

SCALING TEST-TIME COMPUTE OPTIMALLY CAN BE MORE EFFECTIVE THAN SCALING LLM PARAMETERS

Anonymous authors

Paper under double-blind review

ABSTRACT

Enabling LLMs to improve their outputs by using more test-time computation is a critical step towards building generally self-improving agents that can operate on open-ended natural language. In this paper, we scale up inference-time computation in LLMs, with a focus on answering: *if an LLM is allowed to use a fixed but non-trivial amount of inference-time compute, how much can it improve its performance on a challenging prompt?* Answering this question has implications not only on the achievable performance of LLMs, but also on the future of LLM pretraining and how to tradeoff inference-time and pre-training compute. Little research has attempted to understand the scaling behaviors of test-time inference methods, with current work largely providing negative results for a number of these strategies. In this work, we analyze two primary mechanisms to scale test-time computation: (1) searching against dense, process-based verifier reward models; and (2) updating the model’s distribution over a response adaptively, given the prompt at test time. We find that in both cases, the effectiveness of different approaches to scaling test-time compute critically varies depending on the difficulty of the prompt. This observation motivates applying a “compute-optimal” scaling strategy, which acts to most effectively allocate test-time compute adaptively per prompt. Using this compute-optimal strategy, we can improve the efficiency of test-time compute scaling by more than $4\times$ compared to a best-of-N baseline. Additionally, in a FLOPs-matched evaluation, we find that on problems where a smaller base model attains somewhat non-trivial success rates, test-time compute can be used to outperform a $14\times$ larger model.

1 INTRODUCTION

Given a challenging input, *can we enable LLMs to most effectively make use of additional computation at test-time to improve their responses?* In theory, additional test-time compute should enable an LLM to do better than what it was trained to do in zero-shot. Such a capability at test-time bears the potential to unlock agentic and reasoning abilities (Shinn et al., 2023; Qu et al., 2024b). Additionally, if pre-trained model size can be traded off for additional inference compute, this would enable the deployment of smaller on-device models in place of datacenter scale LLMs. Automating the inference-time improvement of model outputs also provides a path towards a general self-improvement algorithm that can function with reduced human supervision.

Prior work studying inference-time computation provides mixed results. On the one hand, some prior work shows that current LLMs can use test-time computation to improve their outputs (Bai et al., 2022; Madaan et al., 2023; Du et al., 2023; Saunders et al., 2022; Yao et al., 2023), on the other hand, several other works show that the effectiveness of these methods on complex tasks such as math reasoning remains limited (Huang et al., 2023; Stechly et al., 2023; Valmeekam et al., 2023; Wang et al., 2024a). However, reasoning is a domain we should expect to benefit from test-time compute, since reasoning involves drawing inferences from existing knowledge as opposed to acquiring new knowledge. Therefore, the disagreement in these prior findings motivates the need for a systematic analysis of the effectiveness of different approaches for scaling test-time compute.

In this paper we understand the pros and cons of scaling up test-time compute, and how it compares with scaling up pre-training compute. While the simplest approach for scaling test-time compute is *best-of-N* sampling – sampling N outputs in “parallel” from a base LLM and selecting the one that scores the highest per a learned verifier or a reward model (Cobbe et al., 2021; Lightman et al., 2023)

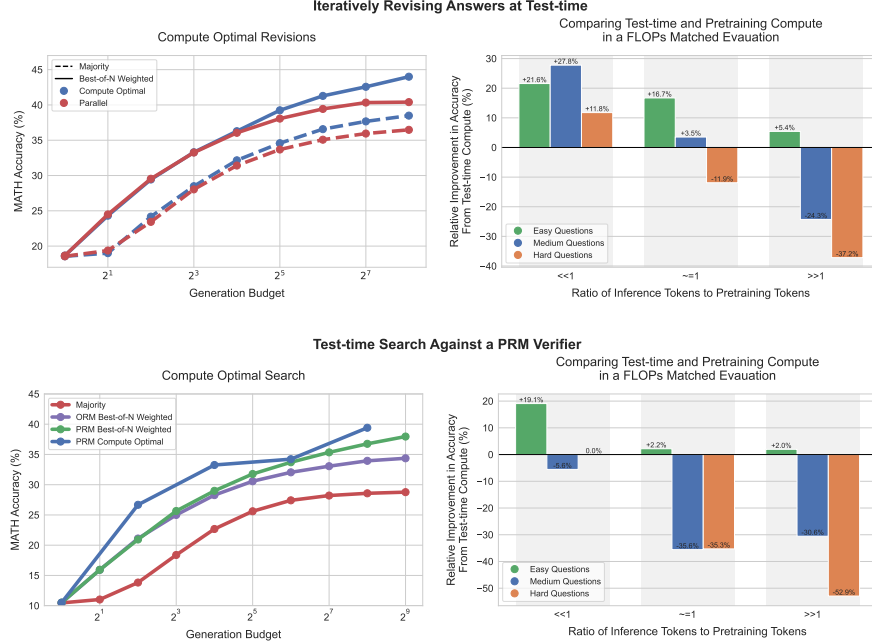


Figure 1: **Summary of results. Left: Compute-optimal scaling for revisions and search.** We compare the compute-optimal scaling policy with PaLM 2-S* against baselines in the revision (top) and PRM search (bottom) settings. **In the revision setting, we find that compute-optimal scaling outperforms best-of-N with $4\times$ less compute.** Similarly, with PRM search, compute-optimal scaling shows large early improvements over best-of-N, nearly outperforming best-of-N with $4\times$ less compute at points (see Sec. 5 and 6). **Right: Comparing test-time compute and parameter scaling.** We compare compute-optimal test-time scaling with PaLM 2-S* against a $\sim 14\times$ larger model without additional test-time compute. We expect X tokens of pretraining for both models and Y tokens of inference. A larger model, multiplies the FLOPs for both. If we were to apply additional test-time compute to the smaller model, to match this FLOPs multiplier, we see that for the revisions (top) when $Y \ll X$, test-time compute is preferable to pretraining. Additionally, as the inference to pretraining ratio increases, test-time compute remains preferable on easy questions. However, on hard questions, pretraining is preferable in this setting. We see a similar trend with PRM search (bottom, see Sec. 7 for details). – there are many other ways we could conceivably scale up test-time compute. We unify methods into those that modify either the *proposal distribution* from which responses are sampled (for e.g., by asking the base model to revise its responses (Qu et al., 2024b)) or those that alter how the *verifier* is used for searching directly in the output space (e.g. by training a PRM (Lightman et al., 2023)).

To understand the benefits of scaling up test-time compute, we carry out experiments on MATH (Hendrycks et al., 2021) using PaLM-2 (Anil et al., 2023) models fine-tuned to either revise incorrect answers (Qu et al., 2024b) (e.g. improving the proposal distribution; Section 6) or verify the correctness of individual steps in an answer using a process-based reward model (PRM) (Lightman et al., 2023; Wang et al., 2023) (Section 5). We find that the efficacy of different test-time strategies depends on both the nature of the specific problem at hand and the base LLM used. For example, on “easier” problems, for which the base LLM can already produce reasonable-looking responses, allowing the model to sequentially revise its initial answer (i.e., modifying the proposal distribution) is a more effective use of compute than reranking N independent answers sampled in parallel. On the other hand, on difficult problems which require searching over many high-level strategies, re-sampling new responses independently in parallel or deploying tree-search against a process reward model is more effective. This underscores the need to deploy an **adaptive “compute-optimal”** strategy for scaling test-time compute, wherein the specific approach for utilizing test-time compute is selected depending on the prompt, so as to make the best use of additional computation. We also show that a notion of *question difficulty* (Section 4) from the perspective of the base LLM can be used to predict the efficacy of test-time computation, enabling us to practically instantiate this ‘compute-optimal’ scaling. By appropriately allocating test-time compute in this way, we are able to greatly improve test-time compute scaling, surpassing the performance of a best-of-N baseline while only using $\sim 4\times$ less computation with both revisions and search (Sections 5 and 6).

Using our scaling strategy, we then study to what extent test-time computation can substitute for additional pretraining. Specifically, we conduct a **FLOPs-matched** comparison between a smaller

model with test-time compute and pretraining a $14\times$ larger model. We find that on easy and intermediate questions, additional test-time compute is often preferable to scaling pretraining. This finding suggests that rather than focusing purely on scaling pretraining, *in some settings it is more efficient to pretrain smaller models with less compute, and then apply test-time compute to improve outputs*. That said, with the most challenging questions, we observe few benefits from scaling up test-time compute. Instead, on these questions, it is more effective to make progress by applying additional pretraining, demonstrating that current approaches to scaling test-time compute may not be 1-to-1 exchangeable with scaling pretraining. Overall, this suggests that even with a fairly naïve methodology, scaling up test-time computation can already be preferable to pretraining *in some settings*, with only more improvements as test-time strategies mature. Longer term, this hints at a future where fewer FLOPs are spent during pretraining and more FLOPs are spent at inference.

2 UNIFIED PERSPECTIVE ON TEST-TIME COMPUTE: PROPOSER & VERIFIER

We first provide a unified abstraction of test-time compute to situate contemporary approaches. We view the use of test-time compute through the lens of modifying the model’s distribution on a given prompt, *adaptively* at test-time. Ideally, test-time compute should allow for the ability to express more complex distributions than naïvely sampling from the LLM. In general, there are two knobs to modify an LLM’s distribution: **(1) at the input level**: by augmenting the given prompt with an additional set of tokens that the LLM conditions on to obtain a modified **proposal distribution**, or **(2) at the output level**: by sampling multiple candidates from the standard LLM and performing surgery on them, using some post-hoc **verifiers or scorers**. This process is reminiscent of Markov chain Monte Carlo (MCMC) (Andrieu et al., 2003) sampling from a complex distribution by combining a simple proposal distribution and a score function. Modifying the proposal distribution by altering inputs tokens and using a verifier form the two independent axes of our study.

(1) Modifying the proposal distribution. One way to improve the proposal distribution is to directly optimize the model for a given reasoning task via RL-inspired finetuning methods such as STaR or ReST^{EM} (Zelikman et al., 2022; Singh et al., 2024). These techniques specifically finetune the model to directly improve the proposal distribution, rather than generating additional tokens at test-time. Instead, techniques such as self-critique (Bai et al., 2022; Madaan et al., 2023; Du et al., 2023; Saunders et al., 2022) enable the model to improve its own proposals at test time by instructing it to critique and revise its outputs iteratively. Since prompting off-the-shelf models is not effective at enabling effective revisions at test time, we specifically finetune models to iteratively revise their answers for complex reasoning, using Best-of-N guidance (Qu et al., 2024b; Kumar et al., 2024).

(2) Optimizing the verifier. The verifier selects the best answer from the proposal distribution. The most canonical way to use such a verifier is by applying best-of-N sampling, wherein we sample N solutions and then select the best one with a verifier (Cobbe et al., 2021). This approach can be further improved by training a process-based reward model (PRM) (Lightman et al., 2023), which produces a prediction of the correctness of each intermediate step in a solution. We can then utilize these per-step predictions to perform tree search over the solution space, enabling a more effective modification of the proposal distribution (Yao et al., 2023; Feng et al., 2024; Chen et al., 2024).

3 HOW TO SCALE TEST-TIME COMPUTATION OPTIMALLY

Using this unified view of different methods, we would like to understand and characterize how to *most effectively* use test-time computation to improve performance on a given prompt by answering the question below. When either refining the proposal distribution or searching against a verifier, there are numerous choices on how to allocate test-time compute. For example, when using a model finetuned for revisions as the proposal distribution and an ORM verifier, we could either spend the full test-time budget on generating N independent samples in parallel from the model and then apply best-of-N, or we could sample N revisions in sequence using a revision model and then select the best answer in the sequence with an ORM, or strike a balance between these extremes. Intuitively, we might expect that problems where the initial samples are more likely to be on the right track to benefit more from revisions. On the other hand, problems that require exploration over high-level problem solving strategies might benefit from sampling more independent answers in parallel. Finally, in the case of verifiers, we also can choose between different search algorithms (e.g. beam-search, lookahead-search, best-of-N), each of which may exhibit different properties depending on the quality of the verifier and proposal distribution at hand. More sophisticated search procedures might be more useful in harder problems compared to a much simpler best-of-N or majority baseline.

Problem setup

We are given a prompt and a test-time compute budget within which to solve the problem. Under the abstraction above, there are different knobs we can tune when utilizing test-time computation. How can we determine the *most effective* way to utilize test-time compute for a given prompt? And how well would this do against simply utilizing a much bigger pretrained model?

3.1 COMPUTE-OPTIMAL TEST-TIME SCALING STRATEGY

Per the discussion above, we would like to prescribe the *optimal* allocation of our test-time compute budget onto a given problem. To this end, for any given approach of utilizing test-time compute (e.g., revisions and search against a verifier in this paper, some combination or other methods in general), we define the “**test-time compute-optimal scaling strategy**” as the strategy that chooses hyperparameters appearing in a given approach for maximal performance benefits on a given prompt at test time. Formally, define $\text{Target}(\theta, N, q)$ as the distribution over natural language output tokens induced by the model for a given prompt q , using test-time compute hyper-parameters θ , and a compute budget of N . We would like to select the hyper-parameters θ which maximize the accuracy of the target distribution for a given problem. We express this formally as:

$$\theta_{q,a^*}^*(N) = \operatorname{argmax}_{\theta} \left(\mathbb{E}_{y \sim \text{Target}(\theta, N, q)} [\mathbb{1}_{y=y^*(q)}] \right), \quad (1)$$

where $y^*(q)$ denotes the ground-truth correct response for input query q , and $\theta_{q,y^*(q)}^*(N)$ represents the test-time compute-optimal scaling strategy for the problem q with compute budget N . We note that our definition of test-time compute-optimal scaling differs slightly from that of concurrent work (Wu et al., 2024) in that our notion of scaling is question dependent.

3.2 QUESTION DIFFICULTY IS A GOOD APPROXIMATION FOR THE OPTIMAL STRATEGY

In order to effectively analyze the test-time scaling properties of the different mechanisms discussed in Section 2 (e.g. proposal distribution and verifier), we will prescribe an approximation to this optimal strategy $\theta_{q,y^*(q)}^*(N)$ as a function of a statistic of a given prompt. Our approximation estimates a notion of **difficulty** for a given prompt. The compute-optimal strategy is then defined as a function of the difficulty of a prompt. Despite being only an heuristic approach to solve Equation 1, we find that it can still induce substantial improvements in performance over a baseline strategy of allocating this inference-time compute in an ad-hoc manner.

