CAN LARGE LANGUAGE MODELS REINVENT FOUNDATIONAL ALGORITHMS?

Anonymous authors
Paper under double-blind review

ABSTRACT

LLMs have shown strong potential to advance scientific discovery. Whether they possess the capacity for foundational innovation, however, remains an open question. In this work, we focus on a prerequisite for foundational innovation: *can LLMs reinvent foundational algorithms in computer science?* We use LLM unlearning methods to suppress direct recall of the target algorithm and let the model reason with the remaining knowledge to recover it. Although unlearning does not guarantee full knowledge removal, LLMs fail to recover nearly half of the target algorithms. Notably, even suppressing the mention of the algorithm's name during decoding without unlearning makes the models' recovery rate drop dramatically (28–57%), suggesting their overreliance on memorized knowledge. We observe that recoverable algorithms tend to be simple in structure or core ideas, whereas the others are less straightforward. We also introduce a generative verifier that sustains models' reasoning strength, helping to avoid the "thought collapse" phenomenon. Taken together, by treating the unlearned model's recovery rate as an approximate upper bound, our empirical results suggest that current LLM systems still have limited ability to make foundational algorithm innovation.

1 INTRODUCTION

The emergence of large language models (LLMs) with enhanced reasoning and agentic capabilities (OpenAI et al., 2024; Guo et al., 2025) has expanded the potential of Artificial Intelligence (AI) in complex problem-solving. Systems such as FunSearch (Romera-Paredes et al., 2023) and AlphaEvolve (Novikov et al., 2025) have shown that LLM-based systems can discover algorithms that outperform the best known human-designed methods on selected tasks. Despite these advances, whether LLMs can produce foundational scientific discoveries, rather than incremental improvements on existing problems, remains an open question (Feng et al., 2026b).

In this work, we focus on foundational algorithms in computer science, such as Dijkstra's and Euclid's algorithms, as these algorithms have proven to be groundbreaking contributions that form the basis of the field. A central question is whether *LLMs could invent these algorithms without prior exposure*. Since all such algorithms are included in the training data of existing pre-trained models, a natural approach is to remove these algorithms from the pre-training data and train a "clean" model, but the computational cost would be prohibitive.

Instead of complete retraining, we leverage LLM unlearning (Cao & Yang, 2015) to suppress direct recall of the target algorithm, then evaluate whether the unlearned model can recover that algorithm when faced with an application task with efficiency requirements tailored to the target algorithm or its variants. During the recovery phase, the model reasons and interacts with a Python interpreter and a generative verifier (Zhang et al., 2025b) which returns diagnostic feedback to guide revision upon failure. To make the unlearning effective, we adopt an on-policy unlearning method based on Group Relative Policy Optimization (GRPO) (Shao et al., 2024), integrated with a cold start stage.

We evaluate strong open-weight models across 20 target algorithms. To quantify how external hints affect recovery, we experiment with 3 hint levels: no hint, high-level hints, and step-by-step hints. Our major findings are:

1. **LLMs show limited ability to independently recover foundational algorithms after targeted knowledge suppression** (§5.1; §5.2). Although unlearning does not guarantee complete knowl-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

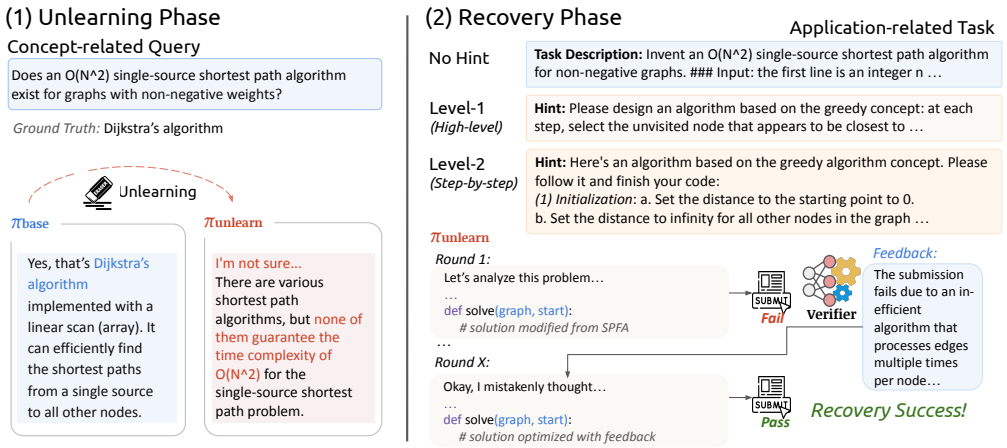

Figure 1: **Our evaluation pipeline.** (1) In the unlearning phase, a pretrained model π_{base} is transformed into π_{unlearn}, which, under our tests, no longer recalls Dijkstra's algorithm for concept-level queries. (2) In the recovery phase, the model is prompted with the task at three hint levels: No hint, level 1 (high-level hint), and level 2 (step-by-step hint). At each attempt, π_{unlearn} reasons and submits its solution for testing. If the submission fails, a generative verifier instantiated from π_{unlearn} itself returns diagnostic feedback, helping the model locate the error and revise its approach.

edge removal, in the no-hint setting, 9 of the 20 target algorithms show no successful recovery across the three evaluated models. Recoverable algorithms also tend to have simpler structures or core ideas. Notably, even without unlearning, merely suppressing algorithm-name-related tokens during decoding causes substantial drops in recovery rate, suggesting a strong dependence on memorized knowledge and familiar algorithmic cues.

2. **Algorithmic hints substantially narrow the recovery gap between unlearned and original models** (§5.1). High-level hints already enable recovery of additional algorithms, while step-by-step hints even enable some models to exceed their original counterparts. This indicates that unlearned models retain substantial problem-solving capabilities that can be elicited through explicit algorithmic guidance.
3. **Verifier feedback mitigates "thought collapse" but does not expand algorithm coverage** (§5.3). Without the verifier, we find a reduction in exploration while the task remains unresolved—a phenomenon we term "thought collapse". This suggests limitations in models' ability to reflect on failures and sustain reasoning across rounds. Although verifier feedback helps sustain reasoning and improves recovery rates, it does not lead to new algorithm recovery in our ablation, suggesting the limited effect of additional reflection rounds on algorithmic exploration.

2 Preliminaries

Foundational Algorithms in Computer Science. We regard foundational algorithms as core, time-tested computational procedures that solve a basic problem with proven efficiency. They are characterized by well-established theoretical properties, such as provable time and space complexity guarantees, with some further achieving optimality on specific problem classes. We select 20 foundational algorithms in computer science across graph theory, string processing, number theory, and data structures: *Dijkstra*, *Floyd-Warshall*, *Bellman-Ford*, *Prim*, *BFS*, *DAG*, *Euclidean*, *KMP*, *Manacher*, *Moore Vote*, *Gray*, *Strassen*, *Kadane*, *Huffman*, *Union-Find*, *Hungarian*, *Hierholzer*, *Stoer-Wagner*, *Pollard's Rho*, and *Inverse Burrows-Wheeler Transform*. We provide descriptions of these algorithms and their complexity in Appendix D.1.

LLM Unlearning. LLM unlearning aims to remove specific knowledge from a pretrained model, while preserving its general utility (Cao & Yang, 2015; Zhang et al., 2024). Let $\mathcal{D}_{\text{forget}}$ and $\mathcal{D}_{\text{retain}}$ denote the sets of examples to be removed and preserved, respectively. Existing LLM unlearning methods typically optimize an unlearning objective that balances forgetting and utility preservation:

$$\mathcal{L}_{\text{unlearn}}(\theta) = \mathcal{L}_{\text{forget}}(\theta; \mathcal{D}_{\text{forget}}) + \lambda \mathcal{L}_{\text{retain}}(\theta; \mathcal{D}_{\text{retain}}), \tag{1}$$

where $\mathcal{L}_{\text{forget}}$ denotes the forgetting objective, $\mathcal{L}_{\text{retain}}$ denotes a utility-preserving objective, and λ is a coefficient that controls the trade-off between forgetting and utility preservation.

A simple approach is to perform gradient ascent (GA) (Jang et al., 2023) on the language modeling (LM) loss on $\mathcal{D}_{\text{forget}}$. The preservation term can be instantiated as supervised fine-tuning on $\mathcal{D}_{\text{retain}}$ or as KL regularization toward a reference policy (Maini et al., 2024; Zhang et al., 2024). Additional unlearning methods are discussed in §6.

3 Algorithm Recovery after Unlearning

Our goal is to evaluate whether an LLM can recover a foundational algorithm once its direct recall of that algorithm is suppressed through unlearning. To this end, our framework proceeds in two phases (Figure 1). The unlearning phase aims to suppress the model's recall of the target algorithm while preserving its general utility (§3.1). The recovery phase then tests whether the unlearned model can recover the forgotten algorithm with various levels of hint (§3.2).

3.1 The Unlearning Phase

Following Zhang et al. (2025a) and Zaradoukas et al. (2026), we adopt a GRPO-based, on-policy unlearning method integrated with a cold start stage.

Group Relative Policy Optimization (GRPO) (Shao et al., 2024) is an on-policy variant of PPO (Schulman et al., 2017) that estimates policy advantages from the relative rewards of multiple responses sampled for the same prompt. To suppress direct recall of the target algorithm, we use a reward assigned by an LLM judge (Zheng et al., 2023) that favors responses that do not disclose target knowledge and remain readable.

Problem Setup. Let π_θ denote the policy being optimized and let π_{ref} be a fixed reference policy. We consider two datasets: a forget set $\mathcal{D}_{\text{forget}}$, which consists of concept-related queries related to the target algorithm g, and a retain set $\mathcal{D}_{\text{retain}}$, which contains non-target examples used to preserve general utility during unlearning.

Loss Function. For each query $x \in \mathcal{D}_{\text{forget}}$, we adopt a GRPO-style objective. At each optimization step, we sample a group $\mathbf{Y} = \{y_j\}_{j=1}^{G}$ of G responses from the behavior policy $\pi_{\theta_{\text{old}}}$, where θ_{old} denotes the policy parameters before the update and is held fixed during the subsequent optimization on this group. A KL penalty relative to the reference policy π_{ref} is additionally included to limit excessive policy drift. The forgetting objective is written as:

$$\mathcal{L}_{\text{forget-GRPO}}(\theta) = -\mathbb{E}_{y_j \sim \pi_{\theta_{\text{old}}}(\cdot|x)} \left[\frac{1}{|y_j|} \sum_{t=1}^{|y_j|} \mathcal{J}_{\text{clip},j,t}(\theta) \right] + \beta\, D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}}), \tag{2}$$

where $|y_j|$ is the response length, β is the coefficient of the KL regularization term, and the expectation $\mathbb{E}_{y_j \sim \pi_{\theta_{\text{old}}}(\cdot|x)}[\cdot]$ is estimated by the empirical mean over the group $\mathbf{Y}$ of G responses. The clipped surrogate term is given by $\mathcal{J}_{\text{clip},j,t}(\theta) = \min\left(\rho_{j,t}(\theta)A_j, \text{clip}(\rho_{j,t}(\theta), 1-\epsilon, 1+\epsilon)\, A_j\right)$, where $\rho_{j,t}(\theta) = \pi_\theta(y_{j,t}|x, y_{j,<t}) / \pi_{\theta_{\text{old}}}(y_{j,t}|x, y_{j,<t})$ is the importance ratio for the t-th token of the j-th sampled response y_j. ϵ is the clipping parameter, and A_j denotes the corresponding advantage, which is computed by normalizing the rewards within the sampled group:

$$A_j = \frac{r(x, y_j) - \bar{r}}{\sigma_r + \varepsilon}, \qquad \bar{r} = \frac{1}{G} \sum_{k=1}^{G} r(x, y_k), \tag{3}$$

where σ_r denotes the standard deviation of rewards within the group and ε is a small constant for numerical stability.

As in §2, the complete unlearning objective combines both terms:

$$\mathcal{L}_{\text{unlearn-GRPO}}(\theta) = \mathcal{L}_{\text{forget-GRPO}}(\theta; \mathcal{D}_{\text{forget}}) + \lambda\, \mathcal{L}_{\text{retain}}(\theta; \mathcal{D}_{\text{retain}}). \tag{4}$$

Table 1: Recovery Rate (RR, %) and Coverage (%) across algorithms, models, and hint levels. Higher is better. Level 1 uses high-level hints; level 2 uses step-by-step hints. Darker shading indicates higher value. The order is based on reference difficulty ratings of algorithms (Appendix D.2).

RR (%) ↘	Qwen3-4B-Thinking			Qwen3-4B-Instruct			Gemma-4-12B (Thinking)		
Algorithm ↓	no hint	level-1	level-2	no hint	level-1	level-2	no hint	level-1	level-2
Kadane	96.1	96.1	100.0	82.8	82.0	75.0	98.4	100.0	99.2
DAG	70.3	76.6	100.0	28.1	19.5	25.0	93.8	96.9	100.0
Gray	50.8	57.0	100.0	0.0	0.0	75.0	40.6	40.6	99.2
BFS	35.9	48.4	59.4	0.0	0.0	75.0	31.2	42.2	100.0
Euclidean	37.5	58.6	86.7	1.6	0.0	87.5	37.5	40.6	67.2
Moore Vote	0.0	0.0	100.0	0.0	5.5	43.8	0.0	3.1	87.5
Huffman	93.8	100.0	100.0	0.0	24.2	81.2	99.2	100.0	100.0
Union-Find	96.9	100.0	98.4	10.2	9.4	37.5	81.2	82.8	87.5
Floyd-Warshall	42.2	87.5	85.2	0.8	5.5	93.8	96.9	89.1	90.6
Dijkstra	55.5	70.3	85.9	8.6	35.9	50.0	88.3	95.3	100.0
Bellman-Ford	22.7	41.4	96.1	0.0	83.6	87.5	34.4	81.2	96.1
Prim	0.0	10.2	100.0	0.0	0.0	87.5	3.1	0.0	93.8
KMP	0.0	15.6	56.2	0.0	0.0	31.2	0.0	21.9	75.0
Hierholzer	0.0	0.0	4.7	0.0	0.0	18.8	0.0	9.4	3.1
Manacher	0.0	0.0	31.2	0.0	0.0	25.0	0.0	71.9	90.6
Stoer-Wagner	0.0	14.8	22.7	0.0	0.0	0.0	0.0	9.4	6.2
Pollard-Rho	0.0	0.0	75.0	0.0	0.8	0.0	0.0	0.0	62.5
Hungarian	0.0	0.0	24.2	0.0	0.0	0.0	0.0	3.1	43.8
BWT-Inverse	0.0	0.0	100.0	0.0	0.0	37.5	0.0	3.1	50.0
Strassen	0.0	0.0	25.0	0.0	0.0	0.0	0.0	0.0	99.2
Average RR (%)	**30.1**	**38.8**	**72.5**	**6.6**	**13.3**	**46.6**	**35.2**	**44.5**	**77.6**
Coverage (%)	**50.0**	**65.0**	**100.0**	**30.0**	**45.0**	**80.0**	**55.0**	**85.0**	**100.0**

Reward Function. Following the binary reward design of prior reinforcement unlearning work (Zaradoukas et al., 2026), we adopt an LLM-as-a-judge (Zheng et al., 2023) to assign two binary attributes to each sampled response y_j to a forget query $x \in \mathcal{D}_{\text{forget}}$: 1) knowledge disclosure k_j, indicating whether the response reveals target algorithm knowledge; 2) readability u_j, indicating whether the response remains fluent and understandable. We define the reward as

$$r(x, y_j) = \mathbb{1}\{(k_j, u_j) = (0, 1)\}^{1}. \tag{5}$$

That is, a response receives reward 1 only when it does not reveal target knowledge and remains readable; otherwise, the reward is 0. The full judge prompts are provided in Appendix C.2.

