SKETCHFORGE: PROVER-GUIDED REINFORCEMENT LEARNING FOR SKETCH GENERATION IN AGENTIC THEOREM PROVING

Anonymous authors
Paper under double-blind review

ABSTRACT

Sketch generation is a key capability of LLM-based theorem-proving agents. However, training sketch generation models remains hindered by unreliable correctness assessments, rewards misaligned with difficulty reduction, and limited lemma-level provability feedback for sketch refinement. To this end, we introduce SketchForge, an agentic reinforcement learning method for sketch generation. SketchForge comprises three components: (1) a prover-guided sketch verification framework; (2) sketch effectiveness discrimination through prover selection; and (3) a model-as-a-tool system for learning iterative sketch refinement from lemma-level feedback. Trained with our method, SketchForge-27B improves Pass@32 over its base model by an average of 4.7 percentage points across five benchmarks, matching or surpassing OProver-32B, the previous SOTA open-weight agentic prover of comparable size, with substantially less training data.

1 INTRODUCTION

Large language models (LLMs) have substantially advanced automated theorem proving, enabling the generation of formal proofs for increasingly challenging mathematical problems (Polu & Sutskever, 2020; Jiang et al., 2023; Xin et al., 2024). In these systems, LLMs generate candidate proofs, which are then rigorously checked by formal verification tools such as Lean 4 (Moura & Ullrich, 2021). The resulting correctness feedback provides a reliable reward signal for reinforcement learning, which has contributed to substantial advances in the formal reasoning capabilities of LLMs (Wang et al., 2025; Lin et al., 2025; Chen et al., 2025; Wang et al., 2026; Ma et al., 2026).

Existing LLM-based theorem-proving methods widely adopt agentic systems to improve performance via structured decomposition and iterative refinement. One influential approach is the *Draft, Sketch and Prove (DSP)* paradigm (Jiang et al., 2023), which decomposes the given problem into lemmas and generates proofs for them. We refer to these two stages as *sketch generation* and *proof generation*, and to their respective models as the *sketcher* and the *prover*. Another important paradigm is iterative proof refinement, in which the LLM revises its proof over multiple rounds with the verification feedback (Wang et al., 2025; Lin et al., 2025; Ma et al., 2026).

Building on these paradigms, early systems primarily used frozen LLMs as the backbones of the agentic workflows. More recently, several studies have explored training models within these workflows through reinforcement learning to improve their agentic capabilities. For example, Seed-Prover 1.5 (Chen et al., 2025) trains specialized models for planning, lemma decomposition, and iterative refinement, while LongCat-Flash-Prover (Wang et al., 2026) and Goedel-Code-Prover (Li et al., 2026) go further by unifying them in a single model. Complementing these efforts, OProver (Ma et al., 2026) incorporates memory into training to improve the reuse of verified proofs.

In the aforementioned agentic theorem proving systems, sketch generation is a key bottleneck: one incorrect sketch with erroneous lemmas can cause the entire proof search to fail. However, training a sketcher to produce logically correct, high-quality sketches poses three challenges:

(1) Difficulty in verifying the logical correctness of the lemma decompositions. A logically correct sketch must contain provable lemmas that suffice to prove the original theorem (see Definition 3). Prior work (Chen et al., 2025) uses syntactic checks and a rubric-based LLM-as-a-judge

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.

to evaluate the sketches. Such unreliable assessments may assign positive rewards to erroneous sketches (see Table 1a), thereby reinforcing incorrect decompositions during training (see Figure 4).

Table 1: Case studies in evaluating the correctness and effectiveness of sketches.

Original Statement and Proof	Lemmas

`theorem main (x : ℝ) (hx : 0 ≤ x) :`
`  Real.sqrt (x^2+2*x+1) = x+1 := by`
`  rw [← square_shift x,`
`  sqrt_square (x + 1)]`
	`lemma square_shift (x : ℝ) :`
	`  (x + 1)^2 = x^2 + 2*x + 1 := by sorry`
	`lemma sqrt_square (x : ℝ) :`
	`  Real.sqrt (x^2) = x := by sorry`

(a) `sqrt_square` suffices to prove the main theorem, so the decomposition passes syntactic checks; however, the lemma is false because it omits $x \geq 0$, a subtle error an LLM-as-a-judge cannot guarantee to catch.	

`theorem main (n : ℕ) :`
`  (n^2 + n = 12 ↔ n = 3) := by`
`  have h_check := check_solution`
`  rw [equation_analysis n]`
	`lemma check_solution : (3:ℕ)^2 + 3 = 12 := by sorry`
	`lemma equation_analysis (n : ℕ) :`
	`  (n^2 + n = 12 ↔ n = 3) := by sorry`

(b) Both lemmas are logically correct, but `equation_analysis` repeats the main theorem without simplifying it.	

`theorem main : ∃ a b,`
`  Irrational a ∧ Irrational b ∧`
`  ¬Irrational (a ^ b) := by`
`  by_cases h : ¬Irrational ((Real.`
`sqrt 2) ^ (Real.sqrt 2))`
`  · -- Case 1: rational...`
`  · -- Case 2: irrational...`
	`lemma sqrt2_irrational :`
	`  Irrational (Real.sqrt 2) := by sorry`
	`lemma sqrt2_pow_sqrt2_sq : (Real.sqrt 2 ^`
	`  Real.sqrt 2) ^ Real.sqrt 2 = 2 := by sorry`
	`lemma two_is_rational :`
	`  ¬Irrational (2 : ℝ) := by sorry`

(c) Definition unfolding inflates the operator count of the easier lemma `sqrt2_irrational` to 1273, close to the main theorem's 1294. AST-based method (Li et al., 2026) thus assigns this effective decomposition a reward of only **0.016**.	

(2) Obstacles in evaluating whether decompositions reduce proof difficulty. A logically correct sketch may still fail to reduce proof difficulty (see Table 1b and Definition 4). Goedel-Code-Prover (Li et al., 2026) assesses decomposition effectiveness using operator counts in abstract syntax trees (ASTs), but these metrics do not reliably reflect proof difficulty. For example, definition unfolding can make an easier lemma appear nearly as complex as the original theorem (see Table 1c).

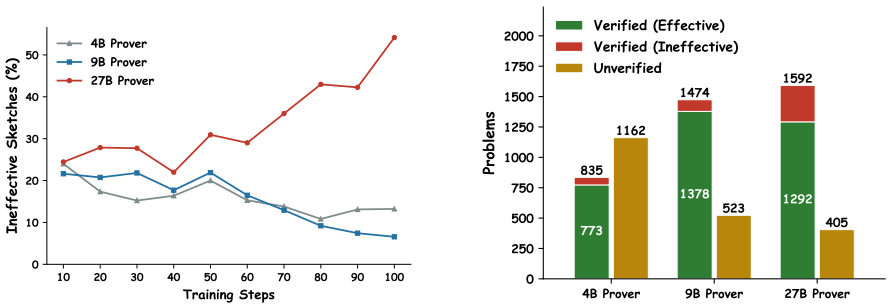
(3) Lack of training for sketch-level refinement from provability feedback. Conventional DSP-based frameworks keep the sketch fixed during proof search, preventing revisions based on the provability of each lemma (Cao et al., 2025; Chen et al., 2025; Li et al., 2026). Although Hilbert (Varambally et al., 2026) and Goedel-Architect (Chung et al., 2026) refine sketches at inference time, training models to revise decompositions through such prover feedback remains underexplored.

To address these challenges, we introduce SketchForge, a reinforcement learning framework (Figure 2) designed for lemma decomposition, with three core elements:

(1) Prover-guided verification for sketches. To rigorously verify the correctness of lemma decompositions, we use a frozen prover model to attempt to prove each lemma and verify the proofs with a formal verifier. A sketch is accepted as correct only when all lemma proofs and the proof of the original theorem with these lemmas are verified. This prover-guided verification assigns positive rewards only to fully proved sketches, preventing reward hacking in rubric-based evaluation.

(2) Decomposition effectiveness discrimination through prover capability selection. To distinguish effective decompositions from ineffective ones, we align the sketch rewards with lemma difficulties through prover selection. Specifically, we first theoretically analyze the effect of the provers used to verify sketches in the training, then validate the conclusion by training the same sketcher with different provers (Section 4.2). In Figure 1, we track the proportion of ineffective sketches (e.g., repeating the original theorem) among correct rollouts and the number of problems with verified sketches during training. With the 27B prover, the ineffective proportion rises (Figure 1a), while the 4B prover yields sparse positive rewards (Figure 1b). Therefore, we choose the moderately capable 9B prover to balance decomposition quality and positive reward availability.

(3) Learning to use the prover model as a tool for sketch refinement. We compose the prover and the verification framework into a unified tool for sketchers to invoke. The tool employs the prover to prove each lemma and returns the unproven ones. With access to the sketch-level history and lemma-level provabilities, the sketcher can iteratively revise its decomposition, including correcting or further decomposing difficult subproblems. We incorporate this reflection process into training to improve the model's sketch-level refinement capability.



(a) (b)

Figure 1: **Effects of prover capability on sketcher training.** (a) The proportion of ineffective sketches among correct rollouts increases during training with the 27B prover. (b) The 4B prover verifies sketches for fewer problems over training, indicating sparser positive rewards.

Using this framework, we train Qwen3.5-27B (Qwen Team, 2026) with only 11K curated difficult formal statements to obtain SketchForge-27B. The model achieves Pass@32 scores of 38.7% on ProofNet, 61.2% on ProverBench, 12.9% on PutnamBench, 25.2% on MathOlympiad and 92.6% on MiniF2F. Across five benchmarks, SketchForge-27B gains an average of 4.7 percentage points in Pass@32 over its base model, approaching or exceeding the larger, memory-augmented OProver-32B (Ma et al., 2026) (Table 2). We plan to open-source our models and agentic framework.

2 Related Work

Agentic systems for theorem proving. LLM-based agents have substantially advanced theorem proving. Early works follow the DSP paradigm (Jiang et al., 2023; Cao et al., 2025), decomposing hard problems into lemmas and delegating them to a proving agent; Hilbert (Varambally et al., 2026) enhances proof generation through recursive decomposition; Ax-Prover (Breen et al., 2026) and Numina-Lean-Agent (Liu et al., 2026) adopt the Model Context Protocol for model-verifier interaction; and Goedel-Architect (Chung et al., 2026) streamlines proving via blueprints. These methods apply agentic frameworks to existing LLMs. In contrast, we train the model within agentic loops, optimizing its capabilities of lemma decomposition and iterative refinement.

Reinforcement learning for theorem proving. Formal verification tools in theorem proving provide a clear reward for reinforcement learning. Kimina-Prover (Wang et al., 2025) and Goedel-Prover-V2 (Lin et al., 2025) employ the formal verifier as a tool in multi-turn RL. Seed-Prover 1.5 (Chen et al., 2025), LongCat-Flash-Prover (Wang et al., 2026), and Goedel-Code-Prover (Li et al., 2026) train models for agentic frameworks. They treat lemma decomposition as a standalone training process, relying on rule-based or LLM-as-a-judge rewards. OProver (Ma et al., 2026) introduces retrieval memory into the agentic training pipeline. In contrast, our work focuses on training the model's lemma decomposition capability within an agentic proving framework, using prover-guided verification as a grounded reward signal.

3 Problem Statement

In this section, we formulate key concepts of agentic theorem proving.

