UTMath: A Benchmark for Math Evaluation with Unit Test

Anonymous ACL submission

Abstract

The evaluation of mathematical reasoning capabilities constitutes a critical pathway toward achieving Artificial General Intelligence (AGI). Prevailing benchmarks including MATH and AIME mainly feature single-instantiation problems with fixed numbers, permitting pattern matching instead of principled deductive reasoning and leaving generalization on isomorphic problem variants untested. To address these limitations, we propose the UTMath Benchmark, employing rigorous unit testing methodology that simultaneously quantifies solution accuracy and solution space generality. It comprises 1,053 problems spanning 9 mathematical domains, each accompanied by an average of 68 varied test cases. With 10^7 answer possibilities per problem on average, UTMath sets new standards for robust reasoning while preventing memorization. UTMath is highly challenging, with the best-performing model, o1-mini, solving only 32.57% of the problems, followed by o1-preview at 27.16%, and GPT-4o at 26.93%. We further propose Reasoning-to-Code Thoughts (RCoT), a prompting strategy that decouples symbolic reasoning from code synthesis. RCoT guides LLMs to first derive formal reasoning structures before generating executable code, producing generalizable solutions rather than situation-specific answers. To help the community push mathematical reasoning further, we release UTMath-Train (70k samples), a companion training set generated under the same protocol. Our benchmark can be accessed via the following link: UTMath

1 Introduction

The pursuit of AGI necessitates strong mathematical reasoning capabilities, making the evaluation of such abilities a crucial area of research (Zhou et al., 2024a). Recent advancements in LLMs have demonstrated remarkable proficiency in solving complex mathematical problems, achieving amazing performance on various datasets of Math Word Problems (MWPs), such as GSM8K (Cobbe et al., 2021), MATH (Hendrycks et al., 2021), TheoremQA (Chen et al., 2023).

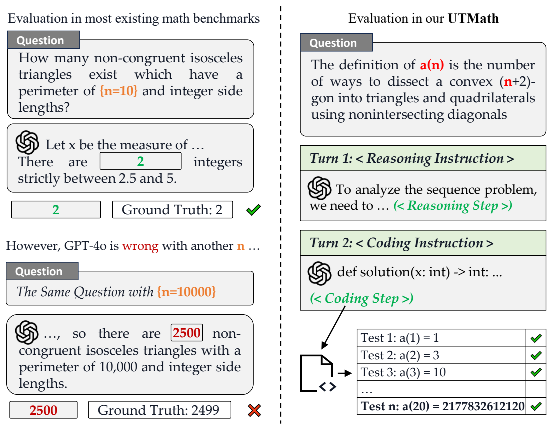

Figure 1: Comparison of UTMath with other benchmarks. On the left, GPT-4o successfully solved the original problem, but failed to generalize when the input was modified by merely changing a single numeric value. On the right, UTMath is shown, where each problem includes multiple test cases, and a solution is deemed correct only if all are passed by the code generated by the model. We also propose a new prompting method RCoT in which the LLM first reasons through the problem and then generates code.

However, conventional benchmarks present several intrinsic limitations that hinder the precise and comprehensive assessment of these models' mathematical reasoning capabilities (Ahn et al., 2024). First, these benchmarks predominantly assess models using narrowly defined problem formats with fixed numerical instantiations, thereby constraining their ability to evaluate generalization across structurally analogous but variationally distinct scenarios, as illustrated in Fig. 1. Second, the evaluation protocols often depend on rule-based matching or the LLM-as-a-Judge paradigm ((Dubois et al., 2024; Zheng et al., 2023)) both of which are vulnerable to inconsistencies due to the stochastic nature of LLM outputs. For example, in datasets

Dataset	Size	Level	Multi-test	Efficiency	Metric	Output
College Math	2,818	University	✗	✗	Accuracy	Text
GSM8K	1,319	Elementary school	✗	✗	Accuracy	Text
MATH	5,000	High school	✗	✗	Accuracy	Text
RobustMath	300	High school	✗	✗	Accuracy	Text
OlympiadBench	8,476	Competition	✗	✗	Accuracy	Text
TheoremQA	800	University	✗	✗	Accuracy	Text
UTMath(ours)	1,053	Cutting-edge	✓	✓	Pass Rate	Code

Table 1: Comparison between UTMath and other benchmarks. UTMath offers a cutting-edge benchmark with a comprehensive set of 1,053 problems across multiple mathematical domains, providing a more accurate evaluation of LLMs' mathematical reasoning capabilities.

such as GSM8K, TheoremQA, and MATH, a correct solution must be extracted in a form that exactly matches the reference answer, which fails to accommodate semantically equivalent but syntactically divergent responses. While recent work has made great progress in developing new benchmarks, many of these approaches still fall short of addressing the fundamental limitations of earlier datasets. For instance, benchmarks like GSM-HARD (Gao et al., 2023), GSM-IC (Shi et al., 2023), GSM-Plus (Li et al., 2024a), MetaMath (Yu et al., 2023) build upon GSM8K or MATH by introducing perturbations—including value substitution, input reversal, and distractor insertion. While these augmentations provide incremental improvements, they are often constrained by limited coverage and incur substantial human and computational costs. Against this backdrop, our work aims to fill these critical gaps by constructing a principled and robust benchmark capable of rigorously evaluating the mathematical reasoning abilities of LLMs.

Inspired by evaluation paradigms in software engineering, we adopt a unit testing-based framework to assess the soundness of LLMs' reasoning processes. In this framework, a solution that passes all unit tests is considered to reflect reliable and consistent reasoning. To this end, we introduce **UTMath**, a novel benchmark derived from the On-Line Encyclopedia of Integer Sequences (OEIS) (OEIS Foundation Inc., 2024). As shown in 1, the benchmark consists of 1,053 cutting-edge problems spanning 9 mathematical domains, such as Number Theory and Geometry. Each problem is accompanied by more than 68 test cases, each consisting of concrete input-output pairs that enable precise evaluation of generalization and correctness.

UTMath employs a unit-test-driven framework to evaluate mathematical reasoning through generalizable code solutions that must pass multiple test cases per problem class. Unlike benchmarks focused on numerical answers, this design explicitly requires executable implementations, testing both conceptual understanding and code-generation rigor—an advantage aligning with real-world problem-solving where precision and adaptability are critical. When testing Program-of-Thoughts (PoT) (Chen et al., 2022), we observed that models' coding limitations directly hindered performance. To address this issue, we decouple reasoning and coding, and propose Reasoning-to-Code of Thoughts (RCoT), which requires the LLM to first perform mathematical reasoning in the initial turn, and then generate code based on that reasoning in the subsequent turn. Our experiments demonstrate that RCoT encourages more thorough reasoning before code generation, leading to more efficient and targeted solutions.

UTMath is highly challenging, we conducted a comprehensive study with 8 LLMs. Some of our key findings are summarized as follows: (1) with the best-performing model, o1-mini, solving only 32.57% of the problems, followed by o1-preview at 27.16%, and GPT-4o at 26.93%, these results demonstrate the difficulty of UTMath. (2) Modern LLMs perform poorly in Graph Theory, Group Theory, Geometry and Topology (Fig. 5). (3) With RCoT, all evaluated LLMs generated more efficient solutions, with most models achieving higher scores (Fig. 3). (4) RCoT can improve the pass@k performance of LLMs (§ 5.4). (5) Both reasoning and coding capabilities substantially influence overall performance. By leveraging the modular design of RCoT, we can disentangle their individual contributions (§ 5.5). More interesting findings can be found in § 5. We hope our findings contribute to a deeper understanding of current reasoning ability of LLMs and the further development of models.

2 Related Work

2.1 Benchmarks

With the rapid development of LLMs, evaluating and exploring the intelligence and limitations of these models has emerged as an urgent issue to address (Chang et al., 2024). Reasoning ability, as a crucial component of general intelligence, has garnered widespread attention since the advent of LLMs (Patel et al., 2021; Cobbe et al., 2021; Valmeekam et al., 2022; Perez et al., 2022; Gupta et al., 2022; Shakarian et al., 2023). Mathematical reasoning, due to its complex mathematical characteristics and rigorous logical relationships, is considered an abstract and high-difficulty task, playing a pivotal role in demonstrating a model's reasoning capabilities.