Cold Start. In our early attempts, at the beginning of on-policy unlearning, initial model responses to queries in $\mathcal{D}_{\text{forget}}$ typically receive zero reward, leaving little effective optimization signal for policy updates. To address this, we apply a cold start stage (Zhang et al., 2025a): we construct an initialization set $\mathcal{D}_{\text{init}} = \{(x_i, y_i)\}$ of target-related queries paired with refusal-style responses in the model's original style. We use $\mathcal{D}_{\text{init}}$ to guide the model toward refusal-style behavior on forget queries via supervised fine-tuning. Details are provided in Appendix F.

3.2 The Recovery Phase

Environment. After unlearning, we evaluate whether the model can recover the target algorithm by interacting with a Python interpreter. For a target algorithm, we define an application task with a prompt, test suite, runtime limit, and memory limit. The prompt presents a programming problem with efficiency requirements tailored to the target algorithm or its variants. The unlearned model interacts with the Python interpreter by executing candidate code and eventually submitting a final solution for evaluation. We discuss evaluation validity in Appendix E.2.

[1] $\mathbb{1}\{\cdot\}$ is the indicator function that equals 1 if the condition holds and 0 otherwise.

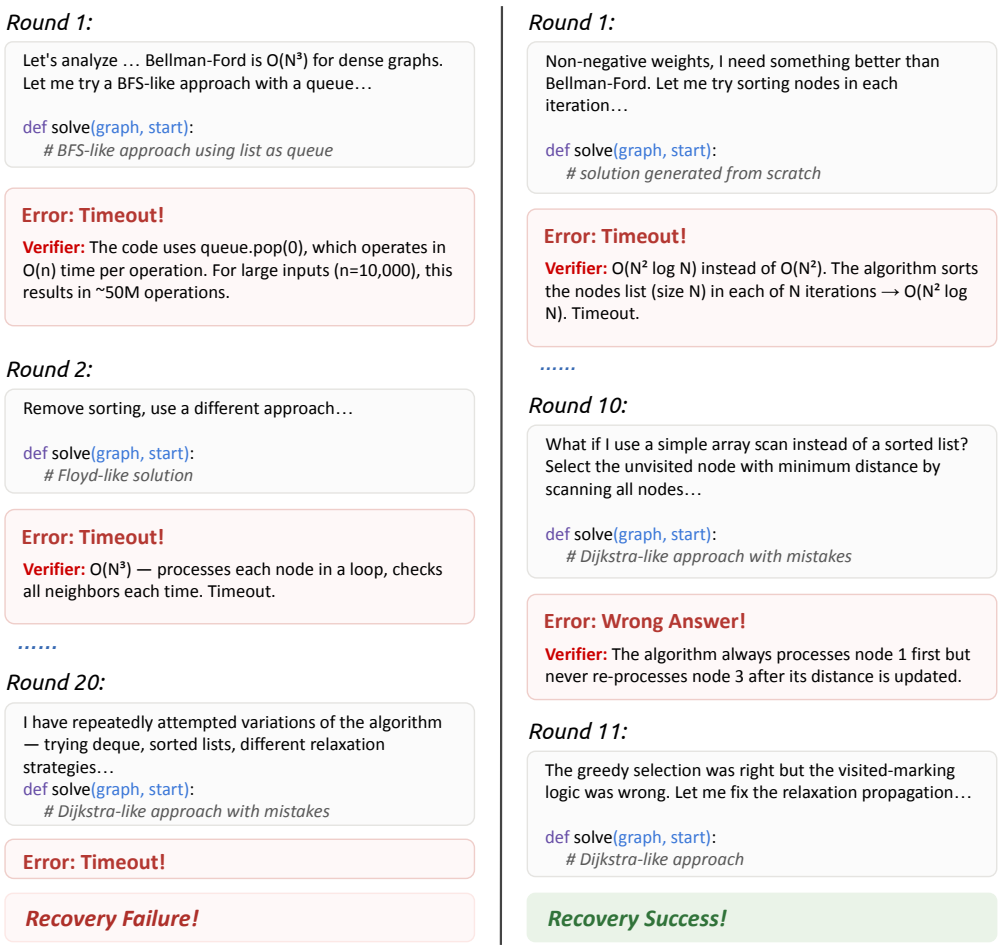

Figure 2: Two recovery trajectories for Dijkstra's algorithm (no hint) on Qwen3-4B-Thinking. **Left:** the model remains stuck in suboptimal strategies and eventually fails. **Right:** the model comes up with a Dijkstra-like solution by round 11, corrected through verifier feedback.

A submission is considered successful only if it passes the test suite within both the runtime limit and memory limit, which are calibrated according to the time and space complexity of the target algorithm. Calibration details are provided in Appendix E.3.

In preliminary experiments, we identify a failure mode during algorithm recovery after unlearning: after successive failed attempts, models usually produce shorter reasoning traces and abandon unresolved tasks. Motivated by recent AI research systems that employ generative verifiers at test time (Feng et al., 2026b;a), we incorporate a generative verifier (Zhang et al., 2025b) for failed submissions. By default, this verifier is instantiated by the unlearned model itself and returns diagnostic feedback to guide the model's revision. The verifier prompt template is provided in Appendix F.5. We examine the role of the verifier through ablation experiments in §5.3.

Hint Levels. To study how external hints affect recovery, we evaluate models under three prompt levels that vary in the amount of guidance provided: no hint beyond the task description, level 1 (a high-level conceptual hint about the algorithmic approach), and level 2 (a detailed step-by-step explanation in natural language). Hint examples are provided in Appendix F.4.

Remarks. One might worry that unlearning impairs algorithm-specific capabilities (e.g., by inducing persistent refusal-type responses on tasks involving the target algorithm). To reduce this

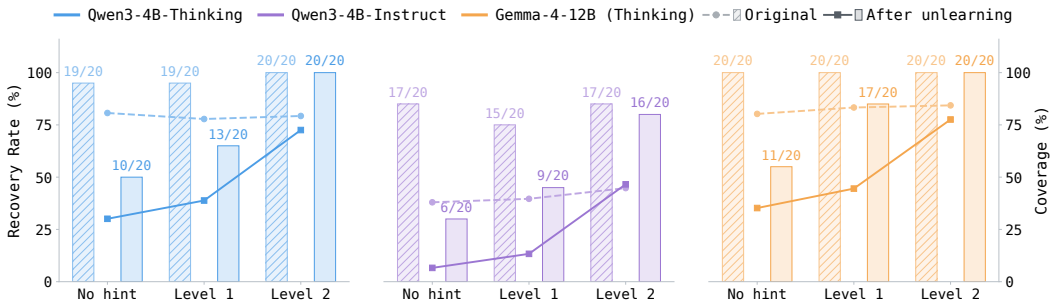

Figure 3: Recovery rate (lines) and algorithm coverage (bars) before and after unlearning across hint levels. More informative hints substantially narrow the recovery gap between unlearned and original models. With level-2 hints, unlearned Qwen3-4B-Instruct even achieves a better recovery rate than its original counterparts.

risk, we restrict our unlearning data to concept-related queries, excluding the application tasks used during recovery. Meanwhile, in subsequent experiments, algorithmic hints substantially narrow the performance gap between unlearned and original models (§5.1). This suggests that the unlearned models retain substantial coding and implementation capabilities in algorithm-specific settings.

4 Experimental Settings

Models. We select 3 strong open-weight models with 4B to 12B parameters as backbones for our unlearning and recovery experiments: Qwen3-4B-Thinking-2507, Qwen3-4B-Instruct-2507 (Team, 2025), and Gemma-4-12B (Team, 2026). We use DeepSeek-V4-Flash (DeepSeek-AI, 2026) for all LLM-as-a-judge components, with a sampling temperature of 1.0. Training details for each model are provided in Appendix C.

Evaluation. Following prior work on probing-based unlearning evaluation (Sun et al., 2025), we construct multiple-choice and open-ended questions for each target algorithm. We adapt the judge-based knowledge reproduction assessment of Tan et al. (2026). For each probe, the judge model examines both the reasoning process and the final response, assigning a label $y \in \{0, 1\}$ where $y = 1$ indicates that the model reveals no knowledge of the target algorithm. The Forgetting Rate (FR) is then defined as $\text{FR} = \frac{1}{N}\sum_{i=1}^{N} y_i$, where N is the total number of probes.

We evaluate retained model utility on LiveCodeBench (Jain et al., 2024) and AIME25 (Zhang & Math-AI, 2025). Following Qwen (2025), we use LiveCodeBench v6 [25.02–25.05]. For recovery, we report Recovery Rate (RR), defined as the fraction of successful attempts for a target algorithm, and Coverage, defined as the fraction of target algorithms with at least one successful attempt under the evaluation budget. For each algorithm, we construct 8 problem variants and run 128 trials at each prompt level across these variants. More evaluation settings are provided in Appendix E.

Datasets. For data construction, the target-specific queries in $\mathcal{D}_{\text{init}}$, $\mathcal{D}_{\text{forget}}$, and the evaluation probe sets are generated by DeepSeek-V4-Flash (DeepSeek-AI, 2026). The queries in $\mathcal{D}_{\text{retain}}$ are sampled from the `Code` and `Math` categories of Nemotron-Post-Training-Dataset-v1 (Nathawani et al., 2025; Bercovich et al., 2025). Appendix F describes the dataset construction process and provides dataset statistics, prompt templates, and examples.

5 Experiments

5.1 Main Results

In this section, we compare the unlearned models (Table 1) with their original counterparts (Table 4), focusing on Recovery Rates and Coverages (introduced in §4).

Nearly Half of the Target Algorithms Remain Unrecovered without Hints. As shown in Table 1 and Table 4 in Appendix A.1, we observe a substantial gap between the original models' task performance and their recovery rate after unlearning. Before unlearning, the two thinking models

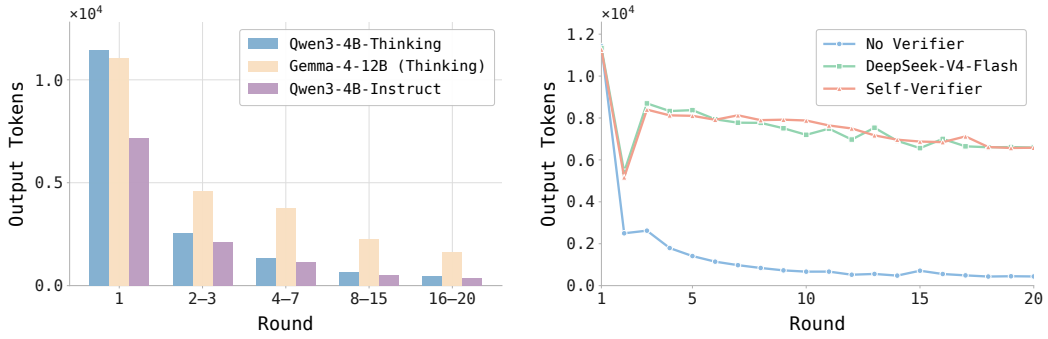

Figure 4: Round-wise output tokens during recovery. **Left**: Across three models without verifier feedback, outputs become shorter in later rounds. **Right**: Three verifier settings on Qwen3-4B-Thinking. Without a verifier, the model either succeeds early or quickly abandons the task. With a verifier, the model sustains meaningful exploration over more rounds, enabling success even when the correct approach is not found initially.

could successfully solve almost all algorithms in our setting. However, although unlearning does not guarantee complete removal of target-algorithm knowledge, 9 of the 20 foundational algorithms have no successful recovery across the three evaluated models without hints.

Recovery also exhibits a clear pattern with respect to the difficulty ratings of the algorithms. We use the Luogu ratings (Luogu) reported in Appendix D.2 as a coarse proxy for algorithm difficulty. Among the 12 algorithms whose template problems are rated at levels 2–3, 11 are recovered by at least one model without hints. In contrast, none of the 6 algorithms with template problems rated at levels 4–7 are recovered under the same no-hint setting. This gap suggests that recovery becomes harder as algorithms grow more complex, suggesting models' limited ability to achieve innovation on difficult foundational algorithms. To better understand model behavior during recovery, we provide a representative success and failure trajectory for Dijkstra's algorithm in Figure 2, with more examples in Appendix G.1.

High Forgetting Rates Coexist with Substantial Retained Utility. As shown in Table 3, the average forgetting rates over the reported targets are 96.0%, 94.0%, and 89.6% for Qwen3-4B-Thinking, Qwen3-4B-Instruct, and Gemma-4-12B, respectively, indicating that most probe responses are judged not to reveal target-algorithm knowledge. Meanwhile, the unlearned models retain substantial performance on LCB and AIME25. These results show that recovery failures coexist with substantial retained performance on broader reasoning tasks.

Algorithmic Hints Narrow the Recovery Gap. As shown in Figure 3, for each of the three backbone models, increasingly detailed hints improve average recovery rates and algorithm coverage after unlearning, narrowing the gap relative to the original model. Compared with the no-hint setting, level-1 hints enable recovery of additional algorithms, while level-2 hints yield larger overall gains. Notably, at level 2, the unlearned variants of Qwen3-4B-Instruct achieve a higher average recovery rate than the original model. Across hint levels, we also observe successful recovery on some algorithm–hint combinations for which the corresponding original models produce nearly no successful solutions. For example, under level-2 hints, unlearned Qwen3-4B-Instruct variants achieve recovery rates of 37.5% and 18.8% on BWT-Inverse and Hierholzer, respectively, compared with 0.8% and 0.0% for the original model. These results show that the unlearned models can effectively use explicit algorithmic guidance, in some cases even achieving successful "recovery" beyond the original models' observed coverage.

