

SBSC: STEP-BY-STEP CODING FOR IMPROVING MATHEMATICAL OLYMPIAD PERFORMANCE

Anonymous authors

Paper under double-blind review

ABSTRACT

We propose Step-by-Step Coding (SBSC): a multi-turn math reasoning framework that enables Large Language Models (LLMs) to generate sequence of programs for solving Olympiad level math problems. At each step/turn, by leveraging the code execution outputs and programs of previous steps, the model generates the next sub-task and the corresponding program to solve it. This way, SBSC, sequentially navigates to reach the final answer. SBSC allows more granular, flexible and precise approach to problem-solving compared to existing methods. Extensive experiments highlight the effectiveness of SBSC in tackling competition and Olympiad-level math problems. For Claude-3.5-Sonnet, we observe SBSC (greedy decoding) surpasses existing state-of-the-art (SOTA) program generation based reasoning strategies by absolute 10.7% on AMC12, 8% on AIME and 12.6% on MathOdyssey. Given SBSC is multi-turn in nature, we also benchmark SBSC’s greedy decoding against self-consistency decoding results of existing SOTA math reasoning strategies and observe performance gain by absolute 6.2% on AMC, 6.7% on AIME and 7.4% on MathOdyssey. Scripts & Data is uploaded at this link.

1 INTRODUCTION

Mathematical reasoning has emerged as a critical benchmark to measure the advanced reasoning and problem-solving abilities of the Large Language Models (LLMs) (Brown et al., 2020; Chowdhery et al., 2022; Achiam et al., 2023; Reid et al., 2024; Anthropic, 2023; OpenAI, June, 2024). This is due to the complex and creative nature of the numerous reasoning steps required to solve the problems.

Chain-of-Thought (Wei et al., 2022) and Scratchpad Nye et al. (2021) prompting strategies helped LLMs to solve a problem using a step-by-step thought process. Program-Aided Language (PAL) (Gao et al., 2022) & Program-Of-Thought (POT) (Chen et al., 2022) introduced problem-solving via program generation where the answer is obtained by executing the generated program. Tool-Integrated Reasoning Agent (ToRA) (Gou et al., 2023) & Mathcoder (Wang et al., 2023a) introduced tool-integrated math problem solving format where model outputs natural language reasoning followed by program generation to solve the entire problem using a single code block and incorporates code-interpreter output for either summarizing the program output to get the final answer and terminate; or re-attempt the problem in the subsequent turn using the same format. For brevity, let’s call ToRA’s defined way of tool-integrated reasoning (TIR) strategy as TIR-ToRA.

The current generation of advanced LLMs such as GPT-4o (Achiam et al., 2023), Claude-3.5-Sonnet (Anthropic, 2023) and Gemini-ultra (Reid et al., 2024) have achieved high scores on elementary GSM8k (Cobbe et al., 2021) high-school level MATH (Hendrycks et al., 2021) by leveraging these reasoning strategies via in-context learning (Brown et al., 2020; Chowdhery et al., 2022). Multiple studies (Yu et al., 2023b; Yue et al., 2023; Toshniwal et al., 2024; Gou et al., 2023; Wang et al., 2023a; Mitra et al., 2024; Beeching et al., 2024; Shao et al., 2024) have tried supervised fine-tuning (SFT) approach to distill these reasoning formats using a propriety models like GPT4 (Achiam et al., 2023). These studies show significant performance improvement over GSM8K and MATH benchmarks.

1.1 MOTIVATION

However, recent math specific competition and Olympiad-level benchmarking on Math Odyssey (Fang et al., 2024), OlymiadBench (He et al., 2024), and the American Invitational Mathematics

054 Examination (AIME) & the American Mathematics Competitions (AMC) (Beeching et al., 2024;
055 DeepSeek-AI et al., 2024; Reid et al., 2024) questions show that the state-of-the-art (SOTA), both
056 generalist and specialist, LLMs continue to struggle with advanced math reasoning. These results
057 highlights the limitation of the existing math prompting techniques. (Tong et al., 2024) highlights the
058 severe bias towards easy problems that exists in the SOTA SFT datasets which originates primarily
059 due to the ineffectiveness of the current prompting strategies in complex math problem-solving.
060 Often, multiple chains are generated via self-consistency decoding (Wang et al., 2022) and majority
061 voting is done to boost the accuracy which is unlike how humans solve problems.

062 Fundamentally, both PAL & TIR-ToRA generate a single program block to solve the entire problem.
063 Additionally, TIR-ToRA framework allows the model to re-attempt the program generation in case
064 of execution error. These approaches show improved performance over COT on elementary & high
065 school level math problems. However, solving olympiad-level math problem requires coming up
066 with complex and creative solution that constitutes of numerous elaborate intermediate steps which
067 eventually leads to the answer. Often, it is not feasible to solve a complex problem entirely using a
068 single program block and as a result, these prompting strategies fail to systematically address each
069 detailed step of the problem-solving process. It tends to overlook specified constraints, edge cases or
070 necessary simplifications, which are often encountered in Olympiad-level problems.

071 1.2 OUR CONTRIBUTION

072 Olympiad level math problem-solving can be viewed as solving/exploring an intermediate sub-
073 task/key-concept in depth; and discovering + solving the next critical sub-task dynamically basis the
074 accumulated knowledge of previous sub-tasks/key-concepts explorations. To this end, we propose
075 Step-by-Step Coding framework (SBSC) which is a multi-turn math reasoning framework that
076 leverages existing programming (Naman Jain, 2024) and in-context learning skills (Brown et al.,
077 2020) of the current generation of LLMs, particularly Claude-3.5-Sonnet (Anthropic, 2023) & GPT-4o
078 (OpenAI, June, 2024). In each turn, it leverages code-interpreter results and knowledge of previous
079 sub-tasks solutions or concept-explorations to define and programmatically solve the next sub-task.
080 Thus it uses code generation as the reasoning strategy to solve an intermediate sub-task or explore
081 an intermediate concept/step. Thus, providing detailed focus to each step of problem solving unlike
082 PAL & TIR-ToRA. SBSC allows an intermediate key-step to be discovered, and be explored and
083 refined (if needed) before being appended to the chain of steps whereas in PAL & TIR-ToRA all the
084 intermediate steps are always stitched together.
085

086 We investigate the performance of SBSC on last 11 years of AIME & AMC-12 questions. We
087 also benchmark on Olympiad-subset of MathOdyssey dataset along with math questions from
088 OlympiadBench. We compare our method (greedy decoding) against greedy-decoding generation
089 of existing reasoning strategies: COT, PAL & TIR-ToRA. We also show SBSC (greedy decoding)
090 effectiveness by benchmarking against self-consistency decoding results of COT, PAL & TIR-ToRA.
091 We conduct extensive ablations to understand the benefits of our approach such as sensitivity to
092 exemplars, topic-wise analysis and measuring improvement in program refinement/debugging ability
093 over TIR-ToRA due to the granular nature of SBSC process.
094

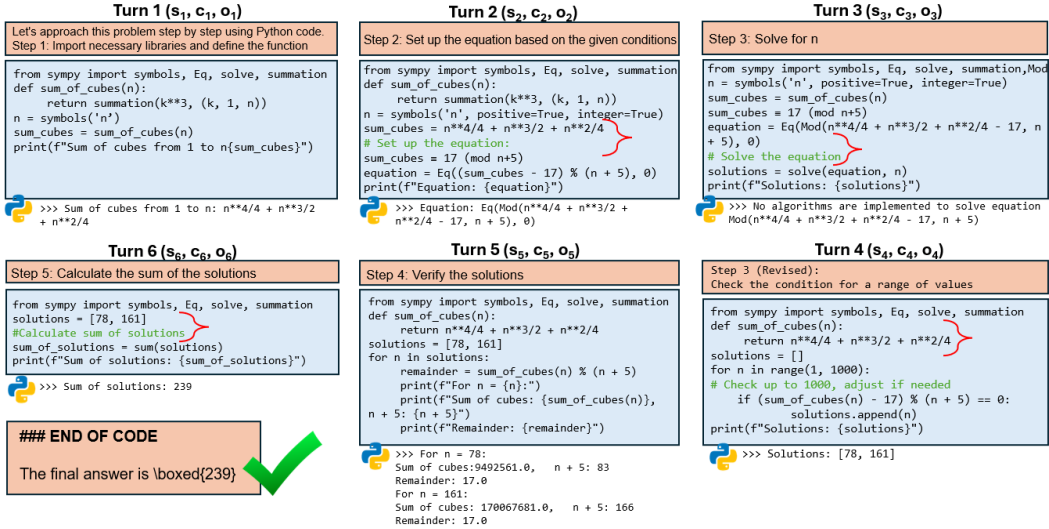
095 2 SBSC: STEP-BY-STEP CODING FRAMEWORK

096 Solving complex math problems, such as competition or Olympiad-level ones, involves creative
097 thinking, applying diverse mathematical knowledge, and dynamically creating subsequent strategies
098 as new insights emerge. One must discover sub-tasks dynamically, rigorously explore intermediate
099 concepts, and carefully handle constraints and edge-cases. Since PAL & TIR-ToRA generates single
100 code block (even during self-correction step; incase of TIR-ToRA) to solve a problem, they lack the
101 flexibility or granularity to emulate this. To address this, we introduce SBSC.
102

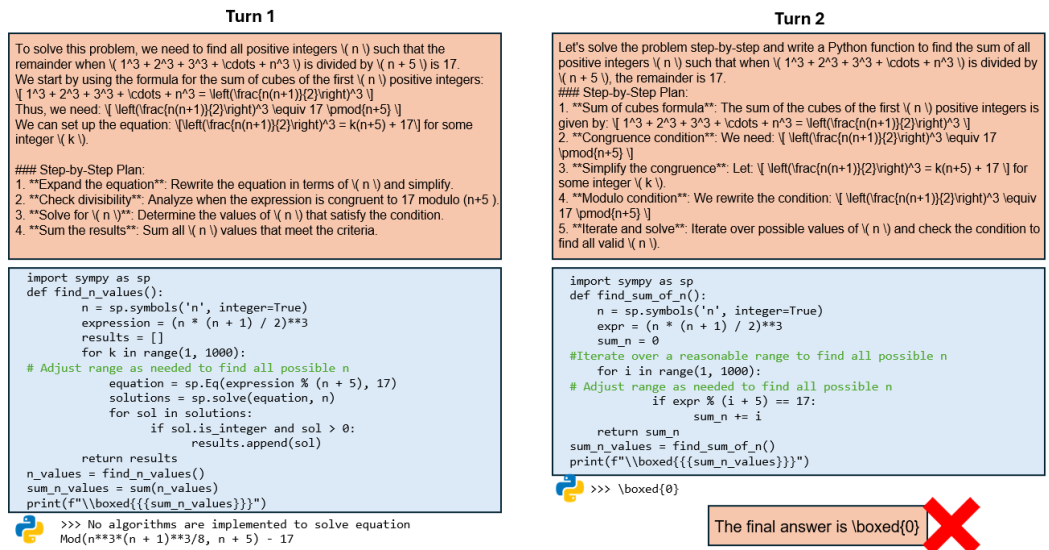
103 SBSC is a multi-turn, code-generation based math reasoning prompting strategy where at each turn:
104 the model generates an intermediate sub-task and corresponding program to solve that sub-task by
105 leveraging the outputs of the previous turns. At the end of each turn, code interpreter is used to
106 execute the program block to generate the solution for the intermediate sub-task. The intermediate
107 sub-task depends on the results of the previous turns and the question. The code snippet for the i^{th}
sub-task directly incorporates the execution results of the previous code snippets by directly defining

them as variables and symbols. This way SBSC makes LLMs generate sequence of programs over multiple turns to solve complex math problems.

Problem: Find the sum of all positive integers n such that when $1^3+2^3+3^3+\dots+n^3$ is divided by $n+5$, the remainder is 17 .



(a) Example multi-turn SBSC response for an AIME problem. Pink boxes denote the sub-task s_i at the i -th step, blue boxes denote the program c_i to solve s_i and $>>>$ denote the corresponding execution output o_i . The red curly brackets indicate reusing outputs from earlier steps.



(b) Example TIR-ToRA response for the same problem, which is not solved correctly. In first turn, it tries to solve the problem at once using a rational and program. It encounters error and in second turn, tries to fix the entire approach and solve again but the solution is incorrect.

Figure 1: Comparison of SBSC and TIR-ToRA frameworks for same AIME problem

Our inference procedure is inspired by ToRA Gou et al. (2023). Solution chain is initialized with the Prompt p containing method instructions followed by exemplars and the current question q . At each step, LLM G first outputs a subtask s_i . If s_i generation ends with stop-word "### END OF CODE", we extract the final answer. Else, it continues to generate program code c_i ending with stop-word "``` `output". We then pass c_i to code interpreter and obtain the execution message or output $o_i \leftarrow E(c_i)$. The solution chain is updated by concatenating it with s_i, c_i, o_i and loop continues

till we get "###END OF CODE". For the i^{th} turn and \oplus denoting concatenation, the sequential process can be generalised as (except for the last turn where just the final answer is generated) :

$$s_i \oplus c_i \sim G(\cdot \mid p \oplus q \oplus (s_1 \oplus c_1 \oplus o_1) \oplus (s_2 \oplus c_2 \oplus o_2) \oplus \dots (s_{i-1} \oplus c_{i-1} \oplus o_{i-1})) \quad (1)$$

Step-wise sequential approach of SBSC ensures that every part of the problem is addressed with exact precision, reducing the risk of errors that might arise from false assumptions or skipped steps. In case the code execution for any step results in an erroneous output, SBSC is better able to rectify that particular step. In depth understanding of SBSC (multiple examples & comparisons) at A.2.

We present example responses from both SBSC and TIR-ToRA for a problem from AIME in figures 1a and 1b respectively. As seen in case of TIR-ToRA, the initial program generated by the model runs into an execution error. At the next turn, it attempts to rectify the error and comes up with a new approach and the corresponding program. This time, the code executes correctly but due to reasoning error the final answer is wrong. On the other hand, we see that SBSC is progressing step-by-step, tackling individual sub-tasks with separate programs and utilising outputs of previous steps. In the third step, it runs into a code execution error but succeeds in rectifying it using a different approach in the very next turn. Further, we observe SBSC checking the validity of the generated solutions in the fourth step before proceeding with the final step and ultimately reaches the correct answer.

2.1 SBSC EXEMPLAR DESIGN

To enable SBSC framework in LLMs, we rely on in-context learning abilities Brown et al. (2020) of LLMs as explored by multiple previous works such as (Chen et al., 2022; Gao et al., 2022; Gou et al., 2023) etc. We also use a system prompt similar to previous works. With respect to exemplar design, to enable program generation, we borrow learning from PAL Gao et al. (2022) & POT (Chen et al., 2022) to have meaningful variable names in the code and using natural language comments within programs (Chen et al., 2022). To enable intermediate tool (code interpreter) usage, we leverage the use of stop words similar to in (Gou et al., 2023). Sample SBSC exemplars can be found at A.5, A.6.

3 EXPERIMENT

3.1 BENCHMARK DATASETS

We mainly use problems from 4 popular math competition datasets for benchmarking our performance: AIME, AMC, MathOdyssey Fang et al. (2024) and OlympiadBench He et al. (2024), covering multiple domains, mainly: Algebra, Combinatorics, Number Theory and Geometry. We use problems of last 11 years from AMC and AIME, obtaining questions and answers (Q&A) in \LaTeX format from the AoPS Wiki website. MathOdyssey Fang et al. (2024), a popular benchmark for LLM math reasoning, consists of problems of varying difficulties. We include the 148 problems belonging to olympiad-level competitions. OlympiadBench is another challenging benchmark for LLMs containing olympiad-level multilingual scientific problems. We select only math related questions, in english language.

3.1.1 DATASET PROCESSING DETAILS:

First, we filter out all questions having reference images associated. Second, we process the questions to have integer type answers if they are already not in that format. All AIME problems have a unique integer answer ranging from 0 to 999, while AMC-12 problems are of Multiple Choice Question(MCQ) format. Similar to NuminaMath (Beeching et al., 2024), we remove all the answer choices from each AMC-12 question and modify the question, wherever necessary, to ensure an integer answer. In case of OlympiadBench and MathOdyssey, we simply modify the question as needed. For this, we prompt GPT-4o to append an additional line at the end of each problem as suitable. Following is an example for demonstration:

Original Question: An urn contains one red ball and one blue ball. A box of extra red and blue balls lies nearby. George performs the following operation four times: he draws a ball from the urn at random and then takes a ball of the same color from the box and returns those two matching balls to the urn. After the four iterations the urn contains six balls. What is the probability that the urn

216 contains three balls of each color?

217 **Answer:** $\frac{1}{5}$

218 **Modified Question:** An urn contains one red ball and one blue ball. A box of extra red and blue
219 balls lies nearby. George performs the following operation four times: he draws a ball from the urn at
220 random and then takes a ball of the same color from the box and returns those two matching balls
221 to the urn. After the four iterations the urn contains six balls. What is the probability that the urn
222 contains three balls of each color? If the answer is represented as a fraction $\frac{m}{n}$ in its simplest terms,
223 what is the value of $m+n$?

224 **Integer Answer:** 6

225 Final test set, contains 330 AIME, 475 AMC-12, 158 MathOdyssey & 504 OlympiadBench problems.

227 3.2 BASELINE & CONFIGURATIONS

229 We benchmark against three prompting/reasoning strategies: COT (Wei et al., 2022), PAL (Gao et al.,
230 2022) TIR-ToRA (Gou et al., 2023). We use `gpt-4o-2024-05-13` and `Claude-3.5-Sonnet`
231 as base LLMs for our experiments. For all datasets and all reasoning frameworks, we use 4-shot
232 setting. Maximum number of turns (n) SBSC is set to 15. For greedy decoding inference, we use
233 `temperature=0` and `max_tokens=1024` and also, we run 3 times and report average. For
234 greedy decoding of TIR-ToRA, we keep $n = 15$ as well (Note: this is because although in TIR-
235 ToRA strategy the model attempts to solve the entire problem in the single turn, in case of execution
236 error or readjustment it tries to re-attempt in subsequent turns). We also benchmark SBSC’s greedy
237 decoding results against self-consistency (SC) Wang et al. (2022) decoding results (majority@7) of
238 COT, PAL & TIR-TORA. We do this primarily for two reasons: First, SBSC takes multiple turns
239 before arriving at the final answer (on average 6-7 turns per problem , Table 3 in Appendix A.1) and
240 Secondly, to benchmark against the reliance of the current existing prompting strategies on majority
241 voting for boosting accuracy. For SC decoding, we use `temperature=0.7` and `top_p=0.9`.
242 Note: we experimentally observe that for $n > 4$, there is insignificant increase in accuracy for
243 TIR-ToRA so we set $n=4$ for TIR-ToRA during SC decoding.