To this end, researchers have proposed various benchmarks focused on mathematical reasoning. A natural and mainstream approach is to evaluate LLMs as humans would take math exams, categorized by required knowledge levels. Examples include GSM8K at elementary school level, Math and GaokaoBench-Math (Zhang et al., 2023) at high school level , College Math (Tang et al., 2024), TheoremQA (Chen et al., 2023), ARB (Sawada et al., 2023) at university level, and Olympiad-Bench (He et al., 2024), AGIeval-Math (Zhong et al., 2023) at competition level.

Besides, researchers have also introduced many others focused on evaluating various aspects of LLMs like the robustness. These include GSM8K-based variants: GSM-8K-Adv (Anantheswaran et al., 2024), GSM-Hard (Gao et al., 2023), GSM-Plus (Li et al., 2024a), GSM-IC (Shi et al., 2023), and several independent benchmarks: RobustMath (Zhou et al., 2024b), MetaMathQA (Yu et al., 2023), PROBLEMATHIC (Anantheswaran et al., 2024), MATHCHECK (Zhou et al., 2024a), as well as other benchmarks (Li et al., 2024b, 2023).

The distinctions between our proposed benchmark and existing ones are as follows. (1) Multiple Case Validation. Instead of using single cases that can be memorized, our questions are sequence-based, allowing numerous cases for validating true understanding. (2) General Solutions. UTMath requires large models to solve problems by generating code, aiming for general solutions rather than problem-specific ones, reflecting a closer alignment with intelligence.

2.2 Building Methods

Constructing effective, high-quality datasets is a complex and labor-intensive process. The advent of LLMs offers an opportunity to change this scenario (Valmeekam et al., 2022; Drori et al., 2023; Perez et al., 2022; Chiang and Lee, 2023; Liu et al., 2023; Fu et al., 2023; Kocmi and Federmann, 2023; Li et al., 2024b). For instance, (Almoubayyed et al., 2023) employed GPT-4o to rewrite mathematics problems based on MATHia (Ritter et al., 2007) to aid students in improving their math performance. These efforts provide a reliable foundation for utilizing LLMs in data processing.

In our study, we utilized GPT-4o to help us deal with data, such as by providing necessary background knowledge for questions and making them more understandable, with more information about the prompts used shown in the Appendix C. Subsequently, human verification was performed to ensure consistency before and after LLM usage.

2.3 Prompting Methods

Considering the attributes of large models, they exhibit significant sensitivity to prompts, rendering prompt engineering a critical area of study.

The Chain-of-Thought (Wei et al., 2022) prompting technique encourages models to express reasoning steps in natural language before concluding. Similarly, the approach by (Kojima et al., 2022) uses the phrase "Let's think step by step" to effectively guide large language models through their reasoning. Inspired by CoT, several effective prompting methods have been developed, such as Tree-of-Thoughts (Yao et al., 2024), Graph-of-Thoughts (Besta et al., 2024). Program-of-Thought prompting (Chen et al., 2022): PoT generates programs as the intermediate steps and integrates external tools like a Python interpreter for precise calculations, as well as other prompting methods (Wang et al., 2023; Gao et al., 2023; Xu et al., 2024b; Qian et al., 2023).

Our RCoT method stands out by dividing reasoning into two steps: reasoning and implementing based on reasoning. The advantages can be summarized as follows. (1) Modularity. By separating reasoning from implementation, the influence of coding on reasoning can be eliminated, providing a new paradigm for evaluating the reasoning ability through the code generated by the model. (2) Enhanced Reasoning. Emphasizing reasoning allows large models to focus more on improving the qual-

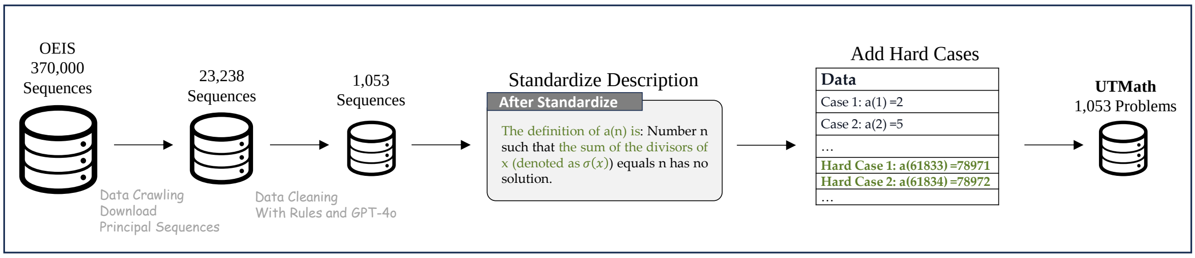

Figure 2: UTMath generation pipeline. After downloading 23,238 Principle Sequences from OEIS and cleaning the data, 1,053 usable sequences were obtained. Descriptions were standardized by adding background information and improving readability (highlighted in green, also shown in Appendix B.2). Hard cases were introduced to enhance discriminative capability, including terms from later positions to prevent simplistic algorithms from passing.

ity of reasoning, thereby delivering higher-quality and more efficient solutions.

3 UTMath Benchmark

3.1 Introduction for OEIS.

The OEIS was established to document integer sequences of interest to both professional and amateur mathematicians, and it has become widely cited in the community. Most sequences are derived or updated from academic papers, contributing to their cutting-edge level of difficulty (Allouche and Shallit, 2003). As of February 2024, it contains over 370,000 sequences (OEIS Foundation Inc., 2024). Each sequence is accompanied by an identification number, a brief description, some sample integers, links to relevant literature, and, where possible, program code for computing the sequences. An example sequence is shown in Appendix A.

3.2 Benchmark Construction.

UTMath is a cutting-edge and expansive benchmark designed to more accurately assess the mathematical reasoning abilities of LLMs. It consists of 1053 math problems, with each problem having an average of 68 test cases. The benchmark covers 9 mathematical domains, including not only common topics like number theory but also graph theory, group theory, topology, and geometry. The difficulty of UTMath is considered Cutting-Edge, with the majority of the sequences that form the problems having been studied in academic papers. UTMath was obtained as follow (see also Fig.2).

Data Crawling. OEIS provides users with a list of principal sequences [1], which are most important sequences defined by OEIS. OEIS categorizes these sequences into sections based on the first 2-3 letters of their content themes. By scraping the category tags within each section and the AIDs of their subordinate sequences, we obtained 23,238 principal sequences' AIDs. OEIS provides an interface to request the JSON data of the HTML page for each sequence using its AID [2]. By passing the sequence AIDs to this interface, we acquired the JSON data for these 23,238 sequences.

Data Cleaning. We found that some of the sequences we collected did not meet our criteria and should be removed, with further details provided in the Appendix B. Here are several main situations.

• **Hard to solve, few terms are discoverable.** A portion of the sequences retrieved are marked as "hard" in the keyword field of their entries in OEIS. According to OEIS, "Any sequence which can be extended only by new ideas, rather than more computation deserves keyword: hard. Similarly, if computing a term of the sequence would probably merit a paper in a peer-reviewed journal (discussing the result, the algorithm, etc.)" [3] Another related keyword attribute is "fin" (finite), indicating sequences with limited length. For our purposes, sequences should be infinitely derivable.

• **Difficult to Generate Programmatically.** In OEIS, most sequences are provided with fields such as Mathematica, program, or formula, but not all sequences include these details. We assume that the sequences without these fields may be difficult to generate programmatically.

• **Simple Sequences.** Some sequences are too simple to require any reasoning. We use GPT-4o to determine if a sequence requires reasoning or just implementation; if mostly implementation, it's

[1] https://oeis.org/wiki/Index_to_OEIS

[2] https://oeis.org/wiki/JSON_Format

[3] https://oeis.org/wiki/User:Charles_R_Greathouse_IV/Keywords/difficulty

excluded. For instance, A000178[4]: 'Superfactorials: product of the first n factorials,' a sequence requiring only implementation, will be excluded.

After addressing the aforementioned issues, we ultimately obtained 1053 sequences.

Standardization of Question Statements. As a academic database in the field of mathematics, OEIS provides a wealth of useful information for each sequence. However, we have found that some sequences cannot be directly used with the descriptions provided by OEIS as problem statements, primarily for the following reasons: (1) Specialized Terminology. Some sequence descriptions use complex math terms that need examples or explanations to be clear. Using them directly as problems might test mathematical knowledge rather than reasoning skills. So, it is important to explain key concepts to focus on reasoning and reduce the extra knowledge needed. (2) Brevity and Ambiguity. Some sequence descriptions are excessively brief and lack a clear definition of what a(n) is. We used GPT-4o to standardize these by adding background info and making the language smoother. The prompts we used are provided in the Appendix C and an example is shown in Appendix B.2.