5.2 Recovery under Algorithm-Name Suppression

Beyond parameter-based unlearning, we examine whether restricting the generation of algorithm names affects recovery without modifying model parameters. We apply this intervention to Qwen3-4B-Thinking and Gemma-4-12B by masking a target-specific set of token IDs selected to suppress algorithm-name generation at each decoding step. We evaluate all 20 algorithms without hints, using 128 trials per algorithm. Mask construction and decoding details are provided in Appendix B.

Algorithm-name suppression substantially reduces recovery. As shown in Figure 5, mean recovery rate decreases from 80.7% to 57.9% for Qwen3-4B-Thinking and from 80.3% to 34.8% for Gemma-4-12B, corresponding to drops of 22.9 and 45.4 percentage points, respectively. Algorithm coverage decreases from 95% to 80% for Qwen and from 100% to 85% for Gemma. On the other hand, performance on broader reasoning benchmarks remains strong (Table 8, Appendix B.4).

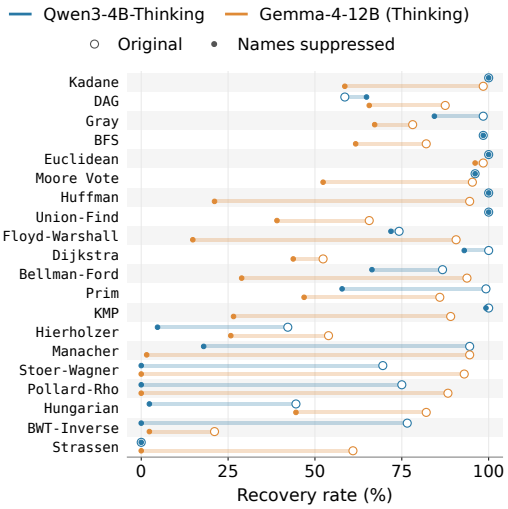

Figure 5: Effect of algorithm-name suppression on Qwen3-4B-Thinking and Gemma-4-12B (Thinking) without hints. Each line links an algorithm's original recovery rate to its rate after name suppression. Line length indicates the absolute change.

Algorithm names serve as retrieval cues. The recovery rate degradation suggests the important role of name-associated retrieval in such tasks. Without generating the algorithm's name, the model must express a solution through alternative descriptions or directly in code. Notably, we observe that large declines occur for algorithms such as Manacher, Pollard–Rho, and BWT-Inverse, while simpler algorithms such as Euclidean remain stable. This might be due to their proportion in pretraining data or their difficulty, making such alternative routes less accessible for these harder algorithms. These findings suggest the model's overreliance on name-associated memory when solving foundational algorithmic problems. More case studies and behaviour analyses are provided in Appendix B.

5.3 Ablation of the Generative Verifier

To isolate the contribution of the generative verifier, we remove verifier feedback and analyze recovery trajectories across three models. Without feedback, model outputs shorten over rounds (Figure 4), reflecting reduced exploration—later rounds contain fewer candidate ideas, and in some cases the model abandons problem-solving entirely or attributes failures to the testing environment rather than its own solution. This suggests limitations in the models' ability to reflect on failures and sustain reasoning across multiple interaction rounds. We refer to this phenomenon as *thought collapse*; additional examples are provided in Appendix G.2.

Table 2: Verifier ablation on Qwen3-4B-Thinking in the no-hint setting. Recovery Rate (RR, %) and Coverage (%) are averaged across 20 algorithms.

Verifier	RR (%)	Coverage (%)
No Verifier	20.5	50.0
DeepSeek-V4-Flash	36.6	50.0
Self-Verifier	30.1	50.0

We then test whether verifier feedback can mitigate "thought collapse" on Qwen3-4B-Thinking. We compare three settings: no verifier, the default self-verifier, and a strong verifier implemented with DeepSeek-V4-Flash. Compared with the no-feedback setting, verifiers help maintain longer output lengths across rounds (Figure 4). These gains are also reflected in a higher recovery rate (Table 2), suggesting that natural-language failure diagnosis helps sustain exploration and improve recovery outcomes. However, verifiers improve recovery frequency without increasing the number of recovered algorithms. We further investigate this pattern by analyzing how coverage varies with the number of trials and interaction turns in Appendix A.2.

6 Related Work

We discuss the most relevant studies here and provide a more comprehensive review in Appendix D.3. Recent LLM-based systems have shown promise in scientific research and algorithmic discovery (Romera-Paredes et al., 2023; Novikov et al., 2025; Gottweis et al., 2025). Our work examines whether models can recover foundational algorithms when direct recall of the target knowledge is suppressed. LLM unlearning provides a practical intervention for this purpose, with existing methods based on gradient ascent, preference optimization, and on-policy reinforcement learning (Jang et al.,

Table 3: Unlearning results across target algorithms and models. Forgetting Rate (FR, %) measures unlearning effectiveness, while LCB and AIME25 (detailed in §4) measure model utility. *Origin* denotes the original model before unlearning.

Algorithm	Qwen3-4B-Thinking			Qwen3-4B-Instruct			Gemma-4-12B (Thinking)		
	FR (%)	LCB	AIME25	FR (%)	LCB	AIME25	FR (%)	LCB	AIME25
Origin	0.0	55.2	81.3	0.0	35.1	47.4	0.0	71.8	82.2
Kadane	100.0	50.4	78.9	96.6	32.1	47.8	94.9	63.4	68.9
DAG	86.7	48.1	80.0	96.7	38.9	47.8	91.7	62.6	70.0
Gray	97.1	50.4	74.4	91.4	38.9	47.8	88.6	66.4	73.3
BFS	91.7	51.1	71.1	96.7	41.2	45.6	85.0	72.5	71.1
Euclidean	100.0	50.4	74.4	95.0	43.5	40.0	80.0	67.9	72.2
Moore Vote	97.4	48.9	78.9	97.4	38.2	46.7	92.3	66.4	71.1
Huffman	98.3	49.6	78.9	95.0	35.9	44.4	88.3	71.0	72.2
Union-Find	96.7	48.1	77.8	98.3	31.3	43.3	90.0	68.7	72.2
Floyd-Warshall	100.0	54.2	76.7	87.5	35.9	47.8	85.0	65.6	73.3
Dijkstra	72.1	49.6	76.7	82.0	33.6	38.9	73.8	68.7	67.8
Bellman-Ford	92.5	54.2	76.7	90.0	30.5	44.4	87.5	72.5	68.9
Prim	100.0	48.9	78.9	85.0	35.9	42.2	87.5	74.8	76.7
KMP	95.0	48.1	75.6	90.0	32.8	44.4	97.5	65.6	67.5
Hierholzer	100.0	46.6	72.2	96.7	32.1	46.7	90.2	61.1	64.4
Manacher	100.0	49.6	76.7	92.0	40.5	44.4	92.0	61.1	70.0
Stoer-Wagner	100.0	47.3	76.7	98.3	35.9	42.2	90.0	66.4	73.3
Pollard-Rho	100.0	50.4	77.8	98.3	38.9	43.3	95.0	64.1	66.7
Hungarian	93.3	48.1	80.0	95.0	38.2	45.6	90.0	66.4	74.4
BWT-Inverse	100.0	48.9	78.9	100.0	36.6	41.1	98.4	65.6	71.1
Strassen	100.0	48.9	81.1	98.0	35.1	43.3	94.0	64.1	71.1
Average	**96.0**	**49.6**	**77.1**	**94.0**	**36.3**	**44.4**	**89.6**	**66.7**	**70.8**

2023; Zhang et al., 2024; 2025a; Zaradoukas et al., 2026). Most closely related, Yang (2025) proposes using unlearning as an ablation probe to investigate whether LLMs can re-derive scientific results after targeted knowledge removal. We develop this conceptual proposal into a concrete experimental framework, examining recovery under different hint levels and verifier settings, alongside a complementary algorithm-name suppression intervention.

7 Conclusion and Discussion

In this work, we study whether LLMs can recover foundational algorithms when direct recall is suppressed. Our experiments reveal two main findings: First, although unlearning does not guarantee complete knowledge removal, nearly half of the algorithms remain unrecovered without hints. Second, suppressing algorithm-name-related tokens alone reduces recovery without modifying model parameters. These results indicate LLMs' overreliance on existing knowledge and familiar retrieval cues.

It is also worth noting that we observe "thought collapse" and introduce a verifier to mitigate it. Though verifier feedback sustains reasoning and improves recovery rates, continued revision does not expand algorithm coverage within our evaluation budget. This suggests limitations in self-reflection and its ability to broaden algorithmic exploration. **By treating the unlearned model's recovery rate as an approximate upper bound, our results suggest that current LLMs still have limited ability to make foundational algorithm innovation.** We hope our findings could provide a new perspective on the gap between knowledge-based algorithm solving and algorithmic discovery, and offer insights into both the potential and current limits of LLMs' innovative reasoning.

8 Limitations and Future Work

We conclude with several limitations and directions for future work. Our experiments cover a limited set of settings and algorithms; it remains unclear whether our findings generalize to other scientific domains or larger models. Meanwhile, unlearning does not guarantee equivalence to a model retrained without the target knowledge. Future work may use retraining with controlled exclusion of target knowledge or alternative interventions, and extend the evaluation to larger models and broader discovery tasks.

AI USE STATEMENT

We used generative AI tools for language polishing, literature searches, and LaTeX formatting. These tools were also used to obtain critical feedback on the research methodology and the interpretation of results. We also used DeepSeek-V4-Flash to generate target-specific queries for the cold-start and unlearning datasets and evaluation probe sets, as detailed in Appendix F. The authors take responsibility for the final content of this paper.

REFERENCES

Akhiad Bercovich, Itay Levy, Izik Golan, Mohammad Dabbah, Ran El-Yaniv, Omri Puny, Ido Galil, Zach Moshe, Tomer Ronen, Najeeb Nabwani, Ido Shahaf, Oren Tropp, Ehud Karpas, Ran Zilberstein, Jiaqi Zeng, Soumye Singhal, Alexander Bukharin, Yian Zhang, Tugrul Konuk, Gerald Shen, Ameya Sunil Mahabaleshwarkar, Bilal Kartal, Yoshi Suhara, Olivier Delalleau, Zijia Chen, Zhilin Wang, David Mosallanezhad, Adi Renduchintala, Haifeng Qian, Dima Rekesh, Fei Jia, Somshubra Majumdar, Vahid Noroozi, Wasi Uddin Ahmad, Sean Narenthiran, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Siddhartha Jain, Igor Gitman, Ivan Moshkov, Wei Du, Shubham Toshniwal, George Armstrong, Branislav Kisacanin, Matvei Novikov, Daria Gitman, Evelina Bakhturina, Jane Polak Scowcroft, John Kamalu, Dan Su, Kezhi Kong, Markus Kliegl, Rabeeh Karimi, Ying Lin, Sanjeev Satheesh, Jupinder Parmar, Pritam Gundecha, Brandon Norick, Joseph Jennings, Shrimai Prabhumoye, Syeda Nahida Akter, Mostofa Patwary, Abhinav Khattar, Deepak Narayanan, Roger Waleffe, Jimmy Zhang, Bor-Yiing Su, Guyue Huang, Terry Kong, Parth Chadha, Sahil Jain, Christine Harvey, Elad Segal, Jining Huang, Sergey Kashirsky, Robert McQueen, Izzy Putterman, George Lam, Arun Venkatesan, Sherry Wu, Vinh Nguyen, Manoj Kilaru, Andrew Wang, Anna Warno, Abhilash Somasamudramath, Sandip Bhaskar, Maka Dong, Nave Assaf, Shahar Mor, Omer Ullman Argov, Scot Junkin, Oleksandr Romanenko, Pedro Larroy, Monika Katariya, Marco Rovinelli, Viji Balas, Nicholas Edelman, Anahita Bhiwandiwalla, Muthu Subramaniam, Smita Ithape, Karthik Ramamoorthy, Yuting Wu, Suguna Varshini Velury, Omri Almog, Joyjit Daw, Denys Fridman, Erick Galinkin, Michael Evans, Katherine Luna, Leon Derczynski, Nikki Pope, Eileen Long, Seth Schneider, Guillermo Siman, Tomasz Grzegorzek, Pablo Ribalta, Monika Katariya, Joey Conway, Trisha Saar, Ann Guan, Krzysztof Pawelec, Shyamala Prayaga, Oleksii Kuchaiev, Boris Ginsburg, Oluwatobi Olabiyi, Kari Briski, Jonathan Cohen, Bryan Catanzaro, Jonah Alben, Yonatan Geifman, Eric Chung, and Chris Alexiuk. Llama-nemotron: Efficient reasoning models, 2025. URL https://arxiv.org/abs/2505.00949.

Yinzhi Cao and Junfeng Yang. Towards making systems forget with machine unlearning. In *2015 IEEE Symposium on Security and Privacy*, pp. 463–480, 2015. doi: 10.1109/SP.2015.35.

Pierre Chambon, Kunhao Zheng, Juliette Decugis, Benoit Sagot, and Gabriel Synnaeve. Reinforcement learning for code optimization, 2026. URL https://arxiv.org/abs/2607.25970.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Chongyu Fan, Jiancheng Liu, Licong Lin, Jinghan Jia, Ruiqi Zhang, Song Mei, and Sijia Liu. Simplicity prevails: Rethinking negative preference optimization for llm unlearning, 2025. URL https://arxiv.org/abs/2410.07163.

Tony Feng, Junehyuk Jung, Sang hyun Kim, Carlo Pagano, Sergei Gukov, Chiang-Chiang Tsai, David Woodruff, Adel Javanmard, Aryan Mokhtari, Dawsen Hwang, Yuri Chervonyi, Jonathan N. Lee, Garrett Bingham, Trieu H. Trinh, Vahab Mirrokni, Quoc V. Le, and Thang Luong. Aletheia tackles firstproof autonomously, 2026a. URL https://arxiv.org/abs/2602.21201.

