WybeCoder: Verified Imperative Code Generation

Fabian Gloeckle [*1,2] **Mantas Bakšys** [*1,3] **Darius Feher** [1,4] **Kunhao Zheng** [1,5] **Amaury Hayat** [2,6] **Sean B. Holden** [3]
Gabriel Synnaeve [1] **Peter O'Hearn** [1,4]

Abstract

Recent progress in large language models (LLMs) has advanced automatic code generation and formal theorem proving, yet software verification has not seen the same improvement. To address this gap, we propose WybeCoder, an agentic code verification framework that enables *prove-as-you-generate* development where code, invariants, and proofs co-evolve. It builds on a recent framework that combines automatic verification condition generation and SMT solvers with interactive proofs in Lean. To enable systematic evaluation, we translate two benchmarks for functional verification in Lean – Verina and Clever – to equivalent imperative code specifications. On complex algorithms such as Heapsort, we observe consistent performance improvements by scaling our approach, synthesizing dozens of valid invariants and dispatching dozens of subgoals, resulting in hundreds of lines of verified code, overcoming plateaus reported in previous works. Our best system solves 74% of Verina tasks and 62% of Clever tasks at moderate compute budgets, significantly surpassing previous evaluations and paving a path to automated construction of large-scale datasets of verified imperative code.

1. Introduction

Large language models (LLMs) have made impressive progress in recent years in two related fields: code generation and interactive theorem proving. In both, they have advanced from toy problems in programming and mathematics to rivaling top human performance in prestigious competitions such as the IOI and the IMO (Hubert et al., 2025; Li et al., 2022).

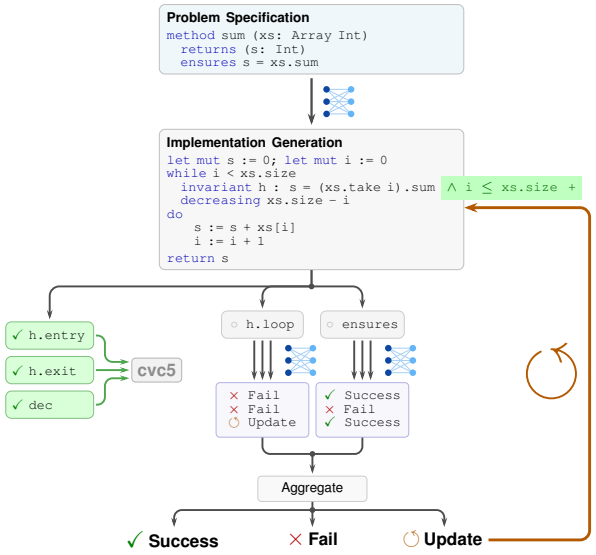


Figure 1. WybeCoder: Subgoal decomposition multi-agent system. Starting from a problem specification, the agent generates an implementation and attempts to discharge verification conditions using CVC5. Remaining goals are tackled interactively in Lean, driving iterative implementation refinement or completing the proof.

In software engineering, this progress has resulted in wide adoption of LLMs to speed-up code generation but still requires human review to ensure safety and quality standards. While scaling up code-generation is straightforward, reviewing remains time-consuming and labour intensive. This discrepancy exposes the need for better automated verification tools. Already commonly used approaches in verification include automated unit testing and fuzzing but provide only partial assurance by exposing defects (see Appendix A). Strong end-to-end guarantees require formal software verification: expressing intended behavior as a formal specification and constructing a machine-checked proof that the implementation satisfies it. For theorem proving, language models coupled with interactive theorem provers now surpass the performance of heuristic-guided automated theorem provers and SMT solvers (Hubert et al., 2025; Achim et al., 2025).

Existing works on LLMs in software verification have focused on two separate approaches: (a) verification of functional programs in interactive theorem provers, which lever-

[*]Equal contribution [1]FAIR, Meta [2]CERMICS, ENPC, Institut Polytechnique de Paris, CNRS [3]Computer Lab, University of Cambridge [4]University College London [5]Miles, LAMSADE, Université Paris-Dauphine-PSL, Paris [6]Korea Institute for Advanced Study. Correspondence to: Fabian Gloeckle <fgloeckle@meta.com>.

Preprint. April 16, 2026.

ages the impressive theorem proving capabilities of LLMs. However, this misses the reality of most existing software relying on imperative programming (Scott & Aldrich, 2025) that involves mutable state and side effects, and (b) verification of imperative code using "auto-active" environments such as Dafny (Leino, 2010), where agents generate ghost-code annotations for the program and SMT solvers handle the remainder of the proof, which matches the utility of imperative code but misses opportunities presented by LLMs as theorem provers.

In this work, we address this gap by developing a framework for verified imperative code generation in a *hybrid* system where the agent recursively verifies and updates the generated code using a combination of SMT solvers and interactive prover agents in Lean (de Moura & Ullrich, 2021), enabled by recent software verification frameworks such as Loom (Gladshtein et al., 2026). Our framework, WybeCoder, is named in homage to Edsger Wybe Dijkstra, whose pioneering work in verification emphasized program and proof generation being done together (Dijkstra, 1972), illustrated with imperative programs (Dijkstra, 1976).

At small inference budgets, our work substantially outperforms prior approaches, improving on the best Verina result from 18% (Ye et al., 2025) to 55% with 32 refinement attempts. Our best results are state of the art, reaching 74% on Verina and 62% on Clever-Loom. Qualitatively, we observe that WybeCoder successfully implements and verifies in-place Selection Sort, Kadane's algorithm, and Heapsort, but makes only partial progress on recursive Quicksort. Overall, our work marks a significant step forward in agentic software synthesis, moving us closer to scalable generation and verification of imperative code.

Our **contributions** can be summarized as follows:

- We investigate best-of-both-worlds **hybrid software verification frameworks** as an alternative to fully interactive verification (e.g. in Lean) and auto-active verification (e.g. in Dafny) **for LLM-based verification**, and introduce the first translated benchmarks tailored to such environments (Section 3).
- We develop and study multiple workflows for large-scale inference on such software verification tasks, exhibiting **subgoal decomposition and goal-directed modification** as key components (Sections 4 and 5).
- Our developed system continues to make progress on challenging problems such as the verification of Heapsort, using hundreds of LLM calls, routinely managing dozens of subgoals and multiple hundreds of lines of code (see example in Listing 4).

Our code is publicly available at facebookresearch/wybecoder.

2. Background

2.1. Formal Software Verification

Formal software verification is the task of proving correctness properties about code. Depending on the type of code and the correctness properties sought, various calculi and frameworks for formal software verification exist.

Functional correctness properties are typically easiest to state and prove in functional programming languages. This is because computations in functional programming languages are *pure* mathematical functions uniquely defined by their input-output behavior without side effects to track in a proof. Correctness properties can be stated as mathematical theorems about these functions and can typically be proved with induction proofs over the inductive data types used in functional programming such as lists and trees. Interactive theorem provers such as Rocq (Team, 2024) and Lean are built upon such functional foundations and support function definitions in functional style, correctness theorems and native execution or code extraction. For example, the fact that list reversion is self-inverse could be stated in Rocq as follows as a mathematical property of a pure total function `rev`:

```
Lemma rev_involutive : forall (A : Type)
  (l : list A), rev (rev l) = l.
```

Functional programming often incurs performance overheads compared to imperative approaches when solving equivalent tasks. This disparity arises because imperative code typically leverages arrays, which provide constant-time access to arbitrary elements. In contrast, functional code relies on immutable data structures such as lists and trees, resulting in linear and logarithmic access times, respectively. An instance and further explanation of this phenomenon is referred to as the "Log N Penalties in Functional Programming" in a standard reference on the formal verification of functional algorithms (Appel, 2024).

For verifying imperative software, mutable state must be tracked explicitly in correctness proofs. This is performed by the *Hoare logic* calculus (Hoare, 1969), where judgements are expressed as triples

$$\{P\}\ C\ \{Q\}$$

where P is a pre-condition that holds before the command, C the command to run and Q a post-condition that holds after the command. Inference rules explain how Hoare triples can be derived for composed constructs such as sequences of commands, `if` clauses or `while` loops. For example, if invariant I is maintained by a `while` loop body, then after the loop the loop condition is false and the invariant still holds:

$$\frac{\{\, I \wedge B \,\}\ C\ \{\, I \,\}}{\{\, I \,\}\ \text{while}\ B\ \text{do}\ C\ \{\, I \wedge \neg B \,\}}$$

Weakest pre-condition calculus (Dijkstra, 1976) derives, for every program construct except `while` loops, the weakest pre-condition under which a given postcondition is guaranteed to hold. This reduces software verification to three main tasks: providing specifications *(spec engineering)*, supplying sufficiently strong loop invariants *(proof-hint synthesis)*, and proving that these invariants are preserved by the loop *(proof engineering)*. Many verification frameworks delegate the first two tasks to the user, while discharging the resulting (often routine) verification conditions with off-the-shelf SMT solvers such as Z3 (de Moura & Bjørner, 2008) or CVC5 (Barbosa et al., 2022). This approach is exemplified by systems such as Dafny (Leino, 2010) or the Frama-C WP plugin (Cuoq et al., 2012).

While we also take termination measures into account *(total correctness)*, extensions to Hoare logic that explicitly model heap memory state (*separation logic,* Reynolds, 2002) or shared memory in concurrent settings (O'Hearn, 2007) are beyond the scope of this work.

2.2. Hybrid Verification Frameworks

A major drawback of SMT-based software verification is the opacity of automated provers. Failures are uninterpretable and provide little guidance on whether and how to refine the invariant annotations, resulting in a challenging proof debugging phase (Mugnier et al., 2025b).

To address this, we adopt a *hybrid* software verification framework that bridges the gap between automated verification condition generation and interactive theorem proving. This approach allows users to revert to manual proofs in Lean or Rocq when automated solvers reach their limits. Specifically, we employ the Loom/Velvet system (Gladshtein et al., 2026) integrated within the Lean interactive theorem prover, which processes imperative syntax and invariant annotations into proof obligations. These obligations are then discharged via the CVC5 prover or, if necessary, through manual step-by-step verification. An example of a Velvet proof is given in Listing 1.

3. Evaluating Imperative Code Verification

Existing benchmarks for verified code generation target functional Lean, specifically Verina (Ye et al., 2025) and Clever (Thakur et al., 2025). Both remain challenging for frontier models: on Verina, Claude Opus 4.5 achieves only 10% proof success rate with 16 refinement attempts (Zhang, 2026), o4-mini solving 22.2% of proof tasks with 32 refinement attempts on Verina-advanced and all models evaluated on Clever solve under 9% of the proof tasks. We translate both benchmarks to equivalent specifications in Velvet to evaluate imperative code synthesis and verification tasks. Due to the benchmark set-up and our translation methodology (see Appendix B), our evaluation results are directly comparable on Verina but not on Clever. We denote the translated version of the Clever benchmark Clever-Loom.

Listing 1. Velvet's hybrid approach: automated verification via `loom_solve` and interactive proofs. GPT-5 on `verina_basic_43`.

```
method sumOfFourthPowerOfOddNumbers (n : Nat) return
    (result : Nat)
  ensures 15 * result = n * (2 * n + 1) * (7 + 24 *
    n^3 - 12 * n^2 - 14 * n)
  do
    -- Accumulate S_i = sum_{k=0}^{i-1} (2k+1)^4
    using a loop
    let mut i : Nat := 0
    let mut acc : Nat := 0
    while i < n
      invariant h_i_le : i ≤ n
      -- Closed-form invariant for the partial sum
     (factored, avoiding powers)
      -- 15 * acc = i * (2*i - 1) * (2*i + 1) *
     (12*i*i - 7)
      invariant h_acc_closed : 15 * acc = i * (2 * i
     - 1) * (2 * i + 1) * (12 * i * i - 7)
      done_with h_done : n ≤ i
      decreasing h_dec : n - i
    do
      let odd : Nat := 2 * i + 1
      acc := acc + odd * odd * odd * odd
      i := i + 1
    return acc

lemma mul_factor_identity_nat (n : ℕ) :
  n * (2 * n - 1) * (2 * n + 1) * (12 * n * n - 7)
    = n * (2 * n + 1) * (7 + 24 * n ^ 3 - 12 * n ^ 2
     - 14 * n) := by
sorry  -- 82 line proof omitted here

prove_correct sumOfFourthPowerOfOddNumbers by
  loom_solve
  case «ensures» => (
  cases i_3
  have hi_eq : i = n := Nat.le_antisymm h_i_le h_done
  have h1 :
      15 * acc
        = n * (2 * n - 1) * (2 * n + 1) * (12 * n * n
    - 7) := by
    simpa [hi_eq] using h_acc_closed
  exact h1.trans (mul_factor_identity_nat n)
  )
```

3.1. Specification Leakage

As observed by Thakur et al. (2025), especially for easier problems such as several of the considered benchmark problems, it can happen that the specification itself contains a computable function that determines the expected output. In such cases, the implementation can be retrieved from the specification and the proof reduces to reflexivity. In other words, *specification leakage* renders the task trivial. An example for this behavior taken from one of our experiments is given below. This issue is common in the Verina benchmark, which complicates the interpretation of evaluation results[1]. We direct the reader to Section 5.4 for a discussion around mitigating specification leakage in our evaluations.

[1] Harmonic (2025) has reported 96.8% performance on Verina. Without controlling for specification leakage and without being able to examine the results, it remains unclear to what extent such results reflect genuine advances in code synthesis

Listing 2. Example of specification leakage on `verina_basic_57`

```
method CountLessThan (numbers : Array Int) (threshold
    : Int) return (result : Nat)
  ensures result = numbers.foldl (fun count n => if n
    < threshold then count + 1 else count) 0 = 0 ∧
    numbers.foldl (fun count n => if n < threshold
    then count + 1 else count) 0 = result = 0
  do
    let res := numbers.foldl (fun count n => if n <
    threshold then count + 1 else count) 0
    return res
```

While the Clever benchmark explicitly acknowledges this limitation (Thakur et al., 2025) and attempts to mitigate it by using non-computable specifications, this mitigation is incomplete. For instance, the "non-computable" specification of Fibonacci numbers defines the graph of the function as an inductive relation, making it straightforward to recover the recursive function definition.

In the realm of imperative code verification, we therefore go another way for avoiding specification leakage. We allow functional leaking specifications in the dataset, and instead enforce imperative code in the implementations. For this, we need to solve the related issue of *functional leakage* in the Velvet framework.

Functional leakage. Due to Velvet's nature of being a *shallow monadic embedding* in Lean, i.e. Velvet functions are terms in a special monad inside Lean, any pure Lean function can directly be used inside Velvet programs. This is part of the strength of the system as no separate standard library is required, but also allows cheating on imperative coding tasks by falling back to a fully functional solution wrapped inside the monad using `return` (the monad's *unit*). The example from one of our experiments given below shows how such *functional leakage* can happen even in the absence of specification leakage.

Listing 3. Example of functional leakage on `verina_basic_33`

```
method smallestMissingNumber (s : List Nat) return
    (result : Nat)
  require List.Pairwise (· ≤ ·) s
  ensures ¬ List.elem result s ∧ (∀ k : Nat, k <
    result → List.elem k s)
  do
    let res := List.foldl (fun acc x => if x = acc
    then acc + 1 else acc) 0 s
    return res
```

Robustly mitigating leakage. We address both specification and functional leakage through a unified strategy: rather than attempting to obfuscate specifications as in Clever, we permit functional specifications but enforce imperative implementations. This targets a strictly harder task that is robust to both forms of leakage. Specifically, we convey detailed rules in the language model prompts and enforce them using LLM-based evaluation. We require solutions not to have runtime complexity contributions from functional primitives, i.e. to only have constant-time primitives on critical paths. *Ghost variables* and invariants are permitted to use functional primitives, as their role is to couple mutable state with functional semantics rather than to contribute to the implementation. The full rules are included in the prompts in Appendix J.

4. Method

We study multiple agentic systems that couple large language models with Lean's compiler feedback in the Velvet verification framework. We consider a sequential baseline that alternates between generation and execution, and a multi-agent system based on subgoal decomposition with proof reconstruction and goal-directed method modification.

4.1. Sequential Agent

The most basic option to couple language models with environment feedback is by means of a sequential agent that proposes code to run and receives Lean's error messages upon failure. The (unsurprising) pseudo-code for sequential agents is given in Algorithm 2. We use sequential agents with compiler feedback as fundamental building blocks for multi-agent systems, where the individual agents fulfill tasks such as implementation generation or lemma proving.

4.2. Subgoal Decomposition

In auto-active verification, failures are addressed by the rising sea (Grothendieck, 1985) of additional helper lemmas and finer-grained invariants – without precise feedback about what is missing. In interactive verification, by contrast, proof failure directly motivates invariant changes: insufficient hypotheses call for stronger invariants, while unprovable conclusions require weaker ones. We design our multi-agent system around this interactive workflow. Concretely, we extract proof obligations as independent theorems and dispatch them to parallel prover agents. If all subgoals succeed, the full correctness theorem is reconstructed. Unlike standalone lemma synthesis, this captures partial progress at the subgoal level. Sequential subagents are launched on each such subgoal, and if all goals are proved, the full correctness theorem is reconstructed from the subgoal proofs with an optional sequential agent that fixes potential issues in the proof reconstruction due to parsing idiosyncrasies.

Modifying the method implementation is not an independent action but has to be *conflict-driven*: informed by a concrete necessity where an agent considers their proof obligation unprovable. To enable proof transfer across method modifications, we use Lean's extensive metaprogramming utilities to introduce named invariants with a deterministic naming scheme for generated verification conditions (e.g., `h_count.loop` for the loop-preservation obligation of an

Algorithm 1 Subgoal decomposition multi-agent proof system

Require: Language model agent M, initial method method_0, agent budget N
Ensure: Reconstructed proof or `FAILURE`

method $\leftarrow \text{method}_0$
while agent budget N not used up **do**
  goals $\leftarrow$ EXTRACT_GOALS(method)
  results $\leftarrow \{\, \text{RUN\_PROVERS\_CONCURRENTLY}(M, g) \mid g \in \text{goals} \,\}$
  if $\forall\, r \in \text{results},\ r$ is `SUCCESS` **then**
    return RECONSTRUCT_PROOF$(M, \text{results})$
  else if $\exists\, r \in \text{results},\ r$ is `CHANGE` **then**
    changes $\leftarrow \{\, r \in \text{results} \mid r \text{ is } \texttt{CHANGE} \,\}$
    method $\leftarrow$ IMPLEMENT$(M, \text{changes})$
  else
    return `FAILURE`
  end if
end while
return `FAILURE`

Table 1. **Results on the Verina benchmark.** Budget is represented as the turn limit $\times$ the number of agents launched per problem. For the Sequential Agent, k agents in parallel corresponds to pass@k.

Method	Turns $\times$ Agents	Solve Rate[†]
Baseline		
DS Prover V2 7B	64×1	20.0
Sequential Agent		
GPT-5	32×16	64.6
Gemini 3 Pro	32×16	55.6
Claude 4.5 Sonnet	32×16	63.3
Claude 4.5 Opus	32×16	**74.1**
GPTOSS-120B	16×4	30.2
Subgoal Decomposition		
GPT-5	8×128	57.7
Gemini 3 Pro	8×128	57.7
Claude 4.5 Sonnet	8×128	51.9
Claude 4.5 Opus	8×128	66.7
GPT-5 + Goedel Prover	8×512	40.7

[†] Solve rate reports the percentage of successful proofs and disproofs. Disproofs are taken from the best sequential-agent run for all methods; see Section 5.3.

invariant named `h_count`). When the method is updated, subgoal proofs from the previous attempt can be matched to new obligations by name, preserving partial progress. In this case, the method implementation is updated eagerly, the new subgoals are extracted but, crucially, we can attempt to transfer subgoal proofs from the previous attempt according to a naming scheme that links them to the invariants from which the goals were generated. The algorithm is summarized in Algorithm 1.

5. Results

We evaluate frontier LLMs on software verification of imperative code using the Lean-based framework described in Section 2.2 using the inference methods described in Section 4, sequential agent and subgoal decomposition multi-agent. Tables 1 and 2 show our main results on end-to-end software implementation and verification success rates on the Verina and Clever-Loom benchmarks.

Our results show that **(i)** hybrid verification is a very effective ingredient for software verification agents, raising evaluation scores at small inference budgets from 15% on Verina to 55%, **(ii)** performance shows consistent scaling without plateaus across several orders of magnitude (Section 5.5), **(iii)** our LLM-based imperativeness evaluation is effective at reducing specification leakage (Section 5.4), and **(iv)** the system scales to verifying complex algorithms such as Heapsort (Section 5.8) and its current limitations can be clearly outlined (Section 5.7).

5.1. Single-Agent vs Multi-Agent

Comparing the inference mechanisms of sequential agents and subgoal decomposition described in Section 4, we find: performance is model-dependent with subgoal decomposition outperforming sequential agents by a significant margin for Gemini-3 while being surpassed for Claude 4.5 Opus, 4.5 Sonnet and GPT-5 (Figure 2). Sequential agents are extremely efficient especially in the low-compute regime. On larger-scale evaluations for sorting algorithms (Section 5.7), we see anecdotal evidence for the scalability of the multi-agent system, continuously improving files over hundreds of LLM calls. As an additional benefit, the progress of the multi-agent system is easily interpretable due to the breakdown by subgoals and *conflict-driven* implementation updates. See Appendix F.1 for a comparison to a naive multi-agent system.

5.2. Sequential vs Parallel Compute

Sequential agents offer two directions to extend inference budgets: sequential compute (maximum number of turns T) and parallel compute (number of independent attempts k in pass@k). As can be seen in Figure 3 and 9, there is a tradeoff between these two depending on the total available budget, with optimal allocation potentially influenced by language model post-training specifics.

5.3. Disproving

Software verification benchmarks can have "label errors" in the sense of unsatisfiable specifications. Due to our translation methodology (see Appendix B), we made the decision

Table 2. **Results on the CLEVER benchmark.** We evaluate imperative code verification using workflows described in Section 4. Correctness is measured as joint correctness of model-generated implementation and proof. Results are not immediately comparable between functional and Loom/Velvet versions (see Section 3). Budget is represented as the turn limit $\times$ the number of agents launched for each problem, except for the baseline COPRA. For the Sequential Agent, k agents in parallel corresponds to pass@k.

Method	Turns $\times$ Agents	Solve Rate[‡]
Baseline		
COPRA (Claude 3.7)	600 seconds	8.7
Sequential Agent		
GPT-5	32×16	53.8
Gemini 3 Pro	32×16	32.8
Claude 4.5 Sonnet	32×16	59.6
Claude 4.5 Opus	32×16	**62.1**
Subgoal Decomposition		
GPT-5	8×128	46.6
Gemini 3 Pro	8×128	34.2
Claude 4.5 Sonnet	8×128	41.0
Claude 4.5 Opus	8×128	57.8

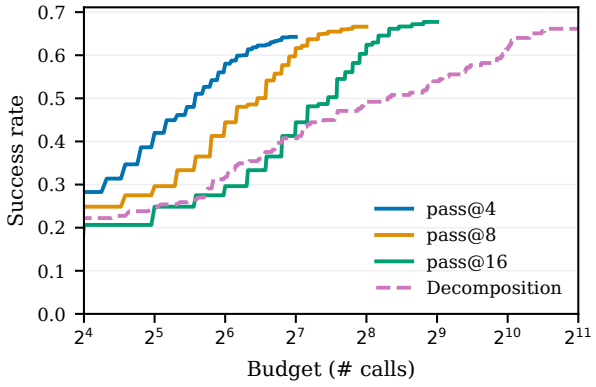
[‡] Solve rate reports the percentage of successful proofs. Disproving is not used on CLEVER.

for Clever-Loom to update the benchmark and fix such errors while for Verina, we aim for exact comparability and instead allow the agent to prove that the specification is unsatisfiable. For this, we automatically generate theorems of the form $\exists x \in X, P(x) \wedge \forall y \in Y, \neg Q(x, y)$ if P is the pre-condition on inputs $x \in X$ and $Q(x, y)$ is the post-condition relating inputs x with outputs $y \in Y$. The model is not informed whether the specification is satisfiable but has to decide whether prove or disprove it.