Hard Test Cases Mining. The primary goal of this paper is to evaluate the reasoning ability of LLMs. Generally, more efficient solutions to a problem imply stronger reasoning capabilities. Therefore, we aim for our evaluation to distinguish whether a solution is efficient. However, in the OEIS, each sequence only lists the first few n terms, normally n<100, which can be obtained without requiring particularly efficient methods. This limitation prevents the evaluation from effectively distinguishing between efficient and inefficient solutions. An obvious fact is that the difficulty of computing the first 10 terms of a sequence within a time limit is significantly different from computing terms starting from the 10^6th term. Therefore, we aim to create more challenging test data to better assess the reasoning capabilities of LLMs.

Fortunately, many OEIS sequences include corresponding Mathematica code that can be regarded as the ground-truth solution for each problem. We extract this Mathematica code for each sequence, formalizing it to compute the first N terms, $A_1, \ldots, A_N$, of the sequence. We determine the maximum value of N_{max} for which the code can compute the sequence within 10 seconds, where we set 10^6 as the upper bound. Finally, we add the last 10 terms $A_{N_{max-9}}, \ldots, A_{N_{max}}$ into our benchmark as the hard test cases to evaluate the complexity of a solution. Our experiments demonstrate that these cases precisely differentiate more efficient and intelligent solutions.

[4] https://oeis.org/A000178

Category	# of Problems
Number Theory	159
Graph Theory	79
Group Theory	65
Discrete Mathematics	158
Combinatorial Mathematics	158
Geometry and Topology	70
Poly. and Series Expan.	151
Special Numbers	157
Formal Languages	56
Total	1053

Table 2: Categories and distribution of problems.

3.3 Evaluation Metrics

We adopt the metric pass@k to evaluate the performance of LLMs. The metric pass@k is a classic metric in code generation, where a problem is solved if any of the k generated samples passes the unit tests. We use the stable method of calculation proposed by (Chen et al., 2021):

$$\text{pass@}k := \mathbb{E}_{\text{Problems}}\left[1 - \binom{n-c}{k} \Big/ \binom{n}{k}\right] \quad (1)$$

3.4 Dataset Statistics

The main statistics of UTMath are shown in Tab. 1. To gain a deeper understanding of the composition of the UTMath Benchmark, we identified nine mathematical fields and used GPT-4o to categorize each problem to these fields as shown in Tab. 2. Our analysis reveals that only 10 out of 1,053 problems have no references. The reference years span from 1950 to 2024, with the maximum number of references exceeding 6,000. These findings underscore the cutting-edge nature of our benchmark. More details can be found in Appendix B.

As noted in (Xu et al., 2024a), digit length plays a critical role in the performance of CoT-based LLMs. UTMath includes 61,582 easy cases, with a median digit length of 2 and a maximum of 18. It also contains 10,530 hard cases, with a median digit length of 8 and a maximum of 712.

Model	Pass@1 (%) ↑		Pass@5(%) ↑		Avg. Run Time (s) ↓		
	PoT	RCoT	PoT	RCoT	PoT	RCoT	Efficiency
closed-source models							
o1-mini	**29.34**	**32.57** (+3.23)	——	——	5.58	3.76	+32.62%
o1-preview	23.74	27.16 (+3.42)	——	——	4.66	3.96	+15.02%
GPT-4o	25.53	26.93 (+1.40)	**32.67**	**35.90** (+3.23)	6.98	6.23	+12.04%
Gemini-1.5-Pro	19.70	19.43 (−0.27)	31.24	33.14 (+1.90)	6.30	6.22	+1.28%
Claude-3.5-Sonnet	18.58	19.11 (+0.53)	27.83	31.34 (+3.51)	6.44	5.32	+21.05%
GPT-3.5-Turbo	11.68	6.82 (−4.86)	17.09	13.30 (−3.79)	5.42	5.06	+7.11%
open-source models							
Qwen2.5-72B	**23.48**	**22.17** (−1.31)	**31.05**	**33.33** (+2.28)	5.88	4.31	+36.42%
DeepSeek-V2.5-236B	20.95	21.63 (+0.68)	30.10	31.72 (+1.62)	6.64	5.44	+22.06%
Qwen2.5-Math-72B	19.72	20.53 (+0.81)	26.69	28.11 (+1.42)	5.04	3.81	+24.40%
Qwen2.5-Coder-32B	18.71	20.23 (+1.52)	26.88	35.04 (+8.16)	8.33	6.83	+18.01%
LLaMA-3.1-405B	15.76	16.09 (+0.33)	25.26	27.35 (+2.09)	5.73	5.12	+11.91%

Table 3: Pass Rate and Average Run Time of LLMs on UTMath. We listed the performance of 11 large models by the PoT or the RCoT methods across a range of metrics. For o1-mini and o1-preview only Pass@1 data is currently available due to resource constraints. The average run time is calculated based on the problems solved by both the PoT and the RCoT methods. The efficiency is calculated as: (Avg.Runtime(PoT) - Avg.Runtime(RCoT)) / Avg.Runtime(RCoT). Two qualitative cases are shown in Appendix D.

4 Reasoning-to-Code Thoughts

Compared to methods that simply check whether the outputs generated by LLMs are identical, the code-based evaluation approach enables more accurate assessment by using multiple test cases.

Initially, we adopted the Program of Thought (PoT) method, where the LLM was required to perform reasoning and code implementation in a single step. However, we observed that the model's code generation capability also influenced its performance on UTMath. To address this, we propose the Reasoning-to-Code of Thoughts (RCoT) framework, which decouples reasoning and coding into two separate rounds of interaction.

In the first round, the model is tasked solely with mathematical reasoning, without generating any code. In the second round, the model generates code based on the reasoning process from the first round. We can either use the same model for both rounds to observe its overall performance on UTMath, or isolate the reasoning capability by fixing the second-round model to a dedicated coding model. The latter setting enables a more accurate evaluation of the model's reasoning ability.

Moreover, we find that by separating reasoning from code generation, RCoT allows the LLM to generate a step-by-step, detailed reasoning chain that includes relevant theorems, formulas, and mathematical properties. Such deeper reasoning leads to the development of more efficient algorithms with lower time complexity. We present qualitative cases in Appendix D.

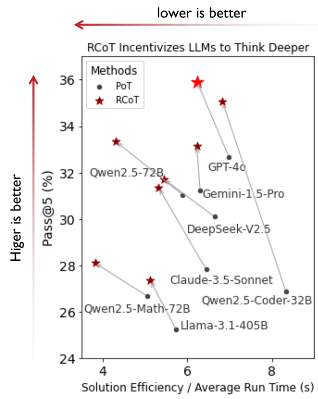

Figure 3: RCoT improves the effectiveness of the solution and significantly enhances its efficiency. It indicates that our RCoT proves to be more effective, suggesting that it encourages the model to reason critically and find more efficient solutions.

5 Experiment

5.1 Experimental Setup

Here, we consider the closed-source models, i.e., GPT-3.5-Turbo, GPT-4o, o1-mini and o1-preview from OpenAI (OpenAI, 2024), Claude-3.5-Sonnet (Claude, 2024), Gemini-1.5-Pro (Reid et al., 2024), as well as the open-source models, i.e., LLaMA-3.1 (Dubey et al., 2024), Qwen2.5 (Qwen, 2024a), Qwen2.5-Math (Qwen, 2024b), Qen2.5-Coder (Hui et al., 2024), DeepSeek-V2.5 (Bi et al., 2024). The metric pass@1 is calculated as the average result over 5 run times. We run all evaluations in a laptop with CPU Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz.