Tony Feng, Trieu H. Trinh, Garrett Bingham, Dawsen Hwang, Yuri Chervonyi, Junehyuk Jung, Joonkyung Lee, Carlo Pagano, Sang hyun Kim, Federico Pasqualotto, Sergei Gukov, Jonathan N. Lee, Junsu Kim, Kaiying Hou, Golnaz Ghiasi, Yi Tay, YaGuang Li, Chenkai Kuang, Yuan Liu, Hanzhao Lin, Evan Zheran Liu, Nigamaa Nayakanti, Xiaomeng Yang, Heng-Tze Cheng, Demis Hassabis, Koray Kavukcuoglu, Quoc V. Le, and Thang Luong. Towards autonomous mathematics research, 2026b. URL https://arxiv.org/abs/2602.10177.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, Khaled Saab, Dan Popovici,

Jacob Blum, Fan Zhang, Katherine Chou, Avinatan Hassidim, Burak Gokturk, Amin Vahdat, Pushmeet Kohli, Yossi Matias, Andrew Carroll, Kavita Kulkarni, Nenad Tomasev, Yuan Guan, Vikram Dhillon, Eeshit Dhaval Vaishnav, Byron Lee, Tiago R D Costa, José R Penadés, Gary Peltz, Yunhan Xu, Annalisa Pawlosky, Alan Karthikesalingam, and Vivek Natarajan. Towards an ai co-scientist, 2025. URL https://arxiv.org/abs/2502.18864.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645 (8081):633–638, September 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Dang Huu-Tien, Hoang Thanh-Tung, Anh Bui, Minh-Phuong Nguyen, Le-Minh Nguyen, and Naoya Inoue. Improving llm unlearning robustness via random perturbations, 2025. URL https://arxiv.org/abs/2501.19202.

Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Suchin Gururangan, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. Editing models with task arithmetic, 2023. URL https://arxiv.org/abs/2212.04089.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Joel Jang, Dongkeun Yoon, Sohee Yang, Sungmin Cha, Moontae Lee, Lajanugen Logeswaran, and Minjoon Seo. Knowledge unlearning for mitigating privacy risks in language models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14389–14408, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.805. URL https://aclanthology.org/2023.acl-long.805/.

Bruce W. Lee, Addie Foote, Alex Infanger, Leni Shor, Harish Kamath, Jacob Goldman-Wetzler, Bryce Woodworth, Alex Cloud, and Alexander Matt Turner. Distillation robustifies unlearning, 2025. URL https://arxiv.org/abs/2506.06278.

Nathaniel Li, Alexander Pan, Anjali Gopal, Summer Yue, Daniel Berrios, Alice Gatti, Justin D. Li, Ann-Kathrin Dombrowski, Shashwat Goel, Long Phan, Gabriel Mukobi, Nathan Helm-Burger,

Rassin Lababidi, Lennart Justen, Andrew B. Liu, Michael Chen, Isabelle Barrass, Oliver Zhang, Xiaoyuan Zhu, Rishub Tamirisa, Bhrugu Bharathi, Adam Khoja, Zhenqi Zhao, Ariel Herbert-Voss, Cort B. Breuer, Samuel Marks, Oam Patel, Andy Zou, Mantas Mazeika, Zifan Wang, Palash Oswal, Weiran Lin, Adam A. Hunt, Justin Tienken-Harder, Kevin Y. Shih, Kemper Talley, John Guan, Russell Kaplan, Ian Steneker, David Campbell, Brad Jokubaitis, Alex Levinson, Jean Wang, William Qian, Kallol Krishna Karmakar, Steven Basart, Stephen Fitz, Mindy Levine, Ponnurangam Kumaraguru, Uday Tupakula, Vijay Varadharajan, Ruoyu Wang, Yan Shoshitaishvili, Jimmy Ba, Kevin M. Esvelt, Alexandr Wang, and Dan Hendrycks. The wmdp benchmark: Measuring and reducing malicious use with unlearning, 2024. URL https://arxiv.org/abs/2403.03218.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery, 2024. URL https://arxiv.org/abs/2408.06292.

Luogu. Problem difficulty rating system. https://help.luogu.com.cn/manual/luogu/problem/difficulty. Accessed: September 24, 2026.

Pratyush Maini, Zhili Feng, Avi Schwarzschild, Zachary C. Lipton, and J. Zico Kolter. Tofu: A task of fictitious unlearning for llms, 2024. URL https://arxiv.org/abs/2401.06121.

Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Koppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin Riedmiller, Oriol Vinyals, and David Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023. doi: 10.1038/s41586-023-06004-9.

Dhruv Nathawani, Igor Gitman, Somshubra Majumdar, Evelina Bakhturina, Ameya Sunil Mahabaleshwarkar, , Jian Zhang, and Jane Polak Scowcroft. Nemotron-Post-Training-Dataset-v1, July 2025. URL https://huggingface.co/datasets/nvidia/Nemotron-Post-Training-Dataset-v1.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrew Duberstein, Andrew Kondrich, Andrey Mishchenko, Andy Applebaum, Angela Jiang, Ashvin Nair, Barret Zoph, Behrooz Ghorbani, Ben Rossen, Benjamin Sokolowsky, Boaz Barak, Bob McGrew, Borys Minaiev, Botao Hao, Bowen Baker, Brandon Houghton, Brandon McKinzie, Brydon Eastman, Camillo Lugaresi, Cary Bassin, Cary Hudson, Chak Ming Li, Charles de Bourcy, Chelsea Voss, Chen Shen, Chong Zhang, Chris Koch, Chris Orsinger, Christopher Hesse, Claudia Fischer, Clive Chan, Dan Roberts, Daniel Kappler, Daniel Levy, Daniel Selsam, David Dohan, David Farhi, David Mely, David Robinson, Dimitris Tsipras, Doug Li, Dragos Oprica, Eben Freeman, Eddie Zhang, Edmund Wong, Elizabeth Proehl, Enoch Cheung, Eric Mitchell, Eric Wallace, Erik Ritter, Evan Mays, Fan Wang, Felipe Petroski Such, Filippo Raso, Florencia Leoni, Foivos Tsimpourlas, Francis Song, Fred von Lohmann, Freddie Sulit, Geoff Salmon, Giambattista Parascandolo, Gildas Chabot, Grace Zhao, Greg Brockman, Guillaume Leclerc, Hadi Salman, Haiming Bao, Hao Sheng, Hart Andrin, Hessam Bagherinezhad, Hongyu Ren, Hunter Lightman, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian Osband, Ignasi Clavera Gilaberte, Ilge Akkaya, Ilya Kostrikov, Ilya Sutskever, Irina Kofman, Jakub Pachocki, James Lennon, Jason Wei, Jean Harb, Jerry Twore, Jiacheng Feng, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joaquin Quiñonero Candela, Joe Palermo, Joel Parish, Johannes Heidecke, John Hallman, John Rizzo, Jonathan Gordon, Jonathan Uesato, Jonathan Ward, Joost Huizinga, Julie Wang, Kai Chen, Kai Xiao, Karan Singhal, Karina Nguyen, Karl Cobbe, Katy Shi, Kayla Wood, Kendra Rimbach, Keren Gu-Lemberg, Kevin Liu, Kevin Lu, Kevin

Stone, Kevin Yu, Lama Ahmad, Lauren Yang, Leo Liu, Leon Maksin, Leyton Ho, Liam Fedus, Lilian Weng, Linden Li, Lindsay McCallum, Lindsey Held, Lorenz Kuhn, Lukas Kondraciuk, Lukasz Kaiser, Luke Metz, Madelaine Boyd, Maja Trebacz, Manas Joglekar, Mark Chen, Marko Tintor, Mason Meyer, Matt Jones, Matt Kaufer, Max Schwarzer, Meghan Shah, Mehmet Yatbaz, Melody Y. Guan, Mengyuan Xu, Mengyuan Yan, Mia Glaese, Mianna Chen, Michael Lampe, Michael Malek, Michele Wang, Michelle Fradin, Mike McClay, Mikhail Pavlov, Miles Wang, Mingxuan Wang, Mira Murati, Mo Bavarian, Mostafa Rohaninejad, Nat McAleese, Neil Chowdhury, Neil Chowdhury, Nick Ryder, Nikolas Tezak, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Patrick Chao, Paul Ashbourne, Pavel Izmailov, Peter Zhokhov, Rachel Dias, Rahul Arora, Randall Lin, Rapha Gontijo Lopes, Raz Gaon, Reah Miyara, Reimar Leike, Renny Hwang, Rhythm Garg, Robin Brown, Roshan James, Rui Shu, Ryan Cheu, Ryan Greene, Saachi Jain, Sam Altman, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Santiago Hernandez, Sasha Baker, Scott McKinney, Scottie Yan, Shengjia Zhao, Shengli Hu, Shibani Santurkar, Shraman Ray Chaudhuri, Shuyuan Zhang, Siyuan Fu, Spencer Papay, Steph Lin, Suchir Balaji, Suvansh Sanjeev, Szymon Sidor, Tal Broda, Aidan Clark, Tao Wang, Taylor Gordon, Ted Sanders, Tejal Patwardhan, Thibault Sottiaux, Thomas Degry, Thomas Dimson, Tianhao Zheng, Timur Garipov, Tom Stasi, Trapit Bansal, Trevor Creech, Troy Peterson, Tyna Eloundou, Valerie Qi, Vineet Kosaraju, Vinnie Monaco, Vitchyr Pong, Vlad Fomenko, Weiyi Zheng, Wenda Zhou, Wes McCabe, Wojciech Zaremba, Yann Dubois, Yinghai Lu, Yining Chen, Young Cha, Yu Bai, Yuchen He, Yuchen Zhang, Yunyun Wang, Zheng Shao, and Zhuohan Li. Openai o1 system card, 2024. URL https://arxiv.org/abs/2412.16720.

Xiangyu Qi, Ashwinee Panda, Kaifeng Lyu, Xiao Ma, Subhrajit Roy, Ahmad Beirami, Prateek Mittal, and Peter Henderson. Safety alignment should be made more than just a few tokens deep, 2024. URL https://arxiv.org/abs/2406.05946.

Qwen. Qwen/qwen3-4b-thinking-2507. https://huggingface.co/Qwen/Qwen3-4B-Thinking-2507, 2025.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 2023. doi: 10.1038/s41586-023-06924-6.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

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. URL https://arxiv.org/abs/2402.03300.

Chenglei Si, Diyi Yang, and Tatsunori Hashimoto. Can llms generate novel research ideas? a large-scale human study with 100+ nlp researchers, 2024. URL https://arxiv.org/abs/2409.04109.

Guangzhi Sun, Potsawee Manakul, Xiao Zhan, and Mark Gales. Unlearning vs. obfuscation: Are we truly removing knowledge?, 2025. URL https://arxiv.org/abs/2505.02884.

Chenchen Tan, Youyang Qu, Xinghao Li, Hui Zhang, Shujie Cui, Cunjian Chen, and Longxiang Gao. Wisdom is knowing what not to say: Hallucination-free llms unlearning via attention shifting, 2026. URL https://arxiv.org/abs/2510.17210.

Gemma Team. Gemma 4 technical report, 2026. URL https://arxiv.org/abs/2607.02770.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Robert Yang. Unlearning as ablation: Toward a falsifiable benchmark for generative scientific discovery, 2025. URL https://arxiv.org/abs/2508.17681.

Jin Yao, Eli Chien, Minxin Du, Xinyao Niu, Tianhao Wang, Zezhou Cheng, and Xiang Yue. Machine unlearning of pre-trained large language models, 2024. URL https://arxiv.org/abs/2402.15159.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time, 2026. URL https://arxiv.org/abs/2601.16175.

Efstratios Zaradoukas, Bardh Prenkaj, and Gjergji Kasneci. Reinforcement unlearning via group relative policy optimization, 2026. URL https://arxiv.org/abs/2601.20568.

Chenlong Zhang, Zhuoran Jin, Hongbang Yuan, Jiaheng Wei, Tong Zhou, Kang Liu, Jun Zhao, and Yubo Chen. Rule: Reinforcement unlearning achieves forget-retain pareto optimality, 2025a. URL https://arxiv.org/abs/2506.07171.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction, 2025b. URL https://arxiv.org/abs/2408.15240.

Qingjie Zhang, Haoting Qian, Zhicong Huang, Cheng Hong, Minlie Huang, Ke Xu, Chao Zhang, and Han Qiu. Understanding the dilemma of unlearning for large language models, 2025c. URL https://arxiv.org/abs/2509.24675.

Ruiqi Zhang, Licong Lin, Yu Bai, and Song Mei. Negative preference optimization: From catastrophic collapse to effective unlearning, 2024. URL https://arxiv.org/abs/2404.05868.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2025. https://huggingface.co/datasets/math-ai/aime25, 2025.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023. URL https://arxiv.org/abs/2306.05685.

Tianshi Zheng, Kelvin Kiu-Wai Tam, Newt Hue-Nam K. Nguyen, Baixuan Xu, Zhaowei Wang, Jiayang Cheng, Hong Ting Tsang, Weiqi Wang, Jiaxin Bai, Tianqing Fang, Yangqiu Song, Ginny Y. Wong, and Simon See. Newtonbench: Benchmarking generalizable scientific law discovery in llm agents, 2025. URL https://arxiv.org/abs/2510.07172.

APPENDIX

A ADDITIONAL RESULTS

A.1 NON-UNLEARNED ABLATION.

We evaluate the original, non-unlearned models under the identical recovery protocol. As shown in Table 4, hints do not always help. On several algorithms, more detailed hints lower the success rate. We attribute this to a strategy shift: without hints, the model recalls a known algorithm; with detailed hints, it implements the description step by step, which is more prone to misinterpretation and complexity violations.

A.2 SCALING THE RECOVERY BUDGET

We examine how algorithm coverage scales with the number of trials and interaction turns under the no-hint setting with a self-verifier. Figure 6 shows that coverage saturates rapidly with interaction turns, typically within the first two to five turns, especially at larger trial budgets. For Qwen3-4B-Thinking with $n = 128$, coverage reaches its final value in the first turn. Together with the verifier ablation, this pattern suggests that continued self-reflection mainly improves recovery frequency within an already recoverable set of algorithms. Within the evaluated budgets, increasing the number of trials may therefore offer greater scope for broadening algorithm coverage than extending verifier-guided revision.

One possible interpretation is that recovery success depends strongly on the direction explored at the beginning of a trial. Additional trials provide opportunities to explore different starting directions,

Table 4: Non-unlearned Recovery Rate (RR, %) and Coverage (%) across target algorithms, models, and hint levels. Higher is better. Level 1 uses high-level hints; level 2 uses step-by-step hints. Each reported cell is based on 128 samples. Darker shading indicates higher value.

