoning into mechanised statements. This section describes the creation of these templates, with the complete Lean code templates provided in Appendix B.

Problem 1

The generated solution for Problem 1 involves three components for verification:

1. Completeness and correctness of the list of prime pairs (p, q) with $p \leq q$ and $p + q < 20$.
2. Accuracy of the computed products $n = p \cdot q$.
3. Correctness of the final count of distinct n -values.

For the first step, the list of valid prime pairs is defined in Lean as follows:

```
def validPrimePairs : List (N × N) :=
  let primes := (List.range 20).filter Nat.Prime
  List.foldr (· ++ ·) [] (
    primes.map (fun p =>
```

```

      (List.range 20).filter Nat.Prime |>.filterMap (fun q =>
        if p + q < 20 then some (p, q) else none))
    )

```

The `validPrimePairs` definition computes all valid prime pairs (p, q) . It begins by generating all prime numbers less than 20. `filterMap` ensures that only qualifying pairs are retained, while `foldr` flattens the nested list structure into a single list. The list produced by the LLM is provided as `inputPairsList`, and `verify_eqv` checks the equivalence between the two lists. If the lists differ, the build fails, indicating a discrepancy.

The next step verifies the accuracy of the distinct products $n = p \cdot q$:

```

def nValues : List ℕ := validPrimePairs.map (fun (p, q) => p * q)
def distinctNValues : Finset ℕ := nValues.toFinset
def inputNValues : Finset ℕ := {inputNValues}
theorem verify_nValues_correct : distinctNValues = inputNValues :=
  by decide

```

The `nValues` computes all products $p \cdot q$ for `validPrimePairs`. To retain only the distinct values, the list is converted into a finite set using `toFinset`. The set of distinct n -values generated by the LLM is provided through `inputNValues`, and the `verify_nValues_correct` formally checks the equivalence of the sets.

The final step verifies the total number of distinct n -values by computing the cardinality of `distinctNValues` and matching it with `inputCount` using `verify_cardinality_correct`:

```

theorem verify_cardinality_correct : distinctNValues.card = inputCount
  := by rfl

```

Problem 2

For Problem 2, the solution produced by the LLM must satisfy four conditions:

1. The four two-digit numbers are all prime.
2. Only the digits 1, 2, 3, 4, 5, 6, 7, and 9 are used across these numbers.
3. Each allowed digit is used exactly once.
4. The sum of the four numbers is correct.

To formally verify these conditions in Lean, each condition is encoded as a Boolean variable that evaluates to true if satisfied, or false otherwise.

```

def digitsDistinct (n : Nat) : Bool :=
  let ds := digits n
  ds.length = ds.eraseDup.length
def digitsInSet (allowed : List Nat) (n : Nat) : Bool :=
  let ds := digits n
  ds.all (fun d => listElemBool d allowed)
def allowedDigits : List Nat := [1, 2, 3, 4, 5, 6, 7, 9]

```

In the above code, the definition `digitsDistinct` ensures no digit is repeated within a number, while `digitsInSet` confirms that all digits are from `allowedDigits` list. The next step verifies if a given number n is prime:

```
def isPrimeBool (n : Nat) : Bool :=
  if n < 2 then false else
    !((List.range (n - 2)).any (fun d => n % (d + 2) = 0))
```

Next, the function `allDigitsDistinct` guarantees that each digit appears exactly once.

```
def allDigitsDistinct (nums : List Nat) : Bool :=
  let ds := allDigits nums
  ds.length = ds.eraseDup.length
```

Finally, the input provided by the LLM is encoded as `inputList`, and its computed sum as `inputSum`. The boolean flags `allPrime`, `noDigit8`, `allDistinctDigits`, and `sumCorrect` check each condition independently, allowing precise identification of which aspect of the solution fails.

```
def allPrime : Bool := inputList.all isPrimeBool
def noDigit8 : Bool := inputList.all (digitsInSet allowedDigits)
def allDistinctDigits : Bool := allDigitsDistinct inputList
def sumCorrect : Bool := inputList.foldl (· + ·) 0 = inputSum
```

Problem 3

In Problem 3, the LLM makes an error by misapplying the Conjugate Root Theorem. Specifically, it assumes that if $3 + \sqrt{5}$ is a root of the polynomial $x^2 - 3x + b$, then $3 - \sqrt{5}$ must also be a root.

To verify the correctness of the LLM's solution, the root $x = 3 + \sqrt{5}$ is substituted into the polynomial with the computed b . The solution is deemed correct only if this yields zero. Since Lean's built-in real number type \mathbb{R} is not practical for numerical evaluation, real numbers are represented as floating-point values, and a small tolerance ($\epsilon = 0.01$) is allowed when comparing to zero:

```
def evalPoly (x b : Float) : Float := xx - 3.0x + b
def approxZero (x : Float) (eps : Float := 0.01) :
  Bool := Float.abs x < eps
def isRoot : Bool := approxZero (evalPoly givenRoot input_b)
```

4.3.2 Interpreting the Lean Output

For each of the three problems, few-shot prompts were used to convert the Lean output to natural language error statements, which could then be provided to the solver as feedback.

For Problem 1, the following error statements were employed:

1. If the list of prime pairs generated was incorrect:

The list of prime pairs generated in your answer were incorrect. Retry the problem again, this time going over each pair carefully, and making sure all valid prime pairs are included."

2. If the computation of $n = p \cdot q$ for some pair was incorrect:

The list of prime pairs are correct. However, the list distinct n values are incorrect. Retry the problem again, this time going over

each `n` value carefully, and making sure all valid `n` values are included.

These error statements may be regarded as overly helpful, since they go beyond merely nudging the solver, instead providing explicit instructions. Unfortunately, DeepSeek-V3.1 was released before further experiments could be conducted.

For Problem 2, the error statements were designed to be more concise and direct, each pointing to a specific type of mistake (Table 4.3.2).

Table 4.1: Natural Language Error Statements for Problem 2

Mistake in LLM's Reasoning	Error Message
The four numbers were not all prime	Not all numbers in the list are prime
The digit 8 was present	The list includes the digit 8, which is not allowed
One of the digits is used more than once	The digits in the numbers are not all distinct.
The sum of the four numbers is incorrect	The sum of the numbers is incorrect.

For Problem 3, since the Lean code performs only one check, the following error was used:

`3+sqrt{5}` is not a root of the polynomial $x^2 - 3x + b = 0$ for the given value of `b`.

This kept the feedback minimal, offering only a direct indication of failure without any further guidance.

4.3.3 A Note on Prompts

In the experiments, the prompt for Lean code generation consisted of the Lean code template along with two demonstration examples, using which the LLM was able to generate over 10,000 correct Lean code across all runs. However, to examine whether such extensive guidance was strictly necessary, a subsequent experiment was conducted where the LLM relied solely on the two demonstration examples. The results showed that the correct Lean code was produced across 100 test runs, indicating that two examples are sufficient.

For feedback error generation, the few-shot prompts directly influenced the LLM's output, resulting in the same or closely related error messages. To explore this further, an additional experiment was conducted on Problem 3, where the LLM was asked to generate the error message itself:

The polynomial evaluation at `'3.0 + 2.2360679775'` with `'b = 4.0'` does not yield a value near zero (within tolerance 0.01), so `'isRoot'` returns `'false'`.

This error message fails to localise the error, resulting in reduced effectiveness of feedback strategies. This highlights the significant role of error message design in influencing feedback effectiveness.

4.4 Implementing the Tests

This section describes the implementation of the feedback strategies using Python.

Firstly, a small discussion on Python functions that were used throughout the implementation is presented. The code for each of these functions is provided in Appendix D.1.

The function `call_deepseek` was used to generate a response from the LLM. For this project, the temperature was set to 0.0 to ensure deterministic behaviour and reproducibility of results across runs. The output tokens were limited using `max_tokens` to prevent overly long or non-terminating outputs, and `stream=false` ensures that the response is returned in full, simplifying parsing. Finally, `content.strip()` removes surrounding whitespace from the generated response.

Next, the functions `write_lean_code` and `run_lean_code` were defined to execute the generated Lean code via `lake build`. `write_lean_code` writes the given Lean code string to the file specified by `LEAN_FILE_PATH`, preparing it for compilation, while `run_lean_code` executes the Lean build process, capturing both the standard output `stdout` and the standard error `stderr`. After execution, the Lean file is cleared to prevent stale code from affecting subsequent runs. The function returns a tuple (`success`, `output`), where `success` is a Boolean indicating build success, and `output` contains the captured Lean output.

4.4.1 Evaluating the Feedback Loop for Problem 1

The feedback loop for Problem 1 was implemented using DeepSeek-V3, prior to the model update. The evaluation focused on “Binary Correctness” and “Error Feedback with Previous Solution.” Unfortunately, a later inspection revealed a coding error; the loop for “Binary Correctness” inadvertently reused the initial prompt at each iteration, reducing it to the baseline (no feedback). Results for Problem 1, therefore, consider only the no-feedback and the error feedback strategies. The prompts are given in Appendix C.1, and the complete code is provided in Appendix D.3.

Implementation began by directing the LLM to solve the problem using the initial prompt. The generated solution was converted into Lean code using the Lean code generation prompt. The code was executed, and if the Lean build succeeded, the code returned the response. Failure in Lean build pointed to an error in the LLM’s reasoning, prompting further iterations.

At this point, the workflow diverged. For “No Feedback,” the LLM was simply re-prompted with the initial prompt, with no information about its previous solution. On the other hand, for “Error Feedback,” a natural language feedback message `error_feedback` was generated using the Lean interpretation prompt. The problem text, previous response, and the `error_feedback` message were then parsed to the LLM via `retry_prompt` to generate a new response. Lean then verified the new solution. This process was iterated until `run_lean_code()` returned success, or until the maximum number of iterations is reached.

In this experiment, the maximum number of iterations was set to 100. The effectiveness of each feedback strategy was evaluated on whether it converged to a correct, Lean-verified solution.

4.4.2 Comparing Feedback Strategies for Problem 2

This experiment was conducted using DeepSeek-V3, prior to the model update. The implementation pipeline mirrors that of Problem 1 (solution generation, translation to Lean code, running the Lean code, and error interpretation), with problem-specific adjustments discussed below. The prompts used are listed in Appendix C.2, and the complete code is given in Appendix D.4.

Initial testing revealed that the previous wording caused the LLM to produce non-terminating outputs on multiple occasions, making them unsuitable for structured feedback. To address this, multiple prompt variations were explored. The final version of the prompt used for the problem is given below:

Given the following math word problem, provide a **concise and brief** solution: include only the minimal necessary explanation and the final answer. Return the answer in the format: `\boxed{answer}`. Do **not** include detailed steps, just enough explanation to justify the answer.

The LLM was instructed to return the final answer in the `\boxed{answer}` format, which provided a clear marker for solution completeness. With this prompt, the LLM produced the correct answer in roughly 30% of the attempts.

The test for Problem 2 examined the effect of each feedback strategy on accuracy within a single iteration. In total, 1000 trials were conducted to provide sufficient statistical power for comparison the two-proportion z-test. Each trial began with the LLM generating an initial solution, and concluded with one of the three outcomes:

1. The initial solution was incomplete. In this case, the trial was rerun without increasing the trial count.
2. The initial solution was successfully verified by Lean. In this case, the trial ended without any feedback implementation. Nevertheless, this trial was still counted to establish a baseline accuracy.
3. The initial solution was incorrect (Boolean flag `false` in Lean output). In this case, each of the feedback strategies were implemented, using the respective feedback from the initial response.

The feedback strategies were implemented using the prompts shown in Figure 4.1. Each strategy was implemented by selecting and applying the corresponding prompt to the LLM. To prevent ordering bias, the sequence of strategies for each retry was randomised.

Evaluation of each feedback strategy was based on whether the Lean code verified successfully, yielding its accuracy proportion. A more effective strategy corresponds to higher accuracy.

4.4.3 Problem 3: Do Single Iteration Improvements Translate to Feedback Loop

The tests for Problem 3 were conducted after the release of DeepSeek-V3.1. Initial testing showed that the LLM produced a correct solution in 9% of attempts. Two tests were carried out on this problem. The prompts used are listed in Appendix C.3, and the code is provided in Appendix D.5.

Initial Prompt / No Feedback

Given the following math word problem, provide a **concise and brief** solution: include only the minimal necessary explanation and the final answer. Return the answer in the format: `\boxed{answer}`. Do **not** include detailed steps, just enough explanation to justify the answer.
Problem: {problem}.

Binary Correctness with Previous Solution

Given the following math problem:
{problem},
You solved it like this:
{previous answer}.
This answer was incorrect. Retry the problem from scratch.
Provide a **concise and brief** solution: include only the minimal necessary explanation and the final answer. Return the answer in the format: `\boxed{{answer}}`. Do **not** include detailed steps, just enough explanation to justify the answer.

Error Feedback without Previous Solution

Given the following math problem:
{problem},
Your previous answer had the following error:
{error}.
Retry the problem from scratch.
Provide a **concise and brief** solution: include only the minimal necessary explanation and the final answer. Return the answer in the format: `\boxed{answer}`. Do **not** include detailed steps, just enough explanation to justify the answer.

Error Feedback with Previous Solution

Given the following math problem:
{problem},
You previously solved it like this:
{previous answer}.
But it had the following error:
{error}.
Retry the problem from scratch.
Provide a **concise and brief** solution: include only the minimal necessary explanation and the final answer. Return the answer in the format: `\boxed{answer}`. Do **not** include detailed steps, just enough explanation to justify the answer.

Figure 4.1: Feedback Prompts for Problem 2.

The first test evaluated the effectiveness of each feedback strategy over a single iteration. The setup and code are nearly identical to those in the Problem 2, with only problem-specific adjustments. The test results are presented in Table 5.5.

The second test for this problem was designed to address a central hypothesis of this dissertation: whether improvements observed in a single feedback iteration translate into better iterative performance within a full feedback loop. This distinction is important, since a feedback strategy may not necessarily sustain performance over multiple rounds of correction.

For this experiment, 100 complete trials were executed. A trial consisted of the following steps:

1. **Initial attempt.** The LLM produced an initial solution using the initial prompt. If no complete answer was detected, or if Lean verified the solution as correct, the trial was discarded and restarted. This ensured that each trial began with a genuinely incorrect, but structurally valid initial attempt.
2. **Error feedback generation.** If Lean returned a failure on the initial solution, the pipeline proceeded to generate feedback. The Lean output was translated into a natural-language description of the error. The initial answer and `error_description` were used for feedback.
3. **Feedback loop execution.** Each strategy was implemented as a feedback loop, which repeatedly prompted the LLM with the problem, its most recent solution, and the relevant feedback message. After each attempt, the new solution was checked for completeness, converted to Lean code, and re-verified. The loop terminated either when a Lean-verified correct solution was obtained, or when a maximum of 20 attempts was reached. Incomplete responses were ignored, and did not count towards the attempt limit.

To operationalise this process, a helper function `run_strategy` was defined. This function implemented the loop logic, maintaining the most recent answer and error message, updating them at each iteration, and enforcing the attempt limit.

In total, 100 feedback loops per strategy were executed, with each loop allowing up to 20 attempts. This produced a robust dataset for comparing the sustained effectiveness of feedback strategies when applied iteratively.

4.5 Limitations and Setbacks in Implementation

The experiments were carried out according to the designed methodology, focusing on evaluating the effect of feedback. However, several limitations should be noted:

1. **Prompt-related Limitations:** Different prompts were used across problems, and the system relied on few-shot examples for generating Lean code. Variations in prompt phrasing, structure, or the examples chosen can influence the LLM's outputs. Thus, the observed results cannot be attributed solely to the feedback strategies.
2. **Model Dependence:** The system's performance is contingent on the specific LLM used. Different models may produce substantially different results, limiting generalisability.
3. **Error Interpretation Constraints:** The feedback loop relies on Lean error messages to guide subsequent attempts. In these experiments, error messages were simplified and

standardised. In a fully automated system, variations in error outputs could unpredictably affect feedback performance.

Two major setbacks arose during implementation, both of which prolonged the process and affected the scope of final testing:

1. **Challenges with Lean Code:** Limited Lean code examples, and the ongoing development of Mathlib 4, made constructing correct Lean code templates difficult. As a result, the Lean code used may not always be optimal.
2. **Model Update Disruption:** A mid-project update from DeepSeek V3 to DeepSeek V3.1 prevented additional planned experiments on Problem 1. The updated model produced different outputs, solving problems it had previously failed, making results incomparable.

Chapter 5

Results

5.1 Performance on the MATH Dataset

Table 5.1 summarises the performance of DeepSeek-chat V3 on the test portion of the MATH dataset, broken down by subarea, highlighting the differences in performance across various topics.

Table 5.1: DeepSeek’s performance on Different Subareas of MATH (Test) Dataset

Area of Mathematics	Total No. of Problems	No. of correct answers	Accuracy
Algebra	1187	1137	95.79
Counting and Probability	474	410	86.50
Geometry	479	399	83.30
Intermediate Algebra	903	693	76.74
Number Theory	540	503	93.15
Prealgebra	871	808	92.77
Precalculus	546	448	82.05
Total	5000	4398	87.96

5.2 Evaluation of Feedback Strategies

5.2.1 Problem 1

The tests for Problem 1 were conducted using DeepSeek-V3, prior to the model update. The evaluation measured how effectively feedback guided the LLM to produce Lean-verified solutions.

No Feedback

The LLM was asked to solve the problem using the initial prompt 100 times. The final answers generated, their frequencies, and the corresponding Lean build results are shown in Table 5.2.

Two main observations emerge from these results. First, the most frequent output was 17, with frequencies decreasing for answers further from this value. Second, the LLM did produce

Table 5.2: Results of Feedback Loop with Binary Correctness for Problem 1

Final Answer	Frequency	Build Result
6	1	Failed
7	1	Failed
8	1	Failed
10	7	Failed
14	5	Failed
15	15	Failed
16	26	Failed
17	36	Failed
18	3	Failed
19	3	Failed
20	1	Failed
25	1	Failed

the “correct” final answer (18) in three cases; however, in each instance, the pair (11, 11) was included and (5, 13) omitted (see Appendix A.2), resulting in failure in Lean build.

Feedback with Error Messages

An important test of our feedback mechanism was conducted using error messages containing detailed diagnostic information. The feedback included the problem statement, the LLM’s previous solution, and an indication of where the reasoning had gone wrong. This was aimed at determining whether targeted corrective prompts could steer the model toward the correct result.

The results from the first run of the feedback loop are detailed below. The complete generated solution for each attempt is provided in Appendix A.2.

Attempt 1: The initial response excluded several valid prime pairs, labeling them as ‘invalid.’ The LLM returned the incorrect final count of 10. Corrective prompt was issued, and the LLM was prompted to generate a new solution based on the prompt.

Attempt 2: With the more detailed error message, the LLM included many previously omitted pairs, showing a clear improvement. However, the pair (5, 13) was still missing, so the solution was still not fully correct.

Attempt 3: Another retry with targeted feedback was initiated. This third attempt not only included the missing pair (5, 13), but also correctly computed the total number of valid n values, and yielding the correct final answer.

This sequence illustrates the effectiveness of detailed error feedback. Each iteration of corrective prompting resulted in tangible improvements, culminating in a fully verifiable solution. To test consistency, the feedback loop was run multiple times; in all cases, the LLM reached the correct answer within three to five iterations. This repeatability strongly suggests that error-message feedback is a robust and reliable strategy.

5.2.2 Problem 2

The tests for Problem 2 were conducted prior to the model update to DeepSeek-V3.1. The evaluation measured the effect of the different feedback strategies after one iteration of feedback application.

Preliminary Observations

For Problem 2, the LLM was prompted to solve the problem over 1,000 independent trials. It produced the correct answer in 342 cases, resulting in an baseline accuracy of 34.2%. In 11 trials, the LLM failed to produce any answer; these were excluded from the analysis. From the 658 incorrect first attempts, the effect of each feedback type strategy is summarised in Table 5.3.

Table 5.3: Results of Different Feedback Strategies for Problem 2

Feedback Strategy	No. of Correct Answers	Accuracy
No Feedback	225	34.19%
Binary Correctness with Previous Solution	348	52.89%
Error Feedback without Previous Solution	138	20.97%
Error Feedback with Previous Solution	469	71.28%

Several observations from the results are noteworthy. First, the “No Feedback” strategy produced the same accuracy as the initial prompt, which was expected. Second, “Error Feedback with Previous Solution” achieved the highest accuracy among all types of feedback. Providing the model with both its earlier reasoning and the identified error may have helped it localise the mistake more effectively, enabling it to make targeted corrections. Third, “Error Feedback Without Previous Solution” performed worst, even below the baseline accuracy. One possible explanation is that simply pointing out mistakes without preserving context can push the model further from the correct path. Together, the results highlight the importance of anchoring feedback in the model’s own prior reasoning.

Statistical Analysis

To assess whether the observed differences are statistically significant, pairwise comparisons were performed using the two-proportion z-test at the 5% significance level. A difference is deemed significant whenever $|z| \geq 1.96$.

As an illustration, consider the comparison between “No Feedback” (A) and “Binary Correctness with Previous Solution” (B). The observed sample proportions are

$$p_A = \frac{225}{658} = 0.3419 \quad \text{and} \quad p_B = \frac{348}{658} = 0.5289,$$

with pooled proportion

$$\hat{p} = \frac{225 + 348}{2 \times 658} = 0.4354,$$

yielding a standard error of

$$SE = \sqrt{0.4354 \times (1 - 0.4354) \left(\frac{1}{658} + \frac{1}{658} \right)} = 0.0273.$$

The resulting z-statistic is

$$z = \frac{0.3419 - 0.5289}{0.0273} = -6.8498$$

Since $|z| = 6.8498 \geq 1.96$, the test shows that the “Binary Correctness” strategy produces significantly higher accuracy than the “No Feedback” strategy.