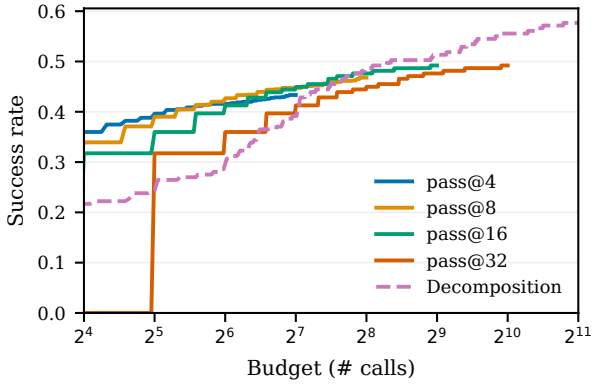
In 32-turn evaluations, all models except Claude 4.5 Sonnet proved 12 specifications to be unsatisfiable (Sonnet: 11), with GPT-5 notably successful on 347/384 of these cases. Note that unsatisfiability (regardless of implementation) is a stronger result than the 23 ground truth implementations that Harmonic (2025) prove incorrect.

5.4. Imperativeness Judge and Functional Leakage

In initial experiments, we observed a high degree of *functional leakage*: formally imperative methods that are only a thin wrapper around an otherwise functional computation (Section 3.1). Indeed, an evaluation of the subgoal decomposition agent on Verina that did not guard against this behavior resulted in a success rate of 75.1% (GPT-5, 8-turn pass@64) and prompted us to include an LLM-based evaluation of *imperativeness*, after which the performance dropped to 51.9%, but with properly imperative programs according to our manual review. We use an imperativeness throughout in all reported scores, and note that as specifications are functional, this also prevents specification leakage.



(a) Claude 4.5 Opus



(b) Gemini-3 Pro

Figure 2. **Inference scaling for sequential agents and multi-agent system on Verina.** Different pass@k settings entail different compute-performance scalings for sequential agents which we compare to a multi-agent system. The optimal mechanism depends on the specific model in use, with Gemini-3 benefiting from the multi-agent scaffold while Claude 4.5 Opus does not.

5.5. Scaling and Model Comparison

Comparing the frontier models GPT-5, Gemini 3 Pro, Claude 4.5 Opus and Claude 4.5 Sonnet as a smaller model on both sequential agent and multi-agent inference, we find that the larger models are roughly matched with Opus best overall, all outperforming the smaller Sonnet model (Tables 1 and 2, Figure 5). Across all models, there is no sign of plateauing performance across several order of magnitude of inference compute. For all experiments, the sampling temperature was set to 1.0, except for GPT-5, where the API did not expose temperature control.

5.6. Scalability of Multi-Agent Systems

Scaling inference via multi-agent systems can outperform naive pass@k because agents can share state, so extra compute can be used to improve a single rollout rather than spawning independent replicas. To test how much this holds

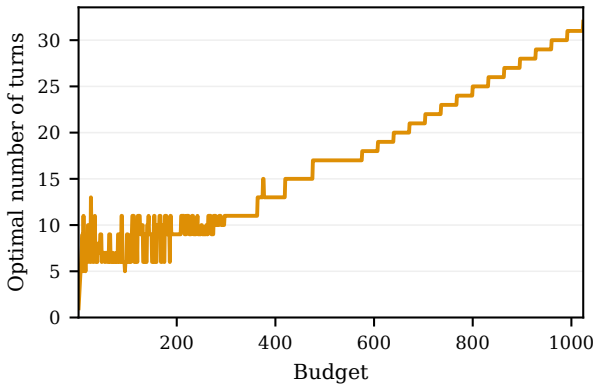


Figure 3. **Sequential vs Parallel Compute.** Given a maximum inference budget of up to C language model calls, we compute the optimal breakdown into $kT \leq C$ with $k \leq 32$ the number of independent attempts and $T \leq 32$ the maximum number of turns per attempt. Optimal T ranges between 6 and 16 turns, increases beyond that are an artifact of bounded k (Figure 9).

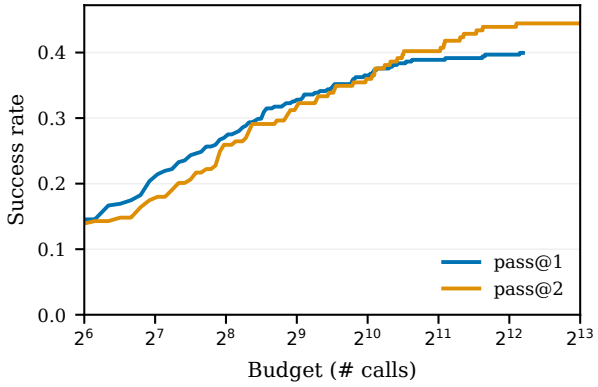


Figure 4. **Multiple attempts with multi-agent systems.**

for our subgoal-decomposition system (Section 4.2), we ran GPT-5 on Clever with $k = 2$ copies. Figure 4 shows that allocating additional compute to a single copy continues to improve performance up to 1200 model calls, after which pass@2 becomes preferable; in contrast, for sequential agents the crossover occurs much earlier, at 22 calls.

5.7. Qualitative Evaluation

By manually reviewing rollout successes and failures on Verina and Clever, we identify the limits of our system's verification capabilities. Employing moderate inference budgets frontier models consistently succeed at verifying constant-time algorithms (e.g. computing the area of a triangle), list scan- and fold-like operations (e.g. maximum, find) and naive brute-force algorithms (e.g. find pair). Tasks that require heavy use of array mutation or dynamic programming are comparatively rarely solved, or solved by employing

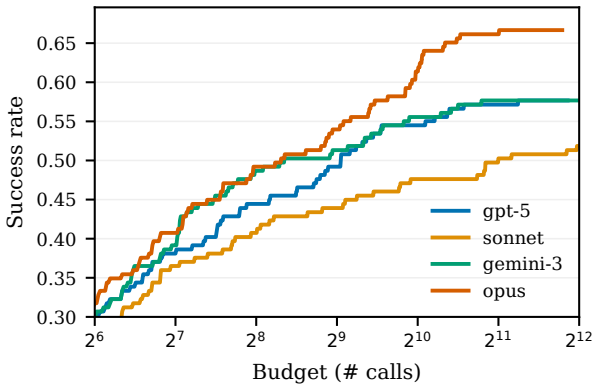


Figure 5. **Inference scaling comparison for multi-agent system with different models.** We evaluate using up to 128 subagents on Verina and plot by the total number of LLM calls spent according to iterative deepening (Section F.1). See Figure 8 for comparisons using latency and the Clever-Loom benchmark.

Table 3. **Sorting algorithms and their sub-procedures.** Ticks denote successful verification.

Selection Sort	✓
Bubble Sort	✓
Insertion Sort	✓
Binary Insertion Sort	✓
Recursive Quicksort	×
Quicksort	
Partition	✓
Step	×
Sort	✓
Heapsort	
Heapify	✓
Maxheap	✓
Sort	✓
Mergesort	
Mergeruns	✓
Mergepass	×
Sort	✓

Lean's libraries to reduce implementation and verification complexity. An example of this we have observed is the agent updating array elements using `Array.push` instead of by *moving* elements as mandated by the description. This analysis puts sorting algorithms at the very edge of what is feasible with moderate compute budget with frontier models. When prompted to implement and prove a specific sorting algorithm, models frequently succeed in the cases of Insertion Sort, Selection Sort, Bubble Sort or Binary Insertion Sort, but fail on the more challenging examples of in-place imperative Quicksort, imperative Mergesort with $\mathcal{O}(n)$ additional memory or Heapsort. After breaking down the last three sorting algorithms manually into three sub-routines with specifications each (but without implementations), Heapsort

completes successfully, while Quicksort and Mergesort each complete 2/3 sub-routines, leaving the most challenging one unverified. See Appendix G for details.

5.8. Full Verification of Heapsort

The successful verification of Heapsort using 215 sub-agents for `heapify`, 8 for `maxheap` and 134 for `heapsort_main` using our subgoal decomposition system and Claude 4.5 Opus demonstrates the scalability of our approach and the capabilities of frontier models, but also a certain degree of inefficiency. Heapsort is often considered a challenging example for verification. The program proving text of Reynolds (1981) gives a by-hand in a section which is starred to connote difficulty, and later formal treatments would sometimes take up an entire paper (e.g., Tafat & Marché (2011)). For the full code and more examples, see Appendix H.

6. Related Work

LLMs for Auto-Active Verification Pioneering works have established the use of LLMs in software verification, initially focusing on single-turn proof-hint synthesis. Benchmarks such as DafnyBench (Loughridge et al., 2024) and VerifyThisBench (Deng et al., 2025) (Dafny/Why3), VerusBench (Chen et al., 2025b) and Verified Cogen (JetBrains Research, 2024) (Verus), and InvBench (Wei et al., 2025) and LIG-MM (Liu et al., 2024) (C/Java) have been used to evaluate the utility of LLMs in generating invariants and reachability proofs.

Building on these benchmarks, frameworks like Clover (Sun et al., 2024), DafnyGym (Mugnier et al., 2025a), and AutoVerus (Yang et al., 2025a) use iterative feedback loops, demonstrating how compiler error messages can guide LLMs to refine proofs over multiple turns. While these studies primarily address the challenge of invariant and other proof-hint infilling for existing code, our work extends this scope to the end-to-end synthesis of implementation and verification in tandem. Additionally, while this work targeted introductory programming problems (such as the MBPP dataset), we benchmark our framework on implementation and verification of more complex algorithms.

Interactive Software Verification While LLMs are increasingly applied to software verification in interactive theorem provers, benchmarks like Verina (Ye et al., 2025), Clever (Thakur et al., 2025), and VeriBench (Anonymous, 2025) focus primarily on functional Lean code. Current frontier models exhibit a stark performance gap in these settings: while OpenAI's o4-mini achieves 61.4% on code generation (Verina pass@64), its proof synthesis task pass rate on Verina-advanced remains modest at 22.2%. Nevertheless, landmark verification success stories like the verification of the seL4 microkernel (Klein et al., 2009) and CompCert (Leroy, 2009) demonstrate the potential and scalability of software verification using ITPs. Furthermore, recent breakthroughs in IMO-level automated reasoning (Chen et al., 2025a; Achim et al., 2025) suggest that LLMs, when paired with robust inference scaling, can navigate complex formal proofs. Diverging from the functional focus of existing work, we introduce a novel approach targeting the *interactive* verification of *imperative* code with LLMs.

7. Conclusion

Following recent breakthroughs in formal mathematical theorem proving and verification of functional code with language models, fully-automated verification of imperative programs using Hoare logic is the next frontier in LLM-based software verification and will pave the way to real-world applications where memory layout (separation logic) and concurrency play a role too.

In this study, we adopted a hybrid verification framework (Loom/Velvet) that combines the strengths of auto-active verification frameworks such as Dafny with interactive theorem proving in Lean. We demonstrated that current frontier language models can leverage such frameworks purely in-context, achieving proficiency comparable to their capabilities in functional code verification despite the added complexity of mutable state and loop invariants.

We designed, evaluated and compared different multi-agent inference workflows for effectively scaling inference compute at reasonable end-to-end verification latencies, and highlighted subgoal decomposition and goal-directed modification as ingredients of a particularly well-suited workflow for this task. Our best system solves 74% of Verina and 62% of Clever tasks, and successfully verifies algorithms requiring deep insight such as array-based Heapsort, raising the bar beyond mere computations and brute-force solutions.

With complexity-aware verification as a natural next step, our results suggest that large-scale automated construction of verified imperative code datasets is now within reach. This opens new avenues for training and reinforcement learning on provably correct programs, and brings us closer to a future where AI-generated code comes with formally-verified correctness guarantees.

Limitations

While our results show that reasoning-focused LLMs bring automatically verified imperative code closer to reality, several limitations currently prevent production deployment. First, our Loom/Velvet pipeline relies on an experimental, actively developing software package with custom modifications (Section 2.2), and results may change as the framework

matures. Second, the generated imperative code targets Lean, which uses managed memory. Our approach does not yet address unmanaged production languages such as C++ or Rust, nor their memory models (e.g. separation logic, ownership types). Third, our scope covers code generation and correctness proofs but omits the equally critical upstream tasks of specification engineering and sub-method decomposition for our sorting algorithm evaluation, which remain manual in our pipeline. Finally, most of our experiments use closed-weight frontier models via API, with open-source models performing substantially worse potentially limiting reproducibility.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Achim, T., Best, A., Bietti, A., Der, K., Fédérico, M., Gukov, S., Halpern-Leistner, D., Henningsgard, K., Kudryashov, Y., Meiburg, A., Michelsen, M., Patterson, R., Rodriguez, E., Scharff, L., Shanker, V., Sicca, V., Sowrirajan, H., Swope, A., Tamas, M., Tenev, V., Thomm, J., Williams, H., and Wu, L. Aristotle: Imo-level automated theorem proving, 2025.

Amazon. AWS introduces Graviton5: the company's most powerful and efficient CPU. About Amazon News, 2025. URL https://www.aboutamazon.com/news/aws/aws-graviton-5-cpu-amazon-ec2. Accessed: 2026-01-22.

Anonymous. Veribench: End-to-end formal verification benchmark for AI code generation in lean 4. In *Submitted to The Fourteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=P7NUVF6wo4. under review.

Appel, A. W. *Software Foundations Volume 3: Verified Functional Algorithms*. Electronic Textbook, 2024. URL https://softwarefoundations.cis.upenn.edu/vfa-current/index.html. Version 2.7.

Asher, J. LeanExplore: A search engine for Lean 4 declarations, 2025. URL https://arxiv.org/abs/2506.11085.

Barbosa, H., Barrett, C. W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., and Zohar, Y. cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 415–442. Springer, 2022. doi: 10.1007/978-3-030-99524-9_24.

Beyer, D. and Strejček, J. Improvements in software verification and witness validation: Sv-comp 2025. In Gurfinkel, A. and Heule, M. (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 151–186, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-90660-2.

Breitner, J. Loogle. https://github.com/nomeata/loogle, 2023. Search tool for Lean/Mathlib.

Chen, L., Gu, J., Huang, L., Huang, W., Jiang, Z., Jie, A., Jin, X., Jin, X., Li, C., Ma, K., Ren, C., Shen, J., Shi, W., Sun, T., Sun, H., Wang, J., Wang, S., Wang, Z., Wei, C., Wei, S., Wu, Y., Wu, Y., Xia, Y., Xin, H., Yang, F., Ying, H., Yuan, H., Yuan, Z., Zhan, T., Zhang, C., Zhang, Y., Zhang, G., Zhao, T., Zhao, J., Zhou, Y., and Zhu, T. H. Seed-prover: Deep and broad reasoning for automated theorem proving, 2025a. URL https://arxiv.org/abs/2507.23726.

Chen, T., Lu, S., Lu, S., Gong, Y., Yang, C., Li, X., Misu, M. R. H., Yu, H., Duan, N., Cheng, P., Yang, F., Lahiri, S. K., Xie, T., and Zhou, L. Automated proof generation for rust code via self-evolution, 2025b. URL https://arxiv.org/abs/2410.15756.

Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-C: A software analysis perspective. In *Software Engineering and Formal Methods (SEFM)*, pp. 233–247. Springer, 2012. doi: 10.1007/978-3-642-33826-7_16.

de Moura, L. and Bjørner, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

de Moura, L. and Ullrich, S. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, pp. 625–635. Springer International Publishing, 2021. doi: 10.1007/978-3-030-79876-5_37.

Deng, X., Zhong, S., Bayazıt, B., Veneris, A., Long, F., and Si, X. Verifythisbench: Generating code, specifications, and proofs all at once, 2025. URL https://arxiv.org/abs/2505.19271.

Dijkstra, E. W. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. doi: 10.1145/355604.361591.

Dijkstra, E. W. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976. ISBN 978-0132158718.

Gladshtein, V., Pîrlea, G., Zhao, Q., Kurin, V., and Sergey, I. Foundational multi-modal program verifiers. *Proceedings of the ACM on Programming Languages*, 10(POPL), 2026. URL https://ilyasergey.net/assets/pdf/papers/loom-preprint.pdf. Discusses the Loom framework and the Velvet verifier.

Grothendieck, A. Récoltes et semailles. https://ncatlab.org/nlab/show/R%C3%A9coltes+et+semailles, 1985. Manuscript, written 1985–1987.

Harmonic. Aristotle learns to code, achieving new state-of-the-art of 96.8% on code verification benchmark. https://harmonic.fun/news#blog-post-verina-bench-sota, December 2025. Blog post, accessed 2026-01-28.

Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259.

Hubert, T., Mehta, R., Sartran, L., Horváth, M. Z., Žužić, G., Wieser, E., Huang, A., Schrittwieser, J., Schroecker, Y., Masoom, H., Bertolli, O., Zahavy, T., Mandhane, A., Yung, J., Beloshapka, I., Ibarz, B., Veeriah, V., Yu, L., Nash, O., Lezeau, P., Mercuri, S., Sönne, C., Mehta, B., Davies, A., Zheng, D., Pedregosa, F., Li, Y., von Glehn, I., Rowland, M., Albanie, S., Velingker, A., Schmitt, S., Lockhart, E., Hughes, E., Michalewski, H., Sonnerat, N., Hassabis, D., Kohli, P., and Silver, D. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 2025. doi: 10.1038/s41586-025-09833-y. URL https://doi.org/10.1038/s41586-025-09833-y.

JetBrains Research. verified-cogen: Repo for plan's verified code generation project. https://github.com/JetBrains-Research/verified-cogen, 2024. URL https://github.com/JetBrains-Research/verified-cogen.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pp. 207–220, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596.

Korf, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(85)90084-0. URL https://www.sciencedirect.com/science/article/pii/0004370285900840.

Lample, G., Lachaux, M.-A., Lavril, T., Martinet, X., Hayat, A., Ebner, G., Rodriguez, A., and Lacroix, T. Hypertree proof search for neural theorem proving, 2022. URL https://arxiv.org/abs/2205.11491.

Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., and Hawblitzel, C. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023. doi: 10.1145/3586037. URL https://doi.org/10.1145/3586037.

Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, pp. 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20.

Leroy, X. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., Hubert, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Sutherland Robson, E., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with alphacode. *Science*, 378 (6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

Liu, C., Yan, J., Feng, Y., Cao, Q., and Wu, X. Towards general loop invariant generation: A benchmark of programs with memory manipulation, 2024. URL https://arxiv.org/abs/2311.10483.

Loughridge, C., Sun, Q., Ahrenbach, S., Cassano, F., Sun, C., Sheng, Y., Mudide, A., Misu, M. R. H., Amin, N., and Tegmark, M. Dafnybench: A benchmark for formal software verification, 2024. URL https://arxiv.org/abs/2406.08467.

Math Inc. The strong prime number theorem. https://github.com/math-inc/strongpnt, 2025. URL https://github.com/math-inc/strongpnt.

mathlib Community, T. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, POPL '20, pp.

367–381. ACM, January 2020. doi: 10.1145/3372885.3373824. URL http://dx.doi.org/10.1145/3372885.3373824.

McCarthy, J. and Hayes, P. J. Some philosophical problems from the standpoint of artificial intelligence. In Michie, D. and Meltzer, B. (eds.), *Machine Intelligence 4*, pp. 463–502, Edinburgh, 1969. Edinburgh University Press.

Mugnier, E., Gonzalez, E. A., Jhala, R., Polikarpova, N., and Zhou, Y. Laurel: Unblocking automated verification with large language models, 2025a. URL https://arxiv.org/abs/2405.16792.

Mugnier, E., Zhou, Y., Jhala, R., and Coblenz, M. On the impact of formal verification on software development. *Proceedings of the ACM on Programming Languages*, 9 (OOPSLA2), 2025b. doi: 10.1145/3763181.

O'Hearn, P. W. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1–3):271–307, April 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.12.035. URL https://doi.org/10.1016/j.tcs.2006.12.035.

OpenAI, :, Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., Helyar, A., Madry, A., Beutel, A., Carney, A., Iftimie, A., Karpenko, A., Passos, A. T., Neitz, A., Prokofiev, A., Wei, A., Tam, A., Bennett, A., Kumar, A., Saraiva, A., Vallone, A., Duberstein, A., Kondrich, A., Mishchenko, A., Applebaum, A., Jiang, A., Nair, A., Zoph, B., Ghorbani, B., Rossen, B., Sokolowsky, B., Barak, B., McGrew, B., Minaiev, B., Hao, B., Baker, B., Houghton, B., McKinzie, B., Eastman, B., Lugaresi, C., Bassin, C., Hudson, C., Li, C. M., de Bourcy, C., Voss, C., Shen, C., Zhang, C., Koch, C., Orsinger, C., Hesse, C., Fischer, C., Chan, C., Roberts, D., Kappler, D., Levy, D., Selsam, D., Dohan, D., Farhi, D., Mely, D., Robinson, D., Tsipras, D., Li, D., Oprica, D., Freeman, E., Zhang, E., Wong, E., Proehl, E., Cheung, E., Mitchell, E., Wallace, E., Ritter, E., Mays, E., Wang, F., Such, F. P., Raso, F., Leoni, F., Tsimpourlas, F., Song, F., von Lohmann, F., Sulit, F., Salmon, G., Parascandolo, G., Chabot, G., Zhao, G., Brockman, G., Leclerc, G., Salman, H., Bao, H., Sheng, H., Andrin, H., Bagherinezhad, H., Ren, H., Lightman, H., Chung, H. W., Kivlichan, I., O'Connell, I., Osband, I., Gilaberte, I. C., Akkaya, I., Kostrikov, I., Sutskever, I., Kofman, I., Pachocki, J., Lennon, J., Wei, J., Harb, J., Twore, J., Feng, J., Yu, J., Weng, J., Tang, J., Yu, J., Candela, J. Q., Palermo, J., Parish, J., Heidecke, J., Hallman, J., Rizzo, J., Gordon, J., Uesato, J., Ward, J., Huizinga, J., Wang, J., Chen, K., Xiao, K., Singhal, K., Nguyen, K., Cobbe, K., Shi, K., Wood, K., Rimbach, K., Gu-Lemberg, K., Liu, K., Lu, K., Stone, K., Yu, K., Ahmad, L., Yang, L., Liu, L., Maksin, L., Ho, L., Fedus, L., Weng, L., Li, L., McCallum, L., Held, L., Kuhn, L., Kondraciuk, L., Kaiser, L., Metz, L., Boyd, M., Trebacz, M., Joglekar, M., Chen, M., Tintor, M., Meyer, M., Jones, M., Kaufer, M., Schwarzer, M., Shah, M., Yatbaz, M., Guan, M. Y., Xu, M., Yan, M., Glaese, M., Chen, M., Lampe, M., Malek, M., Wang, M., Fradin, M., McClay, M., Pavlov, M., Wang, M., Wang, M., Murati, M., Bavarian, M., Rohaninejad, M., McAleese, N., Chowdhury, N., Chowdhury, N., Ryder, N., Tezak, N., Brown, N., Nachum, O., Boiko, O., Murk, O., Watkins, O., Chao, P., Ashbourne, P., Izmailov, P., Zhokhov, P., Dias, R., Arora, R., Lin, R., Lopes, R. G., Gaon, R., Miyara, R., Leike, R., Hwang, R., Garg, R., Brown, R., James, R., Shu, R., Cheu, R., Greene, R., Jain, S., Altman, S., Toizer, S., Toyer, S., Miserendino, S., Agarwal, S., Hernandez, S., Baker, S., McKinney, S., Yan, S., Zhao, S., Hu, S., Santurkar, S., Chaudhuri, S. R., Zhang, S., Fu, S., Papay, S., Lin, S., Balaji, S., Sanjeev, S., Sidor, S., Broda, T., Clark, A., Wang, T., Gordon, T., Sanders, T., Patwardhan, T., Sottiaux, T., Degry, T., Dimson, T., Zheng, T., Garipov, T., Stasi, T., Bansal, T., Creech, T., Peterson, T., Eloundou, T., Qi, V., Kosaraju, V., Monaco, V., Pong, V., Fomenko, V., Zheng, W., Zhou, W., McCabe, W., Zaremba, W., Dubois, Y., Lu, Y., Chen, Y., Cha, Y., Bai, Y., He, Y., Zhang, Y., Wang, Y., Shao, Z., and Li, Z. Openai o1 system card, 2024. URL https://arxiv.org/abs/2412.16720.