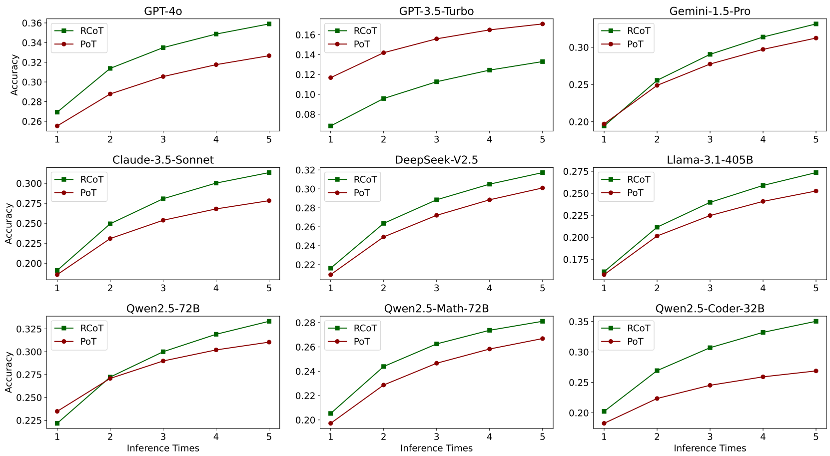

Figure 4: Performance comparison of models across PoT and RCoT tasks at different pass@k levels.

5.2 Evaluation on UTMath

Here we evaluate both open-source and closed-source models using RCoT and PoT in Tab. 3. The experimental results shows that all tested models performed poorly on our benchmark. The best model, o1-mini, only solves 32.57% problem in our benchmark, followed by o1-preview at 27.16% and GPT-4o at 26.93%. Since our problems are sourced from the OEIS, they consist of sequences and solutions proposed by various mathematicians in the context of cutting-edge research. This suggests that our benchmark is challenging enough to help guide future directions for improving LLMs.

Compared to PoT, our method RCoT demonstrates superiority in two aspects. First, prompting with RCoT achieves higher pass@5 performance across 8 LLMs, with the best results observed on GPT-4o. Second, the solutions generated by RCoT for all LLMs demonstrate more efficient performance, particularly Qwen2.5-72B, where the RCoT approach achieves an efficiency improvement of over 36.42% compared to PoT, as shown in Tab. 3 and Fig. 3. It indicates that, RCoT prompting enables the model to engage in deeper reasoning, enhancing solution performance and significantly reducing solution complexity.

However, some models experienced a decrease in pass@1 with RCoT . Specifically, the accuracies of Gemini-1.5-Pro, GPT-3.5-Turbo, and Qwen2.5-72B slightly dropped. Notably, while Gemini-1.5-Pro and Qwen2.5-72B experienced a drop in pass@1, their pass@5 performance improved. It indicates that RCoT brings more room in multiple inference times. The observed decrease in performance may stem from the fact that formulating more efficient solutions often requires higher-level reasoning, which can increase the difficulty of the task and make these models more susceptible to errors when attempting more sophisticated solutions.

	Model	Easy	Easy & Hard
closed	GPT-4o	**34.95**	**26.93**
	Gemini-1.5-Pro	23.84	19.43
	Claude-3.5-Sonnet	24.86	19.11
	GPT-3.5-Turbo	8.72	6.82
open	Qwen2.5-72B	**28.96**	**22.17**
	DeepSeek-V2.5	27.52	21.63
	Qwen2.5-Math-72B	24.60	20.53
	LLaMA-3.1-405B	22.09	16.09

Table 4: Performance (%) of different models on easy and hard test cases. Easy cases: The initial terms in OEIS. Hard cases: mined hard test cases (§ 3.2).

5.3 The Effectiveness of Hard Test Cases

As noted in § 3.2, OEIS sequences typically list only initial terms, which we treat as "easy test cases." To evaluate models under more challenging conditions, we test their ability to predict much later values (e.g., at position 10^6). These values are less likely to appear in pre-training data, reducing the risk of contamination, and are harder to compute within time limits, requiring more precise and efficient implementations. As shown in Tab. 4, model performance drops notably on these cases, suggesting their effectiveness in filtering out simplistic or brute-force solutions and enhancing the benchmark's discriminative power.

5.4 Scaling of the Inference Times

We compared the performance difference between running the LLMs five times and reported the met-

Model	NT	Graph T.	Group T.	DM	CM	GT	PSE	SN	FL	**pass@1**
					closed-source models					
o1-mini	**52.83**	**7.59**	**15.38**	**42.41**	**32.27**	**7.14**	**23.84**	**40.13**	37.50	**32.57**
o1-preview	47.17	6.33	13.85	34.17	25.32	2.86	23.18	29.94	33.93	27.16
GPT-4o	43.90	2.78	11.69	38.23	24.94	3.43	16.42	33.89	**42.50**	26.93
Gemini-1.5-Pro	31.70	1.27	8.92	27.47	15.19	5.71	15.23	27.39	17.86	19.43
Claude-3.5-Sonnet	33.58	1.52	8.00	29.49	12.91	5.43	11.52	26.62	20.36	19.11
GPT-3.5-Turbo	13.08	0.00	1.85	11.39	3.29	0.29	2.78	10.96	8.93	6.82
					open source models					
Qwen2.5-72B	36.86	2.53	**12.00**	30.63	15.95	**6.00**	**18.15**	29.43	**24.29**	**22.17**
DeepSeek-V2.5	**38.24**	1.27	8.92	**33.16**	**17.34**	2.29	12.45	**31.08**	20.00	21.63
Qwen2.5-Math-72B	35.35	1.27	8.62	28.73	14.81	4.00	17.48	28.15	20.00	20.53
Qwen2.5-Coder-32B	27.04	**8.86**	7.69	29.75	16.46	5.71	21.19	26.75	26.79	20.23
LLaMA-3.1-405B	29.56	0.76	4.92	25.44	9.62	2.00	9.54	22.55	21.43	16.09

Table 5: Performance (%) on different problem categories. Categories are represented by abbreviations. NT: Number Theory; T.: Theory; DM: Discrete Mathematics; CM: Combinatorial Mathematics; GT: Geometry and Topology; PSE: Polynomial and Series Expansions; SN: Special Numbers; FL: Formal Languages.

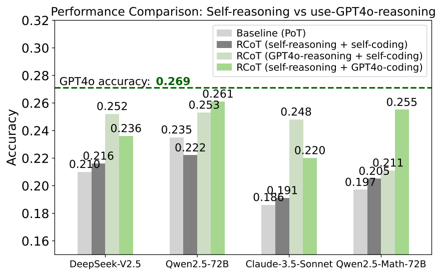

Figure 5: Performance comparison between self-reasoning and using GPT-4o reasoning for coding across different models.

ric of pass@k. As shown in Fig. 4, all models improved their performance with an increasing number of inference times. For Qwen2.5-72B and Gemini-1.5-Pro, RCoT was slightly weaker than PoT in pass@1 but quickly approached and surpassed PoT in subsequent run times. We observed that with an increasing number of inference time, RCoT consistently demonstrated a growing advantage in performance across almost all models, except for GPT-3.5. However, it is worth noting that GPT-3.5 exhibited the lowest pass rate. This suggests that RCoT may perform better in models with stronger reasoning capabilities.

5.5 Disentangling the Impact of Reasoning and Code Generation

We performed cross-model evaluations by pairing different models for the reasoning and code generation stages, as illustrated in Fig. 5. The findings suggest that both reasoning and coding capabilities substantially influence overall performance. By leveraging the modular design of RCoT, we can disentangle their individual contributions; for example, the light green configurations highlight the relative strengths of each model's reasoning ability.

5.6 Performance on Different Categories

Our benchmark comprehensively evaluates the LLMs' ability across various categories of math problems. GPT-4o achieved the highest score in the formal language domain, while o1-mini achieved the best scores in the remaining eight domains. All models performed poorly in the categories of Graph Theory and Geometry and Topology, with accuracy rates below 9%, highlighting the need for further exploration in these areas.

6 Conclusion

In this work, we investigate how to more accurately and effectively evaluate the mathematical reasoning capabilities of LLMs. We propose a cutting-edge benchmark, UTMath, which comprises 1,053 problems spanning nine mathematical domains, with an average of 68 test cases per problem. This benchmark presents challenges: o1-mini, the best-performing model, successfully solves only 32.57% of the problems, followed by o1- preview at 27.16%, and GPT-4o at 26.93%. Additionally, we introduce RCoT (Reasoning-to-Code of Thought). Our study finds that, compared to PoT, RCoT improves pass rates and significantly enhances algorithmic efficiency of most models. Overall, this research contributes to a deeper understanding of the current capabilities of LLMs in mathematical reasoning and lays the groundwork for the development of more advanced models in the future.