Applying the same procedure across all pairwise comparisons, where the strategies are labeled as A: No Feedback, B: Binary Correctness, C: Error Feedback without Previous Solution, and D: Error Feedback with Previous Solution, the results are presented in Table 5.4.

Table 5.4: Statistical Comparison of Different Feedback Strategies for Problem 2

Comparison	z-statistic	Conclusion (5% level)
A vs. B	−6.85	Binary Correctness better
A vs. C	+5.91	No Feedback better
A vs. D	−14.14	Error Feedback with Previous Solution better
B vs. C	+12.57	Binary Correctness better
B vs. D	−7.07	Error Feedback with Previous Solution better
C vs. D	−20.62	Error Feedback with Previous Solution better

Overall, the experiments on Problem 2 indicate that feedback improves performance when grounded in the model’s prior reasoning. Without such anchoring, feedback becomes abstract, and can mislead the model, reducing accuracy.

5.2.3 Problem 3

Preliminary Observations from Test 1

The LLM was prompted to solve the problem over 1,000 independent trials. If the model produced the correct answer on the first attempt, no feedback was applied. Otherwise, the first-attempt solution and the corresponding error statement were used to evaluate the accuracy of the different feedback strategies. The LLM answered correctly on the first attempt in 90 trials, yielding an overall first-attempt accuracy of 9.0%. In 35 trials, the model failed to produce any answer. As with Problem 2, trials with incomplete first attempts were excluded from the experiment. From the remaining 910 trials with incorrect first attempts, the accuracy for each feedback strategy is summarised in Table 5.5.

Table 5.5: Results of Different Feedback Strategies for Problem 3

Feedback Strategy	No. of Correct Answers	Accuracy
No Feedback	107	11.76%
Binary Correctness with Previous Solution	551	60.55%
Error Feedback without Previous Solution	182	20%
Error Feedback with Previous Solution	160	17.58%

These results contrast sharply with those observed for Problem 2. Whereas “Error Feedback with Previous Solution” was the most effective strategy in that case, here it performs poorly,

surpassing only the no-feedback condition. The best-performing strategy is “Binary Correctness with Previous Solution,” while “Error Feedback without Previous Solution” slightly outperforms “Error Feedback with Previous Solution.”

Statistical Analysis

Similar to Problem 2, pairwise two-proportion z -tests were performed to compare the four feedback strategies over the 910 trials with incorrect first attempts. A difference is considered significant at the 5% level, that is, $|z| \geq 1.96$. The results are presented in Table 5.6.

Table 5.6: Statistical Comparison of Different Feedback Strategies for Problem 3

Comparison	z -statistic	Conclusion (5% level)
A vs. B	−21.68	Binary Correctness better
A vs. C	−4.87	Error Feedback without Previous Solution better
A vs. D	−3.86	Error Feedback with Previous Solution better
B vs. C	17.84	Binary Correctness better
B vs. D	19.02	Binary Correctness better
C vs. D	1.34	Difference not significant

Overall, the statistical analysis confirms the preliminary observations. “Binary Correctness with Previous Solution” achieves the highest accuracy and is significantly better than all other strategies. By contrast, the two error-feedback strategies perform comparably. These findings mark a reversal from Problem 2, where “Error Feedback with Previous Solution” was the most effective.

Observations from Test 2 on Feedback Loops

The second test involved executing the full feedback loop for each strategy for 100 trials, with up to 20 attempts per trial. The main results are summarised in Table 5.7. The complete trial-level data is provided as a supplementary zip file submitted to Moodle.

Table 5.7: Results of Second Test on Problem-3

Strategy	Trials Not Solved	Median Attempts to Solve
No Feedback	6	5
Binary Connectness with Previous Solution	7	1
Error Feedback without Previous Solution	23	7
Error Feedback with Previous Solution	47	3

Two main patterns emerge from this test. First, the strategies that incorporated previous solutions showed a disproportionately high number of unsolved trials. For example, “Binary Correctness,” despite achieving the best single-iteration accuracy, still failed to produce the correct solution in more trials than “No Feedback.” Similarly, both error feedback strategies had comparable accuracy, but the one with the previous solution failed far more often. Second, the median attempts indicate that when feedback strategies using the previous solution do succeed, they converge quickly. This suggests that if early attempts fail, the LLM is more

likely to continue failing in subsequent iterations, likely due to the influence of prior incorrect reasoning. In contrast, strategies that do not provide the previous solution treat each attempt independently, reducing the risk of such propagating errors.

This behaviour is evident from trial-level outputs. In trial 9 for “Error Feedback with Previous Solution,” the LLM initially derived the correct value of b . However, since calculated b was not a “nice” number, the LLM then revised its solution based on the prior solution. In trial 11, the LLM concluded that the given problem did not have a solution. Subsequent attempts in this trial, influenced by this incorrect solution, repeated the “no solution” conclusion. Similar behaviors were observed when the LLM inferred a “typo” in the problem, with all the following answers parroting the notion. These answers are provided in Appendix A.3.

These examples illustrate the key limitation of feedback strategies that use previous solutions, in that they may propagate errors and reduce overall problem-solving robustness.

5.3 Insights from Results

The experiments across the three test problems suggest that feedback can improve an LLM’s ability to solve mathematical problems; however, the relative effectiveness of feedback strategies is not uniform. Instead, their usefulness varies with the characteristics of the problem and the reasoning patterns that the LLM is able to adopt.

For Problems 1 and 2, feedback proved particularly valuable when it helped the model correct local computational mistakes, such as enumerating pairs of primes or identifying two-digit numbers with certain properties. In these settings, even a minimal correctness signal, when tied to the previous attempt, supported a meaningful improvement across iterations. Providing both the error and the context of the earlier solution enabled the LLM to directly relate the feedback to its reasoning, identify the error, and generate a corrected response.

In Problem 3, however, the situation was markedly different. Here, the LLM’s reasoning was misdirected from the outset, as it attempted to apply an incorrect theorem to solve the problem. Because the entire solution was flawed, error feedback could not easily guide the model back on track. Instead, the LLM often repeated the same invalid line of reasoning or elaborated on it further. In this context, both error messages and contextualised feedback proved far less effective, suggesting that such strategies are more limited when the model’s conceptual framing of the problem is incorrect rather than its calculations.

Taken together, these results do not point to a universally optimal feedback strategy. Rather, they highlight that the effectiveness of feedback is problem-dependent. Contextualised error feedback is highly effective in problems that permit systematic refinement, whereas binary correctness may be more robust when the problem does not lend itself to stepwise correction or when the model risks overfitting to prior incorrect reasoning. These findings underline the importance of considering not only the form of feedback but also the structure of the problem when designing iterative correction mechanisms for LLMs.

It should also be emphasised that the outcomes observed here are sensitive to experimental variables, particularly the prompts used and the phrasing of error messages. While the project sought to keep feedback as clear and consistent as possible, there remains the possibility that some error messages were misinterpreted by the LLM, thereby reducing their corrective potential or even reinforcing misconceptions. Further controlled studies are needed to disentangle the

effect of different feedback formulations from the intrinsic difficulty of the problems themselves.

Chapter 6

Conclusions

This project examined whether formal systems such as Lean can serve as effective feedback mechanisms for guiding LLMs in mathematical problem-solving. The central idea was to embed Lean-generated error messages and correctness checks into iterative feedback loops, enabling the LLM to refine its reasoning step-by-step. The experiments tested multiple strategies for delivering feedback, with results showing that no single approach is universally optimal. The usefulness of a strategy depends both on the structure of the problem and on the reasoning pathway chosen by the model.

The findings suggest that contextualised feedback referring to the model's most recent solution is particularly effective when the LLM's reasoning is close to correct and only requires local refinements. In such cases, targeted error messages can guide the model towards verifiable solutions. By contrast, when the initial reasoning is substantially flawed, simpler signals such as binary correctness can sometimes be more effective. This distinction highlights the importance of aligning feedback design with the model's problem-solving trajectory.

It is worth emphasising why Lean was used as a verifier rather than as a solver. While Lean could, in principle, compute the correct answers directly, this would miss the purpose of the project. The goal was not to replace the LLM's reasoning with Lean's, but to test whether the LLM can independently generate mathematically valid solutions under structured guidance. This setup demonstrated that LLMs can be nudged into more reliable problem-solving through structured verification,

6.1 Future Work

Several directions for future research follow from this work:

1. **Broader problem coverage:** Only three representative problems were studied here. Future work should include a wider variety of mathematical tasks, spanning more domains of the MATH dataset and beyond. This would allow clearer identification of the contexts in which different feedback strategies are most effective.
2. **Refinement of prompts and feedback messages:** Systematic variation in the structure, phrasing, and detail of feedback messages is needed to understand which elements are most influential. Controlled experiments could help disentangle whether brevity, precision, or contextual depth is most impactful.

3. **Incorporating multiple previous solutions:** This project relied only on the immediately preceding solution as context for feedback. Future studies could include several earlier attempts, creating a richer trace of the LLM’s reasoning trajectory. This may help the model learn not just from the last iteration but from patterns across multiple refinements.
4. **Handling incomplete solutions:** On several occasions, LLM produced only partial reasoning, or became caught in non-terminating loops. Methods for verifying intermediate steps, even when the full solution is missing, would allow feedback loops to remain useful. Designing systems that extract, check, and respond to completed fragments could guide the model more effectively toward a full solution.

6.2 Closing Remarks

This project demonstrates that Lean-guided feedback loops can substantially improve the reliability of LLMs in mathematical reasoning, but also that their success is highly context-dependent. Progress in this area will require not only more capable models, but also carefully designed, adaptive feedback mechanisms sensitive to both the problem and the model’s reasoning state. The findings underscore the promise of integrating formal verification systems with generative models to achieve solutions that are not only correct, but also mathematically rigorous.

Bibliography

- Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E.W., Radev, D. and Avigad, J., 2023. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics [Online]. 2302.12433, Available from: <https://arxiv.org/abs/2302.12433>.
- Barras, B., Boutin, S., Cornes, C., Courant, J., Filliâtre, J.C., Giménez, E., Herbelin, H., Huet, G., Muñoz, C., Murthy, C., Parent, C., Paulin-Mohring, C., Saïbi, A. and Werner, B., 1997. *The Coq Proof Assistant Reference Manual : Version 6.1* [Online]. (Research report RT-0203). Projet COQ. INRIA. Available from: <https://inria.hal.science/inria-00069968>.
- Boye, J. and Moell, B., 2025. Large language models and mathematical reasoning failures [Online]. 2502.11574, Available from: <https://arxiv.org/abs/2502.11574>.
- Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I. and Amodei, D., 2020. Language models are few-shot learners. *Corr* [Online], abs/2005.14165. 2005.14165, Available from: <https://arxiv.org/abs/2005.14165>.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., Oliveira Pinto, H.P. de, Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I. and Zaremba, W., 2021. Evaluating large language models trained on code. *Corr* [Online], abs/2107.03374. 2107.03374, Available from: <https://arxiv.org/abs/2107.03374>.
- Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C. and Schulman, J., 2021. Training verifiers to solve math word problems. *Corr* [Online], abs/2110.14168. 2110.14168, Available from: <https://arxiv.org/abs/2110.14168>.
- Dong, G., Yuan, H., Lu, K., Li, C., Xue, M., Liu, D., Wang, W., Yuan, Z., Zhou, C. and Zhou, J., 2024. How abilities in large language models are affected by supervised fine-tuning data composition [Online]. 2310.05492, Available from: <https://arxiv.org/abs/2310.05492>.

- Dong, K., Mahankali, A. and Ma, T., 2024. Formal theorem proving by rewarding llms to decompose proofs hierarchically [Online]. 2411.01829, Available from: <https://arxiv.org/abs/2411.01829>.
- First, E., Rabe, M.N., Ringer, T. and Brun, Y., 2023. Baldur: Whole-proof generation and repair with large language models [Online]. 2303.04910, Available from: <https://arxiv.org/abs/2303.04910>.
- Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev, A., Tassi, E. and Théry, L., 2013. A machine-checked proof of the odd order theorem. In: S. Blazy, C. Paulin-Mohring and D. Pichardie, eds. *Interactive theorem proving*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp.163–179.
- Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D. and Steinhardt, J., 2021. Measuring mathematical problem solving with the MATH dataset. *Corr* [Online], abs/2103.03874. 2103.03874, Available from: <https://arxiv.org/abs/2103.03874>.
- Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., Laroussilhe, Q. de, Gesmundo, A., Attariyan, M. and Gelly, S., 2019. Parameter-efficient transfer learning for NLP. *Corr* [Online], abs/1902.00751. 1902.00751, Available from: <http://arxiv.org/abs/1902.00751>.
- Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S. and Chen, W., 2021. Lora: Low-rank adaptation of large language models. *Corr* [Online], abs/2106.09685. 2106.09685, Available from: <https://arxiv.org/abs/2106.09685>.
- Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., Chen, Q., Peng, W., Feng, X., Qin, B. and Liu, T., 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *Acm transactions on information systems* [Online], 43(2), January, p.1–55. Available from: <https://doi.org/10.1145/3703155>.
- Jiang, A.Q., Welleck, S., Zhou, J.P., Li, W., Liu, J., Jamnik, M., Lacroix, T., Wu, Y. and Lample, G., 2023. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs [Online]. 2210.12283, Available from: <https://arxiv.org/abs/2210.12283>.
- Karatarakis, M., 2024. Leveraging large language models for autoformalizing theorems: A case study. *9th conference on artificial intelligence and theorem proving* [Online]. AITP 2024. Available from: <https://aitp-conference.org/2024/>.
- Kozyrev, A., Solovev, G., Khramov, N. and Podkopaev, A., 2024. Coqpilot, a plugin for llm-based generation of proofs. *Proceedings of the 39th ieee/acm international conference on automated software engineering* [Online]. ACM, ASE '24, p.2382–2385. Available from: <https://doi.org/10.1145/3691620.3695357>.
- Lai, X., Tian, Z., Chen, Y., Yang, S., Peng, X. and Jia, J., 2024a. Step-dpo: Step-wise preference optimization for long-chain reasoning of llms [Online]. Available from: <https://doi.org/10.48550/arXiv.2406.18629>.
- Lai, X., Tian, Z., Chen, Y., Yang, S., Peng, X. and Jia, J., 2024b. Step-dpo: Step-wise preference optimization for long-chain reasoning of llms [Online]. 2406.18629, Available from: <https://arxiv.org/abs/2406.18629>.
- Lester, B., Al-Rfou, R. and Constant, N., 2021. The power of scale for parameter-efficient

- prompt tuning. *Corr* [Online], abs/2104.08691. 2104.08691, Available from: <https://arxiv.org/abs/2104.08691>.
- Li, X.L. and Liang, P., 2021. Prefix-tuning: Optimizing continuous prompts for generation. *Corr* [Online], abs/2101.00190. 2101.00190, Available from: <https://arxiv.org/abs/2101.00190>.
- Lu, M., Delaware, B. and Zhang, T., 2024. Proof automation with large language models [Online]. 2409.14274, Available from: <https://arxiv.org/abs/2409.14274>.
- Moura, L. de, Kong, S., Avigad, J., Doorn, F.V. and Raumer, J. von, 2018. The Lean Theorem Prover (system description) [Online]. 6. Available from: <https://doi.org/10.1184/R1/6492815.v1>.
- Nipkow, T., Wenzel, M. and Paulson, L.C., 2002. *Isabelle/hol: a proof assistant for higher-order logic*. Berlin, Heidelberg: Springer-Verlag.
- Song, P., Yang, K. and Anandkumar, A., 2025. Lean copilot: Large language models as copilots for theorem proving in lean [Online]. 2404.12534, Available from: <https://arxiv.org/abs/2404.12534>.
2019. the lean mathematical library. *Corr* [Online], abs/1910.09336. 1910.09336, Available from: <http://arxiv.org/abs/1910.09336>.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L.u. and Polosukhin, I., 2017. Attention is all you need. In: I. Guyon, U.V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan and R. Garnett, eds. *Advances in neural information processing systems* [Online]. Curran Associates, Inc., vol. 30. Available from: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.
- Wang, H., Xin, H., Zheng, C., Li, L., Liu, Z., Cao, Q., Huang, Y., Xiong, J., Shi, H., Xie, E., Yin, J., Li, Z., Liao, H. and Liang, X., 2023. Lego-prover: Neural theorem proving with growing libraries [Online]. 2310.00656, Available from: <https://arxiv.org/abs/2310.00656>.
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E.H., Le, Q. and Zhou, D., 2022a. Chain of thought prompting elicits reasoning in large language models. *Corr* [Online], abs/2201.11903. 2201.11903, Available from: <https://arxiv.org/abs/2201.11903>.
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V. and Zhou, D., 2022b. Chain-of-thought prompting elicits reasoning in large language models. *Proceedings of the 36th international conference on neural information processing systems*. Red Hook, NY, USA: Curran Associates Inc., NIPS '22.
- Williams, S. and Huckle, J., 2024. Easy problems that llms get wrong [Online]. 2405.19616, Available from: <https://arxiv.org/abs/2405.19616>.
- Wu, Y., Jiang, A.Q., Li, W., Rabe, M.N., Staats, C.E., Jamnik, M. and Szegedy, C., 2022. Autoformalization with large language models. In: A.H. Oh, A. Agarwal, D. Belgrave and K. Cho, eds. *Advances in neural information processing systems* [Online]. Available from: <https://openreview.net/forum?id=IUikebJ1Bf0>.
- Yang, K., Swope, A.M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R. and

- Anandkumar, A., 2023. Leandojo: Theorem proving with retrieval-augmented language models [Online]. 2306.15626, Available from: <https://arxiv.org/abs/2306.15626>.
- Zheng, K., Han, J.M. and Polu, S., 2021. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *Corr* [Online], abs/2109.00110. 2109.00110, Available from: <https://arxiv.org/abs/2109.00110>.

Appendix A

Solutions Generated By the LLM

A.1 Initial Answers for Selected Problems

Problem 1

****Steps:****

1. Let the roots be α and β , where α and β are prime numbers.
2. The polynomial can be written as $(x - \alpha)(x - \beta) = x^2 - (\alpha + \beta)x + \alpha\beta$.
3. Thus, $m = \alpha + \beta$ and $n = \alpha\beta$.
4. Given $m < 20$, we have $\alpha + \beta < 20$.

****List of prime pairs (α, β) with $\alpha \leq \beta$ and $\alpha + \beta < 20$:**

- (2, 2), (2, 3), (2, 5), (2, 7), (2, 11), (2, 13), (2, 17)
- (3, 3), (3, 5), (3, 7), (3, 11), (3, 13)
- (5, 5), (5, 7), (5, 11)
- (7, 7), (7, 11)
- (11, 11)

****Calculate $n = \alpha\beta$ for each pair:**

- 4, 6, 10, 14, 22, 26, 34
- 9, 15, 21, 33, 39
- 25, 35, 55
- 49, 77
- 121

****Count distinct values of n :**

- Total distinct n : 17

****Final Answer:****

$\boxed{17}$

Problem 2

****Steps:****

1. List all two-digit primes using digits 1-9 without repetition.
2. Identify four primes that use each digit from 1, 2, 3, 4, 5, 6, 7, 9 exactly once.

****Possible primes:****

- 13, 29, 41, 67 (sum: 150)

- 13, 29, 47, 61 (sum: 150)
- 17, 23, 41, 59 (sum: 140)
- 17, 23, 49, 61 (sum: 150)
- 19, 23, 41, 67 (sum: 150)
- 19, 23, 47, 61 (sum: 150)
- 19, 29, 41, 53 (sum: 142)
- 19, 29, 47, 53 (sum: 148)

****Explanation:****

The only valid combination that uses all digits exactly once is 13, 29, 47, 61 (sum: 150).

****Final Answer:****

$\boxed{150}$

Problem 3

****Steps:****

1. Since $(3 + \sqrt{5})$ is a root, its conjugate $(3 - \sqrt{5})$ is also a root.
2. Sum of roots: $((3 + \sqrt{5}) + (3 - \sqrt{5})) = 6$.
3. Product of roots: $((3 + \sqrt{5})(3 - \sqrt{5})) = 9 - 5 = 4$.
4. The equation is $(x^2 - (\text{sum})x + (\text{product})) = 0$, so $(x^2 - 6x + 4 = 0)$.
5. Compare with given equation $(x^2 - 3x + b = 0)$ to find $(b = 4)$.