Our estimate of question difficulty assigns a given question to one of five discrete difficulty levels. We then use these bins to estimate $\theta_{q,y^*(q)}^*(N)$ on a validation set (given a compute budget), and apply the optimal strategy on the test set. Thus, question difficulty acts as a sufficient statistic for designing the compute-optimal strategy. [For example, to optimally allocate test-time compute between parallel best-of-N and sequential sampling, we first pre-compute the accuracy of both techniques within each difficulty bin using a held-out set. Given a new test question, we then determine the difficulty bin it belongs to and select the best performing strategy within that bin.](#)

Defining question difficulty. Following Lightman et al. (2023), we define question difficulty as a function of the given base LLM. Specifically, we bin the model’s pass@1 rate – estimated from 2048 samples – on each test question into five quantiles, each corresponding to increasing difficulty levels. We find this notion of model-specific difficulty bins to be more predictive of the efficacy of using test-time compute compared to the hand-labeled difficulty bins in the MATH dataset.

That said, we note that assessing difficulty as described assumes oracle access to a correctness checker, which is unavailable at deployment. To enable a realistic estimate of difficulty, we approximate difficulty via a **model-predicted notion of difficulty**, which constructs the bins by averaging the score of a learned verifier on the same 2048 samples per problem. We refer to this setting as **model-predicted difficulty** and the setting which relies on ground-truth correctness as **oracle difficulty**. Predicted difficulty removes the reliance on ground truth labels, but still incurs computational cost. Our experiments do not account for this cost largely for simplicity, since our goal is to present some of the first results of *what is in fact possible* by effectively allocating test-time compute.

4 EXPERIMENTAL SETUP

We first outline our experimental setup for conducting this analysis with multiple verifier design choices and proposal distributions, followed by the analysis results in the subsequent sections.

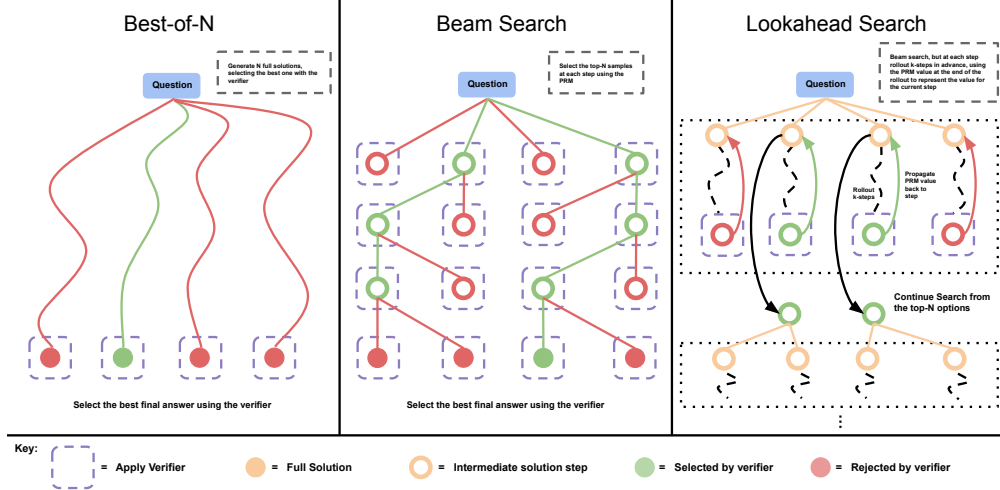


Figure 2: **Comparing different PRM search methods.** **Left:** Best-of-N samples N full answers and then selects the best answer according to the PRM final score. **Center:** Beam search samples N candidates at each step, and selects the top M according to the PRM to continue the search from. **Right:** lookahead-search extends each step in beam-search to utilize a k-step lookahead while assessing which steps to retain and continue the search from. Thus lookahead-search needs more compute.

Datasets. We expect test-time compute to be most helpful when models already have all the basic “knowledge” needed to answer a query, and instead the primary challenge is about drawing (complex) inferences from this knowledge. To this end, we focus on the MATH (Hendrycks et al., 2021) benchmark, which consists of high-school competition level math problems with a range of difficulty levels. For all experiments, we use the dataset split consisting of 12k train and 500 test questions.

Models. We use the PaLM 2-S* (Anil et al., 2023) (Codey) model. We chose this model, as it is representative of the capabilities of many contemporary LLMs, and is small enough to efficiently run many experiments on. Most importantly, this model attains a non-trivial performance on MATH (but not saturated). For these reasons, we expect this model to provide a good test-bed.

5 SCALING TEST-TIME COMPUTE VIA VERIFIERS

In this section, we study how test-time compute can be most effectively scaled by searching against a verifier and keeping the proposal distribution fixed to the base LM. Specifically, we study different search approaches with PRMs and analyze their test-time compute scaling properties, but first we provide a brief overview of how a PRM can be trained.

5.1 TRAINING VERIFIERS AMENABLE TO SEARCH

We follow the approach of Wang et al. (2023), which supervises the PRM using estimates of per-step correctness obtained from running Monte Carlo rollouts from each step in the solution. Our PRM’s per-step predictions therefore correspond to value estimates of reward-to-go for the base model’s sampling policy, similar to recent work (Wang et al., 2023; Setlur et al., 2024). We also compared to an ORM baseline (Appendix H) but found that our PRM consistently outperforms the ORM. Hence, all of the search experiments in this section use a PRM model. Additional details are in Appendix F.

Answer aggregation. At test time, PRMs can be used to score each individual step appearing in a set of solutions sampled from the base model. To pick out the best answer from N samples with the PRM, we need a function that can aggregate across all the per-step scores for each answer to determine the best candidate for the correct answer. To do this, we take the PRM’s prediction at the last step as representative of the full-answer score and then follow Li et al. (2023) by applying “best-of-N weighted” selection across answers. We include more detail on these decisions in Appendix G.

5.2 SEARCH METHODS AGAINST A PRM

We optimize the PRM at test time via tree search methods. We study three search approaches that sample outputs from a few-shot prompted base LLM (see Appendix J). An illustration is shown in

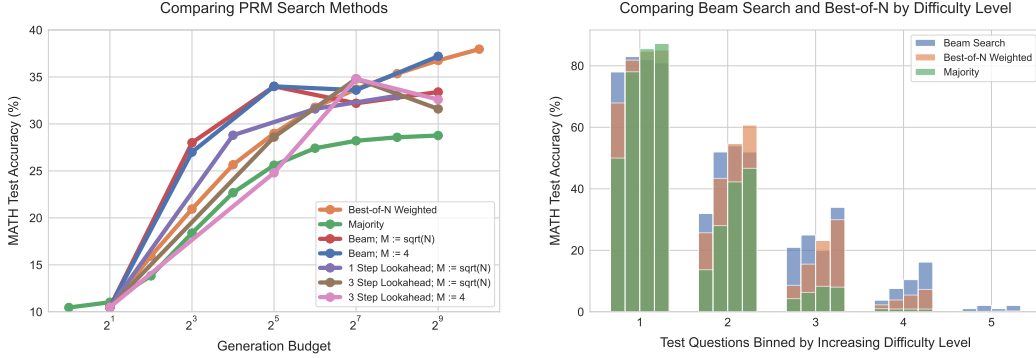


Figure 3: **Left:** *Comparing different methods for conducting search against PRM verifiers.* We see that at low generation budgets, beam search performs best, but as we scale the budget further the improvements diminish, falling below the best-of-N baseline. Lookahead-search generally underperforms other methods at the same generation budget. **Right:** *Comparing beam search and best-of-N binned by difficulty level.* The four bars in each difficulty bin correspond to increasing test-time compute budgets (4, 16, 64, and 256 generations). On the easier problems (bins 1/2), beam search shows signs of over-optimization at higher budgets, whereas best-of-N does not. On the medium difficulty problems (bins 3/4), we see beam search consistently outperforming best-of-N.

Figure 2. We note that for all search algorithms, we use the same PRM verifier, enabling an even comparison. We include additional details about our different search methods in Appendix C.

Best-of-N weighted. We sample N answers independently from the base LLM and then select the best answer according to the PRM’s final answer judgment.

Beam search. Beam search optimizes the PRM by searching over its per-step predictions. Our implementation is similar to BFS-V (Yao et al., 2023; Feng et al., 2024). Concretely, we consider a fixed number of beams N and a beam width M . At the end of the search we have N final answer candidates, to which we apply best-of-N weighted selection to make our final answer prediction.

Lookahead search. Lookahead search modifies how beam search evaluates each step. At each step in the search, rather than using the PRM score at the current step to select the top options, lookahead search performs a simulation, rolling out k steps. We stop early if the end of a solution is reached.

5.3 ANALYSIS RESULTS: TEST-TIME SCALING FOR SEARCH WITH VERIFIERS

We now present our results comparing various search algorithms and identify a prompt difficulty dependent compute-optimal scaling strategy for search methods.

Comparing search algorithms. We first conduct a sweep over different search settings. In addition to the standard best-of-N approach, we sweep over the two main parameters that distinguish these methods: beam-width M and number of lookahead steps k . While we are not able to exhaustively sweep all configurations, we sweep over the following settings with a maximum budget of 256: 1) Beam search with the beam width set to \sqrt{N} , where N is the generation budget; 2) Beam search with a fixed beam width of 4; 3) Lookahead search with $k = 3$ applied to both beam-search settings 1) and 2); 4) Lookahead search with $k = 1$ applied to beam-search setting 1).

To compare search methods as a function of generation budget fairly, we estimate the inference-time cost of each method. For beam search and best-of-N the generation budget corresponds to the number of beams and N respectively. Lookahead search utilizes additional compute: at each step, we sample k additional steps ahead. Therefore, the cost of lookahead-search is $N \times (k + 1)$ samples.

Querying the verifier also adds a 2x overhead for all methods; we account for this in our analysis.

Results. As shown in Figure 3 (left), with small budgets, beam search outperforms best-of-N. However, at high budgets, these improvements diminish, with beam search underperforming. Additionally, lookahead-search underperforms other methods, likely due to the additional computation induced by looking-ahead. It is possible that with further test-time scaling or with an online MCST trained value function, lookahead search may perform better; we leave further exploration of this to future work. The diminishing returns from search are likely due to exploitation of the PRM’s predictions. For example, we see instances (such as in Figure 31), where search causes the model to generate repetitive low-information steps. In other cases, we find that over-optimizing search can result in overly short solutions, of just 1-2 steps. We include several of these examples in Appendix Q.

Which problems does search improve? To understand how to scale search adaptively per problem, we conduct a difficulty bin analysis. Specifically, we compare beam-search ($M = 4$) against best-of-N. In Figure 3 (right), we find that, despite performing similarly in aggregate, the two methods exhibit very different behavior across difficulty levels. For example, on easy questions (levels 1/2), the stronger optimizer of the two, beam search, degrades in performance as the budget increases, suggesting possible exploitation of the PRM signal. In contrast, on the harder questions (levels 3/4), beam search outperforms best-of-N. Finally, on the most difficult questions (level 5), no method makes meaningful progress. These findings match intuition: we might expect that on the easy or medium difficulty questions, the verifier will make mostly correct assessments of correctness. Therefore, by optimizing further, we may be only further amplifying any spurious features learned by the verifier, causing performance degradation. On more difficult questions, the base model is less likely to sample the correct answer, so using search can help steer the model.

Compute-optimal search. Given the above, it is clear that question difficulty is a useful statistic for predicting the best search strategy at each budget. Additionally, the selected best search strategy varies as a function of difficulty. We visualize this “compute-optimal” scaling trend, as represented by the best performing search strategy, between best-of-N and beam search ($M = 4$), at each difficulty level in Figure 4. Interestingly, we see that with low budgets, using both the oracle and predicted difficulty, **compute-optimal scaling can nearly outperform best-of-N using up to $4\times$ less test-time compute** (e.g. 16 versus 64 generations). While at higher budgets, some of these benefits diminish with the use of predicted difficulty, but the oracle bins still see improvements from optimal scaling. This result demonstrates that there are clear performance gains to be obtained by adaptively allocating test-time compute during search using predicted difficulty as an input statistic.

Takeaways for compute-optimal scaling of verifiers

We find that the efficacy of any given verifier search method depends critically on both the compute budget and the question at hand. Specifically, beam-search is more effective on harder questions and at lower compute budgets, whereas best-of-N is more effective on easier questions and at higher budgets. Moreover, by selecting the best search setting for a given question difficulty and test-time compute budget, we can nearly outperform best-of-N using up to $4\times$ less test-time compute.

6 REFINING THE PROPOSAL DISTRIBUTION

Now we study how the proposal distribution can be used for test-time scaling (Section 2). Concretely, we enable to improve its own distribution at test-time, by revising answers iteratively. Simply prompting existing LLMs to correct themselves tends to be largely ineffective on reasoning (Huang et al., 2023). Therefore, we finetune LLMs to iteratively revise their answers.

6.1 TRAINING AND USING REVISION MODELS

Our procedure for finetuning revision models is similar to Qu et al. (2024b), though we introduce some crucial differences. For finetuning, we need trajectories consisting of a sequence of incorrect answers followed by a correct answer, that we can then run SFT on. To do this, we sampled 64 responses *in parallel* and post-hoc constructed multi-turn rollouts from these independent samples. These rollouts consist of up to four incorrect attempts in context followed by a correct revision. We include more details on our revision model finetuning procedure in Appendix K.

Using revisions at inference-time. Given a finetuned model, we can then sample a sequence of revisions from the model at test time. While our revision model is only trained with up to four pre-

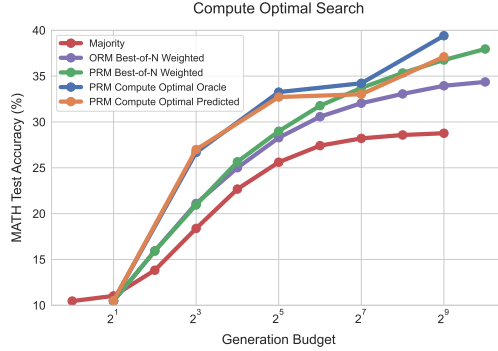


Figure 4: **Comparing compute-optimal test-time scaling against baselines with PRM search.** By scaling test-time compute optimally, we nearly outperform PRM best-of-N using up to $4\times$ less test-time compute (e.g. 32 versus 128 generations). “**Compute-optimal oracle**” refers to using oracle difficulty bins derived from the groundtruth correctness, and “**compute-optimal predicted**” refers to using the PRM’s predictions to generate difficulty bins.

