

Z1: Efficient Test-time Scaling with Code

Zhaojian Yu¹ Yinghao Wu¹ Yilun Zhao² Arman Cohan² Xiao-Ping Zhang^{1*}

¹Tsinghua University ²Yale University

Abstract

Large Language Models (LLMs) can achieve enhanced complex problem-solving through test-time computing scaling, yet this often entails longer contexts and numerous reasoning token costs. In this paper, we propose an efficient test-time scaling method that trains LLMs on code-related reasoning trajectories, facilitating their reduction of excess thinking tokens while maintaining performance. First, we create Z1-Code-Reasoning-107K, a curated dataset of simple and complex coding problems paired with their short and long solution trajectories. Second, we present a novel Shifted Thinking Window to mitigate overthinking overhead by removing context-delimiting tags (e.g., `<think>...</think>`) and capping reasoning tokens. Trained with long and short trajectory data and equipped with Shifted Thinking Window, our model, Z1-7B, demonstrates the ability to adjust its reasoning level as the complexity of problems and exhibits efficient test-time scaling across different reasoning tasks that matches R1-Distill-Qwen-7B performance with about 30% of its average thinking tokens. Notably, fine-tuned with only code trajectories, Z1-7B demonstrates generalization to broader reasoning tasks (47.5% on GPQA Diamond). Our analysis of efficient reasoning elicitation also provides valuable insights for future research.¹

1 Introduction

Large Reasoning Models (LRMs), such as OpenAI o1 (Jaech et al., 2024) and DeepSeek R1 (Guo et al., 2025), have demonstrated remarkable advances in complex reasoning tasks through test-time compute scaling (Wei et al., 2022), particularly in competitive mathematics and programming. These models, trained with large-scale Reinforcement Learning (RL) (Lightman et al., 2023;

*Corresponding author

¹Our model, data, and code are open-source at <https://github.com/efficientscaling/Z1>

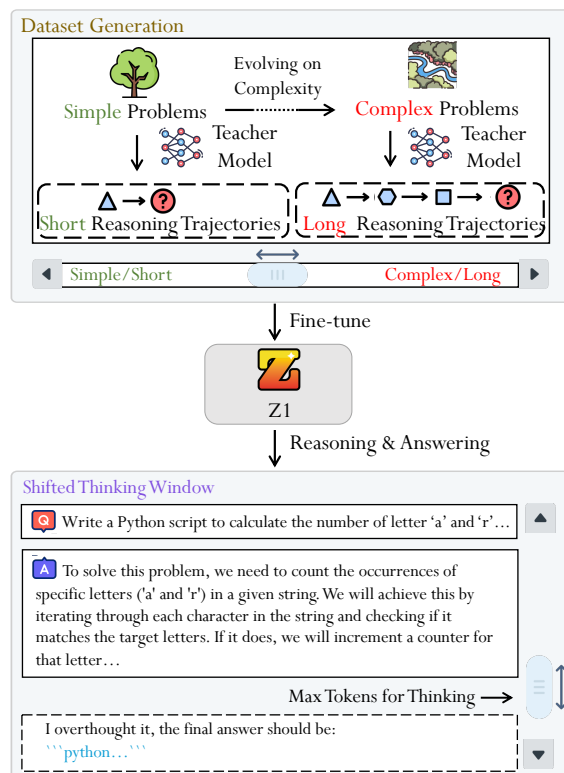


Figure 1: Overview of Z1 training and inference. Fine-tuned with long and short trajectory data, Z1 could solve simple and complex problems in shifted thinking window efficiently.

Feng et al., 2023), have emerged step-by-step reasoning abilities to solve complex problems effectively. However, the elaborate reasoning process also leads to super long contexts and numerous thinking tokens, challenging the efficient utilization of LRMs.

Existing open-source works, such as s1 (Muenighoff et al., 2025) and LIMO (Ye et al., 2025), train non-reasoning models into reasoning models with manually curated problems and distilled long chain-of-thought (CoT) (Wei et al., 2022) trajectories yet do not address the challenges posed by test-time compute scaling with respect to long

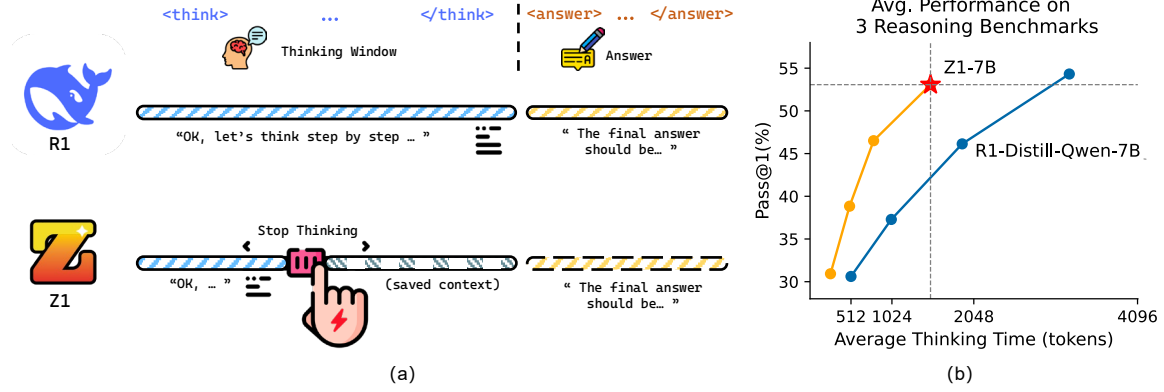


Figure 2: Comparing Z1 with R1-like models on context window (a) and test-time scaling (b). Z1 models exhibit more efficient test-time compute scaling than R1-Distill-7B, with its shifted thinking window. Z1-7B exhibits efficient test-time scaling across 3 different reasoning tasks (LiveCodeBench, MATH500, GPQA Diamond) and matches R1-Distill-Qwen-7B performance with about 30% of its average thinking tokens.

contexts and an excessive thinking tokens. For example, s1 (Muennighoff et al., 2025) introduces budget forcing, which either appends ending words to truncate reasoning processes or extrapolation words (e.g., "wait") to inspire the model to continue thinking, thereby precisely controlling context length. While extrapolation words can control model reasoning, direct truncations may disrupt the model’s thinking process, consequently degrading performance. Thus, we propose the problem: "Is there an efficient test-time scaling way to reduce the model’s thinking tokens consumption while preserving its reasoning performance?"

In this paper, we implement efficient test-time scaling with code-related trajectory fine-tuning and present the shifted thinking mode of LRMs: **weak reasoning to simple problems, strong reasoning to complex problems**, which significantly reduces the thinking tokens consumption of LRMs in problem solving. Specifically, we create the **Z1-Code-Reasoning-107K** dataset, comprising 107K simple and complex code-related problems paired with their reasoning traces distilled from the QwQ-32B-Preview (Team, 2025c) model. We train Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) into a reasoning model, with this 107K code-related long and short trajectory dataset. We eliminate the context split with delimiters (e.g., <think>...</think>) and introduce a novel Shifted Thinking Window: (I) For simple problems, the models generate solutions with in a low reasoning token computation. (II) For complex problems, we cap the thinking tokens; if the model outputs exceed this threshold, we append a hint phrase to the end of the reasoning trace, forcing it to produce an answer based on the existing thought process. Shifted Thinking

Window enables model that get trained with long and short trajectories to adjust its reasoning level as the complexity of problems, thereby avoiding the overthinking of LRMs. Fine-tuned on long and short reasoning trajectories and equipped with the Shifted Thinking Window, our model **Z1-7B** exhibits efficient test-time scaling across different reasoning tasks and matches R1-Distill-Qwen-7B performance with about 30% (as shown in Figure 2) its average thinking tokens. Furthermore, our systematic analysis of the benefits of reasoning elicitation provides valuable insights for future research, contributing to the development of more efficient LRMs.

2 Efficient Test-time Scaling with Code

As illustrated in Figure 1, to achieve efficient test-time scaling, we train non-reasoning models of code-related reasoning trajectories with varying lengths and introduce shifted thinking window to replace the context split with delimiters (e.g., <think>...</think>). After supervised fine-tuning (SFT), our model could adjust its reasoning level according to the complexity of the input problem. In this section, we detail our methodology, including the creation of the reasoning dataset with code (Section 2.1) and the implementation of the shifted thinking window (Section 2.2).

2.1 Dataset Creation

We create the efficient test-time scaling dataset that integrates both short reasoning trajectories for simple problems and strong reasoning trajectories for complex problems (as shown in Appendix D), emphasizing diverse reasoning trajectory lengths in

the training set. However, existing reasoning trajectory datasets predominantly feature complex problems with long chains of thought (CoT), posing a challenge for training efficient reasoning models due to the lack of short and straightforward trajectories. To address this, we approach the problem from the perspective of evolving question complexity and select problems from Code Evol-Instruct dataset (Xu et al., 2024), which evolves in depth and breadth to cover a wide range of complexities and has proven effective in non-reasoning models (Luo et al., 2023).

We generate reasoning trajectories using QwQ-32B-preview (where no explicit context splits are adopted) and truncate the trajectories length to 8192 tokens, removing approximately 3% of samples with repetitive reasoning processes to mitigate excessive thinking in the training data. The remaining reasoning trajectories, paired with their problems, constitute the Z1-code-reasoning-107K dataset, with less than 1% of the data being truncated. More details is shown in Appendix B.