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

Polu, S., Han, J. M., Zheng, K., Baksys, M., Babuschkin, I., and Sutskever, I. Formal mathematics statement curriculum learning, 2022. URL https://arxiv.org/abs/2202.01344.

Ren, Z. Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., Wu, Z. F., Gou, Z., Ma, S., Tang, H., Liu, Y., Gao, W., Guo, D., and Ruan, C. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition, 2025. URL https://arxiv.org/abs/2504.21801.

Reynolds, J. C. *The Craft of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1981. ISBN 978-0131899882.

Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pp. 55–74. IEEE, 2002. doi: 10.1109/LICS.2002.1029817.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. Toolformer: Language models can teach themselves to use tools. In *Thirty-Seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=3mRwyG5ljh.

Scott, M. and Aldrich, J. *Programming Language Pragmatics*. Elsevier Science, 2025. ISBN 9780323984232. URL https://books.google.co.uk/books?id=qjwTEQAAQBAJ.

Sun, C., Sheng, Y., Padon, O., and Barrett, C. Clover: Closed-loop verifiable code generation, 2024. URL https://arxiv.org/abs/2310.17807.

Tafat, A. and Marché, C. *Binary heaps formally verified in Why3*. PhD thesis, INRIA, 2011.

Team, T. C. D. The coq proof assistant, September 2024. URL https://doi.org/10.5281/zenodo.14542673.

Thakur, A., Li, Y., Karalias, G., Min, C., Pikus, B., Rennie, J., Tandon, P., and Chaudhuri, S. CLEVER: A curated benchmark for formally verified code generation. *arXiv preprint arXiv:2505.13938*, 2025. URL https://arxiv.org/abs/2505.13938.

Urban, J. 130k lines of formal topology in two weeks: Simple and cheap autoformalization for everyone?, 2026. URL https://arxiv.org/abs/2601.03298.

Wang, H., Unsal, M., Lin, X., Baksys, M., Liu, J., Santos, M. D., Sung, F., Vinyes, M., Ying, Z., Zhu, Z., Lu, J., de Saxcé, H., Bailey, B., Song, C., Xiao, C., Zhang, D., Zhang, E., Pu, F., Zhu, H., Liu, J., Bayer, J., Michel, J., Yu, L., Dreyfus-Schmidt, L., Tunstall, L., Pagani, L., Machado, M., Bourigault, P., Wang, R., Polu, S., Barroyer, T., Li, W.-D., Niu, Y., Fleureau, Y., Hu, Y., Yu, Z., Wang, Z., Yang, Z., Liu, Z., and Li, J. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning, 2025. URL https://arxiv.org/abs/2504.11354.

Wei, A., Suresh, T., Sun, T., Wu, H., Wang, K., and Aiken, A. Invbench: Can llms accelerate program verification with invariant synthesis?, 2025. URL https://arxiv.org/abs/2509.21629.

Yang, C., Li, X., Misu, M. R. H., Yao, J., Cui, W., Gong, Y., Hawblitzel, C., Lahiri, S., Lorch, J. R., Lu, S., Yang, F., Zhou, Z., and Lu, S. Autoverus: Automated proof generation for rust code. *Proc. ACM Program. Lang.*, 9(OOPSLA2), October 2025a. doi: 10.1145/3763174. URL https://doi.org/10.1145/3763174.

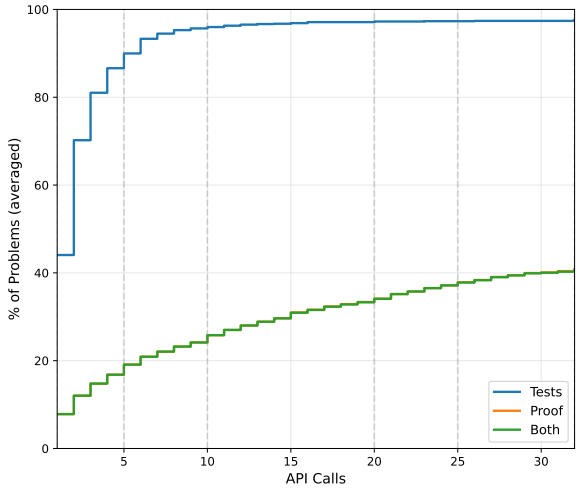
Yang, C., Neamtu, N., Hawblitzel, C., Lorch, J. R., and Lu, S. Verusage: A study of agent-based verification for rust systems, 2025b. URL https://arxiv.org/abs/2512.18436.

Ye, Z., Yan, Z., He, J., Kasriel, T., Yang, K., and Song, D. Verina: Benchmarking verifiable code generation. *arXiv preprint arXiv:2505.23135*, 2025.

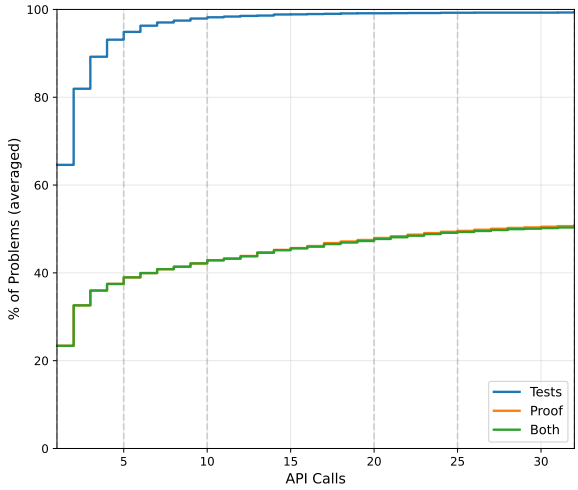
Zhang, T. Running claude opus 4.5 on the verina benchmark. https://www.linkedin.com/pulse/running-claude-opus-45-verina-benchmark-tony-zhan January 2026. LinkedIn article, accessed 2026-01-28.

Appendix

A. Test and Proof Pass-Rate Gap



(a) Clever



(b) Verina

Figure 6. Per-problem cumulative pass rates vs. number of LLM API calls for (a) Clever and (b) Verina. Curves represent the fraction of problems achieving first success on tests, proofs, or both. Statistics are computed per-problem then averaged to avoid overweighting easier problems.

Figure 6 shows that test success rises rapidly and approaches 100%, whereas proof success improves much more slowly. The large and consistent gap between the test and proof curves suggests that proving correctness is substantially harder than passing tests.

B. Benchmark Translation Methology

B.1. Verina-Loom

Verina provides structured metadata for each task, including separated function signatures, pre-conditions, and post-conditions. We automatically extract these components and assemble them into Velvet method specifications using Dafny-inspired syntax. This mechanical translation ensures strict comparability with existing results on functional code verification.

B.2. Clever-Loom

Clever specifications are monolithic propositions relating inputs and outputs without separating pre- and post-conditions. While these could in principle be used as monolithic post-conditions, doing so would undermine the efficiency of SMT-based verification, which excels at discharging many small obligations but struggles with large compound statements. Rather than parsing the specifications to extract logical structure, we adopt a semi-automated pipeline:

1. Prompt Gemini 2.5 Pro to translate each specification into Velvet `require`/`ensures` syntax
2. Conduct systematic human review to ensure correctness and idiomatic style
3. Fix unsatisfiable specifications from the original dataset where identified

We aim to stay as close as possible to the original Clever specifications, but some deviations are necessary for idiomatic Velvet. As a result, benchmark numbers are not strictly comparable to the original Clever results. However, this process yields a high-quality benchmark for verified imperative code, with several specification errors from the original dataset corrected.

From Verina's structured data format, we retrieve the signature, pre-conditions and post-conditions, allowing us to assemble the corresponding Velvet method specification automatically. See Listing 1 for an example including the specification header in Dafny-inspired syntax. With the resulting Verina-Loom benchmark, we aim for strict comparability with existing numbers functional code verification.

C. Tool Use

In initial experiments, we identified missing knowledge about Lean's standard library (generally and in the specific version that Loom/Velvet uses) a limiting factor for LLM agents to produce correctness proofs. We therefore provide library search as *tools* (Schick et al., 2023) for language models with functional calling capability. Specifically, we deploy Loogle (Breitner, 2023) and LeanExplore (Asher, 2025) as local services, two tools for semantic and syntactic search in Lean's standard library and `mathlib` (mathlib Community, 2020).

Instead of a flat design where the agent decides at each step whether to run Lean code or perform a library search and all interactions are concatenated, we opt for a nested design where outer turns are interaction with Lean and inner turns are library search tools. The agent can perform multiple library search tool calls until the inner loop budget runs out or it decides to return to the outer Lean loop. The agent is requested to summarize its library search findings for subsequent rounds, and the inner loop messages are removed from the chat history in order to save context tokens and refocus on the main verification task.

Empirically, tool-augmented library search does not improve pass rates over the baseline, but it increases compute because the agent spends additional turns querying tools and summarising results before returning to the main Lean loop.

D. Extended Related Work

D.1. Agentic software verification

Auto-Active Verification: Datasets and Evaluation In auto-active settings, verification relies on SMT solvers (e.g., CVC5 or Z3) to discharge proof obligations, shifting the agent's burden from full proof construction to *proof hint inference*– specifically the synthesis of loop invariants and pre/post-conditions. DAFNYBENCH (Loughridge et al., 2024) provides the largest corpus for this task, comprised of verified Dafny programs stripped of proof hints where agents must supply missing invariants and assertions to verify existing code. Similarly, VERIFYTHISBENCH (Deng et al., 2025) aggregates tasks across Dafny and Why3, focusing on the joint synthesis of specifications and implementations. More recent efforts target system-level languages, VERUS-BENCH (Yang et al., 2025b) and VERIFIED COGEN(JetBrains Research, 2024) evaluate the synthesis of verified Rust code via the Verus toolchain, testing the model's ability to navigate complex memory safety constraints. These modern datasets build upon foundational competitions like SV-COMP (Beyer & Strejček, 2025), which continue to serve as rich sources for mining safety properties and program lemmas in C and Java. To this end, INVBENCH (Wei et al., 2025) draws verification tasks from SV-COMP's reachability benchmarks to evaluate LLM-based invariant synthesis, while (Liu et al., 2024) incorporate SV-COMP programs into their benchmark for loop invariant generation with memory manipulation in auto-active verification settings. Throughout this line of work, frontier LLMs have shown some emergent capability of aiding with tasks such as invariant, and implementation synthesis but the relative brittleness introduced by the SMT-backends has slowed progress of expanding scope and complexity of the verification tasks studied.

In addition to these datasets, recent systems increasingly treat the verifier as an interactive environment and use its feedback to iteratively repair annotations, specifications, and proofs. Clover (Sun et al., 2024) proposes a framework in Dafny to synthesize programs and annotations using natural language descriptions and applies it to relatively introductory MBPP-level problems with ground-truth implementations. Laurel (Mugnier et al., 2025a) further isolates this approach by focusing on assertion generation rather than end-to-end verified program synthesis. In parallel, Verus-based (Lattuada et al., 2023) agentic work starts from existing Rust code and generates specifications and proofs without a held-out ground-truth specification (Yang et al., 2025a; Chen et al., 2025b), making its evaluation regime closer to spec-and-proof recovery than to end-to-end generation under a reference spec. In contrast, WybeCoder is an end-to-end verified imperative code generation in a hybrid environment, where interactive goals can drive invariant and proof refinement and where inference can be scaled via subgoal decomposition.

ITP-Based Verification: Datasets and Evaluation Conversely, interactive theorem proving requires the explicit construction of proof terms or tactic scripts, presenting a steeper challenge for current models. CLEVER (Thakur et al., 2025)

provides a curated benchmark of 161 problems for end-to-end verified code generation in Lean 4, where each problem requires both generating a specification matching a held-out ground truth and producing a provably correct implementation. Evaluations reveal that state-of-the-art models achieve an end-to-end success rate of less than 1%. VERINA (Ye et al., 2025) offers a complementary benchmark of 189 programming challenges with modular evaluation across code, specification, and proof generation tasks. On this benchmark, the best-performing model (OpenAI o4-mini) achieves 61.4% on code generation, 51.0% on specification generation, but only 22.2% on proof generation at pass@64. Beyond the code-generation focus of these benchmarks, VERIBENCH (Anonymous, 2025) tests translation capabilities, requiring agents to convert Python algorithms into verified Lean 4 implementations including unit tests, correctness theorems, and formal proofs. In their evaluations, frontier LLMs' compilation rates of only 12.5% contrast sharply with mathematical theorem proving performance, a substantial gap in neural methods for software verification applications in ITPs. Weak performance of current agentic systems in software verification is not an inherent limitation of the ITP framework: landmark projects such as the seL4 microkernel verification in Isabelle (Klein et al., 2009), CompCert in Rocq (Leroy, 2009), and AWS Nitro Isolation engine verification (Amazon, 2025) with Isabelle/HOL and Idris demonstrate that large-scale verified software is achievable with sufficient human expertise, promising a clear pathway for scaling multi-agent approaches for this task.

D.2. Neural Formal Theorem Proving

The application of LLMs to formal mathematics has evolved from simple tactic generation to sophisticated neural theorem-proving systems integrated with ITPs. Early efforts, beginning with GPT-f, demonstrated that LLMs could be trained to generate valid proof steps in Lean by training on a curriculum of formal statements and proofs (Polu et al., 2022; Polu & Sutskever, 2020). However, progress was initially constrained by the scarcity of high-quality formal data and the challenges of long-horizon reasoning.

To navigate the vast state-space in the formal theorem proving environment, early neural theorem provers primarily focused on exclusively formal code generation and heavily relied on external search algorithms like Monte-Carlo Tree Search (Lample et al., 2022). However, recent breakthroughs in reasoning (OpenAI et al., 2024) have enabled a tighter integration of informal and formal domains at inference time. By interleaving informal *thinking traces* with formal code blocks, systems like DeepSeek-Prover-v2 (Ren et al., 2025) and Kimina-Prover (Wang et al., 2025) leverage the vast mathematical knowledge accumulated during pre-training while grounding their logic in verifiable rewards.

These advances have substantially improved long-horizon, tool-integrated formal reasoning, leading to large gains on established formal-math benchmarks (Chen et al., 2025a; Hubert et al., 2025). Strong formal reasoners can also enable autoformalization and the construction of large verified corpora(Urban, 2026; Math Inc., 2025), helping alleviate data scarcity for training and evaluation. While primarily developed for formal mathematics, these tool-integrated reasoning and repair techniques are likely transfer to naturally to verified programming, where agents must additionally synthesize and refine proof-hints and loop invariants for programs.

E. Algorithms

We provide pseudocode for the inference workflows described in Section 4.

Sequential Agent (Algorithm 2). The workflow alternates between language model generation and Lean compiler feedback. The agent accumulates a history of (code, error) pairs and conditions each subsequent generation on this context until verification succeeds or the budget is exhausted.

Lemma-Writing Multi-Agent System (Algorithm 3). Multiple agents work concurrently on a shared file, each proposing one of three actions: modify the implementation, conjecture and prove helper lemmas, or edit the final proof. Successful contributions are merged into the shared state for subsequently launched agents. We explored this system in early experiments but observed sub-par results compared to sequential and subgoal decomposition agents. We therefore did not scale this approach further.

Algorithm 2 Sequential agent

Require: Language model M, Lean environment E, problem specification $\mathcal{P}$, budget T
Ensure: Verified Lean code or `FAILURE`

history $\leftarrow$ [ ]
for $t \leftarrow 1$ **to** T **do**
  code $\leftarrow M(\mathcal{P}, \text{history})$
  result $\leftarrow E(\text{code})$
  if result is `SUCCESS` **then**
    return code
  end if
  history $\leftarrow$ history $+$ [(code, result.error)]
end for
return `FAILURE`

Algorithm 3 Lemma-writing multi-agent proof system

Require: Language model agent M, Lean environment E, method specification $\mathcal{P}$, degree of parallelism k, agent budget N, initial method method_0, initial error error_0
Ensure: Verified Lean code or `FAILURE`

file $\leftarrow$ MAKE_FILE(method_0, default_proof)
error $\leftarrow \text{error}_0$
k-**parallel for** $i \leftarrow 1$ **to** N **do**
  action $\leftarrow M(E, \mathcal{P}, \text{file}, \text{error})$
  file $\leftarrow$ UPDATE_FILE(file, action)
  result $\leftarrow E(\text{file})$
  if result is `SUCCESS` **then**
    return file
  end if
  error $\leftarrow$ result.error
end for
return `FAILURE`

F. Additional Quantitative Evaluation Results

F.1. Evaluating Multi-Agent Systems: Accuracy/Latency/Compute Tradeoffs

Evaluations of machine learning systems should not merely report single performance numbers, but their scaling with respect to computational inputs. For single-agent LLM systems, the computational budget is approximated by the number of LLM calls (or tokens) used. For multi-agent setups, the presence of concurrency bifurcates this metric into two separate notions: *compute*, i.e. the total amount of computational resources (calls, tokens) used overall, and *latency*, i.e. the total amount of computational resources spent on the longest critical path, measured as a time duration or likewise in tokens or LLM calls.

Running a single-agent system k times independently in parallel is a basic multi-agent system, namely pass@k evaluation. In this case, latency is the maximum of the lengths of all k trajectories while compute is their sum.

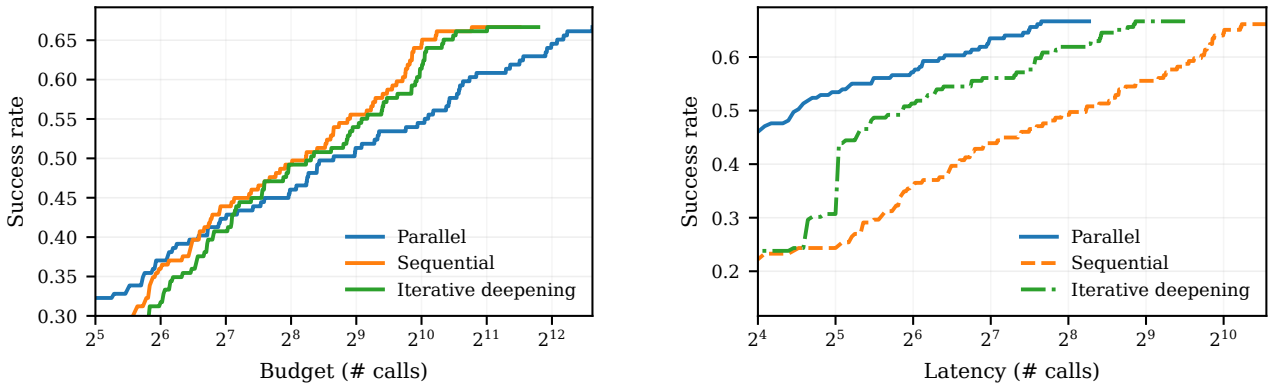
Indeed, for conducting up to k evaluations with early stopping, different parallelism schemes allow to obtain different latency-compute tradeoffs: a purely parallel approach minimizes latency at the cost of wasted compute on easy tasks, while a purely sequential design minimizes compute while creating high latency. *Iterative deepening* (Korf, 1985) provides a mechanism to trade off latency and compute in a different way: assume parallel stages of exponentially growing sizes are conducted sequentially and the average pass rate is $\frac{1}{k}$. Then average latency is $\mathcal{O}(\log k)$ while compute is optimally $\mathcal{O}(k)$.

The subgoal decomposition system described in Section 4.2 employs parallel provers, and we compare performance by latency and by compute for sequential, parallel and iterative deepening prover schedules. Independent goals are always conducted in parallel, while implementation subagents by design are always in sequence after a proving stage. Concretely, our experiment uses Claude 4.5 Opus on Verina with 8 provers per goal, and iterative deepening uses a $1 + 1 + 2 + 4$ schedule. Figure 7 shows how performance scales with compute and latency with the three sampling schemes. Because the first prover succeeds on many goals, iterative deepening approaches the performance of sequential scheduling in terms of total compute, while maintaining low latency that approaches it to the performance of parallel scheduling when comparing by latency. Note that this evaluation could be conducted post-hoc based on a single run with annotated prover agent launch order.

Based on these results, we decide to report multi-agent performance exclusively with iterative deepening.

F.2. Model Comparison on Multi-Agent Inference Scaling

Figure 8 shows how performance scales with respect to total call budget and latency, respectively, for each model.



(a) Compute *(b)* Latency

Figure 7. **Performance by compute and latency.**

F.3. Scalability of Multi-Agent Systems

One reason for scaling inference with multi-agent systems as opposed to pure independent pass@k scaling of single-agent systems is that by a carefully designed way of sharing state, individual subsystems can mutually inform and benefit each other. For an ideal multi-agent system, additional compute would always optimally be used to extend the resources of a single copy of the system, not to spawn multiple independent copies of it in pass@k fashion.

To evaluate the extent to which this applies to the subgoal decomposition multi-agent system from Section 4.2, we evaluate it using GPT-5 with $k = 2$ copies on Clever. As can be seen in Figure 4, increasing the compute allocated to a single copy of the system remains advantageous until a large budget of around 1200 calls is reached. The corresponding crossover point for sequential agents is at around 22.

F.4. Sequential vs Parallel Compute

For sequential agents, there are two ways of increasing the inference compute budget: by increasing the maximum number of turns T or the number of independent attempts k. In Figure 9, we evaluate with $k_{\max} = 32, T_{\max} = 32$ and compute from this data the optimal k and T for varying compute budgets $C \geq kT$. The results indicate that for GPT-5 on Clever, multi-turn iteration is useful at moderate numbers of iterations, and that boundary effects appear for the tool use agent where iteration lengths of 8, 16 and 24 have particular importance.

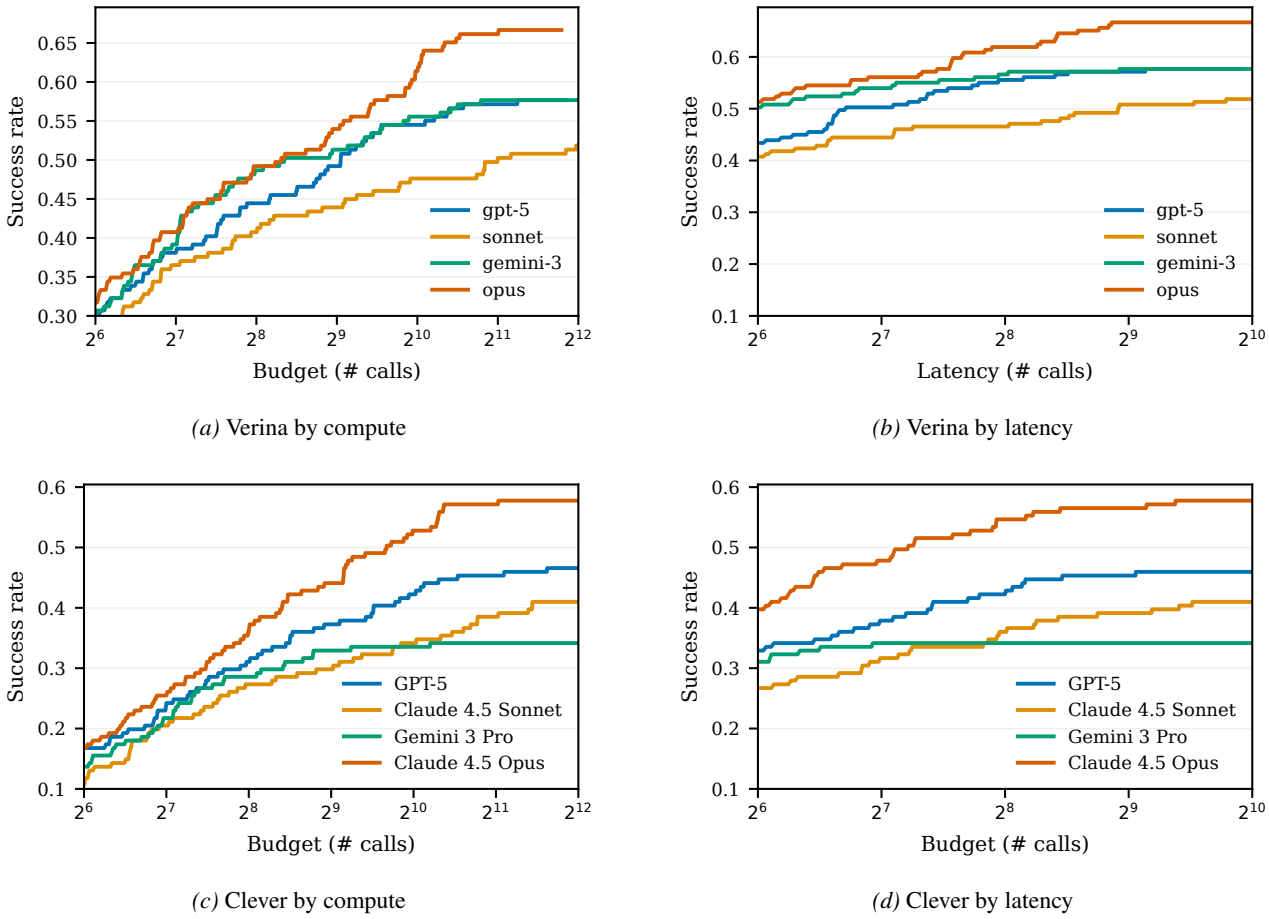
G. Extended Qualitative Evaluation

To better understand the current limitations of WybeCoder, we considered a number of classic sorting algorithms; see Table 3. We prompted asking for well known algorithms and for proofs of them. The first four passed without difficulty. The next three sorts are more advanced and are often presented recursively. Loom currently has limited support for recursion, so we asked for iterative variants. Mergesort is bottom up with $\mathcal{O}(n)$ extra memory, while Heapsort and Quicksort are standard in-place iterative versions. To ease the verification effort, we manually provided specifications of several sub-routines and not only of the top-level main functions.