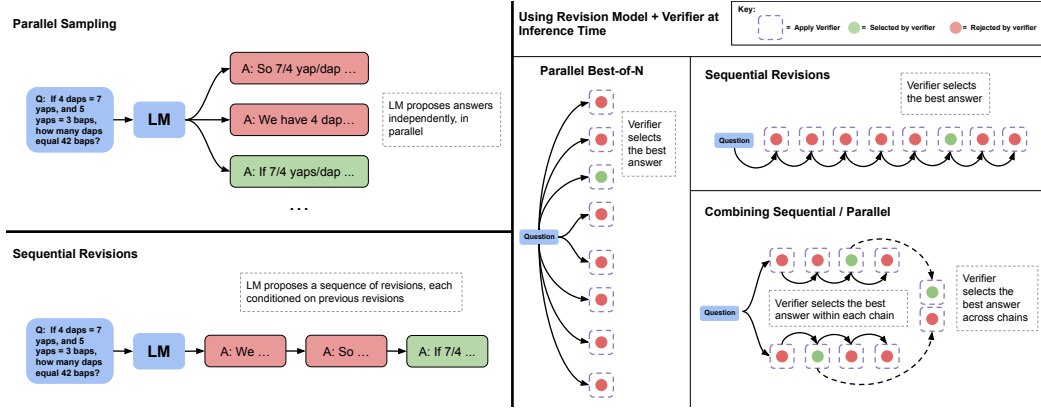


Figure 5: *Parallel sampling (e.g., Best-of-N) versus sequential revisions.* **Left:** Parallel sampling generates N answers independently, whereas sequential revisions generate each one in sequence conditioned on previous attempts. **Right:** In both the sequential and parallel cases, we can use the verifier to determine the best-of- N answers. We can also split our budget between parallel and sequential sampling. In this case, we first use the verifier to select the best answer *within* each sequential chain and then select the best answer *across* chains.

vious answers in-context, we can sample longer chains by truncating the context to the most recent four revisions. In Figure 9(left), we see longer chains gradually improve pass@ k demonstrating that we are able to effectively teach the model to learn from mistakes in previous answers.

That said, there is a distribution shift at inference time: the model was trained on only sequences with incorrect answers in context, but at test-time the model may sample correct answers. Thus, the model may turn a correct answer into an incorrect one. Similar to Qu et al. (2024b), around 38% of correct answers get converted to incorrect with our model. Thus, we employ either sequential majority voting or verifier-guided selection to select the correct answer from the sequence of revisions (see Figure 5). *Querying the verifier adds a 2x compute overhead, and we account for this in our analysis.*

Comparisons. To test the efficacy of modifying the proposal distribution via revisions, we set up a comparison between the performance of sampling N revisions in sequence and sampling N attempts at a question in parallel. We see in Figure 9 (right), that with both the verifier-based and majority-based selection mechanisms, sequential sampling outperforms parallel sampling.

6.2 ANALYSIS RESULTS: TEST-TIME SCALING WITH REVISIONS

We see that sampling sequentially outperforms in parallel. We might expect however, that these approaches have different properties. Intuitively, sampling in parallel acts as a global search process that could, in principle, provide coverage over many different approaches for solving a problem. Sequential sampling, on the other hand, may work more as a local refinement process. This motivates striking a balance between these two approaches by allocating some of our budget to parallel sampling (e.g. \sqrt{N}) and the rest to sequential (e.g. \sqrt{N}). We will now show the existence of a compute-optimal ratio between sequential and parallel sampling, and understand their pros and cons based on the difficulty of a prompt.

Trading off sequential and parallel compute.

To understand how to allocate sequential and parallel compute, we perform a sweep over different configurations. We see, in Figure 7 (left), that indeed, at a given budget, *there exists an ideal sequential to parallel ratio*. We also see in Figure 7 (right) that *this ideal ratio varies depending on question difficulty*. Easy questions benefit more from revisions, whereas on difficult questions it is optimal to strike a balance between sequential and parallel computation. This finding supports the hypothesis that sequential revisions (i.e., varying the

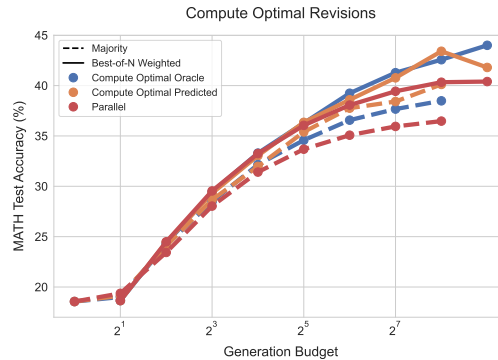


Figure 6: *Compute-optimal scaling with our revision model.* By optimally scaling test-time compute, we outperform best-of- N with $4\times$ less compute (i.e., 128 samples versus 512). “**Compute Optimal Oracle**” refers to difficulty derived from ground truth correctness and “**Compute Optimal Predicted**” refers to using the PRM to estimate difficulty.

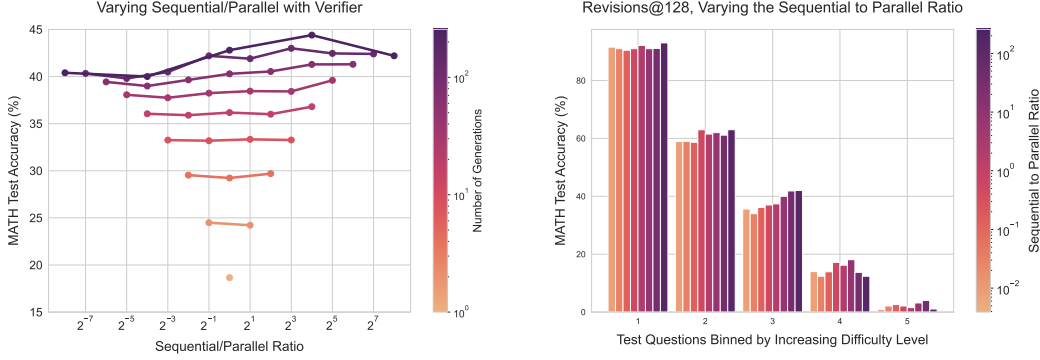


Figure 7: **Left: Varying the ratio of the generation budget allocated sequential revisions to versus parallel samples.** Each line represents a fixed generation budget as the ratio is changed. We use the verifier for answer selection. We see that while increased sequential revisions tends to outperform more parallel compute, at higher generation budgets there is an ideal ratio that strikes a balance between the two extremes. **Right: Varying the sequential to parallel ratio for a generation budget of 128 across difficulty bins.** Using verifier-based selection, we see that the easier questions attain the best performance with full sequential compute. On the harder questions, there is an ideal ratio of sequential to parallel test-time compute.

proposal distribution) and parallel sampling (i.e., search with verifiers) are complementary axes for scaling test-time compute, which may be more effective on a per-prompt basis. We include examples of our model’s generations in Appendix P. Additional results are in Appendix D.

Compute-optimal revisions. Given our finding that the efficacy of sequential and parallel sampling depends on difficulty, we can select the ideal ratio of sequential to parallel compute per difficulty bin (we describe the specific ratios in Appendix N). In Figure 6, we plot results using our compute-optimal scaling when employing both oracle and predicted difficulty. In both cases, we substantially improve test-time compute scaling by optimally scaling the proposal distribution. In particular, we see that at higher generation budgets, parallel sampling plateaus, whereas compute-optimal scaling continues to improve. For both oracle and predicted difficulty, we see that *compute-optimal scaling can outperform best-of-N using up to $4\times$ less test-time compute* (e.g. 64 samples versus 256). Overall, these results demonstrate the potential for improved test-time compute scaling by adjusting the proposal distribution on a per-prompt basis.

Takeaways for compute-optimal scaling by refining the proposal distribution with revisions

We find that there exists a tradeoff between sequential (e.g. revisions) and parallel (e.g. standard best-of-N) test-time computation, and the ideal ratio of sequential to parallel test-time compute depends on both the compute budget and the specific question at hand. Specifically, easier questions benefit from purely sequential test-time compute, whereas harder questions often perform best with an ideal ratio of sequential to parallel compute. By selecting the best setting for a given question difficulty and compute budget, we can outperform the parallel best-of-N baseline using up to $4\times$ less test-time compute.

7 EXCHANGING PRETRAINING AND TEST-TIME COMPUTE

We saw that utilizing additional test-time compute can enable us to represent more complex distributions than the one predicted by the base LLM, thereby increasing performance. We now posit that this increased flexibility of representing distributions means that we can expect additional test-time compute to make up for the lack of a higher-capacity model or training for more FLOPs during pretraining. In this section, *we study to what extent this is possible*. We pose the following question:

Question: Exchanging pretraining and test-time compute

Suppose a model was pre-trained with X FLOPs. Assume that we plan to run Y FLOPs of inference with this model. If we want to improve performance by increasing the total FLOPs budget by a factor of M (i.e., $M(X+Y)$ total FLOPs across both pretraining and inference), should we spend our FLOPs on increased pretraining compute or on additional test-time compute?

Increasing pretraining FLOPs introduces the additional design decision of whether to allocate compute to training with more data or more parameters (Hoffmann et al., 2022). We focus on the setting in which model parameters are scaled up and training data amount is fixed, matching the canonical approach from the LLaMA series of models (Touvron et al., 2023).

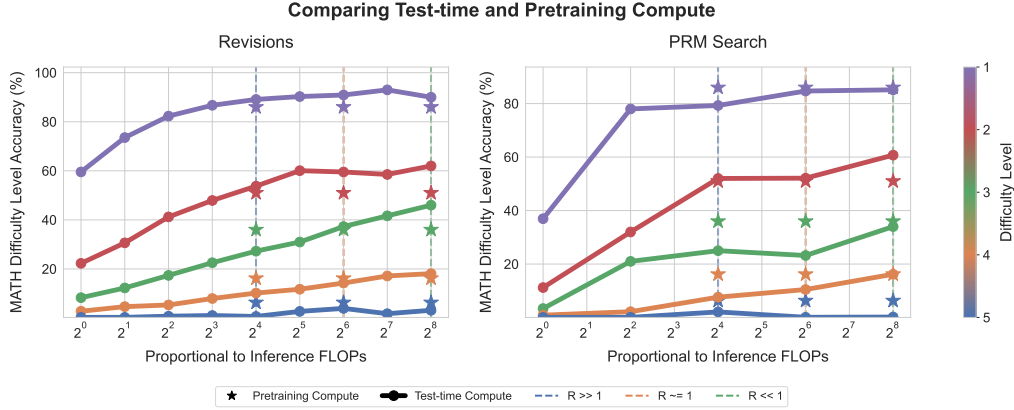


Figure 8: *The tradeoff between pretraining and test-time compute in a FLOPs-matched evaluation.* Each line represents the performance of scaling test-time compute with our compute-optimal policy in each oracle difficulty bin for revisions (left) and search (right). The stars represent the greedy pass@1 performance of a base model pretrained with ~ 14 times more parameters. We plot test-time compute budget on the x-axis and stars at three different locations along the x-axis, each corresponding to the FLOPs equivalent point of comparison between scaling parameters and scaling test-time compute for three different inference compute loads (e.g. $R = \frac{D_{\text{inference}}}{D_{\text{pretrain}}}$). If the star is below the line, this implies that it is more effective to use test-time compute than to scale model parameters, and if the star is above the line this implies that scaling parameters is more effective. We see that on the easy questions or in settings with a lower inference load (e.g. $R \ll 1$), test-time compute can generally outperform scaling model parameters. However, on the harder questions or in settings with a higher inference load (e.g. $R \gg 1$), pretraining is a more effective way to improve performance.

Exchanging FLOPs. We use the common approximation for pretraining FLOPs $X = 6ND_{\text{pretrain}}$ (Hoffmann et al., 2022), and for inference FLOPs, we use $Y = 4ND_{\text{inference}}$ (Sardana & Frankle, 2023), which multiplies the standard $2ND_{\text{inference}}$ by two to account for the overhead of calling the verifier. Here N represents model parameters, D_{pretrain} is the total tokens used for pretraining, and $D_{\text{inference}}$ the total tokens generated at inference. If we multiply N by a factor of M , then both the pretraining and inference FLOPs (due to the cost of greedy decoding with the larger model) increase by a factor of M , giving a total of $M(X + Y)$ FLOPs.

To match the FLOPs between scaling parameters and scaling test-time compute, we multiply the smaller model’s inference compute by $M + \frac{3}{2} (D_{\text{pre}}/D_{\text{inf}}) (M-1)^1$. Notably, this multiplier depends on the ratio $D_{\text{pre}}/D_{\text{inf}}$. We refer to the inverse of this ratio as $R = D_{\text{inf}}/D_{\text{pre}}$. Depending on the specific production setting, we should expect very different values of R . In particular, in large scale production settings, we may expect more inference tokens than pretraining tokens, in which case we have $R \gg 1$. On the other hand, in many self-improvement setups, we would likely generate fewer inference tokens than pretraining tokens, giving $R \ll 1$. Therefore, since the scale of test-time compute depends on this ratio, we expect differing conclusions depending on the specific setting.

In Figure 8, we use this approach to exchanging test-time and pretraining compute to compare our compute-optimal scaling against scaling up model parameters by a factor of ~ 14 . We conduct comparisons for 3 values of R : 0.08 ($R \ll 1$), 0.40 ($R \sim 1$), and 11 ($R \gg 1$), with each ratio corresponding to an inference budget. Observe that if we only expect to see difficult questions (e.g. bins 4/5) or have a larger $D_{\text{inference}}$ (i.e., larger R value), then it is often more effective to allocate compute towards pretraining (e.g. the star is above the line). If instead, we expect mostly easy or intermediate difficulty questions (e.g. bins 1-3 and sometimes 4) or have lower inference requirements (as is the case in self-improvement pipelines), then scaling test-time compute is preferred.

Takeaways for exchanging pretraining and test-time compute

Test-time and pretraining compute are not 1-to-1 “exchangeable”. In settings with a small inference requirement or on questions of moderate difficulty, test-time compute can substitute for pretraining. However, on challenging questions or under higher inference loads, pretraining is likely more effective.

Conclusions. Please see Appendix A for a detailed discussion of limitations and future work.

¹We do not account for finetuning FLOPs, since it is negligible compared to pretraining FLOPs.