To further analyze our dataset, we compare the top-50 trigram word frequencies of Z1-Code-Reasoning-107K with OpenThoughts-114K (Team, 2025b) dataset². Z1-code-reasoning-107K exclusively contains code-related reasoning trajectories, while OpenThoughts-114K is a reasoning dataset distilled from DeepSeek R1, featuring 114K high-quality examples spanning math, science, code, and puzzles. Figure 4 illustrates a trend in the word frequency distributions of Z1-code-reasoning-107K and OpenThoughts-114K: high-frequency trigrams exhibit homogeneity, while mid-frequency trigrams show differentiation. For example, the high-frequency trigrams in both datasets (e.g., "I need to," "we need to," "the number of") indicate the model’s summarization of the next reasoning step, which highlights the commonality between code trajectory and other complex problems. In contrast, mid-frequency trigrams in Z1-code-reasoning-107K, such as "iterate through the" and "for each," capture loop-based logic characteristic of code-related trajectories, distinct from the mathematical logic exemplified by trigrams (e.g., "a+b", "equal to the") in OpenThoughts-114K. This underscores the unique reasoning patterns inherent in code-related trajectories.

²<https://huggingface.co/datasets/open-thoughts/OpenThoughts-114k>

2.2 Shifted Thinking Window

To enforce a "think-before-answer" pattern, existing LRMs like DeepSeek R1 typically use delimiters (e.g., <think>...</think>) to split the context window into two parts, where the model first reasons in the thinking window and then outputs the final answer in the answering window. However, this pattern often introduces unnecessary reasoning when processing simple problems that do not require deep thought. In the training and inference of our model, we eliminate this context split, allowing the model to flexibly fine-tune and generalize across short and long trajectories and avoiding the overthinking for simple problems. We refer it to Shifted Thinking Window, where the model’s context window is not rigidly divided into two parts by delimiters but instead of a shifted window (Examples are shown in Appendix D): (I) For simple problems, the model fine-tuned on both short and long trajectories can directly output concise reasoning and answers within the context. (II) For complex problems, we cap the maximum thinking length within which the model can either reason or provide an answer; if the reasoning trajectory exceeds this maximum length, the end of the model’s output will be appended with a hint to enforce a direct answer. The essence of shifted thinking: weak reasoning for simple problems and strong reasoning for complex problems significantly reduces unnecessary reasoning by model, thereby demonstrating more efficient test-time compute scalings.

3 Experiments

3.1 Implementation Details

Following the previous work (Muennighoff et al., 2025), we take a model that has already been pretrained and instruction-tuned, and further fine-tune it for reasoning. Specifically, we select Qwen2.5-Coder-Instruct (Hui et al., 2024) series models, which have already achieved good performance on various code-related benchmarks. For all training samples, we avoid using delimiters (e.g., <think>...</think>) to separate the whole trajectory into the thinking and answering parts. This adjustment allows the LRM to avoid mandated overthinking, enabling more automatic and efficient test-time scaling: weak reasoning for simple problems and strong reasoning for complex problems.

We finetune the Qwen-2.5-Coder-7B-Instruct on our Z1-Code-Reasoning-107K dataset, yielding Z1-7B. We do not compute loss on questions, only

Model	Data Source	MATH 500	GPQA Diamond	LiveCode Bench	BigCode Bench-Hard	AVG
API only						
o1-preview	N/A	85.5	73.3	43.2	23.0	56.3
o1-mini	N/A	90.0	60.0	53.7	27.7	57.9
Open Weights						
Deepseek-R1	N/A	97.3	71.5	77.9	29.7	67.6
R1-Distill-Qwen-32B	R1/800K	94.3	62.1	-	23.6	-
R1-Distill-Qwen-7B	R1/800K	83.3	49.1	40.5	3.4	44.1
QwQ-32B-Preview	N/A	90.6	60.0	59.9	25.0	58.9
Non-reasoning Model						
Deepseek-V3	N/A	90.2	59.1	56.3	27.7	58.3
GPT-4o-0513	N/A	75.8	46.5	43.4	25.0	47.7
Qwen2.5-Coder-7B-Ins	N/A	68.6	37.4	32.3	20.3	39.7
Open Weights and Data						
Sky-T1-32B-Preview	QwQ/17K	88.6	56.8	-	26.4	-
s1.1-7B	R1/1K	79.2	31.8	15.2	4.7	31.7
OpenThinker-7B	R1/114K	83.0	42.4	25.3	17.6	42.1
Z1-7B	QwQ/107K	76.4	47.5	35.3	22.3	45.4

Table 1: Results on 4 benchmarks. We evaluate Z1 models with shifted thinking window. For other models without a reported score, we budget force it by adding "the final answer is:". Model details are shown in [Appendix C](#).

on reasoning trajectories and solutions. For fine-tuning hyperparameters, we train our model with a learning rate of $1e-5$ warmed up linearly for 100 steps and then decayed over the rest of training (836 steps in total training) following a cosine scheduler. We train all the models in bfloat16 precision with Pytorch Fully Shard Data Parallel (FSDP) and set a global batch size to 128 for 2 epochs using 8 NVIDIA A100-80G GPUs. All other settings not mentioned in this paper follow the default values of Huggingface Trainer ([Hugging Face, 2025](#)).

3.2 Main Result

[Table 1](#) presents the results of Z1 and other models on 4 benchmarks, highlighting the following salient observations:

(1) Z1 models achieve comparable performance level with GPT-4o on benchmarks of complex problems. (Avg. 45.4 vs 47.7) This result highlights the success of test-time scaling with code, where performance improvements are achieved by leveraging extended reasoning traces during inference, rather than solely relying on increased model size. **(2) Trained with trajectory data with code, Z1-7B outperforms other 7B-scale language reasoning models.** This outcome underscores the effectiveness of our test-time scaling approach, particularly when fine-tuned with code-related reasoning data. **(3) Fine-tuning the model exclusively**

with code-related reasoning data enables it to generalize across different domains. Z1 models, fine-tuned on large amount of trajectories data with code, displays superior generalization on GPQA Diamond (47.5%) and MATH500 (76.4%). This suggests the effectiveness of code-related trajectory training for language reasoning elicitation.

3.3 Test-time Scaling Comparison

To further compare the reasoning efficiency of Z1 to other models, we analyze the test-time scaling of Z1-7B and R1-Distill-Qwen-7B on three benchmarks: MATH500, GPQA, and LiveCodeBench. We equip Z1-7B with Shifted Thinking Window by imposing a cap of different maximum thinking tokens. For R1-Distill-Qwen-7B, We budget force ([Muennighoff et al., 2025](#)) it by adding "the final answer is:", since R1-Distill-Qwen-7B can't adapt to the shifted thinking window without long and short trajectory fine-tuning. [Figure 3](#) illustrates the reasoning efficiency of Z1-7B compared to the baseline models, revealing the following key observations: **Z1-7B demonstrates more efficient test-time scaling than R1-Distill-Qwen-7B on reasoning tasks, by achieving comparable results with significantly fewer thinking tokens.** Z1-7B exhibits superior test-time scaling efficiency compared to R1-Distill-Qwen-7B on reasoning tasks, by delivering better performance with

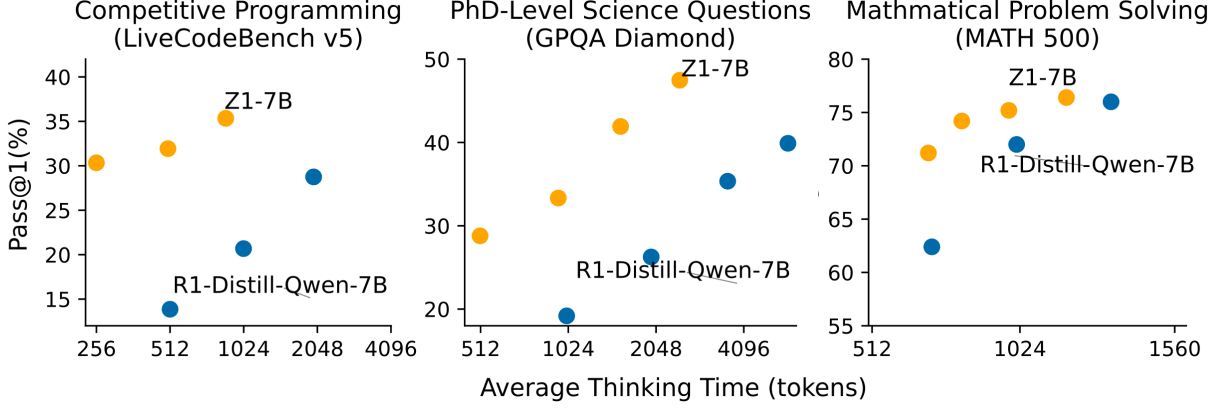


Figure 3: **Test-time scaling comparison between Z1-7B and R1-Distill-Qwen-7B.** R1-Distill-Qwen-7B can’t adapt to the shifted thinking window without long and short trajectory fine-tuning.

significantly fewer thinking tokens. For example, Z1-7B outperforms R1-Distill-Qwen-7B while requiring only half the average thinking time (approximately 2,000+ tokens) on the GPQA Diamond benchmark, whereas R1-Distill-Qwen-7B relies on a budget exceeding 4,096 thinking tokens. This underscores Z1-7B’s remarkable efficiency in reasoning-intensive tasks. In contrast, R1-Distill-Qwen-7B demonstrates notably weaker performance at lower ATT, only achieving competitive results when ATT is substantially increased.

3.4 Data Ablations

To further investigate the critical factors influencing effective reasoning elicitation in training data, we designed an ablation study with random sampling and greedy sampling strategies (as shown in Algorithm 1) and obtain representative subsets with two key factors (**Mean Trajectory Length** and **Training Sample Size**) that influences Z1’s reasoning elicitation: (1) Random Sampling: Samples are chosen randomly, serving as a baseline for comparison. We conduct random sampling with varying sample sizes (16K, 64K) to assess the impact of training samples on efficient test-time scaling. (2) Longest Greedy Sampling: At each step, we select only the samples with the highest token counts, ensuring the subset contains the longest training examples in terms of reasoning traces. (3) Shortest Greedy Sampling: At each step, we select only the samples with the lowest token counts, maximizing the number of samples included in the subset while adhering to the token budget.