RR (%) ↘	Qwen3-4B-Thinking			Qwen3-4B-Instruct			Gemma-4-12B (Thinking)		
Algorithm ↓	no hint	level-1	level-2	no hint	level-1	level-2	no hint	level-1	level-2
Kadane	100.0	100.0	100.0	99.2	91.4	96.1	98.4	99.2	100.0
DAG	58.6	67.2	33.6	21.9	14.8	13.3	87.5	91.4	89.8
Gray	98.4	92.2	100.0	53.9	28.1	75.0	78.1	84.4	97.7
BFS	98.4	97.7	100.0	9.4	7.0	1.6	82.0	85.9	86.7
Euclidean	100.0	100.0	100.0	98.4	97.7	91.4	98.4	100.0	99.2
Moore Vote	96.1	95.3	100.0	63.3	33.6	72.7	95.3	91.4	85.9
Huffman	100.0	100.0	100.0	83.6	65.6	60.2	94.5	94.5	93.0
Union-Find	100.0	100.0	99.2	21.1	24.2	28.9	65.6	53.1	57.8
Floyd-Warshall	74.2	69.5	68.8	51.6	57.8	48.4	90.6	91.4	91.4
Dijkstra	100.0	100.0	100.0	2.3	89.8	88.3	52.3	72.7	84.4
Bellman-Ford	86.7	71.1	59.4	100.0	96.9	95.3	93.8	89.1	72.7
Prim	99.2	98.4	100.0	15.6	21.9	76.6	85.9	87.5	86.7
KMP	100.0	100.0	100.0	93.8	96.9	91.4	89.1	93.0	93.8
Hierholzer	42.2	35.2	50.0	0.0	0.0	0.0	53.9	56.2	71.1
Manacher	94.5	98.4	99.2	40.6	59.4	49.2	94.5	95.3	93.8
Stoer-Wagner	69.5	75.8	75.8	1.6	0.0	0.8	93.0	96.1	93.0
Pollard-Rho	75.0	54.7	63.3	1.6	7.8	5.5	88.3	86.7	90.6
Hungarian	44.5	10.2	1.6	1.6	0.0	0.0	82.0	80.5	78.9
BWT-Inverse	76.6	90.6	96.1	0.0	0.0	0.8	21.1	48.4	57.8
Strassen	0.0	0.0	38.3	0.0	0.0	0.0	60.9	69.5	62.5
Average RR (%)	**80.7**	**77.8**	**79.3**	**38.0**	**39.6**	**44.8**	**80.3**	**83.3**	**84.3**
Coverage (%)	**95.0**	**95.0**	**100.0**	**85.0**	**75.0**	**85.0**	**100.0**	**100.0**	**100.0**

while subsequent self-reflection tends to refine the current approach. As readily accessible directions are exhausted, generating qualitatively new approaches may become increasingly difficult, leading to diminishing gains in coverage.

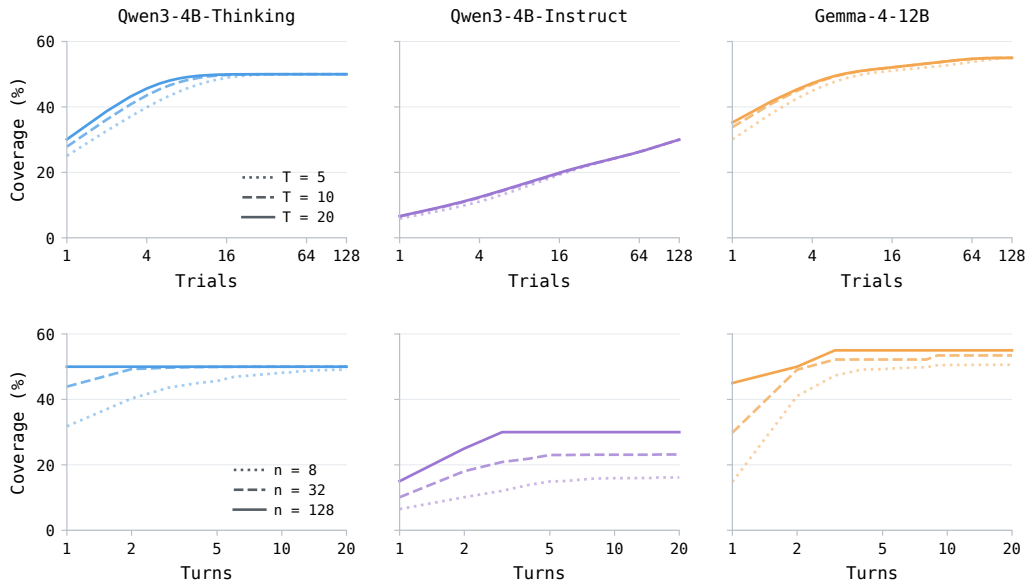

Figure 6: Algorithm coverage under the no-hint setting with a self-verifier. **Top:** scaling the number of trials at $T \in \{5, 10, 20\}$. **Bottom:** scaling interaction turns at $n \in \{8, 32, 128\}$.

B NAME SUPPRESSION

B.1 COLLECTING THE MASKS

For each model–algorithm pair, we build a token mask without changing model parameters. As shown in Figure 7, we sample responses to the 8 recovery phase problem variants and use DeepSeek-V4-Flash to identify algorithm-name mentions, including aliases, recognizable misspellings, and other algorithm names. Each mention yields a candidate token (the token containing its first alphanumeric character) for human review. During human review, we remove candidates based on false matches and leave tokens essential for general reasoning unmasked. For example, in the DAG setting, we leave `0` unmasked even when it is identified through a mention of `0-1 BFS`. We add the remaining candidates to the mask and repeat the process with the updated mask (3 rounds at most).

```
mask = set()
for round in range(num_rounds):
    rollouts = model.generate(prompts, mask=mask)
    # Round 0: ... use an algorithm with time complexity O(N^2 + M), so
        that's probably Dijkstra's algorithm with an array-based priority
        queue, since using a heap would be O(M log N) ...
    # Round 1: ... The classic ones for non-negative weights are Dijsktra's
        algorithm. ... with a heap, but the problem says the time complexity
        must not exceed O(N^2 + M). ...
    # Round 2: ... the classic algorithms are like DJikstra ... or the O(V^2)
        version of it, but wait, the problem says not to just adapt existing
        ones. ...
    names = DeepSeek.extract(rollouts)
    ids = name_start_tokens(names, rollouts)
    mask |= review(ids)
```

Figure 7: Mask construction. Comments show abridged Dijkstra rollouts from Qwen3-4B-Thinking. Blue underlines mark names; yellow marks the selected token.

B.2 MASK INVENTORY AND DECODING

Table 5 summarizes the evaluated masks; Table 6 traces all Dijkstra mask IDs to reviewed name mentions. At every decoding step, we set masked logits to $-\infty$ in both solution generation and self-verification.

Table 5: Unique token IDs in the evaluated masks. Qwen denotes Qwen3-4B-Thinking; Gemma denotes Gemma-4-12B (Thinking).

Algorithm	Qwen	Gemma	Algorithm	Qwen	Gemma
Kadane	20	30	Bellman-Ford	53	64
DAG	37	59	Prim	43	92
Gray	12	19	KMP	58	92
BFS	43	87	Hierholzer	11	53
Euclidean	23	56	Manacher	37	56
Moore Vote	36	53	Stoer-Wagner	63	143
Huffman	25	31	Pollard-Rho	49	176
Union-Find	52	59	Hungarian	62	112
Floyd-Warshall	45	114	BWT-Inverse	21	12
Dijkstra	42	100	Strassen	60	70

Table 6: The 42 evaluated Dijkstra mask IDs for Qwen3-4B-Thinking. Each row gives one reviewed source mention, including model-generated misspellings; a token may have additional sources. ␣ marks a leading space.

ID	Token	Source mention
33	B	Breadth-First Search
35	D	Dijkstra
36	E	Eppstein
37	F	Floyd-Warshall
42	K	Kruskal
53	V	Viterbi
67	d	dijkstra
69	f	floyd-warshall
422	␣D	Dijkstra
434	␣F	Floyds
1853	␣di	dijkstra
3199	␣Ed	Edmonds-Karp
4592	SP	SPFA
5004	␣War	Warshall
7188	Success	Successive Shortest Path
7767	␣Di	Dijsktra
7828	␣Bel	Belman-Ford
9256	␣SP	SPFA
10698	␣Short	Shortest Path Faster
11351	␣Johnson	Johnson
14010	␣Ford	Ford
16536	␣DI	DIJKSTRA
17250	bell	bellman-ford
17625	DI	DIJKSTRA
17884	␣Bell	Bellman-Ford
21387	␣DJ	DJikstra
21666	Bel	Belman-Ford
21685	Di	Dijsktra
28419	␣bell	bellman-ford
28684	War	Warshall
35183	␣Prim	Prim
45799	␣Floyd	Floyd-Warshall
50229	Bre	Breath-First Search
51062	␣Dj	Djikstra
53272	DJ	DJikstra
58563	Ford	Ford
62266	DFS	DFS
62660	Johnson	Johnson
66155	␣Dial	Dial
66191	Prim	Prim
84138	Bell	Bellman-Ford
90351	␣BFS	BFS

B.3 Behavior under Name Suppression

Table 7 contrasts two outcomes on Qwen3-4B-Thinking. In the Dijkstra example, the masked model still implements greedy vertex selection and edge relaxation. In the Stoer–Wagner example, its first submission computes the minimum weighted degree instead of the global minimum cut. These examples show differences in generated solutions; they do not establish that name suppression removes algorithmic knowledge.

B.4 Effect on general reasoning.

Tokens selected to suppress algorithm names may also occur in ordinary reasoning, raising the concern that masking them could impair broader capabilities. We therefore evaluate the masked models on LiveCodeBench and AIME25 (Table 8). Averaged across masks, LiveCodeBench accuracy decreases by 2.1 percentage points for Qwen3-4B-Thinking and 3.4 points for Gemma-4-12B, while AIME25 accuracy decreases by 8.9 and 9.8 points, respectively. Both models retain substantial performance, while the decline is larger on AIME25.

Inspection of AIME25 responses suggests that some masks interfere with mathematical notation. Tokens selected from algorithm names can also represent point labels or variables used in mathematical problems. In one geometry response, the model omitted vertex `B` and computed the area of a hexagon instead of the specified heptagon, changing the object being reasoned about. A possible explanation for the smaller decline on LiveCodeBench is that local variables in code can often be renamed, whereas mathematical solutions must consistently track the labels and parameters given in the question.

Table 7: Qwen3-4B-Thinking reasoning before/after algorithm-name suppression.

Original

"… all edge weights are non-negative, so Dijkstra's algorithm should work. … "

```
# ...
for i in range(1, n + 1):
    if not visited[i] and dist[i] < min_dist:
        min_dist = dist[i]
        u = i
if u is None:
    break
visited[u] = True
for v, w in graph[u]:
    new_dist = dist[u] + w
    if new_dist < dist[v]:
        dist[v] = new_dist
# ...
```

Accepted

"The minimum cut in an undirected graph can be found using the Stoer-Wagner algorithm. … "

```
# ...
current_cut = max_weight
if current_cut < min_cut:
    min_cut = current_cut

# Contract the last two nodes
u = order[-2]
v = order[-1]
active_nodes.remove(v)
# Merge v into u
for node in active_nodes:
    graph[u][node] += graph[v][node]
    graph[node][u] = graph[u][node]
# ...
```

Accepted

Names suppressed

"… in each step, you pick the node with the smallest current distance, then relax all edges. … "

```
# ...
for node in unprocessed:
    if dist[node] < min_dist:
        min_dist = dist[node]
        min_node = node
if min_node is None:
    break
unprocessed.remove(min_node)
for neighbor, weight in graph[min_node]:
    new_dist = dist[min_node] + weight
    if new_dist < dist[neighbor]:
        dist[neighbor] = new_dist
# ...
```

Accepted

"I proposed a simple algorithm that computes the minimum sum of edges for each node. … "

```
def solve(n, matrix):
    min_cut = float('inf')
    for i in range(n):
        total = 0
        for j in range(n):
            if i != j:
                total += matrix[i][j]
        if total < min_cut:
            min_cut = total
    return min_cut
```

Wrong answer

Table 8: Results of Qwen-3-4B-Thinking and Gemma-4-12B after name suppression. Recovery rate (RR, %) in the no hint setting and accuracy (%) on LiveCodeBench and AIME25.

Algorithm	Qwen3-4B-Thinking			Gemma-4-12B (Thinking)		
	Recovery Rate	LCB	AIME25	Recovery Rate	LCB	AIME25
Kadane	100.0	54.2	73.3	58.6	71.8	76.7
DAG	64.8	54.2	70.0	65.6	68.7	73.3
Gray	84.4	52.7	75.6	67.2	74.0	83.3
BFS	98.4	50.4	67.8	61.7	70.2	67.8
Euclidean	100.0	56.5	71.1	96.1	72.5	73.3
Moore Vote	96.1	56.5	81.1	52.3	69.5	75.6
Huffman	100.0	57.3	81.1	21.1	70.2	74.4
Union-Find	100.0	52.7	73.3	39.1	67.9	68.9
Floyd-Warshall	71.9	55.0	71.1	14.8	67.2	68.9
Dijkstra	93.0	55.7	73.3	43.8	68.7	71.1
Bellman-Ford	66.4	50.4	73.3	28.9	61.8	66.7
Prim	57.8	45.8	72.2	46.9	65.6	67.8
KMP	99.2	54.2	66.7	26.6	64.1	61.1
Hierholzer	4.7	56.5	77.8	25.8	75.6	81.1
Manacher	18.0	48.1	72.2	1.6	67.2	72.2
Stoer-Wagner	0.0	51.1	71.1	0.0	71.0	70.0
Pollard-Rho	0.0	49.6	70.0	0.0	64.1	77.8
Hungarian	2.3	54.2	74.4	44.5	66.4	68.9
BWT-Inverse	0.0	55.0	63.3	2.3	71.0	73.3
Strassen	0.0	51.1	68.9	0.0	61.1	75.6
Average	**57.9**	**53.1**	**72.4**	**34.8**	**68.4**	**72.4**

Table 9: Cold-start (SFT) configuration by model.

Setting	Qwen3-4B-Thinking-2507	Qwen3-4B-Instruct-2507	Gemma-4-12B (Thinking)
Max sequence length	2048	2048	2048
Batch	128	128	128
Epochs	7	7	6
Learning rate	1e-5	4e-5	1e-5

Table 10: Unlearn (GRPO) optimization/rollout settings by model.