244 Note: PAL (Gao et al., 2022) work also reports a combined approach with Least-to-Most (L2M)
245 prompting strategy (Wang et al., 2022), L2M-PAL that is essentially two stage. We implemented it as
246 per the reported examples in the PAL work. We benchmark it on AMC + AIME dataset. We observe
247 that L2M-PAL at best matches PAL or TIR-ToRA scores. Detailed results available in appendix A.8.
248 Hence for our main results, we stick to PAL & TIR-ToRA along with self-consistency decoding due
249 to resource optimisation and wider adaption of those prompting strategies for math-problem solving.
250 For more discussion on L2M-PAL please check A.8.

251 3.3 PROMPTING/FEW-SHOT EXEMPLARS

253 For both AIME and AMC, we select 90 questions each, drawn from problems of years other than
254 those included in the evaluation datasets. These questions were prompted with COT, PAL, TIR-ToRA
255 and SBSC to generate corresponding solutions in accurate format. For each dataset, we create a subset
256 of 10 problems correctly solved by every method and finally select a combination of 4 exemplars
257 among them. For MathOdyssey as well as Olympiad Bench, we use AIME exemplars as these
258 datasets are of similar difficulty level. We provide the 4 chosen exemplars and system-prompts, used
259 in the main experiments, for different methods in Appendix (A.3, A.4, A.5, A.6) & repository here.

261 4 RESULTS

263 We report the percentage accuracy of all the methods with different base LLMs and across all the
264 benchmarking datasets in Table 1. On AMC dataset, SBSC shows an absolute improvement over
265 TIR-ToRA (greedy decoding) by roughly 11% using Claude-3.5-Sonnet and 7% using GPT-4o. SBSC
266 greedy decoding results outperforms SC decoding results of TIR-TORA by absolute 6% and 4%,
267 for Claude-3.5-Sonnet and GPT-4o respectively. We see similar absolute improvements in accuracy
268 on our AIME dataset too. SBSC outperforms its nearest competitor (PAL) by 8% and 6% with
269 greedy settings and SC settings by 6.7% and 3.7%, for Claude-3.5-Sonnet and GPT-4o respectively.
For MathOdyssey, SBSC improves by as much as 12.6% and 7% over TIR-ToRA while showing

Table 1: Benchmarking SBSC against different math reasoning methods across 3 datasets: We report the average accuracy(in percentage unit) over 3 runs. Best result in each setting is highlighted in **bold** & second best is underlined. Absolute improvement in performance by SBSC over the previous best method in each setting is indicated in subscript.

Method	AMC		AIME		MathOdyssey		Olympiad Bench	
	greedy	maj@7	greedy	maj@7	greedy	maj@7	greedy	maj@7
Claude-3.5-Sonnet								
COT	31.16	35.79	9.09	10.91	11.89	16.89	39.35	42.46
PAL	35.79	36.42	<u>27.48</u>	<u>28.79</u>	27.23	31.01	41.07	44.44
TIR-ToRA	<u>38.59</u>	<u>43.16</u>	24.64	26.67	<u>27.23</u>	<u>32.43</u>	<u>47.69</u>	<u>50.60</u>
SBSC (Ours)	49.33 _{↑10.7}	−↑6.2	35.45 _{↑8}	−↑6.7	39.86 _{↑12.6}	−↑7.4	53.31 _{↑5.6}	−↑2.7
GPT-4o								
COT	35.94	37.47	10.39	12.12	13.51	17.57	41.80	47.22
PAL	36.48	38.11	<u>24.63</u>	<u>26.97</u>	15.74	20.27	41.67	46.43
TIR-ToRA	<u>37.33</u>	<u>40.42</u>	22.42	25.45	<u>19.59</u>	<u>23.64</u>	<u>43.32</u>	49.61
SBSC (Ours)	44.55 _{↑7.2}	−↑4.1	30.7 _{↑6.1}	−↑3.7	26.55 _{↑7}	−↑2.9	48.74 _{↑5.4}	−↓0.87

improvement of 7.4% and 3% over its SC variant, for Claude-3.5-Sonnet & GPT-4o respectively. On OlympiadBench, for GPT-4o, SBSC matches SC results of TIR-ToRA and is better than the second best greedy variant by more than 5%. While for Claude-3.5-Sonnet, SBSC shows an absolute improvement of nearly 6% and 3% over TIR-ToRA in greedy and SC setting respectively. Standard deviation values at A.7.

5 ABLATIONS & ANALYSIS

5.1 SENSITIVITY TO EXEMPLARS

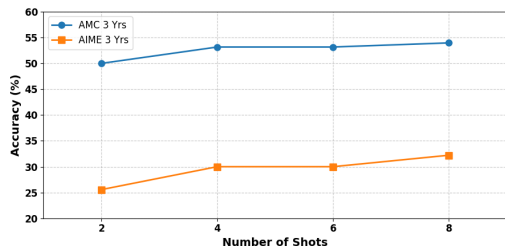


Figure 2: Effect of Number of Exemplars

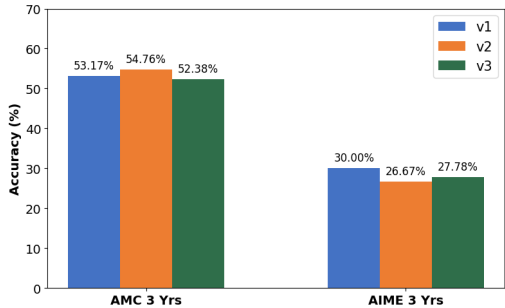


Figure 3: Sensitivity to choice of Exemplars

We study the effect of number/choice of examples in prompting on SBSC’s performance using Claude-3.5-Sonnet on a subset of AIME and AMC data. As shown in Figure 2, we observe a notable increase in performance when increasing the examples from 2 to 4, which then starts to saturate as we further increase the number of examples to 6 and 8. This justifies our decision of using a 4-shot setting. To understand if the choice of exemplars affect the accuracy or not, we conduct a sensitivity analysis. We randomly sample 4 exemplars out of the already created pool of 10 exemplars three times to create 3 variations of 4-shot prompts: v1, v2, and v3. In Figure 3, we can see that the performance remains stable irrespective of the exemplars used.

5.2 SBSC EXEMPLAR TUNING

Natural language comments present within a program have proven to be useful (Gao et al., 2022). So, in each of the SBSC exemplars, we provide suitable comments in natural language within the Python program for each turn to help guide the model.

Table 2: SBSC performance comparison across prompt variations using Claude-3.5-Sonnet

	Full Prompt	Without Comments	Without Line 1
AMC 3 Yrs	67	62	60
AIME 3 Yrs	27	19	16

For few-shot learning, apart from relevant exemplars, the LLM also benefits from a general instruction at the beginning Zheng et al. (2024); Gou et al. (2023); Wang et al. (2023a) that provides a guideline or context about how the model should approach the task, particularly those requiring logical reasoning, multi-step operations, etc. This can be specially useful when the task requires a more nuanced understanding and when the instructions need to be followed rigorously, as is the case with SBSC. Kindly refer to A.5 and A.6 for detailed prompts.

In particular, we highlight one line from the instructions part of the prompt wherein, the model is specifically being instructed to invoke a code rectification step to ensure that the error is not propagated further, leading to a wrong answer. It also ensures the model focuses only on the intermediate step. : **If the executed code snippet returns an error, use it to correct the current step’s code snippet. DO NOT restart solving from Step 1.** 1

In Table 2, we study the importance of these two components in particular: the comments within the code snippets and line 1 mentioned above. Our findings suggest that removal of either of these components lead to a significant decrease in the performance, indicating how each of them are crucial aspects of our exemplar prompts.

5.3 CODE DEBUGGING ABILITY

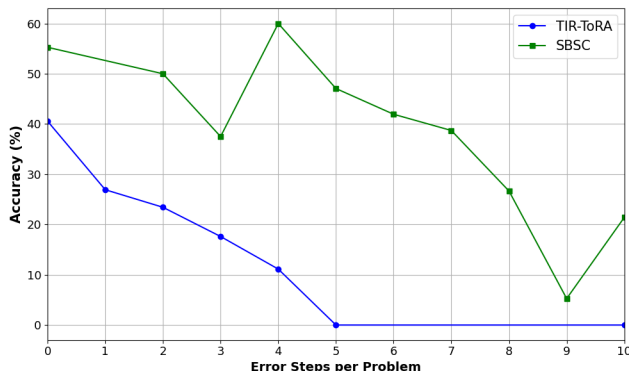


Figure 4: Comparison of Debugging Abilities

We present the superior ability of our method to resolve an error related to code execution. If at any step of the trajectory chain, the program returns an execution error, we consider that to be an error step. We visually represent this, using Claude-3.5-Sonnet responses across AMC, AIME and MathOdyssey datasets in Figure 4, where we see that SBSC is able to recover from even multiple wrong steps and reach the correct final answer quite easily when compared to TIR-ToRA whose performance drops steeply on increasing error steps. This can be attributed to the fact that SBSC, being precise and granular, tackles only a focused part of the problem and finds it easier to correct its mistakes compared to TIR-ToRA which tries to correct the program at the problem level.

5.4 TOPIC-WISE ANALYSIS

We use GPT-4o-mini OpenAI (June, 2024) to classify problems from AIME and AMC, while MathOdyssey and OlympiadBench already contained topic labels. Our test set primarily comprised of: Algebra, Arithmetic, Combinatorics, Number Theory and Geometry. In this study, we benchmark the solutions obtained using Claude-3.5-Sonnet. As can be seen in Figure 5, our method outperforms

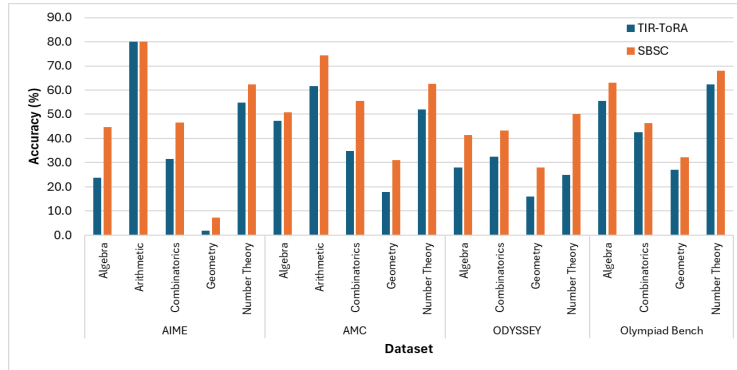


Figure 5: Topic breakdown analysis

TIR-ToRA in all the individual topics and across all the 4 datasets, thereby proving beneficial for all topics. This highlights the generalisation ability of our approach extending to different types and complexities of problems.

5.5 SBSC ACCURACY CORRELATION WITH CODING CAPABILITIES OF LLMs

We study the correlation of code related capabilities of the LLMs with respect to their success with SBSC. Since coding capabilities of a model is pivotal towards successfully following and executing our SBSC approach, we make a comparison involving LLMs with varying coding abilities. Figure 6 shows that the SBSC scores are correlated to the code generation abilities of the corresponding models for all cases that were evaluated on a subset of AIME and AMC data. The code-generation scores were taken from LiveCodeBench (Naman Jain, 2024) benchmark.

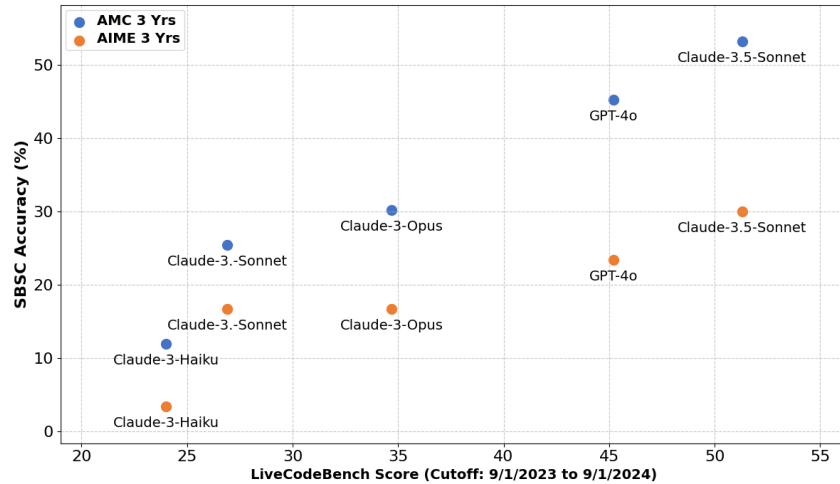


Figure 6: SBSC accuracy correlation with coding ability of LLMs

5.6 SBSC + SELF-CONSISTENCY

Self-consistency (SC) decoding (Wang et al., 2022) has proven to be effective in boosting accuracy via sampling multiple chains and taking a majority voting. We employ SC decoding to assess the upper bound of our approach. For this study, we use `temperature=0.7` and `top_p=0.7`.

We generate 7 chains using Claude-3.5-Sonnet for each problem of last 3 years of AMC and AIME; and consider the majority voted answer as the prediction to be compared against the ground truth. We notice from Figure 7 that the `maj@7` accuracy is higher than that of greedy decoding, following the usual trend with other prompting approaches like COT, PAL, etc.

432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485

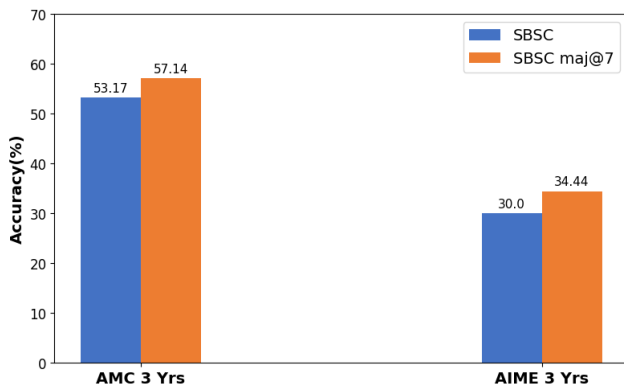


Figure 7: SBSC accuracy correlation with coding ability of LLMs

6 RELATED WORK

In recent times, numerous developments in multiple research directions have taken place to enhance the math ability of the LLMs. One of the major ones has been along the prompting and thinking strategies such as Chain-of-Thought (COT) method (Wei et al., 2022; Kojima et al., 2022) that has shown to evoke multi-step thinking in LLMs before arriving at the answer. These methods struggle with complex and symbolic computations. For this, PAL (Gao et al., 2022) & POT (Chen et al., 2022) suggest making LLMs perform reasoning by writing program and offloading the computations to code interpreter. Another line of research has been around pre-training and supervised fine-tuning (SFT). Multiple studies (Shao et al., 2024; Ying et al., 2024; DeepSeek-AI et al., 2024; Azerbayev et al., 2023; Lewkowycz et al., 2022; Paster et al., 2023; Taylor et al., 2022) have shown pre-training LLMs on high-quality maths tokens results in increased mathematical knowledge and reasoning abilities. Recent approaches (Yu et al., 2023b; Gou et al., 2023; Yue et al., 2023; Wang et al., 2023a; Shao et al., 2024; Toshniwal et al., 2024; Mitra et al., 2024; Beeching et al., 2024; Yin et al., 2024; Tong et al., 2024) have tried query/problem augmentation along with creating synthetic reasoning paths/trajectories using a teacher model like GPT4 (Achiam et al., 2023) for SFT. These methods showed significant improvement in the math reasoning abilities of the model. Also, some studies (Wang et al., 2023b; Yu et al., 2023a; Xi et al., 2024; Chen et al., 2024; Lightman et al., 2023b) provide an alternative to manual annotations for process supervision (Lightman et al., 2023a).

7 CONCLUSION

We introduce SBSC, a multi-turn math reasoning framework that tries to enable LLMs to solve complex math problems. SBSC pursues the solution, step-by-step with each turn dedicated to a step, and arrives at final answer via multiple turns. At each turn, an intermediate sub-task and its corresponding program solution is generated leveraging the execution outputs and solutions of all the previous sub-tasks. We show performance improvements of SBSC over TIR-ToRA, PAL & COT on challenging math problems. We also show that greedy-decoding results of SBSC outperforms self-consistency results of other prompting strategies.

8 FUTURE WORK

Given the detailed, dynamic and flexible step-wise nature of problem-solving along with the fact that its leverage program generation to conclude a key-intermediate step, we believe SBSC reasoning format could be highly useful for guided decoding strategies such as in Outcome-Supervised Value Model (Yu et al., 2023a), AlphaMATH (Chen et al., 2024), Q* framework (Wang et al., 2024). It would be well suited for step-wise preference optimisation for reasoning such as in (Lai et al., 2024). SBSC trajectories could be used also for imitation learning via SFT.