Definition 1 (Formal Statement Provability). A formal statement T consists of hypotheses and a goal. Let $\pi(T)$ denote the set of all candidate proofs that prover π can generate for T. The statement T is provable by π if this set contains a valid proof:

$$\text{Provable}(T, \pi) \iff \exists p \in \pi(T) : \ (\vdash p : T). \tag{3.1}$$

Definition 2 (Formal Statement Difficulty). The difficulty of a formal statement T for prover π is the probability that a proof p generated by π for T is invalid:

$$d(T, \pi) = 1 - \Pr_{p \in \pi(T)}[\vdash p : T]. \tag{3.2}$$

Thus, $\text{Provable}(T, \pi) \iff d(T, \pi) < 1$.

Definition 3 (Sketch Verifiability). A sketch $S = (L, p)$ consists of a nonempty lemma set $L = \{\ell_1, \ldots, \ell_m\}$ and a proof p of T. It is verifiable by π if π can prove every lemma in L and p establishes the original theorem T using them:

$$\text{Verifiable}(T, S, \pi) \iff \big(\forall \ell \in L,\ \text{Provable}(\ell, \pi)\big) \ \wedge\ (L \vdash p : T). \tag{3.3}$$

We find that strong provers may verify low-quality sketches (Table 1b), motivating the definition of effectiveness:

Definition 4 (Sketch Effectiveness). For a fixed reference distribution μ over provers, define the expected difficulty reduction of a sketch S:

$$\Delta(T, S) = \mathbb{E}_{\pi \sim \mu}\left[d(T, \pi) - \max_{\ell \in L} d(\ell, \pi) \right]. \tag{3.4}$$

A sketch is effective when its reconstruction is valid and its expected difficulty reduction is positive:

$$\text{Effective}(T, S) \iff (L \vdash p : T) \ \wedge\ \Delta(T, S) > 0. \tag{3.5}$$

In an effective sketch, the hardest lemma is easier than T on average over reference provers.

4 Methods

This section presents three key components of our training framework (Figure 2). To check the verifiability of a sketch (Definition 3), we introduce a prover-guided verification system (Section 4.1). Beyond verifiability, we aim to distinguish effective sketches (Definition 4) through prover selection, guided by theoretical and empirical analyses in Section 4.2. Building on this verification process, Section 4.3 describes how we train sketch refinement from lemma-level provability feedback.

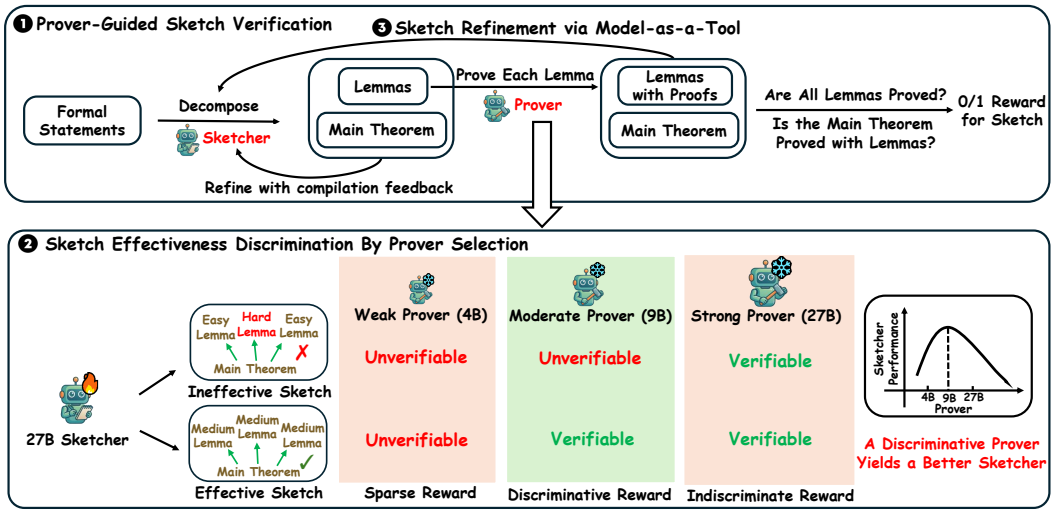


Figure 2: **Overview of SketchForge**. Prover-guided verification supplies rewards and feedback for learning sketch refinement through a model-as-a-tool interface. Selecting a moderately capable prover helps distinguish effective decompositions while avoiding sparse rewards.

4.1 Prover-Guided Sketch Verification Framework

Our sketch generation and verification framework for reinforcement learning consists of three parts:

(1) Sketch Generation. Given a formal statement T from the curated training dataset (see Section 5.1), the sketcher π_θ (e.g., Qwen3.5-27B) generates a candidate sketch $S = (L, p)$. The prompt (Appendix A) asks the sketcher to propose auxiliary lemmas $L = \{\ell_1, \ldots, \ell_m\}$ with proofs left

as `sorry`, together with a proof p of the original theorem based on these lemmas. The sketch is assembled by placing the lemma declarations before the original formal statement T and its proof p.

(2) Neuro-Symbolic Verification of Sketches. We first check that the sketch compiles with admitted lemmas ($L \vdash p : T$) using the Lean 4 verifier. To prove each lemma, we apply heuristic symbolic tactics (e.g., `aesop` (Limperg & From, 2023)); if they fail, a frozen prover π_ϕ (e.g., Qwen3.5-9B) generates up to k proof candidates in parallel. Each candidate is checked by Lean 4. The combined procedure returns a verified proof q_i for each lemma ℓ_i, or $\perp$ if no proof is found:

$$q_i = \mathrm{Prove}(\ell_i; \pi_\phi, k) \in \{q \mid \vdash q : \ell_i\} \cup \{\perp\}, \qquad i = 1, \ldots, m. \tag{4.1}$$

Once all lemmas are proved, we can obtain a completed proof $p^\star$. We recheck $p^\star$ in Lean 4 and confirm that the original statement T is unchanged and no additional axioms are introduced. Let M_ϕ denote the prover π_ϕ augmented with symbolic tactics. Sketch verification can then be expressed as:

$$(\forall i,\ q_i \neq \perp) \ \wedge \ (\vdash p^\star : T) \implies \mathrm{Verifiable}(T, S, M_\phi). \tag{4.2}$$

If the verification fails, compilation errors or unproved lemmas are returned to the sketcher for refinement (Section 4.3).

(3) Accelerating Sketch Verification by Proof Caching. To avoid repeatedly proving the same lemmas, we maintain a cache $\mathcal{C}_T$ of verified lemma-proof pairs, consulted via BEq (Liu et al., 2025b; Wu et al., 2025) equivalence before invoking π_ϕ. Beyond acceleration, accumulating verified proofs across refinement rounds also improves the sketch verification success rate.

4.2 Sketch Effectiveness Discrimination by Prover Selection

Section 4.1 establishes verifiability, but ineffective sketches can still receive positive rewards (Table 1b). To reduce such misalignment, we control the verification strength with a moderate prover π_ϕ: a strong prover may verify sketches with little difficulty reduction (Theorem 1), whereas a weak prover may leave more sketches unverified (Theorem 2). Following the experiments in Figure 3, we use Qwen3.5-9B as the prover π_ϕ.

Definition 5 (Prover Capability). For a fixed distribution ν over formal statements, the capability of prover π is the probability that a statement sampled from ν is provable by π (Definition 1):

$$C(\pi) = \Pr_{T \sim \nu}[\mathrm{Provable}(T, \pi)] \in [0, 1]. \tag{4.3}$$

Theorem 1 (Verifiability Monotonicity for Low-Effectiveness Sketches). Fix a distribution of (T, S) and provers π_s, π_w with $C(\pi_s) > C(\pi_w)$, and assume every lemma occurring in sketches that is provable by π_w is also provable by π_s. Then, for any $\delta \geq 0$ with $\Pr[\Delta(T, S) \leq \delta] > 0$,

$$\Pr[\mathrm{Verifiable}(T, S, \pi_s) \mid \Delta(T, S) \leq \delta] \geq \Pr[\mathrm{Verifiable}(T, S, \pi_w) \mid \Delta(T, S) \leq \delta]\,. \tag{4.4}$$

Proof Sketch. Under the assumption, every sketch verifiable by π_w is verifiable by π_s, and restricting this implication to the subset $\Delta(T, S) \leq \delta$ yields the inequality (complete proof in Appendix B.1).

Remark. A stronger prover is more likely to verify low-effectiveness sketches, which then receive positive rewards despite offering little meaningful difficulty reduction.

Theorem 2 (Capability Gap Bound on Sketch Verification). Let $C(\pi_s) > C(\pi_w)$ and fix a distribution of (T, S) where $S = (L, p)$ has a valid reconstruction and a designated lemma $\ell \in L$ drawn from ν while every other lemma is provable by both provers. Then

$$\Pr[\mathrm{Verifiable}(T, S, \pi_s) \wedge \neg\, \mathrm{Verifiable}(T, S, \pi_w)] \geq C(\pi_s) - C(\pi_w). \tag{4.5}$$

Proof Sketch. The two provers verify fractions $C(\pi_s)$ and $C(\pi_w)$ whose overlap is at most $C(\pi_w)$, leaving a fraction of at least $C(\pi_s) - C(\pi_w)$ verifiable only by π_s (complete proof in Appendix B.2).

Remark. A weak prover may fail to verify sketches verifiable by a strong prover, limiting opportunities for positive reward.

To examine these effects empirically, we train a Qwen3.5-27B sketcher with five prover configurations: symbolic tactics, Qwen3.5-4B, 9B, 27B, and 397B-A17B (details in Appendix C and E). We fix a unified Qwen3.5-27B prover during evaluation (see Sections 5.1 and 5.2 for details). As shown in Figure 3, training with the 9B prover continues to improve and achieves the highest final MiniF2F Pass@8 among the tested configurations, motivating our empirical choice.



Figure 3: Evaluation curves for sketchers trained with five different provers.

4.3 Learning Sketch Refinement via Model-as-a-Tool

(1) Multi-Round Sketch Refinement via Model-as-a-Tool. To support sketch refinement according to lemma provabilities, the sketch verification process is packaged as a callable tool in our framework. This allows the model to apply its tool-use capabilities to multi-round refinement through interaction with the verification framework.

Specifically, at round t, the sketcher π_θ analyzes the theorem T and interaction history H_t to construct $S_t = (L_t, p_t)$, then submits it through a tool call a_t (prompts in Appendix A). The tool uses M_ϕ, the frozen prover augmented with symbolic tactics, to find lemma proofs, then checks the completed proof with Lean 4 (Section 4.1). It returns a binary reward r_t (1 for verified and 0 for unverified) and feedback f_t on compilation errors or unproved lemmas. This lemma-level feedback guides revision of the entire sketch, including correcting statements, decomposing difficult lemmas, and updating the main proof. Iteration stops on success or at the preset limit of K rounds (see Algorithm 1 in Appendix G).

(2) Agentic Reinforcement Learning for Sketch Refinement. We use GRPO (Shao et al., 2024) with token-level averaging and asymmetric clipping. For each formal statement T, the rollout sketcher policy π_θ generates G trajectories with tool calls and refinement actions. Each trajectory τ_i contains $n_i \le K$ rounds and receives the final-round reward $R_i = r^{(i)}_{n_i-1}$. This reward indicates whether the trajectory produces a verified proof. Let o_i denote the sequence of tokens optimized in trajectory τ_i, consisting of the sketcher's thinking and tool calls. Tool responses f_t remain in the context but are excluded from o_i, masking them out of both the objective and its token-count denominator. With the expectation taken over $T \sim \mathcal{D}$ and $\{\tau_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid T)$, we maximize

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{T,\{\tau_i\}}\Big[\frac{1}{\sum_{i=1}^{G}|o_i|}\sum_{i=1}^{G}\sum_{j=1}^{|o_i|}\min\big(\rho_{i,j}\hat{A}_{i,j},\ \text{clip}(\rho_{i,j}, 1-\varepsilon_{\text{low}}, 1+\varepsilon_{\text{high}})\hat{A}_{i,j}\big)\Big], \quad (4.6)$$

where

$$\rho_{i,j} = \frac{\pi_\theta(o_{i,j} \mid T, \tau_{i,<j})}{\pi_{\theta_{\text{old}}}(o_{i,j} \mid T, \tau_{i,<j})}, \qquad \hat{A}_{i,j} = \frac{R_i - \text{mean}(\{R_i\}_{i=1}^{G})}{\text{std}(\{R_i\}_{i=1}^{G})}. \quad (4.7)$$

Here, $\tau_{i,<j}$ is the history before $o_{i,j}$, including tool responses; $\rho_{i,j}$ is the importance sampling ratio, and $\varepsilon_{\text{low}}, \varepsilon_{\text{high}}$ are clipping thresholds. We set $\hat{A}_{i,j} = 0$ for groups with identical rewards.