****Final Answer:****

$\boxed{4}$

A.2 Answers from Tests on Problem 1

No Feedback

The following text is the verbatim reproduction of the LLM's generated solution where the final answer is correct, but the reasoning was flawed:

****Steps:****

1. ****Identify the roots:****
Let the roots be prime numbers (p) and (q) .
The polynomial is $(x^2 - (p+q)x + pq)$, so $(m = p + q)$ and $(n = pq)$.
2. ****List possible primes for (p) and (q) :**
Primes (< 20) : 2, 3, 5, 7, 11, 13, 17, 19.
3. ****Find pairs $((p, q))$ where $(p + q < 20)$:**
 - $((2, 2))$, $((2, 3))$, $((2, 5))$, $((2, 7))$, $((2, 11))$,
 $((2, 13))$, $((2, 17))$
 - $((3, 3))$, $((3, 5))$, $((3, 7))$, $((3, 11))$, $((3, 13))$
 - $((5, 5))$, $((5, 7))$, $((5, 11))$
 - $((7, 7))$, $((7, 11))$
 - $((11, 11))$


```

4. **Calculate  $n = pq$  for each pair:**
   - 4, 6, 10, 14, 22, 26, 34
   - 9, 15, 21, 33, 39
   - 25, 35, 55
   - 49, 77
   - 121
5. **Count distinct  $n$  values  $(n < 20 \times 20)$ :**
   Valid  $n$ : 4, 6, 9, 10, 14, 15, 21, 22, 25, 26, 33, 34, 35, 39, 49,
   55, 77, 121.
**Final Answer:**
 $\boxed{18}$ 

```

Error Feedback

The following answers are verbatim reproductions of the LLM generated solutions for the first run of the feedback loop:

Attempt-1

DeepSeek Response (natural language solution):

****Steps/Formula:****

- Let the roots be p and q , where p and q are positive prime integers.
- The polynomial is $x^2 - (p+q)x + pq$, so $m = p + q$ and $n = pq$.
- Given $m < 20$, find all pairs (p, q) such that $p + q < 20$.

****Explanation:****

- List all prime pairs (p, q) where $p \leq q$ and $p + q < 20$.
- Calculate $n = pq$ for each pair and count the distinct values.

****Prime Pairs and n :****

- $(2, 2): n = 4$
- $(2, 3): n = 6$
- $(2, 5): n = 10$
- $(2, 7): n = 14$
- $(2, 11): n = 22$ (invalid, $m = 13 < 20$)
- $(2, 13): n = 26$ (invalid, $m = 15 < 20$)
- $(2, 17): n = 34$ (invalid, $m = 19 < 20$)
- $(3, 3): n = 9$
- $(3, 5): n = 15$
- $(3, 7): n = 21$
- $(3, 11): n = 33$ (invalid, $m = 14 < 20$)
- $(3, 13): n = 39$ (invalid, $m = 16 < 20$)
- $(5, 5): n = 25$
- $(5, 7): n = 35$
- $(5, 11): n = 55$ (invalid, $m = 16 < 20$)
- $(7, 7): n = 49$
- $(7, 11): n = 77$ (invalid, $m = 18 < 20$)
- $(11, 11): n = 121$ (invalid, $m = 22 \geq 20$)

****Valid n values:****

4, 6, 9, 10, 14, 15, 21, 25, 35, 49

****Final Answer:****

$\boxed{10}$

Attempt-2****Steps/Formula:****

1. Let the roots be α and β , where α and β are positive prime integers.
2. The polynomial is $x^2 - (p+q)x + pq$, so $m = p + q$ and $n = pq$.
3. Given $m < 20$, find all pairs (p, q) such that $p + q < 20$.

****Explanation:****

- List all prime pairs (p, q) where $p \leq q$ and $p + q < 20$.
- Calculate $n = pq$ for each pair and count the distinct values.

****Prime Pairs and n values:****

- (2, 2): $n = 4$
- (2, 3): $n = 6$
- (2, 5): $n = 10$
- (2, 7): $n = 14$
- (2, 11): $n = 22$
- (2, 13): $n = 26$
- (2, 17): $n = 34$
- (3, 3): $n = 9$
- (3, 5): $n = 15$
- (3, 7): $n = 21$
- (3, 11): $n = 33$
- (3, 13): $n = 39$
- (5, 5): $n = 25$
- (5, 7): $n = 35$
- (5, 11): $n = 55$
- (7, 7): $n = 49$
- (7, 11): $n = 77$

****Valid n values:****

4, 6, 9, 10, 14, 15, 21, 22, 25, 26, 33, 34, 35, 39, 49, 55, 77

Attempt-3****Steps/Formula:****

1. Let the roots be α and β , where α and β are positive prime integers.
2. The polynomial is $x^2 - (p+q)x + pq$, so $m = p + q$ and $n = pq$.
3. Given $m < 20$, find all pairs (p, q) such that $p + q < 20$.

****Explanation:****

- List all prime pairs (p, q) where $p \leq q$ and $p + q < 20$.
- Calculate $n = pq$ for each pair and count the distinct values.

****Prime Pairs and n values:****

- (2, 2): $n = 4$
- (2, 3): $n = 6$
- (2, 5): $n = 10$
- (2, 7): $n = 14$
- (2, 11): $n = 22$
- (2, 13): $n = 26$

```

- (2, 17): \(\ n = 34 \)
- (3, 3): \(\ n = 9 \)
- (3, 5): \(\ n = 15 \)
- (3, 7): \(\ n = 21 \)
- (3, 11): \(\ n = 33 \)
- (3, 13): \(\ n = 39 \)
- (5, 5): \(\ n = 25 \)
- (5, 7): \(\ n = 35 \)
- (5, 11): \(\ n = 55 \)
- (5, 13): \(\ n = 65 \)
- (7, 7): \(\ n = 49 \)
- (7, 11): \(\ n = 77 \)
- (11, 11): \(\ n = 121 \) (invalid since \(\ 11 + 11 = 22 \geq 20 \),
  so excluded.
**Valid \(\ n \) values:**
4, 6, 9, 10, 14, 15, 21, 22, 25, 26, 33, 34, 35, 39, 49, 55, 65, 77
**Final Answer:**
\(\boxed{18}\)

```

A.3 Answers from Tests on Problem 3

The following answer is from trial 9 from Test 2 of Problem 3:

"The given root is $(3 + \sqrt{5})$. Substitute it into the equation:

$$[(3 + \sqrt{5})^2 - 3(3 + \sqrt{5}) + b = 0.$$

Compute:

$$[(9 + 6\sqrt{5} + 5) - (9 + 3\sqrt{5}) + b = 0,$$

$$[(14 + 6\sqrt{5}) - 9 - 3\sqrt{5} + b = 0,$$

$$[5 + 3\sqrt{5} + b = 0.$$

Since $(\sqrt{5})$ is irrational, the coefficient must vanish: $(3\sqrt{5})$ implies no real (b) satisfies this. However, the problem states it is a root, so likely (b) is chosen such that the conjugate is also a root. But the substitution shows inconsistency. Alternatively, check: if $(3+\sqrt{5})$ is a root, then (b) must satisfy:

$$b = -[(3+\sqrt{5})^2 - 3(3+\sqrt{5})] = -[14 + 6\sqrt{5} - 9 - 3\sqrt{5}] = -[5 + 3\sqrt{5}] = -5 - 3\sqrt{5}.$$

But (b) is not a nice number. Perhaps the polynomial has rational coefficients? Actually, the problem says "compute (b) ", implying it is a constant. Wait, the polynomial is $(x^2 - 3x + b = 0)$, so (b) is the constant term. For $(3+\sqrt{5})$ to be a root, (b) must be $(-5 - 3\sqrt{5})$, but that is not a simple number.

Maybe there is a misinterpretation: the problem might be that $(3+\sqrt{5})$ is a root, and we need (b) . But from above, $(b = -5 - 3\sqrt{5})$. However, this is not an integer. Perhaps the polynomial has integer coefficients? But it is not stated.

Wait, the original solution used the conjugate, but that requires the polynomial to have real coefficients (which it does, since coefficients are 1, -3, b). So if $(3+\sqrt{5})$ is a root, then $(3-\sqrt{5})$ is also a root. Then the product is $(b = (3+\sqrt{5})(3-\sqrt{5}) = 9-5=4)$.

So $(b=4)$. Then check: if $(b=4)$, the roots are given by quadratic formula: $(x = \frac{3 \pm \sqrt{9-16}}{2} = \frac{3 \pm i\sqrt{7}}{2})$, which are not $(3 \pm \sqrt{5})$. So there is a contradiction.

Therefore, the only consistent way is to assume that the polynomial has rational coefficients, so the conjugate must be a root. Then $(b=4)$. But then $(3+\sqrt{5})$ is not actually a root. However, the problem states it is a root, so we proceed with the conjugate method.

Given the instruction "retry from scratch", and the original solution was to use the conjugate, we do that.

Thus, the other root is $(3 - \sqrt{5})$. The product is $(b = (3+\sqrt{5})(3-\sqrt{5}) = 9 - 5 = 4)$.

So $(b=4)$.

$\boxed{4}$

The following answer is from Trial 11 of Test 2:

"The given root is $(3 + \sqrt{5})$. Since the quadratic has rational coefficients, the conjugate $(3 - \sqrt{5})$ is also a root.

By Vieta's formulas, the sum of the roots is (3) , and the product is (b) . Check the sum: $((3 + \sqrt{5}) + (3 - \sqrt{5}) = 6)$, but the sum should be (3) (from the coefficient). This is a contradiction.

Therefore, $(3 + \sqrt{5})$ cannot be a root of $(x^2 - 3x + b = 0)$ for any rational (b) .

The problem is inconsistent; there is no such (b) .

$\boxed{\text{no solution}}$

Appendix B

Lean Code Templates

This appendix presents the full Lean code template for each problem, including all relevant Mathlib imports, for reference and reproducibility.

B.1 Problem-1

```
import Mathlib.Data.Nat.Prime.Basic
import Mathlib.Data.List.Basic
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Card
import Init.Data.Nat.Basic

open Nat
open Finset
open List

def validPrimePairs : List (ℕ × ℕ) :=
  let primes := (List.range 20).filter Nat.Prime
  List.foldr (· ++ ·) [] (
    primes.map (fun p =>
      (List.range 20).filter Nat.Prime |>.filterMap (fun q =>
        if p ≤ q & p + q < 20 then some (p, q) else none))
    )

def inputPairsList : List (ℕ × ℕ) :=
  <PRIME_PAIRS_LIST>

theorem verify_eqv : validPrimePairs.isEqv inputPairsList (· == ·) = true := by
  decide

def nValues : List ℕ := validPrimePairs.map (fun (p, q) => p * q)

def distinctNValues : Finset ℕ := nValues.toFinset

def inputNValues : Finset ℕ :=
  <N_VALUES_SET>

theorem verify_nValues_correct : distinctNValues = inputNValues := by
  decide

def inputCount : Nat := <COUNT>

theorem verify_cardinality_correct : distinctNValues.card = inputCount := by
  rfl
```

B.2 Problem-2

```
import Mathlib.Data.List.Basic
```

```

open Nat List
def digits (n : Nat) : List Nat :=
  [n / 10, n % 10]
def digitsDistinct (n : Nat) : Bool :=
  let ds := digits n
  ds.length = ds.eraseDup.length
def listElemBool (a : Nat) (l : List Nat) : Bool :=
  l.any (fun x => x = a)
def digitsInSet (allowed : List Nat) (n : Nat) : Bool :=
  let ds := digits n
  ds.all (fun d => listElemBool d allowed)
def isPrimeBool (n : Nat) : Bool :=
  if n < 2 then false else
    !((List.range (n - 2)).any (fun d => n % (d + 2) = 0))
def allDigits (nums : List Nat) : List Nat :=
  nums.flatMap digits
def allDigitsDistinct (nums : List Nat) : Bool :=
  let ds := allDigits nums
  ds.length = ds.eraseDup.length
def allowedDigits : List Nat := [1, 2, 3, 4, 5, 6, 7, 9]
def inputList : List Nat := <REPLACE_WITH_SOLUTION_LIST>
def inputSum : Nat := <REPLACE_WITH_SOLUTION_SUM>
def allPrime : Bool := inputList.all isPrimeBool
def noDigit8 : Bool := inputList.all (digitsInSet allowedDigits)
def allDistinctDigits : Bool := allDigitsDistinct inputList
def sumOfList : Nat := inputList.foldl (· + ·) 0
def sumCorrect : Bool := inputList.foldl (· + ·) 0 = inputSum
#eval ("All prime?", allPrime)
#eval ("No digit 8?", noDigit8)
#eval ("All digits distinct?", allDistinctDigits)
#eval ("Sum correct?", sumCorrect)

```

B.3 Problem 3

```

def givenRoot : Float := 3.0 + 2.2360679775
def input_b : Float := {Float_Value_Input_b}

def evalPoly (x b : Float) : Float := x*x - 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps

def isRoot : Bool := approxZero (evalPoly givenRoot input_b)

#eval ("Is 3+sqrt5 a root of polynomial?", isRoot)

```

Appendix C

Prompts Used

Here, the different prompts used in each problem for generating responses from LLM are listed.

C.1 Problem 1

The purpose of each prompt is given by the following descriptions:

problem The problem statement.

```
problem = (  
"Suppose the roots of the polynomial  $x^2 - mx + n$  are positive prime integers (not necessarily distinct). "  
"Given that  $m < 20$ , how many possible values of  $n$  are there?"  
)
```

INITIAL_PROMPT Prompt for generating the initial answer for the problem.

```
INITIAL_PROMPT = (  
"Given the following math word problem, return only the required steps/formula, only required explanation,  
and the final answer. "  
"Only main explanation, reduce the number of steps. No extra text. \n\n."  
"Problem: {problem}"  
)
```

RETRY_SOLVE_PROMPT_TEMPLATE Prompt for providing feedback and asking the LLM to retry the problem.

```
RETRY_SOLVE_PROMPT_TEMPLATE = (  
"Given the following math problem:\n"  
"{problem}\n\n"  
"You previously solved it like this:\n"  
"{previous_answer}\n\n"  
"But when it had the following error:\n"  
"{error}\n\n"  
"Retry solving the problem again carefully. "  
"Return only the required steps/formula, only required explanation, and the final answer. "  
"Only main explanation, reduce the number of steps. No extra text. \n\n."  
)
```

LEAN_CODE_GENERATION_PROMPT Lean template and few-shot prompts for generating the Lean code.

```
LEAN_CODE_GENERATION_PROMPT = ""  
You are given a natural language solution to a math problem.  
Extract:  
1. The list of valid prime pairs (p, q)  
2. The set of distinct n values  
3. The count of distinct n values  
Then output Lean code using the template exactly. Replace placeholders with extracted values.  
  
Template:  
import Mathlib.Data.Nat.Prime.Basic  
import Mathlib.Data.List.Basic  
import Mathlib.Data.Finset.Basic  
import Mathlib.Data.Finset.Card  
import Init.Data.Nat.Basic
```

```

open Nat
open Finset
open List
def validPrimePairs : List (ℕ × ℕ) :=
  let primes := (List.range 20).filter Nat.Prime
  List.foldr (· ++ ·) [] (
    primes.map (fun p =>
      (List.range 20).filter Nat.Prime |>.filterMap (fun q =>
        if p + q < 20 then some (p, q) else none))
    )
def inputPairsList : List (ℕ × ℕ) :=
  <PRIME_PAIRS_LIST>
theorem verify_eqv : validPrimePairs.isEqv inputPairsList (· == ·) = true := by
  decide
def nValues : List ℕ := validPrimePairs.map (fun (p, q) => p * q)
def distinctNValues : Finset ℕ := nValues.toFinset
def inputNValues : Finset ℕ :=
  <N_VALUES_SET>
theorem verify_nValues_correct : distinctNValues = inputNValues := by
  decide
def inputCount : Nat := <COUNT>
theorem verify_cardinality_correct : distinctNValues.card = inputCount := by
  rfl

```

Use following problems as examples to generate the Lean code with given output format.

Example 1

Input:

****Steps/Formula:****

1. Let the roots be $\backslash(p)$ and $\backslash(q)$, where $\backslash(p)$ and $\backslash(q)$ are positive prime integers.
2. The polynomial is $\backslash(x^2 - (p+q)x + pq)$, so $\backslash(m = p + q)$ and $\backslash(n = pq)$.
3. Given $\backslash(m < 20)$, find all pairs $\backslash((p, q))$ such that $\backslash(p + q < 20)$.

****Explanation:****

- List all prime pairs $\backslash((p, q))$ where $\backslash(p \leq q)$ and $\backslash(p + q < 20)$.
- Calculate $\backslash(n = pq)$ for each pair and count the distinct values.

****Prime Pairs and $\backslash(n)$:**

- (2, 2): $\backslash(n = 4)$
- (2, 3): $\backslash(n = 6)$
- (2, 5): $\backslash(n = 10)$
- (2, 7): $\backslash(n = 14)$
- (2, 11): $\backslash(n = 22)$ (invalid, $\backslash(m = 13 < 20)$)
- (2, 13): $\backslash(n = 26)$ (invalid, $\backslash(m = 15 < 20)$)
- (2, 17): $\backslash(n = 34)$ (invalid, $\backslash(m = 19 < 20)$)
- (3, 3): $\backslash(n = 9)$
- (3, 5): $\backslash(n = 15)$
- (3, 7): $\backslash(n = 21)$
- (3, 11): $\backslash(n = 33)$ (invalid, $\backslash(m = 14 < 20)$)
- (3, 13): $\backslash(n = 39)$ (invalid, $\backslash(m = 16 < 20)$)
- (5, 5): $\backslash(n = 25)$
- (5, 7): $\backslash(n = 35)$
- (5, 11): $\backslash(n = 55)$ (invalid, $\backslash(m = 16 < 20)$)
- (7, 7): $\backslash(n = 49)$
- (7, 11): $\backslash(n = 77)$ (invalid, $\backslash(m = 18 < 20)$)
- (11, 11): $\backslash(n = 121)$ (invalid, $\backslash(m = 22 \geq 20)$)

****Valid $\backslash(n)$ values:****

4, 6, 9, 10, 14, 15, 21, 25, 35, 49

****Final Answer:****

$\boxed{10}$

Output:

```

import Mathlib.Data.Nat.Prime.Basic
import Mathlib.Data.List.Basic
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Card
import Init.Data.Nat.Basic

```

open Nat


```

open Finset
open List
def validPrimePairs : List (ℕ × ℕ) :=
  let primes := (List.range 20).filter Nat.Prime
  List.foldr (· ++ ·) [] (
    primes.map (fun p =>
      (List.range 20).filter Nat.Prime |>.filterMap (fun q =>
        if p ≤ q & p + q < 20 then some (p, q) else none))
    )
def inputPairsList : List (ℕ × ℕ) :=
  [
    (2, 2),
    (2, 3),
    (2, 5),
    (2, 7),
    (3, 3),
    (3, 5),
    (3, 7),
    (5, 5),
    (5, 7),
    (7, 7)
  ]
theorem verify_eqv : validPrimePairs.isEqv inputPairsList (· == ·) = true := by
  decide
def nValues : List ℕ := validPrimePairs.map (fun (p, q) => p * q)
def distinctNValues : Finset ℕ := nValues.toFinset
def inputNValues : Finset ℕ :=
  ({4, 6, 9, 10, 14, 15, 21, 25, 35, 49} : Finset ℕ)
theorem verify_nValues_correct : distinctNValues = inputNValues := by
  decide
def inputCount : Nat := 10
theorem verify_cardinality_correct : distinctNValues.card = inputCount := by
  rfl

```

Example 2:

Input:

****Steps/Formula:****

1. Let the roots be $\backslash(p \backslash)$ and $\backslash(q \backslash)$, where $\backslash(p \backslash)$ and $\backslash(q \backslash)$ are positive prime integers.
2. The polynomial is $\backslash(x^2 - (p+q)x + pq \backslash)$, so $\backslash(m = p + q \backslash)$ and $\backslash(n = pq \backslash)$.
3. Given $\backslash(m < 20 \backslash)$, find all pairs $\backslash(p, q \backslash)$ such that $\backslash(p + q < 20 \backslash)$.

****Explanation:****

- List all prime pairs $\backslash(p, q \backslash)$ where $\backslash(p \leq q \backslash)$ and $\backslash(p + q < 20 \backslash)$.
- Calculate $\backslash(n = pq \backslash)$ for each pair and count the distinct values.