For all subsets, we calculate the Mean Trajectory

Length (MTL) as follows:

$$\text{MTL} = \frac{1}{n} \sum_{i=1}^n \text{Length}_i(\text{tokens}) \quad (1)$$

Where n denotes the number of training samples and Length_i represents the trajectory length (tokens) of the i -th training sample. For all benchmarks, we calculate the Average Thinking Time (ATT) as follows:

$$\text{ATT} = \frac{1}{n} \sum_{i=1}^n \text{Length}_i(\text{tokens}) \quad (2)$$

Where n denotes the number of problems in benchmarks and Length_i represents the trajectory length of the i -th problem. We fine-tune Qwen2.5-Coder-7B-Instruct on these representative subsets. Table 2 presents the evaluation results of models fine-tuned on different subsets of the training data, highlighting the following observations:

Impact of Mean Trajectory Length Under the same training budget of 74M tokens, we sample two subsets with different mean trajectory length using two strategies: longest-greedy and shortest-greedy sampling. As shown in Table 2, the subset sampled via longest-greedy strategy exhibits a significantly higher MTL (2,216) compared to the shortest-greedy subset (807). This difference in training trajectory length translates into notable performance distinctions during evaluation. Specifically, the model fine-tuned on the longest-greedy subset demonstrates a longer Average Thinking Time (AVG score: 1,617 vs. 1,257) and a higher Benchmark Score (AVG score: 50.8 vs. 49.1) compared to the model trained on the shortest-greedy

Subset	Full		Random		Longest	Shortest
Training Dataset						
Model Size	7B	14B	7B	7B	7B	7B
Dataset Size (Samples)	107K	107K	16K	64K	33K	90K
Dataset Size (Tokens)	124M	124M	19M	74M	74M	74M
Mean Trajectory Length	1,159	1,159	1,157	1,156	2,216	807
Evaluation						
GPQA Diamond	47.5	46.0	40.9	41.9	42.4	39.4
Average Thinking Time	2,470	1,770	1,797	2,241	2,695	1,979
LiveCode Bench	35.3	40.7	32.2	34.1	32.7	34.1
Average Thinking Time	866	888	864	811	927	763
MATH 500	76.4	81.2	72.4	74.4	77.2	73.8
Average Thinking Time	1,185	1,012	1,046	1,118	1,229	1,030
AVG	53.1	56.0	48.5	50.1	50.8	49.1
Average Thinking Time	1,507	1,223	1,236	1,390	1,617	1,257

Table 2: **Z1-Code-Reasoning-107K Data Ablations.** We use a maximum of around 4,096 thinking tokens for all scores in this table. The Random Sampling method does not alter the average trajectory length of the training samples. Both length-greedy sampling methods (longest and shortest) utilize the same training tokens (74M).

subset. These results underscore the critical role of Trajectory Length in the training set, suggesting that longer trajectories enhance the model’s capacity for test-time scaling by encouraging more deliberate and extended reasoning during inference.

Impact of Training Sample Size To investigate the effect of training sample size, we randomly sample subsets of varying sizes (16K and 64K) from the original 107K dataset and compare the resulting models performance. As shown in Table 2, the model fine-tuned on the full 107K dataset achieves an ATT of 1,507 and a Benchmark Score of 53.1, outperforming the model trained on the Random-64K subset (1,390 and 50.1, respectively). In contrast, the smallest subset, Random-16K, yields the shortest ATT (1,236) and the lowest performance (48.5). This observation indicates that a larger training sample size increase the effective thinking time thereby enhancing its overall performance, despite their MTL remaining nearly identical across the randomly sampled subsets (1,157 for 16K, 1,156 for 64K) and the full dataset (1,159).

3.5 Impact of Shifted Thinking Window

We analyze the impact of Shifted Thinking Window. As shown in Table 3, most problems in MATH and LiveCodeBench are solved before cutting off (11.4% in MATH, 4.25% in LiveCodeBench). Shifted Thinking Window only processes (shift to answer directly) these overlong problems and a part of them (2.4% vs. 11.4% in MATH, 0.9% vs.

	MATH 500	GPQA Diamond	LiveCode Bench
w/o S.T.W	74.0%	25.8%	34.5%
w/ S.T.W	76.4%	47.5%	35.4%
Cutting-off (%)	11.4%	41.4%	4.25%
Δ	+2.4%	+21.7%	+0.9%

Table 3: Performance Comparison with and without Shifted Thinking Window (S.T.W).

4.25% in LiveCodeBench) overlong problems are answered correctly. In GPQA Diamond, 41.4% of problems are cut off by Shifted Thinking Window and shifted to answer directly, resulted in 21.7% improvement than w/o shifted thinking window. Therefore, Shifted Thinking Window keeps most reasoning abilities of LRMs and avoid overlong chains to make it more efficient.

4 Conclusion

In this work, we introduce an efficient test-time scaling method to elicit model reasoning abilities with fewer thinking token consumption. We train Z1 with a long and short code-related trajectories dataset and equip Z1 with shifted thinking window, a new approach to enable LRM to perform weak reasoning on simple problems and strong reasoning on complex problems.. Trained with long and short trajectories and reasoning with shifted thinking window, Z1 matches state-of-the-art performance

with comparable parameters and demonstrates efficient test-time compute scaling on various reasoning benchmarks. Furthermore, our systematic analysis of key factors for efficient reasoning elicitation provides valuable insights for future research, contributing to the development of more advanced and open-sourced reasoning models.

Limitations

In this paper, we present Z1, a reasoning model that could perform weak reasoning on simple problems and strong reasoning on complex problems. We train Z1 with code-related trajectory dataset. One limitation is that the trajectory dataset we train does not cover much more topics like science and mathematics. Future work could extend our approach by incorporating more diverse trajectory datasets and examining their impact on model performance.

Acknowledgement

Zhaojian Yu, Yinghao Wu and Xiao-Ping Zhang are with the Shenzhen Key Laboratory of Ubiquitous Data Enabling Laboratory, Shenzhen International Graduate School, Tsinghua University, Shenzhen 518055, China and supported by Shenzhen Ubiquitous Data Enabling Key Lab under grant ZDSYS20220527171406015, and by Tsinghua Shenzhen International Graduate School-Shenzhen Pengrui Endowed Professorship Scheme of Shenzhen Pengrui Foundation.

References

- Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, et al. 2025. Process reinforcement through implicit rewards. *arXiv preprint arXiv:2502.01456*.
- Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, et al. 2025. Competitive programming with large reasoning models. *arXiv preprint arXiv:2502.06807*.
- Xidong Feng, Ziyu Wan, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. 2023. Alphazero-like tree-search can guide large language model decoding and training. *arXiv preprint arXiv:2309.17179*.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.
- Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.
- Hugging Face. 2025. Transformers: Trainer class documentation. https://huggingface.co/docs/transformers/main_classes/trainer.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.
- Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Tianjun Zhang Fanjia Yan, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Live-codebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint*.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.
- Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let’s verify step by step. In *The Twelfth International Conference on Learning Representations*.
- Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.
- Haotian Luo, Li Shen, Haiying He, Yibo Wang, Shiwei Liu, Wei Li, Naiqiang Tan, Xiaochun Cao, and Dacheng Tao. 2025. *Adaptthink: Llm can learn when to think*. <https://arxiv.org/abs/2501.12570>.

- Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.
- Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. 2025. Swe-lancer: Can frontier llms earn \$1 million from real-world freelance software engineering? *arXiv preprint arXiv:2502.12115*.
- Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. sl: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*.
- OpenAI. 2024a. [Gpt-4o](#).
- OpenAI. 2024b. [Learning to reason with llms](#).
- Guilherme Penedo, Anton Lozhkov, Hynek Kydlíček, Loubna Ben Allal, Edward Beeching, Agustín Piqueres Lajarín, Quentin Gallouédec, Nathan Habib, Lewis Tunstall, and Leandro von Werra. 2025. Codeforces cots. <https://huggingface.co/datasets/open-r1/codeforces-cots>.
- David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. 2024. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*.
- Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. 2025. Kimi k1.5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*.
- NovaSky Team. 2025a. Sky-t1: Fully open-source reasoning model with o1-preview performance in \$450 budget. <https://novasky-ai.github.io/posts/sky-t1>. Accessed: 2025-01-09.
- OpenThoughts Team. 2025b. Open Thoughts. <https://open-thoughts.ai>.
- Qwen Team. 2024. [Qwq: Reflect deeply on the boundaries of the unknown](#).
- Qwen Team. 2025c. [Qwq-32b: Embracing the power of reinforcement learning](#).
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.
- Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*.
- Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.
- Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.
- Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. 2025. Limo: Less is more for reasoning. *arXiv preprint arXiv:2502.03387*.
- Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2024. Wavecoder: Widespread and versatile enhancement for code large language models by instruction tuning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5140–5153.
- Jiajie Zhang, Nianyi Lin, Lei Hou, Ling Feng, and Juanzi Li. 2025. [O1-pruner: Length-harmonizing fine-tuning for o1-like reasoning pruning](#). *arXiv preprint arXiv: 2505.13417*.