The verifications of the main functions themselves all went through, based on these specs. In Quicksort and Mergesort, the proofs of the most challenging sub-procedures `mergepass` and `quickstep` failed.

We describe our experience with Heapsort because of lessons we learned from it. We began by manually providing specification for the three sub-routines `heapify`, `maxheap` and `heapsort_main` with LLM assistance. The three procedures were then generated and proven. Although the overall specification of sorting was satisfied, human examination of the proof script revealed a curious case. In a comment in `heapify`, the model wrote:

```
-- Since Heapify may permute elements outside the heap, we cannot guarantee
-- sortedness from the heapsort phase. Apply selection sort to ensure sortedness.
```



(a) Verina by compute

(b) Verina by latency

(c) Clever by compute

(d) Clever by latency

Figure 8. **Inference scaling comparison for multi-agent system with different models.** We evaluate using up to 128 subagents on Verina and Clever and plot by the total budget of LLM calls spent and LLM call latency according to iterative deepening (Section F.1).

The `heapsort_main` routine included a selection sort within it! The selection sort code was in fact unnecessary, because `heapify` does not permute outside the heap. The technical difficulty was that the manually provided specification for `heapify` left out a *frame axiom*, saying what does not change (McCarthy & Hayes, 1969). Once we corrected the spec to include the missing frame axiom, a correct version of Heapsort was generated, and the verification went through.

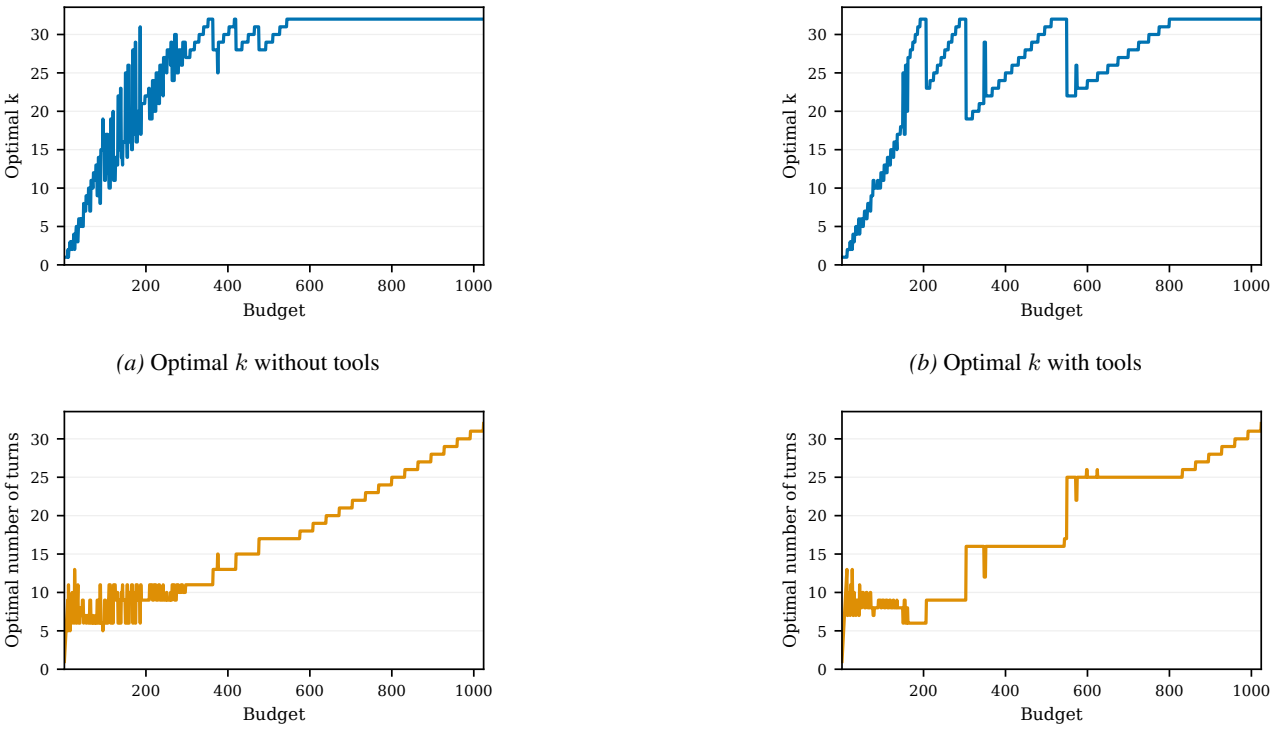
This experience illustrates several points on the gap from current capabilities to a verified form of vibe coding which is effective at scale: (i) humans (with LLM help) wrote the specs for the sub-procedures, where ideally the AI would do so; (ii) neither humans nor LLM spotted the missing frame axiom, or the fact that it could be consistently added to the spec based on the code; and (iii) the AI valued task completion higher than abiding by implictit intention and did not object to the version with embedded selection sort, a human was needed.

All told, these points indicate how proficiency in doing proofs of individual procedures shifts the bottleneck towards *specification engineering and review*. This mirrors a development in machine learning for formal mathematics where increased proving capabilities move the bottleneck toward *formalizing definitions* correctly and in line with the practical requirements of a given interactive theorem prover.

H. Examples

H.1. Heapsort

To evaluate the limits of the WybeCoder framework's ability to synthesize more complex algorithms, we specified heapsort by decomposing it into three methods with formal pre- and post-conditions:



(a) Optimal k without tools

(b) Optimal k with tools

(c) Optimal T without tools

(d) Optimal T with tools

Figure 9. **Optimal inference budget allocation for GPT-5.** Given a maximum inference budget of up to C language model calls, we compute the optimal breakdown into $kT \leq C$ with $k \leq 32$ the number of independent attempts and $T \leq 32$ the maximum number of turns per attempt. Optimal T is larger than 1, meaning that multi-turn interactions are useful. However, T saturates more quickly, suggesting that our bound $k \leq 32$ was suboptimal for a maximal budget of $1024 = 32^2$. For tool calling, the optimal T has plateaus at discrete levels of approximately 8, 16 and 24 turns, respectively. This could be an artifact of discrete post-training recipes, with models optimizing their exploration/exploitation tradeoff based on an inferred maximum number of turns.

1. `Heapify_ArrayPerm`: The sift-down operation that restores the max-heap property at a given index, assuming the subtrees rooted at its children already satisfy the heap property.
2. `BuildMaxHeap_ArrayPerm`: Constructs a max-heap from an arbitrary array by iteratively calling `Heapify_ArrayPerm` from the last non-leaf node down to the root.
3. `HeapSort_ArrayPerm`: The main sorting algorithm that first builds a max-heap, then repeatedly extracts the maximum element by swapping it to the end and restoring the heap property on the reduced prefix.

Each method was specified with formal contracts ensuring:

- **Size preservation**: The output array has the same size as the input.
- **Permutation invariance**: The output is a permutation of the input (`List.Perm arr.toList result.toList`).
- **Correctness properties**: `Heapify_ArrayPerm` establishes the heap property at the target index; `BuildMaxHeap_ArrayPerm` produces a valid max-heap; `HeapSort_ArrayPerm` produces a sorted array (`List.Sorted (· ≤ ·) result.toList`).
- **Frame conditions**: Elements outside the active heap region remain unchanged.

Using Claude Opus 4.5, the framework successfully generated both the executable Lean implementations and complete correctness proofs for all three methods. The resulting verified implementation spans approximately 2,000 lines of

Lean 4 code, including loop invariants, auxiliary lemmas about heap index arithmetic (e.g., `parent`, `left`, `right` relationships), and the main correctness theorems. The proof obligations—particularly the loop invariant preservation for `Heapify_ArrayPerm`'s sift-down loop and the partition/sortedness invariants for `HeapSort_ArrayPerm`—required substantial automated reasoning about array permutations, integer arithmetic, and the recursive structure of the heap property.

This case study demonstrates that WybeCoder can automate verification work that would otherwise require multiple days of effort from a team of expert formal methods practitioners, while producing machine-checked proofs that provide strong correctness guarantees. We note that the proving pipeline does not optimize proof length, maintainability or elegance, and that a minimal proof might be significantly shorter.