5 Experiments

This section presents the implementation details of SketchForge (Section 5.1) and our evaluation settings (Section 5.2). We compare SketchForge-27B with representative open-weight reasoning models and open-weight prover models (Section 5.3). We also validate our core designs through ablation studies (Section 5.4), and analyze test-time scaling of sketch-refinement rounds (Section 5.5) and proof attempts per lemma (Appendix F).

5.1 Implementation Details

Data Curation. To obtain training data, we autoformalize ∼400K natural-language mathematics problems from public datasets (LI et al., 2024; Moshkov et al., 2025) with Reform-32B (Chen et al.,

2026), then use Qwen3.5-Plus as both sketcher and prover in our framework (Section 4.1) to obtain ∼67K verified statements. We retain those successful solutions with at least 4 lemmas and 5 refinement rounds, and exclude statements solved by Qwen3.5-27B within 8 whole-proof attempts. This yields ∼11K difficult but provable statements for sketcher RL training. We apply 13-gram matching (Brown et al., 2020) to decontaminate the training data against the evaluation benchmarks.

Training Configuration. We train a Qwen3.5-27B sketcher for 100 steps with verl (Sheng et al., 2025), using a frozen Qwen3.5-9B prover and Lean 4.17.0-rc1 (Moura & Ullrich, 2021) with a verification timeout of 600 seconds. Training uses 32 GPUs, with 4 additional GPUs serving the prover. Each rollout batch contains 32 problems with 32 trajectories per problem ($G = 32$). The maximum response length is 40,960 tokens (excluding prover tokens), with at most five sketch-refinement rounds ($K = 5$). Detailed training hyperparameters are provided in Appendix H.

5.2 Evaluation Settings

Benchmarks. Following LongCat-Flash-Prover (Wang et al., 2026) and OProver (Ma et al., 2026), we evaluate the models on five formal mathematical benchmarks: ProofNet (Azerbayev et al., 2023) contains 186 undergraduate textbook statements; ProverBench (Ren et al., 2025) combines 325 textbook and competition problems. We also assess competition mathematics using PutnamBench (Tsoukalas et al., 2024) (672 Putnam competition problems), MathOlympiad (Lin et al., 2025) (360 olympiad problems), and MiniF2F (Zheng et al., 2022) (244 high-school problems).

Baselines. We compare SketchForge-27B with its base model to show the absolute performance gain through RL. Besides, we compare with the reasoning models DeepSeek-V3.2 (Liu et al., 2025a) and Kimi-K2.5 (Team, 2026), and the specialized provers Kimina-Prover (Wang et al., 2025), DeepSeek-Prover-V2 (Ren et al., 2025), Leanabell-Prover-V2 (Ji et al., 2025), Goedel-Prover-V2 (Lin et al., 2025), LongCat-Flash-Prover (Wang et al., 2026), and OProver (Ma et al., 2026).

Evaluation Configuration. We report the Pass@k (Chen et al., 2021) performance metric:

$$\text{Pass}\,@k = \mathbb{E}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right], \tag{5.1}$$

where n is the number of samples, and c is the number of verified proofs. To probe the limits of generalization beyond the training budget, evaluation allows up to 81,920 tokens (excluding prover tokens), 15 sketch-refinement rounds and 8 parallel proof attempts per lemma. Since whole-proof performance on MiniF2F is preserved after sketcher training (Appendix D), SketchForge-27B serves as both sketcher and prover at evaluation time. We use MiniF2F, the easiest benchmark in our suite, for validation and prover selection, and the other 4 harder benchmarks to assess generalization. Validation Pass@8 is monitored every 10 steps with a fixed Qwen3.5-27B prover (Figures 3).

5.3 Main Results

The evaluation results are shown on Table 2.

SketchForge-27B substantially outperforms its base model on all five benchmarks. Under the same evaluation settings, SketchForge-27B improves Pass@32 by an average of 4.7 percentage points over Qwen3.5-27B, demonstrating the effectiveness of our reinforcement learning method.

SketchForge-27B surpasses larger open-weight theorem proving systems on three challenging benchmarks. Compared with open-source agentic theorem proving systems OProver-32B and LongCat-Flash-Prover-560B, our model achieves the highest Pass@32 on ProofNet, ProverBench, and PutnamBench. Notably, these gains are achieved through RL alone with 11K formal statements, using substantially less training data than OProver-32B (1.77M statements and 6.86M proofs).

5.4 Ablation Studies

(1) Prover-guided sketch verification outperforms rubric-based rewards. To validate the importance of prover-guided sketch verification (Section 4.1), we compare against a sketch reward adapted from Seed-Prover 1.5 (Chen et al., 2025), which combines rule-based checks with an LLM-

Table 2: Pass@32 performances across five theorem-proving benchmarks. Within each benchmark column, the best listed result is bold and the second-best is underlined. Results except Qwen3.5-27B and SketchForge-27B are from OProver (Ma et al., 2026).

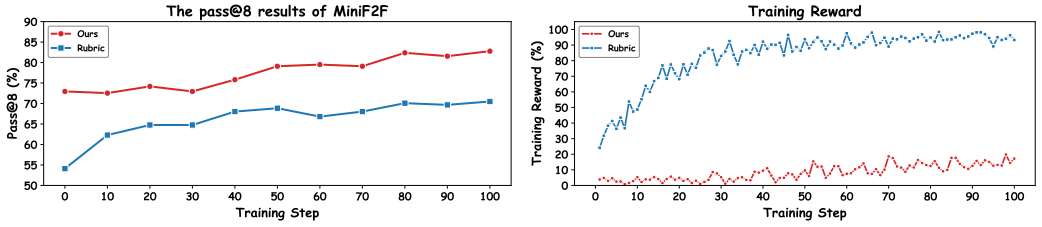
Model	#Params	ProofNet (Pass@32)	ProverBench (Pass@32)	PutnamBench (Pass@32)	MathOlympiad (Pass@32)	MiniF2F (Pass@32)
(*Open-Weight Reasoning Models*)						
DeepSeek-V3.2	671B	20.4	42.8	5.8	14.7	77.9
Kimi-K2.5	1T	19.9	44.3	1.2	7.5	76.6
(*Open-Weight Prover Models*)						
Kimina-Prover-8B	8B	13.0	40.5	2.3	7.6	77.2
Kimina-Prover-72B	72B	16.3	45.0	4.0	11.5	83.4
DeepSeek-Prover-V2-7B	7B	18.1	45.7	1.5	7.9	73.9
DeepSeek-Prover-V2-671B	671B	30.5	52.9	3.3	13.9	82.4
Leanabell-Prover-V2-KM	8B	13.4	39.8	–	–	68.4
Leanabell-Prover-V2-DS	7B	23.7	47.8	–	–	76.6
Goedel-Prover-V2-8B	8B	17.3	48.6	0.7	9.9	79.4
w/ self-correction	8B	–	–	–	–	86.7
Goedel-Prover-V2-32B	32B	22.0	51.0	5.0	16.0	85.8
w/ self-correction	32B	–	–	8.6	20.3	90.4
LongCat-Flash-Prover	560B	19.9	49.9	4.9	16.9	84.4
whole-proof w/ TIR	560B	<u>36.1</u>	57.9	10.4	**27.5**	90.2
OProver-8B	8B	31.9	56.0	9.0	21.7	91.8
OProver-32B	32B	33.2	<u>58.2</u>	<u>11.3</u>	22.8	**93.3**
Qwen3.5-27B	27B	32.8	55.4	8.2	20.3	90.2
SketchForge-27B (Ours)	27B	**38.7** +5.9	**61.2** +5.8	**12.9** +4.7	<u>25.2</u> +4.9	<u>92.6</u> +2.4

as-a-judge assessment of lemma correctness and decomposition quality:

$$R_{\text{rubric}} = 1 \text{ if } N_{\text{lemmas}} \geq 3 \ \wedge \ S_{\text{FL}} \geq 0 \ \wedge \ S_{\text{NL}} \geq 0.7, \text{ else } 0. \tag{5.2}$$

Here, N_{lemmas} is the lemma count, and S_{FL} and S_{NL} are the syntactic and semantic scores, respectively. We adapt this baseline to our agentic training framework by replacing the provability feedback from the prover π_ϕ with the refinement feedback from Qwen3.5-27B's judgement.

It is worth noting that both the rubric baseline and SketchForge are evaluated by our *prover-guided verification framework*. The rubric baseline stops sketch refinement once the LLM judge accepts the sketch and then proceeds to lemma proving, so proof provabilities cannot guide further refinement. This explains the lower evaluation performance of the rubric method (Figure 4a). In contrast, its training reward rises rapidly and later remains mostly above 90% (Figure 4b). This discrepancy suggests potential reward hacking, highlighting the unreliability of rubric scores as sketch rewards.



(a) (b)

Figure 4: Evaluation performance and training reward curves for the rubric-based baseline and prover-guided sketch verification.

(2) A moderately capable 9B training prover yields better performance under a larger sampling budget. We extend the finding in Section 4.2 to the main evaluation budget by replacing the 9B training prover with 4B or 27B and evaluating the resulting models as in Section 5.2. Both alternatives substantially underperform the 9B configuration across all five benchmarks (Table 3), supporting the importance of selecting an appropriate prover for verification.

(3) Sketch-level refinement substantially outperforms single-round decomposition and consistently outperforms whole-proof refinement in end-to-end Pass@32. Table 4 compares our method with *Single-Round Sketch Generation* ($K = 1$, without sketch refinement) and *Whole-*

Table 3: Pass@32 (%) performance of using different provers for verification in training.

Sketcher	Prover	ProofNet (Pass@32)	ProverBench (Pass@32)	PutnamBench (Pass@32)	MathOlympiad (Pass@32)	MiniF2F (Pass@32)
Qwen3.5-27B	Qwen3.5-9B (Ours)	**38.7**	**61.2**	**12.9**	**25.2**	**92.6**
	Qwen3.5-4B	36.6	57.5	8.2	21.1	89.8
	Qwen3.5-27B	33.9	57.8	7.4	20.0	88.9

Table 4: Pass@32 (%) performance of different refinement strategies.

Setting	ProofNet (Pass@32)	ProverBench (Pass@32)	PutnamBench (Pass@32)	MathOlympiad (Pass@32)	MiniF2F (Pass@32)
Sketch Refinement (Ours)	**38.7**	**61.2**	**12.9**	**25.2**	**92.6**
Single-Round Sketch Generation	11.8	42.8	2.5	8.3	77.1
Whole-Proof Refinement	32.8	56.9	7.4	21.9	88.1

Proof Refinement, which uses a prover trained to refine complete proofs under the same settings as SketchForge-27B. Both multi-round configurations allow 15 rounds.

Multi-round sketch refinement substantially outperforms single-round generation, highlighting that iterative refinement is the key to realizing the potential of lemma decomposition. The gains over whole-proof refinement also suggest that the lemma provabilities provide finer-grained feedback on the high-level sketch structure, helping the sketcher localize and correct errors more effectively.