REFERENCES

- 486
487
488 OpenAI Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni
489 Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor
490 Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haim ing Bao, Mo Bavarian, Jeff
491 Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bog-
492 donoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles
493 Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea
494 Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen,
495 Ruby Chen, Jason Chen, Mark Chen, Benjamin Chess, Chester Cho, Casey Chu, Hyung Won
496 Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah
497 Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien
498 Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Sim'on Posada Fish-
499 man, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun
500 Gogineni, Gabriel Goh, Raphael Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray,
501 Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris,
502 Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter
503 Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain,
504 Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie
505 Jonn, Heewoo Jun, Tomer Kaftan, Lukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish
506 Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik
507 Kirchner, Jamie Ryan Kiros, Matthew Knight, Daniel Kokotajlo, Lukasz Kondraciuk, Andrew Kon-
508 drich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan,
509 Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie
510 Lin, Ma teusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini,
511 Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne,
512 Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David
513 Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie
514 Monaco, Evan Morikawa, Daniel P. Mossing, Tong Mu, Mira Murati, Oleg Murk, David M'ely,
515 Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo
516 Noh, Ouyang Long, Cullen O'Keefe, Jakub W. Pachocki, Alex Paino, Joe Palermo, Ashley Pantu-
517 liano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alexandre Passos, Mikhail Pavlov,
518 Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Pondé
519 de Oliveira Pinto, Michael Pokorný, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea
520 Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack W. Rae, Aditya Ramesh,
521 Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick
522 Ryder, Mario D. Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David
523 Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah
524 Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama,
525 Ian Sohl, Benjamin D. Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie
526 Summers, Ilya Sutskever, Jie Tang, Nikolas A. Tezak, Madeleine Thompson, Phil Tillet, Amin
527 Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cer'on
528 Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll L. Wainwright, Justin Jay Wang,
529 Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welin-
530 der, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich,
531 Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah
532 Yoo, Kevin Yu, Qim ing Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang,
533 Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical
534 report. 2023. URL <https://api.semanticscholar.org/CorpusID:257532815>.
- 535
536
537 Anthropic. Introducing claude 3.5, 2023. URL [https://www-cdn.anthropic.com/
538 fed9cc193a14b84131812372d8d5857f8f304c52/Model_Card_Claude_3_
539 Addendum.pdf](https://www-cdn.anthropic.com/fed9cc193a14b84131812372d8d5857f8f304c52/Model_Card_Claude_3_Addendum.pdf).
- 537 Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen Marcus McAleer,
538 Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model
539 for mathematics. *ArXiv*, abs/2310.10631, 2023. URL [https://api.semanticscholar.
org/CorpusID:264172303](https://api.semanticscholar.org/CorpusID:264172303).

- 540 Edward Beeching, Shengyi Costa Huang, Albert Jiang, Jia Li, Benjamin Lipkin, Zihan Qina,
541 Kashif Rasul, Ziju Shen, Roman Soletskyi, and Lewis Tunstall. NuminaMath 7b tir. <https://huggingface.co/AI-MO/NuminaMath-7B-TIR>, 2024.
542
543
- 544 Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal,
545 Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel
546 Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler,
547 Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Ma teusz Litwin, Scott Gray,
548 Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever,
549 and Dario Amodei. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020. URL
550 <https://api.semanticscholar.org/CorpusID:218971783>.
- 551 Guoxin Chen, Minpeng Liao, Chengxi Li, and Kai Fan. Alphamath almost zero: process supervision
552 without process. *ArXiv*, abs/2405.03553, 2024. URL <https://api.semanticscholar.org/CorpusID:269605484>.
553
- 554 Wenhui Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting:
555 Disentangling computation from reasoning for numerical reasoning tasks. *Trans. Mach. Learn. Res.*,
556 2023, 2022. URL <https://api.semanticscholar.org/CorpusID:253801709>.
557
- 558 Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam
559 Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh,
560 Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam M.
561 Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James
562 Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Lev-
563 skaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier García, Vedant Misra, Kevin
564 Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph,
565 Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M.
566 Dai, Thanumalayan Sankaranarayanan Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon
567 Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Díaz,
568 Orhan Firat, Michele Catasta, Jason Wei, Kathleen S. Meier-Hellstern, Douglas Eck, Jeff Dean, Slav
569 Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *ArXiv*, abs/2204.02311,
570 2022. URL <https://api.semanticscholar.org/CorpusID:247951931>.
- 571 Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser,
572 Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John
573 Schulman. Training verifiers to solve math word problems. *ArXiv*, abs/2110.14168, 2021. URL
574 <https://api.semanticscholar.org/CorpusID:239998651>.
- 575 DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu,
576 Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai
577 Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bing-Li Wang,
578 Jun-Mei Song, Deli Chen, Xin Xie, Kang Guan, Yu mei You, Aixin Liu, Qiushi Du, Wenjun Gao,
579 Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan,
580 Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models
581 in code intelligence. *ArXiv*, abs/2406.11931, 2024. URL <https://api.semanticscholar.org/CorpusID:270562723>.
582
- 583 Meng Fang, Xiangpeng Wan, Fei Lu, Fei Xing, and Kai Zou. Mathodyssey: Benchmark-
584 ing mathematical problem-solving skills in large language models using odyssey math data.
585 *ArXiv*, abs/2406.18321, 2024. URL <https://api.semanticscholar.org/CorpusID:270737739>.
586
- 587 Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and
588 Graham Neubig. Pal: Program-aided language models. *ArXiv*, abs/2211.10435, 2022. URL
589 <https://api.semanticscholar.org/CorpusID:253708270>.
590
- 591 Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Minlie Huang, Nan Duan,
592 and Weizhu Chen. Tora: A tool-integrated reasoning agent for mathematical problem solving.
593 *ArXiv*, abs/2309.17452, 2023. URL <https://api.semanticscholar.org/CorpusID:263310365>.

- 594 Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu,
595 Xu Han, Yujie Huang, Yuxiang Zhang, Jie Liu, Lei Qi, Zhiyuan Liu, and Maosong Sun. Olympiad-
596 bench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal
597 scientific problems. *ArXiv*, abs/2402.14008, 2024. URL <https://api.semanticscholar.org/CorpusID:267770504>.
- 599 Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Xi-
600 aodong Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset.
601 *ArXiv*, abs/2103.03874, 2021. URL <https://api.semanticscholar.org/CorpusID:232134851>.
- 603 Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large
604 language models are zero-shot reasoners. *ArXiv*, abs/2205.11916, 2022. URL <https://api.semanticscholar.org/CorpusID:249017743>.
- 607 Xin Lai, Zhuotao Tian, Yukang Chen, Senqiao Yang, Xiangru Peng, and Jiaya Jia. Step-dpo: Step-
608 wise preference optimization for long-chain reasoning of llms, 2024. URL <https://arxiv.org/abs/2406.18629>.
- 611 Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski,
612 Vinay Venkatesh Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo,
613 Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reason-
614 ing problems with language models. *ArXiv*, abs/2206.14858, 2022. URL <https://api.semanticscholar.org/CorpusID:250144408>.
- 616 Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan
617 Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let’s verify step by step, 2023a. URL
618 <https://arxiv.org/abs/2305.20050>.
- 619 Hunter Lightman, Vineet Kosaraju, Yura Burda, Harrison Edwards, Bowen Baker, Teddy Lee,
620 Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let’s verify step by step.
621 *ArXiv*, abs/2305.20050, 2023b. URL <https://api.semanticscholar.org/CorpusID:258987659>.
- 623 Arindam Mitra, Hamed Khanpour, Corby Rosset, and Ahmed Awadallah. Orca-math: Unlocking
624 the potential of slms in grade school math. *ArXiv*, abs/2402.14830, 2024. URL <https://api.semanticscholar.org/CorpusID:267897618>.
- 627 Alex Gu Wen-Ding Li Fanjia Yan Tianjun Zhang Sida Wang Armando Solar-Lezama Koushik Sen
628 Ion Stoica Naman Jain, King Han. Livecodebench: Holistic and contamination free evaluation of
629 large language models for code. *arXiv preprint*, 2024.
- 630 Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David
631 Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and
632 Augustus Odena. Show your work: Scratchpads for intermediate computation with language
633 models, 2021. URL <https://arxiv.org/abs/2112.00114>.
- 634 OpenAI. "hello gpt-4o.", June, 2024. URL <https://openai.com/index/hello-gpt-4o/>.
- 635
636
637
638 Keiran Paster, Marco Dos Santos, Zhangir Azerbayev, and Jimmy Ba. Openwebmath: An open
639 dataset of high-quality mathematical web text. *ArXiv*, abs/2310.06786, 2023. URL <https://api.semanticscholar.org/CorpusID:263829563>.
- 640
641 Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, Jean-
642 Baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis
643 Antonoglou, Rohan Anil, Sebastian Borgeaud, Andrew M. Dai, Katie Millican, Ethan Dyer, Mia
644 Glaese, Thibault Sottiaux, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, James
645 Molloy, Jilin Chen, Michael Isard, Paul Barham, Tom Hennigan, Ross McIlroy, Melvin Johnson,
646 Johan Schalkwyk, Eli Collins, Eliza Rutherford, Erica Moreira, Kareem W. Ayoub, Megha Goel,
647 Clemens Meyer, Gregory Thornton, Zhen Yang, Henryk Michalewski, Zaheer Abbas, Nathan
Schucher, Ankesh Anand, Richard Ives, James Keeling, Karel Lenc, Salem Haykal, Siamak

648 Shakeri, Pranav Shyam, Aakanksha Chowdhery, Roman Ring, Stephen Spencer, Eren Sezener,
649 Luke Vilnis, Oscar Chang, Nobuyuki Morioka, George Tucker, Ce Zheng, Oliver Woodman, Nithya
650 Attaluri, Tomas Kocisky, Evgenii Eltyshev, Xi Chen, Timothy Chung, Vittorio Selo, Siddhartha
651 Brahma, Petko Georgiev, Ambrose Slone, Zhenkai Zhu, James Lottes, Siyuan Qiao, Ben Caine,
652 Sebastian Riedel, Alex Tomala, Martin Chadwick, J Christopher Love, Peter Choy, Sid Mittal,
653 Neil Houlsby, Yunhao Tang, Matthew Lamm, Libin Bai, Qiao Zhang, Luheng He, Yong Cheng,
654 Peter Humphreys, Yujia Li, Sergey Brin, Albin Cassirer, Ying-Qi Miao, Lukas Zilka, Taylor Tobin,
655 Kelvin Xu, Lev Proleev, Daniel Sohn, Alberto Magni, Lisa Anne Hendricks, Isabel Gao, Santiago
656 Ontanon, Oskar Bunyan, Nathan Byrd, Abhanshu Sharma, Biao Zhang, Mario Pinto, Rishika
657 Sinha, Harsh Mehta, Dawei Jia, Sergi Caelles, Albert Webson, Alex Morris, Becca Roelofs, Yifan
658 Ding, Robin Strudel, Xuehan Xiong, Marvin Ritter, Mostafa Dehghani, Rahma Chaabouni, Abhijit
659 Karmarkar, Guangda Lai, Fabian Mentzer, Bibo Xu, YaGuang Li, Yujing Zhang, Tom Le Paine,
660 Alex Goldin, Behnam Neyshabur, Kate Baumli, Anselm Levskaya, Michael Laskin, Wenhao
661 Jia, Jack W. Rae, Kefan Xiao, Antoine He, Skye Giordano, Lakshman Yagati, Jean-Baptiste
662 Lepiau, Paul Natsev, Sanjay Ganapathy, Fangyu Liu, Danilo Martins, Nanxin Chen, Yunhan Xu,
663 Megan Barnes, Rhys May, Arpi Vezar, Junhyuk Oh, Ken Franko, Sophie Bridgers, Ruizhe Zhao,
664 Boxi Wu, Basil Mustafa, Sean Sechrist, Emilio Parisotto, Thanumalayan Sankaranarayanan Pillai,
665 Chris Larkin, Chenjie Gu, Christina Sorokin, Maxim Krikun, Alexey Guseynov, Jessica Landon,
666 Romina Datta, Alexander Pritzel, Phoebe Thacker, Fan Yang, Kevin Hui, A.E. Hauth, Chih-Kuan
667 Yeh, David Barker, Justin Mao-Jones, Sophia Austin, Hannah Sheahan, Parker Schuh, James
668 Svensson, Rohan Jain, Vinay Venkatesh Ramasesh, Anton Briukhov, Da-Woon Chung, Tamara
669 von Glehn, Christina Butterfield, Priya Jhakra, Matt Wiethoff, Justin Frye, Jordan Grimstad,
670 Beer Changpinyo, Charline Le Lan, Anna Bortsova, Yonghui Wu, Paul Voigtlaender, Tara N.
671 Sainath, Charlotte Smith, Will Hawkins, Kris Cao, James Besley, Srivatsan Srinivasan, Mark
672 Omernick, Colin Gaffney, Gabriela de Castro Surita, Ryan Burnell, Bogdan Damoc, Junwhan
673 Ahn, Andrew Brock, Mantas Pajarskas, Anastasia Petrushkina, Seb Noury, Lorenzo Blanco, Kevin
674 Swersky, Arun Ahuja, Thi Avrahami, Vedant Misra, Raoul de Liedekerke, Mariko Iinuma, Alex
675 Polozov, Sarah York, George van den Driessche, Paul Michel, Justin Chiu, Rory Blevins, Zach
676 Gleicher, Adria Recasens, Alban Rustemi, Elena Gribovskaia, Aurko Roy, Wiktor Gworek,
677 S’ebastien M. R. Arnold, Lisa Lee, James Lee-Thorp, Marcello Maggioni, Enrique Piqueras,
678 Kartikeya Badola, Sharad Vikram, Lucas Gonzalez, Anirudh Baddepudi, Evan Senter, Jacob
679 Devlin, James Qin, Michael Azzam, Maja Trebacz, Martin Polacek, Kashyap Krishnakumar,
680 Shuo yiin Chang, Matthew Tung, Ivo Penchev, Rishabh Joshi, Kate Olszewska, Carrie Muir,
681 Mateo Wirth, Ale Jakse Hartman, Joshua Newlan, Sheleem Kashem, Vijay Bolina, Elahe Dabir,
682 Joost R. van Amersfoort, Zafarali Ahmed, James Cobon-Kerr, Aishwarya B Kamath, Arnar Mar
683 Hrafnkelsson, Le Hou, Ian Mackinnon, Alexandre Frechette, Eric Noland, Xiance Si, Emanuel
684 Taropa, Dong Li, Phil Crone, Anmol Gulati, S’ebastien Cevey, Jonas Adler, Ada Ma, David Silver,
685 Simon Tokumine, Richard Powell, Stephan Lee, Michael B. Chang, Samer Hassan, Diana Mincu,
686 Antoine Yang, Nir Levine, Jenny Brennan, Mingqiu Wang, Sarah Hodkinson, Jeffrey Zhao, Josh
687 Lipschultz, Aedan Pope, Michael B. Chang, Cheng Li, Laurent El Shafey, Michela Paganini,
688 Sholto Douglas, Bernd Bohnet, Fabio Pardo, Seth Odoom, Mihaela Rosca, Cicero Nogueira dos
689 Santos, Kedar Soparkar, Arthur Guez, Tom Hudson, Steven Hansen, Chulayuth Asawaroengchai,
690 Ravichandra Addanki, Tianhe Yu, Wojciech Stokowiec, Mina Khan, Justin Gilmer, Jaehoon
691 Lee, Carrie Grimes Bostock, Keran Rong, Jonathan Caton, Pedram Pejman, Filip Pavetic, Geoff
692 Brown, Vivek Sharma, Mario Luvci’c, Rajkumar Samuel, Josip Djolonga, Amol Mandhane,
693 Lars Lowe Sjosund, Elena Buchatskaya, Elspeth White, Natalie Clay, Jiepu Jiang, Hyeontaek
694 Lim, Ross Hemsley, Jane Labanowski, Nicola De Cao, David Steiner, Sayed Hadi Hashemi, Jacob
695 Austin, Anita Gergely, Tim Blyth, Joe Stanton, Kaushik Shivakumar, Aditya Siddhant, Anders
696 Andreassen, Carlos L. Araya, Nikhil Sethi, Rakesh Shivanna, Steven Hand, Ankur Bapna, Ali
697 Khodaei, Antoine Miech, Garrett Tanzer, Andy Swing, Shantanu Thakoor, Zhufeng Pan, Zachary
698 Nado, Stephanie Winkler, Dian Yu, Mohammad Saleh, Lorenzo Maggiore, Iain Barr, Minh
699 Giang, Thais Kagohara, Ivo Danihelka, Amit Marathe, Vladimir Feinberg, Mohamed Elhawaty,
700 Nimesh Ghelani, Dan Horgan, Helen Miller, Lexi Walker, Richard Tanburn, Richard Tanburn, Sina
701 Samangoeei, Fred Alcober, Axel Stjerngren, Paul Komarek, Katerina Tsihlias, Anudhyan Boral,
Ramona Comanescu, Jeremy Chen, Ruibo Liu, Dawn Bloxwich, Charlie Chen, Yanhua Sun,
Fangxiaoyu Feng, Matthew Mauger, Xerxes Dotiwalla, Vincent Hellendoorn, Michael Sharman,
Ivy Zheng, Krishna Haridasan, Gabriel Barth-Maron, Craig Swanson, Dominika Rogozi’nska,
Alek Andreev, Paul Kishan Rubenstein, Ruoxin Sang, Dan Hurt, Gamaleldin Elsayed, Ren shen