Listing 4. Implementation and correctness proof of Heapsort for array inputs

```
import Auto
import Aesop
import Lean
import Mathlib
import CaseStudies.Velvet.Std
import CaseStudies.TestingUtil
set_option loom.semantics.termination "total"
set_option loom.semantics.choice "demonic"
set_option loom.solver "cvc5"
set_option auto.smt.timeout 3
set_option maxHeartbeats 100000
set_option auto.smt.trust true
/-! ## Array helpers -/
def left (i : Nat) : Nat := 2 * i + 1
def right (i : Nat) : Nat := 2 * i + 2
def parent (i : Nat) : Nat := (i - 1) / 2
def isMaxHeapOn (arr : Array Int) (lo hi : Nat) : Prop :=
  ∀ k, lo ≤ k ∧ k < hi →
    ((left k < hi → arr[k]! ≥ arr[left k]!) ∧
     (right k < hi → arr[k]! ≥ arr[right k]!))
def isMaxHeapPrefix (arr : Array Int) (heapSize : Nat) : Prop :=
  isMaxHeapOn arr 0 heapSize
/-! ## Frame helper: unchanged outside prefix [0, heapSize) -/
def frameOutsidePrefix (before after : Array Int) (heapSize : Nat) : Prop :=
  before.size = after.size ∧
    ∀ k, k < before.size → heapSize ≤ k → after[k]! = before[k]!
method Heapify_ArrayPerm
  (arr : Array Int)
  (heapSize : Nat)
  (i : Nat)
  return (result : Array Int)
require heapSize ≤ arr.size
require i < heapSize
require isMaxHeapOn arr (i+1) heapSize
ensures result.size = arr.size
ensures isMaxHeapOn result i heapSize
ensures List.Perm arr.toList result.toList
ensures frameOutsidePrefix arr result heapSize
do
  let mut a := arr
  let mut idx := i
  let mut done := false
  while ¬done
    invariant hsize : a.size = arr.size
    invariant hidx_bound : idx < heapSize
    invariant hidx_ge : idx ≥ i
    -- Heap property holds for all nodes in [i, heapSize) except possibly at idx
    invariant hheap_except : ∀ k, i ≤ k ∧ k < heapSize ∧ k ≠ idx →
      ((left k < heapSize → a[k]! ≥ a[left k]!) ∧
       (right k < heapSize → a[k]! ≥ a[right k]!))
    -- The parent of idx (if idx > i, meaning parent is in range) dominates idx
    -- This is maintained because when we swap idx with child c, new parent of c is idx,
    -- and we put the max value at idx, so idx dominates the new value at c
    invariant hparent_dom : idx > i → a[parent idx]! ≥ a[idx]!
    -- Additionally, parent of idx dominates children of idx (needed for swap preservation)
    invariant hparent_children : idx > i →
      ((left idx < heapSize → a[parent idx]! ≥ a[left idx]!) ∧
       (right idx < heapSize → a[parent idx]! ≥ a[right idx]!))
    -- When done, idx also satisfies heap property
    invariant hdone_ok : done →
      ((left idx < heapSize → a[idx]! ≥ a[left idx]!) ∧
       (right idx < heapSize → a[idx]! ≥ a[right idx]!))
    invariant hperm : List.Perm arr.toList a.toList
    invariant hframe : ∀ k, k < arr.size → heapSize ≤ k → a[k]! = arr[k]!
    decreasing hdec : heapSize - idx + (if done then 0 else 1)
```

```
  do
    let l := left idx
    let r := right idx
    if l ≥ heapSize then
      -- idx is a leaf, heap property trivially satisfied
      done := true
    else
      let val_idx := a[idx]!
      let val_l := a[l]!
      if r < heapSize then
        let val_r := a[r]!
        if val_idx ≥ val_l ∧ val_idx ≥ val_r then
          done := true
        else
          if val_l ≥ val_r then
            swap! a[idx]! a[l]!
            idx := l
          else
            swap! a[idx]! a[r]!
            idx := r
      else
        if val_idx ≥ val_l then
          done := true
        else
          swap! a[idx]! a[l]!
          idx := l
  return a
theorem parent_left_eq (i : Nat) : parent (left i) = i := by
  simp only [parent, left]
  omega
theorem array_get_set_same {α : Type*} [Inhabited α] (a : Array α) (i : Nat) (v : α) (h : i < a.size) :
    (a.set! i v)[i]! = v := by
  simp only [Array.getElem!_eq_getD, Array.set!_eq_setIfInBounds, Array.getD]
  simp only [Array.getElem?_setIfInBounds, Array.size_setIfInBounds, h, ↓reduceIte]
  split_ifs with h1 <;> simp_all [Array.getElem_setIfInBounds]
theorem array_get_set_diff {α : Type*} [Inhabited α] (a : Array α) (i j : Nat) (v : α) (h : i ≠ j) :
    (a.set! i v)[j]! = a[j]! := by
  simp only [Array.getElem!_eq_getD, Array.set!_eq_setIfInBounds, Array.getD]
  simp only [Array.getElem?_setIfInBounds, Array.size_setIfInBounds]
  split_ifs with h1 h2 <;> simp_all
theorem parent_right_eq (i : Nat) : parent (right i) = i := by
  simp only [parent, right]
  omega
theorem array_get_set_same' {α : Type*} [Inhabited α] (a : Array α) (i : Nat) (v : α) (h : i < a.size) :
    (a.set! i v)[i]! = v := by
  simp only [Array.getElem!_eq_getD, Array.set!_eq_setIfInBounds, Array.getD]
  simp only [Array.getElem?_setIfInBounds, Array.size_setIfInBounds, h, ↓reduceIte]
  split_ifs with h1 <;> simp_all [Array.getElem_setIfInBounds]
theorem array_get_set_diff' {α : Type*} [Inhabited α] (a : Array α) (i j : Nat) (v : α) (h : i ≠ j) :
    (a.set! i v)[j]! = a[j]! := by
  simp only [Array.getElem!_eq_getD, Array.set!_eq_setIfInBounds, Array.getD]
  simp only [Array.getElem?_setIfInBounds, Array.size_setIfInBounds]
  split_ifs with h1 h2 <;> simp_all
theorem idx_ne_right_idx (idx : Nat) : idx ≠ right idx := by
  simp only [right]
  omega
theorem swap_perm {α : Type*} [Inhabited α] (a : Array α) (i j : Nat)
    (hi : i < a.size) (hj : j < a.size) :
    a.toList.Perm ((a.set! i a[j]!).set! j a[i]!).toList := by
  -- Array.multiset_swap a j i hj hi gives:
  -- ((a.set! i a[j]!).set! j a[i]!).toMultiset = a.toMultiset
  have h := Array.multiset_swap a j i hj hi
  simp only [Array.toMultiset] at h
  rw [Multiset.coe_eq_coe] at h
  exact h.symm
theorem parent_left_eq' (i : Nat) : parent (left i) = i := by
  simp only [parent, left]
  omega
--
-- theorem array_get_set_same' {α : Type*} [Inhabited α] (a : Array α) (i : Nat) (v : α) (h : i < a.size) :
--     (a.set! i v)[i]! = v := by
--   simp only [Array.getElem!_eq_getD, Array.set!_eq_setIfInBounds, Array.getD]
--   simp only [Array.getElem?_setIfInBounds, Array.size_setIfInBounds, h, ↓reduceIte]
--   split_ifs with h1 <;> simp_all [Array.getElem_setIfInBounds]
--
-- theorem array_get_set_diff' {α : Type*} [Inhabited α] (a : Array α) (i j : Nat) (v : α) (hij : i ≠ j) :
--     (a.set! i v)[j]! = a[j]! := by
--   simp only [Array.getElem!_eq_getD, Array.set!_eq_setIfInBounds, Array.getD]
--   simp only [Array.getElem?_setIfInBounds, Array.size_setIfInBounds]
--   split_ifs with h1 h2 <;> simp_all [Array.getElem_setIfInBounds, hij]
theorem left_ne_self (i : Nat) : left i ≠ i := by
```

```
  simp only [left]
  omega
theorem array_get_double_set_outside {α : Type*} [Inhabited α] (a : Array α) (i j k : Nat) (v w : α)
    (hi : i ≠ k) (hj : j ≠ k) :
    ((a.set! i v).set! j w)[k]! = a[k]! := by
  simp only [Array.getElem!_eq_getD, Array.set!_eq_setIfInBounds, Array.getD]
  simp only [Array.getElem?_setIfInBounds, Array.size_setIfInBounds]
  split_ifs with h1 h2 h3 h4 h5 h6 <;> try rfl
  all_goals simp_all [Array.getElem_setIfInBounds]
theorem array_get_set_neq {α : Type*} [Inhabited α] (a : Array α) (i j : Nat) (v : α) (hij : i ≠ j) :
    (a.set! i v)[j]! = a[j]! := by
  simp only [Array.getElem!_eq_getD, Array.set!_eq_setIfInBounds, Array.getD]
  simp only [Array.getElem?_setIfInBounds, Array.size_setIfInBounds]
  split_ifs with h1 h2 <;> simp_all [Array.getElem_setIfInBounds]
theorem right_ne_self (i : Nat) : right i ≠ i := by
  simp only [right]
  omega
theorem left_gt_self (i : Nat) : left i > i := by
  simp only [left]
  omega
theorem right_gt_self (i : Nat) : right i > i := by
  simp only [right]
  omega
theorem left_child_gt_right (i : Nat) : left (right i + 1) > right i := by
  simp only [left, right]
  omega
theorem right_child_gt_right (i : Nat) : right (right i + 1) > right i := by
  simp only [left, right]
  omega
theorem child_of_ge_right_plus_one_ne_idx (k idx : Nat) (hk : k ≥ right idx + 1) : left k ≠ idx ∧ right k ≠ idx
    ∧ left k ≠ right idx ∧ right k ≠ right idx ∧ k ≠ idx ∧ k ≠ right idx := by
  simp only [left, right] at *
  omega
theorem left_plus_one_eq_right (i : Nat) : left i + 1 = right i := by
  simp only [left, right]
theorem left_lt_left_succ (i : Nat) : left i < left i + 1 := by
 omega
theorem left_idx_plus_one_eq_right_idx (idx : Nat) : left idx + 1 = right idx := by
  simp only [left, right]
theorem left_child_gt (i : Nat) : left i > i := by
  simp only [left]
  omega
theorem right_child_gt (i : Nat) : right i > i := by
  simp only [right]
  omega
theorem left_of_ge_left (k idx : Nat) (h : k ≥ left idx + 1) : left k ≠ idx ∧ left k ≠ left idx := by
  simp only [left] at *
  constructor <;> omega
theorem right_of_ge_left (k idx : Nat) (h : k ≥ left idx + 1) : right k ≠ idx ∧ right k ≠ left idx := by
  simp only [left, right] at *
  constructor <;> omega
theorem k_ge_left_plus_one_ne (k idx : Nat) (h : k ≥ left idx + 1) : k ≠ idx ∧ k ≠ left idx := by
  simp only [left] at *
  constructor <;> omega
theorem child_of_ge_left_plus_one_ne_idx (k idx : Nat) (hk : k ≥ left idx + 1) :
    left k ≠ idx ∧ right k ≠ idx ∧ left k ≠ left idx ∧ right k ≠ left idx ∧ k ≠ idx ∧ k ≠ left idx := by
  simp only [left, right] at *
  omega
theorem children_of_ge_left_plus_one (k idx : Nat) (h : k ≥ left idx + 1) :
    k ≠ idx ∧ k ≠ left idx ∧ left k ≠ idx ∧ left k ≠ left idx ∧ right k ≠ idx ∧ right k ≠ left idx := by
  simp only [left, right] at *
  omega
theorem child_indices_disjoint_from_swap (k idx : Nat) (hk : k ≥ right idx + 1) :
    k ≠ idx ∧ k ≠ right idx ∧ left k ≠ idx ∧ left k ≠ right idx ∧ right k ≠ idx ∧ right k ≠ right idx := by
  simp only [left, right] at *
  omega
theorem left_right_ne_idx (idx : Nat) : left (right idx) ≠ idx := by
  simp only [left, right]
  omega
theorem left_right_ne_right (idx : Nat) : left (right idx) ≠ right idx := by
  simp only [left, right]
  omega
theorem right_idx_ge_i_plus_one (idx i : Nat) (h : right idx > i) : right idx ≥ i + 1 := by
  omega
theorem idx_ne_left_idx (idx : Nat) : idx ≠ left idx := by
  simp only [left]
  omega
theorem right_left_ne_idx (i : Nat) : right (left i) ≠ i := by
  simp only [right, left]
  omega
theorem right_left_ne_left (i : Nat) : right (left i) ≠ left i := by
```

```
  simp only [right, left]
  omega
theorem left_gt_implies_ne (idx : Nat) (h : left idx > idx) : left idx ≠ idx := by
  omega
theorem left_left_ne_idx (idx : Nat) : left (left idx) ≠ idx := by
  simp only [left]
  omega
theorem left_left_ne_left (idx : Nat) : left (left idx) ≠ left idx := by
  simp only [left]
  omega
theorem left_idx_ge_i (idx i : Nat) (h : left idx > i) : left idx ≥ i := by
  omega
theorem left_idx_ne_idx (idx : Nat) : left idx ≠ idx := by
  simp only [left]
  omega
set_option maxHeartbeats 10000000 in
prove_correct Heapify_ArrayPerm by
  loom_solve
  case hsize.loop_neg_pos_neg_pos => (
  try simp_all ; try grind
  )
  case hheap_except.loop_neg_pos_neg_pos.left => (
  -- We have: left idx < heapSize (from if_neg_1)
  have hl_idx : left idx < heapSize := by omega
  -- Size facts
  have hidx_lt : idx < a.size := by omega
  have hl_lt : left idx < a.size := by omega
  -- Key: a[left idx] > a[idx] (since we're swapping because idx doesn't dominate left child)
  have hswap_reason : a[left idx]! > a[idx]! := by
    by_contra hc
    push_neg at hc
    have : a[idx]! ≥ a[left idx]! ∧ a[idx]! ≥ a[right idx]! := ⟨hc, by omega⟩
    exact if_neg this
  -- Case analysis on k
  by_cases hk_eq_idx : k = idx
  · -- Case k = idx: need a'[k] ≥ a'[left k]
    subst hk_eq_idx
    have hne : left k ≠ k := left_ne_self k
    have ha'_k : ((a.set! k a[left k]!).set! (left k) a[k]!)[k]! = a[left k]! := by
      have h1 : (left k) ≠ k := hne
      rw [array_get_set_neq _ (left k) k _ h1]
      rw [array_get_set_same _ _ _ hidx_lt]
    have ha'_l : ((a.set! k a[left k]!).set! (left k) a[k]!)[left k]! = a[k]! := by
      have hsz : (a.set! k a[left k]!).size = a.size := by simp [Array.size_set!]
      rw [array_get_set_same _ _ _ (by omega : left k < (a.set! k a[left k]!).size)]
    rw [ha'_k, ha'_l]
    omega
  · -- Case k ≠ idx: use hheap_except
    have hidx_ne_k : idx ≠ k := fun h => hk_eq_idx h.symm
    have hlidx_ne_k : left idx ≠ k := fun h => h_3 h.symm
    by_cases hlk_idx : left k = idx
    · -- left k = idx means k is the parent of idx
      have hk_parent : k = parent idx := by
        simp only [left, parent] at hlk_idx ⊢
        omega
      -- a'[k] = a[k] since k ≠ idx and k ≠ left idx
      have ha'_k : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[k]! = a[k]! := by
        rw [array_get_double_set_outside a idx (left idx) k _ _ hidx_ne_k hlidx_ne_k]
      -- a'[idx] = a[left idx]
      have hne : left idx ≠ idx := left_ne_self idx
      have hidx_ne_lidx : idx ≠ left idx := hne.symm
      have ha'_idx : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[idx]! = a[left idx]! := by
        rw [array_get_set_neq _ (left idx) idx _ hne]
        rw [array_get_set_same _ _ _ hidx_lt]
      rw [hlk_idx, ha'_k, ha'_idx]
      -- We need a[k] ≥ a[left idx]
      have hidx_gt_i : idx > i := by
        simp only [left] at hlk_idx
        omega
      have hpc := hparent_children hidx_gt_i
      rw [hk_parent]
      exact hpc.1 hl_idx
    · -- left k ≠ idx
      have hlk_ne_idx : left k ≠ idx := hlk_idx
      have hidx_ne_lk : idx ≠ left k := hlk_ne_idx.symm
      by_cases hlk_lidx : left k = left idx
      · -- left k = left idx means k = idx (since left is injective)
        simp only [left] at hlk_lidx
        omega
      · -- left k ≠ idx and left k ≠ left idx, so a'[k] = a[k] and a'[left k] = a[left k]
        have hlidx_ne_lk : left idx ≠ left k := fun h => hlk_lidx h.symm
```

```
      have ha'_k : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[k]! = a[k]! := by
        rw [array_get_double_set_outside a idx (left idx) k _ _ hidx_ne_k hlidx_ne_k]
      have ha'_lk : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[left k]! = a[left k]! := by
        rw [array_get_double_set_outside a idx (left idx) (left k) _ _ hidx_ne_lk hlidx_ne_lk]
      rw [ha'_k, ha'_lk]
      exact (hheap_except k h_1 h_2 hk_eq_idx).1 h
)
case hheap_except.loop_neg_pos_neg_pos.right => (
-- We need to show heap property for k's right child after swap
-- Since k ≠ left idx, we need to check if k = idx
by_cases hk_idx : k = idx
· -- Case k = idx: after swap, a'[idx] = a[left idx] (the larger child)
  subst hk_idx
  -- Now idx is replaced by k everywhere. We have k ≠ left k by h_3
  -- a'[k] = a[left k], and we need a'[k] ≥ a'[right k]
  -- a'[right k] = a[right k] since right k ≠ k and right k ≠ left k
  have hrk_ne_k : right k ≠ k := right_ne_self k
  have hrk_ne_lk : right k ≠ left k := by simp [right, left]; omega
  have hk_ne_lk : k ≠ left k := h_3
  have hk_lt : k < a.size := by omega
  rw [array_get_set_neq _ (left k) k _ hk_ne_lk.symm]
  rw [array_get_set_same _ _ _ hk_lt]
  rw [array_get_set_neq _ (left k) (right k) _ hrk_ne_lk.symm]
  rw [array_get_set_neq _ k (right k) _ hrk_ne_k.symm]
  -- Now we need a[left k] ≥ a[right k], which is if_pos
  exact if_pos
· -- Case k ≠ idx: the swap doesn't affect a[k] unless k = left idx (but k ≠ left idx by h_3)
  -- So a'[k] = a[k]
  -- For a'[right k], we need right k ≠ idx and right k ≠ left idx
  have hk_ne_lidx : k ≠ left idx := h_3
  have hk_ne_idx : k ≠ idx := hk_idx
  -- a'[k] = a[k]
  have hak : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[k]! = a[k]! := by
    rw [array_get_set_neq _ (left idx) k _ hk_ne_lidx.symm]
    rw [array_get_set_neq _ idx k _ hk_ne_idx.symm]
  rw [hak]
  -- Now for right k
  by_cases hrk_idx : right k = idx
  · -- right k = idx: a'[right k] = a'[idx] = a[left idx]
    have hidx_ne_lidx : idx ≠ left idx := idx_ne_left_idx idx
    have hidx_lt : idx < a.size := by omega
    have hark : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[right k]! = a[left idx]! := by
      rw [hrk_idx]
      rw [array_get_set_neq _ (left idx) idx _ hidx_ne_lidx.symm]
      rw [array_get_set_same _ _ _ hidx_lt]
    rw [hark]
    -- We need a[k] ≥ a[left idx]
    -- Since right k = idx, we have k = parent idx
    -- From hparent_children: if idx > i then a[parent idx] ≥ a[left idx]
    have hk_parent : k = parent idx := by
      simp only [parent, right] at hrk_idx ⊢
      omega
    by_cases hidx_gt_i : idx > i
    · have hpc := hparent_children hidx_gt_i
      rw [hk_parent]
      have hleft_in : left idx < heapSize := by omega
      exact hpc.1 hleft_in
    · -- idx = i, so k = parent i
      -- But parent i < i for i > 0, and we have h_1 : i ≤ k
      -- If idx = i, then right k = i, so k = parent i = (i-1)/2
      -- But h_1 says i ≤ k, and parent i < i for i > 0
      -- So we need i = 0, but then parent 0 = 0, and k = 0
      -- But right 0 = 2, not 0. Contradiction.
      push_neg at hidx_gt_i
      have hidx_eq_i : idx = i := Nat.le_antisymm hidx_gt_i hidx_ge
      rw [hidx_eq_i] at hrk_idx hk_parent
      simp only [parent, right] at hk_parent hrk_idx
      omega
  · by_cases hrk_lidx : right k = left idx
    · -- right k = left idx: a'[right k] = a'[left idx] = a[idx]
      have hlidx_lt : left idx < (a.set! idx a[left idx]!).size := by
        simp only [Array.size_set!]
        omega
      have hark : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[right k]! = a[idx]! := by
        rw [hrk_lidx]
        rw [array_get_set_same _ _ _ hlidx_lt]
      rw [hark]
      -- We need a[k] ≥ a[idx]
      -- Since right k = left idx, k = parent (left idx) = idx
      -- But we assumed k ≠ idx. Contradiction.
      have hk_eq_idx : k = parent (left idx) := by
```

```
        simp only [parent, right, left] at hrk_lidx ⊢
        omega
      rw [parent_left_eq] at hk_eq_idx
      exact absurd hk_eq_idx hk_idx
    · -- right k ≠ idx and right k ≠ left idx: a'[right k] = a[right k]
      have hrk_ne_idx : right k ≠ idx := hrk_idx
      have hrk_ne_lidx : right k ≠ left idx := hrk_lidx
      have hark : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[right k]! = a[right k]! := by
        rw [array_get_set_neq _ (left idx) (right k) _ hrk_ne_lidx.symm]
        rw [array_get_set_neq _ idx (right k) _ hrk_ne_idx.symm]
      rw [hark]
      -- Now we use hheap_except since k ≠ idx
      have hhe := hheap_except k h_1 h_2 hk_ne_idx
      exact hhe.2 h
)
case hparent_dom.loop_neg_pos_neg_pos => (
-- First, simplify parent (left idx) = idx
rw [parent_left_eq]
-- We need to show: swapped_array[idx]! ≥ swapped_array[left idx]!
-- After swap: swapped_array[idx]! = a[left idx]! and swapped_array[left idx]! = a[idx]!
have hidx_lt : idx < a.size := by omega
have hleft_lt : left idx < a.size := by
  have : left idx < heapSize := by omega
  omega
have hne : idx ≠ left idx := idx_ne_left_idx idx
have hne' : left idx ≠ idx := fun h => hne h.symm
-- Value at idx after swap
have h1 : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[idx]! = a[left idx]! := by
  rw [array_get_set_neq _ _ _ _ hne']
  rw [array_get_set_same _ _ _ hidx_lt]
-- Value at left idx after swap
have h2 : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[left idx]! = a[idx]! := by
  have hsize_after : (a.set! idx a[left idx]!).size = a.size := by simp [Array.size_set!]
  rw [array_get_set_same _ _ _ (by omega : left idx < (a.set! idx a[left idx]!).size)]
rw [h1, h2]
-- Now we need a[left idx]! ≥ a[idx]!
-- From if_neg we have ¬(a[idx]! ≥ a[left idx]! ∧ a[idx]! ≥ a[right idx]!)
-- From if_pos we have a[left idx]! ≥ a[right idx]!
-- So we must have ¬(a[idx]! ≥ a[left idx]!) OR ¬(a[idx]! ≥ a[right idx]!)
-- Use De Morgan's law
rw [not_and_or] at if_neg
cases if_neg with
| inl h_not_ge_left =>
  -- ¬(a[idx]! ≥ a[left idx]!) means a[idx]! < a[left idx]!
  omega
| inr h_not_ge_right =>
  -- ¬(a[idx]! ≥ a[right idx]!) means a[idx]! < a[right idx]!
  -- Combined with if_pos: a[left idx]! ≥ a[right idx]!
  -- We get a[left idx]! ≥ a[right idx]! > a[idx]!
  omega
)
case hparent_children.loop_neg_pos_neg_pos.left => (
-- parent (left idx) = idx
rw [parent_left_eq]
-- After swap: a'[idx] = a[left idx], a'[left (left idx)] = a[left (left idx)]
have h_left_lt : left idx < heapSize := by omega
have h_left_ge_i : left idx ≥ i := left_idx_ge_i idx i h_1
have h_left_ne_idx : left idx ≠ idx := left_idx_ne_idx idx
-- Get heap property for left idx from hheap_except
have hheap_left := hheap_except (left idx) h_left_ge_i h_left_lt h_left_ne_idx
have h_dom_left_left : a[left idx]! ≥ a[left (left idx)]! := hheap_left.1 h
-- Now simplify the array accesses after swap
have h_ll_ne_idx : left (left idx) ≠ idx := left_left_ne_idx idx
have h_ll_ne_left : left (left idx) ≠ left idx := left_left_ne_left idx
have h_idx_lt : idx < a.size := by omega
have h_left_idx_lt : left idx < a.size := by omega
-- a'[idx] = a[left idx]
-- The outer set is at (left idx), we access at idx, so (left idx) ≠ idx means we look through
-- The inner set is at idx, we access at idx, so we get the value
have h_get_idx : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[idx]! = a[left idx]! := by
  have hne : left idx ≠ idx := left_idx_ne_idx idx
  rw [array_get_set_neq (a.set! idx a[left idx]!) (left idx) idx a[idx]! hne]
  rw [array_get_set_same a idx a[left idx]! h_idx_lt]
-- a'[left (left idx)] = a[left (left idx)]
-- The outer set is at (left idx), we access at (left (left idx)), so (left idx) ≠ (left (left idx))
-- The inner set is at idx, we access at (left (left idx)), so idx ≠ (left (left idx))
have h_get_ll : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[left (left idx)]! = a[left (left idx)]! :=
   by
  have hne1 : left idx ≠ left (left idx) := h_ll_ne_left.symm
  have hne2 : idx ≠ left (left idx) := h_ll_ne_idx.symm
  rw [array_get_set_neq (a.set! idx a[left idx]!) (left idx) (left (left idx)) a[idx]! hne1]
```

```
  rw [array_get_set_neq a idx (left (left idx)) a[left idx]! hne2]
rw [h_get_idx, h_get_ll]
exact h_dom_left_left
)
case hparent_children.loop_neg_pos_neg_pos.right => (
-- parent (left idx) = idx
have hparent : parent (left idx) = idx := parent_left_eq idx
rw [hparent]
-- The value at idx after swap is a[left idx]
have hidx_lt : idx < a.size := by omega
have hleft_lt : left idx < a.size := by
  have : left idx < heapSize := by omega
  omega
have hright_left_ne_idx : right (left idx) ≠ idx := right_left_ne_idx idx
have hright_left_ne_left : right (left idx) ≠ left idx := right_left_ne_left idx
have hleft_ne_idx : left idx ≠ idx := left_gt_implies_ne idx (left_gt_self idx)
-- After first set: (a.set! idx a[left idx]!)[idx] = a[left idx]
-- After second set at left idx: value at idx unchanged since left idx ≠ idx
have hval_idx : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[idx]! = a[left idx]! := by
  rw [array_get_set_neq (a.set! idx a[left idx]!) (left idx) idx a[idx]! hleft_ne_idx]
  rw [array_get_set_same _ _ _ hidx_lt]
-- Value at right (left idx) unchanged by swap since it's different from both idx and left idx
have hval_right : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[right (left idx)]! = a[right (left idx)]!
   := by
  rw [array_get_set_neq (a.set! idx a[left idx]!) (left idx) (right (left idx)) a[idx]!
   hright_left_ne_left.symm]
  rw [array_get_set_neq a idx (right (left idx)) a[left idx]! hright_left_ne_idx.symm]
rw [hval_idx, hval_right]
-- Now need: a[left idx]! ≥ a[right (left idx)]!
-- Since left idx > i, we have left idx ≥ i + 1, so i ≤ left idx
-- left idx < heapSize (from if_neg_1)
-- left idx ≠ idx (proven above)
-- So by hheap_except, a[left idx] dominates its children
have hi_le_left : i ≤ left idx := by omega
have hleft_lt_heap : left idx < heapSize := by omega
have hheap_left := hheap_except (left idx) hi_le_left hleft_lt_heap hleft_ne_idx
exact hheap_left.2 h
)
case hperm.loop_neg_pos_neg_pos => (
have hidx_lt : idx < a.size := by omega
have hl_lt : left idx < a.size := by
  simp only [left] at if_neg_1 ⊢
  omega
have h_swap : a.toList.Perm ((a.set! idx a[left idx]!).set! (left idx) a[idx]!).toList := by
  exact swap_perm a idx (left idx) hidx_lt hl_lt
exact hperm.trans h_swap
)
case hframe.loop_neg_pos_neg_pos => (
-- k ≥ heapSize, idx < heapSize, left idx < heapSize (from if_neg_1)
-- So k ≠ idx and k ≠ left idx
have hl_lt : left idx < heapSize := by omega
have hk_ne_idx : k ≠ idx := by omega
have hk_ne_left : k ≠ left idx := by omega
-- The swap doesn't affect position k
rw [array_get_set_neq _ _ _ _ hk_ne_left.symm]
rw [array_get_set_neq _ _ _ _ hk_ne_idx.symm]
-- Now use the frame hypothesis
exact hframe k h_1 h
)
case hsize.loop_neg_pos_neg_neg => (
try simp_all ; try grind
)
case hheap_except.loop_neg_pos_neg_neg.left => (
-- We need to show the heap property for k after swapping idx with right idx
-- Case split on whether k = idx
by_cases hk_eq_idx : k = idx
· -- k = idx: after swap, a'[idx] = a[right idx] (the larger child)
  -- We need a'[idx] ≥ a'[left idx]
  subst hk_eq_idx
  -- Now idx is replaced by k everywhere
  -- a'[k] = a[right k], a'[left k] = a[left k] (since left k ≠ k and left k ≠ right k)
  have h_left_ne_k : left k ≠ k := by simp only [left]; omega
  have h_left_ne_right : left k ≠ right k := by simp only [left, right]; omega
  rw [array_get_set_neq _ (right k) k _ (by simp only [right]; omega)]
  rw [array_get_set_same _ k _ (by omega)]
  rw [array_get_set_neq _ (right k) (left k) _ h_left_ne_right.symm]
  rw [array_get_set_neq _ k (left k) _ h_left_ne_k.symm]
  -- Now we need a[right k] ≥ a[left k]
  -- From if_neg, we have ¬(a[left k] ≥ a[right k]), so a[right k] > a[left k]
  push_neg at if_neg
  omega
```

```
· -- k ≠ idx: the value at k is unchanged (since k ≠ idx and k ≠ right idx)
  -- Need to check if left k is affected
  have hk_ne_right : k ≠ right idx := h_3
  have hk_ne_idx' : idx ≠ k := fun heq => hk_eq_idx heq.symm
  -- Check cases for left k
  by_cases hlk_eq_idx : left k = idx
  · -- left k = idx: means k is parent of idx
    -- If k is parent of idx, then parent idx = k
    have hpar : parent idx = k := by
      simp only [left] at hlk_eq_idx
      simp only [parent]
      omega
    -- Since k ≠ idx, and parent idx = k, we need a'[k] ≥ a'[idx]
    -- a'[k] = a[k] (k ≠ idx, k ≠ right idx)
    -- a'[idx] = a[right idx]
    rw [array_get_set_neq _ (right idx) k _ hk_ne_right.symm]
    rw [array_get_set_neq _ idx k _ hk_ne_idx']
    rw [hlk_eq_idx]
    rw [array_get_set_neq _ (right idx) idx _ (by simp only [right]; omega)]
    rw [array_get_set_same _ idx _ (by omega)]
    -- Need a[k] ≥ a[right idx]
    -- From hparent_children: if idx > i then parent idx dominates children of idx
    by_cases hidx_gt_i : idx > i
    · have hpc := hparent_children hidx_gt_i
      -- hpc says a[parent idx] ≥ a[right idx], and parent idx = k
      rw [hpar] at hpc
      exact hpc.2 if_pos
    · -- idx = i (since idx ≥ i)
      have hidx_eq : idx = i := by omega