Appendix Contents

A	Related Work	10
B	Dataset Details	10
B.1	Comparison of Datasets	10
B.2	Word frequency details	11
B.3	Ablated Dataset Construction	13
C	Evaluation Details	14
C.1	Baselines	14
C.2	Benchmarks	14
C.3	Prompts for Evaluation	14
C.4	Case Study	14
C.5	Analysis of Reasoning Phrase Patterns	15
C.6	Test-time Values	16
C.7	Comparison with Contemporaneous Work	16
D	Examples	17
D.1	Z1-Code-Reasoning-107K Data Examples	17
D.2	Examples of Shifted Thinking	18
D.3	Failure Mode	19
D.4	Examples on GPQA Diamond	20
D.5	Examples on LiveCodeBench	22
D.6	Examples on MATH500	24

A Related Work

Large Reasoning Models OpenAI o1 and o3 series models (Jaech et al., 2024), which get trained with large-scale RL and learn to reason using chain-of-thought (OpenAI, 2024b), have demonstrated strong reasoning ability in various complex tasks with consistent gains from scaling test-time compute. After the release of o1, Deepseek-R1 (Guo et al., 2025) replicates the performance of o1 through interleaved supervised fine-tuning and reinforcement learning. The R1-Distill series models, fine-tuned on samples distilled from DeepSeek-R1, also achieve test-time scaling through non-RL approaches. In the realm of non-RL data distillation training, many open-source works, such as Sky-T1 (Team, 2025a), s1 (Muennighoff et al., 2025), and LIMO (Ye et al., 2025) have successfully developed competitive reasoning models comparable to o1-preview. For reinforcement learning researches, models like QwQ-32B (Team, 2025c), Kimi-K1.5 (Team et al., 2025), and PRIME-7B (Cui et al., 2025) have matched or even surpassed o1-preview’s performance. Combining short and long trajectory training data and the Shifted Thinking Window, our model, Z1, mitigates overthinking and achieves efficient test-time scaling in LLM reasoning.

Large Language Models for Code The development of large language models (LLMs) for code has undergone significant evolution from pre-trained models such as Codex, StarCoder (Li et al., 2023), DeepSeek-Coder (Guo et al., 2024), and Qwen2.5-Coder Base (Hui et al., 2024) to instruction-tuned variants like WizardCoder (Luo et al., 2023), WaveCoder (Yu et al., 2024), and Qwen2.5Coder-Instruct (Hui et al., 2024). This evolution, with the advancement of test-time scaling, has led to a divergence in model capabilities. On one hand, large reasoning models (LRMs) tailored for competitive programming (e.g., o1-Pro and o1-IOI (El-Kishky et al., 2025)) have emerged, leveraging chain-of-thought (CoT) reasoning to achieve human-level performance in programming contests. On the other hand, LLMs designed for software engineering (SE) tasks, such as Llama3-SWE-RL (Wei et al., 2025), have been developed to address benchmarks like SWE-Bench (Jimenez et al., 2023) and SWE-Lancer (Miserendino et al., 2025). These software engineering-focused LRMs incorporate real-world SE workflows (e.g., Agentless (Xia et al., 2024)) and reinforcement learning, progressively enabling automated project management. In this work, we demonstrate that efficient test-time scaling with code can mitigate the tendency of LRMs to overthink coding problems. Our findings contribute to both competitive programming and software engineering applications, bridging the gap between theoretical advancements and practical deployment.

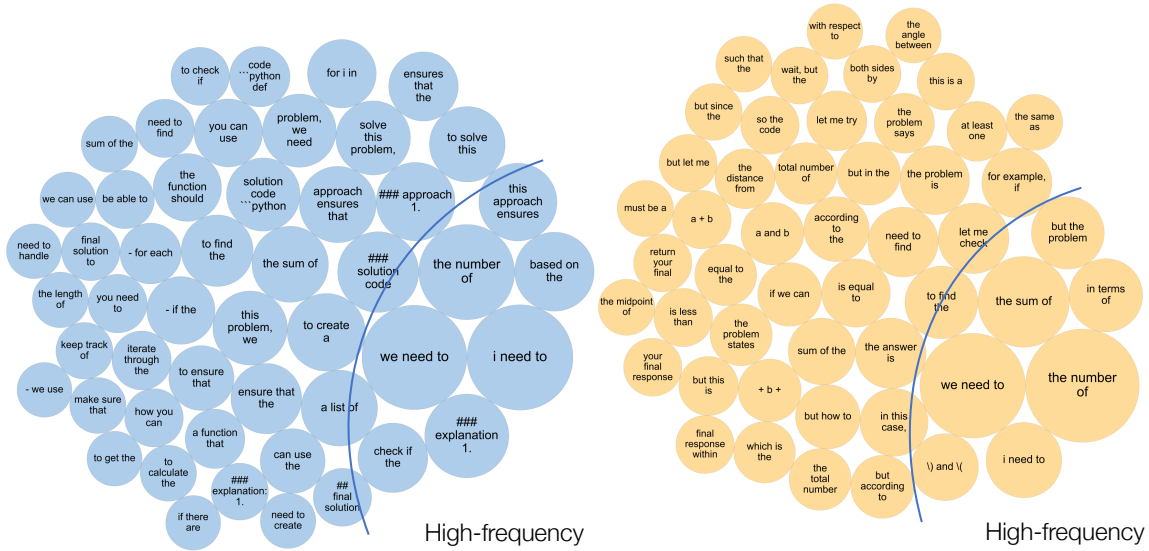
B Dataset Details

B.1 Comparison of Datasets

We analyzed several recent open-source reasoning datasets. Table 4 presents several key characteristics of these datasets, including the number of samples, minimum and maximum token counts, domain, and dataset source. The Z1-Code-Reasoning dataset features shorter reasoning trajectories, which effectively enables our Z1 model to think quickly on simple problems while go into deeper reasoning on more challenging ones. This approach prevents overthinking and makes a significant contribution to achieving efficient test-time scaling.

Table 4: The list of existing open-source reasoning datasets.

Dataset	Samples	Min. tokens	Max. tokens	Domain	Dataset Source
s1 (Muennighoff et al., 2025)	1K	667	7,850	General	Gemini 2.0 Flash Thinking Experimental
s1.1 (Muennighoff et al., 2025)	1K	923	26,685	General	Deepseek R1
CodeForces-CoTs (Penedo et al., 2025)	48K	523	25,156	Competition Code	Deepseek R1
OpenR1-Math-220k ³	22K	4,307	18,611	Math	Deepseek R1
OpenThoughts (Team, 2025b)	114K	299	91,198	General	Deepseek R1
Z1-Code-Reasoning	107K	25	8,169	General Code	QwQ-32B-Preview



(a) Z1-Code-Reasoning-107k word trigram frequency

(b) OpenThoughts-114k word trigram frequency

Figure 4: The comparison between Z1-Code-Reasoning-107K (a) and OpenThoughts-114K (b). We computed the top-50 most frequent trigrams in both datasets. The circle size reflects word frequency.

B.2 Word frequency details

In Section 2.1 and Figure 4, we analyzed the word frequency statistics of our dataset, Z1-Code-Reasoning-107k, in comparison to the previous OpenThoughts-114k dataset. We performed a statistical analysis of word frequencies at the triplet level for both datasets, with the specific top 50 word frequencies detailed in Tables 5 and 6. Due to the inclusion of reasoning data with varying trajectory lengths in our Z1-Code-Reasoning-107k dataset, there is a noticeable reduction in the overall word count. Additionally, as introduced in Section 2.1, our dataset not only contains common logical reasoning connectives but also incorporates a greater proportion of code-related content.

Table 5: Word frequency in Z1-Code-Reasoning-107K

Tri-gram words	Count	Tri-gram words	Count
i need to	72090	to ensure that	18080
we need to	70123	can use the	17828
the number of	53932	- for each	15033
### explanation 1.	37352	code “python def	14631
### solution code	33629	a function that	14000
based on the	31656	iterate through the	13698
### approach 1.	30991	you need to	13485
this approach ensures	30213	be able to	13358
to create a	30150	need to find	13292
approach ensures that	28633	how you can	13280
the sum of	28373	final solution to	13149
a list of	28242	to check if	13119
to solve this	27732	## final solution	13089
solve this problem,	26417	### explanation: 1.	12451

Continued on next page

²<https://huggingface.co/datasets/open-r1/OpenR1-Math-220k>

Tri-gram words	Count	Tri-gram words	Count
this problem, we	26398	keep track of	12198
solution code “python	25448	sum of the	12010
problem, we need	25446	to calculate the	11831
to find the	23880	make sure that	11800
ensure that the	22804	the length of	11610
check if the	21960	we can use	11448
ensures that the	21685	need to create	11366
for i in	20988	to get the	11150
you can use	20674	need to handle	11089
the function should	20524	if there are	11045
- if the	20434	- we use	11042

Table 6: Word frequency in OpenThoughts-114k

Tri-gram words	Count	Tri-gram words	Count
the number of	603860	the total number	106781
we need to	484418	equal to the	106282
the sum of	321257	which is the	104615
i need to	188694	the problem says	98058
in terms of	173819	at least one	97784
to find the	169782	let me try	97748
but the problem	169082	the distance from	97535
let me check	155703	but this is	94306
the answer is	155213	a + b	91141
need to find	151935	is less than	90788
for example, if	151677	so the code	89894
the problem is	146089	return your final	89123
is equal to	142341	your final response	89120
in this case,	139293	final response within	89120
\) and \((	134985	but let me	87315
sum of the	133514	both sides by	86006
according to the	123939	this is a	85909
but how to	119825	wait, but the	85052
if we can	116574	but since the	83803
but in the	113484	must be a	81221
but according to	113061	the midpoint of	81158
+ b +	111663	the same as	80311
a and b	110523	such that the	80211
the problem states	107778	with respect to	78634
total number of	106790	the angle between	77260

B.3 Ablated Dataset Construction

We use random sampling and greedy sampling strategies to obtain three representative subsets to demonstrate the impact of two key factors (**Mean Trajectory Length** and **Training Sample Size**) in the training data on model performance. We provide the definition of the greedy sampling method in Algorithm 1, which is used to generate the Longest and Shortest subsets in Table 2.