5.5 Test-Time Scaling of Sketch Refinement

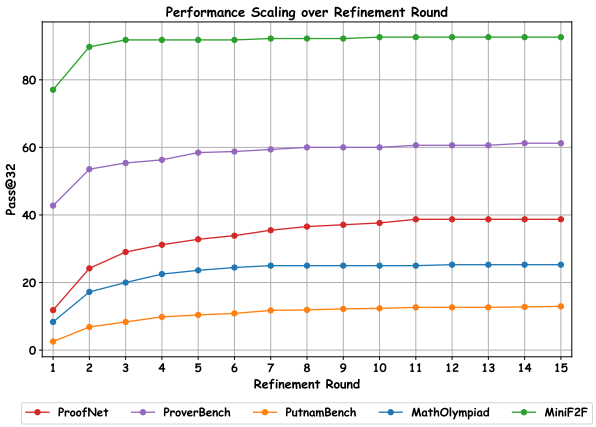


Figure 5: Pass@32 across five benchmarks over the number of sketch refinement rounds.

End-to-end Pass@32 across all benchmarks improve rapidly in early refinement rounds before approaching a plateau. Figure 5 tracks cumulative proof success by the first successful round. Among the five benchmarks, ProverBench shows particularly sustained gains with the refinement. These results show that our framework can support test-time scaling through additional refinement rounds. Appendix I provides an example of sketch refinement.

6 Conclusion

We introduced SketchForge, a reinforcement learning framework for effective lemma decompositions in agentic theorem proving. It combines prover-guided sketch verification, decomposition quality discrimination through prover selection, and sketch refinement from lemma-level feedback. Trained on only 11K formal statements, SketchForge-27B improves over its base model on all five benchmarks and achieves performance comparable to or better than that of OProver-32B. Ablation studies support the three core designs, while additional refinement rounds enable test-time scaling.

AI USE STATEMENT

As described in Section 5.1, LLMs were used to autoformalize mathematical problems and generate candidate sketches and proofs for training-data curation. Generative AI tools also assisted with English translation and language editing, literature search and summarization, manuscript organization, and LaTeX preparation of algorithms, tables, and figure layouts. They supported the refinement of mathematical definitions, theorem statements, and proof drafts and interpretation of experimental results. AI-assisted material was iteratively reviewed and revised during manuscript preparation; formal proofs counted as successful outcomes were checked with Lean 4. The authors take responsibility for the final content of this work, including all AI-assisted text, claims, and artifacts.

REPRODUCIBILITY STATEMENT

To support reproducibility, Section 5.1 describes data curation, decontamination, and the training setup, while Section 5.2 specifies the benchmarks and evaluation protocol. Appendix H lists the training hyperparameters. The prompt templates and sketch verification and refinement algorithm are provided in Appendices A and G, respectively.

REFERENCES

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. 2023. URL https://arxiv.org/abs/2302.12433.

Benjamin Breen, Marco Del Tredici, Jacob McCarran, Javier Aspuru Mijares, Weichen Winston Yin, Kfir Sulimany, Jacob M. Taylor, Frank H. L. Koppens, and Dirk Englund. Ax-prover: A deep reasoning agentic framework for theorem proving in mathematics and quantum physics, 2026. URL https://arxiv.org/abs/2510.12787.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Neelakantan, et al. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Chenrui Cao, Liangcheng Song, Zenan Li, Xinyi Le, Xian Zhang, Hui Xue, and Fan Yang. Reviving dsp for advanced theorem proving in the era of reasoning models. 2025. URL https://arxiv.org/abs/2506.11487.

Guoxin Chen, Jing Wu, Xinjie Chen, Wayne Xin Zhao, Ruihua Song, Chengxi Li, Kai Fan, Dayiheng Liu, and Minpeng Liao. Reform: Reflective autoformalization with prospective bounded sequence optimization, 2026. URL https://arxiv.org/abs/2510.24592.

Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, Zhihong Wang, Mingxuan Wang, Chenrui Wei, Shufa Wei, Huajian Xin, Fan Yang, Weihao Gao, Zheng Yuan, Tianyang Zhan, Zeyu Zheng, Tianxi Zhou, and Thomas Hanwen Zhu. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience, 2025. URL https://arxiv.org/abs/2512.17260.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021. URL https://arxiv.org/abs/2107.03374.

Jui-Hui Chung, Ziyang Cai, Zihao Li, Qishuo Yin, Rohit Agarwal, Simon Park, Rodrigo Porto, Narutatsu Ri, Ziran Yang, Shange Tang, Xingyu Dang, Hongzhou Lin, Mengdi Wang, Danqi Chen, Chi Jin, Liam H Fowl, and Sanjeev Arora. Goedel-architect: Streamlining formal theorem proving with blueprint generation and refinement, 2026. URL https://arxiv.org/abs/2606.06468.

Xingguang Ji, Yahui Liu, Qi Wang, Jingyuan Zhang, Yang Yue, Rui Shi, Chenxi Sun, Fuzheng Zhang, Guorui Zhou, and Kun Gai. Leanabell-prover-v2: Verifier-integrated reasoning for formal theorem proving via reinforcement learning. 2025. URL https://arxiv.org/abs/2507.08649.

Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. 2023. URL https://arxiv.org/abs/2210.12283.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Jia LI, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. Numinamath. https://huggingface.co/datasets/AI-MO/NuminaMath-CoT, 2024.

Zenan Li, Ziran Yang, Deyuan He, Haoyu Zhao, Andrew Zhao, Shange Tang, Kaiyu Yang, Aarti Gupta, Zhendong Su, and Chi Jin. Goedel-code-prover: Hierarchical proof search for open state-of-the-art code verification. 2026. URL https://arxiv.org/abs/2603.19329.

Jannis Limperg and Asta Halkjær From. Aesop: White-box best-first proof search for lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, pp. 253–266, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700262. doi: 10.1145/3573105.3575671. URL https://doi.org/10.1145/3573105.3575671.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction, 2025. URL https://arxiv.org/abs/2508.03613.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3.2: Pushing the frontier of open large language models. 2025a. URL https://arxiv.org/abs/2512.02556.

Junqi Liu, Zihao Zhou, Zekai Zhu, Marco Dos Santos, Weikun He, Jiawei Liu, Ran Wang, Yunzhou Xie, Junqiao Zhao, Qiufeng Wang, Lihong Zhi, Jia Li, and Wenda Li. Numina-lean-agent: An open and general agentic reasoning system for formal mathematics. 2026. URL https://arxiv.org/abs/2601.14027.

Qi Liu, Xinhao Zheng, Xudong Lu, Qinxiang Cao, and Junchi Yan. Rethinking and improving autoformalization: Towards a faithful metric and a dependency retrieval-based approach. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=hUb2At2DsQ.

David Ma, Kaijing Ma, Shawn Guo, Yunfeng Shi, Enduo Zhao, Jiajun Shi, Zhaoxiang Zhang, Gavin Cheung, Jiaheng Liu, and Zili Wang. Oprover: A unified framework for agentic formal theorem proving, 2026. URL https://arxiv.org/abs/2605.17283.

Ivan Moshkov, Darragh Hanley, Ivan Sorokin, Shubham Toshniwal, Christof Henkel, Benedikt Schifferer, Wei Du, and Igor Gitman. Aimo-2 winning solution: Building state-of-the-art mathematical reasoning models with openmathreasoning dataset, 2025. URL https://arxiv.org/abs/2504.16891.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pp. 625–635. Springer, 2021.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. 2020. URL https://arxiv.org/abs/2009.03393.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. 2025. URL https://arxiv.org/abs/2504.21801.

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.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. pp. 1279–1297, March 2025. doi: 10.1145/3689031.3696075. URL http://dx.doi.org/10.1145/3689031.3696075.

Kimi Team. Kimi k2.5: Visual agentic intelligence. 2026. URL https://arxiv.org/abs/2602.02276.

George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. 2024. URL https://arxiv.org/abs/2407.11214.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively building formal proofs with informal reasoning. 2026. URL https://arxiv.org/abs/2509.22819.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. 2025. URL https://arxiv.org/abs/2504.11354.

Jianing Wang, Jianfei Zhang, Qi Guo, Linsen Guo, Rumei Li, Chao Zhang, Chong Peng, Cunguang Wang, Dengchang Zhao, Jiarong Shi, Jingang Wang, Liulin Feng, Mengxia Shen, Qi Li, Shengnan An, Shun Wang, Wei Shi, Xiangyu Xi, Xiaoyu Li, Xuezhi Cao, Yi Lu, Yunke Zhao, Zhengyu Chen, Zhimin Lin, Wei Wang, Peng Pei, and Xunliang Cai. Longcat-flash-prover: Advancing native formal reasoning via agentic tool-integrated reinforcement learning. 2026. URL https://arxiv.org/abs/2603.21065.

Yutong Wu, Di Huang, Ruosi Wan, Yue Peng, Shijie Shang, Chenrui Cao, Lei Qi, Rui Zhang, Zidong Du, Jie Yan, and Xing Hu. Stepfun-formalizer: Unlocking the autoformalization potential of llms through knowledge-reasoning fusion. 2025. URL https://arxiv.org/abs/2508.04440.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. 2024. URL https://arxiv.org/abs/2405.14333.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. 2022. URL https://arxiv.org/abs/2109.00110.

APPENDIX

A Prompt Templates and Tool Schema **14**
A.1 Sketch Generation 14
A.2 Sketch Verification Tool Schema 14
A.3 Lemma Proving 15
A.4 Sketch Refinement from Compilation Feedback 15
A.5 Sketch Refinement from Lemma Feedback 15

B Proofs of the Sketch Effectiveness Discrimination Theorems **16**
B.1 Proof of Verifiability Monotonicity for Low-Effectiveness Sketches 16
B.2 Proof of Capability Gap Bound on Sketch Verification 16

C Whole-Proof Performance of Different Provers **17**

D Whole-Proof Performance After Sketch-Level RL **17**

E Training Reward Curves of Different Verification Configurations **18**

F Performance Scaling with Proof Attempts per Lemma **18**

G The Algorithm of Sketch Verification and Refinement **19**

H Training Hyperparameters **20**

I Case Study: Sketch Refinement Through Lemma Provability **21**
I.1 Overview of the Problem and Refinement 21
I.2 Detailed Thinking and Tool Calls 22

A Prompt Templates and Tool Schema

We show the prompts for sketch generation and iterative refinement, lemma verification by provers, and the tool schema of sketch verification in this section.

A.1 Sketch Generation

This prompt for the sketcher π_θ injects the formal statement T at $\{\texttt{formal_statement}\}$ and asks the model to decompose T into lemmas with `sorry` placeholders, reconstruct T from them, iteratively call `verify_sketch` and refine the sketches until it returns `Correct!`.

```
You are a dual-domain expert in both rigorous mathematical reasoning and Lean 4
theorem proving.
You are given a problem statement in Lean 4. Your task is to propose a proof sketch
for the statement, then use the `verify_sketch` tool to verify the correctness of
your sketch and refine it until it is correct. Once the tool returns `Correct!`,
your final response should only consist of a single word: Finish. There is no need
to output the sketch itself.

**Problem Statement:**
{formal_statement}
```

Figure 6: Sketch generation prompt for the sketcher π_θ.