      -- But then parent i = k and left k = i, so k is parent of i
      -- We have k ≥ i from h_1, so k ≥ i
      -- parent i = (i - 1) / 2
      -- If i = 0, parent 0 = 0, but left 0 = 1 ≠ 0 = i, contradiction with hlk_eq_idx
      -- If i > 0, parent i < i, but k = parent i and k ≥ i, contradiction
      simp only [parent, left] at *
      omega
  · by_cases hlk_eq_right : left k = right idx
    · -- left k = right idx: means k = idx (since left k = 2k + 1 and right idx = 2*idx + 2)
      have : k = idx := by simp only [left, right] at hlk_eq_right; omega
      contradiction
    · -- left k ≠ idx and left k ≠ right idx: both k and left k are unaffected
      have hlk_ne_idx' : idx ≠ left k := fun heq => hlk_eq_idx heq.symm
      have hlk_ne_right' : right idx ≠ left k := fun heq => hlk_eq_right heq.symm
      rw [array_get_set_neq _ (right idx) k _ hk_ne_right.symm]
      rw [array_get_set_neq _ idx k _ hk_ne_idx']
      rw [array_get_set_neq _ (right idx) (left k) _ hlk_ne_right']
      rw [array_get_set_neq _ idx (left k) _ hlk_ne_idx']
      -- Now use hheap_except for k
      have hhe := hheap_except k h_1 h_2 hk_eq_idx
      exact hhe.1 h
)
case hheap_except.loop_neg_pos_neg_neg.right => (
-- Case analysis on whether k = idx
by_cases hk_idx : k = idx
· -- Case k = idx
  subst hk_idx
  -- After substitution, we need to show the result for k (which was idx)
  -- h_3 becomes: k ≠ right k
  -- The swapped array is (a.set! k a[right k]!).set! (right k) a[k]!
  -- We need: swapped[k] ≥ swapped[right k]
  -- swapped[k] = a[right k] (since k ≠ right k, inner set applies)
  -- swapped[right k] = a[k] (outer set applies)
  have hk_ne_rk : k ≠ right k := by simp [right]; omega
  have hk_lt : k < a.size := by omega
  have hrk_lt : right k < a.size := by omega
  -- swapped[k]
  have ha'_k : ((a.set! k a[right k]!).set! (right k) a[k]!)[k]! = a[right k]! := by
    rw [array_get_set_neq _ _ _ _ (Ne.symm h_3)]
    rw [array_get_set_same _ _ _ hk_lt]
  -- swapped[right k]
  have ha'_rk : ((a.set! k a[right k]!).set! (right k) a[k]!)[right k]! = a[k]! := by
    rw [array_get_set_same _ _ _ (by simp [Array.size_set]; omega)]
  rw [ha'_k, ha'_rk]
  -- We need a[right k] ≥ a[k]
  -- From if_neg_2: ¬(a[k]! ≥ a[left k]! ∧ a[k]! ≥ a[right k]!)
  -- From if_neg: ¬(a[left k]! ≥ a[right k]!), i.e., a[left k]! < a[right k]!
  -- So the negation means: a[k]! < a[left k]! ∨ a[k]! < a[right k]!
  -- Since a[left k] < a[right k], if a[k] ≥ a[right k] then a[k] ≥ a[left k] too
  -- So we must have a[k]! < a[right k]!
  have h_neg2 : a[k]! < a[left k]! ∨ a[k]! < a[right k]! := by
    by_contra h_contra
```

```
    push_neg at h_contra
    exact if_neg_2 h_contra
  have h_neg : a[left k]! < a[right k]! := by omega
  rcases h_neg2 with hlt_l | hlt_r
  · -- a[k] < a[left k] < a[right k]
    omega
  · -- a[k] < a[right k]
    omega
· -- Case k ≠ idx
  -- Convert hk_idx to the form needed by the lemma
  have hidx_ne_k : idx ≠ k := fun h => hk_idx h.symm
  have hridx_ne_k : right idx ≠ k := fun h => h_3 h.symm
  -- Since k ≠ idx and k ≠ right idx, a'[k] = a[k]
  have ha'_k : ((a.set! idx a[right idx]!).set! (right idx) a[idx]!)[k]! = a[k]! := by
    rw [array_get_double_set_outside a idx (right idx) k _ _ hidx_ne_k hridx_ne_k]
  -- Now we need to check if right k is idx or right idx
  by_cases hrk_idx : right k = idx
  · -- right k = idx
    have h_idx_ne_ridx : idx ≠ right idx := by simp [right]; omega
    have hidx_lt : idx < a.size := by omega
    have ha'_rk : ((a.set! idx a[right idx]!).set! (right idx) a[idx]!)[right k]! = a[right idx]! := by
      rw [hrk_idx]
      rw [array_get_set_neq _ _ _ _ (Ne.symm h_idx_ne_ridx)]
      rw [array_get_set_same _ _ _ hidx_lt]
    rw [ha'_k, ha'_rk]
    -- We need a[k] ≥ a[right idx]
    -- Since right k = idx, we have k = parent idx
    have hidx_gt_i : idx > i := by
      simp [right] at hrk_idx
      omega
    have hp_children := hparent_children hidx_gt_i
    have hk_parent : k = parent idx := by
      simp [parent, right] at *
      omega
    rw [hk_parent]
    exact hp_children.2 if_pos
  · by_cases hrk_ridx : right k = right idx
    · -- right k = right idx implies k = idx (right is injective), contradiction
      simp [right] at hrk_ridx
      omega
    · -- right k ≠ idx and right k ≠ right idx
      have hidx_ne_rk : idx ≠ right k := fun h => hrk_idx h.symm
      have hridx_ne_rk : right idx ≠ right k := fun h => hrk_ridx h.symm
      have ha'_rk : ((a.set! idx a[right idx]!).set! (right idx) a[idx]!)[right k]! = a[right k]! := by
        rw [array_get_double_set_outside a idx (right idx) (right k) _ _ hidx_ne_rk hridx_ne_rk]
      rw [ha'_k, ha'_rk]
      exact (hheap_except k h_1 h_2 hk_idx).2 h
)
case hparent_dom.loop_neg_pos_neg_neg => (
-- First, simplify parent (right idx) = idx
have hpar : parent (right idx) = idx := parent_right_eq idx
rw [hpar]
-- idx ≠ right idx
have hidx_ne_right : idx ≠ right idx := idx_ne_right_idx idx
-- Get bounds needed for array access
have hidx_lt : idx < a.size := by omega
have hright_lt : right idx < a.size := by omega
-- After the swap, value at idx is a[right idx]!
have hget_idx : ((a.set! idx a[right idx]!).set! (right idx) a[idx]!)[idx]! = a[right idx]! := by
  rw [array_get_set_neq _ _ _ _ hidx_ne_right.symm]
  exact array_get_set_same a idx a[right idx]! hidx_lt
-- After the swap, value at right idx is a[idx]!
have hget_right : ((a.set! idx a[right idx]!).set! (right idx) a[idx]!)[right idx]! = a[idx]! := by
  have hright_lt' : right idx < (a.set! idx a[right idx]!).size := by simp [Array.size_set!, hidx_lt, hright_lt]
  exact array_get_set_same (a.set! idx a[right idx]!) (right idx) a[idx]! hright_lt'
rw [hget_idx, hget_right]
-- Now we need a[right idx]! ≥ a[idx]!
-- From if_neg: ¬(a[left idx]! ≥ a[right idx]!), so a[right idx]! > a[left idx]!
have hright_gt_left : a[right idx]! > a[left idx]! := by
  push_neg at if_neg
  exact if_neg
-- From if_neg_2: ¬(a[idx]! ≥ a[left idx]! ∧ a[idx]! ≥ a[right idx]!)
-- Either a[idx]! < a[left idx]! or a[idx]! < a[right idx]!
by_cases h_case : a[idx]! ≥ a[left idx]!
· -- If a[idx]! ≥ a[left idx]!, then from if_neg_2 we must have a[idx]! < a[right idx]!
  have : ¬(a[idx]! ≥ a[right idx]!) := by
    intro hcontra
    exact if_neg_2 ⟨h_case, hcontra⟩
  omega
· -- If a[idx]! < a[left idx]!, then since a[right idx]! > a[left idx]!, we have a[right idx]! > a[idx]!
  omega
```

```
)
case hparent_children.loop_neg_pos_neg_neg.left => (
-- parent (right idx) = idx
have hpar : parent (right idx) = idx := parent_right_eq idx
rw [hpar]
-- The value at idx after swap is a[right idx]!
have hidx_lt : idx < a.size := by omega
have hr_lt : right idx < a.size := by omega
have h_ne : idx ≠ right idx := idx_ne_right_idx idx
have hget_idx : ((a.set! idx a[right idx]!).set! (right idx) a[idx]!)[idx]! = a[right idx]! := by
  rw [array_get_set_neq _ (right idx) idx _ (Ne.symm h_ne)]
  rw [array_get_set_same _ idx _ hidx_lt]
rw [hget_idx]
-- The value at left (right idx) is unchanged (it's neither idx nor right idx)
have hlr_ne_idx : left (right idx) ≠ idx := left_right_ne_idx idx
have hlr_ne_r : left (right idx) ≠ right idx := left_right_ne_right idx
have hget_lr : ((a.set! idx a[right idx]!).set! (right idx) a[idx]!)[left (right idx)]! = a[left (right idx)]!
   := by
  rw [array_get_set_neq _ (right idx) (left (right idx)) _ (Ne.symm hlr_ne_r)]
  rw [array_get_set_neq _ idx (left (right idx)) _ (Ne.symm hlr_ne_idx)]
rw [hget_lr]
-- Now we need a[right idx]! ≥ a[left (right idx)]!
-- Since right idx > i, we have right idx ≥ i + 1
-- And right idx < heapSize, so right idx is in the heap region [i+1, heapSize)
-- By hheap_except, since right idx ≠ idx, the heap property holds at right idx
have hri_ge : right idx ≥ i + 1 := right_idx_ge_i_plus_one idx i h_1
have hri_ge_i : i ≤ right idx := by omega
have hri_ne_idx : right idx ≠ idx := Ne.symm h_ne
have hheap_ri := hheap_except (right idx) hri_ge_i if_pos hri_ne_idx
exact hheap_ri.1 h
)
case hparent_children.loop_neg_pos_neg_neg.right => (
-- parent (right idx) = idx
have hparent_eq : parent (right idx) = idx := parent_right_eq idx
rw [hparent_eq]
-- idx ≠ right idx
have hne1 : idx ≠ right idx := idx_ne_right_idx idx
-- right idx ≠ right (right idx)
have hne2 : right idx ≠ right (right idx) := by simp only [right]; omega
-- idx ≠ right (right idx)
have hne3 : idx ≠ right (right idx) := by simp only [right]; omega
-- Simplify LHS: ((a.set! idx a[right idx]!).set! (right idx) a[idx]!)[idx]!
have hlhs : ((a.set! idx a[right idx]!).set! (right idx) a[idx]!)[idx]! = a[right idx]! := by
  rw [array_get_set_neq _ _ _ _ hne1.symm]
  have hidx_sz : idx < a.size := by omega
  exact array_get_set_same a idx (a[right idx]!) hidx_sz
-- Simplify RHS: ((a.set! idx a[right idx]!).set! (right idx) a[idx]!)[right (right idx)]!
-- First peel off the outer set at (right idx), accessing at (right (right idx))
-- Need: right idx ≠ right (right idx)
have hrhs : ((a.set! idx a[right idx]!).set! (right idx) a[idx]!)[right (right idx)]! = a[right (right idx)]! :=
    by
  rw [array_get_set_neq _ _ _ _ hne2]
  rw [array_get_set_neq _ _ _ _ hne3]
rw [hlhs, hrhs]
-- Now we need: a[right idx]! ≥ a[right (right idx)]!
-- This follows from hheap_except applied to k = right idx
have hridx_ne_idx : right idx ≠ idx := by simp only [right]; omega
have hridx_ge_i : i ≤ right idx := by
  have : right idx > idx := right_gt_self idx
  omega
have hridx_lt : right idx < heapSize := if_pos
have hheap_ridx := hheap_except (right idx) hridx_ge_i hridx_lt hridx_ne_idx
exact hheap_ridx.2 h
)
case hperm.loop_neg_pos_neg_neg => (
-- First establish size bounds
have hidx_lt : idx < a.size := by omega
have hr_lt : right idx < a.size := by
  simp only [right] at if_pos ⊢
  omega
-- Use swap_perm to show a.toList.Perm (swapped array).toList
have hswap : a.toList.Perm ((a.set! idx a[right idx]!).set! (right idx) a[idx]!).toList := by
  exact swap_perm a idx (right idx) hidx_lt hr_lt
-- Compose the permutations
exact List.Perm.trans hperm hswap
)
case hframe.loop_neg_pos_neg_neg => (
-- k ≥ heapSize and idx < heapSize, so idx ≠ k
have hidx_ne_k : idx ≠ k := by omega
-- k ≥ heapSize and right idx < heapSize, so right idx ≠ k
have hright_ne_k : right idx ≠ k := by omega
```

```
-- Use the double set lemma
rw [array_get_double_set_outside a idx (right idx) k _ _ hidx_ne_k hright_ne_k]
-- Now use hframe
exact hframe k h_1 h
)
case hsize.loop_neg_neg_neg => (
try simp_all ; try grind
)
case hheap_except.loop_neg_neg_neg.left => (
-- We need to show the heap property for k w.r.t. left k after swapping idx and left idx
-- Key facts: k ≠ left idx (given as h_3)
-- We need to consider whether k = idx or not
have hidx_lt : idx < a.size := by omega
have hleft_idx_lt : left idx < a.size := by omega
have hne_idx_lidx : left idx ≠ idx := left_ne_self idx
have hidx_lt_set : idx < (a.set! idx a[left idx]!).size := by
  simp [Array.size_set!]; omega
have hleft_idx_lt_set : left idx < (a.set! idx a[left idx]!).size := by
  simp [Array.size_set!]; omega
by_cases hk_idx : k = idx
· -- Case k = idx: after swap, a'[idx] = a[left idx]
  -- We need a'[idx] ≥ a'[left idx]
  -- a'[idx] = a[left idx], a'[left idx] = a[idx]
  -- So we need a[left idx] ≥ a[idx], but we have ¬(a[idx] ≥ a[left idx])
  -- This means a[left idx] > a[idx], so a[left idx] ≥ a[idx] ✓
  rw [hk_idx]
  -- Now goal is: a'[idx] ≥ a'[left idx]
  -- For a'[idx]: outer set is at (left idx), we access idx, so left idx ≠ idx means outer doesn't affect
  -- Then inner set is at idx, we access idx, so we get the value a[left idx]!
  have h1 : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[idx]! = a[left idx]! := by
    have hne : left idx ≠ idx := left_ne_self idx
    rw [array_get_set_diff (a.set! idx a[left idx]!) (left idx) idx a[idx]! hne]
    rw [array_get_set_same a idx a[left idx]! hidx_lt]
  -- a'[left idx] = a[idx]
  have h2 : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[left idx]! = a[idx]! := by
    rw [array_get_set_same (a.set! idx a[left idx]!) (left idx) a[idx]! hleft_idx_lt_set]
  rw [h1, h2]
  -- Need: a[left idx]! ≥ a[idx]!
  -- From if_neg: ¬(a[idx]! ≥ a[left idx]!), so a[idx]! < a[left idx]!
  omega
· -- Case k ≠ idx: use hheap_except to get the original heap property
  have hheap_k := hheap_except k h_1 h_2 hk_idx
  have hleft_k := hheap_k.1 h
  -- Now we need to show a'[k] ≥ a'[left k] where a' is the swapped array
  -- Since k ≠ idx and k ≠ left idx, and we need to check if left k is affected
  -- left k could be idx or left idx
  -- Get a'[k]
  have hk_ne_lidx : k ≠ left idx := h_3
  have hk_ne_idx : k ≠ idx := hk_idx
  have hk_val : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[k]! = a[k]! := by
    rw [array_get_set_diff (a.set! idx a[left idx]!) (left idx) k a[idx]! (Ne.symm hk_ne_lidx)]
    rw [array_get_set_diff a idx k a[left idx]! (Ne.symm hk_ne_idx)]
  by_cases hlk_idx : left k = idx
  · -- left k = idx: a'[left k] = a'[idx] = a[left idx]
    have h_val : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[left k]! = a[left idx]! := by
      rw [hlk_idx]
      rw [array_get_set_diff (a.set! idx a[left idx]!) (left idx) idx a[idx]! hne_idx_lidx]
      rw [array_get_set_same a idx a[left idx]! hidx_lt]
    rw [hk_val, h_val]
    -- Need: a[k] ≥ a[left idx]
    -- left k = idx means k = parent idx
    have hk_parent : k = parent idx := by
      have : left k = idx := hlk_idx
      simp only [left] at this
      simp only [parent]
      omega
    -- Since k ≥ i and k = parent idx, and idx ≥ i
    -- If idx > i, then hparent_children gives us a[parent idx] ≥ a[left idx]
    by_cases hidx_gt_i : idx > i
    · have hp := hparent_children hidx_gt_i
      have hleft_idx_in : left idx < heapSize := by omega
      have := hp.1 hleft_idx_in
      rw [← hk_parent] at this
      exact this
    · -- idx = i (since idx ≥ i and not idx > i)
      have hidx_eq_i : idx = i := by omega
      -- Then k = parent i, and left k = i means k = parent i
      -- But then k < i (since parent i < i for i > 0)
      -- But we have k ≥ i, contradiction unless i = 0
      simp only [parent, left] at hk_parent hlk_idx
      omega
```

```
  · by_cases hlk_lidx : left k = left idx
    · -- left k = left idx: a'[left k] = a[idx]
      have h_val : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[left k]! = a[idx]! := by
        rw [hlk_lidx]
        rw [array_get_set_same (a.set! idx a[left idx]!) (left idx) a[idx]! hleft_idx_lt_set]
      rw [hk_val, h_val]
      -- Need: a[k] ≥ a[idx]
      -- left k = left idx means k = idx, but k ≠ idx, contradiction
      simp only [left] at hlk_lidx
      omega
    · -- left k is not affected by the swap
      have hlk_ne_lidx : left k ≠ left idx := hlk_lidx
      have hlk_ne_idx : left k ≠ idx := hlk_idx
      have h_val : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[left k]! = a[left k]! := by
        rw [array_get_set_diff (a.set! idx a[left idx]!) (left idx) (left k) a[idx]! (Ne.symm hlk_ne_lidx)]
        rw [array_get_set_diff a idx (left k) a[left idx]! (Ne.symm hlk_ne_idx)]
      rw [hk_val, h_val]
      exact hleft_k
)
case hheap_except.loop_neg_neg_neg.right => (
-- First, k ≠ idx because otherwise right k = right idx < heapSize contradicts if_neg_2
have hk_ne_idx : k ≠ idx := by
  intro hk_eq; subst hk_eq; exact if_neg_2 h
-- Array element at k is unchanged since k ≠ idx and k ≠ left idx
have ha_k : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[k]! = a[k]! := by
  rw [array_get_set_neq _ _ _ _ (Ne.symm h_3)]
  rw [array_get_set_neq _ _ _ _ (Ne.symm hk_ne_idx)]
rw [ha_k]
-- Now consider cases for right k
by_cases hrk_idx : right k = idx
· -- Case: right k = idx
  -- After swap, a'[idx] = a[left idx]
  have ha_rk : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[right k]! = a[left idx]! := by
    rw [hrk_idx]
    rw [array_get_set_neq]
    · have hidx_lt : idx < a.size := by omega
      rw [array_get_set_same _ _ _ hidx_lt]
    · simp only [left]; omega
  rw [ha_rk]
  -- We need a[k]! ≥ a[left idx]!
  -- Since right k = idx, we have parent idx = k
  have hp : parent idx = k := by simp only [parent, right] at *; omega
  -- idx > i because idx = right k = 2*k + 2 and k ≥ i, so idx > i
  have hidx_gt_i : idx > i := by simp only [right] at hrk_idx; omega
  have hpc := hparent_children hidx_gt_i
  have hl_idx : left idx < heapSize := by omega
  rw [← hp]
  exact hpc.1 hl_idx
· -- Case: right k ≠ idx
  by_cases hrk_lidx : right k = left idx
  · -- Case: right k = left idx
    -- This case is impossible: right k = 2*k + 2, left idx = 2*idx + 1
    -- 2*k + 2 = 2*idx + 1 means 2*(k - idx) = -1, impossible for naturals
    exfalso
    simp only [right, left] at hrk_lidx
    omega
  · -- Case: right k ≠ idx and right k ≠ left idx
    have ha_rk : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[right k]! = a[right k]! := by
      rw [array_get_set_neq _ _ _ _ (Ne.symm hrk_lidx)]
      rw [array_get_set_neq _ _ _ _ (Ne.symm hrk_idx)]
    rw [ha_rk]
    -- Use hheap_except since k ≠ idx
    exact (hheap_except k h_1 h_2 hk_ne_idx).2 h
)
case hparent_dom.loop_neg_neg_neg => (
-- First, simplify parent (left idx) = idx
have hparent : parent (left idx) = idx := parent_left_eq idx
rw [hparent]
-- We need to show that the value at idx after swap ≥ value at left idx after swap
-- After swap: a[idx] gets a[left idx], and a[left idx] gets a[idx]
-- So we need: a[left idx]! ≥ a[idx]!
-- This follows from ¬(a[idx]! ≥ a[left idx]!)
have hidx_lt : idx < a.size := by omega
have hleft_lt : left idx < a.size := by
  have : left idx < heapSize := by omega
  omega
have hleft_ne_idx : left idx ≠ idx := left_ne_self idx
-- Value at idx after swap
have hval_idx : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[idx]! = a[left idx]! := by
  rw [array_get_set_neq _ _ _ _ hleft_ne_idx]
  rw [array_get_set_same _ _ _ hidx_lt]
```

```
    -- Value at left idx after swap
    have hval_left : ((a.set! idx a[left idx]!).set! (left idx) a[idx]!)[left idx]! = a[idx]! := by
      have hsize_after : (a.set! idx a[left idx]!).size = a.size := by simp [Array.size_set!]
      rw [array_get_set_same _ _ _ (by rw [hsize_after]; exact hleft_lt)]
    rw [hval_idx, hval_left]
    -- Now we need: a[left idx]! ≥ a[idx]!
    -- From if_neg: ¬(a[idx]! ≥ a[left idx]!), so a[idx]! < a[left idx]!
    omega
    )
    case hperm.loop_neg_neg_neg => (
    -- First establish bounds
    have hidx_lt : idx < a.size := by omega
    have hl_lt_heap : left idx < heapSize := by omega
    have hl_lt : left idx < a.size := by omega
    -- The swap preserves permutation: a.toList.Perm (swap result).toList
    have hswap : a.toList.Perm ((a.set! idx a[left idx]!).set! (left idx) a[idx]!).toList := by
      have h := Array.multiset_swap a (left idx) idx hl_lt hidx_lt
      simp only [Array.toMultiset] at h
      rw [Multiset.coe_eq_coe] at h
      exact h.symm
    -- By transitivity: arr ~ a and a ~ (swap result), so arr ~ (swap result)
    exact hperm.trans hswap
    )
    case hframe.loop_neg_neg_neg => (
    -- k is outside the heap region (k ≥ heapSize)
    -- idx and left idx are inside the heap region (< heapSize)
    -- So the swap doesn't affect position k
    have h_idx_ne_k : idx ≠ k := by omega
    have h_left_lt : left idx < heapSize := by omega
    have h_left_ne_k : left idx ≠ k := by omega
    -- The swap at idx and left idx doesn't change position k
    rw [array_get_double_set_outside a idx (left idx) k _ _ h_idx_ne_k h_left_ne_k]
    -- Now use hframe to conclude
    exact hframe k h_1 h
    )
    case hheap_except.entry.left => (
    have hk_ge : k ≥ i + 1 := by omega
    unfold isMaxHeapOn at require_2
    have hk_in_range : i + 1 ≤ k ∧ k < heapSize := ⟨hk_ge, h_2⟩
    have := require_2 k hk_in_range
    exact this.1 h
    )
    case hheap_except.entry.right => (
    have hk_ge : k ≥ i + 1 := by omega
    have hk_in_range : i + 1 ≤ k ∧ k < heapSize := ⟨hk_ge, h_2⟩
    unfold isMaxHeapOn at require_2
    have := require_2 k hk_in_range
    exact this.2 h
    )
    case hperm.entry => (
    try simp_all ; try grind
    )
    case ensures_3 => (
    obtain ⟨ha_eq, _, _⟩ := h
    subst ha_eq
    unfold frameOutsidePrefix
    constructor
    · exact hsize.symm
    · intro k hk hge
      exact hframe k hk hge
    )
    case ensures_1 => (
    obtain ⟨ha_eq, hcont_eq, hidx_eq⟩ := h
    subst ha_eq
    -- Eliminate double negation
    have hdone_true : done_1 = true := by
      by_contra h_not
      exact done h_not
    have h_at_idx := hdone_ok hdone_true
    unfold isMaxHeapOn
    intro k ⟨hk_ge, hk_lt⟩
    by_cases hk_eq_idx : k = idx_1
    · -- Case k = idx_1: use h_at_idx
      rw [hk_eq_idx]
      exact h_at_idx
    · -- Case k ≠ idx_1: use hheap_except
      exact hheap_except k hk_ge hk_lt hk_eq_idx
    )
theorem Heapify_ArrayPerm_spec (arr : Array Int) (heapSize i : Nat) :
  isMaxHeapOn arr (i + 1) heapSize ∧ i < heapSize ∧ heapSize ≤ arr.size →
  frameOutsidePrefix arr (Heapify_ArrayPerm arr heapSize i).extract heapSize ∧
```

```
    arr.toList.Perm (Heapify_ArrayPerm arr heapSize i).extract.toList ∧
      isMaxHeapOn (Heapify_ArrayPerm arr heapSize i).extract i heapSize ∧
        (Heapify_ArrayPerm arr heapSize i).extract.size = arr.size :=
  VelvetM.extract_spec (Heapify_ArrayPerm arr heapSize i)
    _ (Heapify_ArrayPerm_correct arr heapSize i)
method BuildMaxHeap_ArrayPerm
  (arr : Array Int)
  (heapSize : Nat)
  return (result : Array Int)
require heapSize ≤ arr.size
ensures result.size = arr.size
ensures isMaxHeapPrefix result heapSize
ensures List.Perm arr.toList result.toList
ensures frameOutsidePrefix arr result heapSize
do
  let arr₀ := arr
  let mut a := arr
  -- Start from the last non-leaf node and go backwards
  -- Nodes at indices >= heapSize / 2 are leaves (no children in heap)
  let mut i := heapSize / 2
  -- Loop from heapSize/2 down to 0
  while i > 0
    invariant hsize : a.size = arr₀.size
    invariant hi_bound : i ≤ heapSize / 2
    -- After processing, nodes from i onwards satisfy heap property
    invariant hheap : isMaxHeapOn a i heapSize
    invariant hperm : List.Perm arr₀.toList a.toList
    invariant hframe : frameOutsidePrefix arr₀ a heapSize
    decreasing hdec : i
  do
    i := i - 1
    -- Now i is the index we want to heapify
    -- We need i < heapSize for Heapify_ArrayPerm
    -- Since i < heapSize / 2 (from loop condition), and heapSize > 0 when i was > 0
    -- we have i < heapSize
    if i < heapSize then
      a := (Heapify_ArrayPerm a heapSize i).extract
  return a
theorem frameOutsidePrefix_trans (arr a b : Array Int) (heapSize : Nat)
  (h1 : frameOutsidePrefix arr a heapSize) (h2 : frameOutsidePrefix a b heapSize) :
  frameOutsidePrefix arr b heapSize := by
  unfold frameOutsidePrefix at *
  constructor
  · omega
  · intro k hk1 hk2
    have ha_size : arr.size = a.size := h1.1
    have hb_size : a.size = b.size := h2.1
    have hk_a : k < a.size := by omega
    have h2_eq := h2.2 k hk_a hk2
    have h1_eq := h1.2 k hk1 hk2
    omega
set_option maxHeartbeats 10000000 in
prove_correct BuildMaxHeap_ArrayPerm by
  loom_solve
  case hsize.loop_pos => (
  have h_heap_pre : isMaxHeapOn a (a_snd - 1 + 1) heapSize := by
    cases a_snd with
    | zero => omega
    | succ n => simp_all
  have h_size_pre : heapSize ≤ a.size := by omega
  have h_spec := Heapify_ArrayPerm_spec a heapSize (a_snd - 1)
  have h_combined : isMaxHeapOn a (a_snd - 1 + 1) heapSize ∧ a_snd - 1 < heapSize ∧ heapSize ≤ a.size := by
    exact ⟨h_heap_pre, if_pos, h_size_pre⟩
  have h_result := h_spec h_combined
  omega
  )
  case hheap.loop_pos => (
  have h1 : a_snd - 1 + 1 = a_snd := Nat.sub_add_cancel (Nat.one_le_iff_ne_zero.mpr (Nat.pos_iff_ne_zero.mp
     if_pos_1))
  have h2 : isMaxHeapOn a (a_snd - 1 + 1) heapSize := by rw [h1]; exact hheap
  have h3 : heapSize ≤ a.size := by omega
  have spec := Heapify_ArrayPerm_spec a heapSize (a_snd - 1)
  have precond : isMaxHeapOn a (a_snd - 1 + 1) heapSize ∧ a_snd - 1 < heapSize ∧ heapSize ≤ a.size := ⟨h2,
     if_pos, h3⟩
  exact (spec precond).2.2.1
  )
  case hperm.loop_pos => (
  have hsz : heapSize ≤ a.size := by omega
  have heq : a_snd - 1 + 1 = a_snd := by omega
  have hheap' : isMaxHeapOn a (a_snd - 1 + 1) heapSize := by rw [heq]; exact hheap
  have hspec := Heapify_ArrayPerm_spec a heapSize (a_snd - 1) ⟨hheap', if_pos, hsz⟩
```

```
  have hperm_heapify : a.toList.Perm (Heapify_ArrayPerm a heapSize (a_snd - 1)).extract.toList := hspec.2.1
  exact hperm.trans hperm_heapify
  )
  case hframe.loop_pos => (
  have h_heapsize_le : heapSize ≤ a.size := by omega
  have h_idx : a_snd - 1 < heapSize := if_pos
  have h_heap_on : isMaxHeapOn a (a_snd - 1 + 1) heapSize := by
    simp only [Nat.sub_add_cancel (Nat.one_le_iff_ne_zero.mpr (Nat.pos_iff_ne_zero.mp if_pos_1))]
    exact hheap
  have spec := Heapify_ArrayPerm_spec a heapSize (a_snd - 1)
  have h_cond : isMaxHeapOn a (a_snd - 1 + 1) heapSize ∧ a_snd - 1 < heapSize ∧ heapSize ≤ a.size :=
    ⟨h_heap_on, h_idx, h_heapsize_le⟩
  have h_result := spec h_cond
  have h_frame_a_result : frameOutsidePrefix a (Heapify_ArrayPerm a heapSize (a_snd - 1)).extract heapSize :=
     h_result.1
  exact frameOutsidePrefix_trans arr a (Heapify_ArrayPerm a heapSize (a_snd - 1)).extract heapSize hframe
     h_frame_a_result
  )
  case hheap.entry => (
  unfold isMaxHeapOn
  intro k ⟨hlo, hhi⟩
  constructor
  · -- left child case
    intro hleft
    unfold left at hleft
    -- k ≥ heapSize / 2, so 2*k + 1 ≥ 2*(heapSize/2) + 1 ≥ heapSize
    -- This contradicts hleft : 2*k + 1 < heapSize
    omega
  · -- right child case
    intro hright
    unfold right at hright
    -- k ≥ heapSize / 2, so 2*k + 2 ≥ 2*(heapSize/2) + 2 > heapSize
    -- This contradicts hright : 2*k + 2 < heapSize
    omega
  )
  case hperm.entry => (
  try simp_all ; try grind
  )
  case hframe.entry => (
  unfold frameOutsidePrefix
  constructor
  · rfl
  · intros k _ _
    rfl
  )
theorem BuildMaxHeap_ArrayPerm_spec (arr : Array Int) (heapSize : Nat) :
 heapSize ≤ arr.size →
  frameOutsidePrefix arr (BuildMaxHeap_ArrayPerm arr heapSize).extract heapSize ∧
    arr.toList.Perm (BuildMaxHeap_ArrayPerm arr heapSize).extract.toList ∧
      isMaxHeapPrefix (BuildMaxHeap_ArrayPerm arr heapSize).extract heapSize ∧
        (BuildMaxHeap_ArrayPerm arr heapSize).extract.size = arr.size :=
  VelvetM.extract_spec (BuildMaxHeap_ArrayPerm arr heapSize)
    _ (BuildMaxHeap_ArrayPerm_correct arr heapSize)
method HeapSort_ArrayPerm
  (arr : Array Int)
  return (result : Array Int)
ensures result.size = arr.size
ensures List.Sorted (· ≤ ·) result.toList
ensures List.Perm arr.toList result.toList
do
  let n := arr.size
  if n ≤ 1 then