8 REPRODUCIBILITY STATEMENT

Our work does not propose any new method and instead conducts analysis using methods proposed in prior works (Wang et al., 2023; Kumar et al., 2024; Welleck et al., 2022; Yao et al., 2023; Qu et al., 2024b) on the popular MATH benchmark (Hendrycks et al., 2021) using the PaLM 2-S* (Anil et al., 2023) model. We include extensive details about differences between our work and these prior works that we build on in Sections 5 6 and Appendices C, K, F, G, L, M, J, N including all relevant fine-tuning hyper-parameters used. We also conduct numerous ablations in Sections 5, 6 and Appendix D, E, G, H, M, O, and include a handful of examples outputs from our models in Appendix Q and P. We believe that all of these details included in the paper contribute to our work’s reproducibility. The base LLM that we used for our analysis, PaLM 2-S*, can be accessed for both finetuning and inference on the Google Cloud Vertex API (it is referred to as Codey on the API).

REFERENCES

- Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to mcmc for machine learning. 2003.
- Rohan Anil, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, Eric Chu, Jonathan H. Clark, Laurent El Shafey, Yanping Huang, Kathy Meier-Hellstern, Gaurav Mishra, Erica Moreira, Mark Omernick, Kevin Robinson, Sebastian Ruder, Yi Tay, Kefan Xiao, Yuanzhong Xu, Yujing Zhang, Gustavo Hernandez Abrego, Junwhan Ahn, Jacob Austin, Paul Barham, Jan Botha, James Bradbury, Siddhartha Brahma, Kevin Brooks, Michele Catasta, Yong Cheng, Colin Cherry, Christopher A. Choquette-Choo, Aakanksha Chowdhery, Clément Crepy, Shachi Dave, Mostafa Dehghani, Sunipa Dev, Jacob Devlin, Mark Díaz, Nan Du, Ethan Dyer, Vlad Feinberg, Fangxiaoyu Feng, Vlad Fienber, Markus Freitag, Xavier Garcia, Sebastian Gehrmann, Lucas Gonzalez, Guy Gur-Ari, Steven Hand, Hadi Hashemi, Le Hou, Joshua Howland, Andrea Hu, Jeffrey Hui, Jeremy Hurwitz, Michael Isard, Abe Ittycheriah, Matthew Jagielski, Wenhao Jia, Kathleen Kenealy, Maxim Krikun, Sneha Kudugunta, Chang Lan, Katherine Lee, Benjamin Lee, Eric Li, Music Li, Wei Li, YaGuang Li, Jian Li, Hyeontaek Lim, Hanzhao Lin, Zhongtao Liu, Frederick Liu, Marcello Maggioni, Aroma Mahendru, Joshua Maynez, Vedant Misra, Maysam Moussalem, Zachary Nado, John Nham, Eric Ni, Andrew Nystrom, Alicia Parrish, Marie Pellat, Martin Polacek, Alex Polozov, Reiner Pope, Siyuan Qiao, Emily Reif, Bryan Richter, Parker Riley, Alex Castro Ros, Aurko Roy, Brennan Saeta, Rajkumar Samuel, Renee Shelby, Ambrose Slone, Daniel Smilkov, David R. So, Daniel Sohn, Simon Tokumine, Dasha Valter, Vijay Vasudevan, Kiran Vodrahalli, Xuezhi Wang, Pidong Wang, Zirui Wang, Tao Wang, John Wieting, Yuhuai Wu, Kelvin Xu, Yunhan Xu, Linting Xue, Pengcheng Yin, Jiahui Yu, Qiao Zhang, Steven Zheng, Ce Zheng, Weikang Zhou, Denny Zhou, Slav Petrov, and Yonghui Wu. Palm 2 technical report, 2023.
- Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional ai: Harmlessness from ai feedback, 2022.
- Guoxin Chen, Minpeng Liao, Chengxi Li, and Kai Fan. Alphamath almost zero: process supervision without process, 2024.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.
- Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate, 2023.

- Xidong Feng, Ziyu Wan, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training, 2024.
- Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models, 2023. URL <https://arxiv.org/abs/2211.10435>.
- Sachin Goyal, Ziwei Ji, Ankit Singh Rawat, Aditya Krishna Menon, Sanjiv Kumar, and Vaishnavh Nagarajan. Think before you speak: Training language models with pause tokens, 2024. URL <https://arxiv.org/abs/2310.02226>.
- Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset, 2021.
- Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022.
- Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet, 2023.
- Andy L. Jones. Scaling scaling laws with board games, 2021. URL <https://arxiv.org/abs/2104.03113>.
- Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.
- Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. Training language models to self-correct via reinforcement learning, 2024. URL <https://arxiv.org/abs/2409.12917>.
- Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models, 2022.
- Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. Making large language models better reasoners with step-aware verifier, 2023.
- Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let’s verify step by step, 2023.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023.
- OpenAI. Gpt-4 technical report, 2024.
- Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023. URL <https://arxiv.org/abs/2307.16789>.
- Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Tool learning with large language models: A survey, 2024a. URL <https://arxiv.org/abs/2405.17935>.

- Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve. *arXiv preprint arXiv:2407.18219*, 2024b.
- Jon Saad-Falcon, Adrian Gamarra Lafuente, Shlok Natarajan, Nahum Maru, Hristo Todorov, Etash Guha, E. Kelly Buchanan, Mayee Chen, Neel Guha, Christopher Ré, and Azalia Mirhoseini. Archon: An architecture search framework for inference-time techniques, 2024. URL <https://arxiv.org/abs/2409.15254>.
- Nikhil Sardana and Jonathan Frankle. Beyond chinchilla-optimal: Accounting for inference in language model scaling laws, 2023.
- William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. Self-critiquing models for assisting human evaluators, 2022.
- Amrith Setlur, Saurabh Garg, Xinyang Geng, Naman Garg, Virginia Smith, and Aviral Kumar. RL on incorrect synthetic data scales the efficiency of llm math reasoning by eight-fold. *arXiv preprint arXiv:2406.14532*, 2024.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024.
- Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.
- Avi Singh, John D. Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J. Liu, James Harrison, Jaehoon Lee, Kelvin Xu, Aaron Parisi, Abhishek Kumar, Alex Alemi, Alex Rizkowsky, Azade Nova, Ben Adlam, Bernd Bohnet, Gamaleldin Elsayed, Hanie Sedghi, Igor Mordatch, Isabelle Simpson, Izzeddin Gur, Jasper Snoek, Jeffrey Pennington, Jiri Hron, Kathleen Kenealy, Kevin Swersky, Kshiteej Mahajan, Laura Culp, Lechao Xiao, Maxwell L. Bileschi, Noah Constant, Roman Novak, Rosanne Liu, Tris Warkentin, Yundi Qian, Yamini Bansal, Ethan Dyer, Behnam Neyshabur, Jascha Sohl-Dickstein, and Noah Fiedel. Beyond human data: Scaling self-training for problem-solving with language models, 2024.
- Kaya Stechly, Matthew Marquez, and Subbarao Kambhampati. Gpt-4 doesn’t know it’s wrong: An analysis of iterative prompting for reasoning problems, 2023.
- Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Second edition, 2018.
- Gemini Team. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024.
- Ye Tian, Baolin Peng, Linfeng Song, Lifeng Jin, Dian Yu, Haitao Mi, and Dong Yu. Toward self-improvement of llms via imagination, searching, and criticizing, 2024.
- Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023. URL <https://arxiv.org/abs/2307.09288>.
- Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback, 2022.

- Karthik Valmeekam, Matthew Marquez, and Subbarao Kambhampati. Can large language models really improve by self-critiquing their own plans?, 2023.
- Pablo Villalobos and David Atkinson. Trading off compute in training and inference, 2023. URL <https://epochai.org/blog/trading-off-compute-in-training-and-inference>. Accessed: 2024-07-03.
- Junlin Wang, Siddhartha Jain, Dejiao Zhang, Baishakhi Ray, Varun Kumar, and Ben Athiwaratkun. Reasoning in token economies: Budget-aware evaluation of llm reasoning strategies, 2024a. URL <https://arxiv.org/abs/2406.06461>.
- Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y. Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations, 2023.
- Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah D. Goodman. Hypothesis search: Inductive reasoning with language models, 2024b. URL <https://arxiv.org/abs/2309.05660>.
- Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct, 2022. URL <https://arxiv.org/abs/2211.00053>.
- Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. An empirical analysis of compute-optimal inference for problem-solving with language models, 2024. URL <https://arxiv.org/abs/2408.00724>.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.
- Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling relationship on learning mathematical reasoning with large language models, 2023.
- Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning, 2022.
- Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D. Goodman. Quiet-star: Language models can teach themselves to think before speaking, 2024. URL <https://arxiv.org/abs/2403.09629>.

Appendices

A DISCUSSION AND FUTURE WORK

In this work, we conducted a thorough analysis of the efficacy of different techniques that aim to either improve search against a verifier or to refine an LLM’s proposal distribution, for scaling test-time compute for math reasoning. In general, we found that the efficacy of a given approach heavily correlates with the difficulty of the problem from the perspective of the base LLM’s capabilities. This motivated us to introduce the notion of “compute-optimal” scaling of test-time computation, which prescribes an adaptive, prompt-dependent strategy to improve performance under a given test-time compute budget. By applying such a compute-optimal scaling strategy, we find that we can improve the efficiency of test-time compute scaling by a factor of $2 - 4\times$. When comparing benefits obtained from additional test-time compute against benefits from additional pre-training compute in a FLOPs-matched setting, we show for the first time that using test-time computation with seemingly simple methods (i.e., revisions and search) can already scale well on certain types of prompts, providing gains over spending those FLOPs in pretraining.

Limitations. That said, there are also limitations associated with our study that future work can aim to address. Firstly, while we obtained strong results with beam-search, our lookahead search generally underperformed. It is possible that our lookahead search algorithm could be further optimized by training a PRM verifier with online MCTS training. Future work should explore the degree much PRM search can be improved in this way. Secondly, computing our notion of difficulty requires applying a non-trivial amount of test-time compute. Future work should consider alternative ways of efficiently estimating prompt difficulty. Finally, it is unclear to what extent our findings generalize beyond math and easily verifiable domains more broadly. Furthermore, many limitations of test-time compute are largely unknown; it is possible that in some domains test-time compute may be limited in ways that pretraining is not. While answering this question is out of scope for the present work, we believe it is an important topic for future research.

Future-work. We believe there are many future research directions that can build on our findings. We describe a few of these directions here. While we focused on improving the test-time compute scaling of two primary mechanisms independently (the verifier and the proposal distribution), future work should investigate how test-time compute scaling can be further improved by combining these approaches or via fundamentally different mechanisms than those explored in this paper. Moreover, our work focused purely on test-time compute scaling. In the future, we envision that the outputs of applying additional test-time compute can be distilled back into the base LLM, enabling an iterative self-improvement loop that operates on open-ended natural language, which we believe is an exciting direction for future work to explore. Moreover, future work should conduct additional scaling-law analysis with respect to test-time compute: 1) how does the scaling of test-time compute improve as pretraining is scaled; and 2) if we were to finetune models on much larger datasets (e.g. of millions of question answer pairs) how would test-time compute scaling improve? And how would the cost of fine-tuning affect the balance between scaling up model size vs test-time inference? These are questions that future work can study building on framework built in our work.

B RELATED WORK

Language model reasoning. Language model performance on challenging mathematical reasoning tasks has rapidly improved in recent years (Lewkowycz et al., 2022; Team, 2024; OpenAI, 2024; Shao et al., 2024; Lightman et al., 2023). These improvements can be attributed to three primary factors: 1) running continued pretraining on large corpora of math focused data (Lewkowycz et al., 2022; Team, 2024; Shao et al., 2024; Lightman et al., 2023); 2) improving the LLM proposal distribution by either applying targeted optimization on specific reasoning tasks by finetuning with RL (Singh et al., 2024; Zelikman et al., 2022; Shao et al., 2024; Yuan et al., 2023) enabling models to critique and revise their answers iteratively (Bai et al., 2022; Madaan et al., 2023; Du et al., 2023; Saunders et al., 2022); 3) enabling LLMs to benefit from additional test-time computation by finetuning verifiers (Lightman et al., 2023; Cobbe et al., 2021; Uesato et al., 2022; Wang et al., 2023; Yao et al., 2023; Feng et al., 2024; Chen et al., 2024; Tian et al., 2024). Our work builds on these second and third lines of research by analyzing the extent to which test-time compute scaling can be improved by 1) refining an LLM’s proposal distribution and 2) conducting search against verifiers.

Analyzing test-time compute scaling. The tradeoff between train-time and test-time compute using Monte-Carlo tree search applied to the board game Hex was previously studied by Jones (2021). We instead focus our analysis on full-scale language model math reasoning problems. A survey work by Villalobos & Atkinson (2023) analyzed the tradeoff between training and inference across a number of domains. However, much of their language-model analysis focused on test-time compute scaling in settings where the ground-truth answer is known. In contrast, our analysis focuses on the setting when the ground-truth answer is not known. Additionally, a number of works in the RL literature have proposed methods, such as MCTS (Kocsis & Szepesvári, 2006), which aim to navigate the tradeoff between test-time and training-time compute so as to enable a form of iterative self-play. The findings in our work can be used to help develop similar algorithms that can operate on open-ended natural language.

Augmenting LLMs with test-time compute. Beyond verifiers and revisions, a number of additional works have proposed alternative methods for enabling LMs to use test-time compute for reasoning. Namely, Wang et al. (2024b) conducts a hierarchical hypothesis search to enable inductive reasoning capabilities. A number of related works have proposed augmenting language models with tools at test-time, which can greatly improve their performance on downstream tasks (Gao et al., 2023; Qin et al., 2023; Qu et al., 2024a). Several works have proposed methods for learning thought tokens in an unsupervised manner (Zelikman et al., 2024; Goyal et al., 2024), enabling models to more effectively utilize the additional test-time compute that comes with sampling longer sequences. Finally, Saad-Falcon et al. (2024) explore applying architecture-search based techniques to effectively compose several different test-time scaling techniques. While we focus our analysis on two primary mechanisms by which test-time compute can be scaled in this work (e.g. verifiers and revisions), many of the methods by which we conduct our analysis (e.g. compute optimal scaling according to question difficulty) could, in principle, also be applied to any of these other methods of scaling test-time compute, and we believe that this is an interesting direction for future research.

C SEARCH ALGORITHM DETAILS

Below we include additional details for each of our search algorithms in Section 5.

Best-of-N weighted. We sample N answers independently from the base LLM and then select the best answer according to the PRM’s final answer judgment.