A.2 Sketch Verification Tool Schema

This tool schema accompanies the prompt in Appendix A.1 and declares `verify_sketch` with parameters `lemmas` and `main_theorem`, structuring each sketch for model-as-a-tool verification.

```
VERIFY_SKETCH_TOOL = [{
    "type": "function",
    "function": {
        "name": "verify_sketch",
        "description": "A tool for verifying a Lean4 proof sketch.",
        "parameters": {
            "type": "object",
            "properties": {
                "lemmas": {
                    "type": "array",
                    "items": {"type": "string"},
                    "description": "An non-empty list of auxiliary lemma statements (at least one
lemma) starting with the keyword `lemma` and ending with `by sorry`, used to support the proof of the
original statement. Do not provide any proofs of these lemmas. Each lemma should be a part of
decomposition of the original statement and be strictly simpler than it. The tool will invoke an
external prover to verify the correctness of each lemma. If all lemmas are proven, it will return
`Correct!`.",
                },
                "main_theorem": {
                    "type": "string",
                    "description": "The complete proof of the given problem statement that solve the
goal by directly applying your lemmas, with no `sorry`s. All your `lemmas` must be used in
`main_theorem`. The tool will automatically assemble the proven lemmas to obtain a final proof of the
original statement.",
                },
            },
            "required": ["lemmas", "main_theorem"],
        },
    },
}]
```

Figure 7: Python representation of the `verify_sketch` tool schema.

A.3 LEMMA PROVING

We use the whole-proof generation prompt in Goedel-Prover-V2 (Lin et al., 2025) for lemma verification by a frozen prover π_ϕ, where multiple whole-proof candidates are sampled in parallel and checked by Lean 4.

```
Complete the following Lean 4 code:

```lean4
{formal_statement}
```

Before producing the Lean 4 code to formally prove the given theorem, provide a
detailed proof plan outlining the main proof steps and strategies.
The plan should highlight key ideas, intermediate lemmas, and proof structures that
will guide the construction of the final formal proof.
```

Figure 8: Lemma Proving prompt for the prover π_ϕ.

A.4 SKETCH REFINEMENT FROM COMPILATION FEEDBACK

This feedback is returned when the assembled sketch fails to compile, i.e., when the reconstruction does not type-check under the admitted lemmas, so that $L \vdash p : T$ does not hold. It passes the Lean 4 error messages (injected at {error_info}) back to the sketcher and requests a corrected sketch.

```
There are errors in your lemmas or main theorem. Please correct them.
Error Information:
{error_info}
```

Figure 9: Compilation feedback prompt for sketch refinement.

A.5 SKETCH REFINEMENT FROM LEMMA FEEDBACK

This feedback template is returned when the sketch compiles but the prover fails to prove some lemma. It reports the unproved lemma and the two possible causes, then asks the sketcher either to correct it or to decompose it further.

```
The lemma `{lemma_with_error_name}` can not be proven:
```lean4
{lemma_with_error}
```

This may be due to one of the following two reasons:
1. The lemma is false.
2. The lemma is too difficult and requires further decomposition.
For the first case, please correct the erroneous lemma and the relevant code in the
main theorem that uses it. For the second case, please further decompose the lemma
and modify the main theorem accordingly.
```

Figure 10: Lemma provability feedback prompt for sketch refinement.

B Proofs of the Sketch Effectiveness Discrimination Theorems

We prove the two theorems in Section 4.2 using the definitions of provability, verifiability, and effectiveness (Section 3). We assume ν is the fixed reference distribution over formal statements from Definition 5. Each theorem uses its stated, fixed distribution of pairs (T, S); probabilities are taken over those pairs. Every verifiable sketch has finitely many correct lemmas and a valid reconstruction.

B.1 Proof of Verifiability Monotonicity for Low-Effectiveness Sketches

Theorem 1 (Verifiability Monotonicity for Low-Effectiveness Sketches). Fix a distribution of (T, S) and provers π_s, π_w with $C(\pi_s) > C(\pi_w)$, and assume every lemma occurring in sketches that is provable by π_w is also provable by π_s. Then, for any $\delta \geq 0$ with $\Pr[\Delta(T, S) \leq \delta] > 0$,

$$\Pr[\text{Verifiable}(T, S, \pi_s) \mid \Delta(T, S) \leq \delta] \geq \Pr[\text{Verifiable}(T, S, \pi_w) \mid \Delta(T, S) \leq \delta]\,. \tag{B.1}$$

Proof. Fix a sketch $S = (L, p)$ with a valid reconstruction. Suppose it is verifiable by π_w. By Definition 3, every lemma in L is provable by π_w. Under the coverage assumption of Theorem 1 and $C(\pi_s) > C(\pi_w)$, all these lemmas are also provable by π_s. The reconstruction is unchanged, so

$$\text{Verifiable}(T, S, \pi_w) \Longrightarrow \text{Verifiable}(T, S, \pi_s). \tag{B.2}$$

This implication holds for each sketch, including every sketch with $\Delta(T, S) \leq \delta$. Since Δ uses a fixed reference distribution over provers (Definition 4), this subset does not change when the verification prover changes. Taking conditional probabilities on this same subset proves Equation (4.4). The inequality is strict exactly when the stronger prover newly verifies a set of sketches with positive conditional probability. For $\delta > 0$, this subset can include effective sketches with $0 < \Delta(T, S) \leq \delta$.

A repeated-theorem sketch $S_T = (\{T\}, p)$, whose reconstruction simply invokes its only lemma, is a special case: Definition 4 gives $\Delta(T, S_T) = 0$, and Definition 3 gives $\text{Verifiable}(T, S_T, \pi) \iff \text{Provable}(T, \pi)$. Thus, over $T \sim \nu$, the verifiable fraction of this particular family is exactly $C(\pi)$.

B.2 Proof of Capability Gap Bound on Sketch Verification

Theorem 2 (Capability Gap Bound on Sketch Verification). Let $C(\pi_s) > C(\pi_w)$ and fix a distribution of (T, S) where $S = (L, p)$ has a valid reconstruction and a designated lemma $\ell \in L$ drawn from ν while every other lemma is provable by both provers. Then

$$\Pr[\text{Verifiable}(T, S, \pi_s) \wedge \neg\, \text{Verifiable}(T, S, \pi_w)] \geq C(\pi_s) - C(\pi_w). \tag{B.3}$$

Proof. For each sampled pair (T, S), the reconstruction is valid and every lemma other than the designated ℓ is provable by both provers. Therefore, Definition 3 reduces verification to the provability of ℓ:

$$\text{Verifiable}(T, S, \pi) \iff \text{Provable}(\ell, \pi), \qquad \pi \in \{\pi_s, \pi_w\}. \tag{B.4}$$

The marginal distribution of ℓ is ν. Taking probabilities therefore gives verification fractions $C(\pi_s)$ and $C(\pi_w)$, respectively. The other lemmas may depend on ℓ; their assumed provability makes such dependence irrelevant to this equivalence.

Let A be the event that the sketch is verifiable by π_s, and B the event that it is verifiable by π_w. Decomposing A according to whether B holds gives

$$\Pr(A \cap B^{\mathrm{c}}) = \Pr(A) - \Pr(A \cap B). \tag{B.5}$$

Because $\Pr(A \cap B) \leq \Pr(B)$, it follows that

$$\Pr(A \cap B^{\mathrm{c}}) \geq \Pr(A) - \Pr(B) = C(\pi_s) - C(\pi_w). \tag{B.6}$$

This is Equation (4.5). No independence or inclusion between the two provers' sets of provable statements is needed. If the stronger prover can prove every statement provable by the weaker one, $B \subseteq A$ and the bound holds with equality. Since the bound is strictly positive when $C(\pi_s) > C(\pi_w)$, a positive fraction of sketches is missed by the weaker prover despite being verifiable by the stronger one.

The condition on the remaining lemmas makes the relationship between capability and sketch verifiability explicit. A higher overall capability alone does not guarantee that one prover can verify every sketch verifiable by another. Theorem 2 quantifies missed verification opportunities in the stated setting without assuming such pointwise dominance. In contrast, the monotonicity claim in Theorem 1 explicitly requires the coverage assumption. Both results concern the logical predicates in Section 3, while finite-search verification and its effect on training are evaluated empirically.

Scope of effectiveness. Definition 4 measures the expected reduction in the hardest lemma's difficulty relative to the fixed reference distribution μ. It does not account for the number of lemmas or their joint proof cost, and therefore does not guarantee higher end-to-end success under a fixed search budget.

C Whole-Proof Performance of Different Provers

Larger models achieve higher whole-proof Pass@k on MiniF2F. Table 5 compares the whole-proof performance of symbolic tactics alone, Qwen3.5-4B, Qwen3.5-9B, Qwen3.5-27B, and Qwen3.5-397B-A17B used in Section 4.2. Performance increases with model size on this benchmark.

Table 5: Standalone proof-generation performance (%) on MiniF2F.

Prover	Pass@1	Pass@8	Pass@32
Symbolic Tactics	34.8	34.8	34.8
Qwen3.5-4B	35.8	56.8	67.2
Qwen3.5-9B	44.7	65.8	72.1
Qwen3.5-27B	53.2	74.7	79.1
Qwen3.5-397B-A17B	**63.9**	**79.3**	**83.6**

D Whole-Proof Performance After Sketch-Level RL

Sketch-level RL preserves whole-proof performance on MiniF2F. Table 6 compares the whole-proof performance of Qwen3.5-27B and SketchForge-27B on MiniF2F. Pass@1 and Pass@8 improve after training, while Pass@32 remains unchanged. These results support using SketchForge-27B as both the sketcher and the prover in the main experiments (Section 5.2).

Table 6: Whole-proof performance (%) on MiniF2F before and after RL training for the sketcher.

Model	Pass@1	Pass@8	Pass@32
Qwen3.5-27B	53.2	74.7	**79.1**
SketchForge-27B (Ours)	**58.0**	**76.0**	**79.1**

E Training Reward Curves of Different Verification Configurations

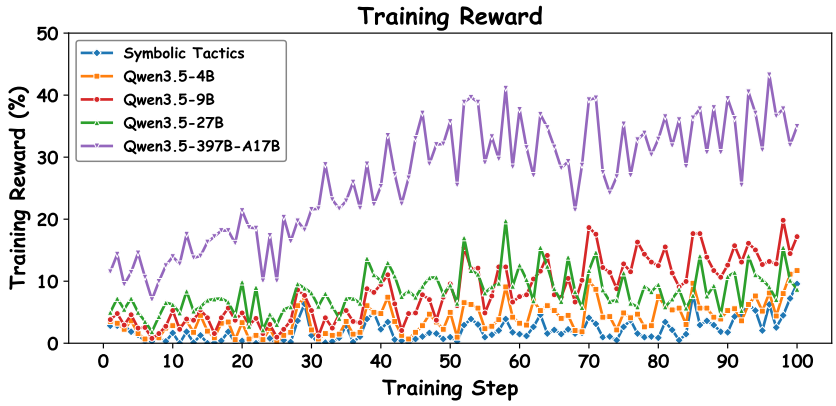


Figure 11: Training reward curves of the five prover configurations in Section 4.2.

Figure 11 reports the training reward curves of the five verification configurations in Section 4.2. Consistent with Theorem 1, the reward level tracks prover capability rather than sketch quality: the 397B prover returns the highest rewards yet yields a weaker sketcher (Figure 3).

F Performance Scaling with Proof Attempts per Lemma

Applying more proof attempts per lemma consistently improves end-to-end performance. The evaluation of our main experiment (Section 5.3) allows up to 8 proof attempts per lemma. Table 7 varies this budget over $\{1, 2, 4, 8\}$ with the number of interaction rounds fixed at 15, and reports the final Pass@32 performance on the five benchmarks. Performance improves overall as more lemma-proving attempts are allowed.

Table 7: Pass@32 (%) on five benchmarks versus the number of proof attempts per lemma at evaluation time.