****Prime Pairs and $\backslash(n \backslash)$:**

- (2, 2): $\backslash(n = 4 \backslash)$
- (2, 3): $\backslash(n = 6 \backslash)$
- (2, 5): $\backslash(n = 10 \backslash)$
- (2, 7): $\backslash(n = 14 \backslash)$
- (2, 11): $\backslash(n = 22 \backslash)$
- (2, 13): $\backslash(n = 26 \backslash)$
- (2, 17): $\backslash(n = 34 \backslash)$
- (3, 3): $\backslash(n = 9 \backslash)$
- (3, 5): $\backslash(n = 15 \backslash)$
- (3, 7): $\backslash(n = 21 \backslash)$
- (3, 11): $\backslash(n = 33 \backslash)$
- (3, 13): $\backslash(n = 39 \backslash)$
- (5, 5): $\backslash(n = 25 \backslash)$
- (5, 7): $\backslash(n = 35 \backslash)$
- (5, 11): $\backslash(n = 55 \backslash)$
- (7, 7): $\backslash(n = 49 \backslash)$
- (7, 11): $\backslash(n = 77 \backslash)$

****Valid $\backslash(n \backslash)$ values:****

4, 6, 9, 10, 14, 15, 21, 22, 25, 26, 33, 34, 35, 39, 49, 55, 77

****Final Answer:****

$\backslash(\boxed{17}) \backslash)$

Output:

```

import Mathlib.Data.Nat.Prime.Basic
import Mathlib.Data.List.Basic
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Card
import Init.Data.Nat.Basic

open Nat
open Finset
open List
def validPrimePairs : List (N × N) :=
  let primes := (List.range 20).filter Nat.Prime
  List.foldr (· ++ ·) [] (
    primes.map (fun p =>
      (List.range 20).filter Nat.Prime |>.filterMap (fun q =>
        if p ≤ q then some (p, q) else none))
  )
def inputPairsList : List (N × N) :=
  [
    (2, 2),
    (2, 3),
    (2, 5),
    (2, 7),
    (2, 11),
    (2, 13),
    (2, 17),
    (3, 3),
    (3, 5),
    (3, 7),
    (3, 11),
    (3, 13),
    (5, 5),
    (5, 7),
    (5, 11),
    (7, 7),
    (7, 11)
  ]
theorem verify_eqv : validPrimePairs.isEqv inputPairsList (· == ·) = true := by
  decide
def nValues : List N := validPrimePairs.map (fun (p, q) => p * q)
def distinctNValues : Finset N := nValues.toFinset
def inputNValues : Finset N :=
  ({4, 6, 9, 10, 14, 15, 21, 22, 25, 26, 33, 34, 35, 39, 49, 55, 77}) : Finset N
theorem verify_nValues_correct : distinctNValues = inputNValues := by
  decide
def inputCount : Nat := 17
theorem verify_cardinality_correct : distinctNValues.card = inputCount := by
  rfl

Input:
{solution_text}

Output:
"""

```

LEAN_OUTPUT_INTERPRETATION_PROMPT Prompt used to convert the Lean output to natural Language error feedback.

```

LEAN_OUTPUT_INTERPRETATION_PROMPT = (
  """Given the following Lean code:
    {lean_code}

    and the following error output:
    {lean_output}

    interpret the error and provide a concise natural language explanation of what went wrong.
    Your error should be one of the following:
    1. The list of prime pairs generated in your answer were incorrect. Retry the problem again,
    this time going over each pair carefully, and making sure all valid prime pairs are included.
    2. The list of prime pairs are correct. However, the list distinct n values are incorrect.
    3. The list of prime pairs and the list distinct n values are correct. However, the count of
    distinct n values is incorrect.

    Consider the following example as a reference:

```

```

Input:
Given the following Lean code:
import Mathlib.Data.Nat.Prime.Basic
import Mathlib.Data.List.Basic
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Card
import Init.Data.Nat.Basic

open Nat
open Finset
open List

def validPrimePairs : List (ℕ × ℕ) :=
let primes := (List.range 20).filter Nat.Prime
List.foldr (· ++ ·) [] (
  primes.map (fun p =>
    (List.range 20).filter Nat.Prime |>.filterMap (fun q =>
      if p < q & p + q < 20 then some (p, q) else none))
)

def inputPairsList : List (ℕ × ℕ) :=
[
  (2, 2),
  (2, 3),
  (2, 5),
  (2, 7),
  (2, 11),
  (2, 13),
  (2, 17),
  (3, 3),
  (3, 5),
  (3, 7),
  (3, 11),
  (3, 13),
  (5, 5),
  (5, 7),
  (5, 11),
  (7, 7),
  (7, 11)
]

theorem verify_eqv : validPrimePairs.isEqv inputPairsList (· == ·) = true := by
  decide

def nValues : List ℕ := validPrimePairs.map (fun (p, q) => p * q)

def distinctNValues : Finset ℕ := nValues.toFinset

def inputNValues : Finset ℕ :=
  ({4, 6, 9, 10, 14, 15, 21, 22, 25, 26, 33, 34, 35, 39, 49, 55, 77} : Finset ℕ)

theorem verify_nValues_correct : distinctNValues = inputNValues := by
  decide

def inputCount : Nat := 17

theorem verify_cardinality_correct : distinctNValues.card = inputCount := by
  rfl

and the following error output:
error: MathlibProject.lean:35:2: tactic 'decide' proved that the proposition
(validPrimePairs.isEqv inputPairsList fun x1 x2 x1 == x2) = true
is false
error: MathlibProject.lean:45:2: tactic 'decide' proved that the proposition
distinctNValues = inputNValues
is false
error: MathlibProject.lean:50:2: tactic 'rfl' failed, the left-hand side
#distinctNValues
is not definitionally equal to the right-hand side
inputCount
#distinctNValues = inputCount
error: Lean exited with code 1

```

```

Some required builds logged failures:
- MathlibProject

interpret the error and provide a concise natural language explanation of what went wrong.
Your error should be one of the following:
1. The list of prime pairs generated in your answer were incorrect. Retry the problem again,
this time going over each pair carefully, and making sure all valid prime pairs are included.
2. The set of distinct n values is incorrect.
3. The count of distinct n values is incorrect.

Output:
The list of prime pairs generated in your answer were incorrect. Retry the problem again,
this time going over each pair carefully, and making sure all valid prime pairs are included.
"""
)

```

C.2 Problem 2

```

problem = (
    "The digits $1$, $2$, $3$, $4$, $5$, $6$, $7$, and $9$ are used to form four two-digit prime numbers, "
    "with each digit used exactly once. What is the sum of these four primes? "
)

INITIAL_PROMPT = (
    "Given the following math word problem, provide a concise and brief solution: "
    "include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format: \boxed{{answer}}.\n"
    "Do not include detailed steps, just enough explanation to justify the answer."
    "Problem: {problem}"
)

BINARY_CORRECT_PROMPT = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "You solved it like this:\n"
    "{previous_answer}\n\n"
    "This answer was incorrect. Retry the problem from scratch.\n"
    "Provide a concise and brief solution: include only the minimal necessary explanation "
    "and the final answer.\n"
    "Return the answer in the format: \boxed{{answer}}.\n"
    "Do not include detailed steps, just enough explanation to justify the answer."
)

RETRY_SOLVE_ERROR_PROMPT = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "You previously solved it like this:\n"
    "{previous_answer}\n\n"
    "But it had the following error:\n"
    "{error}\n\n"
    "Retry from scratch\n"
    "Provide a concise and brief solution: include only the minimal necessary explanation "
    "and the final answer.\n"
    "Return the answer in the format: \boxed{{answer}}.\n"
    "Do not include detailed steps, just enough explanation to justify the answer."
)

RETRY_SOLVE_WITHOUT_PREVIOUS_ANSWER_PROMPT = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "Your previous answer had the following error:\n"
    "{error}\n\n"
    "Retry the problem from scratch.\n"
    "Provide a concise and brief solution: include only the minimal necessary explanation "
    "and the final answer.\n"
    "Return the answer in the format: \boxed{{answer}}.\n"
    "Do not include detailed steps, just enough explanation to justify the answer."
)

LEAN_CODE_GENERATION_PROMPT = ""
You are given a natural language solution to a math problem.
Extract the valid list of 4 two digit numbers (used to calculate the sum) from the solution
and the sum of the list (final answer), then output Lean code using the template exactly.

```

Replace placeholders with extracted values.

```

Template:
import Mathlib.Data.List.Basic
open Nat List
def digits (n : Nat) : List Nat :=
  [n / 10, n % 10]
def digitsDistinct (n : Nat) : Bool :=
  let ds := digits n
  ds.length = ds.eraseDup.length
def listElemBool (a : Nat) (l : List Nat) : Bool :=
  l.any (fun x => x = a)
def digitsInSet (allowed : List Nat) (n : Nat) : Bool :=
  let ds := digits n
  ds.all (fun d => listElemBool d allowed)
def isPrimeBool (n : Nat) : Bool :=
  if n < 2 then false else
    !((List.range (n - 2)).any (fun d => n % (d + 2) = 0))
def allDigits (nums : List Nat) : List Nat :=
  nums.flatMap digits
def allDigitsDistinct (nums : List Nat) : Bool :=
  let ds := allDigits nums
  ds.length = ds.eraseDup.length
def allowedDigits : List Nat := [1, 2, 3, 4, 5, 6, 7, 9]
def inputList : List Nat := <REPLACE_WITH_SOLUTION_LIST>
def inputSum : Nat := <REPLACE_WITH_SOLUTION_SUM>
def allPrime : Bool := inputList.all isPrimeBool
def noDigit8 : Bool := inputList.all (digitsInSet allowedDigits)
def allDistinctDigits : Bool := allDigitsDistinct inputList
def sumOfList : Nat := inputList.foldl (· + ·) 0
def sumCorrect : Bool := inputList.foldl (· + ·) 0 = inputSum
#eval ("All prime?", allPrime)
#eval ("No digit 8?", noDigit8)
#eval ("All digits distinct?", allDistinctDigits)
#eval ("Sum correct?", sumCorrect)

```

Use following problems as examples to generate the Lean code with given output format.

Example 1:

```

Input:
### Steps:
1. **List all two-digit primes using digits 1-9 without repetition.**
  - Possible primes: 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97.
2. **Select four primes that use each digit (1, 2, 3, 4, 5, 6, 7, 9) exactly once.**
  - One valid combination: 13, 29, 47, 61 (digits used: 1,2,3,4,6,7,9).
  - Another valid combination: 13, 29, 67, 41 (digits used: 1,2,3,4,6,7,9).
  - Another valid combination: 17, 23, 59, 41 (digits used: 1,2,3,4,5,7,9).
  - Another valid combination: 17, 23, 59, 61 (digits used: 1,2,3,5,6,7,9).
  - Another valid combination: 17, 29, 43, 61 (digits used: 1,2,3,4,6,7,9).
3. **Calculate the sum for one valid combination.**
  - For 13, 29, 47, 61:
    Sum = 13 + 29 + 47 + 61 = 150.
### Final Answer:
\boxed{{150}}

```

```

Output:
import Mathlib.Data.List.Basic
open Nat List
def digits (n : Nat) : List Nat :=
  [n / 10, n % 10]
def digitsDistinct (n : Nat) : Bool :=
  let ds := digits n
  ds.length = ds.eraseDup.length
def listElemBool (a : Nat) (l : List Nat) : Bool :=
  l.any (fun x => x = a)
def digitsInSet (allowed : List Nat) (n : Nat) : Bool :=
  let ds := digits n
  ds.all (fun d => listElemBool d allowed)
def isPrimeBool (n : Nat) : Bool :=
  if n < 2 then false else
    !((List.range (n - 2)).any (fun d => n % (d + 2) = 0))
def allDigits (nums : List Nat) : List Nat :=

```

```

    nums.flatMap digits
def allDigitsDistinct (nums : List Nat) : Bool :=
  let ds := allDigits nums
  ds.length = ds.eraseDup.length
def allowedDigits : List Nat := [1, 2, 3, 4, 5, 6, 7, 9]
def inputList : List Nat := [13, 29, 47, 61]
def inputSum : Nat := 150
def allPrime : Bool := inputList.all isPrimeBool
def noDigit8 : Bool := inputList.all (digitsInSet allowedDigits)
def allDistinctDigits : Bool := allDigitsDistinct inputList
def sumOfList : Nat := inputList.foldl (· + ·) 0
def sumCorrect : Bool := inputList.foldl (· + ·) 0 = inputSum
#eval ("All prime?", allPrime)
#eval ("No digit 8?", noDigit8)
#eval ("All digits distinct?", allDistinctDigits)
#eval ("Sum correct?", sumCorrect)

```

Example 2:

Input:

```

### Steps:
1. **List all two-digit primes using digits 1-9 without repetition:**
   - 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97.
2. **Select four primes using each digit (1, 2, 3, 4, 5, 6, 7, 9) exactly once:**
   - Possible set: 61, 43, 47, 29 (uses all digits without repetition).
3. **Calculate the sum:**
   -  $(61 + 43 + 47 + 29 = 180)$ .
### Final Answer:
\boxed{{180}}

```

Output:

```

import Mathlib.Data.List.Basic
open Nat List
def digits (n : Nat) : List Nat :=
  [n / 10, n % 10]
def digitsDistinct (n : Nat) : Bool :=
  let ds := digits n
  ds.length = ds.eraseDup.length
def listElemBool (a : Nat) (l : List Nat) : Bool :=
  l.any (fun x => x = a)
def digitsInSet (allowed : List Nat) (n : Nat) : Bool :=
  let ds := digits n
  ds.all (fun d => listElemBool d allowed)
def isPrimeBool (n : Nat) : Bool :=
  if n < 2 then false else
  !((List.range (n - 2)).any (fun d => n % (d + 2) = 0))
def allDigits (nums : List Nat) : List Nat :=
  nums.flatMap digits
def allDigitsDistinct (nums : List Nat) : Bool :=
  let ds := allDigits nums
  ds.length = ds.eraseDup.length
def allowedDigits : List Nat := [1, 2, 3, 4, 5, 6, 7, 9]
def inputList : List Nat := [61, 43, 47, 29]
def inputSum : Nat := 180
def allPrime : Bool := inputList.all isPrimeBool
def noDigit8 : Bool := inputList.all (digitsInSet allowedDigits)
def allDistinctDigits : Bool := allDigitsDistinct inputList
def sumOfList : Nat := inputList.foldl (· + ·) 0
def sumCorrect : Bool := inputList.foldl (· + ·) 0 = inputSum
#eval ("All prime?", allPrime)
#eval ("No digit 8?", noDigit8)
#eval ("All digits distinct?", allDistinctDigits)
#eval ("Sum correct?", sumCorrect)

```

Now do this for the following input:

```

{initial_solution}
"""

```

LEAN_CODE_INTERPRETATION_PROMPT = """

Given the given Lean code input, and its output, interpret the results and provide a concise explanation of the error. Do not include any additional information or context, just the interpretation of the Lean output.

Use the following examples as a guide for your response:

Example 1:

Input:

Lean code:

```
import Mathlib.Data.List.Basic
open Nat List
def digits (n : Nat) : List Nat :=
  [n / 10, n % 10]
def digitsDistinct (n : Nat) : Bool :=
  let ds := digits n
  ds.length = ds.eraseDup.length
def listElemBool (a : Nat) (l : List Nat) : Bool :=
  l.any (fun x => x = a)
def digitsInSet (allowed : List Nat) (n : Nat) : Bool :=
  let ds := digits n
  ds.all (fun d => listElemBool d allowed)
def isPrimeBool (n : Nat) : Bool :=
  if n < 2 then false else
  !((List.range (n - 2)).any (fun d => n % (d + 2) = 0))
def allDigits (nums : List Nat) : List Nat :=
  nums.flatMap digits
def allDigitsDistinct (nums : List Nat) : Bool :=
  let ds := allDigits nums
  ds.length = ds.eraseDup.length
def allowedDigits : List Nat := [1, 2, 3, 4, 5, 6, 7, 9]
def inputList : List Nat := [13, 29, 47, 61]
def inputSum : Nat := 150
def allPrime : Bool := inputList.all isPrimeBool
def noDigit8 : Bool := inputList.all (digitsInSet allowedDigits)
def allDistinctDigits : Bool := allDigitsDistinct inputList
def sumOfList : Nat := inputList.foldl (· + ·) 0
def sumCorrect : Bool := inputList.foldl (· + ·) 0 = inputSum
#eval ("All prime?", allPrime)
#eval ("No digit 8?", noDigit8)
#eval ("All digits distinct?", allDistinctDigits)
#eval ("Sum correct?", sumCorrect)
```

Lean output:

```
info: MathlibProject.lean:30:0: ("All prime?", true)
info: MathlibProject.lean:31:0: ("No digit 8?", true)
info: MathlibProject.lean:32:0: ("All digits distinct?", false)
info: MathlibProject.lean:33:0: ("Sum correct?", true)
```

Output:

The digits in the numbers are not all distinct.

Example 2:

Input:

Lean code:

```
import Mathlib.Data.List.Basic
open Nat List
def digits (n : Nat) : List Nat :=
  [n / 10, n % 10]
def digitsDistinct (n : Nat) : Bool :=
  let ds := digits n
  ds.length = ds.eraseDup.length
def listElemBool (a : Nat) (l : List Nat) : Bool :=
  l.any (fun x => x = a)
def digitsInSet (allowed : List Nat) (n : Nat) : Bool :=
  let ds := digits n
  ds.all (fun d => listElemBool d allowed)
def isPrimeBool (n : Nat) : Bool :=
  if n < 2 then false else
  !((List.range (n - 2)).any (fun d => n % (d + 2) = 0))
```

```

def allDigits (nums : List Nat) : List Nat :=
  nums.flatMap digits
def allDigitsDistinct (nums : List Nat) : Bool :=
  let ds := allDigits nums
  ds.length = ds.eraseDup.length
def allowedDigits : List Nat := [1, 2, 3, 4, 5, 6, 7, 9]
def inputList : List Nat := [13, 29, 47, 65]
def inputSum : Nat := 154
def allPrime : Bool := inputList.all isPrimeBool
def noDigit8 : Bool := inputList.all (digitsInSet allowedDigits)
def allDistinctDigits : Bool := allDigitsDistinct inputList
def sumOfList : Nat := inputList.foldl (· + ·) 0
def sumCorrect : Bool := inputList.foldl (· + ·) 0 = inputSum
#eval ("All prime?", allPrime)
#eval ("No digit 8?", noDigit8)
#eval ("All digits distinct?", allDistinctDigits)
#eval ("Sum correct?", sumCorrect)

```

Lean output:

```

info: MathlibProject.lean:30:0: ("All prime?", false)
info: MathlibProject.lean:31:0: ("No digit 8?", true)
info: MathlibProject.lean:32:0: ("All digits distinct?", false)
info: MathlibProject.lean:33:0: ("Sum correct?", true)

```

Output:

Not all numbers in the list are prime.

Now do this for the following input:

```

Lean code:
{lean_code}

```

```

Lean output:
{lean_output}
"""

```

C.3 Problem 3

```

problem = (
  "If  $\sqrt{5}$  is a root of the equation  $x^2 - 3x + b = 0$ , compute  $b$ ."
)

INITIAL_PROMPT = (
  "Given the following math word problem, provide a concise and brief solution:  

  include only the minimal necessary explanation and the final answer.\n"
  "Return the answer in the format: \boxed{{answer}}.\n"
  "Problem: {problem}"
)

BINARY_CORRECT_PROMPT = (
  "Given the following math problem:\n"
  "{problem}\n\n"
  "You previously solved it like this:\n"
  "{previous_answer}\n\n"
  "This answer was incorrect. Retry the problem from scratch.\n"
  "Provide a concise and brief solution: include only the minimal necessary explanation  

  and the final answer.\n"
  "Return the answer in the format: \boxed{{answer}}.\n"
)

RETRY_SOLVE_WITHOUT_PREVIOUS_ANSWER_PROMPT = (
  "Given the following math problem:\n"
  "{problem}\n\n"
  "Your previous answer had the following error:\n"
  "{error}\n\n"
  "Provide a concise and brief solution: include only the minimal necessary explanation  

  and the final answer.\n"
  "Return the answer in the format: \boxed{{answer}}.\n"
)

RETRY_SOLVE_ERROR_PROMPT = (

```



```

"Given the following math problem:\n"
"{problem}\n\n"
"You previously solved it like this:\n"
"{previous_answer}\n\n"
"But it had the following error:\n"
"{error}\n\n"
"Retry from scratch\n"
"Provide a concise and brief solution: include only the minimal necessary explanation
and the final answer.\n"
"Return the answer in the format: \boxed{{answer}}.\n"
)

```

LEAN_CODE_GENERATION_PROMPT = ""

You are given a natural language solution to a math problem.
 Extract the valid values of 'b' (final boxed answer), then output Lean code using the template exactly.
 Replace the value of 'b' with floating point number.

Template:

```

def givenRoot : Float := 3.0 + 2.2360679775 -- 3 + 5
def input_b : Float := <INPUT_B_VALUE_IN_FLOAT> -- Float value of b

def evalPoly (x b : Float) : Float := x*x - 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps

def isRoot : Bool := approxZero (evalPoly givenRoot input_b)

#eval ("Is 3+sqrt5 a root of polynomial?", isRoot)

```

Use following problems as examples to generate the Lean code with given output format.

Example 1:

Input:

```

**Required steps/formula:**
- For a quadratic equation  $(x^2 + px + q = 0)$ , if  $(r)$  is a root, then the conjugate  $(r')$ 
(if irrational) is also a root.
- Sum of roots:  $(r + r' = -p)$ 
- Product of roots:  $(r \cdot r' = q)$ 
- Here,  $(p = -3)$ , so sum of roots =  $(3)$ .
- Given root:  $(3 + \sqrt{5})$ , so conjugate root:  $(3 - \sqrt{5})$ .
- Compute  $(b = (3 + \sqrt{5})(3 - \sqrt{5}))$ .

**Explanation:**
Since the coefficients are rational and  $(3 + \sqrt{5})$  is irrational, its conjugate
 $(3 - \sqrt{5})$  must also be a root. The product of the roots gives  $(b)$ .

**Calculation:**
\[
(3 + \sqrt{5})(3 - \sqrt{5}) = 3^2 - (\sqrt{5})^2 = 9 - 5 = 4.
\]

**Final answer:**
\[
\boxed{4}
\]

```

Output:

```

def givenRoot : Float := 3.0 + 2.2360679775 -- 3 + 5
def input_b : Float := 4.0 -- Float value of b

def evalPoly (x b : Float) : Float := x*x - 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps

def isRoot : Bool := approxZero (evalPoly givenRoot input_b)

#eval ("Is 3+sqrt5 a root of polynomial?", isRoot)

```

Example 2:

Input:

****Step 1:**** Since $\sqrt{3 + \sqrt{5}}$ is a root and the coefficients are real, the conjugate $\sqrt{3 - \sqrt{5}}$ must also be a root.

****Step 2:**** For a quadratic equation $x^2 - 3x + b = 0$, the sum of the roots is $\sqrt{3}$ and the product is b .

****Step 3:**** Sum of roots:

```
\[
(3 + \sqrt{5}) + (3 - \sqrt{5}) = 6.
\]
```

But the coefficient of x is -3 , so the sum should be $\sqrt{3}$. There is a discrepancy.