Beam search. Beam search optimizes the PRM by searching over its per-step predictions. Our implementation is similar to BFS-V (Yao et al., 2023; Feng et al., 2024). Concretely, we consider a fixed number of beams N and a beam width M . We then run the following steps:

1. sample N initial predictions for the first step in the solution
2. score the generated steps according to the PRM’s predicted step-wise reward-to-go estimate (which also corresponds to the total reward from the prefix since the reward is sparse in this setting)
3. filter for only the top $\frac{N}{M}$ highest scoring steps
4. now from each candidate, sample M proposals from the next step, resulting in a total of $N/M \times M$ candidate prefixes again. Then repeat steps 2-4 again.

We run this algorithm until the end of a solution or the maximum number of rounds of beam expansion are attained (40 in our case). We conclude the search with N final answer candidates, to which we apply best-of- N weighted selection described above to make our final answer prediction.

Lookahead search. Lookahead search modifies how beam search evaluates individual steps. It uses lookahead rollouts to improve the accuracy of the PRM’s value estimation in each step of the search process. Specifically, at each step in the beam search, rather than using the PRM score at the current step to select the top candidates, lookahead search performs a simulation, rolling out up to k steps further while stopping early if the end of solution is reached. To minimize variance in the simulation rollout, we perform rollouts using temperature 0. The PRM’s prediction at the end of this rollout is then used to score the current step in the beam search. That is, in other words, we can view beam search as a special case of lookahead search with $k = 0$. Given an accurate PRM, increasing k should improve the accuracy of the per-step value estimates at the cost of additional compute. Also note that this version of lookahead search is a special case of MCTS (Sutton & Barto, 2018), wherein the stochastic elements of MCTS, designed to facilitate exploration, are removed since the

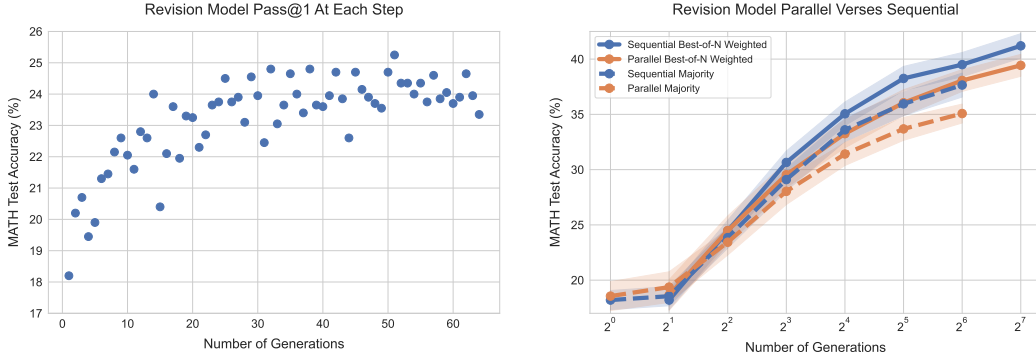


Figure 9: **Left:** *Our revision model’s pass@1 at each revision step.* Pass@1 gradually improves after each revision step, even improving beyond the 4 revision steps that it was trained for. We estimate pass@1 at each step by averaging over the performance of 4 revision trajectories of length 64 for each question in the test-set. **Right:** *Sequential vs parallel sampling from the revision model.* Comparing performance when generating N initial answers in parallel from our revision model, versus generating N revisions sequentially, with the model. To account for the cost of querying the verifier relative to majority voting, we shift the curves which involve a verifier over by one point. When using both the verifier and majority voting to select the answer, we see that generating answers sequentially with the revision model narrowly outperforms generating them in parallel.

PRM is already trained and is frozen. These stochastic elements are largely useful for learning the value function (which we’ve already learned with our PRM), but less useful at test-time when we want to exploit rather than explore. Therefore, lookahead search is largely representative of how MCTS-style methods would be applied at test-time.

D ADDITIONAL REVISION RESULTS

In Figure 9 left, we plot the pass@1 for our revision model at each revision step. We see that pass@1 gradually improves after each step. In Figure 9 right, we compare performance when generating N initial answers in parallel from our revision model, versus generating N revisions sequentially, with the model. When using both the verifier and majority voting to select the answer, we see that generating answers sequentially with the revision model narrowly outperforms generating them in parallel.

We plot additional results for majority voting selection using our PaLM 2-S* revision model in Figure 10. With majority selection, we see largely similar trends to those found in Figure 7 for verifier selection.

E DIFFICULTY BINS

Oracle difficulty bins. We compute our oracle difficulty bins by obtaining the ground-truth pass@1 correctness rate for each question, and then using this statistic to bin questions into quantiles representing 5 distinct difficulty bins. There are in total 500 questions in the test-set. Difficulty levels 1, 2, and 3 all have 100 questions in them. Difficulty levels 4 and 5 have 105 and 95 questions respectively. This imbalance in the last two bins is merely due to a boundary condition/ties in the quantile computation, causing one bin to inherit slightly more questions than the other.

Predicted difficulty bins. We compute difficulty bins without oracle ground-truth correctness information by averaging the PRM final-answer score over 2048 samples on each question, so as to obtain a value estimate corresponding to the question. Similar to the oracle case, we then bin the value for each question in the test-set into five quintiles (using the same procedure as the oracle difficulty bins). We refer to this as “predicted difficulty”. *Each of the bins has 100 questions in this*

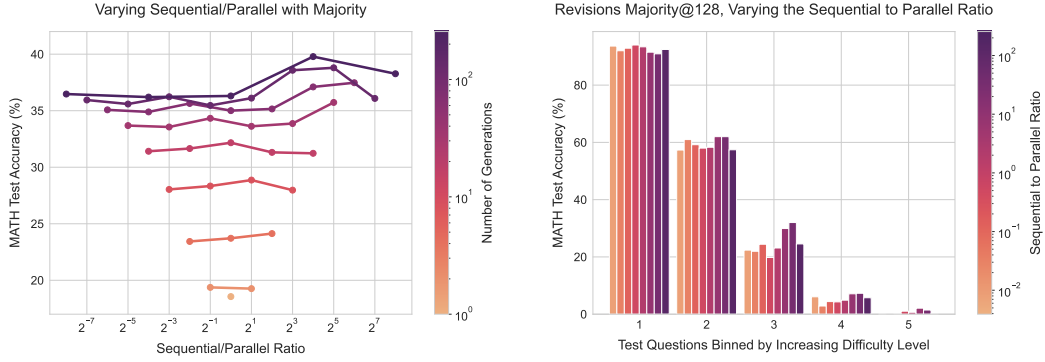


Figure 10: Varying the ratio of generation budget allocated to sequential versus parallel samples, using majority voting to select the answer, rather than the verifier. **Left:** Each line represents a fixed generation budget as the ratio is changed. We see that similar to the verifier case, in the majority case, there exists an ideal ratio of sequential to parallel test-time compute at a given budget. **Right:** Analyzing performance across difficulty bins, we see that the easier questions are mostly invariant to the ratio of sequential to parallel, whereas on the harder questions there is an ideal ratio of sequential to parallel test-time compute.

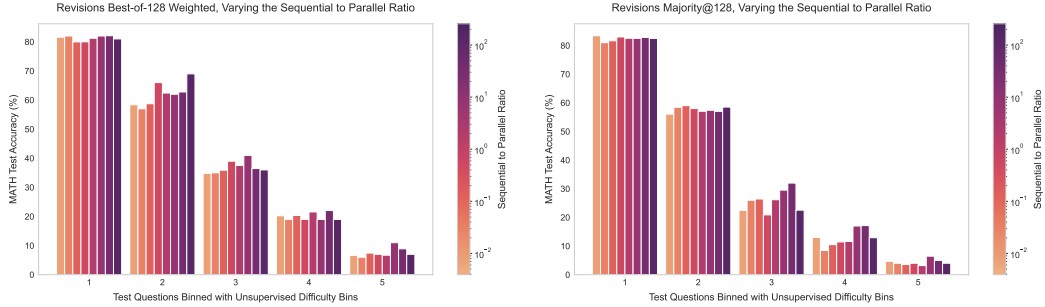


Figure 11: Using our PaLM 2-S* PRM to compute difficulty bins without ground truth correctness information for revisions. On the left we plot verifier selection and on the right we plot majority selection. We see largely similar performance trends with these bins as we do with the ground truth trends in Figures 7 and 10.

case. Technically this procedure is extremely costly because it requires generating many samples. While we do not account for this cost in our analysis, in a practical production setting, this cost would be problematic. A more efficient approach would be to finetune a model to predict correctness directly, given the question. We do not explore this in our work, but leave such exploration of cheaper methods of estimating difficulty to future work.

In Figure 12 we plot PRM-search results using our difficulty bins, and in Figure 11 we plot the corresponding revision results. We see that in both settings these predicted bins demonstrate similar trends to the oracle bins.

F PRM TRAINING DETAILS

Originally PRM training (Uesato et al., 2022; Lightman et al., 2023) used human crowd-worker labels. While Lightman et al. (2023) released their PRM training data (i.e., the PRM800k dataset), we found this data to be largely ineffective for us. We found that it was easy to exploit a PRM trained on this dataset via even naïve strategies such as best-of-N sampling. We hypothesize that this is likely a result of the distribution shift between the GPT-4 generated samples in their dataset and our PaLM 2 models. Rather than proceeding with the expensive process of collecting crowd-worker PRM labels for our PaLM 2 models, we instead apply the approach of Wang et al. (2023) to supervise

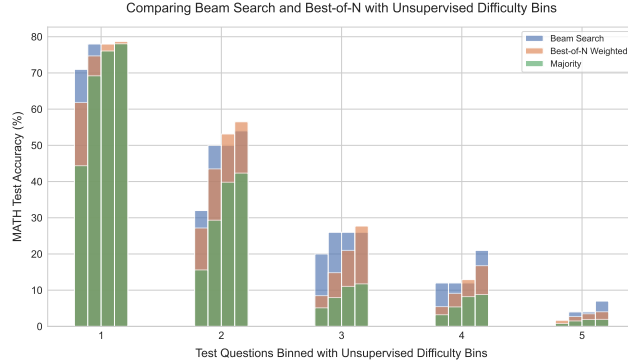


Figure 12: Using our PaLM 2-S* PRM to compute difficulty bins without ground truth correctness information for PRM search. We see largely similar performance trends with these bins as we do with the ground truth ones in Figure 3.

PRMs without human labels, using estimates of per-step correctness obtained from running Monte Carlo rollouts from each step in the solution. Our PRM’s per-step predictions therefore correspond to value estimates of reward-to-go for the base model’s sampling policy, similar to recent work (Wang et al., 2023; Setlur et al., 2024). We also compare to an ORM baseline (Appendix H) but found that our PRM consistently outperforms the ORM. Hence, all of the search experiments in this section use a PRM model.

We finetune our PRM as a binary classifier, where the model predicts a value between 0 and 1 at each step in the solution. We train the model with soft values obtained from the monte-carlo rollouts, using a binary cross entropy loss function (e.g. $-(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$) where y corresponds to the soft ground-truth value and \hat{y} the model’s predicted value). We finetune the model base model using the AdamW optimizer, with lr 3e-5, batch size 128, dropout 0.05, and Adam betas (0.9, 0.95). We conduct early stopping, selecting the checkpoint with the lowest validation loss on a random held-out validation set, consisting of 10% of the questions in the original PRM800k training split.

We finetune the PRM on 16 samples per question from the corresponding few-shot prompted base model. At each step, we use 16 monte-carlo rollouts, using the same base model and prompt, to estimate the step-level value. We filter out all samples which fail to output a valid, parsable final answer from the training data, as we found these to hurt PRM performance in initial experiments.

When generating the samples, the base model is prompted to output answers in newline separated step-by-step format, as done in Lightman et al. (2023). We then separate each of the answers into steps using a simple newline splitting procedure. We include details about our prompt in Appendix J.

G PRM AGGREGATION

At test time, process-based verifiers can be used to score each individual step in a set of solutions sampled from the base model. In order to select the best-of-N answers with the PRM, we need a function that can aggregate across all the per-step scores for each answer to determine the best candidate for the correct answer. To do this, we first aggregate each individual answer’s per-step scores to obtain a final score for the full answer (step-wise aggregation). We then aggregate across answers to determine the best answer (inter-answer aggregation). Concretely, we handle step-wise and inter-answer aggregation as follows:

- **Step-wise aggregation.** Rather than aggregating the per-step scores by taking the product or minimum (Wang et al., 2023; Lightman et al., 2023), we instead use the PRM’s prediction at the last step as the full-answer score. We found this to perform the best out of all aggregation methods we studied (see below).
- **Inter-answer aggregation.** We follow Li et al. (2023) and apply “best-of-N weighted” selection rather than standard best-of-N. Best-of-N weighted selection marginalizes the

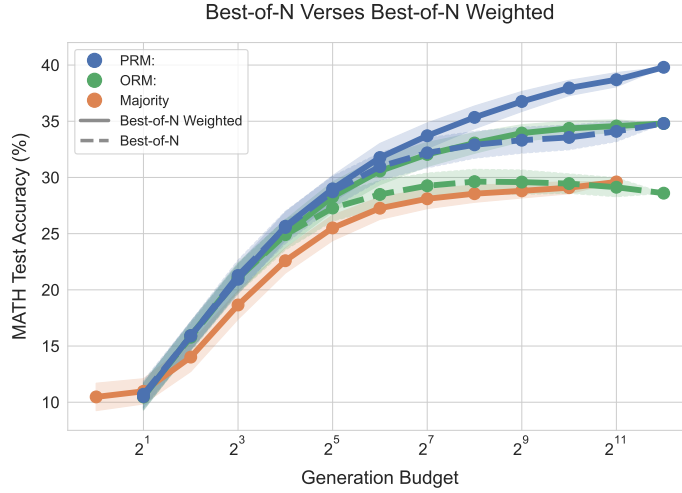


Figure 13: We compare best-of-N and best-of-N weighted for our ORM and PRM verifiers finetuned from PaLM 2-S*. We use the PaLM 2-S* base LM to sample outputs, using a few-shot prompt. To account for the cost of querying the verifier relative to majority voting, we shift the curves which involve a verifier over by one point. We see that while best-of-N weighted shows superior performance in both settings, the best-of-N performance with the PRM is still very competitive. On the other hand, in the ORM best-of-N setting, we observe Goodharting at higher budgets.

verifier’s correctness scores across all solutions with the same final answer, selecting final answer with the greatest total sum.