Limitation

The primary limitation of UTMath lies in the evaluation metrics: the performance of the evaluation machine affects the runtime of the generated code, making the absolute numerical results incomparable across different machines. We utilized an i7-10750H processor to execute the reference solutions and conduct evaluations, and we recommend using the same machine for testing and replication. There are two main limitations of RCoT. First, we only installed a set of common packages, such as sympy, in the standard testing environment. This avoids allowing LLMs to call highly integrated packages while also preventing the generation of potentially harmful code that could damage the evaluation system. Second, while our experiments demonstrate the critical role of reasoning quality in determining success rates, we have not further explored methods for enhancing reasoning quality, which remains an area for future investigation.

Ethics Statements

The UTMath Benchmark is designed to advance the evaluation of mathematical reasoning in LLMs. We recognize the potential ethical concerns associated with this work, particularly the risk of data misuse. To mitigate this, we strictly adhere to usage guidelines and licensing terms for the UTMath-Train dataset, which is intended solely for academic and research purposes. While the UTMath Benchmark evaluates model performance in terms of accuracy and generality, automated evaluations may introduce biases due to the nature of the datasets and evaluation algorithms. Additionally, while UTMath covers a wide range of mathematical domains, it may not fully represent diverse cultural or educational perspectives. We encourage further development of benchmarks that incorporate a broader array of reasoning styles to ensure more inclusive evaluations. By releasing UTMath, we aim to foster responsible AI development, promoting better, more generalizable mathematical reasoning systems.

References

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. *arXiv preprint arXiv:2402.00157*.

Jean-Paul Allouche and Jeffrey Shallit. 2003. *Automatic Sequences: Theory, Applications, Generalizations*. Cambridge University Press, Cambridge, UK.

Husni Almoubayyed, Rae Bastoni, Susan R Berman, Sarah Galasso, Megan Jensen, Leila Lester, April Murphy, Mark Swartz, Kyle Weldon, Stephen E Fancsali, et al. 2023. Rewriting math word problems to improve learning outcomes for emerging readers: a randomized field trial in carnegie learning's mathia. In *International Conference on Artificial Intelligence in Education*, pages 200–205. Springer.

Ujjwala Anantheswaran, Himanshu Gupta, Kevin Scaria, Shreyas Verma, Chitta Baral, and Swaroop Mishra. 2024. Investigating the robustness of llms on math word problems. *arXiv preprint arXiv:2406.15444*.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Wenhu Chen, Ming Yin, Max Ku, Pan Lu, Yixin Wan, Xueguang Ma, Jianyu Xu, Xinyi Wang, and Tony Xia. 2023. Theoremqa: A theorem-driven question answering dataset. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7889–7901.

Cheng-Han Chiang and Hung-yi Lee. 2023. Can large language models be an alternative to human evaluations? *arXiv preprint arXiv:2305.01937*.

Claude. 2024. Claude-3.5-sonnet. https://www.anthropic.com/claude/.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Iddo Drori, Sarah J Zhang, Reece Shuttleworth, Sarah Zhang, Keith Tyser, Zad Chin, Pedro Lantigua, Saisamrit Surbehera, Gregory Hunter, Derek Austin, et al. 2023. From human days to machine seconds: Automatically answering and generating machine learning final exams. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3947–3955.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Yann Dubois, Balázs Galambosi, Percy Liang, and Tatsunori B Hashimoto. 2024. Length-controlled alpacaeval: A simple way to debias automatic evaluators. *arXiv preprint arXiv:2404.04475*.

Yao Fu, Litu Ou, Mingyu Chen, Yuhao Wan, Hao Peng, and Tushar Khot. 2023. Chain-of-thought hub: A continuous effort to measure large language models' reasoning performance. *arXiv preprint arXiv:2305.17306*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Himanshu Gupta, Neeraj Varshney, Swaroop Mishra, Kuntal Kumar Pal, Saurabh Arjun Sawant, Kevin Scaria, Siddharth Goyal, and Chitta Baral. 2022. " john is 50 years old, can his son be 65?" evaluating nlp models' understanding of feasibility. *arXiv preprint arXiv:2210.07471*.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. 2024. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. *arXiv preprint arXiv:2402.14008*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Tom Kocmi and Christian Federmann. 2023. Large language models are state-of-the-art evaluators of translation quality. *arXiv preprint arXiv:2302.14520*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Qintong Li, Leyang Cui, Xueliang Zhao, Lingpeng Kong, and Wei Bi. 2024a. Gsm-plus: A comprehensive benchmark for evaluating the robustness of llms as mathematical problem solvers. *arXiv preprint arXiv:2402.19255*.

Tianle Li, Wei-Lin Chiang, Evan Frick, Lisa Dunlap, Tianhao Wu, Banghua Zhu, Joseph E Gonzalez, and Ion Stoica. 2024b. From crowdsourced data to high-quality benchmarks: Arena-hard and benchbuilder pipeline. *arXiv preprint arXiv:2406.11939*.

Zekun Li, Baolin Peng, Pengcheng He, and Xifeng Yan. 2023. Do you really follow me? adversarial instructions for evaluating the robustness of large language models. *arXiv preprint arXiv:2308.10819*.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. G-eval: Nlg evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634*.

OEIS Foundation Inc. 2024. The on-line encyclopedia of integer sequences. Published electronically at https://oeis.org.

OpenAI. 2024. Hello gpt-4o, 2024. https://openai.com/index/hello-gpt-4o/.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. 2021. Are nlp models really able to solve simple math word problems? *arXiv preprint arXiv:2103.07191*.

Ethan Perez, Sam Ringer, Kamilė Lukošiūtė, Karina Nguyen, Edwin Chen, Scott Heiner, Craig Pettit, Catherine Olsson, Sandipan Kundu, Saurav Kadavath, et al. 2022. Discovering language model behaviors with model-written evaluations. *arXiv preprint arXiv:2212.09251*.

Cheng Qian, Chi Han, Yi R Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. *arXiv preprint arXiv:2305.14318*.

Team Qwen. 2024a. Qwen2.5-72b-instruct. https://qwenlm.github.io/blog/qwen2.5/.

Team Qwen. 2024b. Qwen2.5-math-72b-instruct. https://qwenlm.github.io/blog/qwen2.5-math/.

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Steven Ritter, John R Anderson, Kenneth R Koedinger, and Albert Corbett. 2007. Cognitive tutor: Applied research in mathematics education. *Psychonomic bulletin & review*, 14:249–255.

Tomohiro Sawada, Daniel Paleka, Alexander Havrilla, Pranav Tadepalli, Paula Vidas, Alexander Kranias, John J Nay, Kshitij Gupta, and Aran Komatsuzaki. 2023. Arb: Advanced reasoning benchmark for large language models. *arXiv preprint arXiv:2307.13692*.

Paulo Shakarian, Abhinav Koyyalamudi, Noel Ngu, and Lakshmivihari Mareedu. 2023. An independent evaluation of chatgpt on mathematical word problems (mwp). *arXiv preprint arXiv:2302.13814*.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pages 31210–31227. PMLR.

Zhengyang Tang, Xingxing Zhang, Benyou Wan, and Furu Wei. 2024. Mathscale: Scaling instruction tuning for mathematical reasoning. *arXiv preprint arXiv:2403.02884*.

Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. 2022. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). In *NeurIPS 2022 Foundation Models for Decision Making Workshop*.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Ancheng Xu, Minghuan Tan, Lei Wang, Min Yang, and Ruifeng Xu. 2024a. Numcot: Numerals and units of measurement in chain-of-thought reasoning using large language models. *arXiv preprint arXiv:2406.02864*.

Jundong Xu, Hao Fei, Liangming Pan, Qian Liu, Mong-Li Lee, and Wynne Hsu. 2024b. Faithful logical reasoning via symbolic chain-of-thought. *arXiv preprint arXiv:2405.18357*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2023. Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*.

Xiaotian Zhang, Chunyang Li, Yi Zong, Zhengyu Ying, Liang He, and Xipeng Qiu. 2023. Evaluating the performance of large language models on gaokao benchmark. *arXiv preprint arXiv:2305.12474*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.

Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. 2023. Agieval: A human-centric benchmark for evaluating foundation models. *arXiv preprint arXiv:2304.06364*.

Zihao Zhou, Shudong Liu, Maizhen Ning, Wei Liu, Jindong Wang, Derek F Wong, Xiaowei Huang, Qiufeng Wang, and Kaizhu Huang. 2024a. Is your model really a good math reasoner? evaluating mathematical reasoning with checklist. *arXiv preprint arXiv:2407.08733*.