****Step 4:**** Re-examine the equation: it is $x^2 - 3x + b = 0$. The sum of the roots is indeed $\sqrt{3}$. However, the sum we computed is 6 , which is not $\sqrt{3}$. This indicates that the conjugate root is not necessarily present because the coefficients are integers? Actually, b is to be computed, and it might not be integer. But the problem states "compute b ", and the equation has integer coefficients for x^2 and x .

Wait: The equation is $x^2 - 3x + b = 0$. The coefficient of x^2 is 1, and of x is -3 , both integers. Therefore, if $\sqrt{3 + \sqrt{5}}$ is a root, then its conjugate $\sqrt{3 - \sqrt{5}}$ must also be a root, because the irrational part $\sqrt{5}$ must cancel out.

But then the sum would be 6 , but it should be $\sqrt{3}$. This is a contradiction.

****Step 5:**** Perhaps the conjugate is not required. Alternatively, maybe the equation has only one root given, and we can substitute to find b .

Substitute $x = 3 + \sqrt{5}$ into the equation:

```
\[
(3 + \sqrt{5})^2 - 3(3 + \sqrt{5}) + b = 0.
\]
```

Compute:

```
\[
(9 + 6\sqrt{5} + 5) - 9 - 3\sqrt{5} + b = 0,
\]
\[[
14 + 6\sqrt{5} - 9 - 3\sqrt{5} + b = 0,
\]
\[[
5 + 3\sqrt{5} + b = 0.
\]
```

So,

```
\[
b = -5 - 3\sqrt{5}.
\]
```

This is a valid value for b , and it is not an integer. The conjugate root is not necessarily present because b is not constrained to be an integer? The problem does not state that b is integer, only to compute it.

Therefore, we can proceed with this.

****Final Answer:****

```
\[
\boxed{-5 - 3\sqrt{5}}
\]
```

Output:

```
def givenRoot : Float := 3.0 + 2.2360679775
def input_b : Float := -5.0 - 3*2.2360679775 -- Float value of b= -5 - 35

def evalPoly (x b : Float) : Float := x*x - 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps

def isRoot : Bool := approxZero (evalPoly givenRoot input_b)

#eval ("Is 3+sqrt5 a root of polynomial?", isRoot)
```

Now do this for the following input:

```
{initial_solution}
"""
```

```

LEAN_CODE_INTERPRETATION_PROMPT = """
Given the given Lean code input, and its output, interpret the results and provide a
concise explanation of the error.
Do not include any additional information or context, just the interpretation of the
Lean output.

Use the following example as a guide for your response:

Input:

Lean code:
def candidateRoot : Float := 3.0 + 2.2360679775 --- 3 + 5
def input_b : Float := 4.0 --- Float value of b=4

def evalPoly (x b : Float) : Float := x*x - 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps

#eval ("Is 3+sqrt5 a root of polynomial?", isRoot)

Lean output:

#eval result: ("Is 3+sqrt5 a root of polynomial?", false)

Output:
3+sqrt{{5}} is not a root of the polynomial  $x^2 - 3x + b = 0$  for the given value of b.

Now this for the following input:

Lean code:
{lean_code}

Lean output:
{lean_output}
"""

```

Appendix D

Code

D.1 Helper Functions

```
def call_deepseek(user_prompt: str, system_prompt: str) -> str:
    response = client.chat.completions.create(
        model="deepseek-chat",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": user_prompt}
        ],
        temperature=0.0,
        max_tokens=2048,
        stream=False
    )
    return response.choices[0].message.content.strip()

def write_lean_code(code: str):
    with open(LEAN_FILE_PATH, "w", encoding="utf-8") as f:
        f.write(code)

def run_lean_code() -> tuple[bool, str]:
    result = subprocess.run(
        ["lake", "build"],
        cwd=LEAN_PROJECT_DIR,
        capture_output=True,
        text=True
    )
    stdout, stderr = result.stdout, result.stderr
    success = "error" not in stderr.lower()
    output = f"=== STDOUT ===\n{stdout}\n\n=== STDERR ===\n{stderr}"

    with open(LEAN_FILE_PATH, "w", encoding="utf-8") as f:
        f.write("")

    return success, output
```

D.2 Asking the LLM to Solve MATH Dataset Problems

```

import os
import json
import re
from openai import OpenAI

# Config
MATH_DATASET_DIR = r"D:\MSc Research Project\MATH\test\counting_and_probability"
MAX_PROBLEMS = None

# DeepSeek API setup
client = OpenAI(
    api_key="sk-cfc0f495b62a4cb38b004830047247db",
    base_url="https://api.deepseek.com"
)

# Prompt
SYSTEM_PROMPT = (
    "Given a math word problem, return only the required steps/formula, only required explanation, and the final answer. "
    "Only main explanation, reduce the number of steps. No extra text."
)

def ask_deepseek(problem_text):
    response = client.chat.completions.create(
        model="deepseek-chat",
        messages=[
            {"role": "system", "content": SYSTEM_PROMPT},
            {"role": "user", "content": problem_text}
        ],
        temperature=0.0,
        max_tokens=2048,
        stream=False
    )
    return response.choices[0].message.content.strip()

def load_math_problems(folder, max_count=None):
    problems = []
    files = [f for f in os.listdir(folder) if f.endswith('.json')]
    for filename in files[:max_count]:
        with open(os.path.join(folder, filename), 'r', encoding='utf-8') as f:
            data = json.load(f)
            problems.append({
                "filename": filename,
                "problem": data["problem"],
                "solution": data["solution"]
            })
    return problems

def extract_boxed_answer(solution_text):
    match = re.search(r'\boxed{([^\}]*\})}', solution_text)
    return match.group(1).strip() if match else None

def main():
    problems = load_math_problems(MATH_DATASET_DIR, max_count=MAX_PROBLEMS)

    correct = 0
    incorrect = 0

    for idx, item in enumerate(problems, start=1):
        print(f"\n=== Problem {idx}: {item['filename']} ===")
        print(item["problem"])

        try:
            # Send the problem
            response = ask_deepseek(item["problem"])
            print(response)

            # Compare with boxed answer
            expected = extract_boxed_answer(item["solution"])
            if expected:
                if expected in response:
                    correct += 1
                    print(f"Matched answer: {expected}")
                else:
                    incorrect += 1
                    print(f"Expected answer: {expected}")
            else:
                print("No answer found in ground truth.")
        except Exception as e:
            print(f"Error: {e}")
            incorrect += 1

    print(f"\nProgress: Solved = {correct + incorrect}, Correct = {correct}, Incorrect = {incorrect}")

    print("\n=== Final Report ===")
    print(f"Total Solved: {correct + incorrect}")
    print(f"Correct: {correct}")
    print(f"Incorrect: {incorrect}")

if __name__ == "__main__":
    main()

```

D.3 Problem 1

No Feedback

```
import subprocess
import time
import os
import re
from openai import OpenAI

# Lean Config
LEAN_FILE_PATH = r"D:\MSc Research Project\MathlibProject\MathlibProject.lean"
LEAN_PROJECT_DIR = r"D:\MSc Research Project\MathlibProject"
MAX_ATTEMPTS = 100

# DeepSeek API Setup
client = OpenAI(
    api_key="sk-cfc0f495b62a4cb38b004830047247db",
    base_url="https://api.deepseek.com"
)

problem = (
    "Suppose the roots of the polynomial  $x^2 - mx + n$  are positive prime integers (not necessarily distinct). "
    "Given that  $m < 20$ , how many possible values of  $n$  are there?"
)

# ----- Prompts -----
INITIAL_PROMPT = (
    "Given the following math word problem, return only the required steps/formula, only required explanation, and the final answer. "
    "Only main explanation, reduce the number of steps. No extra text. \n\n"
    "Problem: {problem}"
)

RETRY_SOLVE_PROMPT_TEMPLATE = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "You previously solved it like this:\n"
    "{previous_answer}\n\n"
    "There is an error in your reasoning.\n"
    "Retry solving the problem again carefully. "
    "Return only the required steps/formula, only required explanation, and the final answer. "
    "Only main explanation, reduce the number of steps. No extra text. \n\n"
)

LEAN_CODE_GENERATION_PROMPT = """
You are given a natural language solution to a math problem.
Extract:
1. The list of valid prime pairs (p, q)
2. The set of distinct n values
3. The count of distinct n values

Then output Lean code using the template exactly. Replace placeholders with extracted values.

Template:
import Mathlib.Data.Nat.Prime.Basic
import Mathlib.Data.List.Basic
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Card
import Init.Data.Nat.Basic

open Nat
open Finset
open List

def validPrimePairs : List ( ) :=
  let primes := (List.range 20).filter Nat.Prime
  List.foldl ( ++ ) [] (
    primes.map (fun p =>
      (List.range 20).filter Nat.Prime |>.filterMap (fun q =>
        if p * q + q < 20 then some (p, q) else none))
  )

def inputPairsList : List ( ) :=
  <PRIME_PAIRS_LIST>

theorem verify_eqv : validPrimePairs.isEqv inputPairsList ( == ) = true := by
  decide

def nValues : List := validPrimePairs.map (fun (p, q) => p * q)

def distinctNValues : Finset := nValues.toFinset

def inputNValues : Finset :=
  <N_VALUES_SET>

theorem verify_nValues_correct : distinctNValues = inputNValues := by
  decide

def inputCount : Nat := <COUNT>

theorem verify_cardinality_correct : distinctNValues.card = inputCount := by
  rfl

```

Use following problems as examples to generate the Lean code with given output format.

Example 1

Input:

****Steps/Formula:****

1. Let the roots be $\lfloor p \rfloor$ and $\lfloor q \rfloor$, where $\lfloor p \rfloor$ and $\lfloor q \rfloor$ are positive prime integers.
2. The polynomial is $\lfloor x^2 - (p+q)x + pq \rfloor$, so $\lfloor m = p + q \rfloor$ and $\lfloor n = pq \rfloor$.
3. Given $\lfloor m < 20 \rfloor$, find all pairs $\lfloor (p, q) \rfloor$ such that $\lfloor p + q < 20 \rfloor$.

****Explanation:****

- List all prime pairs $\lfloor (p, q) \rfloor$ where $\lfloor p \rfloor \leq q \rfloor$ and $\lfloor p + q < 20 \rfloor$.
- Calculate $\lfloor n = pq \rfloor$ for each pair and count the distinct values.

****Prime Pairs and $\lfloor n \rfloor$:**

- (2, 2): $\lfloor n = 4 \rfloor$
- (2, 3): $\lfloor n = 6 \rfloor$
- (2, 5): $\lfloor n = 10 \rfloor$
- (2, 7): $\lfloor n = 14 \rfloor$
- (2, 11): $\lfloor n = 22 \rfloor$ (invalid, $\lfloor m = 13 < 20 \rfloor$)
- (2, 13): $\lfloor n = 26 \rfloor$ (invalid, $\lfloor m = 15 < 20 \rfloor$)
- (2, 17): $\lfloor n = 34 \rfloor$ (invalid, $\lfloor m = 19 < 20 \rfloor$)
- (3, 3): $\lfloor n = 9 \rfloor$
- (3, 5): $\lfloor n = 15 \rfloor$
- (3, 7): $\lfloor n = 21 \rfloor$
- (3, 11): $\lfloor n = 33 \rfloor$ (invalid, $\lfloor m = 14 < 20 \rfloor$)
- (3, 13): $\lfloor n = 39 \rfloor$ (invalid, $\lfloor m = 16 < 20 \rfloor$)
- (5, 5): $\lfloor n = 25 \rfloor$
- (5, 7): $\lfloor n = 35 \rfloor$
- (5, 11): $\lfloor n = 55 \rfloor$ (invalid, $\lfloor m = 16 < 20 \rfloor$)
- (7, 7): $\lfloor n = 49 \rfloor$
- (7, 11): $\lfloor n = 77 \rfloor$ (invalid, $\lfloor m = 18 < 20 \rfloor$)
- (11, 11): $\lfloor n = 121 \rfloor$ (invalid, $\lfloor m = 22 \geq 20 \rfloor$)

****Valid $\lfloor n \rfloor$ values:****

4, 6, 9, 10, 14, 15, 21, 25, 35, 49

****Final Answer:****

$\lfloor \boxed{\{10\}} \rfloor$

Output:

```
import Mathlib.Data.Nat.Prime.Basic
import Mathlib.Data.List.Basic
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Card
import Init.Data.Nat.Basic
```

```
open Nat
open Finset
open List
```

```
def validPrimePairs : List ( ) :=
  let primes := (List.range 20).filter Nat.Prime
  List.foldr ( ++ ) [] (
    primes.map (fun p =>
      (List.range 20).filter Nat.Prime |>.filterMap (fun q =>
        if p * q + q < 20 then some (p, q) else none))
  )
```

```
def inputPairsList : List ( ) :=
```

```
[
  (2, 2),
  (2, 3),
  (2, 5),
  (2, 7),
  (3, 3),
  (3, 5),
  (3, 7),
  (5, 5),
  (5, 7),
  (7, 7)
]
```

```
theorem verify_eqv : validPrimePairs.isEqv inputPairsList ( == ) = true := by
  decide
```

```
def nValues : List := validPrimePairs.map (fun (p, q) => p * q)
```

```
def distinctNValues : Finset := nValues.toFinset
```

```
def inputNValues : Finset :=
  ({4, 6, 9, 10, 14, 15, 21, 25, 35, 49} : Finset)
```

```
theorem verify_nValues_correct : distinctNValues = inputNValues := by
  decide
```

```
def inputCount : Nat := 10
```

```
theorem verify_cardinality_correct : distinctNValues.card = inputCount := by
  rfl
```

Example 2:

Input:

****Steps/Formula:****

1. Let the roots be $\lfloor p \rfloor$ and $\lfloor q \rfloor$, where $\lfloor p \rfloor$ and $\lfloor q \rfloor$ are positive prime integers.
2. The polynomial is $\lfloor x^2 - (p+q)x + pq \rfloor$, so $\lfloor m = p + q \rfloor$ and $\lfloor n = pq \rfloor$.
3. Given $\lfloor m < 20 \rfloor$, find all pairs $\lfloor (p, q) \rfloor$ such that $\lfloor p + q < 20 \rfloor$.

****Explanation:****

- List all prime pairs $\lfloor (p, q) \rfloor$ where $\lfloor p \rfloor \leq q \rfloor$ and $\lfloor p + q < 20 \rfloor$.
- Calculate $\lfloor n = pq \rfloor$ for each pair and count the distinct values.

```

**Prime Pairs and  $\setminus (n \setminus)$ :**
- (2, 2):  $\setminus (n = 4 \setminus)$ 
- (2, 3):  $\setminus (n = 6 \setminus)$ 
- (2, 5):  $\setminus (n = 10 \setminus)$ 
- (2, 7):  $\setminus (n = 14 \setminus)$ 
- (2, 11):  $\setminus (n = 22 \setminus)$ 
- (2, 13):  $\setminus (n = 26 \setminus)$ 
- (2, 17):  $\setminus (n = 34 \setminus)$ 
- (3, 3):  $\setminus (n = 9 \setminus)$ 
- (3, 5):  $\setminus (n = 15 \setminus)$ 
- (3, 7):  $\setminus (n = 21 \setminus)$ 
- (3, 11):  $\setminus (n = 33 \setminus)$ 
- (3, 13):  $\setminus (n = 39 \setminus)$ 
- (5, 5):  $\setminus (n = 25 \setminus)$ 
- (5, 7):  $\setminus (n = 35 \setminus)$ 
- (5, 11):  $\setminus (n = 55 \setminus)$ 
- (7, 7):  $\setminus (n = 49 \setminus)$ 
- (7, 11):  $\setminus (n = 77 \setminus)$ 

**Valid  $\setminus (n \setminus)$  values:**
4, 6, 9, 10, 14, 15, 21, 22, 25, 26, 33, 34, 35, 39, 49, 55, 77

**Final Answer:**
 $\setminus (\boxed{\{17\}} \setminus)$ 

Output:
import Mathlib.Data.Nat.Prime.Basic
import Mathlib.Data.List.Basic
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Card
import Init.Data.Nat.Basic

open Nat
open Finset
open List

def validPrimePairs : List ( ) :=
  let primes := (List.range 20).filter Nat.Prime
  List.foldr ( ++ ) [] (
    primes.map (fun p =>
      (List.range 20).filter Nat.Prime |>.filterMap (fun q =>
        if p * q + q < 20 then some (p, q) else none)
    )
  )

def inputPairsList : List ( ) :=
  [
    (2, 2),
    (2, 3),
    (2, 5),
    (2, 7),
    (2, 11),
    (2, 13),
    (2, 17),
    (3, 3),
    (3, 5),
    (3, 7),
    (3, 11),
    (3, 13),
    (5, 5),
    (5, 7),
    (5, 11),
    (7, 7),
    (7, 11)
  ]

theorem verify_eqv : validPrimePairs.isEqv inputPairsList ( == ) = true := by
  decide

def nValues : List := validPrimePairs.map (fun (p, q) => p * q)

def distinctNValues : Finset := nValues.toFinset

def inputNValues : Finset :=
  ({4, 6, 9, 10, 14, 15, 21, 22, 25, 26, 33, 34, 35, 39, 49, 55, 77} : Finset)

theorem verify_nValues_correct : distinctNValues = inputNValues := by
  decide

def inputCount : Nat := 17

theorem verify_cardinality_correct : distinctNValues.card = inputCount := by
  rfl

Input:
{solution_text}

Output:
"""

# --- Helper functions ---

def call_deepseek(user_prompt: str, system_prompt: str) -> str:
  response = client.chat.completions.create(
    model="deepseek-chat",
    messages=[
      {"role": "system", "content": system_prompt},
      {"role": "user", "content": user_prompt}
    ]
  )

```



```

    ],
    temperature=0.0,
    max_tokens=2048,
    stream=False
)
return response.choices[0].message.content.strip()

def write_lean_code(code: str):
    with open(LEAN_FILE_PATH, "w", encoding="utf-8") as f:
        f.write(code)

def run_lean_code() -> tuple[bool, str]:
    result = subprocess.run(
        ["lake", "build"],
        cwd=LEAN_PROJECT_DIR,
        capture_output=True,
        text=True
    )
    stdout, stderr = result.stdout, result.stderr

    success = "error" not in stderr.lower()

    output = f"=== STDOUT ===\n{stdout}\n\n=== STDERR ===\n{stderr}"

    # Clear Lean file after build
    with open(LEAN_FILE_PATH, "w", encoding="utf-8") as f:
        f.write("")

    return success, output

# --- Main Feedback Loop ---

def feedback_loop(problem_text: str, max_attempts: int = MAX_ATTEMPTS):
    print("\n Asking DeepSeek to solve the problem...\n")
    user_prompt = INITIAL_PROMPT.format(problem=problem_text)
    response = call_deepseek(user_prompt, system_prompt="You are a concise math problem solver.")

    for attempt in range(1, max_attempts + 1):
        print(f"\n Attempt {attempt} -----")
        print(f"\n DeepSeek Response (natural language solution):\n{response}")

        lean_code_prompt = LEAN_CODE_GENERATION_PROMPT.format(solution_text=response)
        lean_code = call_deepseek(lean_code_prompt, system_prompt="You are a Lean code generator")
        print("\nLean code:\n", lean_code)
        write_lean_code(lean_code)

        print("\n Running Lean build...")
        success, lean_output = run_lean_code()
        print(f"\n Lean Output ({'Success' if success else 'Failure'}):\n{lean_output}")

        if success:
            print("\n Reasoning verified by Lean successfully.")
            return response
        else:
            print("\n Lean build failed. Sending error feedback to DeepSeek.")

            retry_prompt = RETRY_SOLVE_PROMPT_TEMPLATE.format(
                problem=problem_text,
                previous_answer=response,
            )

            print(f"\n Retry Prompt:\n{retry_prompt}")

            response = call_deepseek(user_prompt, system_prompt="You are a concise math problem solver.")
            time.sleep(1)

    print("\n Maximum attempts reached. Verification unsuccessful.")
    return None

# === Run Example ===
if __name__ == "__main__":
    feedback_loop(problem)

```

Error Feedback

```

import subprocess
import time
import os
import re
from openai import OpenAI

# Lean Config
LEAN_FILE_PATH = r"D:\MSc Research Project\MathlibProject\MathlibProject.lean"
LEAN_PROJECT_DIR = r"D:\MSc Research Project\MathlibProject"
MAX_ATTEMPTS = 10

# DeepSeek API Setup
client = OpenAI(
    api_key="sk-cfc0f495b62a4cb38b004830047247db",
    base_url="https://api.deepseek.com"
)

# Problem
problem = (
    "Suppose the roots of the polynomial  $x^2 - mx + n$  are positive prime integers (not necessarily distinct). "