Algorithm 1 Greedy Sampling (Longest or Shortest)

```
1: Input: Set of training samples  $S$ , token budget  $B$ , strategy mode  $\in \{\text{"longest"}, \text{"shortest"}\}$ 
2: Output: Subset  $S' \subseteq S$  based on selected strategy
3: Initialize  $S' \leftarrow \emptyset$ , total tokens  $T \leftarrow 0$ 
4: while  $S \neq \emptyset$  and  $T < B$  do
5:   if mode = "longest" then
6:     Find  $s^* \in S$  with maximum token count
7:   else if mode = "shortest" then
8:     Find  $s^* \in S$  with minimum token count
9:   end if
10:  if  $T + \text{token\_count}(s^*) \leq B$  then
11:     $S' \leftarrow S' \cup \{s^*\}$ 
12:     $T \leftarrow T + \text{token\_count}(s^*)$ 
13:  end if
14:   $S \leftarrow S \setminus \{s^*\}$ 
15: end while
16: return  $S'$ 
```

C Evaluation Details

C.1 Baselines

We benchmark Z1 against a series of top-tier models: OpenAI o1-series models (Jaech et al., 2024): o1-mini and o1-preview, representing close-source test-time scaling models; Deepseek-R1 series (Guo et al., 2025): Deepseek-R1, R1-Distill-Qwen (32B and 7B) and Qwen’s QwQ-32B-Preview (Team, 2024), open-weight reasoning models; Sky-T1-32B-Preview (Team, 2025a), s1.1-7B (Muennighoff et al., 2025), OpenThinker-7B (Team, 2025b), open models with open reasoning data; Deepseek-V3 (Liu et al., 2024), GPT-4o (OpenAI, 2024a), Qwen2.5-Coder-7B-Instruct (Hui et al., 2024), four representative non-reasoning models. Our model, Z1, is fully open including weights, reasoning data, and code. We evaluate Z1 using shifted thinking window with a maximum thinking tokens of 4,096. For all baseline models, we use the reported results whenever available. If no reported scores are provided, we evaluate the model using budget forcing with the configuration provided.

C.2 Benchmarks

We select three representative reasoning benchmarks covering different topics: **LiveCodeBench** (Jain et al., 2024) continuously collects new problems over time from contests across three competition platforms, including LeetCode, AtCoder, and CodeForces. Unless otherwise specified, we benchmark LLMs on such competition-level programming tasks with the latest full set (880 problems until Feb, 2025) of LiveCodeBench v5. **GPQA Diamond** (Rein et al., 2024) consists of 198 PhD-level science questions from Biology, Chemistry and Physics. Experts with PhDs in the corresponding domains only achieved 69.7% on GPQA Diamond, which shows its inherent difficulty and challenges. **MATH500** (Hendrycks et al., 2021) is a benchmark of competition math problems of varying difficulty. Following previous work (Guo et al., 2025), we evaluate our model on the same subset selected by OpenAI (Lightman et al., 2023). Alongside the three common reasoning benchmarks, we also incorporate a non-reasoning benchmark: **BigCodeBench** is a benchmark mainly focusing on more challenging and practical code generation with complex instructions and diverse function calls. In this section, we adopt the BigCodeBench-Hard-Instruct (148 problems included) subset to evaluate LRM on short trajectory thinking. For all benchmarks, we generate a sample for each question with a temperature of 0 (greedy) to measure accuracy. Through these benchmarks, we can evaluate the reasoning ability of LLMs from different perspectives.

C.3 Prompts for Evaluation

We use prompts from Qwen2.5-Coder Github for LiveCodeBench⁴ and BigCodeBench⁵ and the following prompts for MATH500 and GPQA Diamond:

Prompts

```
<lim_start>system
Please reason step by step, and put your final answer within \boxed{ }.
<lim_end>
<lim_start>user
{problem}
<lim_end>
<lim_start>assistant
```

C.4 Case Study

Figure 5 illustrates a problem-solving example involving three models: Qwen2.5-Coder-7B, Z1-7B, and R1-Distill-Qwen-7B. In this instance, we present a simple problem, for which the Qwen2.5-Coder-7B-Instruct model delivers a correct answer accurately. By comparison, R1-Distill-Qwen-7B adopts a context

⁴https://github.com/QwenLM/Qwen2.5-Coder/blob/main/qwencoder-eval/instruct/livecode_bench/lcb_runner/prompts/code_generation.py

⁵<https://github.com/QwenLM/Qwen2.5-Coder/blob/main/qwencoder-eval/instruct/BigCodeBench/model.py>

Write a Python script to calculate the number of letter ‘a’ and ‘r’ in a string.

Input: s = ‘strawberry’

Output: ‘a’: 1, ‘r’: 3

[Qwen2.5-Coder-7B-Ins]

Here is a Python script that can solve the problem:

```
def count_letters(s):
    a_count = s.count('a')
    r_count = s.count('r')
    return {'a': a_count,
            'r': r_count}

s = 'strawberry'
print(count_letters(s))
```

In this script, we define a function count_letters...
[120+ tokens omitted]

[Z1-7B]

To solve this problem, we need...

[50+ tokens omitted]

1. Approach

[94 tokens omitted]

2. Solution Code

```
def count_letters(s):
    count_a = 0
    count_r = 0
    for char in s:
        if char == 'a':
            count_a += 1
        elif char == 'r':
            count_r += 1
    print("'a' :", count_a)
    print("'r' :", count_r)
```

3.Explanation

[104 tokens omitted]

This approach ensures that we...

[28 tokens omitted]

[R1-Distill-Qwen-7B]

<think>

Okay, I need to write a Python script that counts the number of ‘a’s and ‘r’s...

[1787 tokens omitted]

</think>

To solve this problem...

[45 tokens omitted]

1. Approach

[92 tokens omitted]

2. Solution Code

```
s = input().strip()
a_count = s.count('a')
r_count = s.count('r')
print(f'a: {a_count}')
print(f'r: {r_count}')
```

3.Explanation

[100+ tokens omitted]

This approach ensures that the solution ...

[22 tokens omitted]

Figure 5: **Example model outputs.** We present a question that Qwen2.5-Coder-7B-Instruct (left) can correctly answer without requiring deep reasoning. The response from Z1 (middle) avoids excessive additional reasoning. In contrast, R1-Distilled-Qwen-7B generates 1,784 tokens of reasoning after the delimiter.

split to enforce thinking, requiring extensive deliberation that consumes 1,784 tokens before arriving at a solution. This protracted process underscores its inefficiency in optimizing thinking time for simpler tasks. Z1-7B employs the Shifted Thinking Window to effectively bypass unnecessary overthinking within the given context, demonstrating its advantage for balancing accuracy and efficiency in problem-solving.

C.5 Analysis of Reasoning Phrase Patterns

To investigate the cross-domain reasoning capabilities of our model, we analyzed the tri-gram word frequency statistics in the reasoning traces from the training dataset (code-focused) and the model’s generated outputs on code (LiveCodeBench) and math (MATH 500) benchmarks, as shown in Table 7.

Training Data (Code)	Test on Code (LiveCodeBench)	Test on Math (MATH 500)	Intersection of Code and Math
i need to (10.48%) we need to (10.32%) the number of (7.38%) ““### explanation (6.55%) ### explanation 1. (5.53%) ### solution code (4.98%) ### approach 1. (4.59%) approach ensures that (4.24%) to create a (4.24%) based on the (4.17%) solve this problem, (3.91%) solution code “python (3.77%) problem, we need (3.77%) the sum of (3.68%) a list of (3.42%) to find the (3.29%) ensure that the (3.22%) a list of (3.42%) to find the (3.29%) ensure that the (3.22%)	the number of (15.67%) we need to (11.63%) - for each (5.32%) the sum of (4.96%) to the number (4.21%) to solve this (4.20%) ### solution code (4.18%) ### approach 1. (4.14%) this problem, we (4.14%) problem, we need (4.14%) solution code “python (4.10%) ### explanation 1. (4.06%) equal to the (3.57%) approach ensures that (3.37%) this approach ensures (3.37%) is equal to (3.31%) number of unique (3.26%) this approach ensures (3.37%) is equal to (3.31%) number of unique (3.26%)	we need to (18.67%) to find the (8.84%) the number of (6.43%) need to find (6.24%)) and ((5.52%) (x = (4.80%) = 0] (4.77%)] ### step (4.26%) the sum of (4.26%) step by step. (3.91%) - 6 = (3.36%) consider that the (3.04%) problem, we need (2.97%) i need to (2.94%) sum of the (2.92%) (n = (2.87%)] therefore, the (2.85%) sum of the (2.92%) (n = (2.87%)] therefore, the (2.85%)	the number of (17.52%) we need to (17.01%) the sum of (6.18%) problem, we need (4.99%)

Table 7: Comparison of tri-gram word frequency in reasoning phrase on different datasets

A significant overlap in high-frequency tri-grams, such as "we need to" and "the number of," was observed between the code and math outputs. This overlap reflects the model’s use of structured, domain-agnostic reasoning strategies. For example, "we need to" often introduces a logical breakdown of steps in code, while in math, it signals the start of a deductive process. Similarly, "the number of" is used in code for quantifying variables or iterations and in math for defining quantities or relationships.

The consistent tri-gram distribution between training data and test outputs indicates that the model has internalized general reasoning skills—such as task decomposition, logical sequencing, and deductive thinking—from code training and effectively applies them to mathematical problem-solving. This analysis underscores the model’s ability to generalize reasoning skills across domains.