Zihao Zhou, Qiufeng Wang, Mingyu Jin, Jie Yao, Jianan Ye, Wei Liu, Wei Wang, Xiaowei Huang, and Kaizhu Huang. 2024b. Mathattack: Attacking large language models towards math solving ability. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19750–19758.

A An Example Sequence in OEIS

The OEIS is supported by the many generous donors to the OEIS Foundation.

THE ON-LINE ENCYCLOPEDIA OF INTEGER SEQUENCES ®

founded in 1964 by N. J. A. Sloane

Search Hints

(Greetings from The On-Line Encyclopedia of Integer Sequences!)

A007369 Numbers n such that sigma(x) = n has no solution. 51

(Formerly M1355)

2, 5, 9, 10, 11, 16, 17, 19, 21, 22, 23, 25, 26, 27, 29, 33, 34, 35, 37, 41, 43, 45, 46, 47, 49, 50, 51, 52, 53, 55, 58, 59, 61, 64, 65, 66, 67, 69, 70, 71, 73, 75, 76, 77, 79, 81, 82, 83, 85, 86, 87, 88, 89, 92, 94, 95, 97, 99, 100, 101, 103, 105, 106, 107, 109, 111, 113

(list; graph; refs; listen; history; text; internal format)

OFFSET 1,1

COMMENTS
With an initial 1, may be constructed inductively in stages from the list L = {1,2,3,....} by the following sieve procedure. Stage 1. Add 1 as the first term of the sequence a(n) and strike off 1 from L. Stage n+1. Add the first (i.e. leftmost) term k of L as a new term of the sequence a(n) and strike off k, sigma(k), sigma(sigma(k)),.... from L. - Joseph L. Pe, May 08 2002
This sieve is a special case of a more general sieve. Let D be a subset of N and let f be an injection on D satisfying f(n) > n. Define the sieve process as follows: 1. Start with the empty sequence S and let E = D. 2. Append the smallest element s of E to S. 3. Remove s, f(s), f(f(s)), f(f(f(s))), ... from E. 4. Go to step 2. After this sieving process, S = D - f(D). To get the current sequence, take f = sigma and D = {n | n >= 2}. - Max Alekseyev, Aug 08 2005
By analogy with the untouchable numbers (A005114), these numbers could be named "sigma-untouchable". - Daniel Lignon, Mar 28 2014
The asymptotic density of this sequence is 1 (Niven, 1951, Rao and Murty, 1979). - Amiram Eldar, Jul 23 2020

REFERENCES
M. Abramowitz and I. A. Stegun, eds., Handbook of Mathematical Functions, National Bureau of Standards Applied Math. Series 55, 1964 (and various reprintings), p. 840.
N. J. A. Sloane and Simon Plouffe, The Encyclopedia of Integer Sequences, Academic Press, 1995 (includes this sequence).

LINKS
M. F. Hasler, Table of n, a(n) for n = 1..10000 (first 1000 terms from T. D. Noe)
M. Abramowitz and I. A. Stegun, eds., Handbook of Mathematical Functions, National Bureau of Standards, Applied Math. Series 55, Tenth Printing, 1972 [alternative scanned copy].
Ivan Niven, The asymptotic density of sequences, Bull. Amer. Math. Soc., Vol. 57 (1951), pp. 420-434.
R. Sita Rama Chandra Rao and G. Sri Rama Chandra Murty, On a theorem of Niven, Canadian Mathematical Bulletin, Vol 22, No. 1 (1979), pp. 113-115.
R. G. Wilson, V, Letter to N. J. A. Sloane, Jul. 1992

FORMULA
A175192(a(n)) = 0, A054973(a(n)) = 0. - Jaroslav Krizek, Mar 01 2010
a(n) < 2n + sqrt(8n). - Charles R Greathouse IV, Oct 23 2015

EXAMPLE
a(4) = 10 because there is no x < 10 whose sigma(x) = 10.

MATHEMATICA
```
a = {}; Do[s = DivisorSigma[1, n]; a = Append[a, s], {n, 1, 115} ]; Complement[ Table[ n, {n, 1, 115} ],
  Union[a] ]
```

PROG
```
(PARI) list(lim)=my(v=List(), u=vectorsmall(lim\1), t); for(n=1, lim, t=sigma(n); if(t<=lim, u[t]=1));
  for(n=2, lim, if(u[n]==0, listput(v, n))); Vec(v) \\ Charles R Greathouse IV, Mar 09 2017
(PARI) A007369_list(LIM, m=0, L=List(), s)={for(n=2, LIM, (s=sigma(n-1))>LIM || bittest(m, s) || m+=1<<s;
  bittest(m, n)||listput(L, n)); L} \\ A bit slower, but bitmask requires less memory, avoiding stack
  overflow produced by the earlier code for lim = 1e6 with standard gp setup. - M. F. Hasler, Mar 12 2018
```

CROSSREFS
Complement of A002191.
See A083532 for the gaps, i.e., first differences.
See A048995 for the missed sums of nontrivial divisors.
Sequence in context: A133508 A125969 A070240 * A307213 A362398 A295567
Adjacent sequences: A007366 A007367 A007368 * A007370 A007371 A007372

KEYWORD nonn

AUTHOR N. J. A. Sloane, Mira Bernstein, Robert G. Wilson v

EXTENSIONS More terms from David W. Wilson

STATUS approved

Figure 6: Sequence A007369 in OEIS. Its description: "Numbers n such that sigma(x) = n has no solution." (Clearly, without specific background knowledge, we cannot fully understand what the function sigma() represents, which is one of the reasons we perform standardization. §B.2) Next, OEIS shows the first 67 terms of this sequence, which we classify as easy cases. Below that, additional metadata is provided, including comments, references, links, formulas, examples, programs, author, status, and more. It is evident that this sequence has garnered significant attention from researchers, reflecting the Cutting-Edge difficulty of our benchmark. We used the Mathematica program included in the metadata to generate Hard cases, with detailed procedures provided in § 3.2. As a scientific database, each sequence submitted to OEIS undergoes a review process, and the status "approve" indicates that the sequence has been validated and approved by OEIS administrators.

B Dataset Construction Details

This section primarily presents some details on the construction of UTMath. B.1 discusses the issues encountered when observing data crawled from OEIS, along with the corresponding cleaning rules. UTMath applies all 14 rules. Additionally, we crawled all sequences from OEIS and, for convenience, applied only the first 12 rules to create UTMath_Train, which contains over 70k sequences. B.2 outlines the process followed for standardizing the descriptions of problems in UTMath, while B.3 explains the referencing of sequences within UTMath, highlighting both the Cutting-Edge difficulty level of UTMath and its scalability.

B.1 Rules for Data Cleaning

1. Issues: The sequence is too difficult, requiring extensive background knowledge, or only a limited number of terms are found.
   Method: Remove sequences with keywords containing ‘hard’, ‘fin’ (finite).
2. Issues: The sequence is hard to generate with a program.
   Method: Check if it contains program, formula, or Mathematica fields in the sequence’s json data.
3. Issues: The sequence is too simple with an explicit recurrence or closed formula.
   Method: Search if the description includes ‘a(n) =’.
4. Issues: Solving the sequence requires information from other OEIS sequences.
   Method: Search if the sequence’s description contains the AID of other sequences (‘A’ + six-digit number with leading zeros).
5. Issues: The sequence is decimal expansion of a certain number.
   Method: Search if the description includes ‘decimal’.
6. Issues: The sequence consists of repetitions or a constant value.
   Method: Search if the description includes both ’repeat’ and ’period’ or ’constant sequence’.
7. Issues: The description is too vague.
   Method: Search if the description includes ‘related to’.
8. Issues: Another version of a concept.
   Method: Search if the title includes ‘another version’, ‘second kind’, etc.
9. Issues: The sequence is formed by taking mod of a constant.
   Method: Search if the description includes ‘module’.
10. Issues: The values in the sequence are too large, which might cause LLM tokenization errors.
    Method: Check if any term’s length exceeds 18 digits(i.e., greater than $1e18$), remove it.
11. Issues: Coefficient triangles or ‘read by row’ topics.
    Method: Search if the sequence’s description includes ‘read by row’, ‘triangle of coefficient’.
12. Issues: The description is too short, either purely implementation or lacks necessary information.
    Method: Check if the title length is below 5.
13. Issues: More like a reasoning puzzle.
    Method: Use GPT-4o to judge, with the prompt outlined in the Appendix C.
14. Issues: Non-mathematical topics.
    Method: Use GPT-4o to judge, with the prompt outlined in the Appendix C.