```

```

)
    "Given that  $m < 20$ , how many possible values of  $n$  are there?"
)

# ----- Prompts -----

INITIAL_PROMPT = (
    "Given the following math word problem, return only the required steps/formula, only required explanation, and the final answer. "
    "Only main explanation, reduce the number of steps. No extra text. \n\n"
    "Problem: {problem}"
)

RETRY_SOLVE_PROMPT_TEMPLATE = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "You previously solved it like this:\n"
    "{previous_answer}\n\n"
    "But when it had the following error:\n"
    "{error}\n\n"
    "Retry solving the problem again carefully. "
    "Return only the required steps/formula, only required explanation, and the final answer. "
    "Only main explanation, reduce the number of steps. No extra text. \n\n"
)

LEAN_CODE_GENERATION_PROMPT = """
You are given a natural language solution to a math problem.
Extract:
1. The list of valid prime pairs  $(p, q)$ 
2. The set of distinct  $n$  values
3. The count of distinct  $n$  values

Then output Lean code using the template exactly. Replace placeholders with extracted values.

Template:
import Mathlib.Data.Nat.Prime.Basic
import Mathlib.Data.List.Basic
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Card
import Init.Data.Nat.Basic

open Nat
open Finset
open List

def validPrimePairs : List ( ) :=
  let primes := (List.range 20).filter Nat.Prime
  List.foldr ( ++ ) [] (
    primes.map (fun p =>
      (List.range 20).filter Nat.Prime |>.filterMap (fun q =>
        if p * q + q < 20 then some (p, q) else none))
  )

def inputPairsList : List ( ) :=
  <PRIME_PAIRS_LIST>

theorem verify_eqv : validPrimePairs.isEqv inputPairsList ( == ) = true := by
  decide

def nValues : List := validPrimePairs.map (fun (p, q) => p * q)

def distinctNValues : Finset := nValues.toFinset

def inputNValues : Finset :=
  <N_VALUES_SET>

theorem verify_nValues_correct : distinctNValues = inputNValues := by
  decide

def inputCount : Nat := <COUNT>

theorem verify_cardinality_correct : distinctNValues.card = inputCount := by
  rfl

Use following problems as examples to generate the Lean code with given output format.

Example 1

Input:
**Steps/Formula:**
1. Let the roots be  $\sqrt{p}$  and  $\sqrt{q}$ , where  $\sqrt{p}$  and  $\sqrt{q}$  are positive prime integers.
2. The polynomial is  $x^2 - (p+q)x + pq$ , so  $\sqrt{m = p + q}$  and  $\sqrt{n = pq}$ .
3. Given  $\sqrt{m < 20}$ , find all pairs  $\sqrt{(p, q)}$  such that  $\sqrt{p + q < 20}$ .

**Explanation:**
- List all prime pairs  $\sqrt{(p, q)}$  where  $\sqrt{p \leq q}$  and  $\sqrt{p + q < 20}$ .
- Calculate  $\sqrt{n = pq}$  for each pair and count the distinct values.

**Prime Pairs and  $\sqrt{n}$ :\sqrt{n = 4}
- (2, 3):  $\sqrt{n = 6}$ 
- (2, 5):  $\sqrt{n = 10}$ 
- (2, 7):  $\sqrt{n = 14}$ 
- (2, 11):  $\sqrt{n = 22}$  (invalid,  $\sqrt{m = 13 < 20}$ )
- (2, 13):  $\sqrt{n = 26}$  (invalid,  $\sqrt{m = 15 < 20}$ )
- (2, 17):  $\sqrt{n = 34}$  (invalid,  $\sqrt{m = 19 < 20}$ )
- (3, 3):  $\sqrt{n = 9}$ 
- (3, 5):  $\sqrt{n = 15}$ 
- (3, 7):  $\sqrt{n = 21}$ 
- (3, 11):  $\sqrt{n = 33}$  (invalid,  $\sqrt{m = 14 < 20}$ )

```

```

- (3, 13): \ ( n = 39 \ ) (invalid, \ ( m = 16 < 20 \ ))
- (5, 5): \ ( n = 25 \ )
- (5, 7): \ ( n = 35 \ )
- (5, 11): \ ( n = 55 \ ) (invalid, \ ( m = 16 < 20 \ ))
- (7, 7): \ ( n = 49 \ )
- (7, 11): \ ( n = 77 \ ) (invalid, \ ( m = 18 < 20 \ ))
- (11, 11): \ ( n = 121 \ ) (invalid, \ ( m = 22 \geq 20 \ ))

**Valid \ ( n \ ) values:**
4, 6, 9, 10, 14, 15, 21, 25, 35, 49

**Final Answer:**
\ (\boxed{\{10\}} \)

Output:
import Mathlib.Data.Nat.Prime.Basic
import Mathlib.Data.List.Basic
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Card
import Init.Data.Nat.Basic

open Nat
open Finset
open List

def validPrimePairs : List ( ) :=
  let primes := (List.range 20).filter Nat.Prime
  List.foldr ( ++ ) [] (
    primes.map (fun p =>
      (List.range 20).filter Nat.Prime |>.filterMap (fun q =>
        if p * p + q < 20 then some (p, q) else none))
    )

def inputPairsList : List ( ) :=
  [
    (2, 2),
    (2, 3),
    (2, 5),
    (2, 7),
    (3, 3),
    (3, 5),
    (3, 7),
    (5, 5),
    (5, 7),
    (7, 7)
  ]

theorem verify_eqv : validPrimePairs.isEqv inputPairsList ( == ) = true := by
  decide

def nValues : List := validPrimePairs.map (fun (p, q) => p * q)

def distinctNValues : Finset := nValues.toFinset

def inputNValues : Finset :=
  ({4, 6, 9, 10, 14, 15, 21, 25, 35, 49} : Finset)

theorem verify_nValues_correct : distinctNValues = inputNValues := by
  decide

def inputCount : Nat := 10

theorem verify_cardinality_correct : distinctNValues.card = inputCount := by
  rfl

Example 2:

Input:

**Steps/Formula:**
1. Let the roots be \ ( p \ ) and \ ( q \ ), where \ ( p \ ) and \ ( q \ ) are positive prime integers.
2. The polynomial is \ ( x^2 - (p+q)x + pq \ ), so \ ( m = p + q \ ) and \ ( n = pq \ ).
3. Given \ ( m < 20 \ ), find all pairs \ ((p, q)) \ such that \ ( p + q < 20 \ ).

**Explanation:**
- List all prime pairs \ ((p, q)) \ where \ ( p \leq q \ ) and \ ( p + q < 20 \ ).
- Calculate \ ( n = pq \ ) for each pair and count the distinct values.

**Prime Pairs and \ ( n \ ):**
- (2, 2): \ ( n = 4 \ )
- (2, 3): \ ( n = 6 \ )
- (2, 5): \ ( n = 10 \ )
- (2, 7): \ ( n = 14 \ )
- (2, 11): \ ( n = 22 \ )
- (2, 13): \ ( n = 26 \ )
- (2, 17): \ ( n = 34 \ )
- (3, 3): \ ( n = 9 \ )
- (3, 5): \ ( n = 15 \ )
- (3, 7): \ ( n = 21 \ )
- (3, 11): \ ( n = 33 \ )
- (3, 13): \ ( n = 39 \ )
- (5, 5): \ ( n = 25 \ )
- (5, 7): \ ( n = 35 \ )
- (5, 11): \ ( n = 55 \ )
- (7, 7): \ ( n = 49 \ )
- (7, 11): \ ( n = 77 \ )

**Valid \ ( n \ ) values:**
4, 6, 9, 10, 14, 15, 21, 22, 25, 26, 33, 34, 35, 39, 49, 55, 77

```

****Final Answer:****
 $\backslash(\backslash\boxed{\{17\}}\backslash)$

Output:

```
import Mathlib.Data.Nat.Prime.Basic
import Mathlib.Data.List.Basic
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Card
import Init.Data.Nat.Basic

open Nat
open Finset
open List

def validPrimePairs : List ( ) :=
  let primes := (List.range 20).filter Nat.Prime
  List.foldr ( ++ ) [] (
    primes.map (fun p =>
      (List.range 20).filter Nat.Prime |>.filterMap (fun q =>
        if p * q + q < 20 then some (p, q) else none))
    )

def inputPairsList : List ( ) :=
  [
    (2, 2),
    (2, 3),
    (2, 5),
    (2, 7),
    (2, 11),
    (2, 13),
    (2, 17),
    (3, 3),
    (3, 5),
    (3, 7),
    (3, 11),
    (3, 13),
    (5, 5),
    (5, 7),
    (5, 11),
    (7, 7),
    (7, 11)
  ]

theorem verify_eqv : validPrimePairs.isEqv inputPairsList ( == ) = true := by
  decide

def nValues : List := validPrimePairs.map (fun (p, q) => p * q)

def distinctNValues : Finset := nValues.toFinset

def inputNValues : Finset :=
  ({4, 6, 9, 10, 14, 15, 21, 22, 25, 26, 33, 34, 35, 39, 49, 55, 77} : Finset)

theorem verify_nValues_correct : distinctNValues = inputNValues := by
  decide

def inputCount : Nat := 17

theorem verify_cardinality_correct : distinctNValues.card = inputCount := by
  rfl

Input:
{solution_text}

Output:
"""

LEAN_OUTPUT_INTERPRETATION_PROMPT = (
  """Given the following Lean code:
  {lean_code}

  and the following error output:
  {lean_output}

  interpret the error and provide a concise natural language explanation of what went wrong. Your error should be one of the following:
  1. The list of prime pairs generated in your answer were incorrect. Retry the problem again, this time going over each pair carefully,
  and making sure all valid prime pairs are included.
  2. The list of prime pairs are correct. However, the list distinct n values are incorrect.
  3. The list of prime pairs and the list distinct n values are correct. However, the count of distinct n values is incorrect.

  Consider the following example as a reference:

  Input:
  Given the following Lean code:
  import Mathlib.Data.Nat.Prime.Basic
  import Mathlib.Data.List.Basic
  import Mathlib.Data.Finset.Basic
  import Mathlib.Data.Finset.Card
  import Init.Data.Nat.Basic

  open Nat
  open Finset
  open List

  def validPrimePairs : List ( ) :=
  let primes := (List.range 20).filter Nat.Prime
  List.foldr ( ++ ) [] (
    primes.map (fun p =>
```

```

(List.range 20).filter Nat.Prime />.filterMap (fun q =>
  if p < q & p + q < 20 then some (p, q) else none))
)

def inputPairsList : List ( ) :=
[
(2, 2),
(2, 3),
(2, 5),
(2, 7),
(2, 11),
(2, 13),
(2, 17),
(3, 3),
(3, 5),
(3, 7),
(3, 11),
(3, 13),
(5, 5),
(5, 7),
(5, 11),
(7, 7),
(7, 11)
]

theorem verify_eqv : validPrimePairs.isEqv inputPairsList ( == ) = true := by
  decide

def nValues : List := validPrimePairs.map (fun (p, q) => p * q)

def distinctNValues : Finset := nValues.toFinset

def inputNValues : Finset :=
  ({4, 6, 9, 10, 14, 15, 21, 22, 25, 26, 33, 34, 35, 39, 49, 55, 77}) : Finset

theorem verify_nValues_correct : distinctNValues = inputNValues := by
  decide

def inputCount : Nat := 17

theorem verify_cardinality_correct : distinctNValues.card = inputCount := by
  rfl

and the following error output:
error: MathlibProject.lean:35:2: tactic 'decide' proved that the proposition
(validPrimePairs.isEqv inputPairsList fun x1 x2 => x1 == x2) = true
is false
error: MathlibProject.lean:45:2: tactic 'decide' proved that the proposition
distinctNValues = inputNValues
is false
error: MathlibProject.lean:50:2: tactic 'rfl' failed, the left-hand side
#distinctNValues
is not definitionally equal to the right-hand side
inputCount
#distinctNValues = inputCount
error: Lean exited with code 1
Some required builds logged failures:
— MathlibProject

interpret the error and provide a concise natural language explanation of what went wrong. Your error should be one of the following:
1. The list of prime pairs generated in your answer were incorrect. Retry the problem again, this time going over each pair carefully,
and making sure all valid prime pairs are included.
2. The set of distinct n values is incorrect.
3. The count of distinct n values is incorrect.

Output:
The list of prime pairs generated in your answer were incorrect. Retry the problem again, this time going over each pair carefully,
and making sure all valid prime pairs are included.
"""
)

# ----- Helper functions -----

def call_deepseek(user_prompt: str, system_prompt: str) -> str:
  response = client.chat.completions.create(
    model="deepseek-chat",
    messages=[
      {"role": "system", "content": system_prompt},
      {"role": "user", "content": user_prompt}
    ],
    temperature=0.0,
    max_tokens=2048,
    stream=False
  )
  return response.choices[0].message.content.strip()

def write_lean_code(code: str):
  with open(LEAN_FILE_PATH, "w", encoding="utf-8") as f:
    f.write(code)

def run_lean_code() -> tuple[bool, str]:
  result = subprocess.run(
    ["lake", "build"],
    cwd=LEAN_PROJECT_DIR,
    capture_output=True,
    text=True
  )
  stdout, stderr = result.stdout, result.stderr

```

```

success = "error" not in stderr.lower()

output = f"=== STDOUT ===\n{stdout}\n\n=== STDERR ===\n{stderr}"

# Clear Lean file after build
with open(LEAN_FILE_PATH, "w", encoding="utf-8") as f:
    f.write("")

return success, output

# ----- Main Feedback Loop -----

def feedback_loop(problem_text: str, max_attempts: int = MAX_ATTEMPTS):
    print(f"\n Asking DeepSeek to solve the problem...\n")
    user_prompt = INITIAL_PROMPT.format(problem=problem_text)
    response = call_deepseek(user_prompt, system_prompt="You are a concise math problem solver.")

    for attempt in range(1, max_attempts + 1):
        print(f"\n Attempt {attempt} -----")
        print(f"\n DeepSeek Response (natural language solution):\n{response}")

        lean_code_prompt = LEAN_CODE_GENERATION_PROMPT.format(solution_text=response)
        lean_code = call_deepseek(lean_code_prompt, system_prompt="You are a Lean code generator")
        print(f"\n Lean code:\n", lean_code)
        write_lean_code(lean_code)

        print(f"\n Running Lean build...")
        success, lean_output = run_lean_code()
        print(f"\n Lean Output ({'Success' if success else 'Failure'}):\n{lean_output}")

        if success:
            print(f"\n Reasoning verified by Lean successfully.")
            return response
        else:
            print(f"\n Lean build failed. Sending error feedback to DeepSeek.")

            # Interpret the error to get a natural language explanation
            error_feedback = call_deepseek(LEAN_OUTPUT_INTERPRETATION_PROMPT, system_prompt="You are a Lean code interpreter.")

            retry_prompt = RETRY_SOLVE_PROMPT_TEMPLATE.format(
                problem=problem_text,
                previous_answer=response,
                error=error_feedback
            )

            print(f"\n Retry Prompt:\n{retry_prompt}")

            response = call_deepseek(retry_prompt, system_prompt="You are a concise math problem solver.")
            time.sleep(1)

    print(f"\n Maximum attempts reached. Verification unsuccessful.")
    return None

# === Run Example ===
if __name__ == "__main__":
    feedback_loop(problem)

```

D.4 Problem 2

```

import subprocess
import time
import random
import re
from openai import OpenAI

# === Configuration ===

LEAN_FILE_PATH = r"D:\MSc Research Project\MathlibProject\MathlibProject.lean"
LEAN_PROJECT_DIR = r"D:\MSc Research Project\MathlibProject"
MAX_ATTEMPTS = 1000

# DeepSeek API Setup
client = OpenAI(
    api_key="sk-cfc0f495b62a4cb38b004830047247db",
    base_url="https://api.deepseek.com"
)

problem = (
    "The digits $1$, $2$, $3$, $4$, $5$, $6$, $7$, and $9$ are used to form four two-digit prime numbers, "
    "with each digit used exactly once. What is the sum of these four primes? "
)

# === Prompts ===

INITIAL_PROMPT = (
    "Given the following math word problem, provide a **concise and brief** solution: include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format: \boxed{{answer}}.\n"
    "Do **not** include detailed steps, just enough explanation to justify the answer."
    "Problem: {problem}"
)

RETRY_SOLVE_ERROR_PROMPT = (
    "Given the following math problem:\n"

```

```

    "{problem}\n\n"
    "You previously solved it like this:\n"
    "{previous_answer}\n\n"
    "But it had the following error:\n"
    "{error}\n\n"
    "Retry from scratch\n"
    "Provide a concise and brief solution: include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format: \boxed{{{answer}}}. \n"
    "Do not include detailed steps, just enough explanation to justify the answer."
)

BINARY_CORRECT_PROMPT = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "You solved it like this:\n"
    "{previous_answer}\n\n"
    "This answer was incorrect. Retry the problem from scratch.\n"
    "Provide a concise and brief solution: include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format: \boxed{{{answer}}}. \n"
    "Do not include detailed steps, just enough explanation to justify the answer."
)

RETRY_SOLVE_WITHOUT_PREVIOUS_ANSWER_PROMPT = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "Your previous answer had the following error:\n"
    "{error}\n\n"
    "Retry the problem from scratch.\n"
    "Provide a concise and brief solution: include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format: \boxed{{{answer}}}. \n"
    "Do not include detailed steps, just enough explanation to justify the answer."
)

LEAN_CODE_GENERATION_PROMPT = """
You are given a natural language solution to a math problem.
Extract the valid list of 4 two digit numbers (used to calculate the sum) from the solution and the sum of the list (final answer), then output Lean code using the
template exactly.
Replace placeholders with extracted values.

Template:
import Mathlib.Data.List.Basic
open Nat List
def digits (n : Nat) : List Nat :=
  [n / 10, n % 10]
def digitsDistinct (n : Nat) : Bool :=
  let ds := digits n
  ds.length == ds.eraseDup.length
def listElemBool (a : Nat) (l : List Nat) : Bool :=
  l.any (fun x => x == a)
def digitsInSet (allowed : List Nat) (n : Nat) : Bool :=
  let ds := digits n
  ds.all (fun d => listElemBool d allowed)
def isPrimeBool (n : Nat) : Bool :=
  if n < 2 then false else
    !((List.range (n - 2)).any (fun d => n % (d + 2) == 0))
def allDigits (nums : List Nat) : List Nat :=
  nums.flatMap digits
def allDigitsDistinct (nums : List Nat) : Bool :=
  let ds := allDigits nums
  ds.length == ds.eraseDup.length
def allowedDigits : List Nat := [1, 2, 3, 4, 5, 6, 7, 9]
def inputList : List Nat := <REPLACE WITH SOLUTION LIST>
def inputSum : Nat := <REPLACE WITH SOLUTION SUM>
def allPrime : Bool := inputList.all isPrimeBool
def noDigit8 : Bool := inputList.all (digitsInSet allowedDigits)
def allDistinctDigits : Bool := allDigitsDistinct inputList
def sumOfList : Nat := inputList.foldl ( + ) 0
def sumCorrect : Bool := inputList.foldl ( + ) 0 == inputSum
#eval ("All prime?", allPrime)
#eval ("No digit 8?", noDigit8)
#eval ("All digits distinct?", allDistinctDigits)
#eval ("Sum correct?", sumCorrect)

Use following problems as examples to generate the Lean code with given output format.

Example 1:

Input:
### Steps:
1. List all two-digit primes using digits 1–9 without repetition.
  — Possible primes: 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97.
2. Select four primes that use each digit (1, 2, 3, 4, 5, 6, 7, 9) exactly once.
  — One valid combination: 13, 29, 47, 61 (digits used: 1,2,3,4,6,7,9).
  — Another valid combination: 13, 29, 67, 41 (digits used: 1,2,3,4,6,7,9).
  — Another valid combination: 17, 23, 59, 41 (digits used: 1,2,3,4,5,7,9).
  — Another valid combination: 17, 23, 59, 61 (digits used: 1,2,3,5,6,7,9).
  — Another valid combination: 17, 29, 43, 61 (digits used: 1,2,3,4,6,7,9).
3. Calculate the sum for one valid combination.
  — For 13, 29, 47, 61:
    Sum = 13 + 29 + 47 + 61 = 150.
### Final Answer:
\boxed{{150}}

Output:
import Mathlib.Data.List.Basic
open Nat List
def digits (n : Nat) : List Nat :=
  [n / 10, n % 10]
def digitsDistinct (n : Nat) : Bool :=

```

```

let ds := digits n
ds.length = ds.eraseDup.length
def listElemBool (a : Nat) (l : List Nat) : Bool :=
  l.any (fun x => x = a)
def digitsInSet (allowed : List Nat) (n : Nat) : Bool :=
  let ds := digits n
  ds.all (fun d => listElemBool d allowed)
def isPrimeBool (n : Nat) : Bool :=
  if n < 2 then false else
  !((List.range (n - 2)).any (fun d => n % (d + 2) = 0))
def allDigits (nums : List Nat) : List Nat :=
  nums.flatMap digits
def allDigitsDistinct (nums : List Nat) : Bool :=
  let ds := allDigits nums
  ds.length = ds.eraseDup.length
def allowedDigits : List Nat := [1, 2, 3, 4, 5, 6, 7, 9]
def inputList : List Nat := [13, 29, 47, 61]
def inputSum : Nat := 150
def allPrime : Bool := inputList.all isPrimeBool
def noDigit8 : Bool := inputList.all (digitsInSet allowedDigits)
def allDistinctDigits : Bool := allDigitsDistinct inputList
def sumOfList : Nat := inputList.foldl ( + ) 0
def sumCorrect : Bool := inputList.foldl ( + ) 0 = inputSum
#eval ("All prime?", allPrime)
#eval ("No digit 8?", noDigit8)
#eval ("All digits distinct?", allDistinctDigits)
#eval ("Sum correct?", sumCorrect)

```

Example 2:

```

Input:
### Steps:
1. **List all two-digit primes using digits 1–9 without repetition:**
   – 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97.
2. **Select four primes using each digit (1, 2, 3, 4, 5, 6, 7, 9) exactly once:**
   – Possible set: 61, 43, 47, 29 (uses all digits without repetition).
3. **Calculate the sum:**
   –  $(61 + 43 + 47 + 29 = 180)$ .
### Final Answer:
\boxed{{180}}

```

Output:

```

import Mathlib.Data.List.Basic
open Nat List
def digits (n : Nat) : List Nat :=
  [n / 10, n % 10]
def digitsDistinct (n : Nat) : Bool :=
  let ds := digits n
  ds.length = ds.eraseDup.length
def listElemBool (a : Nat) (l : List Nat) : Bool :=
  l.any (fun x => x = a)
def digitsInSet (allowed : List Nat) (n : Nat) : Bool :=
  let ds := digits n
  ds.all (fun d => listElemBool d allowed)
def isPrimeBool (n : Nat) : Bool :=
  if n < 2 then false else
  !((List.range (n - 2)).any (fun d => n % (d + 2) = 0))
def allDigits (nums : List Nat) : List Nat :=
  nums.flatMap digits
def allDigitsDistinct (nums : List Nat) : Bool :=
  let ds := allDigits nums
  ds.length = ds.eraseDup.length
def allowedDigits : List Nat := [1, 2, 3, 4, 5, 6, 7, 9]
def inputList : List Nat := [61, 43, 47, 29]
def inputSum : Nat := 180
def allPrime : Bool := inputList.all isPrimeBool
def noDigit8 : Bool := inputList.all (digitsInSet allowedDigits)
def allDistinctDigits : Bool := allDigitsDistinct inputList
def sumOfList : Nat := inputList.foldl ( + ) 0
def sumCorrect : Bool := inputList.foldl ( + ) 0 = inputSum
#eval ("All prime?", allPrime)
#eval ("No digit 8?", noDigit8)
#eval ("All digits distinct?", allDistinctDigits)
#eval ("Sum correct?", sumCorrect)

```

Now do this for the following input:

```
{initial_solution}
"""
```

LEAN_CODE_INTERPRETATION_PROMPT = """

Given the given Lean code input, and its output, interpret the results and provide a concise explanation of the error.
Do not include any additional information or context, just the interpretation of the Lean output.