G.1 COMPARING STEP-WISE AGGREGATION STRATEGIES

We compare different methods of aggregating per-step PRM scores to produce a final score for the full solution. Specifically we compare: 1) taking the minimum score across all steps as done in Lightman et al. (2023) (e.g. “min”); 2) taking the product of all step correctness probabilities (e.g. “prod”); and 3) taking just the last step prediction (e.g. “last”). We see in Figure 14 that taking the last step outperforms the other two approaches. Prior works (Lightman et al., 2023; Wang et al., 2023) found min to be the best aggregator. We believe that the discrepancy is due to the fact that our verifier was trained with soft MC return labels, which surface very differently from binary correctness labels, and therefore other aggregation strategies may not have the same effect.

Interestingly, when using the last step aggregation, we are effectively using the PRM like an ORM. However, we see that the PRM outperforms the ORM, suggesting that in our case the per-step PRM training may be largely useful as a form of representation learning, rather than purely as a tool at inference time. Future work should further explore this line of reasoning.

G.2 COMPARING INTER-ANSWER AGGREGATION STRATEGIES

In Figure 13 we compare best-of-N against best-of-N weighted for both our ORM and PRM verifiers. We find that while best-of-N weighted shows superior performance in both settings, the best-of-N performance with the PRM is still very competitive. On the other hand, in the ORM best-of-N setting, we observe Goodharting at higher budgets.

H COMPARING PRM AND ORM

We trained a PRM and ORM model using the PaLM 2-S* base LM. We see in Figure 15, that the PRM outperforms the ORM, and the gap between the PRM and ORM grows with the number of samples used. We use the last step prediction from the PRM to score the answers as described in Appendix G.

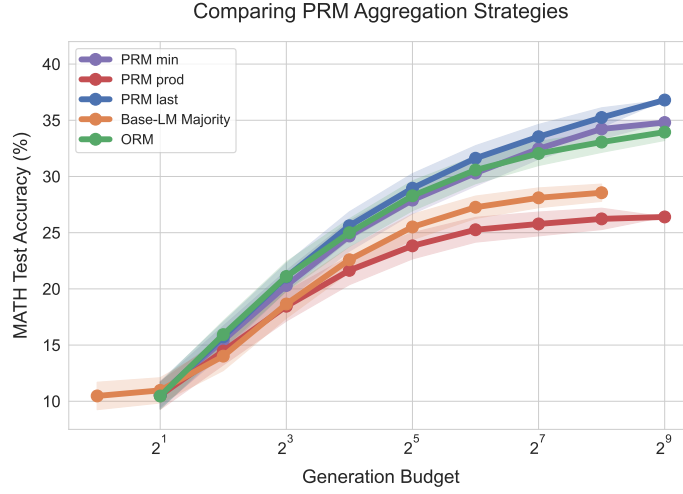


Figure 14: We compare different methods of aggregating per-step PRM scores to produce a final score for the full solution: “min” refers to taking the minimum score accross all steps, “prod” takes the product of all step correctness probabilities, and “last” just uses the last step score. To account for the cost of querying the verifier relative to majority voting, we shift the curves which involve a verifier over by one point. We see that “PRM last” performs the best across all aggregation strategies.

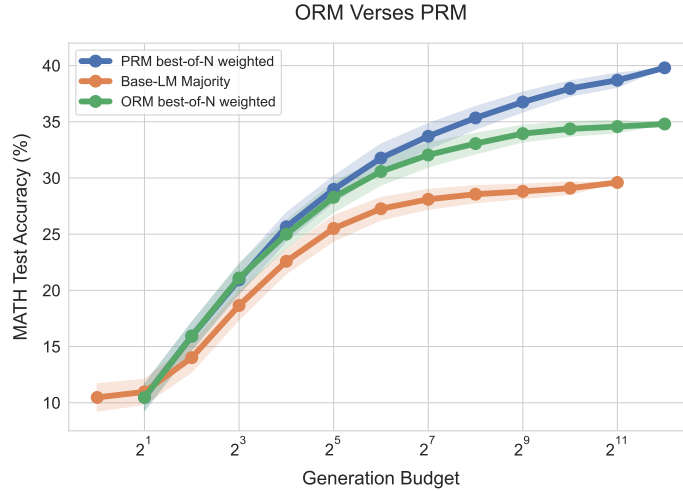


Figure 15: We compare PRM and ORM models finetuned from PaLM 2-S* in a best-of-N evaluation. We use the PaLM 2-S* base LM to sample outputs, using a few-shot prompt. To account for the cost of querying the verifier relative to majority voting, we shift the curves which involve a verifier over by one point. We see that the PRM greatly outperforms the ORM at a large number of samples.

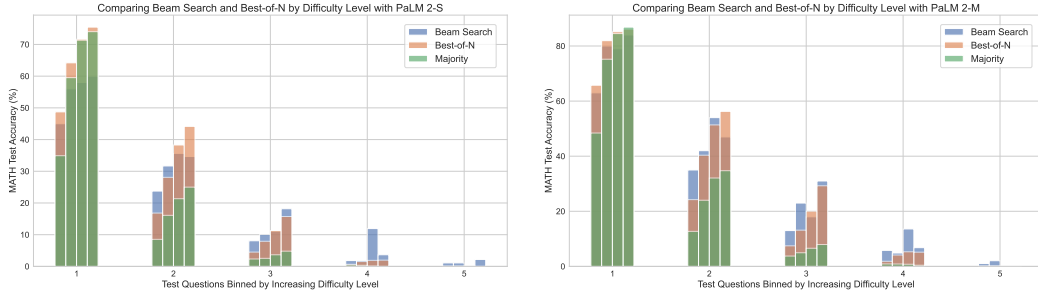


Figure 16: Comparing beam search and best-of-N binned by difficulty level with PaLM 2-S (left) and PaLM 2-M (right). The four bars in each difficulty bin correspond to increasing test-time compute budgets (4, 16, 64, and 256 generations). We observe broadly similar trends to those in Figure 3 on PaLM 2-S*, demonstrating that our findings likely transfer to other base LLMs.

I SEARCH USING PALM 2-S AND M

In Figure 16, we plot the performance of beam-search and best-of-N binned by difficulty levels using PaLM 2-S and PaLM 2-M as the base models. We observe broadly similar trends to those in Figure 3 on PaLM 2-S*, demonstrating that our findings likely transfer to other base LLMs.

J PROMPTING DETAILS

In order to enable the base model to output answers in a step-by-step format to which a PRM can be applied, we use a 4-shot prompt consisting of randomly selected correct answer examples from the PRM800k data released by Lightman et al. (2023). Specifically we use answers from the phase 1 training split. These answers correspond to GPT-4 generated correct answer examples, which include the correct step-by-step format. In initial experiments, we found that this prompting procedure produces similar results to the prompt used in Lewkowycz et al. (2022). We use this prompt for generating training data for the PRM and the revision model. We also use this prompt when conducting search against the PRM on the test-set. To grade the final answer predicted by this prompt, we use the grading function released by Lightman et al. (2023).

K REVISION MODEL FINETUNING DETAILS

Our procedure for finetuning revision models is similar to (Qu et al., 2024b), though we introduce some crucial differences. For finetuning, we need trajectories consisting of a sequence of incorrect answers followed by a correct answer, that we can then run SFT on. Ideally, we want the correct answer to be *correlated* with the incorrect answers provided in context, so as to effectively teach the model to *implicitly* identify mistakes in examples provided in-context, followed by correcting those mistakes by making edits as opposed to ignoring the in-context examples altogether, and trying again from scratch.

Generating revision data. The on-policy approach of Qu et al. (2024b) for obtaining several multi-turn rollouts was shown to be effective, but it was not entirely feasible in our infrastructure due to compute costs associated with running multi-turn rollouts. Therefore, we sampled 64 responses *in parallel* at a higher temperature and post-hoc constructed multi-turn rollouts from these independent samples. Specifically, following the recipe of (Kumar et al., 2024), we pair up each correct answer with a sequence of incorrect answers from this set as context to construct multi-turn finetuning data. We include up to four incorrect answers in context, where the specific number of solutions in context is sampled randomly from a uniform distribution over categories 0 to 4. The correct answer is used as the last answer in the trajectory (which we train the model to produce) and the incorrect answers are included in context. If the sampled number is greater than 0, we then find the closest incorrect answer according to a character-level edit distance metric to include as the last incorrect answer in the trajectory. The goal here is to select an incorrect answer which is somewhat correlated with

the correct answer, to improve learning. Note that token edit distance is not a perfect measure of correlation, but we found this heuristic to be sufficient to correlate incorrect in-context answers with correct target answers to facilitate training a meaningful revision model, as opposed to randomly pairing incorrect and correct responses with uncorrelated responses. Finally, in the case where there are fewer than 4 incorrect answers sampled, we truncate the uniform distribution’s max to match the number of incorrect samples. We use this procedure to generate trajectories for all questions in the training data.

We then finetune the base language model on the correct answer solutions in these generated trajectories. We use the AdamW optimizer with lr $1e-5$, batch size 128, dropout 0.0, and Adam betas (0.9, 0.95).

We find that generally evaluating loss on an evaluation set consisting of trajectories generated as described above, does not provide a good signal for early stopping. Rather, we find that checkpoints much after the evaluation loss begins increasing are much more capable of revisions. This is likely because after finetuning the revision model, the evaluation set represents off-policy data, which will naturally be out-of-distribution compared to the trajectories that the model itself would generate on-policy. We therefore select our revision model checkpoint slightly after the point where we observe overfitting on the validation set.

L REVISION MODEL SELECTION CRITERIA

As described in Section 6.1, in order to effectively use our revision model we need to deploy a criteria for selecting the best answer both within a revision trajectory and between multiple parallel trajectories. We use two approaches: 1) ORM verifier; and 2) majority voting.

For the ORM verifier, we train an ORM on the revision model’s outputs according to the procedure in Appendix M. At inference time we then use this verifier to select the best answer. Since we have two axes across which to aggregate (within each revision trajectories and between multiple trajectories), we deploy a hierarchical strategy, first selecting the best answer within each revision trajectory and then aggregating these selected answers across trajectories. To select the best answer within each trajectory, we perform best-of-N weighted aggregation and then choose the highest scoring solution with the maximum best-of-N weighted answer. Then, to select the final answer across all revision chains, we perform another round of best-of-N weighted selection using the best answer from each revision chain. The answer after this second round of best-of-N weighted represents our final answer prediction.

For majority voting we found hierarchical aggregation to create problems when the length of the trajectory or the number of trajectories was too small. The problem being that without enough samples, majority voting is unable to effectively select the best option. Therefore, for majority voting, we simply take all answers, across all trajectories, at once and take their majority as the final answer. We found this to produce much smoother scaling behavior than the hierarchical approach.

M REVISION MODEL VERIFIER TRAINING

We found that the PRM we finetuned on the PaLM 2-S* base model outputs was not as effective when applied to the PaLM 2-S* revision model’s outputs (see Figure 17(a)), likely due to distribution shift with the revision model. We therefore, trained a separate ORM verifier to use with our PaLM 2-S* revision model. We could have trained a PRM as well, but opted for an ORM due to the high cost of generating per-step PRM labels.

We modified the standard ORM slightly for the revision setting, by finetuning the ORM with previous revision in context, such that the verifier has access to the same context as the revision model, allowing the verifier see the revision model’s previous answer attempts when scoring the current answer. All other experiment details are identical to those used for training the PRM.

Empirically, we find that including the revision history in context improves performance slightly (see Figure 17(b)). Additionally, even without the revisions in context, we see that sequential revisions still slightly outperforms parallel, demonstrating improvements from sequential sampling are not just due to the verifier’s context.

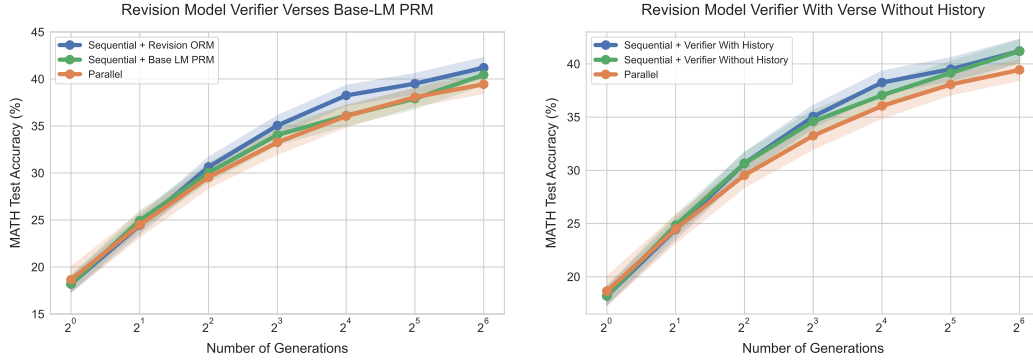


Figure 17: Left: we compare the ORM we trained on the revision model’s outputs against the PRM we trained on the PaLM 2-S* base model’s outputs. We see that when applied to outputs from the revision model, the ORM adapted to the revision model outperforms the PRM, likely due to distribution shift with the revision model. Right: we ablate the effect of including previous revisions in the revision model verifier’s context. We see that including revisions in-context helps the verifier slightly, but both settings still outperform the parallel baseline.

N	Sequential:Parallel Ratios
1	1:1
2	1:2, 2:1
4	1:4, 1:1, 4:1
8	1:8, 1:2, 2:1, 8:1
16	1:16, 1:4, 1:1, 4:1, 16:1
32	1:32, 1:8, 1:2, 2:1, 8:1, 32:1
64	1:64, 1:16, 1:4, 1:1, 4:1, 16:1, 64:1
128	1:128, 1:32, 1:8, 1:2, 2:1, 8:1, 32:1, 128:1
256	1:256, 1:16, 1:1, 16:1, 256:1

Table 1: We list the set of sequential to parallel ratios that we consider at each generation budget N . These are the ratios we select between when defining compute optimal scaling in Figure 6.

N COMPUTE OPTIMAL REVISIONS HYPERPARAMETERS

In Table 1 we list the sequential/parallel ratios that we select between at each generation budget when estimating compute optimal scaling in Figure 6.

O ReST^{EM} REVISION MODEL EXPERIMENTS

We experimented with further optimizing our PaLM 2-S* revision model by training the model with a simplified RL algorithm: ReST^{EM} (Singh et al., 2024). Specifically, we generated 64 revision trajectories of maximum length 5 for each question on the MATH training set. We stopped the revision model at the first correct answer in each trajectory. Using this generated data, we then finetuned the base LM on the correct answer data. To help the model learn the task, we explicitly balanced the distribution of trajectory lengths.