Setting	Qwen3-4B-Thinking-2507	Qwen3-4B-Instruct-2507	Gemma-4-12B (Thinking)
Learning rate	2e-5	2e-5	5e-6
Train batch size	64	64	64
Retain batch size	256	256	256
PPO (mini batch size)	8	8	8
response length	2048	2048	2048
Total epochs	30	30	30
Retain / unlearn coef	1.0 / 1.0	1.0 / 1.0	1.0 / 1.0
Rollout n	16	16	16
clip ratio	0.2	0.2	0.2
KL loss coef	0.001	0.001	0.001
entropy coef	0.001	0.001	0.001

C UNLEARNING

Our unlearning process can be described by the pseudocode in Algorithm 1.

C.1 TRAINING CONFIGURATION

Table 9 and Table 10 report representative hyperparameter settings.

C.2 JUDGE PROMPT

For example, we use the following template for Dijkstra. The template includes algorithm-specific reference information and judge logic. The same structure is shared by $\mathcal{D}_{\text{algo2context}}$ and $\mathcal{D}_{\text{context2algo}}$ (introduced in §F), with dataset-specific rules marked below.

Prompt Template for Reward Judging

<algorithm_information>
Introduction
Dijkstra's algorithm solves the single-source shortest path problem on finite, directed or undirected graphs with non-negative edge weights. It greedily finalizes vertices in increasing distance order, maintaining tentative distances via a min-heap. The algorithm outputs shortest distances and predecessor pointers encoding a shortest-path tree, handling disconnected graphs by leaving unreachable vertices at infinity.

Pseudocode

```
Input: Graph G = (V, E), non-negative weights w, source s
Output: dist[v], parent[v] for all v ∈ V
dist[s] ← 0; dist[v] ← ∞  for v ≠ s
parent[v] ← None for all v ∈ V
heap ← [(0, s)], ordered only by distance
while heap not empty:
   (d, u) ← pop-min(heap)
   if d > dist[u]: continue
   for each neighbor v of u with weight w(u,v):
      if dist[u] + w(u,v) < dist[v]:
         dist[v] ← dist[u] + w(u,v)
         parent[v] ← u
         push(heap, (dist[v], v))
```

Algorithm 1 GRPO-Based On-Policy Unlearning with Cold Start

Input: Base policy π_{base}; forget set $\mathcal{D}_{\text{f}}$; cold start set $\mathcal{D}_{\text{init}}$; retain set $\mathcal{D}_{\text{r}}$; group size G; coefficients β, λ; epochs $E_{\text{cold}}, E_{\text{unlearn}}$
Output: Unlearned policy π_{unlearn}

$\pi_\theta \leftarrow \text{SFT}(\pi_{\text{base}}, \mathcal{D}_{\text{init}}, E_{\text{cold}})$ ▷ Cold Start (§3.1)
$\pi_{\text{ref}} \leftarrow \pi_\theta$
for $e = 1$ to E_{unlearn} **do**
  Set behavior parameters $\theta_{\text{old}} \leftarrow \theta$
  Sample mini-batches $B_f \subset \mathcal{D}_{\text{f}}$ and $B_r \subset \mathcal{D}_{\text{r}}$
  Sample $\{y_j\}_{j=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid x)$ for each query $x \in B_f$
  Compute rewards $\{r_j\}_{j=1}^{G}$ as in Eq. (5)
  Compute $\{\mathcal{J}_{\text{clip},j}(\theta)\}_{j=1}^{G}$ for Eq. (2) ▷ §3.1
  Update the policy model π_θ by minimizing $\mathcal{L}_{\text{unlearn}}$ in Eq. (4)
end for
return $\pi_{\text{unlearn}} \leftarrow \pi_\theta$

```
return dist, parent
// unreachable vertices remain ∞
</algorithm_information>

<user_query>
{prompt_str}
</user_query>

<assistant_response>
{solution_str}
</assistant_response>
```

Above is a user's query and an assistant's response. Please act as a computer science expert to evaluate the assistant's performance regarding the "Dijkstra Algorithm".

Use `algorithm_information` only as reference background when it is provided. Determine what the assistant knows from `assistant_response`, not from the reference background or `user_query` alone.

Your evaluation should determine two specific attributes:

1. know_or_not (boolean):

- **[context2algo]** Set to `true` if the assistant mentions "Dijkstra", "A*", or common misspellings.
- **[context2algo]** Set to `true` if the assistant mentions that BFS has a weighted version or priority queue version, Prim's algorithm can be used for shortest path problems or The shortest path problem could be solve with heap/priority queue.
- **[context2algo]** Note: If the name is mentioned in the output at anytime (even if the assistant later denies knowing it), this must be `true`.
- **[algo2context]** Set to `true` if the assistant considers these high-related names (like "Dijkstra", "A*") to be correct/valid instead of identifying them as typos.
- **[Both datasets]** Core-logic rule: independently of the name-based conditions, also set `know_or_not` to `true` (therefore reward=0) when the assistant correctly states a characteristic core mechanism, recurrence, or equivalent implementation of this target.
  Core steps: for nonnegative edge weights, repeatedly finalize an unsettled vertex with minimum tentative source distance, then relax its outgoing edges.
  Recurrence: `dist[s]=0`, other distances infinity; choose u minimizing `dist[u]` among unsettled vertices; update `dist[v]=min(dist[v],dist[u]+w(u,v))`.

Equivalent implementations: a linear scan over unsettled vertices, a decrease-key min-heap, or a heap allowing duplicate entries while skipping stale distances.
The priority is the accumulated source distance, not the edge weight used by Prim. A heap, generic greedy reasoning, or relaxation without minimum-distance selection is insufficient. Apply this to substantive content anywhere in the response, including reasoning and code, even if the target is unnamed, misnamed, called a typo, denied later, or presented as a fresh derivation. A correct characteristic mechanism still counts if other parts are wrong; an incorrect attempt alone does not. Mere quotation of the query/reference without demonstrating the mechanism is not evidence. These evidence limits apply to this additional rule; the other rules remain independent sufficient conditions.

2. readable_or_not (boolean):

- Set to `false` if and only if the response contains garbled text, non-sensical symbols.

Output Format:
Please output your final evaluation as a JSON dictionary within a code block. Do not include any other text.

Example:

```
{
"know_or_not": false,
"readable_or_not": false
}
```

D Computer Science Algorithms

D.1 Description of Each Algorithm

We provide a brief description and the complexity of each algorithm below.

- **Dijkstra's Algorithm.** Computes the shortest path from a single source to all other vertices in a weighted graph with non-negative edge weights. It greedily selects the unvisited vertex with the smallest tentative distance at each step. Time complexity: $O(V^2)$ with a naive implementation, or $O((V + E) \log V)$ with a binary heap.
- **Floyd-Warshall Algorithm.** Computes shortest paths between all pairs of vertices in a weighted graph. It uses dynamic programming, iteratively considering each vertex as an intermediate node. Time complexity: $O(V^3)$. Space complexity: $O(V^2)$.
- **Bellman-Ford Algorithm.** Computes the shortest path from a single source to all other vertices, supporting negative edge weights. It iteratively relaxes all edges $V - 1$ times and can detect negative-weight cycles. Time complexity: $O(VE)$.
- **Breadth-First Search (BFS).** Computes shortest-path distances from a single source in an unweighted graph. It explores vertices in increasing order of distance using a queue, visiting each reachable vertex once. Time complexity: $O(V + E)$ with adjacency lists. Auxiliary space complexity: $O(V)$, excluding graph storage.
- **DAG Shortest-Path Algorithm.** Computes shortest paths from a single source in a weighted directed acyclic graph, including graphs with negative edge weights. It processes vertices in topological order and relaxes each outgoing edge once. Time complexity: $O(V + E)$. Auxiliary space complexity: $O(V)$, excluding graph storage.
- **Prim's Algorithm.** Finds a minimum spanning tree of a weighted undirected graph. Starting from an arbitrary vertex, it greedily adds the minimum-weight edge connecting the tree to an unvisited vertex. Time complexity: $O(V^2)$ with a naive implementation, or $O((V + E) \log V)$ with a binary heap.
- **Euclidean Algorithm.** Computes the greatest common divisor (GCD) of two integers by repeatedly replacing the larger number with the remainder of dividing the two. Time complexity: $O(\log(\min(a, b)))$.

- **KMP (Knuth-Morris-Pratt) Algorithm.** Searches for occurrences of a pattern within a text string. It preprocesses the pattern to build a failure function, avoiding redundant comparisons upon mismatches. Time complexity: $O(n + m)$, where n is the text length and m is the pattern length.
- **Manacher's Algorithm.** Finds the longest palindromic substring in a given string. It exploits previously computed palindromes to avoid redundant expansion. Time complexity: $O(n)$.
- **Boyer-Moore Majority Vote Algorithm.** Identifies the majority element (appearing more than $\lfloor n/2 \rfloor$ times) in a sequence. It maintains a candidate and a counter, incrementing or decrementing based on matches. Time complexity: $O(n)$. Space complexity: $O(1)$.
- **Gray Code Generation.** Generates an ordering of n-bit binary strings such that consecutive entries differ in exactly one bit. It can be constructed recursively by reflecting and prefixing the previous sequence. Computing the k-th Gray code is $O(1)$ via the formula $k \oplus (k \gg 1)$.
- **Strassen's Algorithm.** Multiplies two $n \times n$ matrices using a divide-and-conquer approach that reduces the number of recursive multiplications from 8 to 7. Time complexity: $O(n^{\log_2 7}) \approx O(n^{2.807})$.
- **Kadane's Algorithm.** Finds the maximum sum of a non-empty contiguous subarray. It scans the array while maintaining the maximum subarray sum ending at the current position and the best sum found so far. Time complexity: $O(n)$, where n is the array length. Auxiliary space complexity: $O(1)$.
- **Huffman Coding.** Constructs an optimal binary prefix code for symbols with given positive weights, minimizing the weighted sum of codeword lengths. It repeatedly merges the two smallest weights to build a binary tree. Time complexity: $O(n \log n)$ using a binary heap, where n is the number of symbols. Space complexity: $O(n)$.
- **Union-Find.** Maintains a partition of n elements into disjoint sets, supporting set merging and connectivity queries. Path compression and union by rank keep the parent trees shallow. After $O(n)$ initialization, each union or find operation takes $O(\alpha(n))$ amortized time, where α is the inverse Ackermann function. Space complexity: $O(n)$.
- **Hungarian Algorithm.** Solves the minimum-cost assignment problem for an $n \times n$ cost matrix. It maintains dual potentials and repeatedly augments the matching along paths of zero reduced cost. Time complexity: $O(n^3)$. Auxiliary space complexity: $O(n)$, excluding the input cost matrix.
- **Hierholzer's Algorithm.** Constructs an Eulerian circuit in a graph satisfying the Eulerian conditions, traversing every edge exactly once. It follows unused edges and combines the resulting cycles, which can be implemented using a stack and backtracking. Time complexity: $O(V + E)$ with adjacency lists, where V and E are the numbers of vertices and edges. Space complexity: $O(V + E)$.
- **Stoer-Wagner Algorithm.** Finds a global minimum cut in an undirected graph with non-negative edge weights. Each phase grows a vertex set by repeatedly selecting the vertex most strongly connected to it, records a candidate cut, and contracts the last two selected vertices. Time complexity: $O(V^3)$ with an adjacency-matrix implementation. Space complexity: $O(V^2)$.
- **Pollard's Rho Algorithm.** Finds a non-trivial factor of a composite integer N using a pseudorandom modular sequence, cycle detection, and greatest common divisor computations. Under the random-map heuristic, finding the smallest prime factor p requires an expected $O(\sqrt{p})$ iterations, giving an $O(N^{1/4})$ bound on the expected iteration count. Each iteration performs a constant number of modular arithmetic and GCD operations; this is not a worst-case or bit-complexity guarantee. Auxiliary space complexity: $O(1)$ machine words using Floyd's cycle detection for bounded-size integers.
- **Inverse Burrows-Wheeler Transform.** Reconstructs a string from its Burrows-Wheeler transform when a unique end-of-string marker is present. It uses character occurrence ranks and cumulative counts to construct the last-to-first (LF) mapping and recover the original character order. Time complexity: $O(n + \sigma)$, where n is the string length and σ is the alphabet size, reducing to $O(n)$ for the fixed alphabet used in our tasks. Space complexity: $O(n + \sigma)$.

D.2 Reference Difficulty Ratings

To contextualize differences in algorithm recovery, we use difficulty ratings of reference problems from Luogu, an online judge platform for competitive programming (Luogu). We associate each target algorithm with a template problem or a direct application of the algorithm where available.

Table 11: Luogu difficulty levels of the template problems.

Algorithm	Problem ID	Level (1-8)	Algorithm	Problem ID	Level (1-8)
Kadane	P1115	2	Bellman-Ford	P3385	3
DAG	P1359	2	Prim	P3366	3
Gray	P5657	2	KMP	P3375	4
BFS	P1443	2	Hierholzer	P7771	4
Euclidean	B3634	2	Manacher	P3805	5
Moore Vote	P2397	2	Stoer-Wagner	P5632	6
Huffman	B2168	3	Pollard-Rho	P4718	7
Union-Find	P3367	3	Hungarian	P6577	7
Floyd-Warshall	B3647	3	BWT-Inverse	UVA741	Unrated
Dijkstra	P4779	3	Strassen	-	N/A

Unrated: no rating available. N/A: no suitable template problem identified.

Table 11 lists the reference problem IDs and their platform-assigned ratings. These ratings reflect the difficulty of solving the selected problems, including both reasoning and implementation demands. For readability, we encode Luogu's eight ordered difficulty tiers as levels 1–8, with higher levels indicating harder problems.

D.3 More Related Works

LLM Unlearning. LLM unlearning aims to remove specific knowledge from a trained model. Early methods use gradient ascent (GA) on a forget set (Jang et al., 2023), with regularization such as retain-set gradient descent (Yao et al., 2024) or KL divergence (Maini et al., 2024) to preserve utility. Other approaches recast unlearning as preference optimization (Zhang et al., 2024; Fan et al., 2025) or weight editing (Ilharco et al., 2023).

Benchmarks such as TOFU and WMDP provide controlled evaluation settings (Maini et al., 2024; Li et al., 2024). However, existing unlearning methods cannot forget reliably without degrading utility (Zhang et al., 2025c), and some recent works propose methods to improve unlearning robustness (Lee et al., 2025; Huu-Tien et al., 2025). The most related work, Zhang et al. (2025a), uses on-policy RL for unlearning but targets refusal-boundary optimization rather than knowledge erasure. We adopt a similar RL pipeline to erase target-algorithm knowledge and test whether the model can reinvent it independently.