Attempts	MathOlympiad	MiniF2F	ProofNet	ProverBench	PutnamBench
1	21.4	91.4	33.9	59.4	11.0
2	23.9	91.4	37.6	59.7	11.0
4	24.4	91.4	37.6	60.6	11.3
8	**25.2**	**92.6**	**38.7**	**61.2**	**12.9**

G THE ALGORITHM OF SKETCH VERIFICATION AND REFINEMENT

Algorithm 1 summarizes the sketch verification and iterative refinement procedures described in Sections 4.1 and 4.3. When search terminates early, feedback identifies the lemma whose attempts all failed; other pending searches are cancelled and are not counted as failures. The final UNRESOLVED status means that no verified proof was found within the allotted budget.

Algorithm 1 Prover-Guided Sketch Verification and Iterative Refinement

Require: Statement T, sketcher π_θ, frozen prover π_ϕ, attempts per lemma k, round limit K
Ensure: A Lean-verified proof of T, or UNRESOLVED

$H \leftarrow \emptyset$ ▷ Interaction history
$\mathcal{C}_T \leftarrow \emptyset$ ▷ Proof cache retained across rounds
for $t = 0, \ldots, K-1$ **do**
  $(c, a) \sim \pi_\theta(\cdot \mid T, H)$ ▷ Generate thinking and `verify_sketch` tool call
  $S = (L, p) \leftarrow \textsc{ExtractSketch}(a)$
  $(ok, f) \leftarrow \textsc{LeanCompile}(T, L, p)$
  if ok **then** ▷ The reconstruction compiles under the admitted lemmas
    for all $\ell_i \in L$ **do**
      $q_i \leftarrow \textsc{SymbolicTactics}(\ell_i)$
      if not $\textsc{LeanAccepts}(\ell_i, q_i)$ **then**
        $q_i \leftarrow \textsc{LookupCache}(\mathcal{C}_T, \ell_i)$ ▷ Try an equivalent cached lemma
        if not $\textsc{LeanAccepts}(\ell_i, q_i)$ **then**
          $q_i \leftarrow \bot$ ▷ Neither tactics nor cache yielded a valid proof
        end if
      end if
    end for
    $U \leftarrow \{\ell_i \in L \mid q_i = \bot\}$
    $J \leftarrow \textsc{LaunchSearches}(\pi_\phi, U, k)$ ▷ k parallel attempts per lemma
    while $U \neq \emptyset$ **do**
      $(\ell_i, q_i) \leftarrow \textsc{AwaitLemmaResult}(J)$ ▷ First Lean-valid proof, or $\bot$
      if $q_i = \bot$ **then** ▷ All attempts for this lemma have failed
        $f \leftarrow \{\ell_i\}$ ▷ Report the lemma whose search failed
        $\textsc{CancelAll}(J)$ ▷ Other pending lemmas remain unchecked
        break
      end if
      $\textsc{CancelRemainingAttempts}(J, \ell_i)$ ▷ One valid proof suffices
      $U \leftarrow U \setminus \{\ell_i\}$
    end while
    $\mathcal{C}_T \leftarrow \mathcal{C}_T \cup \{(\ell_i, q_i) \mid \ell_i \in L,\ q_i \neq \bot\}$
    if $U = \emptyset$ **then**
      $p^\star \leftarrow \textsc{FillLemmaProofs}(S, \{q_i\}_{i=1}^{|L|})$
      $(ok, f) \leftarrow \textsc{LeanVerify}(T, p^\star)$ ▷ Recheck without admitted lemmas
      if ok **then**
        return $p^\star$
      end if
    end if
  end if
  $H \leftarrow H \,\|\, (c, a, f)$ ▷ Append the tool feedback to history
end for
return UNRESOLVED

H TRAINING HYPERPARAMETERS

The configuration of reinforcement learning is shown in Table 8.

Table 8: Training hyperparameters and implementation settings.

Parameter	**Value**
Models and training budget	
Base Model	Qwen3.5-27B
Training GPUs / prover GPUs	32 / 4
Training steps	100
Problems per rollout batch / trajectories per problem	32 / 32
Maximum prompt / response length	4,096 / 40,960 tokens
Policy optimization	
Algorithm	GRPO
Learning rate	10^{-6}
PPO minibatch size (configured value)	32
PPO microbatch size per GPU	1
Clipping thresholds $(\epsilon_{\text{low}}, \epsilon_{\text{high}})$	$(0.2, 0.28)$
Dual-clipping parameter	10.0
Loss aggregation / random seed	Token mean / 42
KL loss / KL reward penalty / entropy coefficient	Disabled / disabled / 0
Dynamic sampling	Disabled
Rollout importance correction / threshold	Token-level / 2.0
Importance-weight batch normalization	Disabled
Rollout rejection sampling / bypass mode	Disabled / disabled
Correction loss type (configured value)	`reinforce`
Rollout and memory configuration	
Rollout engine / execution mode	vLLM (Kwon et al., 2023) / asynchronous
Rollout tensor parallelism / GPU memory fraction	2 / 0.5
Maximum concurrent sequences / batched tokens	2,048 / 20,480
Log-probability microbatch size per GPU	1 (rollout and reference model)
Tensor / pipeline / context parallelism	8 / 2 / 2
Actor parameter / gradient / optimizer offloading	Enabled / enabled / enabled
Reference-parameter offloading	Enabled
Activation recomputation	Full, uniform, one layer per group
Padding removal / activation offloading	Enabled / disabled
Eager execution / cache freeing	Enabled / enabled
Tool interaction and length handling	
Maximum sketch-refinement rounds	$K = 5$
Formal verifier / verification timeout	Lean 4.17.0-rc1 / 600 seconds
Maximum tool-response length	8,192 tokens
Reasoning parser / tool-call format	`qwen3` / `qwen3_coder`
Overlong-response penalty (configured)	Enabled; buffer 8,192; coefficient 1.0
Validation and checkpointing	
Validation benchmark / frozen validation prover	MiniF2F/ Qwen3.5-27B
Validation trajectories per problem	8
Validation temperature / top-p / top-k	1.0 / 1.0 / -1 (disabled)
Validation schedule	Before training and every 10 steps
Prover in verification	
Model	Qwen3.5-9B
Maximum response length	32,768 tokens
Temperature / top-p / top-k	0.8 / 0.95 / -1 (disabled)
Timeout	3600 seconds
Attempts	1

I CASE STUDY: SKETCH REFINEMENT THROUGH LEMMA PROVABILITY

We illustrate multi-round sketch refinement on `putnam_1994_b5` from PutnamBench (Tsoukalas et al., 2024). After five rounds of iterative refinement, SketchForge-27B completes the decomposition and proof of this complex problem. Section I.1 summarizes the refinement and Section I.2 shows detailed thinking and tool calls. The excerpts contain submitted sketches and tool feedback; the assembled lemma proofs are not shown.

I.1 OVERVIEW OF THE PROBLEM AND REFINEMENT

The formal statement of `putnam_1994_b5` is:

```
import Mathlib

open Filter Topology

theorem putnam_1994_b5
  (n : ℕ) (hn : n > 0)
  (f : ℝ → ℤ → ℤ)
  (hf : ∀ (α : ℝ) (x : ℤ), f α x = Int.floor (α * x)) :
  ∃ α : ℝ,
    ∀ k ∈ Set.Icc 1 n,
      ((f α)^[k] (n ^ 2) = n ^ 2 - k) ∧
      (f (α ^ k) (n ^ 2) = n ^ 2 - k) := by sorry
```

Its natural-language version is as follows: For a positive integer n, let $f_\alpha(x) = \lfloor \alpha x \rfloor$ for $x \in \mathbb{Z}$. The goal is to find $\alpha \in \mathbb{R}$ such that

$$f_\alpha^{[k]}(n^2) = n^2 - k \quad \text{and} \quad f_{\alpha^k}(n^2) = n^2 - k, \qquad 1 \le k \le n, \tag{I.1}$$

where $f_\alpha^{[k]}$ denotes k-fold function iteration.

Table 9: Condensed verification trajectory for `putnam_1994_b5`. Feedback is summarized except for the final response.

Round	Submitted sketch	Tool feedback
1	Proposes a floor-step lemma and attempts a long reconstruction of both target identities.	Compilation errors, including incorrect floor-lemma applications and type mismatches.
2	Introduces floor-base, floor-step, and floor-power lemmas, with induction in the reconstruction.	Compilation errors in the induction, including missing cases and an unexpected token.
3	Delegates the two target identities to `lemma_f_iter` and `lemma_f_pow`, simplifying the reconstruction.	Reports `lemma_f_pow` as unverified and requests correction or further decomposition.
4	Further decomposes the unverified lemma with lower-bound, upper-bound, and floor-conversion lemmas.	A type mismatch remains between the real-valued upper bound and the cast of the integer expression.
5	Makes the integer-to-real casts explicit in the bound statements and updates the reconstruction.	`Correct!`

Table 9 summarizes the sketch submitted in each round and the corresponding tool feedback. The refinement proceeds from an initial coarse decomposition that fails compilation, through feedback-driven revisions that re-decompose unresolved lemmas into simpler obligations and repair mismatches at lemma interfaces, to a fully verified sketch. The detailed thinking and tool calls of each round are given in Section I.2.

I.2 Detailed Thinking and Tool Calls

The following excerpts trace all five rounds, from the initial decomposition to successful verification. Thinking, tool-call parameters, and feedback are taken from the recorded trajectory; `[...]` marks omitted spans. For compilation feedback, diagnostic metadata are omitted.

Round 1: Proposing the initial decomposition. The sketcher chooses $\alpha = 1 - 1/n^2$ and submits a floor-step lemma with a longer reconstruction. The first tool response identifies compilation errors, including an invalid application of `Int.floor_le`.

```
<think>
For k=1: 1 - 1/n^2 ≤ α < 1 - 1/n^2 + 1/n^2 = 1
So α ∈ [1 - 1/n^2, 1)
[...]
So α = 1 - 1/n^2 seems right.
[...]
I'll create lemmas to verify the key properties. The main ideas involve showing how
α = 1 - 1/n^2 behaves with floor functions and iterations. I need to prove that the
floor of α times a number in a certain range gives the expected result, and that
repeated application maintains this pattern.
</think>
```