- 702 Wang, Dave Lacey, Anastasija Ilić, Yao Zhao, Woohyun Han, Lora Aroyo, Chimezie Iwuanyanwu,
703 Vitaly Nikolaev, Balaji Lakshminarayanan, Sadegh Jazayeri, Raphael Lopez Kaufman, Mani
704 Varadarajan, Chetan Tekur, Doug Fritz, Misha Khalman, David Reitter, Kingshuk Dasgupta,
705 Shourya Sarcar, T. Ornduff, Javier Snaider, Fantine Huot, Johnson Jia, Rupert Kemp, Nejc Trdin,
706 Anitha Vijayakumar, Lucy Kim, Christof Angermueller, Li Lao, Tianqi Liu, Haibin Zhang, David
707 Engel, Somer Greene, Anais White, Jessica Austin, Lilly Taylor, Shereen Ashraf, Dangyi Liu,
708 Maria Georgaki, Irene Cai, Yana Kulizhskaya, Sonam Goenka, Brennan Saeta, Kiran Vodrahalli,
709 Christian Frank, Dario de Cesare, Brona Robenek, Harry Richardson, Mahmoud Alnahlawi,
710 Christopher Yew, Priya Ponnappalli, Marco Tagliasacchi, Alex Korchemniy, Yelin Kim, Dinghua
711 Li, Bill Rosgen, Kyle Levin, Jeremy Wiesner, Praseem Banzal, Praveen Srinivasan, Hongkun Yu,
712 cCauglar Unlu, David Reid, Zora Tung, Daniel F. Finchelstein, Ravin Kumar, Andre Elisseeff,
713 Jin Huang, Ming Zhang, Rui Zhu, Ricardo Aguilar, Mai Gim'enez, Jiawei Xia, Olivier Dousse,
714 Willi Gierke, Soheil Hassas Yeganeh, Damion Yates, Komal Jalan, Lu Li, Eri Latorre-Chimoto,
715 Duc Dung Nguyen, Ken Durden, Praveen Kallakuri, Yaxin Liu, Matthew Johnson, Tomy Tsai, Alice
716 Talbert, Jasmine Liu, Alexander Neitz, Chen Elkind, Marco Selvi, Mimi Jasarevic, Livio Baldini
717 Soares, Albert Cui, Pidong Wang, Alek Wenjiao Wang, Xinyu Ye, Krystal Kallarackal, Lucia Loher,
718 Hoi Lam, Josef Broder, Daniel Niels Holtmann-Rice, Nina Martin, Bramandia Ramadhana, Daniel
719 Toyama, Mrinal Shukla, Sujoy Basu, Abhi Mohan, Nicholas Fernando, Noah Fiedel, Kim Paterson,
720 Hui Li, Ankush Garg, Jane Park, Donghyun Choi, Diane Wu, Sankalp Singh, Zhishuai Zhang, Amir
721 Globerson, Lily Yu, John Carpenter, Félix de Chaumont Quitry, Carey Radebaugh, Chu-Cheng Lin,
722 Alex Tudor, Prakash Shroff, Drew Garmon, Dayou Du, Neera Vats, Han Lu, Shariq Iqbal, Alexey
723 Yakubovich, Nilesh Tripuraneni, James Manyika, Haroon Qureshi, Nan Hua, Christel Ngani,
724 Maria Abi Raad, Hannah Forbes, Anna Bulanova, Jeff Stanway, Mukund Sundararajan, Victor
725 Ungureanu, Colton Bishop, Yunjie Li, Balaji Venkatraman, Bo Li, Chloe Thornton, Salvatore
726 Scellato, Nishesh Gupta, Yicheng Wang, Ian Tenney, Xihui Wu, Ashish Shenoy, Gabriel Carvajal,
727 Diana Gage Wright, Ben Bariach, Zhuyun Xiao, Peter Hawkins, Sid Dalmia, Clément Farabet,
728 Pedro Valenzuela, Quan Yuan, Christopher A. Welty, Ananth Agarwal, Mianna Chen, Wooyeol
729 Kim, Brice Hulse, Nandita Dukkupati, Adam Paszke, Andrew Bolt, Elnaz Davoodi, Kiam Choo,
730 Jennifer Beattie, Jennifer Prendki, Harsha Vashisht, Rebeca Santamaria-Fernandez, Luis C. Cobo,
731 Jarek Wilkiewicz, David Madras, Ali Elqursh, Grant Uy, Kevin Ramirez, Matt Harvey, Tyler
732 Liechty, Heiga Zen, Jeff Seibert, Clara Huiyi Hu, A. Ya. Khorlin, Maigo Le, Asaf Aharoni, Megan
733 Li, Lily Wang, Sandeep Kumar, Alejandro Lince, Norman Casagrande, Jay Hoover, Dalia El
734 Badawy, David Soergel, Denis Vnukov, Matt Miecniowski, *Jiimsa*, Anna Koop, Praveen Kumar,
735 Thibault Sellam, Daniel Vlasic, Samira Daruki, Nir Shabat, John Zhang, Guolong Su, Kalpesh
736 Krishna, Jiageng Zhang, Jeremiah Liu, Yi Sun, Evan Palmer, Alireza Ghaffarkhah, Xi Xiong, Victor
737 Cotruta, Michael Fink, Lucas Dixon, Ashwin Sreevatsa, Adrian Goedeckemeyer, Alek Dimitriev,
738 Mohsen Jafari, Remi Crocker, Nicholas Fitzgerald, Aviral Kumar, Sanjay Ghemawat, Ivan Philips,
739 Frederick Liu, Yannie Liang, Rachel Sterneck, Alena Repina, Marcus Wu, Laura Knight, Marin
740 Georgiev, Hyo Lee, Harry Askham, Abhishek Chakladar, Annie Louis, Carl Crous, Hardie Cate,
741 Dessie Petrova, Michael Quinn, Denese Owusu-Afriyie, Achintya Singhal, Nan Wei, Solomon
742 Kim, Damien Vincent, Milad Nasr, Christopher A. Choquette-Choo, Reiko Tojo, Shawn Lu,
743 Diego de Las Casas, Yuchung Cheng, Tolga Bolukbasi, Katherine Lee, Saaber Fatehi, Rajagopal
744 Ananthanarayanan, Miteyan Patel, Charbel El Kaed, Jing Li, Jakub Sygnowski, Shreyas Rammohan
745 Belle, Zhe Chen, Jaelyn Konzelmann, Siim Poder, Roopal Garg, Vinod Koverkathu, Adam Brown,
746 Chris Dyer, Rosanne Liu, Azade Nova, Jun Xu, Junwen Bai, Slav Petrov, Demis Hassabis, Koray
747 Kavukcuoglu, Jeffrey Dean, Oriol Vinyals, and Alexandra Chronopoulou. Gemini 1.5: Unlocking
748 multimodal understanding across millions of tokens of context. *ArXiv*, abs/2403.05530, 2024.
749 URL <https://api.semanticscholar.org/CorpusID:268297180>.
750
751
752
753 Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Jun-Mei Song, Mingchuan Zhang, Y. K. Li,
754 Yu Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open
755 language models. *ArXiv*, abs/2402.03300, 2024. URL <https://api.semanticscholar.org/CorpusID:267412607>.
- Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony S. Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. Galactica: A large language model for science. *ArXiv*, abs/2211.09085, 2022. URL <https://api.semanticscholar.org/CorpusID:253553203>.

- 756 Yuxuan Tong, Xiwen Zhang, Rui Wang, Rui Min Wu, and Junxian He. Dart-math: Difficulty-
757 aware rejection tuning for mathematical problem-solving. *ArXiv*, abs/2407.13690, 2024. URL <https://api.semanticscholar.org/CorpusID:271270574>.
758
- 759 Shubham Toshniwal, Ivan Moshkov, Sean Narenthiran, Daria Gitman, Fei Jia, and Igor Gitman.
760 Openmathinstruct-1: A 1.8 million math instruction tuning dataset. *ArXiv*, abs/2402.10176, 2024.
761 URL <https://api.semanticscholar.org/CorpusID:267681752>.
762
- 763 Chaojie Wang, Yanchen Deng, Zhiyi Lyu, Liang Zeng, Jujie He, Shuicheng Yan, and Bo An.
764 Q*: Improving multi-step reasoning for llms with deliberative planning, 2024. URL <https://arxiv.org/abs/2406.14283>.
765
- 766 Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang,
767 Linqi Song, Mingjie Zhan, and Hongsheng Li. Mathcoder: Seamless code integration in
768 llms for enhanced mathematical reasoning. *ArXiv*, abs/2310.03731, 2023a. URL <https://api.semanticscholar.org/CorpusID:263671510>.
769
- 770 Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Y.Wu, and Zhi-
771 fang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations.
772 *ArXiv*, abs/2312.08935, 2023b. URL <https://api.semanticscholar.org/CorpusID:266209760>.
773
- 774 Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Huai hsin Chi, and Denny Zhou. Self-
775 consistency improves chain of thought reasoning in language models. *ArXiv*, abs/2203.11171,
776 2022. URL <https://api.semanticscholar.org/CorpusID:247595263>.
777
- 778 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Huai hsin Chi, F. Xia, Quoc
779 Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models.
780 *ArXiv*, abs/2201.11903, 2022. URL <https://api.semanticscholar.org/CorpusID:246411621>.
781
- 782 Zhiheng Xi, Wenxiang Chen, Boyang Hong, Senjie Jin, Rui Zheng, Wei He, Yiwen Ding, Shichun
783 Liu, Xin Guo, Junzhe Wang, Honglin Guo, Wei Shen, Xiaoran Fan, Yuhao Zhou, Shihan Dou, Xiao
784 Wang, Xinbo Zhang, Peng Sun, Tao Gui, Qi Zhang, and Xuanjing Huang. Training large language
785 models for reasoning through reverse curriculum reinforcement learning. *ArXiv*, abs/2402.05808,
786 2024. URL <https://api.semanticscholar.org/CorpusID:267547500>.
787
- 788 Shuo Yin, Weihao You, Zhilong Ji, Guoqiang Zhong, and Jinfeng Bai. Mumath-code: Combining tool-
789 use large language models with multi-perspective data augmentation for mathematical reasoning.
790 *ArXiv*, abs/2405.07551, 2024. URL <https://api.semanticscholar.org/CorpusID:269756851>.
791
- 792 Huaiyuan Ying, Shuo Zhang, Linyang Li, Zhejian Zhou, Yunfan Shao, Zhaoye Fei, Yichuan Ma,
793 Jiawei Hong, Kuikun Liu, Ziyi Wang, Yudong Wang, Zijian Wu, Shuaibin Li, Fengzhe Zhou,
794 Hongwei Liu, Songyang Zhang, Wenwei Zhang, Hang Yan, Xipeng Qiu, Jiayu Wang, Kai Chen,
795 and Dahua Lin. Internlm-math: Open math large language models toward verifiable reasoning.
796 *ArXiv*, abs/2402.06332, 2024. URL <https://api.semanticscholar.org/CorpusID:267617098>.
797
- 798 Fei Yu, Anningzhe Gao, and Benyou Wang. Ovm, outcome-supervised value models for planning in
799 mathematical reasoning. In *NAACL-HLT*, 2023a. URL <https://api.semanticscholar.org/CorpusID:265221057>.
800
- 801 Long Long Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T. Kwok,
802 Zheng Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical
803 questions for large language models. *ArXiv*, abs/2309.12284, 2023b. URL <https://api.semanticscholar.org/CorpusID:262084051>.
804
- 805 Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhao
806 Chen. Mammoth: Building math generalist models through hybrid instruction tuning.
807 *ArXiv*, abs/2309.05653, 2023. URL <https://api.semanticscholar.org/CorpusID:261696697>.
808
- 809

Huaixiu Steven Zheng, Swaroop Mishra, Xinyun Chen, Heng-Tze Cheng, Ed H. Chi, Quoc V Le, and Denny Zhou. Take a step back: Evoking reasoning via abstraction in large language models, 2024. URL <https://arxiv.org/abs/2310.06117>.

Denny Zhou, Nathanael Scharli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed Huai hsin Chi. Least-to-most prompting enables complex reasoning in large language models. *ArXiv*, abs/2205.10625, 2022. URL <https://api.semanticscholar.org/CorpusID:248986239>.

A APPENDIX

A.1 NUMBER OF STEPS IN SBSC

In Table 3, we present the number of turns taken per question by SBSC responses obtained using Claude-3.5-Sonnet across the different datasets.

Table 3: Table showing number of turns/steps used by SBSC

Number of turns or steps	AMC	AIME	MathOdyssey
2	21	12	8
3	57	19	17
4	101	47	19
5	79	51	21
6	63	43	28
7	41	43	14
8	42	31	10
9	12	18	8
others	59	66	23
Average turns or steps/Problem	6.0	6.9	6.4

A.2 UNDERSTANDING SBSC IN DETAIL

In this section, we demonstrate some scenarios where SBSC has been successful while TIR-ToRA has failed, with the help of some example questions and investigating the responses obtained from the two models.

Let’s consider the question in Example 1, involving a geometric progression of numbers written in logarithmic form, which TIR-ToRA gets wrong. The method uses a binary search technique, which is not very precise when dealing with exact values required for mathematical problems, especially when fractions are involved. The solution uses a function to check whether the logarithms form a geometric progression which introduces additional complexity and potential inaccuracies because it involves comparing ratios that may not be exactly equal due to floating-point arithmetic. Also, this single-turn method tends to overlook specified constraints or necessary simplifications, which are often encountered in Olympiad level problems and instead makes false assumptions.

The question in Example 2 is an example scenario where TIR-ToRA fails because it makes an incorrect assumption. It misinterprets the Lipschitz condition and incorrectly makes a simpler assumption that the difference $f(800) - f(400)$ is equal to the maximum possible difference, which is 200. While the magnitude of the difference is bounded by 200, it does not mean that the actual difference will always be 200. Iterative solutions, as are often the only way out in single program based solutions, can sometimes lead to infinite loops, especially in cases where the stopping condition is not clearly defined or understood by the LLM.

As can be seen in Example 3, the single code is unable to take advantage of the factorization of 20^{20} , which is key to solving the problem efficiently and instead iterates over a very large range of potential values for m , leading to inefficiency. The upper bound 2020 is extremely large and the sheer number of iterations causes a timeout.

Example 4 presents a scenario where TIR-ToRA makes up an assumption about the problem and

864 writes the code for terminating a loop accordingly, which leads to a timeout error, as the incorrect
 865 assumption leads to an infinite loop. It lacks intermediate checks that would provide insights into
 866 whether the sequence terms are of the form $\frac{t}{t+1}$, which is crucial for solving the problem and would
 867 have enabled it to chalk out the termination conditions suitably.

868 On the other hand, our Step-By-Step Coding method enforces a decomposition of the problem into
 869 smaller sub-task. Each sub-task is tackled independently by the LLM, which generates code to solve
 870 it and then uses the resulting output to suitably proceed to the next sub-task and this process continues
 871 till the final answer is reached. Such an approach ensures that every part of the problem is addressed
 872 with exact precision, reducing the risk of errors that might arise from skipped steps. Dividing the
 873 problem into multiple sub-tasks also allows it to make necessary simplifications that would make the
 874 future sub-tasks, and hence the entire problem, easier to solve.

875 Going back to the problem in Example 1, SBSC starts by defining the logarithms and setting up the
 876 equations based on the geometric progression condition. It then simplifies the equations to reduce
 877 them to a more manageable form, eliminating unnecessary complexity and allowing straightforward
 878 solving. Throughout the problem, it uses precise mathematical formulations of the problem, ensuring
 879 the solution is accurate. Since this method isn't trying to solve the entire problem at one go, it doesn't
 880 need to make any assumptions to simplify the problem statement.

881 For the question in Example 2, it correctly interprets the problem, keeps applying the given Lipschitz
 882 condition as it solves each sub-task and finds the correct maximum possible value of $f(f(800)) -$
 883 $f(f(400))$. By systematically checking for constraints and edge cases at each stage, our method
 884 guarantees that solutions are not only accurate but also valid under all possible conditions. This is
 885 particularly crucial for problems with intricate conditions or multiple solution paths. Additionally,
 886 our process explicitly defines loop conditions to ensure that termination criteria are clear, allowing
 for correction if issues like infinite loops arise.

887 For the problem in Example 3, the first sub-task correctly identified by SBSC is to factorize the large
 888 number 20^{20} into smaller powers, paving the path towards an efficient solution. In the subsequent
 889 steps, it takes care of constraints like ensuring that m and n are integers and avoids over-counting by
 890 properly iterating over valid powers, giving us the correct answer.

891 Likewise, in Example 4, we see that our method generates each term of the sequence one at a time,
 892 allowing for intermediate checks to confirm whether each term is of the desired form $\frac{t}{t+1}$. Thus,
 893 it accurately identifies the terms that meet the required condition and ensures that only the correct
 894 indices are considered. This avoids the infinite loop risk and ensures the correctness of each step
 895 before moving on. We also notice that the model runs into an error at the first sub-task itself but based
 896 on the corresponding code output, it is able to take an alternate approach and successfully solves the
 897 sub-task, moving onto later ones. This example also highlights our superior code debugging ability,
 as discussed earlier.

898 **Example Question 1:** There is a unique positive real number x such that the three numbers $\log_8 2x$,
 899 $\log_4 x$, and $\log_2 x$, in that order, form a geometric progression with positive common ratio. The
 900 number x can be written as $\frac{m}{n}$, where m and n are relatively prime positive integers. Find $m + n$
 901

Incorrect solution by TIR-ToRA

902 Let's approach this problem step-by-step:

- 905 1. Define a function to check if three numbers form a geometric progression.
- 906 2. Use binary search to find x that satisfies the condition.
- 907 3. Convert x to a fraction and return the sum of its numerator and denominator.

```
909 import math
910 from fractions import Fraction
911 def solution():
912     # Function to check if three numbers form
913     # a geometric progression
914     def is_geometric_progression(a, b, c):
915         ratio1 = b / a
916         ratio2 = c / b
917         return math.isclose(ratio1, ratio2) and ratio1 > 1
918     # Binary search to find x
```

```

918
919     left, right = 1, 1000
920     while right - left > 1e-10:
921         x = (left + right) / 2
922         log8_2x = math.log(2*x, 8)
923         log4_x = math.log(x, 4)
924         log2_x = math.log(x, 2)
925         if is_geometric_progression(log8_2x, log4_x, log2_x):
926             break
927         elif (log4_x / log8_2x) > (log2_x / log4_x):
928             left = x
929         else:
930             right = x
931     # Convert x to a fraction
932     frac = Fraction(x).limit_denominator()
933     return frac.numerator + frac.denominator
934 result = solution()
935 print(f"\\boxed{{{result}}}")
936
937 >>> output:
938 \\boxed{2}

```

The answer is

Correct solution by SBSC:

Let's solve this problem step by step using SymPy-based Python code.

Step 1: Define the symbols and set up the equations

```

940
941
942
943 from sympy import symbols, log, Eq, solve
944 # Define symbols
945 x, r = symbols('x r')
946 # Define the three logarithms
947 log1 = log(2*x, 8)
948 log2 = log(x, 4)
949 log3 = log(x, 2)
950 # Set up equations for geometric progression
951 eq1 = Eq(log2 / log1, r)
952 eq2 = Eq(log3 / log2, r)
953 print("Equation 1:", eq1)
954 print("Equation 2:", eq2)
955
956 >>> output
957 Equation 1: Eq(log(8)*log(x)/(log(4)*log(2*x)), r)
958 Equation 2: Eq(log(4)/log(2), r)

```

Let's continue solving this problem step by step using SymPy-based Python code.