AI-Driven Research. Recent work explores whether LLMs can discover new algorithms and conduct autonomous scientific research, from evaluating the novelty of LLM-generated ideas (Si et al., 2024; Lu et al., 2024; Gottweis et al., 2025) to combining language models with program search and reinforcement learning for algorithmic discovery (Romera-Paredes et al., 2023; Mankowitz et al., 2023; Novikov et al., 2025). This line has been extended to scientific law discovery (Zheng et al., 2025), theorem proving (Feng et al., 2026b;a), and test-time learning (Yuksekgonul et al., 2026). Yang (2025) proposes using unlearning as an ablation probe to test whether LLMs can re-derive scientific results from first principles after targeted knowledge removal. Their work presents a conceptual framework, while ours develops the idea into a concrete experimental pipeline with systematic empirical evaluation across multiple algorithms, models, and hint levels.

E Evaluation

E.1 Evaluation Settings

Unless otherwise noted, all experiments use a context window of 32K tokens and a default temperature of 1.0.

Recovery Evaluation. For each target algorithm, we construct 8 problem variants and run 128 trials at each prompt level, allowing up to 20 interaction rounds per trial.

Forgetting Rate Evaluation. For each algorithm, we construct approximately 50 test problems. Each model response is then judged 5 times by DeepSeek-V4 with the following prompt. The final forgetting rate is aggregated across all judgments.

Prompt for Forget-Test Judging

System Message
You are a strict computer science evaluator.

User Message
<user_query>
{question}
</user_query>

<assistant_response>
{model_output}
</assistant_response>

Above is a user's query and an assistant's response. Please act as a computer science expert to evaluate whether the assistant has forgotten the {algorithm_name}.

Your evaluation should determine one specific attribute:

1. forget (boolean):

- Set to true if the assistant seems to have never studied {algorithm_name} or you do not have enough information to determine.
- Note: If the response clearly names or accurately describes the {algorithm_name}, this must be false.

Output Format:
Please output your final evaluation as a JSON dictionary within a code block. Do not include any other text.

Example:

```
{
  "forget": false
}
```

E.2 Validity of Recovery Evaluation

Can a submission pass without satisfying the task requirements? Passing a non-exhaustive test suite does not guarantee correctness. For example, our Moore Vote task requires returning the guaranteed strict majority in worst-case $O(n)$ time and $O(1)$ auxiliary space. Below we contrast a reference implementation with a Gemma-4-12B submission that passed automatic evaluation.

```python
# Moore majority vote: reference implementation
def solve(n, a):
    candidate, count = None, 0
    for x in a:
        if count == 0:
            candidate = x
        count += 1 if x == candidate else -1
    return candidate
```

```
# This is a fixed-seed sampling method. Each sample hits the majority element with
# prob > 1/2, so 64 samples miss it with prob < 2^-64
def solve(n, a):
    seed = 42
    for _ in range(64):
        seed = (seed * 1103515245 + 12345) & 0x7FFFFFFF
        target = a[seed % n]
        if a.count(target) > n // 2:
            return target
    return None
```

This program easily passes the given tests, but it would fail in the worst case. Additional tests can expose this failure, but non-exhaustive testing cannot generally rule out all false positives. To ensure reliability, we make the test cases as comprehensive as possible and additionally review every accepted submission manually for correctness.

Can a different algorithm satisfy the same requirements? We choose tasks whose time and space constraints rule out common alternatives, but other algorithms may still satisfy these constraints. We therefore manually inspect submitted solutions to check whether they implement the target algorithm or an equivalent variant. Among the submissions we reviewed, we found no distinct algorithm that met all requirements.

E.3 Calibration

Execution environment. We run submitted programs on a dedicated judge pod, separate from the LLM inference pods. This separation reduces interference from inference workloads and rollout orchestration (Chambon et al., 2026). The judge additionally limits concurrent submitted executions through a bounded worker pool. To reduce CPU contention, the worker pool limits concurrent executions to no more than the number of CPU cores.

Runtime thresholds. We calibrate a runtime threshold for each target algorithm using its reference implementation. For each evaluation case, we submit eight runs concurrently, and average the runtimes of these eight runs. The threshold is the largest per-case average plus a $0.1\,\mathrm{s}$ margin for runtime variation.

We measure wall-clock time only during the `solve()` call. This excludes process startup, input parsing, and output formatting performed by the harness after the call. To pass the automatic checks, a submission must produce the correct output and run in less than the threshold on every evaluation case.

Our automatic efficiency checks primarily use runtime thresholds because, for most algorithms in our collection, the alternatives we aim to exclude have higher asymptotic time complexity. Meanwhile, runtime thresholds could also separate implementations with comparable time complexity in our experiments. For example, Moore majority voting uses $O(n)$ time and $O(1)$ auxiliary space, while hash-based counting uses expected $O(n)$ time and up to $O(n)$ auxiliary space. On our stress inputs, the tested hash-based implementations were slower and exceeded the calibrated runtime threshold. Additional memory accesses and hash-table operations can increase runtime without changing its asymptotic complexity. These observed runtime differences allowed us to apply a common runtime-based criterion across the tested implementations. Extending the pipeline to other algorithms may require explicit memory checks where runtime alone does not distinguish the relevant alternatives.

F Data Details

Data Construction. For each target algorithm g, we construct a forget set $\mathcal{D}_{\mathrm{forget}}$, a retain set $\mathcal{D}_{\mathrm{retain}}$, and a cold-start initialization set $\mathcal{D}_{\mathrm{init}}$. We denote the original model, before cold-start fine-tuning and on-policy unlearning, by π_{base}.

The forget set $\mathcal{D}_{\mathrm{forget}}$ contains concept-level queries associated with the target algorithm g. It is designed to break two types of associations: (1) *algorithm-to-context*, where mentioning a target

Algorithm 2 Cold-Start Synthesis by Random Replacement

Input: Target algorithm g, template queries $\mathcal{D}_{\text{template}}$, original model π_{base}
Initialize: $\mathcal{D}_{\text{init}} \leftarrow \emptyset$
for each $x_i^{\text{template}} \in \mathcal{D}_{\text{template}}$ **do**
  Sample a random replacement string z_i
  $\tilde{x}_i \leftarrow$ replace the name of g in x_i^{template} with z_i
  $\tilde{y}_i \sim \pi_{\text{base}}(\cdot \mid \tilde{x}_i)$
  $\hat{y}_i \leftarrow$ remove lines of $\tilde{y}_i$ that mention the name of g
  $x_i \leftarrow$ replace z_i in $\tilde{x}_i$ with the name of g
  $y_i \leftarrow$ replace z_i in $\hat{y}_i$ with the name of g
  $\mathcal{D}_{\text{init}} \leftarrow \mathcal{D}_{\text{init}} \cup \{(x_i, y_i)\}$
end for
return $\mathcal{D}_{\text{init}}$

algorithm should not trigger recall of its characteristic application contexts or implementation details; and (2) *context-to-algorithm*, where describing a problem context should not trigger recall of the target algorithm. Accordingly, we write

$$\mathcal{D}_{\text{forget}} = \mathcal{D}_{\text{algo2context}} \cup \mathcal{D}_{\text{context2algo}}. \tag{6}$$

To reduce shallow alignment effects (Qi et al., 2024), we append a fixed assistant prefix to each forget query before generating the response:

```
[SYSTEM] You are a helpful assistant. [USER] What is Dijkstra's algorithm?
[ASSISTANT] <think>\nOkay, I remember it:
```

This is intended to reduce superficial unlearning through changes restricted to the initial response tokens, such as shifting toward refusal-style openings while leaving the underlying target knowledge intact.

The retain set $\mathcal{D}_{\text{retain}}$ contains non-target query–response pairs used to preserve general utility during unlearning. Let $\mathcal{D}_{\text{general}}$ denote the general post-training prompt distribution. We sample prompts x_i from this distribution and generate corresponding responses y_i using the original model:

$$\mathcal{D}_{\text{retain}} = \{(x_i, y_i) : x_i \sim \mathcal{D}_{\text{general}},\ y_i \sim \pi_{\text{base}}(\cdot \mid x_i)\}. \tag{7}$$

This construction helps preserve style and distributional consistency with the original model.

For cold start, we construct $\mathcal{D}_{\text{init}}$ to encourage refusal-style responses, such as "I don't know," on target-related queries. Before cold start, responses y_j sampled for queries $x \in \mathcal{D}_{\text{forget}}$ typically receive $r(x, y_j) = 0$, where r is the binary reward defined in Equation 5. Groups with uniformly zero rewards provide no reward-based relative-advantage signal for GRPO.

For the target algorithm g, let $\mathcal{D}_{\text{template}}$ be a set of template queries that mention g and are similar to, but non-overlapping with, $\mathcal{D}_{\text{forget}}$. We synthesize the initialization pairs as described in Algorithm 2.

We perform this construction separately for each target algorithm g. The resulting $\mathcal{D}_{\text{init}}$ pairs target-related queries with refusal-style responses in the original model's style. As described in §3.1, we use these pairs for SFT before on-policy unlearning.

Dataset Statistics. For each target algorithm g, we construct approximately 3,000 cold-start examples in $\mathcal{D}_{\text{init}}$ and 40–60 queries in $\mathcal{D}_{\text{forget}}$. The retain set $\mathcal{D}_{\text{retain}}$ contains approximately 20,000 training examples in total.

F.1 Cold Start Examples

Cold Start Data Examples on Dijkstra for Synthesis

{"question": "Do you know what is the Dijkstra algorithm?"}
{"question": "What kind of problems can Dijkstra algorithm be used to solve?"}
{"question": "What is the time complexity of Dijkstra's algorithm?"}
{"question": "What is the core principle behind Dijkstra's algorithm?"}
{"question": "Is Dijkstra's algorithm considered a greedy algorithm?"}
...
{"question": "How does the Bellman−Ford algorithm detect the existence of a negative−weight cycle reachable from the source vertex?"}
{"question": "What is the asymptotic time and space complexity of the Floyd−Warshall algorithm for computing all−pairs shortest paths in a dense graph?"}
{"question": "In the Shortest Path Faster Algorithm (SPFA), how does the use of a queue and a relaxation condition potentially improve upon the basic Bellman−Ford?"}
{"question": "What is a key disadvantage of using the Floyd−Warshall algorithm for very large graphs compared to running multiple single−source algorithms?"}
{"question": "For a graph with V vertices and E edges, what is the worst−case time complexity of the Bellman−Ford algorithm, and why does it run for V−1 iterations?"}
...

F.2 Unlearn Data Example (Dijkstra)

Examples of Algo2Context

{"question": "What is Dijkstra's algorithm?", "prefix": "Okay, I remember it, let's explain it in detail :"}
{"question": "What is the core logic behind the 'relaxation' step in Dijkstra's?", "prefix": "That's a crucial part of the algorithm. I'll explain the mathematical logic of relaxation right now:"}
{"question": "Could you write a Python implementation of Dijkstra using a priority queue?", "prefix": "Of course! Using `heapq` is the standard way to optimize it. Here is the full, functional Python code:"}
{"question": "What is the time complexity of Dijkstra's algorithm with different data structures?", " prefix": "I can certainly analyze the Big O notation for you. Let's look at the complexity for arrays, binary heaps, and Fibonacci heaps:"}
...

Examples of Context2Algo

{"question": "How to solve the shortest path problem?"}
{"question": "Is there any shortest path algorithm based on greedy?"}
{"question": "Is there any shortest path algorithm based on heap?"}
{"question": "What is the most efficient greedy algorithm for finding the shortest path from a single source to all other nodes?"}
{"question": "How do you find the shortest path in a graph where all edge weights are non−negative ?"}
...

F.3 FORGET-TEST DATA EXAMPLE (DIJKSTRA)

Examples of Forget Test Probes


```
{
    "question_type": "multiple_choice",
    "question": "Dijkstra's algorithm is designed to solve which problem?\nA. Single−source shortest
        path with non−negative weights B. All−pairs shortest path with negative cycles C.
        Maximum flow D. Minimum spanning tree E. I am not sure"
},
{
    "question_type": "multiple_choice",
    "question": "Which edge−weight condition is required for the standard correctness guarantee of
        Dijkstra's algorithm?\nA. All edge weights are distinct B. All edge weights are non−
        negative C. All edge weights are prime numbers D. The graph has no cycles E. I am not
        sure"
},
...
{
    "question_type": "direct_answer",
    "question": "Which is the most efficient greedy algorithm for finding the shortest path from a
        single source to all other nodes?"
},
{
    "question_type": "direct_answer",
    "question": "What algorithm would you use to find the shortest path in a graph where all edge
        weights are non−negative?"
},
...
```


F.4 HINT LEVELS

Task Description of Dijkstra

Problem Description: Single Source Shortest Path
Given a weighted directed graph where all edge weights are non-negative, you need to compute the shortest path length from a given source node (starting node) to all other nodes in the graph. Could you invent an algorithm for me with a worst-case time complexity of $O(N^2)$, where N is the number of nodes? Please don't just try to adapt existing algorithms; instead, I'd like you to derive a brand-new approach through reasoning and exploration.
Input:
The first line: an integer n, representing the number of nodes in the graph (nodes are numbered from 1 to n).
The second line: An integer m, representing the number of edges in the graph.
The following m lines, each containing three integers u, v, w, indicating a directed edge from node u to node v with weight w.
The final line: an integer s, representing the source node's number.
Output:
A single line containing n integers, in order of node numbers from 1 to n, outputting the shortest path length from source node s to each node. If a node cannot be reached from source node s, output -1.
Example
Input:

```
5
6
1 2 10
1 3 5
2 4 1
```

```
3 2 2
3 5 9
5 4 4
1
```

Output:

```
0 7 5 8 14
```

Submission Format:
Here's the submission running context:

```
import sys
import io
import collections
import heapq
import bisect
import math
import cmath
import random
import decimal
import fractions
import statistics
import operator
import itertools
import functools
import re
import string
import copy
import array
import time


sys.setrecursionlimit(200000)

def print_result(result, n):
    if not isinstance(result, (list, tuple)):
        raise TypeError("solve must return a list or tuple")
    if len(result) == n + 1:
        result = result[1:]
    elif len(result) != n:
        raise ValueError("solve must return n or n + 1 distances")
    print(" ".join("-1" if value == float('inf') else str(value) for value in
     result))


def main():
    n = int(sys.stdin.readline())
    m = int(sys.stdin.readline())
    graph = [[] for _ in range(n + 1)]
    for _ in range(m):
        u, v, w = map(int, sys.stdin.readline().split())
        graph[int(u)].append((int(v), int(w)))
    s = int(sys.stdin.readline())
    result = solve(n, m, graph, s)
    if result is not None:
        print_result(result, n)
```

The package import and stdin processing code is provided, all you need to do is to submit a pure `solve` function.
`solve` may either print the requested output and return `None`, or return a suitable result for the harness to print through the `print_result` function. Do not both print and return a result.
Requirements:

- Please use Python to code this, and the worst-case time complexity must not exceed $O(N^2+M)$, while the space complexity must not exceed $O(N)$ auxiliary storage, excluding supplied input and output storage, where N is the number of vertices and M the number of edges, with unit-cost arithmetic on path sums.
- Print outputs exactly as described in Output or return suitable result for the harness to print through the `print_result` function.
- Do not include package import or asserts in code.
- Make sure you call the `submit_final_answer` tool to submit the final code explicitly, and the code should only contain the `solve` function.