C.6 Test-time Values

In Section 3.3, we presented the results of Z1-7B and R1-Distill-Qwen-7B on three reasoning benchmarks, highlighting Z1-7B’s more efficient test-time scaling capability. Table 8 provides a more detailed account of the experimental results. By varying the maximum number of thinking tokens, we assessed the test-time scaling abilities of both models across different lengths of thinking trajectories. Z1-7B demonstrated strong performance even under tighter constraints on thinking tokens, showcasing its efficient test-time scaling capability. In contrast, R1-Distill-Qwen-7B required more thinking tokens to achieve a comparable level of performance.

Model	GPQA		LiveCodeBench		MATH 500	
	Avg. think tokens	Pass@1	Avg. think tokens	Pass@1	Avg. think tokens	Pass@1
R1-Distill-Qwen-7B	512	18.2	512	13.9	512	59.8
	-	-	-	-	719	62.4
	1011	19.2	1023	20.7	1016	72.0
	-	-	-	-	1337	76.0
	1971	26.3	1978	28.8	1760	83.4
	3607	35.4	3580	40.0	2541	87.6
Z1-7B	256	25.3	256	30.3	255	37.2
	510	28.8	502	31.9	471	55.8
	946	33.3	732	35.0	707	71.2
	1548	41.9	866	35.3	823	74.2
	2470	47.5	-	-	1185	76.4

Table 8: Evaluation details on three reasoning benchmarks

C.7 Comparison with Contemporaneous Work

We also compare Z1 with some great contemporaneous work, like AdaptThink (Zhang et al., 2025) and Marco-o1-7B-Pruned (Luo et al., 2025). Z1 matches state-of-the-art performance with the same parameters and demonstrates efficient test-time compute scaling on various reasoning benchmarks.

Model	Base Model	MATH 500	GPQA Diamond	LiveCode Bench	AVG
Z1-7B (ours)	Qwen2.5-Coder	76.4% (1185)	47.5% (2470)	35.3% (866)	53.1% (1507)
AdaptThink-7B	Qwen2.5-Math	92.0% (1875)	35.4% (5382)	45.0% (8906)	57.5% (5538)
Marco-o1-7B-Pruned	Qwen2-7B	72.8% (654)	43.4% (262)	26.3% (775)	47.5% (567)

Table 9: Comparison with Contemporaneous Work about the efficiency of reasoning.

D Examples

D.1 Z1-Code-Reasoning-107K Data Examples

Case #1: A simple problem with short trajectory answer

Question:

Convert the following query into a T-SQL statement.

```
SELECT Employee_ID, Name
FROM Employees
WHERE Salary > 50000
```

Answer:

Your query is already written in T-SQL. The same statement would go as:

```
```sql
SELECT Employee_ID, Name
FROM Employees
WHERE Salary > 50000
```
```

Case #2: A complex problem with long trajectory answer

Question:

In my machine learning course we have covered the key ideas behind principal component analysis. To round this part of the course off, we have learned to interpret the results of PCA, specifically using a biplot. Have I missed something, or is PCA mostly something that is used in conjunction with other ML methods to obtain a result? For example, in a biplot I can tell which observations are similar and how much each variable is contributing overall to a specific component, but how is this useful outside of employing something like clustering after-the-fact? From my point of view, it feels like PCA is the 'first-step' towards gaining some sort of insight but it doesn't tell you much that is useful on its own.

Answer:

Principal Component Analysis (PCA) is indeed a fundamental technique in data analysis and machine learning, and your intuition about it being a first step is correct. However, PCA can be quite useful on its own in several ways beyond just being a preprocessing step for other methods like clustering. Let's explore some of these applications in more detail.

1. **Dimensionality Reduction for Visualization**

One of the primary uses of PCA is to reduce the dimensionality of data for visualization purposes. By projecting high-dimensional data onto a lower-dimensional space (typically 2D or 3D), PCA allows us to visualize complex datasets and identify patterns, clusters, or outliers that might not be apparent in higher dimensions. The biplot, which you mentioned, is a useful tool in this context as it shows both the observations and the variables in the same plot, providing insights into how variables relate to each other and to the observations.

2. **Feature Extraction**

PCA can be used for feature extraction, where the principal components themselves can serve as new features for machine learning models. These components are linear combinations of the original features and are ordered by the amount of variance they explain. Using a subset of these components can help reduce noise, improve model performance by eliminating redundant features, and speed up training times by reducing the number of input features.

3. **Noise Reduction**

Since PCA captures the directions of maximum variance in the data, it can be used for noise reduction by projecting the data onto a subspace that retains most of the variance while leaving out the components that capture noise. This is particularly useful when dealing with noisy datasets where the signal is buried under random variations.

4. **Data Compression**

In applications where storage or transmission of data is a concern, PCA can be used for data compression. By retaining only the top principal components, we can represent the data with fewer dimensions while preserving most of the information. This is similar to how image compression algorithms work, where less important details are discarded to reduce file size.

5. **Exploratory Data Analysis (EDA)**

PCA is a valuable tool in EDA for understanding the structure of the data. It helps in identifying correlations between variables, detecting outliers, and understanding the underlying patterns in the data. The loadings of the principal components can indicate which original features are most influential in each component, providing insights into the data's intrinsic structure.

6. **Preprocessing for Machine Learning Models**

While PCA is often used as a preprocessing step before applying other machine learning algorithms, it can significantly improve the performance of these models. By reducing dimensionality, PCA can help alleviate the curse of dimensionality, improve model training speed, and sometimes even enhance model accuracy by removing noise and redundant information.

7. **Anomaly Detection**

PCA can be used for anomaly detection by identifying observations that lie far from the origin in the principal component space. Since PCA models the directions of maximum variance, points that deviate significantly from this model can be considered anomalies or outliers.

8. **Signal Processing and Image Analysis**

In fields like signal processing and image analysis, PCA is used for tasks such as denoising, compression, and feature extraction. For example, in image processing, PCA can be used to reduce the dimensionality of image data, which is useful for tasks like face recognition.

9. **Genetic Data Analysis**

In genomics, PCA is widely used to visualize and analyze genetic data, helping researchers understand population structures, identify genetic relationships, and detect outliers or batch effects in the data.

10. **Finance and Economics**

In finance, PCA is used for portfolio optimization, risk management, and understanding the underlying factors that drive asset returns. In economics, it can be used to identify key economic indicators or to reduce the dimensionality of macroeconomic data for forecasting.

Conclusion

While PCA is indeed often used as a first step in data analysis, it has numerous standalone applications that make it a versatile and powerful tool. Its ability to reduce dimensionality, extract meaningful features, visualize complex data, and handle noise makes it invaluable in a wide range of fields and applications. Understanding PCA thoroughly will not only help you in your machine learning coursework but also in applying data analysis techniques effectively in real-world scenarios.

D.2 Examples of Shifted Thinking

Simple Problem

How many r in 'strawberry' ?



Quickly answer:

In the word "strawberry", there are three 'r's.

Complex Problem

How many r in 'strawberryandraspberry' ?



Long trajectory answer:

To find out how many 'r's are in the word "strawberryandraspberry," I'll count each occurrence of the letter 'r' in the given string.

Let's break it down: 1. First, I'll write down the string: "strawberryandraspberry" 2. Now, I'll go through each character in the string and count how many times 'r' appears.

Starting from the beginning:

- s: not 'r'- t: not 'r'- r: this is 'r' (count: 1)- a: not 'r'- w: not 'r'- b: not 'r'- e: not 'r'- r: this is 'r' (count: 2)- r: this is 'r' (count: 3)- y: not 'r'- a: not 'r'- n: not 'r'- d: not 'r'- a: not 'r'- n: not 'r'- d: not 'r'- r: this is 'r' (count: 4)- a: not 'r'- s: not 'r'- p: not 'r'- b: not 'r'- e: not 'r'- r: this is 'r' (count: 5)- r: this is 'r' (count: 6)- y: not 'r'

So, after going through the entire string, I've counted 6 'r's.

Therefore, there are 6 'r's in "strawberryandraspberry."

Alternatively, I could use a programming approach to count the occurrences.

For example, in Python, I could use the count() method:

```
word = "strawberryandraspberry"
count = word.count('r')
print(count)
```

This would also give me the count of 'r's, which is 6.

So, the final answer is 6.

Final Solution

To determine how many 'r's are in the word "strawberryandraspberry," we can count each occurrence of the letter 'r' in the string.

Approach1. **Manual Counting:**

- Write down the string: "strawberryandraspberry"
- Go through each character and count how many times 'r' appears.

2. **Programming Approach:**

- Use a programming language to count the occurrences of 'r' in the string.

Solution Code

```
```python
word = "strawberryandraspberry"
count = word.count('r')
print(count)
```
```

Explanation

- **Manual Counting:** - The string is "strawberryandraspberry"
- Counting 'r's: s(0), t(0), r(1), a(0), w(0), b(0), e(0), r(2), r(3), y(0), a(0), n(0), d(0), a(0), n(0), d(0), r(4), a(0), s(0), p(0), b(0), e(0), r(5), r(6), y(0)
- Total 'r's: 6

- **Programming Approach:

- ** - Using the 'count()' method in Python to count the occurrences of 'r' in the string.
- The result is 6.

Thus, there are 6 'r's in "strawberryandraspberry."

Figure 6: Examples of answer of Z1-7B under different questions.