Step 2: Simplify the equations

```

959 from sympy import symbols, log, Eq, solve, simplify
960 # Define symbols
961 x, r = symbols('x r')
962 # Define the three logarithms
963 log1 = log(2*x, 8)
964 log2 = log(x, 4)
965 log3 = log(x, 2)
966 # Set up equations for geometric progression
967 eq1 = Eq(log2 / log1, r)
968 eq2 = Eq(log3 / log2, r)
969 # Simplify equations
970 eq1_simplified = simplify(eq1)
971 eq2_simplified = simplify(eq2)
972 print("Simplified Equation 1:", eq1_simplified)
973 print("Simplified Equation 2:", eq2_simplified)

```

972

973

974

>>> output:

Simplified Equation 1: Eq(r, 3*log(x)/(2*log(x) + 2*log(2)))

975

Simplified Equation 2: Eq(r, 2)

976

977

Let's continue solving this problem step by step using SymPy-based Python code.

978

Step 3: Solve the equations

979

```
from sympy import symbols, log, Eq, solve, simplify
```

980

```
# Define symbols
```

981

```
x, r = symbols('x r')
```

982

```
# Define the simplified equations
```

983

```
eq1 = Eq(r, 3*log(x)/(2*log(x) + 2*log(2)))
```

984

```
eq2 = Eq(r, 2)
```

985

```
# Solve the system of equations
```

986

```
solution = solve((eq1, eq2), (x, r))
```

987

```
print("Solution:", solution)
```

987

>>> output

988

Solution: [(1/16, 2)]

989

Let's continue solving this problem step by step using SymPy-based Python code.

990

Step 4: Extract the value of x and convert to a fraction

991

```
from sympy import symbols, log, Eq, solve, simplify, Rational
```

992

```
# Define symbols
```

993

```
x, r = symbols('x r')
```

994

```
# Define the solution from the previous step
```

995

```
solution = [(Rational(1, 16), 2)]
```

996

```
# Extract the value of x
```

997

```
x_value = solution[0][0]
```

998

```
print("x =", x_value)
```

999

```
# Convert x to a fraction
```

1000

```
m = x_value.numerator
```

1001

```
n = x_value.denominator
```

1002

```
print("x as a fraction: {}/{}".format(m, n))
```

1003

>>> output

1004

x = 1/16

1005

x as a fraction: 1/16

1006

Let's continue solving this problem step by step using SymPy-based Python code.

1007

Step 5: Calculate m + n

1008

```
from sympy import symbols, log, Eq, solve, simplify, Rational
```

1009

```
# Define symbols
```

1010

```
x, r = symbols('x r')
```

1011

```
# Define the solution from the previous step
```

1012

```
solution = [(Rational(1, 16), 2)]
```

1013

```
# Extract the value of x
```

1014

```
x_value = solution[0][0]
```

1015

```
# Convert x to a fraction
```

1016

```
m = x_value.numerator
```

1017

```
n = x_value.denominator
```

1018

```
# Calculate m + n
```

1019

```
result = m + n
```

1020

```
print("m =", m)
```

1021

```
print("n =", n)
```

1022

```
print("m + n =", result)
```

1023

>>> output:

1024

m = 1

1025

n = 16

m + n = 17

1026
1027
1028
1029

```
### END OF CODE
The final answer is 
```

1030
1031
1032

Example Question 2: Consider functions f that satisfy $|f(x) - f(y)| \leq 0.5|x - y|$ for all real numbers x and y . Of all such functions that also satisfy the equation $f(300) = f(900)$, what is the greatest possible value of $f(f(800)) - f(f(400))$?

1033
1034

Incorrect solution by TIR-ToRA:

1035
1036

To solve this problem, we need to consider the properties of the given function and find the maximum possible difference between $f(f(800))$ and $f(f(400))$. Let's solve it using python:

1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050

```
def solution():
    #Given: |f(x) - f(y)| <= 0.5|x - y| for all real x and y
    #Also, f(300) = f(900)
    #Maximum difference between f(800) and f(400)
    max_diff_800_400 = 0.5 * (800 - 400) # 200
    #Maximum value of f(800) - f(400)
    max_f800_f400 = max_diff_800_400 # 200
    #Maximum difference between f(f(800)) and f(f(400))
    max_final_diff = 0.5 * max_f800_f400 # 100
    return max_final_diff
result = solution()
print(f"\boxed{{{result}}}")

>>> output:
\boxed{100}
```

1051
1052

The answer is

1053
1054

Correct solution by SBSC:

1055
1056
1057
1058
1059
1060

Let's approach this problem step by step using SymPy-based Python code where appropriate.
Step 1: Understand the given condition The condition $|f(x) - f(y)| \leq 0.5|x - y|$ implies that f is a Lipschitz continuous function with Lipschitz constant $1/2$. This means f doesn't change too rapidly.
Step 2: Analyze the equation $f(300)=f(900)$. This equation tells us that f takes the same value at $x=300$ and $x=900$. Let's call this value k .

1061
1062
1063
1064
1065

```
from sympy import symbols, Abs
k = symbols('k')
print(f"f(300) = f(900) = {k}")

>>> output:
f(300) = f(900) = k
```

1066
1067
1068

Let's continue solving this problem step by step.
Step 3: Apply the Lipschitz condition to $f(800)$ and $f(400)$ We can use the Lipschitz condition to find bounds for $f(800)$ and $f(400)$ in terms of k .

1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079

```
from sympy import symbols, Abs, Min, Max
k = symbols('k')
# For f(800)
f_800_min = k - Abs(800 - 900)/2
f_800_max = k + Abs(800 - 900)/2
# For f(400)
f_400_min = k - Abs(400 - 300)/2
f_400_max = k + Abs(400 - 300)/2
print(f"f(800) is between {f_800_min} and {f_800_max}")
print(f"f(400) is between {f_400_min} and {f_400_max}")

>>> output:
```

1080

1081 `f(800) is between k - 50 and k + 50`1082 `f(400) is between k - 50 and k + 50`

1083

Let's continue solving this problem step by step.

1084

Step 4: Analyze $f(f(800))$ and $f(f(400))$ Now we need to apply the Lipschitz condition again

1085

to $f(f(800))$ and $f(f(400))$.

1086

`from sympy import symbols, Abs, Min, Max`

1087

`k = symbols('k')`

1088

`# For f(f(800))`

1089

`ff_800_min = k - Abs(k + 50 - k)/2`

1090

`ff_800_max = k + Abs(k - 50 - k)/2`

1091

`# For f(f(400))`

1092

`ff_400_min = k - Abs(k + 50 - k)/2`

1093

`print(f"f(f(800)) is between {ff_800_min} and {ff_800_max}")`

1094

`print(f"f(f(400)) is between {ff_400_min} and {ff_400_max}")`

1095

1096

`>>> output`

1097

`f(f(800)) is between k - 25 and k + 25`

1098

`f(f(400)) is between k - 25 and k + 25`

1099

Let's continue solving this problem step by step.

1100

Step 5: Calculate the maximum possible difference Now we need to find the maximum possible

1101

value of $f(f(800)) - f(f(400))$.

1102

`from sympy import symbols, Abs, Min, Max`

1103

`k = symbols('k')`

1104

`# Maximum possible difference`

1105

`max_diff = (k + 25) - (k - 25)`

1106

`print(f"The maximum possible difference f(f(800))-f(f(400))`

1107

`is {max_diff}")`

1108

`>>> output:`

1109

`The maximum possible difference f(f(800)) - f(f(400)) is 50`

1110

`### END OF CODE`

1111

The final answer is `50`

1112

1113

Example Question 3: Find the number of ordered pairs of positive integers (m, n) such that $m^2n = 20^{20}$

1114

1115

1116

Incorrect solution by TIR-ToRA:

1117

Let's approach this problem step-by-step:

1118

1119

1. Iterate through possible values of m up to the square root of 20^{20}

1120

2. For each m , check if m^2 divides 20^{20} evenly

1121

3. Return the total count of valid pairs

1122

1123

`import math`

1124

`def solution():`

1125

`count = 0`

1126

`target = 20**20`

1127

`for m in range(1, int(math.sqrt(target)) + 1):`

1128

`if target % (m**2) == 0:`

1129

`n = target // (m**2)`

1130

`count += 1`

1131

`return count`

1132

`result = solution()`

1133

`print(f"The number of ordered pairs (m,n) is:``f"\boxed{{{result}}}")`

```

1134
1135 >>> output:
1136 TimeoutError: [Errno 110] Connection timed out
1137 Looks like we got a timeout error, let's try a different approach that should not result in any
1138 error:
1139
1140 import math
1141 def solution():
1142     count = 0
1143     target = 20**20
1144     for m in range(1, int(math.sqrt(target)) + 1):
1145         if target % (m**2) == 0:
1146             n = target // (m**2)
1147             count += 1
1148     return count
1149 result = solution()
1150 print(f"The number of ordered pairs (m,n) is:
1151 f"\boxed{{{result}}}")
1152
1153 >>> output:
1154 TimeoutError: [Errno 110] Connection timed out
1155
1156 ...
1157 This loop continues without ultimately getting an answer.

```

Correct solution by SBSC:

```

1156
1157 Let's approach this problem step by step using SymPy-based Python code.
1158 Step 1: Define the equation and initialize variables
1159
1160 from sympy import symbols, Eq, solve, divisors, prime_factors
1161 m, n = symbols('m n', positive=True, integer=True)
1162 equation = Eq(m**2 * n, 20**20)
1163 print("Equation:", equation)
1164
1165 >>> output:
1166 ImportError: cannot import name 'prime_factors' from 'sympy'
1167
1168 I apologize for the error. It seems the 'prime factors' function is not available in the version of
1169 SymPy you're using. Let's modify our approach and use the 'factorint' function instead, which
1170 is more commonly available. Here's the corrected code:
1171 Step 1: Define the equation and initialize variables
1172
1173 from sympy import symbols, Eq, solve, divisors, factorint
1174 m, n = symbols('m n', positive=True, integer=True)
1175 equation = Eq(m**2 * n, 20**20)
1176 print("Equation:", equation)
1177
1178 >>> output
1179 Equation: Eq(m**2*n, 1048576000000000000000000000000)
1180
1181 Let's continue solving this problem step by step.
1182 Step 2: Factor  $20^{20}$  and analyze its structure
1183
1184 from sympy import symbols, Eq, solve, divisors, factorint
1185 m, n = symbols('m n', positive=True, integer=True)
1186 equation = Eq(m**2 * n, 20**20)
1187 # Factor  $20^{20}$ 
1188 factorization = factorint(20**20)
1189 print("Factorization of  $20^{20}$ :", factorization)
1190 # Analyze the structure
1191 power_of_2 = factorization[2]
1192 power_of_5 = factorization[5]
1193 print("Power of 2:", power_of_2)

```

```

1188
1189 print("Power of 5:", power_of_5)
1190
1191 >>> output
1192 Factorization of 20^20: {2: 40, 5: 20}
1193 Power of 2: 40
1194 Power of 5: 20
1195
1196 Let's continue solving this problem step by step.
1197 Step 3: Determine possible values for m and n
1198
1199 from sympy import symbols, Eq, solve, divisors, factorint
1200 m, n = symbols('m n', positive=True, integer=True)
1201 equation = Eq(m**2 * n, 20**20)
1202 # Factor 20^20
1203 factorization = factorint(20**20)
1204 power_of_2 = factorization[2]
1205 power_of_5 = factorization[5]
1206 # Initialize counter for valid pairs
1207 valid_pairs = 0
1208 # Iterate through possible values of m
1209 for m_power_of_2 in range(0, power_of_2 + 1, 2):
1210     for m_power_of_5 in range(0, power_of_5 + 1):
1211         # Calculate m
1212         m_value = 2**m_power_of_2 * 5**m_power_of_5
1213         # Calculate corresponding n
1214         n_power_of_2 = power_of_2 - 2*m_power_of_2
1215         n_power_of_5 = power_of_5 - m_power_of_5
1216         # Check if n is a positive integer
1217         if n_power_of_2 >= 0 and n_power_of_5 >= 0:
1218             valid_pairs += 1
1219 print("Number of valid (m,n) pairs:", valid_pairs)
1220
1221 >>> output
1222 Number of valid (m,n) pairs: 231
1223
1224 ### END OF CODE
1225 The final answer is 231

```

Example Question 4: Consider the sequence $(a_k)_{k \geq 1}$ of positive rational numbers defined by $a_1 = \frac{2020}{2021}$ and for $k \geq 1$, if $a_k = \frac{m}{n}$ for relatively prime positive integers m and n , then $a_{k+1} = \frac{m+18}{n+19}$. Determine the sum of all positive integers j such that the rational number a_j can be written in the form $\frac{t}{t+1}$ for some positive integer t

Incorrect solution by TIR-ToRA:

Let's proceed step-by-step:

1. Generate the sequence of fractions starting from $a_1 = \frac{2020}{2021}$.
2. For each term, check if it is in the form $\frac{t}{t+1}$ for some integer t . If so, add its index to a running sum.
3. Generate the next term using the given formula.
4. Continue this process until we cycle back to the initial fraction. The sum of indices where the fraction is in the form $\frac{t}{t+1}$ is our final answer.

```

1236 from fractions import Fraction
1237 def next_term(a):
1238     return Fraction(a.numerator + 18, a.denominator + 19)
1239 def is_of_form_t_over_t_plus_1(frac):
1240     return frac.denominator == frac.numerator + 1
1241 def solution():

```

```

1242
1243     a = Fraction(2020, 2021)
1244     j_sum = 0
1245     j = 1
1246     while True:
1247         if is_of_form_t_over_t_plus_1(a):
1248             j_sum += j
1249             a = next_term(a)
1250             j += 1
1251             # Check if we've reached the initial fraction again
1252             if a == Fraction(2020, 2021):
1253                 break
1254     return j_sum
1255 result = solution()
1256 print(f"\\boxed{{{result}}}")
1257

```

Similar to the earlier example, this code results in a TimeOut error and then the loop keeps repeating.

Correct solution by SBSC:

```

1258
1259
1260 Let's solve this problem step by step using SymPy-based Python code.
1261 Step 1: Define the sequence and initial value
1262
1263 from sympy import Rational, gcd
1264 def next_term(a):
1265     m, n = a.numerator, a.denominator
1266     return Rational(m + 18, n + 19)
1267 a = [Rational(2020, 2021)]
1268 print(f"a[1] = {a[0]}")
1269
1270 >>> output
1271 a[1] = 2020/2021
1272
1273 Let's continue solving this problem step by step using SymPy-based Python code.
1274 Step 2: Generate the sequence until we find a term of the form  $t/(t+1)$ 
1275
1276 from sympy import Rational, gcd
1277 def next_term(a):
1278     m, n = a.numerator, a.denominator
1279     return Rational(m + 18, n + 19)
1280 a = [Rational(2020, 2021)]
1281 j = 1
1282 result = []
1283 while True:
1284     if a[-1].denominator == a[-1].numerator + 1:
1285         result.append(j)
1286         print(f"Found a[{j}] = {a[-1]} of the form  $t/(t+1)$ ")
1287         j += 1
1288         a.append(next_term(a[-1]))
1289         if j > 100: # Limit to prevent infinite loop
1290             break
1291 print(f"Indices found: {result}")
1292
1293 >>> output
1294 Found a[1] = 2020/2021 of the form  $t/(t+1)$ 
1295 Found a[2] = 1019/1020 of the form  $t/(t+1)$ 
1296 Found a[8] = 161/162 of the form  $t/(t+1)$ 
1297 Found a[18] = 31/32 of the form  $t/(t+1)$ 
1298 Found a[30] = 19/20 of the form  $t/(t+1)$ 
1299 Indices found: [1, 2, 8, 18, 30]
1300
1301 Let's continue solving this problem step by step using SymPy-based Python code.
1302 Step 3: Calculate the sum of the indices
1303 from sympy import Rational, gcd

```



```

1296
1297 def next_term(a):
1298     m, n = a.numerator, a.denominator
1299     return Rational(m + 18, n + 19)
1300 a = [Rational(2020, 2021)]
1301 j = 1
1302 result = []
1303 while True:
1304     if a[-1].denominator == a[-1].numerator + 1:
1305         result.append(j)
1306         j += 1
1307         a.append(next_term(a[-1]))
1308         if j > 100: # Limit to prevent infinite loop
1309             break
1310 sum_of_indices = sum(result)
1311 print(f"Indices found: {result}")
1312 print(f"Sum of indices: {sum_of_indices}")
1313
1314 >>> output
1315 Indices found: [1, 2, 8, 18, 30]
1316 Sum of indices: 59
1317
1318 ### END OF CODE
1319 The final answer is 59

```

1318 A.3 PAL EXEMPLARS

1319
1320 In this section, we provide the prompts for Program-Aided Language models (PAL) method. We
1321 initially used the default prompt as mentioned in the original PAL paper, but the results were poor.
1322 We noticed that the response often contained textual reasoning before or after the program, which
1323 isn't the desired format for PAL. Hence, we modify the instructions to confine the responses only to
1324 include Python program and subsequently, also notice improved accuracy.

1325 For AIME

1326 Let's use python program to solve math problems.

1327 DO NOT USE ANY TEXTUAL REASONING.

1328 Your response must start with: "python

1329 Your response must end with: print(result)

1330 Here are some examples you may refer to.

1331 **Example Problem:** A frog begins at $P_0 = (0, 0)$ and makes a sequence of jumps according to
1332 the following rule: from $P_n = (x_n, y_n)$, the frog jumps to P_{n+1} , which may be any of the points
1333 $(x_n + 7, y_n + 2)$, $(x_n + 2, y_n + 7)$, $(x_n - 5, y_n - 10)$, or $(x_n - 10, y_n - 5)$. There are M points
1334 (x, y) with $|x| + |y| \leq 100$ that can be reached by a sequence of such jumps. Find the remainder
1335 when M is divided by 1000.