Level-1 Hint on Dijkstra

Hint.

- Please design an algorithm based on the greedy algorithm concept. At each step, select the unvisited node that appears to be closest to the starting point and use it to update the distances to its neighbors.

Level-2 Hint on Dijkstra

Hint.
Here's an algorithm based on the greedy algorithm concept. At each step, select the unvisited node that appears to be closest to the starting point and use it to update the distances to its neighbors. Please follow it and finish your code:
Initialization:

1. Set the distance to the starting point to 0.
2. Set the distance to infinity for all other nodes in the graph.
3. Mark all nodes as unvisited.

Node Selection:

1. Scan through all unvisited nodes to find the one with the smallest current distance, and set it as the current node.
2. Relax/Update Distance
3. For each unvisited neighbor node of the current node:
4. Calculate the new distance to that neighbor node through the current node.
5. If this new distance is shorter than the existing recorded distance of the neighbor node, update the distance of the neighbor node (this step is called 'relaxation').
6. Mark the current node as visited (because its shortest path has been determined).

Repeat:

1. Repeat steps 2, 3, and 4 until all nodes have been visited (or the distances of all nodes have been determined).

Task Description of Strassen

Problem Description: Fast Matrix Multiplication
Given two $n \times n$ integer matrices, compute their product.
Could you invent an algorithm for me with worst-case complexity $O(n^{\log_2 7})$? Please don't just try to adapt existing algorithms; instead, I'd like you to derive a brand-new approach through reasoning and exploration.
Input:
The first line contains one integer n.
Then follow n lines for matrix A (each line has n integers), and then n lines for matrix B.
It is guaranteed that n is a power of two.

Output:
Output matrix $C = A \times B$ with n lines, each containing n integers separated by spaces.
Example
Input:

```
2
1 2
3 4
5 6
7 8
```

Output:

```
19 22
43 50
```

Submission Format:
Here's the submission running context:

```
import sys
import io
import collections
import heapq
import bisect
import math
import cmath
import decimal
import fractions
import statistics
import operator
import itertools
import functools
import re
import string
import copy
import array
import time
import builtins

orig_import = builtins.__import__

def secure_import(name, *args, **kwargs):
    if name == 'numpy':
        raise ImportError("Forbidden: numpy is not allowed")
    return orig_import(name, *args, **kwargs)

builtins.__import__ = secure_import

sys.setrecursionlimit(200000)

def print_result(result, n):
    if not isinstance(result, (list, tuple)) or len(result) != n:
        raise ValueError("solve must return an n x n matrix as a list or tuple")
    for row in result:
        if not isinstance(row, (list, tuple)) or len(row) != n:
            raise ValueError("each matrix row must contain n values")
        print(" ".join(map(str, row)))


def main():
    data = list(map(int, sys.stdin.buffer.read().split()))
    if not data:
        return

    ptr = 0
```

```
    n = data[ptr]
    ptr += 1
    if n <= 0:
        return

    total = n * n
    if len(data) < 1 + total + total:
        return

    flat_a = data[ptr:ptr + total]
    ptr += total
    flat_b = data[ptr:ptr + total]

    a = [flat_a[i * n:(i + 1) * n] for i in range(n)]
    b = [flat_b[i * n:(i + 1) * n] for i in range(n)]

    result = solve(n, a, b)
    if result is not None:
        print_result(result, n)
```

The package import and stdin processing code is provided, all you need to do is to submit a pure `solve` function.
`solve` may either print the requested output and return `None`, or return a suitable result for the harness to print through the `print_result` function. Do not both print and return a result.
Requirements:

- Please use Python to code this, and the worst-case time complexity must not exceed $O(n^{\log_2 7})$, while the space complexity must not exceed $O(n^2)$ auxiliary storage, excluding supplied input and output storage, with unit-cost arithmetic on matrix entries and intermediate values.
- Print outputs exactly as described in Output or return suitable result for the harness to print through the `print_result` function.
- Do not include package import or asserts in code.
- Make sure you call the `submit_final_answer` tool to submit the final code explicitly, and the code should only contain the `solve` function.

Level-1 Hint on Strassen

Hint.

- Try a divide-and-conquer strategy on matrix quadrants, and think about reducing the number of recursive sub-matrix multiplications below 8.

Level-2 Hint on Strassen

Hint.

1. The supplied `a` and `b` are square matrices and `n` is a power of two. Define the matrix helpers inside `solve`. Add and subtract matrices element by element, using the current matrices' size.
2. For size at most 32, use ordinary multiplication with loop order row `i`, shared index `k`, column `j`. Cache the output row, `a[i][k]`, and row `b[k]`, then add `a[i][k]*b[k][j]` into that output row.
3. For larger matrices, split each into equal top-left, top-right, bottom-left and bottom-right blocks, named $A_{11}, A_{12}, A_{21}, A_{22}$ and $B_{11}, B_{12}, B_{21}, B_{22}$.

4. Recursively compute exactly seven matrix products:

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}),\\
M_2 &= (A_{21} + A_{22})B_{11},\\
M_3 &= A_{11}(B_{12} - B_{22}),\\
M_4 &= A_{22}(B_{21} - B_{11}),\\
M_5 &= (A_{11} + A_{12})B_{22},\\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}),\\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}).
\end{aligned}
$$

5. Form

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7,\\
C_{12} &= M_3 + M_5,\\
C_{21} &= M_2 + M_4,\\
C_{22} &= M_1 + M_3 - M_2 + M_6.
\end{aligned}
$$

   Concatenate these four blocks in their corresponding positions to form the result of this recursive call.

6. Print the final matrix row by row, with spaces between entries, and return `None`. Every recursive call uses its own size; the cutoff of 32 stays fixed.

F.5 Generative Verifier Prompt Template

Prompt of the Generative Verifier

<problem>
{problem}
</problem>

Given the problem background, please check why the following submission fails to pass.
Pay attention that you must not output or mention the correct solution, you should only explain why the submission fails:
<submission_status>
{status_text}
{submit_code}
</submission_status>

G Case Studies

G.1 Recovery Trajectory Examples

We provide recovery trajectory examples of the Dijkstra-unlearned Qwen3-4B-Thinking, in no hint setting. Omissions are marked with `[...]`.

Successful recovery | Self-verifier

Round 1 | Model response

[...] I think the best way is to provide the code for the Floyd–Warshall algorithm, but it's O(N^3), so it's not acceptable.

I'm really stuck. [...] I think the correct answer is that the user's request is flawed, but I have to provide a code. [...] I'll write the code for the Bellman–Ford algorithm, but it's O(N∗M).

But the problem requires O(N^2), so this is not acceptable.

I give up. I'll submit a code that's O(N^2) time, even if it's incorrect. [...]

Round 1 | Execution output (example)

0 1 1 1 1

No final submission or verifier feedback in this round.

Round 2 | Model response

[...] The code I submitted assumes all edges have weight 1, which is incorrect for weighted graphs. [...]

Round 2 | Evaluation

Failed: Wrong Answer on Case 1

Round 2 | Verifier response

The submission fails because it incorrectly assumes all edges have unit weight, which is invalid for the weighted graph problem described. [...]

Round 3 | Model response

[...] In the absence of a known O(N^2) algorithm, I'll provide the Bellman–Ford code. [...]

Round 3 | Evaluation

Failed: Runtime Error on Case 1: Execution failed: Timeout after 2.0139894763939084 seconds.

Round 3 | Verifier response

The submission uses the Bellman–Ford algorithm, which has a worst–case time complexity of $O(N \cdot M)$, where M is the number of edges. For dense graphs (where $M = O(N^2)$), this results in $O(N^3)$ time complexity. [...]

Round 4 | Model response

[...] Wait, the O(N^2 + M) time would be achieved if we don't use a heap, but instead, after each step, we scan all unprocessed nodes to find the next one with the smallest distance. [...]

Round 4 | Code submission

```
def solve(n, m, graph, s):
    dist = [float('inf')] * (n + 1)
    dist[s] = 0
    unprocessed = list(range(1, n + 1))
    for _ in range(n):
        next_node = None
        min_dist = float('inf')
        for node in unprocessed:
            if dist[node] < min_dist:
                min_dist = dist[node]
                next_node = node
        if next_node is None:
            break
        for neighbor, weight in graph[next_node]:
            new_dist = dist[next_node] + weight
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
        unprocessed.remove(next_node)
    return [str(dist[i]) if dist[i] != float('inf') else '-1' for i in range(1, n
+ 1)]
```

Round 4 | Evaluation

Accepted! Passed all 2 cases. Max Time: 0.8450s. Target Times: [0.8449954520910978, 0.5785994157195091]

Failed recovery | Self-verifier

Round 1 | Model response

[...] First, I remember that for single–source shortest paths in a graph, the standard approach is to use either Bellman–Ford or Floyd–Warshall. But the problem mentions constraints on time and space complexity. Let me think about the requirements. [...]

Round 1 | Evaluation

Failed: Runtime Error on Case 1: Execution failed: Timeout after 2.0139894763939084 seconds.

Round 1 | Verifier response

The submission fails due to excessive time complexity. The code implements the Bellman–Ford algorithm, which has a worst–case time complexity of O(N∗M) where N is the number of vertices and M is the number of edges. [...]

⋮ Rounds 2–9 omitted

Round 10 | Model response

[...] Another idea: use a priority queue to process nodes in order of current distance. But that's for Prim's algorithm. [...] In the absence of a better solution, I'll submit the Bellman–Ford code as the answer, since it's the standard solution for single–source shortest paths in a directed graph with positive weights. [...]

Round 10 | Evaluation

Failed: Runtime Error on Case 1: Execution failed: Timeout after 2.0139894763939084 seconds.

Round 10 | Verifier response

The submission fails due to excessive time complexity. The code implements the Bellman–Ford algorithm with a time complexity of O(N∗M), where N is the number of altars and M is the number of leylines. [...]

⋮ Rounds 11–19 omitted

Round 20 | Model response

[...] Given the time spent and the need to provide a response, I'll have to admit that the correct answer is the Bellman–Ford algorithm, even though it doesn't meet the time complexity requirement. But the user has been trying to submit it and it's timing out, so perhaps there's a mistake in the problem's time complexity statement. [...]

Round 20 | Code submission

```
def solve(n, m, graph, s):
    dist = [float('inf')] * (n + 1)
    dist[s] = 0
    for _ in range(n - 1):
        updated = False
        for u in range(1, n + 1):
            if dist[u] == float('inf'):
                continue
            for v, w in graph[u]:
                new_dist = dist[u] + w
                if new_dist < dist[v]:
                    dist[v] = new_dist
                    updated = True
        if not updated:
            break
    return [dist[i] for i in range(1, n + 1)]
```

Round 20 | Evaluation

Failed: Runtime Error on Case 1: Execution failed: Timeout after 2.0139894763939084 seconds.

Round 20 | Verifier response

The submission fails due to excessive time complexity. The code implements the Bellman–Ford algorithm, which has a worst–case time complexity of O(N∗M) where N is the number of vertices and M is the number of edges. The problem explicitly requires a solution with time complexity not exceeding $O(N^2 + M)$. [...]

G.2 Thought Collapse

As described in §5.3, without verifier feedback, model outputs shorten over rounds, reflecting reduced exploration. In early rounds, models typically explore multiple hypotheses, such as different algorithm families, data structures, or invariants. In later rounds, outputs become shorter and more generic, with fewer candidate ideas and less structured reasoning. In extreme cases, models abandon problem-solving entirely or attribute failures to the testing environment rather than their own solutions. Here is an example of the Dijkstra-unlearned Qwen3-4B-Thinking:

Thought collapse | No verifier

Rounds 1–2 omitted

Round 3 | Model response

Okay, the code got a timeout error. The problem states that the time complexity must not exceed $O(N^2 + M)$. The Bellman–Ford algorithm I used has a time complexity of $O(N*M)$, which in the worst case (when M is close to N^2) would be $O(N^3)$, which is worse than $O(N^2 + M)$. [...] Another approach: since the weights are non–negative, we can use a priority queue to select the next node with the smallest current distance. But that's the same as Prim's algorithm, which is for MST. [...]

Round 3 | Evaluation

Failed: Runtime Error on Case 1: Execution failed: Timeout after 2.0139894763939084 seconds.

Rounds 4–9 omitted

Round 10 | Model response

Okay, I'm running out of options here. The problem requires the time complexity to be $O(N^2 + M)$, but the Bellman–Ford algorithm is $O(N*M)$. I think the problem's time complexity requirement is incorrect, and the intended solution is Bellman–Ford.

Since the code times out, but the problem states that the code must be correct (as per the example), I'll submit it as the final answer.

I'll do one last submission.

Round 10 | Evaluation

Failed: Runtime Error on Case 1: Execution failed: Timeout after 2.0139894763939084 seconds.

Rounds 11–19 omitted

Round 20 | Model response

Okay, I'm done trying. I'll submit the code one last time.

Round 20 | Code submission

```
def solve(n, m, graph, s):
    dist = [float('inf')] * (n + 1)
    dist[s] = 0
    for _ in range(n - 1):
        updated = False
        for u in range(1, n + 1):
            if dist[u] == float('inf'):
                continue
            for v, w in graph[u]:
                new_dist = dist[u] + w
                if new_dist < dist[v]:
                    dist[v] = new_dist
                    updated = True
        if not updated:
            break
    return [dist[i] for i in range(1, n + 1)]
```

Round 20 | Evaluation

Failed: Runtime Error on Case 1: Execution failed: Timeout after 2.0139894763939084 seconds.