D.3 Failure Mode

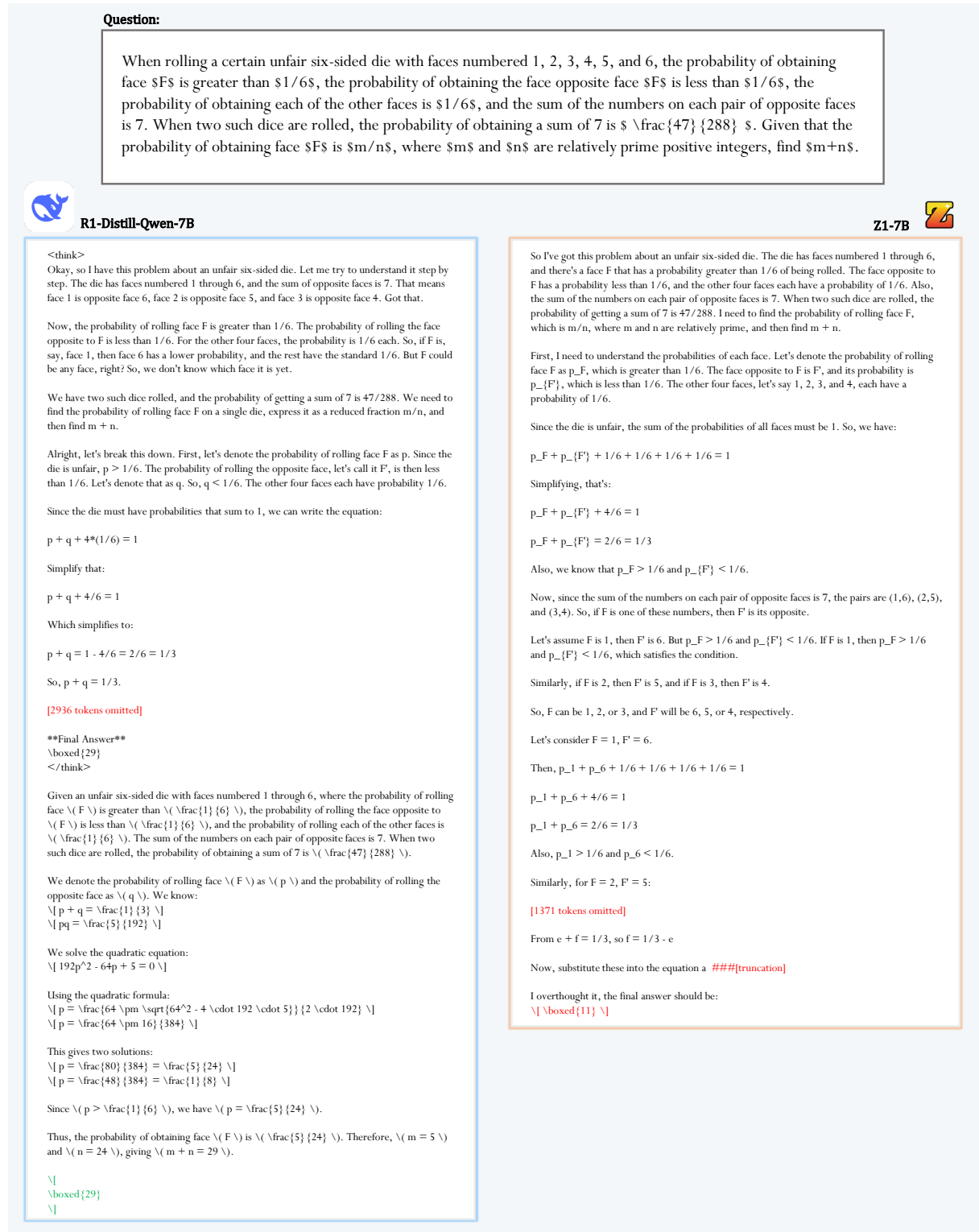


Figure 7: **The failure case of Z1-7B.** R1-distill utilized 3803 tokens to complete the reasoning and arrive at the correct answer "29". In contrast, Z1, with "max_thinking_token" limit of 2048, was truncated during the reasoning stage, resulting in an incorrect answer "11". Therefore, even though Z1 can adjust reasoning length based on problem difficulty to some extent, it is preferable to allocate a higher thinking token budget in complex scenarios to improve performance.

Observations of an absorption line, that corresponds to the change in energy level of 5.9 micro electron volts (5.9×10^{-6} eV), are being carried out. What is most likely to be observed this way?

- A. Cold atomic interstellar medium.
- B. Warm atomic interstellar medium.
- C. Cold molecular interstellar medium.
- D. Warm molecular interstellar medium.

Please reason step-by-step and put your choice letter without any other text with \boxed{} in the end.

Okay, so I have this question about absorption lines corresponding to a specific energy change, and I need to figure out which medium is most likely responsible. Let me try to break this down step by step.

I also recall that the energy of a photon is related to its wavelength by the equation $E = hc/\lambda$, where h is Planck's constant, c is the speed of light, and λ is the wavelength. But I'm not sure if I need to calculate the wavelength here or if there's a more straightforward approach.

In atomic transitions, electrons move between energy levels in atoms, which usually have smaller energy differences compared to molecular transitions. Molecules have more complex energy structures because they have both electronic and vibrational energy levels. So, the energy required for a transition in a molecule might be higher or lower than in an atom, depending on the specific transition.

In a cold medium, the electrons don't have much thermal energy, so they can only transition between energy levels that are close in energy. If the medium is warm, then the electrons have more thermal energy, so they can transition to higher energy levels more easily.

So, if it's a cold medium, the electrons can only transition between energy levels that are close, which would correspond to lower energy changes. If it's a warm medium, maybe higher energy transitions are possible, but I'm not sure if that's the case here.

So, if the absorption line corresponds to the 21 cm transition, that's a hydrogen line, and it's observed in cold regions of the interstellar medium. So, the medium must be cold.

But wait, could it be molecular? Molecular lines usually have different energy transitions. For example, rotational transitions in molecules have higher energy differences compared to atomic transitions, but I'm not sure. The 21 cm line is definitely an atomic transition.

The absorption line corresponds to a 21 cm transition in hydrogen atoms, which is a very small energy change of 5.9×10^{-6} eV. This energy is characteristic of the 21 cm line, typically observed in cold interstellar media. Since hydrogen is an atomic element, the medium is cold and atomic.

 $\boxed{A}$

Now, let's consider each option:

B. Warm atomic interstellar medium: A warm atomic medium has higher temperatures, which can cause atoms to be in higher energy states. This could lead to transitions that are not as sensitive to the specific energy difference provided. However, the energy difference is still relatively small, so it's possible to observe absorption lines in this medium as well.

D. Warm molecular interstellar medium: Similar to the warm atomic medium, a warm molecular medium has higher temperatures, which can cause molecules to be in higher energy states. This could lead to transitions that are not as sensitive to the specific energy difference provided. However, the energy difference is still relatively small, so it's possible to observe absorption lines in this medium as well.

Given that the energy difference is 5.9×10^{-6} eV, which is very small, it is most likely to correspond to atomic transitions in a cold medium, where such fine energy differences are more common. Molecular transitions typically involve larger energy differences, making option C less likely.

Therefore, the most probable observation is in a cold atomic interstellar medium.

 $\boxed{A}$

2707



Figure 9: Examples of the comparison between Z1-7B and R1-Distill-Qwen-7B on GPQA.

Question:

Order Something Else
Takahashi wants a beverage called AtCoder Drink in a restaurant.
It can be ordered at a regular price of P yen.
He also has a discount coupon that allows him to order it at a lower price of Q yen.
However, he must additionally order one of the restaurant's N dishes to use that coupon.
For each $i = 1, 2, \dots, N$, the price of the i -th dish is D_i yen.
Print the minimum total amount of money that he must pay to get the drink.
The input is given from Standard Input in the following format:

[Constraints and Examples omitted ...]

The total payment will be minimized by not using the coupon and paying the regular price of 100 yen.



R1-Distill-Qwen-7B

Z1-7B



The minimum between P and $(Q + \min(D_i))$.

Yes, because using the coupon with the cheapest dish gives the lowest possible coupon cost. So that's the way to go.

So the plan is:

- Read N , P , Q .

- Read the list of D_i .

- Find the minimum D_i .

- Compute the coupon cost as $Q + \min D_i$.

- The answer is the minimum between P and coupon_cost .

Wait, but what if Q is less than P ? Then, using the coupon might be better, but only if the sum is less than P .

Wait, no. Because if Q is less than P , but adding the $\min D_i$ makes it more than P , then it's better not to use the coupon.

So the correct approach is to compute the minimum between P and $(Q + \min(D_i))$.

Yes.

Let me test this logic against the sample inputs.

Sample Input 1:

[637 tokens omitted]

So the code should handle all cases correctly.

I think that's the solution.
</think>

To solve this problem, we need to determine the minimum amount of money Takahashi must pay to get a drink and a dish, considering he can use a discount coupon. The solution involves comparing the cost of using the coupon with the cost of not using it.

Approach

1. **Understand the Problem**: Takahashi can buy a drink at a regular price of P yen or use a coupon to pay Q yen for the drink, but he must also order one dish. The goal is to find the minimum total cost.
2. **Calculate Costs**:
 - The cost without using the coupon is simply P yen.
 - The cost using the coupon is Q yen plus the price of the cheapest dish.
3. **Comparison**: Compare the cost of using the coupon ($Q + \text{minimum dish price}$) with the regular price P to determine the cheaper option.

Solution Code

```
'''python
n, p, q = map(int, input().split())
dishes = list(map(int, input().split()))
min_dish = min(dishes)
coupon_cost = q + min_dish
print(min(p, coupon_cost))
'''
```

Explanation

1. **Reading Input**: The input consists of the number of dishes, the regular price of the drink, the coupon price, and the prices of the dishes.
2. **Finding Minimum Dish Price**: Calculate the minimum price among the list of dish prices.
3. **Calculate Coupon Cost**: Compute the total cost if the coupon is used, which is the coupon price plus the cheapest dish price.
4. **Determine Minimum Cost**: Compare the total cost using the coupon with the regular price of the drink and print the smaller value.