1336 Example Solution:

```

1338 def solution():
1339     jumps = [(7, 2), (2, 7), (-5, -10), (-10, -5)]
1340     # Set to keep track of all reachable points, starting from the origin
1341     (0, 0).
1342     reachable = set([(0, 0)])
1343     # Queue to process points, starting with the origin (0, 0).
1344     queue = [(0, 0)]
1345     # Breadth-first search (BFS) to explore reachable points.
1346     while queue:
1347         # Pop the first point from the queue.
1348         x, y = queue.pop(0)
1349         # Iterate over all possible jumps.
1350         for dx, dy in jumps:
1351             # Calculate new coordinates after the jump.
1352             nx, ny = x + dx, y + dy

```

```

1350     # Check if the Manhattan distance is within 100 and the point
1351     hasn't been visited.
1352     if abs(nx) + abs(ny) <= 100 and (nx, ny) not in reachable:
1353         # Add the new point to the reachable set.
1354         reachable.add((nx, ny))
1355         # Add the new point to the queue to explore further.
1356         queue.append((nx, ny))
1357     return len(reachable) % 1000
1358 result = solution()
1359 print(result)

```

Example Problem: The AIME Triathlon consists of a half-mile swim, a 30-mile bicycle ride, and an eight-mile run. Tom swims, bicycles, and runs at constant rates. He runs five times as fast as he swims, and he bicycles twice as fast as he runs. Tom completes the AIME Triathlon in four and a quarter hours. How many minutes does he spend bicycling?

Example Solution:

```

1365 from sympy import symbols, Eq, solve, Rational
1366 def solution():
1367     x = symbols('x')
1368     # Set up the equation
1369     eq = Eq(Rational(1,2)/x + 30/(10*x) + 8/(5*x), Rational(17,4))
1370     # Solve the equation
1371     solution = solve(eq)[0]
1372     # Calculate bicycling time in hours
1373     bike_time = 30 / (10 * solution)
1374     # Convert to minutes
1375     bike_time_minutes = int(bike_time * 60)
1376     return bike_time_minutes
1377 result = solution()
1378 print(result)

```

Example Problem: Let S be the increasing sequence of positive integers whose binary representation has exactly 8 ones. Let N be the 1000th number in S . Find the remainder when N is divided by 1000

Example Solution:

```

1382 def solution():
1383     count = 0 # Initialize a counter to track how many numbers have been
1384     found
1385     n = 1 # Start checking numbers from 1 upwards
1386     while count < 1000: # Continue the loop until we find the 1000th
1387     number
1388         # Check if the binary representation of the number 'n' has
1389         exactly 8 '1's
1390         if bin(n).count('1') == 8:
1391             count += 1 # Increment the counter when a number with 8 '1's
1392             is found
1393             # If this is the 1000th such number, return the remainder of
1394             n divided by 1000
1395             if count == 1000:
1396                 return n % 1000
1397             n += 1 # Move to the next number
1398 result = solution()
1399 print(result)

```

Example Problem: Two geometric sequences a_1, a_2, a_3, \dots and b_1, b_2, b_3, \dots have the same common ratio, with $a_1 = 27$, $b_1 = 99$, and $a_{15} = b_{11}$. Find a_9

Example Solution:

```

1402 def solution():
1403     # Initialize known values
1404     a1 = 27

```

```

1404     b1 = 99
1405     # Calculate the common ratio
1406     # We know that a15 = b11, so:
1407     # a1 * r^14 = b1 * r^10
1408     # 27 * r^14 = 99 * r^10
1409     # 27 * r^4 = 99
1410     # r^4 = 99/27 = 11/3
1411     r = (11/3) ** (1/4)
1412     # Calculate a9
1413     a9 = a1 * (r ** 8)
1414     return round(a9)
1415 result = solution()
1416 print(result)

```

For AMC:

Let's use python program to solve math problems.

DO NOT USE ANY TEXTUAL REASONING.

Your response must start with: "python

Your response must end with: print(result)

Here are some examples you may refer to.

Example Problem: Small lights are hung on a string 6 inches apart in the order red, red, green, green, green, red, red, green, green, green, and so on continuing this pattern of 2 red lights followed by 3 green lights. How many feet separate the 3rd red light and the 21st red light? Note: 1 foot is equal to 12 inches.

Example Solution:

```

1428 def solution():
1429     # Find position of 3rd red light
1430     n_3rd = 3
1431     complete_cycles_3rd = (n_3rd - 1) // 2
1432     remaining_lights_3rd = (n_3rd - 1) % 2
1433     pos_3rd = complete_cycles_3rd * 5 * 6 + remaining_lights_3rd * 6
1434     # Find position of 21st red light
1435     n_21st = 21
1436     complete_cycles_21st = (n_21st - 1) // 2
1437     remaining_lights_21st = (n_21st - 1) % 2
1438     pos_21st = complete_cycles_21st * 5 * 6 + remaining_lights_21st * 6
1439     # Calculate the distance in inches
1440     distance_inches = pos_21st - pos_3rd
1441     # Convert to feet
1442     distance_feet = distance_inches / 12
1443     return distance_feet
1444 result = solution()
1445 print(result)

```

Example Problem: A fruit salad consists of blueberries, raspberries, grapes, and cherries. The fruit salad has a total of 280 pieces of fruit. There are twice as many raspberries as blueberries, three times as many grapes as cherries, and four times as many cherries as raspberries. How many cherries are there in the fruit salad?

Example Solution:

```

1450 from sympy import symbols, Eq, solve
1451 def solution():
1452     # Define the symbols for the variables
1453     b, r, g, c = symbols('b r g c')
1454     # Define the equations based on the problem statement
1455     eq1 = Eq(r, 2*b) # Equation 1: r = 2b
1456     eq2 = Eq(g, 3*c) # Equation 2: g = 3c
1457     eq3 = Eq(c, 4*r) # Equation 3: c = 4r
1458     eq4 = Eq(b + r + g + c, 280) # Equation 4: b + r + g + c = 280
1459     # Solve the system of equations

```

```

1458     sol = solve((eq1, eq2, eq3, eq4))
1459     return sol[c]
1460 result = solution()
1461 print(result)

```

1462

Example Problem: Last summer 30% of the birds living on Town Lake were geese, 25% were swans, 10% were herons, and 35% were ducks. What percent of the birds that were not swans were geese?

1465

1466 **Example Solution:**

1467

```

1468 def solution():
1469     # Total percentage of all birds
1470     total = 100
1471     # Percentages of each bird type
1472     geese = 30
1473     swans = 25
1474     herons = 10
1475     ducks = 35
1476     # Calculate percentage of birds that are not swans
1477     not_swans = total - swans
1478     # Calculate percentage of geese among birds that are not swans
1479     geese_among_not_swans = (geese / not_swans) * 100
1480     # Round to nearest whole number
1481     return round(geese_among_not_swans)
1482 result = solution()
1483 print(result)

```

1481

Example Problem: At a twins and triplets convention, there were 9 sets of twins and 6 sets of triplets, all from different families. Each twin shook hands with all the twins except his/her siblings and with half the triplets. Each triplet shook hands with all the triplets except his/her siblings and with half the twins. How many handshakes took place?

1485

1486 **Example Solution:**

1487

```

1488 def solution():
1489     # Number of twins and triplets
1490     twins = 9 * 2
1491     triplets = 6 * 3
1492     # Handshakes between twins
1493     twin_handshakes = (twins * (twins - 2)) // 2
1494     # Handshakes between triplets
1495     triplet_handshakes = (triplets * (triplets - 3)) // 2
1496     # Handshakes between twins and triplets
1497     twin_triplet_handshakes = (twins * triplets) // 2
1498     # Total handshakes
1499     total_handshakes = twin_handshakes + triplet_handshakes +
1500     twin_triplet_handshakes
1501     return total_handshakes
1502 result = solution()
1503 print(result)

```

1501

1502

1503 A.4 TIR-TORA EXEMPLARS

1504

1505 For AIME

1506 Integrate step-by-step reasoning and Python code to solve math problems using the following
1507 guidelines:

- 1508 - Analyze the question and write functions to solve the problem; the function should not take any
1509 arguments.
- 1510 - Present the final result in LaTeX using a \square without any units.
- 1511 - Utilize the 'pi' symbol and 'Rational' from Sympy for π and fractions, and simplify all fractions
and square roots without converting them to decimal values.

1512 Once you get the code output, just display the output answer within `'□'` and terminate.

1513 Here are some examples you may refer to:

1514 **Example Problem:** A frog begins at $P_0 = (0, 0)$ and makes a sequence of jumps according to
 1515 the following rule: from $P_n = (x_n, y_n)$, the frog jumps to P_{n+1} , which may be any of the points
 1516 $(x_n + 7, y_n + 2)$, $(x_n + 2, y_n + 7)$, $(x_n - 5, y_n - 10)$, or $(x_n - 10, y_n - 5)$. There are M points
 1517 (x, y) with $|x| + |y| \leq 100$ that can be reached by a sequence of such jumps. Find the remainder
 1518 when M is divided by 1000.

1519 **Example Solution:**

1520 Let's approach this problem step-by-step:

- 1521 1. We need to find all points (x, y) that the frog can reach within the given constraint $|x| + |y|$
- 1522 100.
- 1523 2. We'll use a breadth-first search (BFS) algorithm to explore all possible points.
- 1524 3. We'll keep track of visited points to avoid revisiting them.
- 1525 4. We'll count the number of unique points visited.

1526 Here's the Python code to solve this:

```

1527 from collections import deque
1528 def count_reachable_points():
1529     # Possible jumps
1530     jumps = [(7, 2), (2, 7), (-5, -10), (-10, -5)]
1531     # Set to store visited points
1532     visited = set([(0, 0)])
1533     # Queue for BFS
1534     queue = deque([(0, 0)])
1535     while queue:
1536         x, y = queue.popleft()
1537         for dx, dy in jumps:
1538             new_x, new_y = x + dx, y + dy
1539             # Check if the new point is within the constraint and not
1540             # visited
1541             if abs(new_x) + abs(new_y) <= 100 and (new_x, new_y) not in
1542             visited:
1543                 visited.add((new_x, new_y))
1544                 queue.append((new_x, new_y))
1545     # Count of reachable points is the size of visited set
1546     M = len(visited)
1547     # Return the remainder when M is divided by 1000
1548     return M % 1000
1549 result = count_reachable_points()
1550 print(f"\boxed{{{result}}}")
1551
1552 >>> output
1553 \boxed{373}

```

1554 The answer is `□` `373`

1555 **Example Problem:** The AIME Triathlon consists of a half-mile swim, a 30-mile bicycle ride, and
 1556 an eight-mile run. Tom swims, bicycles, and runs at constant rates. He runs five times as fast as he
 1557 swims, and he bicycles twice as fast as he runs. Tom completes the AIME Triathlon in four and a
 1558 quarter hours. How many minutes does he spend bicycling?

1559 **Example Solution:**

1560 Let's approach this problem step-by-step:

- 1561 1. Let's define variables:
- 1562 - Let x be Tom's swimming speed in miles per hour
- 1563 - Then $5x$ is his running speed in miles per hour
- 1564 - And $10x$ is his bicycling speed in miles per hour

- 1566 2. We can set up equations based on the given information:
 1567 - Swimming time: $0.5 / x$ hours
 1568 - Bicycling time: $30 / (10x)$ hours
 1569 - Running time: $8 / (5x)$ hours
 1570 3. The total time is 4.25 hours, so we can set up an equation:
 1571 $0.5/x + 30/(10x) + 8/(5x) = 4.25$
 1572 4. Let's solve this equation and find the bicycling time.

1574 Here's the Python code to solve this:

```
1576 from sympy import symbols, Eq, solve, Rational
1577 def solve_triathlon():
1578     x = symbols('x')
1579     # Set up the equation
1580     eq = Eq(Rational(1,2)/x + 30/(10*x) + 8/(5*x), Rational(17,4))
1581     # Solve the equation
1582     solution = solve(eq)[0]
1583     # Calculate bicycling time in hours
1584     bike_time = 30 / (10 * solution)
1585     # Convert to minutes
1586     bike_time_minutes = int(bike_time * 60)
1587     return f"\boxed{{{bike_time_minutes}}}"
1588 print(solve_triathlon())
1589
1590 >>> output:
1591 \boxed{150}
```

1592 The answer is 150

1593 **Example Problem:** Let S be the increasing sequence of positive integers whose binary representation
 1594 has exactly 8 ones. Let N be the 1000th number in S . Find the remainder when N is divided by
 1595 1000

1596 **Example Solution:**

1597 Let's approach this step-by-step:

- 1598 1. We need to generate the sequence S of numbers with exactly 8 ones in their binary represen-
 1599 tation.
 1600 2. We'll find the 1000th number in this sequence.
 1601 3. Finally, we'll calculate the remainder when this number is divided by 1000.
 1602

1603 Here's the Python code to solve this:

```
1605 def count_ones(n):
1606     return bin(n).count('1')
1607 def solution():
1608     count = 0
1609     n = 1
1610     while count < 1000:
1611         if count_ones(n) == 8:
1612             count += 1
1613             if count == 1000:
1614                 return n % 1000
1615         n += 1
1616 result = solution()
1617 print(result)
1618
1619 >>> output:
1620 \boxed{32}
```

1621 The answer is 32

1620 **Example Problem:** Two geometric sequences a_1, a_2, a_3, \dots and b_1, b_2, b_3, \dots have the same com-
 1621 mon ratio, with $a_1 = 27$ $b_1 = 99$, and $a_{15} = b_{11}$. Find a_9

1622 **Example Solution:**

1623 Let's approach this step-by-step:

- 1625 1. Let's denote the common ratio as r .
- 1626 2. We can express a and b in terms of their first terms and r .
- 1627 3. We can then equate these expressions to find r .
- 1628 4. Once we have r , we can calculate a .

1630 Here's the Python code to solve this:

```
1632 from sympy import symbols, Eq, solve, Rational
1633 def find_a9():
1634     r = symbols('r')
1635     # a = a * r
1636     # b = b * r
1637     # a = b
1638     eq = Eq(27 * r**14, 99 * r**10)
1639     # Solve for r
1640     solution = solve(eq)
1641     # We expect only one positive real solution
1642     r_value = [sol for sol in solution if sol.is_real and sol > 0][0]
1643     # Calculate a
1644     a9 = 27 * r_value**8
1645     return a9
1646 result = find_a9()
1647 print(f"\boxed{{{result}}}")
1648
1649 >>> output
1650 \boxed{363}
```

1649 The answer is $\boxed{363}$

1650 **For AMC**

1651 Integrate step-by-step reasoning and Python code to solve math problems using the following
 1652 guidelines:

- 1653 - Analyze the question and write functions to solve the problem; the function should not take any
 1654 arguments. - Present the final result in LaTeX using a $\boxed{}$ without any units. - Utilize the 'pi' symbol
 1655 and 'Rational' from Sympy for π and fractions, and simplify all fractions and square roots without
 1656 converting them to decimal values.

1657 Once you get the code output, just display the output answer within $\boxed{}$ and terminate. Here are some
 1658 examples you may refer to:

1659 **Example Problem:** Small lights are hung on a string 6 inches apart in the order red, red, green,
 1660 green, green, red, red, green, green, and so on continuing this pattern of 2 red lights followed
 1661 by 3 green lights. How many feet separate the 3rd red light and the 21st red light? Note: 1 foot is
 1662 equal to 12 inches.

1663 **Example Solution:**

1664 Let's approach this step-by-step:

- 1665 1. First, we need to understand the pattern of lights:
 - 1666 - Every 5 lights (2 red + 3 green) complete one cycle of the pattern.
 - 1667 - Each cycle is 30 inches long (5 lights * 6 inches).
- 1668 2. We need to find how many complete cycles are between the 3rd and 21st red lights:
 - 1669 - 3rd red light is at the end of the 2nd cycle (as each cycle has 2 red lights).
 - 1670 - 21st red light is at the end of the 11th cycle (as $21 \div 2 = 10.5$, rounded up to 11).
 - 1671 - So there are 9 complete cycles between them ($11 - 2 = 9$).

- 1674 3. Calculate the distance:
 1675 - 9 cycles * 30 inches per cycle = 270 inches
 1676 - Convert 270 inches to feet: $270 \div 12 = 22.5$ feet
 1677

1678 Here's the Python code to solve this:

```
1679 from sympy import Rational
1680 def calculate_distance():
1681     lights_per_cycle = 5
1682     inches_per_light = 6
1683     inches_per_cycle = lights_per_cycle * inches_per_light
1684     start_cycle = 2 # 3rd red light is at the end of 2nd cycle
1685     end_cycle = 11 # 21st red light is at the end of 11th cycle
1686     cycles_between = end_cycle - start_cycle
1687     total_inches = cycles_between * inches_per_cycle
1688     total_feet = Rational(total_inches, 12)
1689     return f"\boxed{{{total_feet}}}"
1690 print(calculate_distance())
1691
1692 >>> output:
1693 \boxed{45/2}
```

1694 The answer is $\boxed{45/2}$

1695 **Example Problem:** A fruit salad consists of blueberries, raspberries, grapes, and cherries. The fruit
 1696 salad has a total of 280 pieces of fruit. There are twice as many raspberries as blueberries, three times
 1697 as many grapes as cherries, and four times as many cherries as raspberries. How many cherries are
 1698 there in the fruit salad?

1699 **Example Solution:**

1700 Let's approach this problem step-by-step:

- 1701
- 1702 1. Let's define variables for each fruit:
 1703 b = number of blueberries
 1704 r = number of raspberries
 1705 g = number of grapes
 1706 c = number of cherries
 - 1707 2. We can set up equations based on the given information:
 1708 $r = 2b$ (twice as many raspberries as blueberries)
 1709 $g = 3c$ (three times as many grapes as cherries)
 1710 $c = 4r$ (four times as many cherries as raspberries)
 - 1711 3. We also know that the total number of fruits is 280:
 1712 $b + r + g + c = 280$
 - 1713 4. Now, let's write a Python function to solve this system of equations:
 1714

```
1715 from sympy import symbols, Eq, solve
1716 def solve_fruit_salad():
1717     b, r, g, c = symbols('b r g c')
1718     eq1 = Eq(r, 2*b)
1719     eq2 = Eq(g, 3*c)
1720     eq3 = Eq(c, 4*r)
1721     eq4 = Eq(b + r + g + c, 280)
1722     solution = solve((eq1, eq2, eq3, eq4))
1723     return f"\boxed{{{solution[c]}}}"
1724 print(solve_fruit_salad())
1725
1726 >>> output:
1727 \boxed{64}
```

1727 The answer is $\boxed{64}$

1728 **Example Problem:** Last summer 30% of the birds living on Town Lake were geese, 25% were
 1729 swans, 10% were herons, and 35% were ducks. What percent of the birds that were not swans were
 1730 geese?