In Figure 18, we plot the performance of this new revision model as we vary the sequential to parallel ratio. We see that additional sequential revisions substantially hurts performance with this new model. We hypothesize that this degradation is due to the fact that the online data obtained from running ReST^{EM} exacerbates spurious correlations in revision data, causing the optimized model to fail to learn the revision task. We believe that using a more offline data collection strategy, as done in Qu et al. (2024b), may be more effective, and leave further exploration to future work.

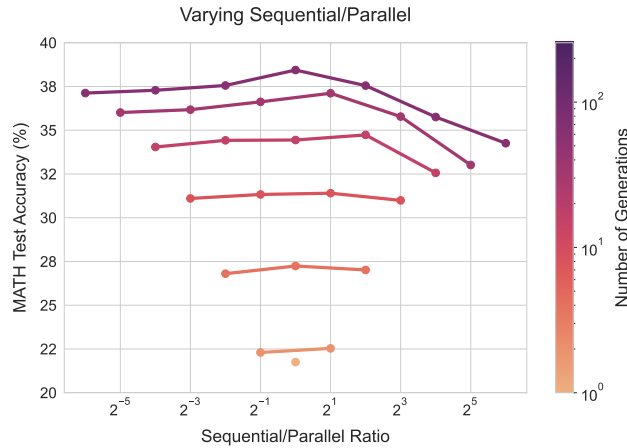


Figure 18: Performance of our ReST^{EM} optimized revision model as the sequential to parallel ratio is varied. We use majority voting to select the answer. We see that this optimized revision model demonstrates substantial performance degradations with additional sequential revisions.

P REVISION MODEL EXAMPLE OUTPUTS

In Figures 19, 20, 21, 22, 23, 24, and 25, we include select examples of our revision model’s outputs.

Q PRM BEAM SEARCH EXAMPLE OUTPUTS

In Figures 26, 27, 28, 29, 30, and 31, we include select examples of PRM beam search. We include the PRM score, between 0 and 1, for each step in the examples.

If $f(x) = \frac{3x-2}{x-2}$, what is the value of $f(-2) + f(-1) + f(0)$? Express your answer as a common fraction.

Attempt 1:

Let's evaluate $f(-2)$.
 We have $f(-2) = \frac{3(-2)-2}{-2-2} = -\frac{4}{-4} = \frac{1}{1} = 1$.
 So $f(-2) = 1$.
 Let's evaluate $f(-1)$.
 We have $f(-1) = \frac{3(-1)-2}{-1-2} = -\frac{5}{-3} = \frac{5}{3}$.
 So $f(-1) = \frac{5}{3}$.
 Let's evaluate $f(0)$.
 We have $f(0) = \frac{3(0)-2}{0-2} = \frac{-2}{-2} = 1$.
 So $f(0) = 1$.
 So we want to compute
 $1 + \frac{5}{3} + 1 = \frac{3}{3} + \frac{5}{3} + \frac{3}{3} = \frac{11}{3}$.
 So the answer is $\boxed{\frac{11}{3}}$.
 ##### $\frac{11}{3}$

Attempt 2:

Let's first find $f(-2)$.
 $f(-2) = \frac{3(-2)-2}{-2-2} = \frac{-6-2}{-4} = \frac{-8}{-4} = 2$.
 Now let's find $f(-1)$.
 $f(-1) = \frac{3(-1)-2}{-1-2} = \frac{-3-2}{-3} = \frac{-5}{-3} = \frac{5}{3}$.
 Now let's find $f(0)$.
 $f(0) = \frac{3(0)-2}{0-2} = \frac{-2}{-2} = 1$.
 So we have $f(-2) + f(-1) + f(0) = 2 + \frac{5}{3} + 1 = \frac{7}{3} + \frac{5}{3} + \frac{3}{3} = \frac{15}{3} = 5$.
 So the answer is $\boxed{5}$.
 ##### 5

Attempt 3:

Let's plug in -2 , -1 , and 0 for x in $f(x)$.
 We get $f(-2) = \frac{3(-2)-2}{-2-2} = \frac{-8}{-4} = \frac{2}{1} = 2$, $f(-1) = \frac{3(-1)-2}{-1-2} = \frac{-5}{-3} = \frac{5}{3}$, $f(0) = \frac{3(0)-2}{0-2} = \frac{-2}{-2} = 1$.
 So we have $2 + \frac{5}{3} + 1 = \frac{6}{3} + \frac{5}{3} + \frac{3}{3} = \frac{14}{3}$.
 So the answer is $\boxed{\frac{14}{3}}$.
 ##### $\frac{14}{3}$

Figure 19: Revision model example 1. The model calculates the sum at the end incorrectly on the first two attempts, but on the third attempt it succeeds and gets the answer correct.

In how many ways can 7 people sit around a round table if no two of the 3 people Pierre, Rosa, and Thomas can sit next to each other? (Seating arrangements which are rotations of each other are treated as the same.)

Attempt 1:

Let's start by figuring out how many ways there are to seat the three people who can't sit next to each other.
 So we need to seat them in three seats that are not right next to each other.
 So there are 4 chairs that they can sit in.
 So we have 4 slots to fill with 3 people.
 We can do that in 4 choose 3 ways.
 That is $4!/(3! * 1!) = 4$ ways.
 Now we need to figure out how many ways there are to seat the remaining 4 people.
 So we can seat the first person in any of the seven chairs.
 And then we can seat the second person in any of the remaining six chairs.
 And so on.
 So we can seat the four people in $7 * 6 * 5 * 4$ ways.
 So we need to multiply the number of ways to seat the three people who can't sit next to each other, and the number of ways to seat the other four people.
 That is $4 * 7 * 6 * 5 * 4 = 840$.
 So the answer is $\boxed{840}$.
 #### 840

Attempt 2:

So we have to figure out how many ways there are to seat 7 people around a table if no two of the 3 people can sit next to each other.
 Let's start by seating the 3 people who can't sit next to each other.
 We know that they can't sit next to each other, but they can sit on the ends of the table.
 So they can sit in 3 different ways.
 Now let's seat the 4 people who can sit anywhere.
 We can do that in 4! ways.
 Now let's seat the 3 people who can't sit next to each other.
 Let's seat them in the 3 spots we found earlier.
 So there are 3 different ways to seat them.
 So we have $3 * 4! * 3 = 3 * 24 * 3 = 216$ ways to do it.
 So the answer is $\boxed{216}$.
 #### 216

Attempt 3:

So we have to find the number of ways to arrange 4 people around a round table such that 3 of the people can't sit next to each other.
 So let's start with the 3 people who can sit next to each other.
 There are 3! ways to arrange them.
 Now we have to arrange the other 4 people.
 There are 4! ways to do that.
 So the total number of ways to arrange the 7 people around the table is $3! * 4! = 6 * 24 = 144$.
 So the answer is $\boxed{144}$.
 #### 144

Figure 20: Revision model example 2. On the first attempt the model takes the incorrect approach, on the second attempt it gets closer but then makes a mistake towards the end. On the final attempt it gets to the correct answer.

Find the product of $6_8 \cdot 7_8$. Express your answer in base 8.
 52_8

Attempt 1:

So we have $6_8 \cdot 7_8$.
 Right.
 So, we can convert these numbers to base 10.
 Right, we have $6_8 = 6_{10}$ and $7_8 = 7_{10}$.
 So our equation becomes $6_{10} \cdot 7_{10}$.
 So, we have $6_{10} \cdot 7_{10} = 42_{10}$.
 So, we need to convert 42_{10} to base 8.
 Right.
 So, we know that $4_8 = 4_{10}$ and $2_8 = 2_{10}$.
 So, we can divide 42_{10} by 8_{10} and get $5 \text{ R } 2$.
 So, our answer is 52_8 .
 Correct.
 #### 52

Attempt 2:

Well, we can start by expanding the numbers in base 10.
 Right. So $6_8 = 6_{10}$ and $7_8 = 7_{10}$.
 So we get $6_{10} \cdot 7_{10}$.
 Right. The product is 42_{10} .
 So, we need to convert 42_{10} to base 8.
 Right. So, we divide 42_{10} by 8 and get a remainder of 2_{10} and a quotient of 5_{10} .
 So, we write 52_8 .
 Correct.
 #### 52_8

Figure 21: Revision model example 3. On the first attempt the model makes a mistake with the formatting of the final answer; it corrects this on the second attempt.

Find the product of $6_8 \cdot 7_8$. Express your answer in base 8 .
 52_8

Attempt 1:

Let's multiply them together.
 We get $6_8 \cdot 7_8 = 42_8$.
 So the answer is 42_8 .
 #### 42_8

Attempt 2:

We know that $6_8 = 6$, and $7_8 = 7$. So the problem is finding $6 \cdot 7$.
 Yep.
 $6 \cdot 7 = 42$.
 Correct. So that means $6_8 \cdot 7_8 = 42_8$.
 So our answer is 42_8 .
 #### 42_8

Attempt 3:

Let's multiply 6_8 by 7_8 in base 8 .
 $6 \cdot 7 = 42$ in base 10 .
 That's correct. So $6_8 \cdot 7_8 = 42_{10} = 40_8 + 2_8$.
 We can rewrite this as $40_8 + 2_8 = 40_8 + 10_8 + 10_8 + 8_8 + 8_8 + 8_8 + 8_8 + 8_8 + 1_8$.
 And we can rewrite that as $40_8 + 10_8 + 10_8 + 10_8 + 10_8 + 10_8 + 10_8 + 10_8 + 1_8$.
 And we can rewrite that as $50_8 + 1_8$.
 So $6_8 \cdot 7_8 = 51_8$.
 #### 51

Attempt 4:

So $6_8 \cdot 7_8$ is $6 \cdot 7$ in base 8 .
 That's right.
 Let's convert 6 and 7 to decimal. $6 = 6_8 = 6 \cdot 8^0 + 0 \cdot 8^1 = 6$.
 Right. And $7 = 7_8 = 7 \cdot 8^0 + 0 \cdot 8^1 = 7$.
 Right. So $6 \cdot 7 = 6 \cdot 7 = 42$.
 And 42 in octal is 52_8 .
 That's correct.
 #### 52_8

Figure 22: Revision model example 4. On the first few attempts the model fails the base 10 to base 8 conversion. On the final attempt it makes the correct calculation.

Convert the point $(0,3)$ in rectangular coordinates to polar coordinates. Enter your answer in the form (r,θ) , where $r > 0$ and $0 \leq \theta < 2\pi$.

Attempt 1:

To convert $(0, 3)$ to polar coordinates, we need to find the radius r and the angle θ .
 We know that the x -coordinate of the point is $r\cos\theta$ and the y -coordinate of the point is $r\sin\theta$.
 So $r\cos\theta = 0$ and $r\sin\theta = 3$.
 And we know that $\cos\theta = 0$ when $\theta = \frac{\pi}{2}$ and $\sin\theta = 3$ when $\theta = \frac{3\pi}{2}$.
 So we know that $(0, 3)$ corresponds to the point (r, θ) where $\theta = \frac{3\pi}{2}$ and $r\cos\theta = 0$ and $r\sin\theta = 3$.
 So $r = \frac{3}{\sin\theta} = \frac{3}{\sin(\frac{3\pi}{2})} = \frac{3}{-1} = -3$.
 So the polar coordinates of $(0, 3)$ are $(-3, \frac{3\pi}{2})$.
 #### $(-3, \frac{3\pi}{2})$

Attempt 2:

To convert the point $(0,3)$ in rectangular coordinates to polar coordinates, we need to find the equivalent of the angle θ that makes the polar coordinate (r, θ) equal to the rectangular coordinate (x, y) .
 We know that $(r, \theta) = (0,3)$ and $(x, y) = (0, 3)$.
 So let's set up an equation where r is equal to x and θ is equal to y .
 We have $r = x$ and $\theta = y$.
 So we can substitute these into the polar equation $r = \sqrt{x^2 + y^2}$.
 We get $\sqrt{x^2 + y^2} = x$.
 We can square both sides of the equation to get $x^2 + y^2 = x^2$.
 And we can subtract x^2 from both sides of the equation to get $y^2 = 0$.
 And we know that to satisfy $y^2 = 0$ we must have $y = 0$.
 So the angle θ that makes the polar coordinate (r, θ) equal to the rectangular coordinate (x, y) is $\theta = 0$.
 And the radius r is equal to the x -coordinate, which is $x = r = 0$.
 So the polar coordinate equivalent to $(0,3)$ is $(r, \theta) = (0,0)$.
 And we know that $r > 0$ and $0 \leq \theta < 2\pi$.
 So the polar coordinate equivalent to $(0,3)$ is $(r, \theta) = (0,0)$.
 #### $(0,0)$

Attempt 3:

To convert the point $(0,3)$ in rectangular coordinates to polar coordinates, we need to find the radius r and the angle θ .
 The x -coordinate of the point is 0, so $x = r\cos\theta = 0$.
 We know that $\cos\theta = 0$ when $\theta = \frac{\pi}{2}$ or $\theta = \frac{3\pi}{2}$.
 So we can use either $\theta = \frac{\pi}{2}$ or $\theta = \frac{3\pi}{2}$.
 The y -coordinate of the point is 3, so $y = r\sin\theta = 3$.
 We know that $\sin\theta = \frac{3}{r}$.
 So we can use the equation $\sin\theta = \frac{3}{r}$ to find r .
 We know that $\sin\frac{\pi}{2} = 1$, so $r = \frac{3}{1} = 3$.
 So our polar coordinates are $(3, \frac{\pi}{2})$ or $(3, \frac{3\pi}{2})$.
 And since $0 \leq \theta < 2\pi$, we have $\theta = \frac{\pi}{2}$.
 So our polar coordinates are $(3, \frac{\pi}{2})$.
 #### $(3, \frac{\pi}{2})$

Figure 23: Revision model example 5. On the first two attempts the model makes an error when converting euclidean to polar coordinates. On the final attempt it does not make these mistakes.

The proper divisors of 12 are 1, 2, 3, 4 and 6. A proper divisor of an integer N is a positive divisor of N that is less than N . What is the sum of the proper divisors of the sum of the proper divisors of 284?