    return arr
  else
    -- Build max heap on the entire array
    let mut a := (BuildMaxHeap_ArrayPerm arr n).extract
    -- heapSize tracks the unsorted portion at the front
    let mut heapSize := n
    -- Repeatedly extract max and place at end
    while heapSize > 1
      invariant hsize : a.size = n
      invariant hheapSize : 1 ≤ heapSize ∧ heapSize ≤ n
      -- The heap property holds on [0, heapSize)
      invariant hheap : isMaxHeapPrefix a heapSize
      -- Elements from heapSize onwards are sorted
      invariant hsorted : ∀ i j, heapSize ≤ i ∧ i < j ∧ j < n → a[i]! ≤ a[j]!
      -- Every heap element is ≤ every sorted element (partition property)
      invariant hpartition : ∀ i j, i < heapSize ∧ heapSize ≤ j ∧ j < n → a[i]! ≤ a[j]!
      -- Permutation is preserved
      invariant hperm : List.Perm arr.toList a.toList
      decreasing hdec : heapSize
```

```
    do
      -- Swap the max element (at 0) with the last element of the heap
      let lastIdx := heapSize - 1
      let maxVal := a[0]!
      let lastVal := a[lastIdx]!
      a := Array.set! a 0 lastVal
      a := Array.set! a lastIdx maxVal
      -- Shrink the heap
      heapSize := heapSize - 1
      -- Restore heap property if heap is non-empty
      if heapSize > 0 then
        a := (Heapify_ArrayPerm a heapSize 0).extract
    return a
-- Helper lemma to convert getElem! to getElem when in bounds
theorem getElem!_eq_getElem {arr : Array Int} {i : Nat} (h : i < arr.size) :
    arr[i]! = arr[i] := by
  simp only [getElem!_def, getElem?_def, h, ↓reduceDIte]
-- Helper: sorted array from pointwise condition
theorem array_sorted_of_forall_lt {arr : Array Int} (n : Nat)
    (hsize : arr.size = n)
    (hpw : ∀ i j, i < j → j < n → arr[i]! ≤ arr[j]!) :
    List.Sorted (· ≤ ·) arr.toList := by
  rw [List.Sorted]
  rw [List.pairwise_iff_get]
  intro i j hij
  simp only [Array.length_toList] at i j hij
  simp only [List.get_eq_getElem, Array.getElem_toList]
  have hi : i.val < n := by rw [← hsize]; exact i.isLt
  have hj : j.val < n := by rw [← hsize]; exact j.isLt
  have := hpw i.val j.val hij hj
  rw [getElem!_eq_getElem (by omega : i.val < arr.size)] at this
  rw [getElem!_eq_getElem (by omega : j.val < arr.size)] at this
  exact this
-- Helper lemma: if first element ≤ all others and rest is sorted, then whole list is sorted
theorem sorted_from_partition_and_tail {arr : Array Int} (n : Nat)
    (hsize : arr.size = n) (hn : n > 1)
    (hpartition : ∀ j, 1 ≤ j → j < n → arr[0]! ≤ arr[j]!)
    (hsorted : ∀ i j, 1 ≤ i → i < j → j < n → arr[i]! ≤ arr[j]!) :
    List.Sorted (· ≤ ·) arr.toList := by
  apply array_sorted_of_forall_lt n hsize
  intro i j hij hjn
  by_cases h0 : i = 0
  · subst h0
    have hj1 : 1 ≤ j := by omega
    exact hpartition j hj1 hjn
  · have hi1 : 1 ≤ i := by omega
    exact hsorted i j hi1 hij hjn
-- Helper: In a max heap, every element is ≤ the root
-- This requires induction on the path from element i to the root
theorem maxHeap_root_is_max {arr : Array Int} {n : Nat} (hheap : isMaxHeapPrefix arr n)
    (hn : n > 0) (hsz : n ≤ arr.size) (i : Nat) (hi : i < n) : arr[i]! ≤ arr[0]! := by
  -- We prove by strong induction: every element ≤ its parent, so transitively ≤ root
  induction i using Nat.strong_induction_on with
  | _ i ih =>
    by_cases h0 : i = 0
    · simp [h0]
    · -- i > 0, so i has a parent
      have hi_pos : i > 0 := Nat.pos_of_ne_zero h0
      let p := parent i
      have hp_def : p = (i - 1) / 2 := rfl
      have hp_lt_i : p < i := by
        simp only [p, parent]
        omega
      have hp_lt_n : p < n := by omega
      -- From heap property: arr[p]! ≥ arr[left p]! and arr[p]! ≥ arr[right p]!
      have hparent_heap := hheap p ⟨Nat.zero_le p, hp_lt_n⟩
      -- i is either left child or right child of p
      have h_child : i = left p ∨ i = right p := by
        simp only [left, right, p, parent]
        have : i = 2 * ((i - 1) / 2) + 1 ∨ i = 2 * ((i - 1) / 2) + 2 := by
          have := Nat.div_add_mod (i - 1) 2
          by_cases hmod : (i - 1) % 2 = 0
          · left
            have : i - 1 = 2 * ((i - 1) / 2) := by omega
            omega
          · right
            have hmod1 : (i - 1) % 2 = 1 := by omega
            have : i - 1 = 2 * ((i - 1) / 2) + 1 := by omega
            omega
        exact this
      -- In either case, arr[p]! ≥ arr[i]!
```

```
      have hpi : arr[p]! ≥ arr[i]! := by
        cases h_child with
        | inl hl =>
          have := hparent_heap.1
          rw [← hl] at this
          exact this hi
        | inr hr =>
          have := hparent_heap.2
          rw [← hr] at this
          exact this hi
      -- By IH, arr[p]! ≤ arr[0]!
      have hp0 := ih p hp_lt_i hp_lt_n
      omega
set_option maxHeartbeats 400000 in
theorem swap_preserves_heap_on_inner (a : Array Int) (heapSize : Nat)
    (hheap : isMaxHeapPrefix a heapSize) (hgt : heapSize > 1)
    (hsize : heapSize ≤ a.size) :
    let swapped := (a.set! 0 a[heapSize - 1]!).set! (heapSize - 1) a[0]!
    isMaxHeapOn swapped 1 (heapSize - 1) := by
  intro swapped
  unfold isMaxHeapOn
  intro k ⟨hk1, hklt⟩
  have hklt' : k < heapSize := by omega
  have hk_ne_0 : k ≠ 0 := by omega
  have hk_ne_last : k ≠ heapSize - 1 := by omega
  have hk_bounds : k < a.size := by omega
  have h0_bounds : 0 < a.size := by omega
  have hlast_bounds : heapSize - 1 < a.size := by omega
  -- Key: for k in [1, heapSize-1), swapped[k] = a[k]
  have hswapped_k : swapped[k]! = a[k]! := by
    simp only [swapped, Array.set!, Array.setIfInBounds, h0_bounds, hlast_bounds, ↓reduceDIte,
               Array.size_set]
    simp only [getElem!_def, getElem?_def, hk_bounds, ↓reduceDIte, Array.size_set]
    simp only [Array.getElem_set]
    split_ifs with h1 h2
    · omega
    · omega
    · rfl
  constructor
  · intro hleft
    have hleft' : left k < heapSize := by omega
    have hleft_ne_0 : left k ≠ 0 := by unfold left; omega
    have hleft_ne_last : left k ≠ heapSize - 1 := by omega
    have hleft_bounds : left k < a.size := by omega
    have hswapped_left : swapped[left k]! = a[left k]! := by
      simp only [swapped, Array.set!, Array.setIfInBounds, h0_bounds, hlast_bounds, ↓reduceDIte,
                 Array.size_set]
      simp only [getElem!_def, getElem?_def, hleft_bounds, ↓reduceDIte, Array.size_set]
      simp only [Array.getElem_set]
      split_ifs with h1 h2
      · omega
      · omega
      · rfl
    rw [hswapped_k, hswapped_left]
    have := hheap k ⟨by omega, hklt'⟩
    exact this.1 hleft'
  · intro hright
    have hright' : right k < heapSize := by omega
    have hright_ne_0 : right k ≠ 0 := by unfold right; omega
    have hright_ne_last : right k ≠ heapSize - 1 := by omega
    have hright_bounds : right k < a.size := by omega
    have hswapped_right : swapped[right k]! = a[right k]! := by
      simp only [swapped, Array.set!, Array.setIfInBounds, h0_bounds, hlast_bounds, ↓reduceDIte,
                 Array.size_set]
      simp only [getElem!_def, getElem?_def, hright_bounds, ↓reduceDIte, Array.size_set]
      simp only [Array.getElem_set]
      split_ifs with h1 h2
      · omega
      · omega
      · rfl
    rw [hswapped_k, hswapped_right]
    have := hheap k ⟨by omega, hklt'⟩
    exact this.2 hright'
-- Helper lemma for swap access
theorem swap_getElem!_eq {a : Array Int} {idx0 idx1 k : Nat}
    (h0 : idx0 < a.size) (h1 : idx1 < a.size) (hk : k < a.size)
    (hk_ne_0 : k ≠ idx0) (hk_ne_1 : k ≠ idx1) :
    ((a.set! idx0 (a[idx1]!)).set! idx1 (a[idx0]!))[k]! = a[k]! := by
  simp only [Array.set!, Array.setIfInBounds, h0, h1, hk, ↓reduceDIte, Array.size_set]
  simp only [getElem!_def, getElem?_def, Array.size_set, hk, ↓reduceDIte]
  simp only [Array.get_set]
```

```
  split_ifs <;> simp_all
theorem swap_getElem!_at_idx1 {a : Array Int} {idx0 idx1 : Nat}
    (h0 : idx0 < a.size) (h1 : idx1 < a.size) :
    ((a.set! idx0 (a[idx1]!)).set! idx1 (a[idx0]!))[idx1]! = a[idx0]! := by
  simp only [Array.set!, Array.setIfInBounds, h0, h1, ↓reduceDIte, Array.size_set]
  simp only [getElem!_def, getElem?_def, Array.size_set, h1, h0, ↓reduceDIte]
  simp only [Array.get_set]
  simp
theorem getElem!_set!_ne {arr : Array α} [Inhabited α] {i j : Nat} (v : α) (hne : i ≠ j) :
    (arr.set! j v)[i]! = arr[i]! := by
  simp only [Array.set!, Array.setIfInBounds]
  split_ifs with hj
  · simp only [getElem!_def, getElem?_def, Array.size_set]
    split_ifs with hi
    · simp only [Array.getElem_set]
      split_ifs with heq
      · exact (hne heq.symm).elim
      · rfl
    · rfl
  · rfl
theorem swap_preserves_perm {arr : Array Int} {i j : Nat} (hi : i < arr.size) (hj : j < arr.size) :
    arr.toList.Perm ((arr.set! i arr[j]!).set! j arr[i]!).toList := by
  have h1 : arr.toMultiset = ((arr.set! i arr[j]!).set! j arr[i]!).toMultiset := by
    have := Array.multiset_swap arr i j hi hj
    simp only [Array.toMultiset] at this ⊢
    convert this.symm using 2
    apply List.ext_getElem
    · simp [Array.size_set]
    · intro n hn1 hn2
      simp only [Array.getElem_toList, Array.size_set] at hn1 hn2 ⊢
      simp only [Array.set!, Array.setIfInBounds, hi, hj, ↓reduceDIte, Array.size_set, Array.getElem_set]
      by_cases hnj : n = j
      · subst hnj
        by_cases hni : n = i
        · subst hni
          simp [getElem!_pos, *]
        · simp only [hni, ↓reduceIte, Ne.symm hni, getElem!_pos, hi]
      · by_cases hni : n = i
        · subst hni
          have hne : j ≠ n := Ne.symm hnj
          simp only [hnj, ↓reduceIte, hne, getElem!_pos, hj]
        · have hne1 : i ≠ n := Ne.symm hni
          have hne2 : j ≠ n := Ne.symm hnj
          simp only [hni, hnj, hne1, hne2, ↓reduceIte]
  simp only [Array.toMultiset] at h1
  rw [← Multiset.coe_eq_coe]
  exact h1
-- Helper: elements in a permutation prefix are bounded by the max of the original prefix
theorem perm_prefix_bounded {arr1 arr2 : Array Int} {prefixSize : Nat}
    (hperm : arr1.toList.Perm arr2.toList)
    (hframe : ∀ k, k < arr1.size → prefixSize ≤ k → arr2[k]! = arr1[k]!)
    (hsize_eq : arr1.size = arr2.size)
    (bound : Int)
    (hbound : ∀ k, k < prefixSize → arr1[k]! ≤ bound)
    (i : Nat) (hi : i < prefixSize) (hi_sz : i < arr2.size) :
    arr2[i]! ≤ bound := by
  -- We use the fact that arr2[i] appears in arr2.toList
  have hi_mem : arr2[i]! ∈ arr2.toList := by
    rw [getElem!_eq_getElem hi_sz]
    exact Array.getElem_mem_toList hi_sz
  -- By permutation, arr2[i] also appears in arr1.toList
  have hi_mem' : arr2[i]! ∈ arr1.toList := hperm.symm.mem_iff.mp hi_mem
  -- So arr2[i] = arr1[k] for some k
  obtain ⟨k, hk_lt, hk_eq⟩ := List.getElem_of_mem hi_mem'
  simp only [Array.length_toList] at hk_lt
  rw [Array.getElem_toList] at hk_eq
  -- Now we show k < prefixSize
  by_cases hk_ge : prefixSize ≤ k
  · -- If k ≥ prefixSize, use counting argument
    have hsuffix_eq : ∀ m, prefixSize ≤ m → m < arr1.size → arr2[m]! = arr1[m]! := by
      intro m hm_ge hm_lt
      exact hframe m hm_lt hm_ge
    have hcount_total : arr1.toList.count arr2[i]! = arr2.toList.count arr2[i]! :=
      hperm.count_eq arr2[i]!
    have hcount_suffix : (arr1.toList.drop prefixSize).count arr2[i]! =
                         (arr2.toList.drop prefixSize).count arr2[i]! := by
      congr 1
      apply List.ext_getElem
      · simp only [List.length_drop, Array.length_toList, hsize_eq]
      · intro n hn1 hn2
        simp only [List.length_drop, Array.length_toList] at hn1 hn2
```

```
        simp only [List.getElem_drop, Array.getElem_toList]
        have hpn : prefixSize + n < arr1.size := by omega
        rw [← getElem!_eq_getElem hpn]
        rw [← getElem!_eq_getElem (by omega : prefixSize + n < arr2.size)]
        exact (hframe (prefixSize + n) hpn (by omega)).symm
    have hcount_prefix : (arr1.toList.take prefixSize).count arr2[i]! =
                         (arr2.toList.take prefixSize).count arr2[i]! := by
      have h1 : arr1.toList.count arr2[i]! = (arr1.toList.take prefixSize).count arr2[i]! +
                (arr1.toList.drop prefixSize).count arr2[i]! := by
        rw [← List.count_append]
        rw [List.take_append_drop]
      have h2 : arr2.toList.count arr2[i]! = (arr2.toList.take prefixSize).count arr2[i]! +
                (arr2.toList.drop prefixSize).count arr2[i]! := by
        rw [← List.count_append]
        rw [List.take_append_drop]
      omega
    -- arr2[i] appears at least once in arr2.toList.take prefixSize
    have hi_count_pos : 0 < (arr2.toList.take prefixSize).count arr2[i]! := by
      rw [List.count_pos_iff]
      rw [getElem!_eq_getElem hi_sz]
      have h_take_len : i < (arr2.toList.take prefixSize).length := by
        simp only [List.length_take, Array.length_toList]
        omega
      have heq : (arr2.toList.take prefixSize)[i]'h_take_len = arr2[i] := by
        simp only [List.getElem_take, Array.getElem_toList]
      rw [← heq]
      exact List.getElem_mem h_take_len
    -- So arr2[i] appears at least once in arr1.toList.take prefixSize
    have hi_count_pos' : 0 < (arr1.toList.take prefixSize).count arr2[i]! := by
      omega
    -- So arr2[i] ∈ arr1.toList.take prefixSize
    have hi_in_prefix : arr2[i]! ∈ arr1.toList.take prefixSize :=
      List.count_pos_iff.mp hi_count_pos'
    -- So arr2[i] = arr1[m] for some m < prefixSize
    obtain ⟨m, hm_lt, hm_eq⟩ := List.getElem_of_mem hi_in_prefix
    simp only [List.length_take, Array.length_toList, Nat.min_def] at hm_lt
    split_ifs at hm_lt with hcmp
    · simp only [List.getElem_take, Array.getElem_toList] at hm_eq
      rw [← hm_eq, ← getElem!_eq_getElem (by omega : m < arr1.size)]
      exact hbound m hm_lt
    · omega
  · -- k < prefixSize, direct
    push_neg at hk_ge
    rw [← hk_eq, ← getElem!_eq_getElem (by omega : k < arr1.size)]
    exact hbound k hk_ge
set_option maxHeartbeats 10000000 in
prove_correct HeapSort_ArrayPerm by
  loom_solve
  case ensures_2 => (
  try simp_all ; try grind
  )
  case ensures_1 => (
  cases h : arr.size with
  | zero =>
    have : arr = #[] := Array.size_eq_zero.mp h
    simp [this]
  | succ n =>
    have hn : n = 0 := by omega
    subst hn
    have hsize : arr.size = 1 := h
    have hlist : arr.toList.length = 1 := by simp [hsize]
    obtain ⟨x, hx⟩ := List.length_eq_one.mp hlist
    rw [hx]
    exact List.sorted_singleton x
  )
  case hsize.loop_pos => (
  -- Define the swapped array
  set swapped := (a.set! 0 a[a_snd - 1]!).set! (a_snd - 1) a[0]! with hswapped_def
  -- The swapped array has the same size as a
  have hswapped_size : swapped.size = a.size := by
    simp only [hswapped_def, Array.size_set!]
  -- We need to verify preconditions for Heapify_ArrayPerm_spec
  have h_bounds : a_snd - 1 ≤ swapped.size := by
    rw [hswapped_size, hsize]; omega
  have h_zero_lt : 0 < a_snd - 1 := if_pos
  have spec := Heapify_ArrayPerm_spec swapped (a_snd - 1) 0
  -- The spec says result.size = input.size when preconditions hold
  suffices hsuff : (Heapify_ArrayPerm swapped (a_snd - 1) 0).extract.size = swapped.size by
    rw [hsuff, hswapped_size, hsize]
  -- We need isMaxHeapOn swapped 1 (a_snd - 1)
  -- Helper lemma for accessing elements in swapped array at indices other than 0 and a_snd-1
```

```
have ha_size_pos : 0 < a.size := by omega
have ha_snd_lt : a_snd - 1 < a.size := by omega
have hget_swapped : ∀ i, i ≠ 0 → i ≠ a_snd - 1 → swapped[i]! = a[i]! := by
  intro i hi0 hilast
  simp only [hswapped_def]
  have h0_lt : 0 < a.size := ha_size_pos
  have hlast_lt : a_snd - 1 < a.size := ha_snd_lt
  unfold Array.set! Array.setIfInBounds
  simp only [h0_lt, hlast_lt, ↓reduceDIte, Array.size_set]
  -- Now we have (a.set 0 ... |>.set (a_snd-1) ...)[i]!
  -- Use getElem! definition and split on indices
  conv_lhs => simp only [getElem!_def, Array.getElem?_set, Array.size_set]
  split_ifs with h1 h2
  · exfalso; exact hilast h1.symm
  · exfalso; exact hi0 h2.symm
  · simp only [getElem!_def]
have h_heap_on : isMaxHeapOn swapped 1 (a_snd - 1) := by
  intro k ⟨hk_lo, hk_hi⟩
  have hk_ne_0 : k ≠ 0 := by omega
  have hk_ne_last : k ≠ a_snd - 1 := by omega
  constructor
  · intro h_left
    have hl_ne_0 : left k ≠ 0 := by unfold left; omega
    have hl_ne_last : left k ≠ a_snd - 1 := by omega
    rw [hget_swapped k hk_ne_0 hk_ne_last, hget_swapped (left k) hl_ne_0 hl_ne_last]
    have h_orig := hheap k ⟨by omega, by omega⟩
    exact h_orig.1 (by omega)
  · intro h_right
    have hr_ne_0 : right k ≠ 0 := by unfold right; omega
    have hr_ne_last : right k ≠ a_snd - 1 := by omega
    rw [hget_swapped k hk_ne_0 hk_ne_last, hget_swapped (right k) hr_ne_0 hr_ne_last]
    have h_orig := hheap k ⟨by omega, by omega⟩
    exact h_orig.2 (by omega)
have h_all_pre : isMaxHeapOn swapped (0 + 1) (a_snd - 1) ∧ 0 < a_snd - 1 ∧ a_snd - 1 ≤ swapped.size := by
  exact ⟨h_heap_on, h_zero_lt, h_bounds⟩
have h_spec := spec h_all_pre
exact h_spec.2.2.2
)
case hheap.loop_pos => (
let swapped := (a.set! 0 a[a_snd - 1]!).set! (a_snd - 1) a[0]!
have h0_bounds : 0 < a.size := by omega
have hlast_bounds : a_snd - 1 < a.size := by omega
have hswap_size : swapped.size = a.size := by
  simp only [swapped, Array.set!, Array.setIfInBounds, h0_bounds, hlast_bounds, ↓reduceDIte,
             Array.size_set]
have hnewHeapSize : a_snd - 1 ≤ swapped.size := by omega
have hi_lt : (0 : Nat) < a_snd - 1 := by omega
have hsize_bound : a_snd ≤ a.size := by omega
have hheap_on_1 : isMaxHeapOn swapped 1 (a_snd - 1) :=
  swap_preserves_heap_on_inner a a_snd hheap if_pos_1 hsize_bound
have hheap_on : isMaxHeapOn swapped (0 + 1) (a_snd - 1) := hheap_on_1
have spec := Heapify_ArrayPerm_spec swapped (a_snd - 1) 0
have hcond : isMaxHeapOn swapped (0 + 1) (a_snd - 1) ∧ 0 < a_snd - 1 ∧ a_snd - 1 ≤ swapped.size :=
  ⟨hheap_on, hi_lt, hnewHeapSize⟩
have result := spec hcond
exact result.2.2.1
)
case hsorted.loop_pos => (
-- Let swapped be the array after swapping
set swapped := (a.set! 0 a[a_snd - 1]!).set! (a_snd - 1) a[0]! with hswapped_def
set result := (Heapify_ArrayPerm swapped (a_snd - 1) 0).extract with hresult_def
-- Get bounds
have ha_sz : a.size = arr.size := hsize
have h0_lt : 0 < a.size := by omega
have hlast_lt : a_snd - 1 < a.size := by omega
have hswapped_sz : swapped.size = a.size := by simp [swapped, Array.size_set]
-- Heapify preconditions
have hheap_pre1 : a_snd - 1 ≤ swapped.size := by omega
have hheap_pre2 : 0 < a_snd - 1 := if_pos
have hheap_pre3 : isMaxHeapOn swapped 1 (a_snd - 1) :=
  swap_preserves_heap_on_inner a a_snd hheap if_pos_1 (by omega)
-- Get Heapify spec
have hspec := Heapify_ArrayPerm_spec swapped (a_snd - 1) 0 ⟨hheap_pre3, hheap_pre2, hheap_pre1⟩
have hframe := hspec.1
have hresult_sz : result.size = swapped.size := hspec.2.2.2
-- frameOutsidePrefix tells us elements at positions ≥ a_snd - 1 are unchanged
have hframe_eq : ∀ k, k < swapped.size → a_snd - 1 ≤ k → result[k]! = swapped[k]! := by
  intro k hk_lt hk_ge
  have := hframe.2 k hk_lt hk_ge
  exact this
-- i and j are both ≥ a_snd - 1, so they're outside the heap region
```

```
have hi_ge : a_snd - 1 ≤ i := h_3
have hj_ge : a_snd - 1 ≤ j := by omega
have hi_lt_arr : i < arr.size := by omega
have hj_lt_arr : j < arr.size := h
have hi_lt_swapped : i < swapped.size := by omega
have hj_lt_swapped : j < swapped.size := by omega
-- result[i]! = swapped[i]! and result[j]! = swapped[j]!
have hresult_i : result[i]! = swapped[i]! := hframe_eq i hi_lt_swapped hi_ge
have hresult_j : result[j]! = swapped[j]! := hframe_eq j hj_lt_swapped hj_ge
rw [hresult_i, hresult_j]
-- Now analyze swapped[i]! and swapped[j]!
-- swapped[a_snd - 1] = a[0]! and for k > a_snd - 1, swapped[k] = a[k]
by_cases hi_eq : i = a_snd - 1
· -- i = a_snd - 1, so swapped[i]! = a[0]!
  have hswapped_i : swapped[i]! = a[0]! := by
    rw [hi_eq]
    exact swap_getElem!_at_idx1 h0_lt hlast_lt
  rw [hswapped_i]
  -- j > i = a_snd - 1, so j ≥ a_snd
  have hj_ge_asnd : a_snd ≤ j := by omega
  have hj_ne_0 : j ≠ 0 := by omega
  have hj_ne_last : j ≠ a_snd - 1 := by omega
  have hswapped_j : swapped[j]! = a[j]! :=
    swap_getElem!_eq h0_lt hlast_lt (by omega : j < a.size) hj_ne_0 hj_ne_last
  rw [hswapped_j]
  -- Use hpartition: a[0]! ≤ a[j]! since 0 < a_snd and a_snd ≤ j
  have h0_lt_asnd : 0 < a_snd := by omega
  exact hpartition 0 j h0_lt_asnd hj_ge_asnd hj_lt_arr
· -- i > a_snd - 1, so i ≥ a_snd
  have hi_ge_asnd : a_snd ≤ i := by omega
  have hi_ne_0 : i ≠ 0 := by omega
  have hi_ne_last : i ≠ a_snd - 1 := hi_eq
  have hswapped_i : swapped[i]! = a[i]! :=
    swap_getElem!_eq h0_lt hlast_lt (by omega : i < a.size) hi_ne_0 hi_ne_last
  rw [hswapped_i]
  -- j > i ≥ a_snd, so j > a_snd - 1
  have hj_ge_asnd : a_snd ≤ j := by omega
  have hj_ne_0 : j ≠ 0 := by omega
  have hj_ne_last : j ≠ a_snd - 1 := by omega
  have hswapped_j : swapped[j]! = a[j]! :=
    swap_getElem!_eq h0_lt hlast_lt (by omega : j < a.size) hj_ne_0 hj_ne_last
  rw [hswapped_j]
  -- Use hsorted: a_snd ≤ i < j < arr.size
  exact hsorted i j hi_ge_asnd h_4 hj_lt_arr
)
case hpartition.loop_pos => (
let swapped := (a.set! (0 : ℕ) a[a_snd - (1 : ℕ)]!).set! (a_snd - (1 : ℕ)) a[(0 : ℕ)]!
let result := (Heapify_ArrayPerm swapped (a_snd - 1) 0).extract
have h0_lt : 0 < a.size := by omega
have hlast_lt : a_snd - 1 < a.size := by omega
have hswapped_size : swapped.size = a.size := by
  simp only [swapped, Array.set!, Array.setIfInBounds, h0_lt, hlast_lt, ↓reduceDIte, Array.size_set]
have hheap_inner : isMaxHeapOn swapped 1 (a_snd - 1) :=
  swap_preserves_heap_on_inner a a_snd hheap if_pos_1 (by omega)
have hspec := Heapify_ArrayPerm_spec swapped (a_snd - 1) 0 ⟨hheap_inner, if_pos, by omega⟩
have hresult_size : result.size = swapped.size := hspec.2.2.2
have hresult_perm : swapped.toList.Perm result.toList := hspec.2.1
have hframe : frameOutsidePrefix swapped result (a_snd - 1) := hspec.1
have hresult_j : result[j]! = swapped[j]! := by
  have hj_lt_swapped : j < swapped.size := by omega
  exact hframe.2 j hj_lt_swapped h_4
have hmax_heap : ∀ k, k < a_snd → a[k]! ≤ a[0]! := fun k hk =>
  maxHeap_root_is_max hheap (by omega) (by omega) k hk
have hswapped_bounded : ∀ k, k < a_snd - 1 → swapped[k]! ≤ a[0]! := by
  intro k hk
  by_cases hk0 : k = 0
  · subst hk0
    have : swapped[0]! = a[a_snd - 1]! := by
      simp only [swapped, Array.set!, Array.setIfInBounds, h0_lt, hlast_lt, ↓reduceDIte, Array.size_set]
      simp only [getElem!_def, getElem?_def, h0_lt, ↓reduceDIte, Array.size_set, Array.getElem_set]
      split_ifs <;> simp_all only [getElem!_def, getElem?_def, hlast_lt, ↓reduceDIte]
    rw [this]
    exact hmax_heap (a_snd - 1) (by omega)
  · have hk_ne_last : k ≠ a_snd - 1 := by omega
    have : swapped[k]! = a[k]! := swap_getElem!_eq h0_lt hlast_lt (by omega) (by omega) hk_ne_last
    rw [this]
    exact hmax_heap k (by omega)
have hframe_fn : ∀ k, k < swapped.size → a_snd - 1 ≤ k → result[k]! = swapped[k]! := hframe.2
have hi_lt_result : i < result.size := by omega
have hresult_i_bounded : result[i]! ≤ a[0]! := by
  apply perm_prefix_bounded hresult_perm hframe_fn (by omega) a[0]! hswapped_bounded i h_3 hi_lt_result
```

```
by_cases hj_eq : j = a_snd - 1
· subst hj_eq
  have : swapped[a_snd - 1]! = a[0]! := swap_getElem!_at_idx1 h0_lt hlast_lt
  rw [hresult_j, this]
  exact hresult_i_bounded
· have hj_gt : j > a_snd - 1 := by omega
  have hswapped_j : swapped[j]! = a[j]! :=
    swap_getElem!_eq h0_lt hlast_lt (by omega) (by omega) (by omega)
  rw [hresult_j, hswapped_j]
  have hold_part : a[0]! ≤ a[j]! := hpartition 0 j (by omega) (by omega) h
  calc result[i]! ≤ a[0]! := hresult_i_bounded
    _ ≤ a[j]! := hold_part
)
case hperm.loop_pos => (
have h0_lt : 0 < a.size := by omega
have hlast_lt : a_snd - 1 < a.size := by omega
have swap_perm : a.toList.Perm ((a.set! 0 a[a_snd - 1]!).set! (a_snd - 1) a[0]!).toList := by
  exact swap_preserves_perm h0_lt hlast_lt
let a' := (a.set! 0 a[a_snd - 1]!).set! (a_snd - 1) a[0]!
have heapify_perm : a'.toList.Perm (Heapify_ArrayPerm a' (a_snd - 1) 0).extract.toList := by
  have hspec := Heapify_ArrayPerm_spec a' (a_snd - 1) 0
  have hsize' : a'.size = a.size := by simp [a', Array.size_set]
  have hheapSize : a_snd - 1 ≤ a'.size := by simp [hsize']; omega
  have hi : 0 < a_snd - 1 := if_pos
  have hheapOn : isMaxHeapOn a' (0 + 1) (a_snd - 1) := by
    unfold isMaxHeapOn
    intro k ⟨hk1, hk2⟩
    constructor
    · intro hleft
      have hkne0 : k ≠ 0 := by omega
      have hkne_last : k ≠ a_snd - 1 := by omega
      have hleft_ne0 : left k ≠ 0 := by unfold left; omega
      have hleft_ne_last : left k ≠ a_snd - 1 := by omega
      rw [show a'[k]! = a[k]! by
        simp only [a']
        rw [getElem!_set!_ne _ hkne_last, getElem!_set!_ne _ hkne0]]
      rw [show a'[left k]! = a[left k]! by
        simp only [a']
        rw [getElem!_set!_ne _ hleft_ne_last, getElem!_set!_ne _ hleft_ne0]]
      have horig := hheap
      unfold isMaxHeapPrefix isMaxHeapOn at horig
      have := horig k ⟨by omega, by omega⟩
      exact this.1 (by omega : left k < a_snd)
    · intro hright
      have hkne0 : k ≠ 0 := by omega
      have hkne_last : k ≠ a_snd - 1 := by omega
      have hright_ne0 : right k ≠ 0 := by unfold right; omega
      have hright_ne_last : right k ≠ a_snd - 1 := by omega
      rw [show a'[k]! = a[k]! by
        simp only [a']
        rw [getElem!_set!_ne _ hkne_last, getElem!_set!_ne _ hkne0]]
      rw [show a'[right k]! = a[right k]! by
        simp only [a']
        rw [getElem!_set!_ne _ hright_ne_last, getElem!_set!_ne _ hright_ne0]]
      have horig := hheap
      unfold isMaxHeapPrefix isMaxHeapOn at horig
      have := horig k ⟨by omega, by omega⟩
      exact this.2 (by omega : right k < a_snd)
  have hcond : isMaxHeapOn a' (0 + 1) (a_snd - 1) ∧ 0 < a_snd - 1 ∧ a_snd - 1 ≤ a'.size := ⟨hheapOn, hi,
   hheapSize⟩
  exact (hspec hcond).2.1
exact hperm.trans (swap_perm.trans heapify_perm)
)
case hsize.entry_neg => (
have h : arr.size ≤ arr.size := le_refl _
have spec := BuildMaxHeap_ArrayPerm_spec arr arr.size h
exact spec.2.2.2
)
case hheap.entry_neg => (
have h : arr.size ≤ arr.size := le_refl _
have spec := BuildMaxHeap_ArrayPerm_spec arr arr.size h
exact spec.2.2.1
)
case hperm.entry_neg => (
have h : arr.size ≤ arr.size := le_refl _
have spec := BuildMaxHeap_ArrayPerm_spec arr arr.size h
exact spec.2.1
)
case ensures_1_1 => (
obtain ⟨rfl, rfl⟩ := h
have hhs : a_snd = 1 := by omega
```

```
  subst hhs
  have hn : arr.size > 1 := by omega
  apply sorted_from_partition_and_tail arr.size hsize hn
  · intro j hj1 hjn
    -- We need to show a[0]! ≤ a[j]! for 1 ≤ j < arr.size
    -- hpartition gives us: for i < 1 and 1 ≤ j < arr.size, a[i]! ≤ a[j]!
    -- With i = 0, j ≥ 1, this is exactly what we need
    exact hpartition 0 j (by omega) hj1 hjn
  · intro i' j hi1 hij hjn
    exact hsorted i' j (by omega) hij hjn
  )
```