1731 **Example Solution:**

1732 Let's approach this step-by-step:

- 1734 1. First, we need to calculate the percentage of birds that were not swans.
- 1735 2. Then, we'll calculate what percentage of this non-swan group were geese.

1737 Here's the Python code to solve this:

```
1738
1739 from sympy import Rational
1740 def calculate_geese_percentage():
1741     # Total percentage of birds
1742     total = 100
1743     # Percentage of swans
1744     swans = 25
1745     # Percentage of non-swans
1746     non_swans = total - swans
1747     # Percentage of geese among all birds
1748     geese = 30
1749     # Calculate the percentage of geese among non-swans
1750     geese_among_non_swans = Rational(geese, non_swans) * 100
1751     # Simplify the fraction
1752     result = geese_among_non_swans.simplify()
1753     return f"\\boxed{{{result}}}"
1754 print(calculate_geese_percentage())
1755
1756 >>> output
1757 \\boxed{40}
```

1755 The answer is 40

1757 **Example Problem:** At a twins and triplets convention, there were 9 sets of twins and 6 sets of triplets,
 1758 all from different families. Each twin shook hands with all the twins except his/her siblings and with
 1759 half the triplets. Each triplet shook hands with all the triplets except his/her siblings and with half the
 1760 twins. How many handshakes took place?

1761 **Example Solution:**

1762 Let's approach this problem step-by-step:

- 1764 1. Calculate the number of handshakes between twins
- 1765 2. Calculate the number of handshakes between triplets
- 1766 3. Calculate the number of handshakes between twins and triplets
- 1767 4. Sum up all handshakes

1769 Here's the Python code to solve this:

```
1770
1771 from sympy import Rational
1772 def calculate_handshakes():
1773     # Number of twins and triplets
1774     twins = 9 * 2
1775     triplets = 6 * 3
1776     # Handshakes between twins
1777     twin_handshakes = (twins * (twins - 2)) // 2 # Each twin shakes
1778     # Handshakes between triplets
1779     triplet_handshakes = (triplets * (triplets - 3)) // 2 # Each triplet
1780     # Handshakes between twins and triplets
1781     twin_triplet_handshakes = twins * triplets * Rational(1, 2) # Each
1782     # twin shakes hands with half the triplets
```

```

1782     # Total handshakes
1783     total_handshakes = twin_handshakes + triplet_handshakes +
1784     twin_triplet_handshakes
1785     return f"\boxed{{{int(total_handshakes)}}}"
1786 print(calculate_handshakes())
1787
1787 >>> output
1788 \boxed{441}
1789

```

The answer is 441

A.5 SBSC PROMPT FOR AMC

You are given a math problem and you need to think of a stepwise approach/process to be followed to solve the problem. Use sympy-based python code to codify each of these steps in sequential manner. You must end each of your code snippet with a print statement followed by *output* in the next line. Use the results from the output of the code snippets of each step and continue to the next step until you finish the final step and solve the problem completely.

In each new code block, you must define the variables and symbols again based on the previous code snippets or their outputs as suitable.

Use sympy in appropriate way, as and when required.

Once you complete the last step, write this line: `### END OF CODE`

Then just print the final numeric answer (without any units) like this: The final answer is boxed{ }.

If the executed code snippet returns an error, use it to correct the current step's code snippet. DO NOT restart solving from Step 1.

Here are a few examples for your reference.

Example Problem: Small lights are hung on a string 6 inches apart in the order red, red, green, green, green, red, red, green, green, green, and so on continuing this pattern of 2 red lights followed by 3 green lights. How many feet separate the 3rd red light and the 21st red light? Note: 1 foot is equal to 12 inches.

Example Solution: Let's solve this problem step by step using SymPy-based Python code.

Step 1: Define the pattern and calculate the number of lights in one complete pattern

```

1814 from sympy import symbols, Rational
1815 # Define the number of red and green lights in one pattern
1816 red_lights_per_pattern = 2
1817 green_lights_per_pattern = 3
1818 # Calculate total lights in one pattern
1819 lights_per_pattern = red_lights_per_pattern + green_lights_per_pattern
1820 print(f"Number of lights in one complete pattern: {lights_per_pattern}")
1821
1821 >>> output:
1822 Number of lights in one complete pattern: 5
1823

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.

Step 2: Calculate the number of complete patterns between the 3rd and 21st red lights

```

1826 from sympy import symbols, Rational
1827 # Define the number of red lights we're considering
1828 start_red_light = 3
1829 end_red_light = 21
1830 # Calculate the number of red lights between the 3rd and 21st (inclusive)
1831 red_lights_between = end_red_light - start_red_light + 1
1832 # Calculate the number of complete patterns
1833 complete_patterns = (red_lights_between - 1) // 2
1834 print(f"Number of complete patterns between the 3rd and 21st red lights:
1835     {complete_patterns}")
1836
1836 >>> output:
1837 Number of complete patterns between the 3rd and 21st red lights: 9
1838

```

1836 Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
 1837 Step 3: Calculate the total number of lights between the 3rd and 21st red lights

```
1838
1839 from sympy import symbols, Rational
1840 # Define the number of red and green lights in one pattern
1841 red_lights_per_pattern = 2
1842 green_lights_per_pattern = 3
1843 lights_per_pattern = red_lights_per_pattern + green_lights_per_pattern
1844 # Number of complete patterns
1845 complete_patterns = 9
1846 # Calculate total lights
1847 total_lights = complete_patterns * lights_per_pattern + 1 # +1 for the
1848 21st red light
1849 print(f"Total number of lights between the 3rd and 21st red lights (
1850 inclusive): {total_lights}")
1851
1852 >>> output:
1853 Total number of lights between the 3rd and 21st red lights (inclusive):
1854 46
```

1853 Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
 1854 Step 4: Calculate the total distance in inches

```
1855 from sympy import symbols, Rational
1856 # Total number of lights
1857 total_lights = 46
1858 # Distance between lights in inches
1859 distance_between_lights = 6
1860 # Calculate total distance in inches
1861 total_distance_inches = (total_lights - 1) * distance_between_lights
1862 print(f"Total distance in inches: {total_distance_inches}")
1863
1864 >>> output:
1865 Total distance in inches: 270
```

1865 Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
 1866 Step 5: Convert the distance from inches to feet

```
1867 from sympy import symbols, Rational
1868 # Total distance in inches
1869 total_distance_inches = 270
1870 # Convert inches to feet
1871 inches_per_foot = 12
1872 total_distance_feet = Rational(total_distance_inches, inches_per_foot)
1873 print(f"Total distance in feet: {total_distance_feet}")
1874
1875 >>> output:
1876 Total distance in feet: 45/2
```

1877 ### END OF CODE

1878 The final answer is $\frac{45}{2}$

1881 **Example Problem:** A fruit salad consists of blueberries, raspberries, grapes, and cherries. The fruit
 1882 salad has a total of 280 pieces of fruit. There are twice as many raspberries as blueberries, three times
 1883 as many grapes as cherries, and four times as many cherries as raspberries. How many cherries are
 1884 there in the fruit salad?

1885 **Example Solution:** Let's solve this problem step by step using SymPy-based Python code.

1886 Step 1: Define the variables and set up the equations

```
1887 from sympy import symbols, Eq, solve
1888 # Define symbols for each type of fruit
1889 b, r, g, c = symbols('b r g c')
1890 # Set up equations based on the given information
```

```

1890 eq1 = Eq(b + r + g + c, 280) # Total pieces of fruit
1891 eq2 = Eq(r, 2*b) # Twice as many raspberries as blueberries
1892 eq3 = Eq(g, 3*c) # Three times as many grapes as cherries
1893 eq4 = Eq(c, 4*r) # Four times as many cherries as raspberries
1894 print("Equations:")
1895 print(eq1)
1896 print(eq2)
1897 print(eq3)
1898 print(eq4)
1899
1900 >>> output:
1901 Equations:
1902 Eq(b + c + g + r, 280)
1903 Eq(r, 2*b)
1904 Eq(g, 3*c)
1905 Eq(c, 4*r)

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 2: Solve the system of equations

```

1907 from sympy import symbols, Eq, solve
1908 # Define symbols for each type of fruit
1909 b, r, g, c = symbols('b r g c')
1910 # Set up equations based on the given information
1911 eq1 = Eq(b + r + g + c, 280) # Total pieces of fruit
1912 eq2 = Eq(r, 2*b) # Twice as many raspberries as blueberries
1913 eq3 = Eq(g, 3*c) # Three times as many grapes as cherries
1914 eq4 = Eq(c, 4*r) # Four times as many cherries as raspberries
1915 # Solve the system of equations
1916 solution = solve((eq1, eq2, eq3, eq4))
1917 print("Solution:")
1918 print(solution)
1919
1920 >>> output:
1921 Solution:
1922 {b: 8, c: 64, g: 192, r: 16}

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 3: Extract the number of cherries from the solution

```

1923 from sympy import symbols, Eq, solve
1924 # Define symbols for each type of fruit
1925 b, r, g, c = symbols('b r g c')
1926 # Set up equations based on the given information
1927 eq1 = Eq(b + r + g + c, 280) # Total pieces of fruit
1928 eq2 = Eq(r, 2*b) # Twice as many raspberries as blueberries
1929 eq3 = Eq(g, 3*c) # Three times as many grapes as cherries
1930 eq4 = Eq(c, 4*r) # Four times as many cherries as raspberries
1931 # Solve the system of equations
1932 solution = solve((eq1, eq2, eq3, eq4))
1933 # Extract the number of cherries
1934 cherries = solution[c]
1935 print(f"The number of cherries in the fruit salad is: {cherries}")
1936
1937 >>> output:
1938 The number of cherries in the fruit salad is: 64

```

END OF CODE The final answer is 64

Example Problem: Last summer 30% of the birds living on Town Lake were geese, 25% were swans, 10% were herons, and 35% were ducks. What percent of the birds that were not swans were geese?

Example Solution: Let's solve this problem step by step using SymPy-based Python code.

Step 1: Define the percentages of each type of bird

```

1944 from sympy import symbols, Rational
1945 geese = Rational(30, 100)
1946 swans = Rational(25, 100)
1947 herons = Rational(10, 100)
1948 ducks = Rational(35, 100)
1949 print(f"Percentage of geese: {geese}")
1950 print(f"Percentage of swans: {swans}")
1951 print(f"Percentage of herons: {herons}")
1952 print(f"Percentage of ducks: {ducks}")
1953
1954 >>> output:
1955 Percentage of geese: 3/10
1956 Percentage of swans: 1/4
1957 Percentage of herons: 1/10
1958 Percentage of ducks: 7/20

```

Let's continue with the next steps to solve this problem. Step 2: Calculate the percentage of birds that are not swans

```

1960 from sympy import symbols, Rational
1961 geese = Rational(30, 100)
1962 swans = Rational(25, 100)
1963 herons = Rational(10, 100)
1964 ducks = Rational(35, 100)
1965 not_swans = 1 - swans
1966 print(f"Percentage of birds that are not swans: {not_swans}")
1967
1968 >>> output:
1969 Percentage of birds that are not swans: 3/4

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps. Step 3: Calculate the percentage of geese among the birds that are not swans

```

1972 from sympy import symbols, Rational
1973 geese = Rational(30, 100)
1974 swans = Rational(25, 100)
1975 herons = Rational(10, 100)
1976 ducks = Rational(35, 100)
1977 not_swans = 1 - swans
1978 geese_among_not_swans = geese / not_swans
1979 print(f"Percentage of geese among birds that are not swans: {
1980     geese_among_not_swans}")
1981
1982 >>> output:
1983 Percentage of geese among birds that are not swans: 2/5

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps. Step 4: Convert the fraction to a percentage

```

1986 from sympy import symbols, Rational
1987 geese = Rational(30, 100)
1988 swans = Rational(25, 100)
1989 herons = Rational(10, 100)
1990 ducks = Rational(35, 100)
1991 not_swans = 1 - swans
1992 geese_among_not_swans = geese / not_swans
1993 percentage = geese_among_not_swans * 100
1994 print(f"Percentage of geese among birds that are not swans: {percentage}%
1995     ")
1996
1997 >>> output:
1998 Percentage of geese among birds that are not swans: 40%

```

END OF CODE

1998 The final answer is 40

1999
2000 Example Problem: At a twins and triplets convention, there were 9 sets of twins and 6 sets of triplets,
2001 all from different families. Each twin shook hands with all the twins except his/her siblings and with
2002 half the triplets. Each triplet shook hands with all the triplets except his/her siblings and with half the
2003 twins. How many handshakes took place?

2004 **Example Solution:** Let's solve this problem step by step using SymPy-based Python code.

2005 Step 1: Define the given information

```
2006
2007 from sympy import symbols, Rational
2008 # Number of sets of twins and triplets
2009 twin_sets = 9
2010 triplet_sets = 6
2011 # Number of individuals
2012 twins = twin_sets * 2
2013 triplets = triplet_sets * 3
2014 print(f"Number of twins: {twins}")
2015 print(f"Number of triplets: {triplets}")
2016
2017 >>> output:
2018 Number of twins: 18
2019 Number of triplets: 18
```

2018 Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.

2019 Step 2: Calculate handshakes between twins

```
2020
2021 from sympy import symbols, Rational
2022 # Number of sets of twins and triplets
2023 twin_sets = 9
2024 triplet_sets = 6
2025 # Number of individuals
2026 twins = twin_sets * 2
2027 triplets = triplet_sets * 3
2028 # Handshakes between twins
2029 twin_handshakes = (twins * (twins - 2)) // 2 # Each twin shakes hands
2030 with all twins except self and sibling
2031 print(f"Handshakes between twins: {twin_handshakes}")
2032
2033 >>> output
2034 Handshakes between twins: 144
```

2033 Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.

2034 Step 3: Calculate handshakes between triplets

```
2035
2036 from sympy import symbols, Rational
2037 # Number of sets of twins and triplets
2038 twin_sets = 9
2039 triplet_sets = 6
2040 # Number of individuals
2041 twins = twin_sets * 2
2042 triplets = triplet_sets * 3
2043 # Handshakes between twins
2044 twin_handshakes = (twins * (twins - 2)) // 2
2045 # Handshakes between triplets
2046 triplet_handshakes = (triplets * (triplets - 3)) // 2 # Each triplet
2047 shakes hands with all triplets except self and two siblings
2048 print(f"Handshakes between triplets: {triplet_handshakes}")
2049
2050 >>> output
2051 Handshakes between triplets: 135
```

2050 Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.

2051 Step 4: Calculate handshakes between twins and triplets

```

2052 from sympy import symbols, Rational
2053 # Number of sets of twins and triplets
2054 twin_sets = 9
2055 triplet_sets = 6
2056 # Number of individuals
2057 twins = twin_sets * 2
2058 triplets = triplet_sets * 3
2059 # Handshakes between twins
2060 twin_handshakes = (twins * (twins - 2)) // 2
2061 # Handshakes between triplets
2062 triplet_handshakes = (triplets * (triplets - 3)) // 2
2063 # Handshakes between twins and triplets
2064 twin_triplet_handshakes = twins * (triplets // 2) # Each twin shakes
2065     hands with half the triplets
2066 print(f"Handshakes between twins and triplets: {twin_triplet_handshakes}"
2067     )
2068
2069 >>> output:
2070 Handshakes between twins and triplets: 162

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 5: Calculate the total number of handshakes

```

2071 from sympy import symbols, Rational
2072 # Number of sets of twins and triplets
2073 twin_sets = 9
2074 triplet_sets = 6
2075 # Number of individuals
2076 twins = twin_sets * 2
2077 triplets = triplet_sets * 3
2078 # Handshakes between twins
2079 twin_handshakes = (twins * (twins - 2)) // 2
2080 # Handshakes between triplets
2081 triplet_handshakes = (triplets * (triplets - 3)) // 2
2082 # Handshakes between twins and triplets
2083 twin_triplet_handshakes = twins * (triplets // 2)
2084 # Total handshakes
2085 total_handshakes = twin_handshakes + triplet_handshakes +
2086     twin_triplet_handshakes
2087 print(f"Total number of handshakes: {total_handshakes}")
2088
2089 >>> output
2090 Total number of handshakes: 441

```

END OF CODE

The final answer is 441

A.6 SBSC PROMPT FOR AIME

You are given a math problem and you need to think of a stepwise approach/process to be followed to solve the problem. Use sympy-based python code to codify each of these steps in sequential manner. You must end each of your code snippet with a print statement followed by *output* in the next line. Use the results from the output of the code snippets of each step and continue to the next step until you finish the final step and solve the problem completely.

In each new code block, you must define the variables and symbols again based on the previous code snippets or their outputs as suitable.

Use sympy in appropriate way, as and when required.

Once you complete the last step, write this line: ### END OF CODE

Then just print the final numeric answer (without any units) like this: The final answer is boxed{ }.

If the executed code snippet returns an error, use it to correct the current step's code snippet. DO NOT restart solving from Step 1.

Here are a few examples for your reference.

2106 **Example Problem:** A frog begins at $P_0 = (0, 0)$ and makes a sequence of jumps according to
 2107 the following rule: from $P_n = (x_n, y_n)$, the frog jumps to P_{n+1} , which may be any of the points
 2108 $(x_n + 7, y_n + 2)$, $(x_n + 2, y_n + 7)$, $(x_n - 5, y_n - 10)$, or $(x_n - 10, y_n - 5)$. There are M points
 2109 (x, y) with $|x| + |y| \leq 100$ that can be reached by a sequence of such jumps. Find the remainder
 2110 when M is divided by 1000.

2111 **Example Solution:** Let's approach this problem step by step using SymPy-based Python code.