Use the following examples as a guide for your response:

Example 1:

Input:

Lean code:

```

import Mathlib.Data.List.Basic
open Nat List
def digits (n : Nat) : List Nat :=
  [n / 10, n % 10]
def digitsDistinct (n : Nat) : Bool :=
  let ds := digits n
  ds.length = ds.eraseDup.length
def listElemBool (a : Nat) (l : List Nat) : Bool :=

```



```

l.any (fun x => x = a)
def digitsInSet (allowed : List Nat) (n : Nat) : Bool :=
  let ds := digits n
  ds.all (fun d => listElemBool d allowed)
def isPrimeBool (n : Nat) : Bool :=
  if n < 2 then false else
  !((List.range (n - 2)).any (fun d => n % (d + 2) = 0))
def allDigits (nums : List Nat) : List Nat :=
  nums.flatMap digits
def allDigitsDistinct (nums : List Nat) : Bool :=
  let ds := allDigits nums
  ds.length = ds.eraseDup.length
def allowedDigits : List Nat := [1, 2, 3, 4, 5, 6, 7, 9]
def inputList : List Nat := [13, 29, 47, 61]
def inputSum : Nat := 150
def allPrime : Bool := inputList.all isPrimeBool
def noDigit8 : Bool := inputList.all (digitsInSet allowedDigits)
def allDistinctDigits : Bool := allDigitsDistinct inputList
def sumOfList : Nat := inputList.foldl ( + ) 0
def sumCorrect : Bool := inputList.foldl ( + ) 0 = inputSum
#eval ("All prime?", allPrime)
#eval ("No digit 8?", noDigit8)
#eval ("All digits distinct?", allDistinctDigits)
#eval ("Sum correct?", sumCorrect)

```

Lean output:

```

info: MathlibProject.lean:30:0: ("All prime?", true)
info: MathlibProject.lean:31:0: ("No digit 8?", true)
info: MathlibProject.lean:32:0: ("All digits distinct?", false)
info: MathlibProject.lean:33:0: ("Sum correct?", true)

```

Output:

The digits in the numbers are not all distinct.

Example 2:

Input:

Lean code:

```

import Mathlib.Data.List.Basic
open Nat List
def digits (n : Nat) : List Nat :=
  [n / 10, n % 10]
def digitsDistinct (n : Nat) : Bool :=
  let ds := digits n
  ds.length = ds.eraseDup.length
def listElemBool (a : Nat) (l : List Nat) : Bool :=
  l.any (fun x => x = a)
def digitsInSet (allowed : List Nat) (n : Nat) : Bool :=
  let ds := digits n
  ds.all (fun d => listElemBool d allowed)
def isPrimeBool (n : Nat) : Bool :=
  if n < 2 then false else
  !((List.range (n - 2)).any (fun d => n % (d + 2) = 0))
def allDigits (nums : List Nat) : List Nat :=
  nums.flatMap digits
def allDigitsDistinct (nums : List Nat) : Bool :=
  let ds := allDigits nums
  ds.length = ds.eraseDup.length
def allowedDigits : List Nat := [1, 2, 3, 4, 5, 6, 7, 9]
def inputList : List Nat := [13, 29, 47, 65]
def inputSum : Nat := 154
def allPrime : Bool := inputList.all isPrimeBool
def noDigit8 : Bool := inputList.all (digitsInSet allowedDigits)
def allDistinctDigits : Bool := allDigitsDistinct inputList
def sumOfList : Nat := inputList.foldl ( + ) 0
def sumCorrect : Bool := inputList.foldl ( + ) 0 = inputSum
#eval ("All prime?", allPrime)
#eval ("No digit 8?", noDigit8)
#eval ("All digits distinct?", allDistinctDigits)
#eval ("Sum correct?", sumCorrect)

```

Lean output:

```

info: MathlibProject.lean:30:0: ("All prime?", false)
info: MathlibProject.lean:31:0: ("No digit 8?", true)
info: MathlibProject.lean:32:0: ("All digits distinct?", false)
info: MathlibProject.lean:33:0: ("Sum correct?", true)

```

Output:

Not all numbers in the list are prime.

Now do this for the following input:

Lean code:

```
{lean_code}
```

Lean output:

```
{lean_output}
"""
```

=== Helper functions ===

```

def call_deepseek(user_prompt: str, system_prompt: str) -> str:
  response = client.chat.completions.create(

```

```

        model="deepseek-chat",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": user_prompt}
        ],
        temperature=0.0,
        max_tokens=4000,
        stream=False
    )
    return response.choices[0].message.content.strip()

def write_lean_code(code: str):
    with open(LEAN_FILE_PATH, "w", encoding="utf-8") as f:
        f.write(code)

def run_lean_code() -> tuple[bool, str]:
    result = subprocess.run(
        ["lake", "build"],
        cwd=LEAN_PROJECT_DIR,
        capture_output=True,
        text=True
    )
    stdout, stderr = result.stdout, result.stderr

    # Assume build success if no 'error' in stderr
    success = "error" not in stderr.lower()

    output = f"=== STDOUT ===\n{stdout}\n\n=== STDERR ===\n{stderr}"

    # Clear Lean file after build (optional)
    # with open(LEAN_FILE_PATH, "w", encoding="utf-8") as f:
    #     f.write("")

    return success, output

def clean_lean_code(raw_code: str) -> str:
    """
    Extract Lean code from API response, stripping any markdown fences and leading/trailing text.
    """
    # Remove markdown triple backticks if present
    if raw_code.startswith("```"):
        # Find the first and last triple backtick
        parts = raw_code.split("```")
        # The code is usually the second element after split: ```lang\ncode\n```
        if len(parts) >= 3:
            code = parts[1]
        else:
            code = raw_code
    else:
        code = raw_code

    # Strip leading/trailing whitespace
    code = code.strip()

    return code

# === Main Feedback Loop ===

def feedback_loop(problem_text: str, max_attempts: int = MAX_ATTEMPTS):
    first_try_correct = 0
    no_feedback_correct = 0
    binary_correct = 0
    error_feedback = 0
    error_feedback_without_previous_answer = 0
    incomplete_answer = 0

    for attempt in range(1, max_attempts + 1):
        print(f"\n--- Trial {attempt} ---")

        # Step 1: Initial solve attempt
        user_prompt = INITIAL_PROMPT.format(problem=problem_text)
        print("User Prompt:", user_prompt)
        response = call_deepseek(user_prompt, system_prompt="You are a concise math problem solver.")
        print("\nInitial Response:\n", response)

        match = re.search(r'\boxed\{([^\}]*)\}', response)
        if not match:
            print("No boxed answer found — retrying without incrementing attempt count.")
            incomplete_answer += 1
            attempt -= 1
            continue

        answer = match.group(1).strip() if match else None

        # Check if answer is correct
        if answer == "190":
            print("First try correct — skipping retries.")
            first_try_correct += 1
            continue # or 'return' to exit loop/function immediately

        else:
            print("First try incorrect, proceeding with retries.")

        lean_code = call_deepseek(
            user_prompt=LEAN_CODE_GENERATION_PROMPT.format(initial_solution=response),
            system_prompt="You are a Lean code generator and verifier for math solutions."
        )
        print("\nGenerated Lean code:\n", lean_code)

```

```

# Write Lean code to file
clean_code = clean_lean_code(clean_code)
write_lean_code(clean_code)

# Step 4: Run Lean build and get checks output
success, lean_output = run_lean_code()
print(f"Lean build success: {success}")
print(f"Lean output:\n{lean_output}")

# Step 5: Parse Lean output for error messages
error_description = call_deepseek(
    user_prompt=LEAN_CODE_INTERPRETATION_PROMPT.format(lean_code=clean_code, lean_output=lean_output),
    system_prompt="You are a Lean code interpreter and verifier for math solutions."
)
print("Parsed error description:", error_description)

# Step 6: Prepare retry arms with prompts
arms = [
    ("no_feedback", INITIAL_PROMPT.format(problem=problem_text)),
    ("binary_correctness", BINARY_CORRECT_PROMPT.format(problem=problem_text, previous_answer=response)),
    ("error_feedback_without_previous_answer", RETRY_SOLVE_WITHOUT_PREVIOUS_ANSWER_PROMPT.format(
        problem=problem_text,
        error=error_description
    )),
    ("error_feedback", RETRY_SOLVE_ERROR_PROMPT.format(
        problem=problem_text,
        previous_answer=response,
        error=error_description
    ))
]

random.shuffle(arms)

# Step 7: Run retries with different feedback styles
for arm_type, retry_prompt in arms:
    # Random delay to avoid timing patterns
    time.sleep(random.uniform(0.2, 1.0))
    print("Retry prompt:", retry_prompt)
    retry_response = call_deepseek(retry_prompt, system_prompt="You are a concise math problem solver.")
    print(f"\n[{arm_type.upper()}] Retry Response:\n", retry_response)

    match_retry = re.search(r'\boxed\{([*])\}', retry_response)
    retry_answer = match_retry.group(1).strip() if match_retry else None

    if retry_answer == "190":
        print(f"[{arm_type.upper()}] Retry correct!")
        if arm_type == "no_feedback":
            no_feedback_correct += 1
        elif arm_type == "binary_correctness":
            binary_correct += 1
        elif arm_type == "error_feedback":
            error_feedback += 1
        elif arm_type == "error_feedback_without_previous_answer":
            error_feedback_without_previous_answer += 1
    else:
        print(f"[{arm_type.upper()}] Retry incorrect or no boxed answer.")

# == Final summary ==
print("\n=== RESULTS ===")
print(f"First try correct: {first_try_correct}")
print(f"No feedback correct: {no_feedback_correct}")
print(f"Binary correctness correct: {binary_correct}")
print(f"Error feedback without previous answer correct: {error_feedback_without_previous_answer}")
print(f"Error feedback correct: {error_feedback}")
print(f"Incomplete answers (no boxed answer): {incomplete_answer}")

# == Run if main ==
if __name__ == "__main__":
    feedback_loop(problem)

```

D.5 Problem 3

Test-1

```

import subprocess
import time
import random
import re
from openai import OpenAI
import csv
import os

# == Configuration ==

LEAN_FILE_PATH = r"D:\MSc Research Project\MathlibProject\MathlibProject.lean"
LEAN_PROJECT_DIR = r"D:\MSc Research Project\MathlibProject"
MAX_ATTEMPTS = 1000

# DeepSeek API Setup
client = OpenAI(
    api_key="sk-cfc0f495b62a4cb38b004830047247db",

```

```

    base_url="https://api.deepseek.com"
)

problem = (
    "If  $3 + \sqrt{5}$  is a root of the equation  $\backslash x^2 - 3x + b = 0 \backslash$  compute  $b$ ."
)

# === Prompts ===

INITIAL_PROMPT = (
    "Given the following math word problem, provide a concise and brief solution: include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format:  $\boxed{\text{answer}}$ .\n"
    "Problem: {problem}"
)

RETRY_SOLVE_ERROR_PROMPT = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "You previously solved it like this:\n"
    "{previous_answer}\n\n"
    "But it had the following error:\n"
    "{error}\n\n"
    "Retry from scratch\n"
    "Provide a concise and brief solution: include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format:  $\boxed{\text{answer}}$ .\n"
)

BINARY_CORRECT_PROMPT = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "You previously solved it like this:\n"
    "{previous_answer}\n\n"
    "This answer was incorrect. Retry the problem from scratch.\n"
    "Provide a concise and brief solution: include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format:  $\boxed{\text{answer}}$ .\n"
)

RETRY_SOLVE_WITHOUT_PREVIOUS_ANSWER_PROMPT = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "Your previous answer had the following error:\n"
    "{error}\n\n"
    "Provide a concise and brief solution: include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format:  $\boxed{\text{answer}}$ .\n"
)

LEAN_CODE_GENERATION_PROMPT = """
You are given a natural language solution to a math problem.
Extract the valid values of 'b' (final boxed answer), then output Lean code using the template exactly.
Replace the value of 'b' with floating point number.

Template:

def givenRoot : Float := 3.0 + 2.2360679775 — 3 + 5
def input_b : Float := <INPUT_B_VALUE_IN_FLOAT> — Float value of b

def evalPoly (x b : Float) : Float := x*x — 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps

def isRoot : Bool := approxZero (evalPoly givenRoot input_b)

#eval ("Is  $3 + \sqrt{5}$  a root of polynomial?", isRoot)

Use following problems as examples to generate the Lean code with given output format.

Example 1:

Input:

Required steps/formula:
— For a quadratic equation  $\backslash (x^2 + px + q = 0) \backslash$ , if  $\backslash (r) \backslash$  is a root, then the conjugate  $\backslash (r') \backslash$  (if irrational) is also a root.
— Sum of roots:  $\backslash (r + r' = -p) \backslash$ 
— Product of roots:  $\backslash (r \cdot r' = q) \backslash$ 
— Here,  $\backslash (p = -3) \backslash$ , so sum of roots =  $\backslash (3) \backslash$ .
— Given root:  $\backslash (3 + \sqrt{5}) \backslash$ , so conjugate root:  $\backslash (3 - \sqrt{5}) \backslash$ .
— Compute  $\backslash (b = (3 + \sqrt{5})(3 - \sqrt{5})) \backslash$ .

Explanation:
Since the coefficients are rational and  $\backslash (3 + \sqrt{5}) \backslash$  is irrational, its conjugate  $\backslash (3 - \sqrt{5}) \backslash$  must also be a root. The product of the roots gives  $\backslash (b) \backslash$ .

Calculation:

$$\backslash (3 + \sqrt{5})(3 - \sqrt{5}) = 3^2 - (\sqrt{5})^2 = 9 - 5 = 4.$$


Final answer:

$$\backslash \backslash \boxed{4} \backslash \backslash$$


Output:

def givenRoot : Float := 3.0 + 2.2360679775 — 3 + 5
def input_b : Float := 4.0 — Float value of b

def evalPoly (x b : Float) : Float := x*x — 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps

def isRoot : Bool := approxZero (evalPoly givenRoot input_b)

```

```
#eval ("Is 3+sqrt5 a root of polynomial?", isRoot)
Example 2:
```

Input:

****Step 1:**** Since $\sqrt{3 + \sqrt{5}}$ is a root and the coefficients are real, the conjugate $\sqrt{3 - \sqrt{5}}$ must also be a root.

****Step 2:**** For a quadratic equation $x^2 - 3x + b = 0$, the sum of the roots is $\sqrt{3}$ and the product is \sqrt{b} .

****Step 3:**** Sum of roots:

$$\sqrt{3 + \sqrt{5}} + (3 - \sqrt{5}) = 6.$$

But the coefficient of x is -3 , so the sum should be $\sqrt{3}$. There is a discrepancy.

****Step 4:**** Re-examine the equation: it is $x^2 - 3x + b = 0$. The sum of the roots is indeed $\sqrt{3}$. However, the sum we computed is $\sqrt{6}$, which is not $\sqrt{3}$. This indicates that the conjugate root is not necessarily present because the coefficients are integers? Actually, \sqrt{b} is to be computed, and it might not be integer. But the problem states "compute \sqrt{b} ", and the equation has integer coefficients for x^2 and x .

Wait: The equation is $x^2 - 3x + b = 0$. The coefficient of x^2 is 1, and of x is -3 , both integers. Therefore, if $\sqrt{3 + \sqrt{5}}$ is a root, then its conjugate $\sqrt{3 - \sqrt{5}}$ must also be a root, because the irrational part $\sqrt{\sqrt{5}}$ must cancel out.

But then the sum would be $\sqrt{6}$, but it should be $\sqrt{3}$. This is a contradiction.

****Step 5:**** Perhaps the conjugate is not required. Alternatively, maybe the equation has only one root given, and we can substitute to find \sqrt{b} .

Substitute $x = 3 + \sqrt{5}$ into the equation:

$$(3 + \sqrt{5})^2 - 3(3 + \sqrt{5}) + b = 0.$$

Compute:

$$9 + 6\sqrt{5} + 5 - 9 - 3\sqrt{5} + b = 0,$$

$$14 + 6\sqrt{5} - 9 - 3\sqrt{5} + b = 0,$$

$$5 + 3\sqrt{5} + b = 0.$$

So,

$$b = -5 - 3\sqrt{5}.$$

This is a valid value for \sqrt{b} , and it is not an integer. The conjugate root is not necessarily present because \sqrt{b} is not constrained to be an integer? The problem does not state that \sqrt{b} is integer, only to compute it.

Therefore, we can proceed with this.

****Final Answer:****

$$\boxed{-5 - 3\sqrt{5}}$$

Output:

```
def givenRoot : Float := 3.0 + 2.2360679775
def input_b : Float := -5.0 - 3*2.2360679775 --- Float value of b=-5 - 35
```

```
def evalPoly (x b : Float) : Float := x*x - 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps
```

```
def isRoot : Bool := approxZero (evalPoly givenRoot input_b)
```

```
#eval ("Is 3+sqrt5 a root of polynomial?", isRoot)
```

Now do this for the following input:

```
{initial_solution}
"""
```

LEAN_CODE_INTERPRETATION_PROMPT = """

Given the given Lean code input, and its output, interpret the results and provide a concise explanation of the error. Do not include any additional information or context, just the interpretation of the Lean output.

Use the following example as a guide for your response:

Input:

```
Lean code:
def candidateRoot : Float := 3.0 + 2.2360679775 --- 3 + 5
def input_b : Float := 4.0 --- Float value of b=4

def evalPoly (x b : Float) : Float := x*x - 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps

#eval ("Is 3+sqrt5 a root of polynomial?", isRoot)
```

Lean output:

```
#eval result: ("Is 3+sqrt5 a root of polynomial?", false)
```

Output:

$3 + \sqrt{5}$ is not a root of the polynomial $x^2 - 3x + b = 0$ for the given value of b .