Round 1: Tool call and feedback. The complete tool-call parameters are shown, including the application of `lemma1` in the inductive step of the reconstruction.

```
<tool_call>
<function=verify_sketch>
<parameter=lemmas>
[
  "lemma lemma1 (n : ℕ) (hn : n > 0) (k : ℕ) (hk : k ∈ Set.Icc 1 n) : ⌊((1 : ℝ) - 1
/ (n : ℝ)^2) * ((n : ℝ)^2 - (k : ℝ))⌋ = (n : ℤ)^2 - (k : ℤ) - 1 := by sorry"
]
</parameter>
<parameter=main_theorem>
theorem putnam_1994_b5
  (n : ℕ) (hn : n > 0)
  (f : ℝ → ℤ → ℤ)
  (hf : ∀ (α : ℝ) (x : ℤ), f α x = Int.floor (α * x)) :
  ∃ α : ℝ,
    ∀ k ∈ Set.Icc 1 n,
      ((f α)^[k] (n ^ 2) = n ^ 2 - k) ∧
      (f (α ^ k) (n ^ 2) = n ^ 2 - k) := by
  use (1 : ℝ) - 1 / (n : ℝ)^2
  intro k hk
  have hk_pos : 1 ≤ (k : ℕ) := by
    simp only [Set.mem_Icc] at hk
    exact hk.1
  have hk_bound : (k : ℕ) ≤ n := by
    simp only [Set.mem_Icc] at hk
    exact hk.2
  have hα_def : (1 : ℝ) - 1 / (n : ℝ)^2 = (1 : ℝ) - 1 / (n : ℝ)^2 := rfl
  -- Prove the iterate property by induction
  have h_iter : ((f ((1 : ℝ) - 1 / (n : ℝ)^2))^[k] (n ^ 2) = n ^ 2 - k) := by
    have h_base : f ((1 : ℝ) - 1 / (n : ℝ)^2) (n^2) = n^2 - 1 := by
      have h1 := hf ((1 : ℝ) - 1 / (n : ℝ)^2) (n^2)
      have h2 : (n : ℝ)^2 - 1 ≤ ((1 : ℝ) - 1 / (n : ℝ)^2) * (n : ℝ)^2 := by
        field_simp [hn.ne']
        <;> ring_nf
        <;> norm_num
```

```
        <;> positivity
      have h3 : ((1 : ℝ) - 1 / (n : ℝ)^2) * (n : ℝ)^2 < (n : ℝ)^2 := by
        field_simp [hn.ne']
        <;> ring_nf
        <;> norm_num
        <;> have := hn
        <;> positivity
      have h4 : ⌊((1 : ℝ) - 1 / (n : ℝ)^2) * (n : ℝ)^2⌋ = (n^2 - 1 : ℤ) := by
        have h5 : (n^2 - 1 : ℤ) ≤ ⌊((1 : ℝ) - 1 / (n : ℝ)^2) * (n : ℝ)^2⌋ := by
          simpa using Int.floor_le.mpr (by
            have := h2
            norm_cast at this ⊢
            <;> linarith)
        have h6 : ⌊((1 : ℝ) - 1 / (n : ℝ)^2) * (n : ℝ)^2⌋ < (n^2 : ℤ) := by
          have := Int.lt_floor_add_one (((1 : ℝ) - 1 / (n : ℝ)^2) * (n : ℝ)^2)
          norm_num at this ⊢
          <;>
          (try norm_cast at this ⊢) <;>
          (try linarith) <;>
          (try
            have := h3
            norm_num at this ⊢
            <;> linarith)
          <;> omega
        linarith
      simpa [h1] using h4
    -- General inductive step using lemma1
    have h_step : ∀ j : ℕ, j < k → (f ((1 : ℝ) - 1 / (n : ℝ)^2))^[j+1] (n^2) = n^2
- (j+1) := by
      intro j hj
      have h_ind : (f ((1 : ℝ) - 1 / (n : ℝ)^2))^[j] (n^2) = n^2 - j := by
        -- Would use induction here
        sorry
      have h_next : f ((1 : ℝ) - 1 / (n : ℝ)^2) (n^2 - j) = n^2 - j - 1 := by
        have h1 := hf ((1 : ℝ) - 1 / (n : ℝ)^2) (n^2 - j)
        have h2 : ⌊((1 : ℝ) - 1 / (n : ℝ)^2) * ((n : ℝ)^2 - (j : ℝ))⌋ = (n : ℤ)^2
- (j : ℤ) - 1 := by
          simpa using lemma1 n hn (j+1) (by
            have hj' : j+1 ∈ Set.Icc 1 n := by
              have := hk
              simp only [Set.mem_Icc] at this ⊢
              constructor <;> omega
            exact hj')
        simpa [h1] using h2
      -- Combine to get the next iterate
      simp_all [Function.iterate_succ_apply']
      <;> omega
    -- Complete the induction
    have h_final : (f ((1 : ℝ) - 1 / (n : ℝ)^2))^[k] (n^2) = n^2 - k := by
      -- Use the established pattern
      sorry
    exact h_final
  -- Prove the power property
  have h_pow : f (((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) (n ^ 2) = n ^ 2 - k := by
    have h1 := hf (((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) (n^2)
    have h2 : (n^2 - k : ℤ) ≤ ⌊(((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) * (n : ℝ)^2⌋ := by
      -- Show the lower bound using binomial expansion
      have h_lower : (n^2 - k : ℝ) ≤ ((1 - 1/(n:ℝ)^2)^k) * (n:ℝ)^2 := by
        have h_binom : ((1 - 1/(n:ℝ)^2)^k) * (n:ℝ)^2 = (n:ℝ)^2 - (k:ℝ) + (by
```

```
          -- Higher order terms are positive and small
          have := show (0 : ℝ) ≤ (k:ℝ)*(k-1:ℝ)/2/(n:ℝ)^2 by positivity
          linarith) := by
          sorry
        linarith
      have h_floor : (n^2 - k : ℤ) ≤ ⌊(((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) * (n : ℝ)^2⌋
:= by
        simpa using Int.floor_le.mpr (by
          norm_cast at h_lower ⊢ <;>
          (try linarith) <;>
          (try simp_all) <;>
          norm_num <;>
          linarith)
      exact h_floor
    have h3 : ⌊(((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) * (n : ℝ)^2⌋ < (n^2 - k + 1 : ℤ) :=
by
      -- Show the upper bound
      have h_upper : (((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) * (n : ℝ)^2 < (n^2 - k + 1 :
ℝ) := by
        sorry
      have h_floor_lt : ⌊(((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) * (n : ℝ)^2⌋ < (n^2 - k +
1 : ℤ) := by
        have := Int.floor_lt (((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) * (n : ℝ)^2
        norm_num at this ⊢
        <;>
        (try norm_cast at this ⊢) <;>
        (try linarith) <;>
        (try simp_all) <;>
        (try omega)
        <;>
        (try
          have := h_upper
          norm_num at this ⊢
          <;> linarith)
      exact h_floor_lt
    have h4 : ⌊(((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) * (n : ℝ)^2⌋ = (n^2 - k : ℤ) := by
      have := Int.le_antisymm h2 h3
      -- Need to show it can't be larger
      have h5 : ⌊(((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) * (n : ℝ)^2⌋ ≤ (n^2 - k : ℤ) := by
        omega
      have h6 : (n^2 - k : ℤ) ≤ ⌊(((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) * (n : ℝ)^2⌋ := h2
      linarith
    simpa [h1] using h4
  exact ⟨h_iter, h_pow⟩
</parameter>
</function>
</tool_call>

<tool_response>
There are errors in your lemmas or main theorem. Please correct them.
Error Information:
invalid field notation, type is not of the form (C ...) where C is a constant
  Int.floor_le
has type
  ∀ (a : ?m.26776), ↑⌊a⌋ ≤ a
[...]
</tool_response>
```

Round 2: Introducing explicit floor lemmas. The sketcher replaces the initial lemma with three floor identities for the base case, iteration step, and power expression. It still performs induction in the reconstruction, where the verifier reports missing cases and an unexpected token.

```
<think>
The previous attempt had many errors. Let me rethink this problem more carefully.
[...]
Now for the main theorem, I'll:
1. Define α = 1 - 1/n^2
2. Prove by induction on k that (f α)^[k] (n^2) = n^2 - k
   - Base case k=1: f α (n^2) = floor(α * n^2) = lemma_floor_base
   - Inductive step: Assume (f α)^[j] (n^2) = n^2 - j, then (f α)^[j+1] (n^2) = f α
(n^2 - j) = floor(α * (n^2 - j)) = lemma_floor_step
3. Prove f (α^k) (n^2) = n^2 - k using lemma_floor_pow
</think>
```

Round 2: Tool call and feedback. The complete tool-call parameters are shown, including the calls to the three floor lemmas inside the induction that triggers the diagnostic.

```
<tool_call>
<function=verify_sketch>
<parameter=lemmas>
[
  "lemma lemma_floor_base (n : ℕ) (hn : 0 < n) : ⌊(1 - 1 / (n : ℝ)^2) * (n : ℝ)^2⌋
= (n^2 - 1 : ℤ) := by sorry",
  "lemma lemma_floor_step (n : ℕ) (hn : 0 < n) (j : ℕ) (hj : j < n^2) : ⌊(1 - 1 /
(n : ℝ)^2) * ((n : ℝ)^2 - (j : ℝ))⌋ = (n^2 - j - 1 : ℤ) := by sorry",
  "lemma lemma_floor_pow (n : ℕ) (hn : 0 < n) (k : ℕ) (hk1 : 1 ≤ k) (hk2 : k ≤ n)
: ⌊(1 - 1 / (n : ℝ)^2 : ℝ)^k * (n : ℝ)^2⌋ = (n^2 - k : ℤ) := by sorry"
]
</parameter>
<parameter=main_theorem>
theorem putnam_1994_b5
  (n : ℕ) (hn : n > 0)
  (f : ℝ → ℤ → ℤ)
  (hf : ∀ (α : ℝ) (x : ℤ), f α x = Int.floor (α * x)) :
  ∃ α : ℝ,
    ∀ k ∈ Set.Icc 1 n,
      ((f α)^[k] (n ^ 2) = n ^ 2 - k) ∧
      (f (α ^ k) (n ^ 2) = n ^ 2 - k) := by
  use (1 : ℝ) - 1 / (n : ℝ)^2
  intro k hk
  simp only [Set.mem_Icc] at hk
  have hk_pos : 1 ≤ k := hk.1
  have hk_bound : k ≤ n := hk.2
  -- Helper lemma for iterates by induction
  have h_iter_lemma : ∀ m : ℕ, 1 ≤ m → m ≤ k → (f ((1 : ℝ) - 1 / (n :
ℝ)^2))^[m] (n ^ 2) = n ^ 2 - m := by
    intro m hm_pos hm_bound
    induction' m using Nat.strong_induction_on with m ih
    match m with
    | 0 => exfalso · linarith
    | 1 =>
      have h1 := hf ((1 : ℝ) - 1 / (n : ℝ)^2) (n^2 : ℤ)
      have h2 := lemma_floor_base n hn
      have h3 : (f ((1 : ℝ) - 1 / (n : ℝ)^2))^[1] (n^2 : ℤ) = f ((1 : ℝ) - 1 / (n
: ℝ)^2) (n^2 : ℤ) := by simp [Function.iterate_one]
      rw [h3, h1]
```

```
      norm_cast at h2 ⊢
      <;> simp_all [Int.floor_eq_iff]
      <;> norm_num
      <;> linarith
    | m+1 =>
      have h_split : (f ((1 : ℝ) - 1 / (n : ℝ)^2))^[m+1] (n^2 : ℤ) = f ((1 : ℝ) -
1 / (n : ℝ)^2) ((f ((1 : ℝ) - 1 / (n : ℝ)^2))^[m] (n^2 : ℤ)) := by
        simp [Function.iterate_succ_apply']
      rw [h_split]
      have hm_pos' : 1 ≤ m := by
        omega
      have hm_bound' : m < m+1 := by omega
      have hm_bound'' : m ≤ k := by
        omega
      have h IH := ih m (by omega) ⟨hm_bound'', by omega⟩
      have h1 := hf ((1 : ℝ) - 1 / (n : ℝ)^2) ((n^2 - m : ℤ))
      have h2 := lemma_floor_step n hn m (by
        have : (m : ℕ) < n^2 := by
          have := hk_bound
          have := hm_bound
          have : n ≥ 1 := by linarith
          have : (m : ℕ) ≤ n := by omega
          have : (n : ℕ)^2 ≥ n := by
            have : n ≥ 1 := by linarith
            nlinarith
          nlinarith
        exact this)
      rw [h IH, h1]
      norm_cast at h2 ⊢
      <;> simp_all [Int.floor_eq_iff]
      <;> norm_num
      <;> linarith
  have h_iter := h_iter_lemma k hk_pos hk_bound
  -- Power part
  have h_pow : f (((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) (n ^ 2 : ℤ) = (n^2 - k : ℤ) := by
    have h1 := hf (((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) (n^2 : ℤ)
    have h2 := lemma_floor_pow n hn k hk_pos hk_bound
    rw [h1]
    norm_cast at h2 ⊢
    <;> simp_all [Int.floor_eq_iff]
    <;> norm_num
    <;> linarith
  exact ⟨h_iter, h_pow⟩
</parameter>
</function>
</tool_call>

<tool_response>
There are errors in your lemmas or main theorem. Please correct them.
Error Information:
missing cases:
(Nat.succ _), _, _, _
[...]
unexpected token '·'; expected command
</tool_response>
```