2112
 2113 Step 1: Import necessary libraries and define the jump rules

```
2114 from sympy import symbols, Abs
2115 import numpy as np
2116 # Define the possible jumps from any position (x, y)
2117 def jumps(x, y):
2118     return [
2119         (x + 7, y + 2), # Jump by (7, 2)
2120         (x + 2, y + 7), # Jump by (2, 7)
2121         (x - 5, y - 10), # Jump by (-5, -10)
2122         (x - 10, y - 5) # Jump by (-10, -5)
2123     ]
2124 print("Jump rules defined.")
```

```
2124 >>> output
2125 Jump rules defined.
```

2126
 2127 Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.

2128 Step 2: Implement a breadth-first search to find all reachable points

```
2129 from sympy import symbols, Abs
2130 import numpy as np
2131 from collections import deque
2132 # Reuse the jumps function to define possible moves
2133 def jumps(x, y):
2134     return [
2135         (x + 7, y + 2),
2136         (x + 2, y + 7),
2137         (x - 5, y - 10),
2138         (x - 10, y - 5)
2139     ]
2140 # Define the breadth-first search (BFS) function
2141 def bfs():
2142     queue = deque([(0, 0)]) # Start BFS from the point (0, 0)
2143     visited = set([(0, 0)]) # Keep track of visited points to avoid
2144     reprocessing
2145     # Process each point in the queue
2146     while queue:
2147         x, y = queue.popleft()
2148         for nx, ny in jumps(x, y): # Try all possible jumps from the
2149             current point
2150                 if (nx, ny) not in visited and abs(nx) + abs(ny) <= 100:
2151                     visited.add((nx, ny)) # Mark the new point as visited
2152                     queue.append((nx, ny)) # Add the new point to the queue
2153     to explore further
2154     return len(visited)
2155 M = bfs()
2156 print(f"Number of reachable points (M): {M}")
```

```
2153 >>> output:
2154 Number of reachable points (M): 1373
```

2155
 2156 Let's continue with the final step to solve this problem. Step 3: Calculate the remainder when M is
 2157 divided by 1000

```
2158 from sympy import symbols, Abs
2159 import numpy as np
2160 M = 1373 # M_value from previous step
```



```

2160 # Find the remainder when M is divided by 1000
2161 remainder = M % 1000
2162 print(f"Remainder when M is divided by 1000: {remainder}")
2163
2164 >>> output
2165 Remainder when M is divided by 1000: 373

```

END OF CODE

The final answer is 373

Example Problem: The AIME Triathlon consists of a half-mile swim, a 30-mile bicycle ride, and an eight-mile run. Tom swims, bicycles, and runs at constant rates. He runs five times as fast as he swims, and he bicycles twice as fast as he runs. Tom completes the AIME Triathlon in four and a quarter hours. How many minutes does he spend bicycling?

Example Solution: Let's solve this problem step by step using SymPy-based Python code.

Step 1: Define the variables and set up the basic equations

```

2176 from sympy import symbols, Eq, solve
2177 # Define variables
2178 s, b, r = symbols('s b r') # s: swim speed, b: bike speed, r: run speed
2179 # Set up equations based on the given information
2180 eq1 = Eq(r, 5*s) # Tom runs 5 times as fast as he swims
2181 eq2 = Eq(b, 2*r) # Tom bicycles twice as fast as he runs
2182 print("Equation 1:", eq1)
2183 print("Equation 2:", eq2)
2184
2185 >>> output
2186 Equation 1: Eq(r, 5*s)
2187 Equation 2: Eq(b, 2*r)

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.

Step 2: Set up the equation for the total time

```

2190 from sympy import symbols, Eq, solve
2191 # Reuse earlier code
2192 s, b, r = symbols('s b r')
2193 eq1 = Eq(r, 5*s)
2194 eq2 = Eq(b, 2*r)
2195 # Total time equation (in hours)
2196 total_time = 0.5/s + 30/b + 8/r #The AIME Triathlon consists of a half-
2197     mile swim, a 30-mile bicycle ride, and an eight-mile run.
2198 eq3 = Eq(total_time, 4.25) # Total time is 4.25 hours
2199 print("Total time equation:", eq3)
2200
2201 >>> output:
2202 Total time equation: Eq(0.5/s + 8/r + 30/b, 4.25)

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.

Step 3: Substitute the relationships between speeds into the total time equation

```

2205 from sympy import symbols, Eq, solve, simplify
2206 # Reuse earlier code
2207 s, b, r = symbols('s b r')
2208 eq1 = Eq(r, 5*s)
2209 eq2 = Eq(b, 2*r)
2210 total_time = 0.5/s + 30/b + 8/r
2211 eq3 = Eq(total_time, 4.25) # Total time is 4.25 hours
2212 # Substitute r and b in terms of s to simplify the equation
2213 eq3_substituted = eq3.subs([(r, 5*s), (b, 10*s)])
2214 eq3_simplified = simplify(eq3_substituted)
2215 print("Simplified equation:", eq3_simplified)

```

```
2214 >>> output:
2215 Simplified equation: Eq(5.1/s, 4.25)
2216
```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 4: Solve for the swim speed (s)

```
2219 from sympy import symbols, Eq, solve, simplify
2220 # Redefine variables
2221 s, b, r = symbols('s b r') # s: swim speed, b: bike speed, r: run speed
2222 # Simplified equation from previous step
2223 eq3_simplified = Eq(5.1/s, 4.25)
2224 # Solve for s
2225 s_solution = solve(eq3_simplified, s)
2226 print("Swim speed (s):", s_solution[0])
2227
2228 >>> output
2229 Swim speed (s): 1.2000000000000000
```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 5: Calculate the bike speed (b)

```
2231 from sympy import symbols, Eq, solve, simplify
2232 # Redefine variables
2233 s, b, r = symbols('s b r') # s: swim speed, b: bike speed, r: run speed
2234 # Swim speed from previous step
2235 s_value = 1.2000000000000000
2236 # Calculate bike speed
2237 b_value = 10 * s_value # b = 2r = 2(5s) = 10s
2238 print("Bike speed (b):", b_value)
2239
2240 >>> output
2241 Bike speed (b): 12.0
```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 6: Calculate the time spent bicycling

```
2244 from sympy import symbols, Eq, solve, simplify
2245 # Redefine variables
2246 s, b, r = symbols('s b r') # s: swim speed, b: bike speed, r: run speed
2247 # Bike speed from previous step
2248 b_value = 12.0
2249 # Calculate time spent bicycling
2250 bike_distance = 30 # miles
2251 bike_time = bike_distance / b_value # hours
2252 # Convert to minutes
2253 bike_time_minutes = bike_time * 60
2254 print("Time spent bicycling (in minutes):", bike_time_minutes)
2255
2256 >>> output
2257 Time spent bicycling (in minutes): 150.0
```

END OF CODE

The final answer is 150

Example Problem: Let S be the increasing sequence of positive integers whose binary representation has exactly 8 ones. Let N be the 1000th number in S . Find the remainder when N is divided by 1000

Example Solution: Let's solve this problem step by step using SymPy-based Python code.

Step 1: Import necessary libraries and define a function to count ones in binary representation

```
2265 from sympy import symbols, binomial
2266 import math
2267 # Function to count the number of '1's in the binary representation of a
    number
```

```

2268 def count_ones(n):
2269     return bin(n).count('1') # Convert n to binary and count occurrences
2270     of '1'
2271 print("Function to count ones in binary representation defined.")
2272
2273 >>> output:
2274 Function to count ones in binary representation defined.

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 2: Find the smallest number in the sequence S

```

2275 from sympy import symbols, binomial
2276 import math
2277 # Reuse the function to count the number of '1's in binary
2278 def count_ones(n):
2279     return bin(n).count('1')
2280 # The smallest number in the sequence S is represented by 8 consecutive
2281 '1's in binary
2282 smallest_number = int('1' * 8, 2) # Convert '11111111' (binary) to
2283 decimal
2284 print(f"The smallest number in sequence S is: {smallest_number}")
2285
2286 >>> output
2287 The smallest number in sequence S is: 255

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 3: Find the largest number in the sequence S

```

2292 from sympy import symbols, binomial
2293 import math
2294 # Reuse the function to count the number of '1's in binary
2295 def count_ones(n):
2296     return bin(n).count('1')
2297 smallest_number = int('1' * 8, 2)
2298 # The largest number in sequence S is represented by 8 consecutive '1's
2299 followed by 24 '0's
2300 largest_number = int('1' * 8 + '0' * 24, 2) # Convert
2301 '11111111000000000000000000000000' to decimal
2302 print(f"The largest number in sequence S is: {largest_number}")
2303
2304 >>> output
2305 The largest number in sequence S is: 4278190080

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 4: Calculate the total number of elements in sequence S

```

2307 from sympy import symbols, binomial
2308 import math
2309 def count_ones(n):
2310     return bin(n).count('1')
2311 smallest_number = int('1' * 8, 2)
2312 largest_number = int('1' * 8 + '0' * 24, 2)
2313 # Calculate the total number of elements in sequence S using binomial(32,
2314 8)
2315 # This represents the number of ways to choose 8 positions out of 32 bits
2316 to set to '1'
2317 total_elements = binomial(32, 8)
2318 print(f"The total number of elements in sequence S is: {total_elements}")
2319
2320 >>> output
2321 The total number of elements in sequence S is: 10518300

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 5: Find the 1000th number in sequence S

```

2322 from sympy import symbols, binomial
2323 import math
2324 #Reuse all earlier code
2325 def count_ones(n):
2326     return bin(n).count('1')
2327 smallest_number = int('1' * 8, 2)
2328 largest_number = int('1' * 8 + '0' * 24, 2)
2329 total_elements = binomial(32, 8)
2330 # Find the 1000th number in the sequence
2331 count = 0 # Initialize a counter for the sequence
2332 N = smallest_number # Start from the smallest number in S
2333 # Iterate through numbers and check for 8 '1's in the binary
2334 # representation
2335 while count < 1000:
2336     if count_ones(N) == 8: # If the current number has exactly 8 '1's
2337         count += 1 # Increment the count
2338     if count == 1000: # Stop if we've found the 1000th number
2339         break
2340     N += 1 # Move to the next number
2341 print(f"The 1000th number (N) in sequence S is: {N}")
2342
2343 >>> output
2344 The 1000th number (N) in sequence S is: 7032

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 6: Calculate the remainder when N is divided by 1000

```

2345 from sympy import symbols, binomial
2346 import math
2347 def count_ones(n):
2348     return bin(n).count('1')
2349 N = 7032 # From the previous step
2350 # Calculate the remainder when N is divided by 1000
2351 remainder = N % 1000
2352 print(f"The remainder when N is divided by 1000 is: {remainder}")
2353
2354 >>> output
2355 The remainder when N is divided by 1000 is: 32

```

END OF CODE

The final answer is 32

Example Problem: Two geometric sequences a_1, a_2, a_3, \dots and b_1, b_2, b_3, \dots have the same common ratio, with $a_1 = 27$, $b_1 = 99$, and $a_{15} = b_{11}$. Find a_9

Example Solution: Let's solve this problem step by step using SymPy-based Python code.

Step 1: Define the variables and set up the initial conditions

```

2363 from sympy import symbols, Eq, solve
2364 # Define symbols
2365 r = symbols('r') # common ratio
2366 a1, b1 = 27, 99 # first terms of sequences a and b
2367 print(f"a1 = {a1}, b1 = {b1}")
2368 print("Common ratio is represented by r")
2369
2370 >>> output
2371 a1 = 27, b1 = 99
2372 Common ratio is represented by r

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 2: Set up the equation for $a_{15} = b_{11}$

```

2375 from sympy import symbols, Eq, solve
2376 # Redefine symbols

```

```

2376 r = symbols('r')
2377 a1, b1 = 27, 99
2378 # Set up the equation a15 = b11
2379 eq = Eq(a1 * r**14, b1 * r**10)
2380 print("Equation: a15 = b11")
2381 print(f"Symbolically: {eq}")
2382
2382 >>> output
2383 Equation: a15 = b11
2384 Symbolically: Eq(27*r**14, 99*r**10)

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 3: Solve the equation for r

```

2388 from sympy import symbols, Eq, solve
2389 # Redefine symbols
2390 r = symbols('r')
2391 a1, b1 = 27, 99
2392 # Set up the equation a15 = b11
2393 eq = Eq(a1 * r**14, b1 * r**10)
2394 # Solve for r
2395 solution = solve(eq, r)
2396 print("Solutions for r:")
2397 print(solution)
2398
2398 >>> output
2399 Solutions for r:
2400 [0, -11**(1/4)*3**(3/4)/3, 11**(1/4)*3**(3/4)/3]

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 4: Identify the correct value for r

```

2403 from sympy import symbols, Eq, solve, N
2404 # Reuse earlier code
2405 r = symbols('r')
2406 a1, b1 = 27, 99
2407 eq = Eq(a1 * r**14, b1 * r**10)
2408 solution = solve(eq, r)
2409 # The correct value for r is the positive real solution
2410 r_value = max(sol for sol in solution if sol.is_real)
2411 print(f"The common ratio r is: {N(r_value, 6)}")
2412
2412 >>> output
2413 The common ratio r is: 1.38378

```

Let's continue with the next steps to solve this problem and leveraging the outputs of previous steps.
Step 5: Calculate a9

```

2416 from sympy import symbols, Eq, solve, N
2417 # Reuse earlier code
2418 r = symbols('r')
2419 a1, b1 = 27, 99
2420 eq = Eq(a1 * r**14, b1 * r**10)
2421 solution = solve(eq, r)
2422 r_value = max(sol for sol in solution if sol.is_real)
2423 # Calculate a9
2424 a9 = a1 * r_value**8
2425 print(f"a9 = {N(a9, 10)}")
2426
2426 >>> output
2427 a9 = 363.0000000

```

END OF CODE

The final answer is 363

A.7 RESULTS WITH STANDARD DEVIATIONS

We present the main table results along with standard deviation scores.

Table 4: Benchmarking SBSC against different math reasoning methods across three datasets. We report average accuracy over 3 runs with standard deviation within parentheses. Best result in each setting is highlighted in **bold** and second best is underlined. Absolute improvement in performance by SBSC over the previous best method in each setting is indicated in subscript.

Method	AMC		AIME		MathOdyssey		Olympiad Bench	
	greedy	maj@7	greedy	maj@7	greedy	maj@7	greedy	maj@7
Claude-3.5-Sonnet								
COT	31.16 (± 1.0)	35.79	9.09 (± 1.0)	10.91	11.89 (± 0.6)	16.89	39.35 (± 0.47)	42.46
PAL	35.79 (± 1.0)	36.42	<u>27.48</u> (± 0.6)	<u>28.79</u>	27.23 (± 0.6)	31.01	41.07 (± 0.82)	44.44
TIR-ToRA	<u>38.59</u> (± 0.6)	<u>43.16</u>	24.64 (± 3.2)	26.67	<u>27.23</u> (± 0.6)	<u>32.43</u>	47.69 (± 0.47)	<u>50.60</u>
SBSC (Ours)	49.33 (± 3.1) _{$\uparrow 10.7$}	$\uparrow 6.2$	35.45 (± 1.7) _{$\uparrow 8$}	$\uparrow 6.7$	39.86 (± 1.0) _{$\uparrow 12.6$}	$\uparrow 7.4$	53.31 (± 0.94) _{$\uparrow 5.6$}	$\uparrow 2.7$
GPT-4o								
COT	35.94 (± 0.6)	37.47	10.39 (± 2.1)	12.12	13.51 (± 1.0)	17.57	41.80 (± 1.89)	47.22
PAL	36.48 (± 0.6)	38.11	<u>24.63</u> (± 0.6)	<u>26.97</u>	15.74 (± 0.6)	20.27	41.67 (± 2.16)	46.43
TIR-ToRA	<u>37.33</u> (± 2.5)	<u>40.42</u>	22.42 (± 1.7)	25.45	<u>19.59</u> (± 2.6)	<u>23.64</u>	43.32 (± 1.70)	49.61
SBSC (Ours)	44.55 (± 0.6) _{$\uparrow 7.2$}	$\uparrow 4.1$	30.7 (± 1.1) _{$\uparrow 6.1$}	$\uparrow 3.7$	26.55 (± 1.1) _{$\uparrow 7$}	$\uparrow 2.9$	48.74 (± 1.89) _{$\uparrow 5.4$}	$\uparrow 0.87$

A.8 LEAST-TO-MOST PROMPTING

Least-to-Most (L2M) (Zhou et al., 2022) is a two-stage prompting strategy where the aim is: in first stage, to break down a complex problem into a series of simpler subproblems and then, in second stage, solve these predefined subproblems. PAL (Gao et al., 2022) reported a L2M version of PAL in their work. We follow the reported prompts and replicate it by designing exemplars for both the stages. We find L2M-PAL inherits the same issues that PAL / TIR-TORA has. L2M-PAL comes up with entire sub-problems at once and also its uses single program-block to solve those sub-problems. SBSC dynamically generates the next sub-task and the corresponding program to solve it leveraging the previous turns results. In Table 5, we show the results obtained from L2M + PAL using Claude-3.5-Sonnet on our AMC and AIME test datasets. Even after allowing self-correction for stage 2 with max turns $n=15$, L2M-PAL approaches PAL scores. Hence for our main results, we stick to PAL & TIR-ToRA along with self-consistency (Shao et al., 2024) due to resource optimisation and widee adaption of those prompting strategies for math-problem solving.

Table 5: Least-to-Most Prompting results on AIME and AMC

Method	AMC		AIME	
	greedy	maj@7	greedy	maj@7
COT	31.16	35.79	9.09	10.91
PAL	35.79	36.42	<u>27.48</u>	<u>28.79</u>
L2M-PAL (n=1)	33.47	38.53	25.45	<u>28.79</u>
L2M-PAL (n=15)	34.32		25.45	
TIR-ToRA	<u>38.59</u>	<u>43.16</u>	24.64	26.67
SBSC (Ours)	49.33 _{$\uparrow 10.7$}	$\uparrow 6.2$	35.45 _{$\uparrow 8$}	$\uparrow 6.7$