B.2 Standardization of Problems’ Description

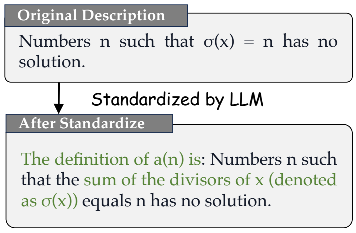

Figure 7: Comparison between original and standardized problem description. The standardized version includes hints and explains the specific meaning of $\sigma(x)$.

B.3 Dataset Statistics

To demonstrate that our benchmark is of cutting-edge level, we have analyzed the distribution of the publication years and the number of references included in the problems of the benchmark as shown in Fig. 8. Additionally, OEIS is a dynamic database. Over the past five years, more than 35,000 sequences in UTMath_Train have been further researched, and over 2,000 new sequences have been added. This ongoing development makes it possible to continuously update UTMath_Train and UTMath, helping to address the challenges posed by data leakage.

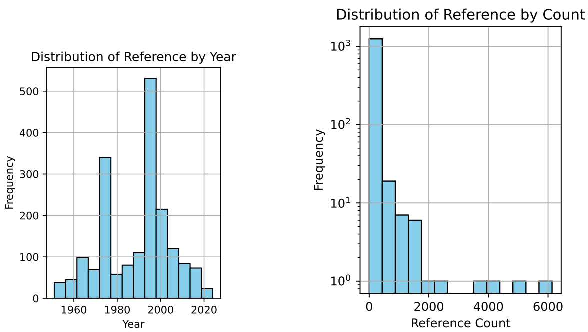

Figure 8: Distribution of references in UTMath.

C Prompts

Prompt 1: the prompt for Program-of-Thoughts

Please reason through the following sequence problem and implement your reasoning using code. You need to follow these requirements:

1. The code must use the Python language.
2. Use the function signature def solution(x: int), and make sure the code part is in markdown format.
3. To ensure the code is runnable, please import any necessary libraries.
4. Provide the reasoning process first.
5. Use the solution with the lowest time complexity.

Question Statements:
{The statement of the question}

Examples:
Solution(1) == a_1
Solution(2) == a_2
Solution(3) == a_3

Prompt 3: the prompt used to determine reasoning or implementation questions

This is a description of a sequence. Please judge whether solving this sequence requires more reasoning or implementation. You need to follow these rules:

1. If the problem statement already has a clear recurrence relation or explicit formula, the question should be considered an implementation question.
2. If the problem does not include a direct calculation formula and requires reasoning to derive it, the question should be considered a reasoning question.
3. Implementation questions usually just require translating the problem requirements directly into code without designing complex algorithms or using advanced data structures.
4. If the question requires more reasoning, answer "reasoning question"; otherwise, answer "implementation question."
5. Your answer should be in italics.

Question Statements:
{The statement of the question}

Prompt 4: the prompt used to determine whether the question belongs to the field of mathematics:

This is information about a sequence from OEIS (The On-Line Encyclopedia of Integer Sequences) and contains four types of information: 'name', 'data', 'comment' and 'formula'. Please use this information and your knowledge to judge the domain to which the sequence belongs. Please follow these rules:

1. Your response should only contain the answer, without any other explanations or examples.
2. Your answer can only be selected from these five options: {'math question', 'physics question', 'chemistry question', 'biology question', 'other question'}
3. Your answer should be in italics

Question Statements:
{The statement of the question}

Data:
{The items of the sequence}

Comment:
{The comment of the question}

Formula:
{The formula of the question}

Prompt 2: the prompt for Reasoning-to-Coding of Thoughts

-------------- Turn 1: < Reasoning Instruction > --------------

Please analyze the following sequence problem and provide a detailed reasoning process for the sequence. You need to follow these requirements:

1. Use the solution with the lowest time complexity.
2. Not to implement the solution.

Question Statements:
{The statement of the question}

Examples:
a(1) == a_1
a(2) == a_2
a(3) == a_3

--------------Turn 2: < Coding Instruction > --------------

Please implement the above solution using Python code, adhering to the following requirements:

1. The code must be written in Python.
2. Use the function signature def solution(x: int), and ensure the code portion is in markdown format.
3. To ensure the code is runnable, please import any necessary libraries.
4. You do not need to provide any explanations or examples, just the implementation code.
5. test contains multiple test cases, each of which will call the solution function.

Examples:
Solution(1) == a_1
Solution(2) == a_2
Solution(3) == a_3

Prompt 5: the prompt used to standardize question statements

I want to create a math problem based on a sequence from the OEIS. The output should be the first n terms of the sequence, but the original problem statement for the sequence may be vague or difficult to understand or might require additional background knowledge. I will provide you with the original problem statement and other information. Please use this information and your knowledge to complete the original problem statement without revealing the sequence's reasoning method or content, and you need to follow these rules:

1. If the original problem statement is sufficient to deduce the entire sequence, no additional information is needed.
2. You should retain as much of the original problem statement as possible.
3. The completed problem statement should include necessary background knowledge.
4. The completed problem statement should maintain the same meaning as the original.
5. The completed problem statement should not contain direct recurrence relations or explicit formulas.
6. The completed problem statement should remain in English.
7. The completed problem statement should begin with "The definition of a(n) is".
8. Your response should only include the completed problem statement without any explanations or examples.

Question Statements:
{The statement of the question}

Other information:
{The information about the question}

Figure 9: The main prompts we used.

D Case Studies

Task_Id: UTMath_948
OEIS_Id: A019567
Main_Category: Special Numbers

Problem Statement:
The definition of a(n) is the least number m for which either $2^m + 1$ or $2^m - 1$ is divisible by $4n + 1$, where n represents the number of pairs of cards in a Mongean shuffle.

Case 1 : a(0) = 1
Case 2 : a(1) = 2
…

Hard Case 1 : a(1882) = 941
Hard Case 2 : a(1883) = 810
…
Hard Case 10 : a(1891) = 88

The Responses of GPT-4o by Program of Thought Methods

To solve this problem, we need to find the smallest integer (m) such that either $2^m + 1$ or $2^m - 1$ is divisible by $(4n + 1)$. This means we need to check for the smallest (m) where:
1. $2^m + 1 \equiv 0 \ (mod\ 4n + 1)$ or
2. $2^m - 1 \equiv 0 \ (mod\ 4n + 1)$

Given the examples:
• For n = 0, $4n + 1 = 1$. The smallest (m) is 1 because $2^1 - 1 = 1$ which is divisible by 1.
• For n = 1, $4n + 1 = 5$. The smallest (m) is 2 because $2^2 - 1 = 3$ and $2^2 + 1 = 5$, and 5 is divisible by 5.
• For n = 2, $4n + 1 = 9$. The smallest (m) is 3 because $2^3 + 1 = 9$, and 9 is divisible by 9.

The approach is to iterate over increasing values of (m) starting from 1, and for each (m) check the divisibility conditions. The first (m) that satisfies either condition is the answer.
Here's the implementation in Python:

```
def solution(x: int) -> int:
    n = x
    divisor = 4 * n + 1

    m = 1
    while True:
        if (2**m + 1) % divisor == 0 or (2**m
- 1) % divisor == 0:
            return m
        m += 1
```

This code iterates over possible values of (m) starting from 1, checking the divisibility conditions for each (m). It returns the first (m) that satisfies either condition, ensuring the solution is found with the lowest time complexity possible given the problem constraints.

The Responses of GPT-4o by Reasoning-To-Coding Thoughts of Thought Methods

------------------------- *Turn 1: < Reasoning >* -------------------------

To analyze the sequence problem, we need to determine the least number (m) such that either $(2^m + 1)$ or $(2^m - 1)$ is divisible by $(4n + 1)$. Let's break down the problem and find a solution with the lowest time complexity.

Understanding the Divisibility Condition:
We need to find the smallest (m) such that either $2^m + 1 \equiv 0 \ (mod\ 4n + 1)$ or $2^m - 1 \equiv 0 \ (mod\ 4n + 1)$.
This means $2^m \equiv -1 \ (mod\ 4n + 1)$ or $2^m \equiv 1 \ (mod\ 4n + 1)$.