Attempt 1:

Ok, let's find the proper divisors of 284 first.
 Right. So the proper divisors of 284 are 1, 2, 4, 71, and 142.
 So the sum of the proper divisors of 284 is $1+2+4+71+142$, which is 219.
 Now let's find the proper divisors of 219.
 The proper divisors of 219 are 1, 3, 73, and 219.
 So the sum of the proper divisors of 219 is $1+3+73+219$, which is 306.
 So the answer to the question is 306.
 #### 306

Attempt 2:

So the sum of the proper divisors of 284 is $1+2+4+71+142$.
 Ok.
 And the proper divisors of 284 are 1,2,4,71, and 142.
 Right.
 And the sum of those proper divisors is 219.
 So the sum of the proper divisors of 284 is 219.
 Right.
 So the sum of the proper divisors of the sum of the proper divisors of 284 is the sum of the proper divisors of 219.
 Right.
 And the proper divisors of 219 are 1, 3, 73, and 219.
 Right.
 So the sum of the proper divisors of 219 is $1+3+73+219=306$.
 So the sum of the proper divisors of the sum of the proper divisors of 284 is 306.
 Right.
 So the answer to the question is 306.
 #### 306

Attempt 3:

Right, so the proper divisors of 284 are 1, 2, 4, 71, and 142.
 So the sum of the proper divisors of 284 is $1+2+4+71+142$, which is 220.
 Now let's find the proper divisors of 220.
 The proper divisors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 55, and 110.
 So the sum of the proper divisors of 220 is $1+2+4+5+10+11+20+22+44+55+110$, which is 284.
 So the answer to the question is 284.
 #### 284

Figure 24: Revision model example 6. On the first two attempts the model makes a mistake when summing the proper divisors of 284. On the third attempt, it evaluates this sum correctly.

1674 If $-6 \leq a \leq -2$ and $3 \leq b \leq 5$, what is the greatest possible value of
 1675 $\displaystyle\left(a+\frac{1}{b}\right)\left(\frac{1}{b}-a\right)$? Express your answer as a
 1676 common fraction.
 1677
 1678 Attempt 26:
 1679 So we have $-6 \leq a \leq -2$ and $3 \leq b \leq 5$.
 1680 And we want to find the greatest possible value of
 1681 $\displaystyle\left(a+\frac{1}{b}\right)\left(\frac{1}{b}-a\right)$.
 1682 So we want to find the greatest possible value of
 1683 $\displaystyle\left(a+\frac{1}{b}\right)\left(\frac{1}{b}-a\right)$, where $-6 \leq a \leq -2$ and $3 \leq b \leq 5$.
 1684 Ok, let's start by finding the greatest possible value of a and the least
 1685 possible value of b .
 1686 Ok, so the greatest possible value of a is -2 .
 1687 And the least possible value of b is 3 .
 1688 So we want to find the greatest possible value of
 1689 $\displaystyle\left(a+\frac{1}{b}\right)\left(\frac{1}{b}-a\right)$, where $a = -2$ and $b = 3$.
 1690 So we want to find the greatest possible value of
 1691 $\displaystyle\left(-2+\frac{1}{3}\right)\left(\frac{1}{3}-(-2)\right)$.
 1692 So we want to find the greatest possible value of
 1693 $\displaystyle\left(-2+\frac{1}{3}\right)\left(\frac{1}{3}+2\right)$.
 1694 So we want to find the greatest possible value of
 1695 $\displaystyle\left(-2+\frac{1}{3}\right)\left(\frac{5}{3}\right)$.
 1696 So we want to find the greatest possible value of
 1697 $\displaystyle\left(\frac{-6+1}{3}\right)\left(\frac{5}{3}\right)$.
 1698 So we want to find the greatest possible value of
 1699 $\displaystyle\left(\frac{-5}{3}\right)\left(\frac{5}{3}\right)$.
 1700 So we want to find the greatest possible value of $\displaystyle\frac{-5}{3} \cdot \frac{5}{3}$.
 1701 So the greatest possible value of $\displaystyle\frac{-25}{9}$.
 1702 So the greatest possible value of $\displaystyle\left(a+\frac{1}{b}\right)\left(\frac{1}{b}-a\right)$, where $-6 \leq a \leq -2$ and $3 \leq b \leq 5$ is $\boxed{-\frac{25}{9}}$.
 1703 ##### $\frac{-25}{9}$
 1704
 1705 Attempt 27:
 1706 Ok, so we want to find the greatest possible value of
 1707 $\displaystyle\left(a+\frac{1}{b}\right)\left(\frac{1}{b}-a\right)$.
 1708 So we want to find the greatest possible value of
 1709 $\displaystyle\left(a+\frac{1}{b}\right)\left(\frac{1}{b}-a\right)$, where $-6 \leq a \leq -2$ and $3 \leq b \leq 5$.
 1710 Ok, so let's start by finding the greatest possible value of a and the smallest
 1711 possible value of b .
 1712 Ok, so the greatest possible value of a is -2 .
 1713 And the smallest possible value of b is 3 .
 1714 So we want to find the greatest possible value of
 1715 $\displaystyle\left(a+\frac{1}{b}\right)\left(\frac{1}{b}-a\right)$, where $a = -2$ and $b = 3$.
 1716 So we want to find the greatest possible value of
 1717 $\displaystyle\left(-2+\frac{1}{3}\right)\left(\frac{1}{3}-(-2)\right)$.
 1718 So we want to find the greatest possible value of
 1719 $\displaystyle\left(-2+\frac{1}{3}\right)\left(\frac{1}{3}+2\right)$.
 1720 So we want to find the greatest possible value of
 1721 $\displaystyle\left(-2+\frac{1}{3}\right)\left(\frac{5}{3}\right)$.
 1722 So we want to find the greatest possible value of
 1723 $\displaystyle\left(\frac{-6+1}{3}\right)\left(\frac{5}{3}\right)$.
 1724 So the greatest possible value of $\displaystyle\frac{-5}{3} \cdot \frac{5}{3} = \frac{-25}{9}$.
 1725 So the greatest possible value of $\displaystyle\left(a+\frac{1}{b}\right)\left(\frac{1}{b}-a\right)$, where $-6 \leq a \leq -2$ and $3 \leq b \leq 5$ is $\boxed{-\frac{25}{9}}$.
 1726 ##### $-\frac{25}{9}$

Figure 25: Revision model example 7. On the first attempt the model evaluates $\frac{1}{3} + 2$ incorrectly. On the second attempt it corrects this error.

1721 The sum of the first n terms in the infinite geometric sequence $\frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \dots$ is $\frac{255}{512}$. Find n .
 1722 The sum of the first n terms in the sequence is just $\frac{a}{1-r}$, where a is the first term in the sequence and r is the common ratio. [score = 0.085]
 1723 And if we take the sum of the first n terms in this particular sequence and set it equal to $\frac{255}{512}$, we get $n = \boxed{7}$. [score = 0.11]
 1724 Actually, it turns out that there are no 7s in the decimal expansion of $\frac{255}{512}$. [score = 0.10]
 1725 Right, so we can set $\frac{1}{4}(1-\frac{1}{2^n}) = \frac{1}{512}$ and solve for n . [score = 0.10]
 1726 So we get $n = 7$. [score = 0.10]
 1727 That gives $1 - (\frac{1}{2})^n = \frac{255}{512}$. [score = 0.15]
 1728 So, $1 - (\frac{1}{2})^n = \frac{255}{512}$. [score = 0.15]
 1729 ##### 7 [score = 0.30]

Figure 26: PRM beam search example 1.

The sum of the first n terms in the infinite geometric sequence $\sum_{k=0}^{\infty} \left(\frac{1}{4}\right)^k$ is $\frac{4}{3}$. Find n .

Since the common ratio in each term is $\frac{1}{4}$, the sum of the first n terms of this geometric sequence is $\frac{a_1(1-r^n)}{1-r}$, where a_1 is the first term and r is the common ratio. [score = 0.09]
 So, the sum of the first n terms of this geometric sequence is $\frac{\frac{1}{4}(1-\left(\frac{1}{4}\right)^n)}{1-\frac{1}{4}}$. [score = 0.09]
 So, the sum of the first n terms of this geometric sequence is $\frac{\frac{1}{4}(1-\left(\frac{1}{4}\right)^n)}{\frac{3}{4}}$. [score = 0.11]
 Right. And we want $\frac{4}{3} = \frac{\frac{1}{4}(1-\left(\frac{1}{4}\right)^n)}{\frac{3}{4}}$. So we have $\frac{1}{4}(1-\left(\frac{1}{4}\right)^n) = \frac{4}{3} \cdot \frac{3}{4}$. [score = 0.14]
 We can multiply both sides by 2. [score = 0.10]
 We have $\frac{1}{4}(1-\left(\frac{1}{4}\right)^n) = \frac{4}{3}$. [score = 0.14]
 So $\frac{1}{4}(1-\left(\frac{1}{4}\right)^n) = 1 - \frac{1}{4}$. [score = 0.16]
 So, $\frac{1}{4}(1-\left(\frac{1}{4}\right)^n) = \frac{3}{4}$. [score = 0.30]
 And then, $1-\left(\frac{1}{4}\right)^n = 3$. [score = 0.80]
 So $1-\left(\frac{1}{4}\right)^n = 3$. [score = 0.78]
 Right. And since the bases are the same, the exponents must be equal. [score = 0.83]
 ##### 0 [score = 0.69]

Figure 27: PRM beam search example 2.

$441+2(21)(19)+361=x^2$. Solve for x .

We know that $a^2+2ab+b^2=(a+b)^2$. [score = 0.33]
 So $441+2(21)(19)+361=(21+19)^2$. [score = 0.49]
 So we get $(21+19)^2=x^2$. [score = 0.56]
 So we get $x=40$. [score = 0.90]
 So we get $x=1600$. [score = 0.94]
 And we know that $x=1600$. [score = 0.95]
 That's our answer. [score = 0.93]
 So $441+2(21)(19)+361=1600$. [score = 0.93]
 So $x=1600$. [score = 0.93]
 Right. So $441+2(21)(19)+361=1600$. [score = 0.94]
 ##### 1600 [score = 0.91]

Figure 28: PRM beam search example 3.

Determine the coordinates of the point P on the line $y=-x+6$ such that P is equidistant from the points $A(10,-10)$ and $O(0,0)$ (that is, so that $PA=PO$). Express your answer as an ordered pair (a,b) .

So we're given the line $y=-x+6$ and we need to find the point on that line where the line segment from that point to the point with coordinates $(10,-10)$ is the same length as the line segment from the point to the origin. [score = 0.01]
 Right. So one way to do that is to use the distance formula. Let's find the distance between P and $(10,-10)$ and the distance between P and the origin. [score = 0.01]
 Right. So the distance between P and $(10,-10)$ will be the square root of $(x-10)^2+(y+10)^2$. Similarly the distance between P and the origin will be the square root of x^2+y^2 . [score = 0.01]
 So the two distances must be equal. So $(x-10)^2+(y+10)^2=(x)^2+(y)^2$. Expanding the left side gives $x^2-20x+100+y^2+20y+100=x^2+y^2$. [score = 0.02]
 So now we can subtract the x^2+y^2 from both sides to get $-20x+20y+200=0$. [score = 0.02]
 But we know that $y=-x+6$. So we can substitute that value in to get $-20x+20(-x+6)+200=0$. [score = 0.03]
 Then we can distribute the 20 to get $-20x-20x+120+200=0$. Now we can combine the terms to get $-40x+320=0$. Then we can subtract 320 from both sides to get $-40x=-320$. Divide both sides by -40 to get $x=8$. [score = 0.12]
 We know that $y=-x+6$ and $x=8$, so if we substitute 8 for x we get $y=-8+6$. [score = 0.22]
 Right. Plugging in the values for x and y we found into the equation $y=-x+6$, we get $y=-8+6$, which simplifies to $y=-2$. [score = 0.24]
 So the point $(8,-2)$ is equidistant from the given points. [score = 0.36]
 That's correct. [score = 0.35]
 ##### (8,-2) [score = 0.39]

Figure 29: PRM beam search example 4.

Determine the coordinates of the point P on the line $y=-x+6$ such that P is equidistant from the points $A(10,-10)$ and $O(0,0)$ (that is, so that $PA=PO$). Express your answer as an ordered pair (a,b) .

So we're given the line $y=-x+6$ and we need to find the point on that line where the line segment from that point to the point with coordinates $(10,-10)$ is the same length as the line segment from the point to the origin. [score = 0.01]
 Right. So one way to do that is to use the distance formula. Let's find the distance between P and $(10,-10)$ and the distance between P and the origin. [score = 0.01]
 Right. So the distance between P and $(10,-10)$ will be the square root of $(x-10)^2+(y+10)^2$. Similarly the distance between P and the origin will be the square root of x^2+y^2 . [score = 0.01]
 So the two distances must be equal. So $(x-10)^2+(y+10)^2=(x)^2+(y)^2$. Expanding the left side gives $x^2-20x+100+y^2+20y+100=x^2+y^2$. [score = 0.02]
 So now we can subtract the x^2+y^2 from both sides to get $-20x+20y+200=0$. [score = 0.02]
 But we know that $y=-x+6$. So we can substitute that value in to get $-20x+20(-x+6)+200=0$. [score = 0.03]
 Then we can distribute the 20 to get $-20x-20x+120+200=0$. Now we can combine the terms to get $-40x+320=0$. Then we can subtract 320 from both sides to get $-40x=-320$. Divide both sides by -40 to get $x=8$. [score = 0.12]
 So now we know that $x=8$. Then we can substitute that value into the equation $y=-x+6$ to find y . That gives $y=-8+6$ which simplifies to $y=-2$. [score = 0.30]
 ##### 3,3 [score = 0.12]

Figure 30: PRM beam search example 5.

If 4 daps = 7 yaps, and 5 yaps = 3 baps, how many daps equal 42 baps?

So 4 daps is equal to 7 yaps, so that means that 1 dap is equal to 7/4 yaps or 1.75 yaps. [score = 0.21]
 And 5 yaps is equal to 3 baps, so that means that 1 yap is equal to 3/5 baps or 0.6 baps. [score = 0.32]
 And we can plug these values in to find the answer. [score = 0.31]
 So 42 baps is equal to 42/0.6 yaps or 70 yaps. [score = 0.60]
 And 70 yaps is equal to 70/1.75 daps or 40 daps. [score = 0.87]
 So the answer is 40. [score = 0.92]
 And we're done! [score = 0.91]
 And that is our final answer. [score = 0.90]
 [score = 0.91]
 So 40 daps is equal to 42 baps. [score = 0.90]
 And we're done! [score = 0.91]
 And that is our final answer. [score = 0.90]
 [score = 0.64]
 [score = 0.78]
 ##### 40 [score = 0.83]

Figure 31: PRM beam search example 6.