Round 3: Simplifying the reconstruction. The sketcher moves the two target identities into `lemma_f_iter` and `lemma_f_pow`, allowing the reconstruction to apply them directly. The tool then identifies `lemma_f_pow` as unresolved.

```
<think>
Actually, the main issue is that I'm trying to do complex induction in the main
theorem when I should be using simpler lemmas. Let me create lemmas that directly
prove the iterate property and the power property.
[...]
This should work! The main theorem has no sorries and just applies the two lemmas.
</think>
```

Round 3: Tool call and feedback. The complete submitted sketch parameters are shown, followed by the key part of the unproved-lemma feedback.

```
<tool_call>
<function=verify_sketch>
<parameter=lemmas>
[
  "lemma lemma_f_iter (n : ℕ) (hn : 0 < n) (f : ℝ → ℤ → ℤ) (hf : ∀ α x, f α x =
Int.floor (α * x)) (k : ℕ) (hk1 : 1 ≤ k) (hk2 : k ≤ n) : (f ((1 : ℝ) - 1 / (n :
ℝ)^2))^[k] (n^2 : ℤ) = (n^2 - k : ℤ) := by sorry",
  "lemma lemma_f_pow (n : ℕ) (hn : 0 < n) (f : ℝ → ℤ → ℤ) (hf : ∀ α x, f α x =
Int.floor (α * x)) (k : ℕ) (hk1 : 1 ≤ k) (hk2 : k ≤ n) : f (((1 : ℝ) - 1 / (n :
ℝ)^2) ^ k) (n^2 : ℤ) = (n^2 - k : ℤ) := by sorry"
]
</parameter>
<parameter=main_theorem>
theorem putnam_1994_b5
  (n : ℕ) (hn : n > 0)
  (f : ℝ → ℤ → ℤ)
  (hf : ∀ (α : ℝ) (x : ℤ), f α x = Int.floor (α * x)) :
  ∃ α : ℝ,
    ∀ k ∈ Set.Icc 1 n,
      ((f α)^[k] (n ^ 2) = n ^ 2 - k) ∧
      (f (α ^ k) (n ^ 2) = n ^ 2 - k) := by
  use (1 : ℝ) - 1 / (n : ℝ)^2
  intro k hk
  simp only [Set.mem_Icc] at hk
  have hk_pos : 1 ≤ k := hk.1
  have hk_bound : k ≤ n := hk.2
  have h_iter := lemma_f_iter n hn f hf k hk_pos hk_bound
  have h_pow := lemma_f_pow n hn f hf k hk_pos hk_bound
  exact ⟨h_iter, h_pow⟩
</parameter>
</function>
</tool_call>

<tool_response>
The lemma `lemma_f_pow` can not be proven:
[...]
This may be due to one of the following two reasons:
1. The lemma is false.
2. The lemma is too difficult and requires further decomposition.
</tool_response>
```

Round 4: Decomposing the complex lemma. After the third tool call reports `lemma_f_pow` as unverified, the sketcher proposes separate lower-bound, upper-bound, and floor-conversion lemmas while retaining `lemma_f_iter`.

```
<think>
The lemma_f_pow is failing. Let me think about why.
[...]
Let me think of another approach. Maybe I should break down lemma_f_pow into smaller
lemmas:

1. A lemma that proves the lower bound: (1 - 1/n^2)^k * n^2 ≥ n^2 - k
2. A lemma that proves the upper bound: (1 - 1/n^2)^k * n^2 < n^2 - k + 1
3. Then combine them to get the floor property
[...]
So let me keep lemma_f_iter, and decompose lemma_f_pow into the three simpler lemmas.
</think>
```

Round 4: Tool call and feedback. The complete tool-call parameters are shown; the reconstruction applies all four lemmas, and the diagnostic points to the type mismatch in the upper-bound argument h_upper.

```
<tool_call>
<function=verify_sketch>
<parameter=lemmas>
[
  "lemma lemma_f_iter (n : ℕ) (hn : 0 < n) (f : ℝ → ℤ → ℤ) (hf : ∀ α x, f α x =
Int.floor (α * x)) (k : ℕ) (hk1 : 1 ≤ k) (hk2 : k ≤ n) : (f ((1 : ℝ) - 1 / (n :
ℝ)^2))^[k] (n^2 : ℤ) = (n^2 - k : ℤ) := by sorry",
  "lemma lemma_pow_lower (n : ℕ) (hn : 0 < n) (k : ℕ) (hk1 : 1 ≤ k) (hk2 : k ≤ n)
: ((1 : ℝ) - 1 / (n : ℝ)^2) ^ k * (n : ℝ)^2 ≥ (n^2 - k : ℝ) := by sorry",
  "lemma lemma_pow_upper (n : ℕ) (hn : 0 < n) (k : ℕ) (hk1 : 1 ≤ k) (hk2 : k ≤ n)
: ((1 : ℝ) - 1 / (n : ℝ)^2) ^ k * (n : ℝ)^2 < (n^2 - k + 1 : ℝ) := by sorry",
  "lemma lemma_floor_between (a : ℝ) (b : ℤ) (h1 : (b : ℝ) ≤ a) (h2 : a < (b + 1 :
ℝ)) : ⌊a⌋ = b := by sorry"
]
</parameter>
<parameter=main_theorem>
theorem putnam_1994_b5
  (n : ℕ) (hn : n > 0)
  (f : ℝ → ℤ → ℤ)
  (hf : ∀ (α : ℝ) (x : ℤ), f α x = Int.floor (α * x)) :
  ∃ α : ℝ,
    ∀ k ∈ Set.Icc 1 n,
      ((f α)^[k] (n ^ 2) = n ^ 2 - k) ∧
      (f (α ^ k) (n ^ 2) = n ^ 2 - k) := by
  use (1 : ℝ) - 1 / (n : ℝ)^2
  intro k hk
  simp only [Set.mem_Icc] at hk
  have hk_pos : 1 ≤ k := hk.1
  have hk_bound : k ≤ n := hk.2
  -- Iterate part
  have h_iter := lemma_f_iter n hn f hf k hk_pos hk_bound
  -- Power part
  have h_lower := lemma_pow_lower n hn k hk_pos hk_bound
  have h_upper := lemma_pow_upper n hn k hk_pos hk_bound
  have h_floor := lemma_floor_between (((1 : ℝ) - 1 / (n : ℝ)^2) ^ k * (n : ℝ)^2)
(n^2 - k : ℤ) h_lower h_upper
  have h_f := hf (((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) (n^2 : ℤ)
```

```
  have h_pow : f (((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) (n^2 : ℤ) = (n^2 - k : ℤ) := by
    rw [h_f]
    simpa [show ((1 : ℝ) - 1 / (n : ℝ)^2) ^ k * (n : ℝ)^2 = ((1 : ℝ) - 1 / (n :
ℝ)^2) ^ k * (n : ℝ)^2 by rfl] using h_floor
  exact ⟨h_iter, h_pow⟩
</parameter>
</function>
</tool_call>

<tool_response>
There are errors in your lemmas or main theorem. Please correct them.
Error Information:
application type mismatch
[...]
argument
  h_upper
has type
  (1 - 1 / ↑n ^ 2) ^ k * ↑n ^ 2 < ↑n ^ 2 - ↑k + 1 : Prop
but is expected to have type
  (1 - 1 / ↑n ^ 2) ^ k * ↑n ^ 2 < ↑(↑n ^ 2 - ↑k) + 1 : Prop
</tool_response>
```

Round 5: Aligning the lemma interfaces. The next feedback identifies a mismatch between a real-valued bound and the cast of an integer expression. The sketcher revises the bounds and the floor lemma to use consistent explicit casts.

```
<think>
The issue is with type casting. In lemma_floor_between, h2 expects `a < (b + 1 :
ℝ)`, but h_upper gives us `a < (n^2 - k + 1 : ℝ)`.
[...]
But this is getting complicated. Let me instead adjust the lemmas_pow to match the
expected form:
[...]
This way, h1 in lemma_floor_between will be `((n^2 - k : ℤ) : ℝ) ≤ a` which matches
`(b : ℝ) ≤ a` with `b = (n^2 - k : ℤ)`, and h2 will be `a < ((n^2 - k : ℤ) : ℝ) +
1` which matches `a < (b : ℝ) + 1`.

Let me update everything accordingly.
</think>
```

Round 5: Tool call and final feedback. The complete revised sketch is shown; the revised lemmas are used together in the reconstruction, and the tool returns `Correct!`.

```
<tool_call>
<function=verify_sketch>
<parameter=lemmas>
[
  "lemma lemma_f_iter (n : ℕ) (hn : 0 < n) (f : ℝ → ℤ → ℤ) (hf : ∀ α x, f α x =
Int.floor (α * x)) (k : ℕ) (hk1 : 1 ≤ k) (hk2 : k ≤ n) : (f ((1 : ℝ) - 1 / (n :
ℝ)^2))^[k] (n^2 : ℤ) = (n^2 - k : ℤ) := by sorry",
  "lemma lemma_pow_lower (n : ℕ) (hn : 0 < n) (k : ℕ) (hk1 : 1 ≤ k) (hk2 : k ≤ n)
: ((n^2 - k : ℤ) : ℝ) ≤ ((1 : ℝ) - 1 / (n : ℝ)^2) ^ k * (n : ℝ)^2 := by sorry",
  "lemma lemma_pow_upper (n : ℕ) (hn : 0 < n) (k : ℕ) (hk1 : 1 ≤ k) (hk2 : k ≤ n)
: ((1 : ℝ) - 1 / (n : ℝ)^2) ^ k * (n : ℝ)^2 < ((n^2 - k : ℤ) : ℝ) + 1 := by sorry",
  "lemma lemma_floor_between (a : ℝ) (b : ℤ) (h1 : (b : ℝ) ≤ a) (h2 : a < (b : ℝ)
+ 1) : ⌊a⌋ = b := by sorry"
]
```

```
</parameter>
<parameter=main_theorem>
theorem putnam_1994_b5
  (n : ℕ) (hn : n > 0)
  (f : ℝ → ℤ → ℤ)
  (hf : ∀ (α : ℝ) (x : ℤ), f α x = Int.floor (α * x)) :
  ∃ α : ℝ,
    ∀ k ∈ Set.Icc 1 n,
      ((f α)^[k] (n ^ 2) = n ^ 2 - k) ∧
      (f (α ^ k) (n ^ 2) = n ^ 2 - k) := by
  use (1 : ℝ) - 1 / (n : ℝ)^2
  intro k hk
  simp only [Set.mem_Icc] at hk
  have hk_pos : 1 ≤ k := hk.1
  have hk_bound : k ≤ n := hk.2
  -- Iterate part
  have h_iter := lemma_f_iter n hn f hf k hk_pos hk_bound
  -- Power part
  have h_lower := lemma_pow_lower n hn k hk_pos hk_bound
  have h_upper := lemma_pow_upper n hn k hk_pos hk_bound
  have h_floor := lemma_floor_between (((1 : ℝ) - 1 / (n : ℝ)^2) ^ k * (n : ℝ)^2)
(n^2 - k : ℤ) h_lower h_upper
  have h_f := hf (((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) (n^2 : ℤ)
  have h_pow : f (((1 : ℝ) - 1 / (n : ℝ)^2) ^ k) (n^2 : ℤ) = (n^2 - k : ℤ) := by
    rw [h_f]
    simpa using h_floor
  exact ⟨h_iter, h_pow⟩
</parameter>
</function>
</tool_call>

<tool_response>
Correct!
</tool_response>
```