Order of 2 Modulo $(4n + 1)$:
- The order of 2 modulo $(\mathbf{4n + 1})$ is the smallest positive integer $(\mathbf{m})$ such that $\mathbf{2^m \equiv 1 \ (mod\ 4n + 1)}$.
- **If $2^m \equiv -1 \ (mod\ 4n + 1)$, then $2^{2m} \equiv 1 \ (mod\ 4n + 1)$ and (m) is half the order of 2 modulo (4n + 1).**

↪Case merging after classification discussion

Efficient Calculation :
- To find the order of 2 modulo (4n + 1), we can use the fact that the order must divide φ(4n + 1) where φ is Euler's totient function.
- Calculate $\boldsymbol{\varphi}(\mathbf{4n + 1})$ and check divisors of this value to find the smallest (**m**) such that $\mathbf{2^m \equiv 1 \ (mod\ 4n + 1)}$ or $\mathbf{2^m \equiv -1 \ (mod\ 4n + 1)}$.

↪Associative Euler function

Steps to Solve :
1. Compute $(\mathbf{4n + 1})$: For a given (n), compute $(4n + 1)$.
2. Calculate $\boldsymbol{\varphi}(\mathbf{4n + 1})$: Use the prime factorization of $(\mathbf{4n + 1})$ to compute $\boldsymbol{\varphi}(\mathbf{4n + 1})$.
3. Find the Order of 2: Check each divisor d of $\varphi(4n + 1)$ to see if $2^d \equiv 1 \ (mod\ 4n + 1)$ or $2^d \equiv -1 \ (mod\ 4n + 1)$.

The smallest such d is the desired (m).

Time Complexity
- Prime Factorization: The time complexity for factorizing $(4n + 1)$ is $O(\sqrt{4n + 1})$
- Checking Divisors: For each divisor d of $\varphi(4n + 1)$, compute $2^d \ mod\ (4n + 1)$ using fast exponentiation, which is $O(\log d)$.
- Overall Complexity: The overall complexity is dominated by the factorization step, making it $O(\sqrt{4n + 1})$.

This approach efficiently finds the least (m) with the lowest time complexity by leveraging number theory concepts such as modular arithmetic and Euler's totient function.

------------------------- *Turn 2: < Coding >* -------------------------

```
def solution(x: int) -> int:
    import math
    def euler_totient(n):
        result = n
        p = 2
        while p * p <= n:
            if n % p == 0:
                while n % p == 0:
                    n //= p
                result -= result // p
            p += 1
        if n > 1:
            result -= result // n
        return result

    def order_of_2_modulo(n):
        phi = euler_totient(n)
        for d in range(1, phi + 1):
            if phi % d == 0:
                if pow(2, d, n) == 1 or pow(2, d, n) == n - 1:
                    return d
        return phi

    mod = 4 * x + 1
    return order_of_2_modulo(mod)
```

Figure 10: GPT-4o solves UTMath_948 by the PoT method, by the RCoT method, respectively. The input prompt is omitted here but can be found in Appendix C. PoT simply performs brute-force solving, while RCoT involves deeper reasoning through Case merging after a classification discussion and the application of Euler's formula, providing a solution with lower time complexity.

Task_Id: UTMath_629
OEIS_Id: A002309
Main_Category: Polynomial and Series Expansions

Problem Statement:
The definition of a(n) is the sum of the fourth powers of the first n odd numbers.

Case 1 : a(1) = 1
Case 2 : a(2) = 82
…

Hard Case 1 : a(63647) = 3342255666620196206703263
Hard Case 2 : a(63648) = 3342518236787942696553888
…
Hard Case 10 : a(64656) = 3344619392264074754875880

The Responses of GPT-4o by Program of Thought Methods

To solve the problem of finding the sum of the fourth powers of the first n odd numbers, we need to follow these steps:

Identify the Sequence of Odd Numbers: The sequence of the first n odd numbers is $1, 3, 5, 7, \dots,$ up to the n-th odd number. The k-th odd number can be expressed as $2k - 1$.

Calculate the Fourth Power: For each odd number in the sequence, calculate its fourth power. The fourth power of a number x is x^4.

Sum the Fourth Powers: Sum up all the fourth powers calculated in the previous step to get the result for a(n).

Optimize the Calculation: Instead of recalculating the odd numbers each time, we can directly compute the sum of the fourth powers using a loop that iterates n times, calculating $(2k - 1)^4$ for each k from 1 to n.

The time complexity of this approach is $O(n)$ because we are iterating through the first n odd numbers and performing a constant amount of work (raising to the fourth power and summing) for each.

Here is the implementation in Python:

```
def solution(x: int) -> int:
    total_sum = 0
    for k in range(1, x + 1):
        odd_number = 2 * k - 1
        total_sum += odd_number ** 4
    return total_sum
```

The Responses of GPT-4o by Reasoning-To-Coding Thoughts of Thought Methods

------------------------- *Turn 1: < Reasoning >* -------------------------

To analyze the sequence problem where a(n) is defined as the sum of the fourth powers of the first n odd numbers, we need to derive a formula or method to compute a(n) efficiently.

Step-by-Step Analysis:
Understanding the Sequence: The sequence involves summing the fourth powers of the first n odd numbers. The first n odd numbers are:

$$1, 3, 5, 7, \dots, (2n - 1)$$

Expressing the Sequence: The nth odd number can be expressed as $(2k - 1)$ for $k = 1$ to n.Therefore:

$$a(n) = (1^4) + (3^4) + (5^4) + \dots + ((2n - 1)^4)$$

General Term: The general term for the kth odd number is $(2k - 1)$.The fourth power of this term is:

$$(2k - 1)^4$$

Summation: We need to compute the sum:

$$a(n) = \sum_{k=1}^{n} (2k - 1)^4$$

Simplifying the Expression: Expand $(2k - 1)^4$ using the binomial theorem:

$$\mathbf{(2k - 1)^4 = (16k\hat{\ }4 - 32k\hat{\ }3 + 24k\hat{\ }2 - 8k + 1)}$$

↪ Associative binomial theorem

Therefore, the sum becomes:

$$a(n) = \sum_{k=1}^{n} (16k^4 - 32k^3 + 24k^2 - 8k + 1)$$

Separate the Summation: Break down the summation into individual components:

$$a(n) = 16 \sum_{k=1}^{n} k^4 - 32 \sum_{k=1}^{n} k^3 + 24 \sum_{k=1}^{n} k^2 - 8 \sum_{k=1}^{n} k + \sum_{k=1}^{n} 1$$

Use Known Summation Formulas: Use the formulas for the sum of powers of integers:

$$\textstyle\sum_{k=1}^{n} k = \frac{n(n+1)}{2}, \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}, \sum_{k=1}^{n} k^3 = (\frac{n(n+1)}{2})^2, \sum_{k=1}^{n} k^4 = \frac{n(n+1)(2n+1)(3n^2+3n+1)}{30}$$

↪Consider the power sum formula

Substitute and Simplify: Substitute these formulas into the expression for a(n) and simplify to get a closed-form expression.
Time Complexity: The derived formula will allow us to compute a(n) in constant time, $O(1)$, since it involves only arithmetic operations on n. By following these steps, we can derive a closed-form expression for a(n) that allows us to compute the sum of the fourth powers of the first n odd numbers efficiently, without iterating through each term. This approach ensures the lowest time complexity, O(1), for computing the sequence.

------------------------- *Turn 2: < Coding >* -------------------------

```
def solution(x: int) -> int:
    # Using the derived formula for the sum of fourth powers of the first n odd numbers
    n = x
    sum_k = n * (n + 1) // 2
    sum_k2 = n * (n + 1) * (2 * n + 1) // 6
    sum_k3 = (n * (n + 1) // 2) ** 2
    sum_k4 = n * (n + 1) * (2 * n + 1) * (3 * n**2 + 3 * n - 1) // 30

    result = 16 * sum_k4 - 32 * sum_k3 + 24 * sum_k2 - 8 * sum_k + n
    return result
```

Figure 11: GPT-4o solves UTMath_629 by the PoT method, by the RCoT method, respectively. The input prompt is omitted here but can be found in Appendix C. PoT only performs brute-force traversal, whereas RCoT engages in deeper reasoning by associating the problem with the binomial theorem and using the power sum formula, yielding a closed-form expression with lower time complexity from O(n) to O(1).