Now this for the following input:

```
Lean code:
{lean_code}
```

```

Lean output:
{lean_output}
"""

# === Helper functions ===

def call_deepseek(user_prompt: str, system_prompt: str) -> str:
    response = client.chat.completions.create(
        model="deepseek-chat",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": user_prompt}
        ],
        temperature=0.0,
        max_tokens=1024,
        stream=False
    )
    return response.choices[0].message.content.strip()

def write_lean_code(code: str):
    with open(LEAN_FILE_PATH, "w", encoding="utf-8") as f:
        f.write(code)

def run_lean_code() -> tuple[bool, str]:
    result = subprocess.run(
        ["lake", "build"],
        cwd=LEAN_PROJECT_DIR,
        capture_output=True,
        text=True
    )
    stdout, stderr = result.stdout, result.stderr

    # Assume build success if no 'error' in stderr
    success = "error" not in stderr.lower()

    output = f"=== STDOUT ===\n{stdout}\n\n=== STDERR ===\n{stderr}"

    # Clear Lean file after build (optional)
    # with open(LEAN_FILE_PATH, "w", encoding="utf-8") as f:
    #     f.write("")

    return success, output

def clean_lean_code(raw_code: str) -> str:
    """
    Extract Lean code from API response, stripping any markdown fences and leading/trailing text.
    """
    if raw_code.startswith("```"):
        parts = raw_code.split("```")
        if len(parts) >= 3:
            code = parts[1]
        else:
            code = raw_code
    else:
        code = raw_code

    code = code.strip()

    return code

# === Main Feedback Loop ===

def feedback_loop(problem_text: str, max_attempts: int = MAX_ATTEMPTS):
    # Success counters
    first_try_correct = 0
    no_feedback_correct = 0
    binary_correct = 0
    error_feedback = 0
    error_feedback_without_previous_answer = 0
    incomplete_answer = 0

    # Create logging folder:
    log_folder = r"D:\MSc Research Project\Problem-3 logs"
    os.makedirs(log_folder, exist_ok=True)

    attempt = 0
    while attempt < max_attempts:
        attempt += 1

        # Dictionaries to store retry info
        retry_responses = {}
        retry_lean_codes = {}
        retry_lean_outputs = {}

        # Step 1: Initial solve attempt
        user_prompt = INITIAL_PROMPT.format(problem=problem_text)
        print(f"Attempt {attempt}: {user_prompt}")
        response = call_deepseek(user_prompt, system_prompt="You are a concise math problem solver.")
        print(f"Response: {response}")

        match = re.search(r'\boxed{\{([^\}]*)\}}', response)
        if not match:
            incomplete_answer += 1
            attempt -= 1 # do not count incomplete answer
            continue

```

```

lean_code = call_deepseek(
    user_prompt=LEAN_CODE_GENERATION_PROMPT.format(initial_solution=response),
    system_prompt="You are a Lean code generator and verifier for math solutions."
)
print(f"Lean Code: {lean_code}")
clean_code = clean_lean_code(lean_code)
write_lean_code(clean_code)
print(f"Clean Lean Code: {clean_code}")

# Step 4: Run Lean build
success, lean_output = run_lean_code()
print(f"Lean Output: {lean_output}")
answer = match.group(1).strip() if match else None

# Check if first try correct
if "false" not in lean_output.lower():
    first_try_correct += 1
    # Log CSV and skip retries
    log_file = os.path.join(log_folder, f"Attempt {attempt}.csv")
    with open(log_file, "w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(["Field", "Content"])
        writer.writerow(["Initial Prompt", user_prompt])
        writer.writerow(["Initial Response", response])
        writer.writerow(["Lean Code", clean_code])
        writer.writerow(["Lean Output", lean_output])
        writer.writerow(["First Try Correct", first_try_correct])
        writer.writerow(["No Feedback Correct", no_feedback_correct])
        writer.writerow(["Binary Correct", binary_correct])
        writer.writerow(["Error Feedback Without Previous Answer", error_feedback_without_previous_answer])
        writer.writerow(["Error Feedback", error_feedback])
        writer.writerow(["Incomplete Answer", incomplete_answer])
    continue

# Step 5: Interpret Lean output
error_description = call_deepseek(
    user_prompt=LEAN_CODE_INTERPRETATION_PROMPT.format(
        lean_code=clean_code,
        lean_output=lean_output
    ),
    system_prompt="You are a Lean code interpreter and verifier for math solutions."
)
print(f"Error Description: {error_description}")

# Step 6: Prepare retry arms
arms = [
    ("no_feedback", INITIAL_PROMPT.format(problem=problem_text)),
    ("binary_correctness", BINARY_CORRECT_PROMPT.format(problem=problem_text, previous_answer=response)),
    ("error_feedback_without_previous_answer", RETRY_SOLVE_WITHOUT_PREVIOUS_ANSWER_PROMPT.format(
        problem=problem_text,
        error=error_description
    )),
    ("error_feedback", RETRY_SOLVE_ERROR_PROMPT.format(
        problem=problem_text,
        previous_answer=response,
        error=error_description
    ))
]

random.shuffle(arms)

# Step 7: Run retries
for arm_type, retry_prompt in arms:
    time.sleep(random.uniform(0.2, 1.0))
    retry_response = call_deepseek(retry_prompt, system_prompt="You are a concise math problem solver.")
    retry_responses[arm_type] = retry_response

    retry_lean_code = call_deepseek(
        user_prompt=LEAN_CODE_GENERATION_PROMPT.format(initial_solution=retry_response),
        system_prompt="You are a Lean code generator and verifier for math solutions."
    )
    clean_retry_code = clean_lean_code(retry_lean_code)
    retry_lean_codes[arm_type] = clean_retry_code
    write_lean_code(clean_retry_code)

    success_retry, retry_lean_output = run_lean_code()
    retry_lean_outputs[arm_type] = retry_lean_output

    if "false" not in retry_lean_output.lower():
        if arm_type == "no_feedback":
            no_feedback_correct += 1
        elif arm_type == "binary_correctness":
            binary_correct += 1
        elif arm_type == "error_feedback":
            error_feedback += 1
        elif arm_type == "error_feedback_without_previous_answer":
            error_feedback_without_previous_answer += 1

# Step 8: Log everything to CSV
log_file = os.path.join(log_folder, f"Attempt {attempt}.csv")
with open(log_file, "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerow(["Field", "Content"])
    writer.writerow(["Initial Prompt", user_prompt])
    writer.writerow(["Initial Response", response])
    writer.writerow(["Lean Code", clean_code])
    writer.writerow(["Lean Output", lean_output])
    writer.writerow(["Error Description", error_description])

```

```

for arm_type in arms:
    arm_name = arm_type[0]
    writer.writerow([f"{arm_name} Retry Prompt", arms[a[0] for a in arms].index(arm_name)][1]])
    writer.writerow([f"{arm_name} Retry Response", retry_responses[arm_name]])
    writer.writerow([f"{arm_name} Retry Lean Code", retry_lean_codes[arm_name]])
    writer.writerow([f"{arm_name} Retry Lean Output", retry_lean_outputs[arm_name]])

writer.writerow(["First Try Correct", first_try_correct])
writer.writerow(["No Feedback Correct", no_feedback_correct])
writer.writerow(["Binary Correct", binary_correct])
writer.writerow(["Error Feedback Without Previous Answer", error_feedback_without_previous_answer])
writer.writerow(["Error Feedback", error_feedback])
writer.writerow(["Incomplete Answer", incomplete_answer])

# === Run if main ===

if __name__ == "__main__":
    feedback_loop(problem)

```

Test-2

```

import subprocess
import time
import random
import re
from openai import OpenAI
import csv
import os

# === Configuration ===

LEAN_FILE_PATH = r"D:\MSc Research Project\MathlibProject\MathlibProject.lean"
LEAN_PROJECT_DIR = r"D:\MSc Research Project\MathlibProject"
LOG_ROOT = r"D:\MSc Research Project\Problem-3 Logs 200 trials with 20 attempts (Correct Prompt)"
MAX_TRIALS = 100
MAX_ATTEMPTS = 20

# DeepSeek API Setup
client = OpenAI(
    api_key="sk-cfc0f495b62a4cb38b004830047247db",
    base_url="https://api.deepseek.com"
)

problem = (
    "If  $\sqrt{5}$  is a root of the equation  $x^2 - 3x + b = 0$ , compute  $b$ ."
)

# === Prompts ===

INITIAL_PROMPT = (
    "Given the following math word problem, provide a **concise and brief** solution: include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format: \boxed{{answer}}.\n"
    "Problem: {problem}"
)

RETRY_SOLVE_ERROR_PROMPT = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "You previously solved it like this:\n"
    "{previous_answer}\n\n"
    "But it had the following error:\n"
    "{error}\n\n"
    "Retry from scratch\n"
    "Provide a **concise and brief** solution: include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format: \boxed{{answer}}.\n"
)

BINARY_CORRECT_PROMPT = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "You previously solved it like this:\n"
    "{previous_answer}\n\n"
    "This answer was incorrect. Retry the problem from scratch.\n"
    "Provide a **concise and brief** solution: include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format: \boxed{{answer}}.\n"
)

RETRY_SOLVE_WITHOUT_PREVIOUS_ANSWER_PROMPT = (
    "Given the following math problem:\n"
    "{problem}\n\n"
    "Your previous answer had the following error:\n"
    "{error}\n\n"
    "Provide a **concise and brief** solution: include only the minimal necessary explanation and the final answer.\n"
    "Return the answer in the format: \boxed{{answer}}.\n"
)

LEAN_CODE_GENERATION_PROMPT = """
You are given a natural language solution to a math problem.
Extract the valid values of 'b' (final boxed answer), then output Lean code using the template exactly.
Replace the value of 'b' with floating point number.

Template:

def givenRoot : Float := 3.0 + 2.2360679775 -- 3 + 5

```



```
def input_b : Float := <INPUT_B_VALUE_IN_FLOAT> --- Float value of b
```

```
def evalPoly (x b : Float) : Float := x*x - 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps
```

```
def isRoot : Bool := approxZero (evalPoly givenRoot input_b)
```

```
#eval ("Is 3+sqrt5 a root of polynomial?", isRoot)
```

Use following problems as examples to generate the Lean code with given output format.

Example 1:

Input:

```
**Required steps/formula:**
- For a quadratic equation  $(x^2 + px + q = 0)$ , if  $(r)$  is a root, then the conjugate  $(r')$  (if irrational) is also a root.
- Sum of roots:  $(r + r' = -p)$ 
- Product of roots:  $(r \cdot r' = q)$ 
- Here,  $(p = -3)$ , so sum of roots =  $(3)$ .
- Given root:  $(3 + \sqrt{5})$ , so conjugate root:  $(3 - \sqrt{5})$ .
- Compute  $(b = (3 + \sqrt{5})(3 - \sqrt{5}))$ .

**Explanation:**
Since the coefficients are rational and  $(3 + \sqrt{5})$  is irrational, its conjugate  $(3 - \sqrt{5})$  must also be a root. The product of the roots gives  $(b)$ .
```

```
**Calculation:**
|
|  $(3 + \sqrt{5})(3 - \sqrt{5}) = 3^2 - (\sqrt{5})^2 = 9 - 5 = 4.$ 
|
```

```
**Final answer:**
```

```
|
|  $\boxed{4}$ 
|
```

Output:

```
def givenRoot : Float := 3.0 + 2.2360679775 --- 3 + 5
def input_b : Float := 4.0 --- Float value of b

def evalPoly (x b : Float) : Float := x*x - 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps

def isRoot : Bool := approxZero (evalPoly givenRoot input_b)

#eval ("Is 3+sqrt5 a root of polynomial?", isRoot)
Example 2:
```

Input:

```
**Step 1:** Since  $(3 + \sqrt{5})$  is a root and the coefficients are real, the conjugate  $(3 - \sqrt{5})$  must also be a root.

**Step 2:** For a quadratic equation  $(x^2 - 3x + b = 0)$ , the sum of the roots is  $(3)$  and the product is  $(b)$ .

**Step 3:** Sum of roots:
|
|  $(3 + \sqrt{5}) + (3 - \sqrt{5}) = 6.$ 
|
But the coefficient of  $(x)$  is  $(-3)$ , so the sum should be  $(3)$ . There is a discrepancy.

**Step 4:** Re-examine the equation: it is  $(x^2 - 3x + b = 0)$ . The sum of the roots is indeed  $(3)$ . However, the sum we computed is  $(6)$ , which is not  $(3)$ . This indicates that the conjugate root is not necessarily present because the coefficients are integers? Actually,  $(b)$  is to be computed, and it might not be integer. But the problem states "compute  $(b)$ ", and the equation has integer coefficients for  $(x^2)$  and  $(x)$ .

Wait: The equation is  $(x^2 - 3x + b = 0)$ . The coefficient of  $(x^2)$  is 1, and of  $(x)$  is  $(-3)$ , both integers. Therefore, if  $(3 + \sqrt{5})$  is a root, then its conjugate  $(3 - \sqrt{5})$  must also be a root, because the irrational part  $(\sqrt{5})$  must cancel out.

But then the sum would be  $(6)$ , but it should be  $(3)$ . This is a contradiction.

**Step 5:** Perhaps the conjugate is not required. Alternatively, maybe the equation has only one root given, and we can substitute to find  $(b)$ .
```

Substitute $(x = 3 + \sqrt{5})$ into the equation:

```
|
|  $(3 + \sqrt{5})^2 - 3(3 + \sqrt{5}) + b = 0.$ 
|
```

Compute:

```
|
|  $(9 + 6\sqrt{5} + 5) - 9 - 3\sqrt{5} + b = 0,$ 
|
```

```
|
|  $14 + 6\sqrt{5} - 9 - 3\sqrt{5} + b = 0,$ 
|
```

```
|
|  $5 + 3\sqrt{5} + b = 0.$ 
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

```
|
```

This is a valid value for (b) , and it is not an integer. The conjugate root is not necessarily present because (b) is not constrained to be an integer? The problem does not state that (b) is integer, only to compute it.

Therefore, we can proceed with this.

```
**Final Answer:**
```

```
|
|  $\boxed{-5 - 3\sqrt{5}}$ 
```

∨

Output:

```
def givenRoot : Float := 3.0 + 2.2360679775
def input_b : Float := -5.0 - 3*2.2360679775 --- Float value of b= -5 - 35
```

```
def evalPoly (x b : Float) : Float := x*x - 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps
```

```
def isRoot : Bool := approxZero (evalPoly givenRoot input_b)
```

```
#eval ("Is 3+sqrt5 a root of polynomial?", isRoot)
```

Now do this for the following input:

```
{initial_solution}
"""
```

LEAN_CODE_INTERPRETATION_PROMPT = """

Given the given Lean code input, and its output, interpret the results and provide a concise explanation of the error.
Do not include any additional information or context, just the interpretation of the Lean output.

Use the following example as a guide for your response:

Input:

Lean code:

```
def candidateRoot : Float := 3.0 + 2.2360679775 ---- 3 + 5
def input_b : Float := 4.0 ---- Float value of b=4
```

```
def evalPoly (x b : Float) : Float := x*x - 3.0*x + b
def approxZero (x : Float) (eps : Float := 0.01) : Bool := Float.abs x < eps
```

```
#eval ("Is 3+sqrt5 a root of polynomial?", isRoot)
```

Lean output:

```
#eval result: ("Is 3+sqrt5 a root of polynomial?", false)
```

Output:

3+sqrt{5} is not a root of the polynomial $x^2 - 3x + b = 0$ for the given value of b .

Now this for the following input:

Lean code:

```
{lean_code}
```

Lean output:

```
{lean_output}
"""
```

=== Helper functions ===

```
def call_deepseek(user_prompt: str, system_prompt: str) -> str:
```

```
    response = client.chat.completions.create(
        model="deepseek-chat",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": user_prompt}
        ],
        temperature=0.0,
        max_tokens=1024,
        stream=False
    )
```

```
    return response.choices[0].message.content.strip()
```

```
def write_lean_code(code: str):
```

```
    with open(LEAN_FILE_PATH, "w", encoding="utf-8") as f:
        f.write(code)
```

```
def run_lean_code() -> tuple[bool, str]:
```

```
    result = subprocess.run(
        ["lake", "build"],
        cwd=LEAN_PROJECT_DIR,
        capture_output=True,
        text=True
    )
```

```
    stdout, stderr = result.stdout, result.stderr
```

```
    # Assume build success if no 'error' in stderr
```

```
    success = "error" not in stderr.lower()
```

```
    output = f"=== STDOUT ===\n{stdout}\n\n=== STDERR ===\n{stderr}"
```

```
    # Clear Lean file after build (optional)
```

```
    # with open(LEAN_FILE_PATH, "w", encoding="utf-8") as f:
    #     f.write("")
```

```
    return success, output
```

```
def clean_lean_code(raw_code: str) -> str:
```

```
    """
    Extract Lean code from API response, stripping any markdown fences and leading/trailing text.
    """
```

```
    # Remove markdown triple backticks if present
```

```
    if raw_code.startswith("```");
```

```
        # Find the first and last triple backtick
```

```
        parts = raw_code.split("```")
```

```
        # The code is usually the second element after split: ```lang\ncode\n```
```

```
        if len(parts) >= 3:
```

```

        code = parts[1]
    else:
        code = raw_code
    else:
        code = raw_code

    # Strip leading/trailing whitespace
    code = code.strip()

    return code

def run_strategy(trial_dir, strategy_name, prompt_template, problem, previous_answer=None, error=None):
    log_path = os.path.join(trial_dir, f"{strategy_name}.csv")
    with open(log_path, "w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(["attempt", "prompt", "response", "lean_code", "lean_output", "Correct/Incorrect"])

    current_previous_answer = previous_answer
    current_error = error

    for attempt in range(1, MAX_ATTEMPTS + 1):
        if strategy_name == "no_feedback":
            prompt = prompt_template.format(problem=problem)
        elif strategy_name == "binary":
            prompt = prompt_template.format(problem=problem, previous_answer=current_previous_answer)
        elif strategy_name == "error_prev_sol":
            prompt = prompt_template.format(problem=problem, previous_answer=current_previous_answer, error=current_error)
        elif strategy_name == "error_no_prev_sol":
            prompt = prompt_template.format(problem=problem, error=current_error)

        print(f"Attempt {attempt} with strategy {strategy_name}")

        response = call_deepseek(prompt, system_prompt="You are a concise math problem solver.")
        match = re.search(r'\boxed{\{([^\}]*)\}}', response)
        if not match:
            solved = False
            writer.writerow([attempt, prompt, response, "", "", 0])
            continue

        # Generate Lean code
        lean_code = call_deepseek(
            LEAN_CODE_GENERATION_PROMPT.format(initial_solution=response),
            system_prompt="You are a Lean code generator and verifier for math solutions."
        )
        lean_code_clean = clean_lean_code(lean_code)
        write_lean_code(lean_code_clean)

        success, lean_output = run_lean_code()
        solved = ("false" not in lean_output.lower())

        if solved:
            writer.writerow([attempt, prompt, response, lean_code_clean, lean_output, "Correct"])
            return (attempt, "Solved")

        error_description = call_deepseek(
            LEAN_CODE_INTERPRETATION_PROMPT.format(
                lean_code=lean_code_clean,
                lean_output=lean_output
            ),
            system_prompt="You are a Lean code interpreter and verifier for math solutions."
        )

        writer.writerow([attempt, prompt, response, lean_code_clean, lean_output, "Incorrect"])
        current_previous_answer = response
        current_error = error_description

    return (MAX_ATTEMPTS, "Not Solved")

# === Main Loop ===
def feedback_loop():
    os.makedirs(LOG_ROOT, exist_ok=True)
    summary_path = os.path.join(LOG_ROOT, "trial_summaries.csv")
    with open(summary_path, "w", newline="", encoding="utf-8") as f_sum:
        summary_writer = csv.writer(f_sum)
        summary_writer.writerow(["Trial No.", "No Feedback Attempts", "Binary Attempts", "Error With Solution Attempts", "Error Without Solution Attempts"])

    trial = 0
    first_try_correct = 0
    incomplete_answer = 0
    while trial < MAX_TRIALS:
        print(f"\n=== Starting Trial {trial + 1} ===")
        # Initial solution
        initial_prompt = INITIAL_PROMPT.format(problem=problem)
        initial_solution = call_deepseek(initial_prompt, system_prompt="You are a concise math problem solver.")

        match = re.search(r'\boxed{\{([^\}]*)\}}', initial_solution)
        if not match:
            incomplete_answer += 1
            print("Incomplete answer, retrying trial.")
            continue

        lean_code = call_deepseek(
            user_prompt=LEAN_CODE_GENERATION_PROMPT.format(initial_solution=initial_solution),
            system_prompt="You are a Lean code generator and verifier for math solutions."
        )

```

```

clean_code = clean_lean_code(lean_code)
write_lean_code(clean_code)

# Step 4: Run Lean build
success, lean_output = run_lean_code()

# Check if first try correct
if "false" not in lean_output.lower():
    first_try_correct += 1
    print("First try correct, skipping trial (not counted).")
    continue

print("First try incorrect, proceeding with feedback strategies.")

trial += 1

trial_dir = os.path.join(LOG_ROOT, f"trial_{trial}")
os.makedirs(trial_dir, exist_ok=True)

error_description = call_deepseek(
    user_prompt=LEAN_CODE_INTERPRETATION_PROMPT.format(
        lean_code=clean_code,
        lean_output=lean_output
    ),
    system_prompt="You are a Lean code interpreter and verifier for math solutions."
)

# Step 3: Run feedback arms

print(f"Running No Feedback strategy for Trial {trial}")
attempts_no_feedback, solved_no_feedback = run_strategy(trial_dir, "no_feedback", INITIAL_PROMPT, problem=problem)
if solved_no_feedback == "Not Solved":
    attempts_no_feedback = "Did not solve"

print(f"Running Binary Correctness strategy for Trial {trial}")
attempts_binary, solved_binary = run_strategy(trial_dir, "binary", BINARY_CORRECT_PROMPT, problem=problem,
    previous_answer=initial_solution)
if solved_binary == "Not Solved":
    attempts_binary = "Did not solve"

print(f"Running Error Feedback with Previous Solution strategy for Trial {trial}")
attempts_error_prev_sol, solved_error_prev_sol = run_strategy(trial_dir, "error_prev_sol", RETRY_SOLVE_ERROR_PROMPT, problem=problem,
    previous_answer=initial_solution, error=error_description)
if solved_error_prev_sol == "Not Solved":
    attempts_error_prev_sol = "Did not solve"

print(f"Running Error Feedback without Previous Solution strategy for Trial {trial}")
attempts_error_no_prev_sol, solved_error_no_prev_sol = run_strategy(trial_dir, "error_no_prev_sol",
    RETRY_SOLVE_WITHOUT_PREVIOUS_ANSWER_PROMPT, problem=problem,
    error=error_description)
if solved_error_no_prev_sol == "Not Solved":
    attempts_error_no_prev_sol = "Did not solve"

# Step 4: Log trial summary
print(f"Trial {trial} Summary: No Feedback Attempts: {attempts_no_feedback}, Binary Attempts: {attempts_binary}, Error with Prev Attempts: {attempts_error_prev_sol}, Error without Prev Attempts: {attempts_error_no_prev_sol}")
summary_writer.writerow([trial, attempts_no_feedback, attempts_binary, attempts_error_prev_sol, attempts_error_no_prev_sol])

print(f"\n=== Experiment Completed ===")
print(f"Total Trials Conducted: {trial}")
print(f"First Try Correct (not counted in trials): {first_try_correct}")
print(f"Incomplete Answers (not counted in trials): {incomplete_answer}")

if __name__ == "__main__":
    feedback_loop()

```