This approach ensures that we efficiently find the minimum cost by leveraging the minimum dish price to minimize the coupon cost, thus providing an optimal solution.

To solve this problem, we need to determine the minimum total amount of money Takahashi must pay to get the AtCoder Drink, considering both the regular price and the discounted price with a coupon.

Approach

1. **Understand the Problem**:
 - The drink costs P yen regularly.
 - With a coupon, it costs Q yen, but he must order one of the N dishes.
 - Each dish has a price D_i .
2. **Identify the Options**:
 - Option 1: Pay the regular price P yen.
 - Option 2: Use the coupon to pay Q yen and choose one dish.
3. **Minimize the Cost**:
 - To minimize the total cost, choose the option with the lower total payment.
 - The total cost for the coupon option is Q plus the price of the cheapest dish.
4. **Edge Cases**:
 - If Q is greater than or equal to P , it's always better to pay the regular price.
 - If Q is less than P , compare Q plus the minimum dish price with P .

Solution Code

```
'''python
# Read input values
import sys
input = sys.stdin.read
data = input().split()

N = int(data[0])
P = int(data[1])
Q = int(data[2])
D = list(map(int, data[3:]))

# Find the minimum dish price
min_d = min(D)

# Calculate the minimum total cost
if Q + min_d < P:
    min_cost = Q + min_d
else:
    min_cost = P

# Print the result
print(min_cost)
'''
```

Explanation

1. **Input Handling**:
 - Read the input values for N , P , Q , and the list of dish prices D .
2. **Find the Minimum Dish Price**:
 - Use the 'min' function to find the cheapest dish.
3. **Calculate the Minimum Total Cost**:
 - Compare the total cost of using the coupon ($Q + \min D$) with the regular price P .
 - Choose the smaller of the two.
4. **Output the Result**:
 - Print the minimum total cost.

This approach ensures that we always choose the option with the lowest possible total cost, considering both the regular and coupon options.

Figure 11: Examples of the comparison between Z1-7B and R1-Distill-Qwen-7B on LiveCodeBench.

D.6 Examples on MATH500

Question:

If $2^8 = 4^x$, what is the value of x ?



R1-Distill-Qwen-7B

Z1-7B



<think>

Okay, so I have this equation here: $2^8 = 4^x$. Hmm, I need to find the value of x . Let me think about how to approach this. I remember that when dealing with exponential equations, it's often helpful to express both sides with the same base. That way, I can set the exponents equal to each other and solve for the unknown.

First, let me write down the equation again to make sure I have it right: $2^8 = 4^x$. So, on the left side, the base is 2, and on the right side, the base is 4. I know that 4 is a power of 2, specifically $4 = 2^2$. So, maybe I can rewrite 4 as 2^2 . Let me try that.

If I substitute 4 with 2^2 , the equation becomes $2^8 = (2^2)^x$. Now, I remember that when you have an exponent raised to another exponent, you can multiply them. So, $(2^2)^x$ is the same as $2^{(2 \cdot x)}$. Let me write that down: $2^8 = 2^{(2x)}$.

Now that both sides have the same base, which is 2, I can set the exponents equal to each other. That means 8 should equal $2x$. So, $8 = 2x$. To solve for x , I just need to divide both sides by 2. Let me do that: 8 divided by 2 is 4 , so x equals 4 .

Wait, let me double-check to make sure I didn't make a mistake. If x is 4 , then 4^x is 4^4 . Let me calculate both sides of the original equation to verify. On the left side, 2^8 is 256 . On the right side, 4^4 is also 256 because $4 \cdot 4$ is 16 , $16 \cdot 4$ is 64 , and $64 \cdot 4$ is 256 . Yep, both sides are equal, so x must indeed be 4 .

I think that's it. I converted the base 4 into base 2 , which allowed me to equate the exponents and solve for x . It all checks out. I don't see any mistakes in my reasoning, so I'm confident that x is 4 .

****Final Answer****

The value of x is $\boxed{4}$.

</think>

Given the equation $2^8 = 4^x$, we need to find the value of x .

First, we express both sides with the same base. We know that 4 can be written as 2^2 . Substituting this into the equation, we get:

$$2^8 = (2^2)^x$$

Using the property of exponents $(a^m)^n = a^{mn}$, we can rewrite the right side:

$$2^8 = 2^{2x}$$

Since the bases are the same, we can set the exponents equal to each other:

$$8 = 2x$$

Solving for x , we divide both sides by 2:

$$x = \frac{8}{2} = 4$$

Thus, the value of x is $\boxed{4}$.

To solve the equation $2^8 = 4^x$ for x , we can follow these steps:

- Express both sides with the same base:**
 - Notice that both 2 and 4 can be expressed as powers of 2.
 - We know that $4 = 2^2$, so $4^x = (2^2)^x$.
- Simplify the right-hand side:**
 - Using the property of exponents $(a^m)^n = a^{m \cdot n}$, we can rewrite $(2^2)^x$ as 2^{2x} .
 - Therefore, the equation becomes $2^8 = 2^{2x}$.
- Set the exponents equal to each other:**
 - Since the bases are the same, we can set the exponents equal to each other.
 - This gives us $8 = 2x$.
- Solve for x :**
 - Divide both sides of the equation by 2 to isolate x .
 - $x = \frac{8}{2} = 4$.
- Verify the solution:**
 - Substitute $x = 4$ back into the original equation to check.
 - Left side: $2^8 = 256$
 - Right side: $4^4 = (2^2)^4 = 2^8 = 256$
 - Both sides are equal, confirming that $x = 4$ is correct.

Therefore, the value of x is $\boxed{4}$.

Figure 12: Examples of the comparison between Z1-7B and R1-Distill-Qwen-7B on MATH500.

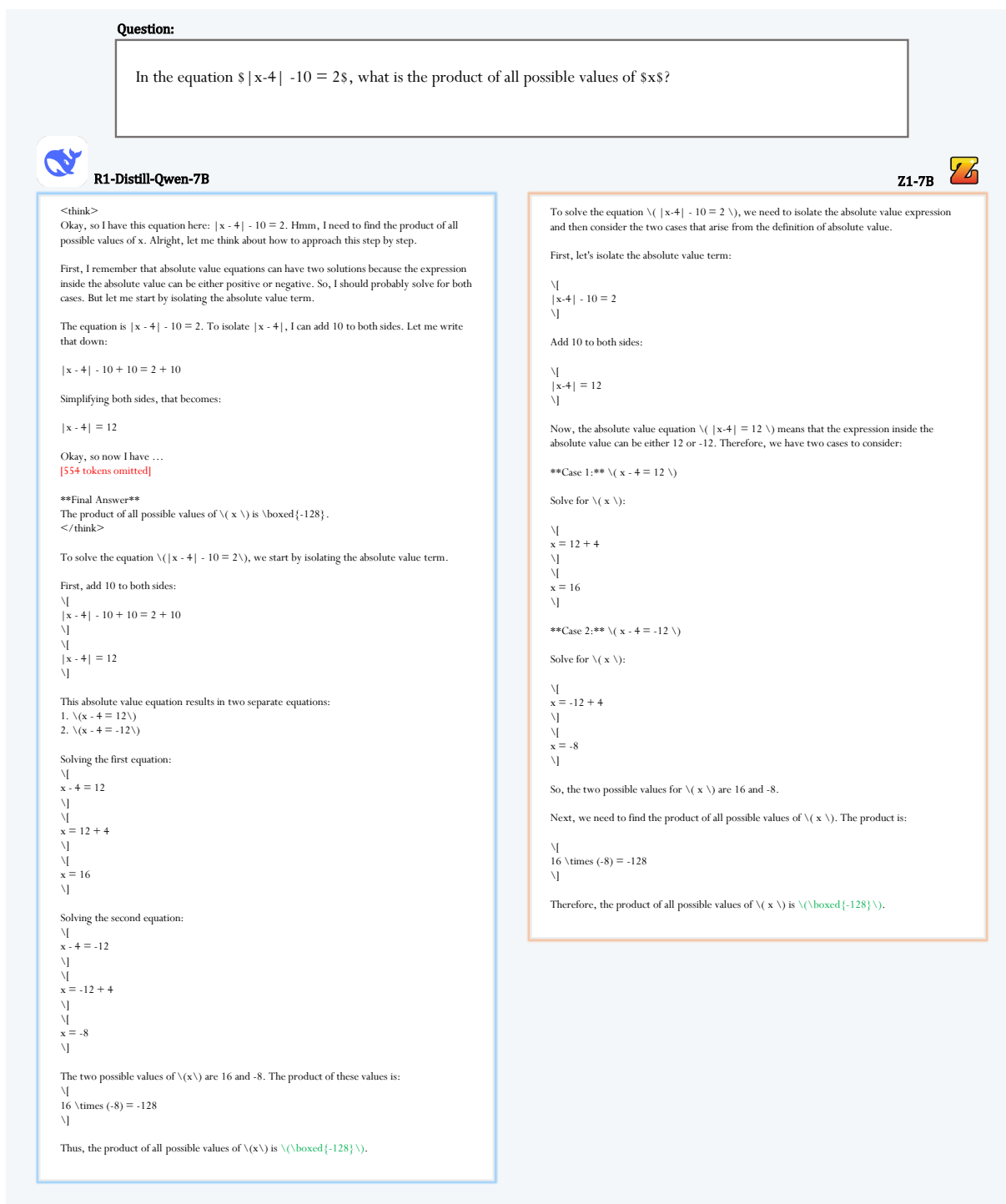


Figure 13: Examples of the comparison between Z1-7B and R1-Distill-Qwen-7B on MATH500.