H.2. Prime Factorization

WybeCoder using Clause Opus 4.5 successfully verified a prime factorization algorithm that decomposes a natural number $n \geq 2$ into its prime factors with multiplicity. This is problem `clever_id_24` from the Clever dataset. The algorithm maintains a trial divisor `factor` (starting at 2) and repeatedly divides out factors from `remaining` until the quotient becomes 1. The termination proof of the `while` loop uses the non-standard decreasing measure $2 \cdot \texttt{remaining} - \texttt{factor}$. In the division branch, `remaining` strictly decreases; in the increment branch, `factor` increases while `remaining` stays fixed. Proving the measure decreases in the latter case requires establishing that $\texttt{factor} \leq \texttt{remaining}$, a non-trivial fact following from the invariant that some prime must divide the remaining quotient. The complete verified implementation spans approximately 200 lines of Lean 4 code, including five helper lemmas for sorted list extension, product preservation under division, and primality/divisibility propagation. This case study demonstrates WybeCoder's ability to handle algorithms requiring non-trivial termination arguments and interdependent loop invariants.

Listing 5. Implementation and correctness proof of a Prime Factorization (`clever_id_24`)

```
import Auto
import Aesop
import Lean
import Mathlib

import CaseStudies.Velvet.Std
import CaseStudies.TestingUtil

set_option loom.semantics.termination "total"
set_option loom.semantics.choice "demonic"
set_option loom.solver "cvc5"
set_option auto.smt.timeout 3
set_option maxHeartbeats 100000
set_option auto.smt.trust true


method factorize (n: ℕ) return (res: List ℕ)
  require 2 ≤ n
  ensures res.prod = n
  ensures List.Sorted (· ≤ ·) res
  ensures ∀ i ∈ res, n % i = 0 ∧ Nat.Prime i
  do
    let n₀ := n
    let mut remaining := n
    let mut factor := 2
    let mut factors : List ℕ := []

    while remaining > 1
      invariant remaining ≥ 1
      invariant factor ≥ 2
      invariant factors.prod * remaining = n₀
      invariant ∀ p ∈ factors, Nat.Prime p ∧ n₀ % p = 0
      invariant List.Sorted (· ≤ ·) factors
      invariant ∀ p ∈ factors, p ≤ factor
      invariant ∀ p, Nat.Prime p → p < factor → remaining % p ≠ 0
      decreasing remaining + remaining - factor
    do
      if remaining % factor = 0 then
        factors := factors ++ [factor]
        remaining := remaining / factor
      else
        factor := factor + 1

    return factors

-- Helper lemma for sorted append
theorem sorted_append_singleton {l : List ℕ} {x : ℕ}
```

```
    (hs : List.Sorted (· ≤ ·) l) (hx : ∀ y ∈ l, y ≤ x) :
    List.Sorted (· ≤ ·) (l ++ [x]) := by
  induction l with
  | nil => simp [List.Sorted]
  | cons head tail ih =>
    simp only [List.cons_append]
    rw [List.sorted_cons] at hs ⊢
    constructor
    · intro b hb
      simp only [List.mem_append, List.mem_singleton] at hb
      cases hb with
      | inl ht => exact hs.1 b ht
      | inr hx' =>
        rw [hx']
        have hmem : head ∈ head :: tail := by simp
        exact hx head hmem
    · apply ih hs.2
      intro y hy
      have hmem : y ∈ head :: tail := List.mem_cons_of_mem head hy
      exact hx y hmem

-- Helper: factor is prime when it divides remaining and no smaller prime does
theorem factor_is_prime (remaining factor : ℕ)
    (hrem : remaining > 1) (hfac : factor ≥ 2)
    (hdiv : factor ∣ remaining)
    (hno_smaller : ∀ p, Nat.Prime p → p < factor → ¬(p ∣ remaining)) :
    Nat.Prime factor := by
  have hne1 : remaining ≠ 1 := by omega
  have hmin_le : remaining.minFac ≤ factor := Nat.minFac_le_of_dvd hfac hdiv
  have hmin_prime := Nat.minFac_prime hne1
  have hmin_dvd := Nat.minFac_dvd remaining
  by_cases heq : remaining.minFac = factor
  · rw [← heq]; exact hmin_prime
  · have hlt : remaining.minFac < factor := Nat.lt_of_le_of_ne hmin_le heq
    exfalso
    exact hno_smaller remaining.minFac hmin_prime hlt hmin_dvd

-- Helper for product preservation
theorem prod_append_div (factors : List ℕ) (remaining factor n : ℕ)
    (hdiv : factor ∣ remaining) (hprod : factors.prod * remaining = n) :
    (factors ++ [factor]).prod * (remaining / factor) = n := by
  simp only [List.prod_append, List.prod_singleton]
  have heq : remaining / factor * factor = remaining := Nat.div_mul_cancel hdiv
  calc factors.prod * factor * (remaining / factor)
      = factors.prod * (remaining / factor * factor) := by ring
    _ = factors.prod * remaining := by rw [heq]
    _ = n := hprod

-- Helper for primality in append
theorem prime_of_mem_append_factor {p factor : ℕ} {factors : List ℕ} {remaining n : ℕ}
    (h : p ∈ factors ++ [factor])
    (hfac_ge : factor ≥ 2) (hrem_gt : remaining > 1)
    (hmod : remaining % factor = 0)
    (hinv : ∀ q ∈ factors, Nat.Prime q ∧ n % q = 0)
    (hno_smaller : ∀ q, Nat.Prime q → q < factor → remaining % q ≠ 0) :
    Nat.Prime p := by
  simp only [List.mem_append, List.mem_singleton] at h
  rcases h with hfactors | heq
  · exact (hinv p hfactors).1
  · rw [heq]
    have hdiv : factor ∣ remaining := Nat.dvd_of_mod_eq_zero hmod
    have hno_smaller' : ∀ q, Nat.Prime q → q < factor → ¬(q ∣ remaining) := by
      intro q hq hlt hdvd
      have := hno_smaller q hq hlt
      have : remaining % q = 0 := Nat.mod_eq_zero_of_dvd hdvd
      contradiction
    exact factor_is_prime remaining factor hrem_gt hfac_ge hdiv hno_smaller'

-- Helper for divisibility in append
theorem div_of_mem_append_factor {p factor : ℕ} {factors : List ℕ} {remaining n : ℕ}
    (h : p ∈ factors ++ [factor])
    (hmod : remaining % factor = 0)
    (hprod : factors.prod * remaining = n)
    (hinv : ∀ q ∈ factors, Nat.Prime q ∧ n % q = 0) :
    n % p = 0 := by
  simp only [List.mem_append, List.mem_singleton] at h
  rcases h with hfactors | heq
  · exact (hinv p hfactors).2
  · rw [heq]
    have hdiv : factor ∣ remaining := Nat.dvd_of_mod_eq_zero hmod
    have hrem_dvd_n : remaining ∣ n := by use factors.prod; linarith
```

```
    have hdiv_n : factor ∣ n := Nat.dvd_trans hdiv hrem_dvd_n
    exact Nat.mod_eq_zero_of_dvd hdiv_n

-- Solver hints
attribute [local solverHint] Nat.div_lt_self
attribute [local solverHint] Nat.Prime.two_le
attribute [local solverHint] List.prod_append
attribute [local solverHint] List.prod_singleton

set_option maxHeartbeats 4000000 in
set_option auto.smt.timeout 10 in
prove_correct factorize by
  loom_solve <;> try grind
  all_goals try simp only [List.prod_nil, List.Sorted, List.Pairwise.nil, one_mul]
  -- Handle product preservation
  all_goals try (
    have hdiv : factor ∣ remaining := Nat.dvd_of_mod_eq_zero ‹remaining % factor = 0›
    have hprod : factors.prod * remaining = n := ‹factors.prod * remaining = n›
    exact prod_append_div factors remaining factor n hdiv hprod
  )
  -- Handle remaining / factor ≥ 1
  all_goals try (
    have hdiv : factor ∣ remaining := Nat.dvd_of_mod_eq_zero ‹remaining % factor = 0›
    have hfac_pos : factor > 0 := by omega
    have hrem_ge : remaining ≥ factor := Nat.le_of_dvd (by omega) hdiv
    have := Nat.div_pos hrem_ge hfac_pos
    omega
  )
  -- Handle sorted append
  all_goals try (
    apply sorted_append_singleton ‹List.Sorted (· ≤ ·) factors›
    exact ‹∀ p ∈ factors, p ≤ factor›
  )
  -- Handle decreasing measure for if_pos branch
  all_goals try (
    have hdiv : factor ∣ remaining := Nat.dvd_of_mod_eq_zero ‹remaining % factor = 0›
    have hfac_ge : factor ≥ 2 := ‹factor ≥ 2›
    have hrem_gt : remaining > 1 := ‹remaining > 1›
    have hrem_ge : remaining ≥ factor := Nat.le_of_dvd (by omega) hdiv
    have hdiv_lt := Nat.div_lt_self (by omega : remaining > 0) (by omega : factor > 1)
    omega
  )
  -- Handle decreasing for else branch
  all_goals try (
    have hrem_gt : remaining > 1 := ‹remaining > 1›
    have hfac_ge : factor ≥ 2 := ‹factor ≥ 2›
    have hfac_le : factor ≤ remaining := by
      by_contra hgt
      push_neg at hgt
      have : remaining.minFac ≤ remaining := Nat.minFac_le (by omega)
      have hprime : Nat.Prime remaining.minFac := Nat.minFac_prime (by omega)
      have hdvd : remaining.minFac ∣ remaining := Nat.minFac_dvd remaining
      have hge2 : remaining.minFac ≥ 2 := Nat.Prime.two_le hprime
      have hlt : remaining.minFac < factor := by omega
      have := ‹∀ p, Nat.Prime p → p < factor → remaining % p ≠ 0› remaining.minFac hprime hlt
      have : remaining % remaining.minFac = 0 := Nat.mod_eq_zero_of_dvd hdvd
      contradiction
    omega
  )
  -- Handle Nat.Prime p for p ∈ factors ++ [factor]
  all_goals try (
    exact prime_of_mem_append_factor h ‹factor ≥ 2› ‹remaining > 1› ‹remaining % factor = 0›
      ‹∀ p ∈ factors, Nat.Prime p ∧ n % p = 0› ‹∀ p, Nat.Prime p → p < factor → remaining % p ≠ 0›
  )
  -- Handle n % p = 0 for p ∈ factors ++ [factor]
  all_goals try (
    exact div_of_mem_append_factor h ‹remaining % factor = 0› ‹factors.prod * remaining = n›
      ‹∀ p ∈ factors, Nat.Prime p ∧ n % p = 0›
  )
  -- Handle p ≤ factor for p ∈ factors ++ [factor]
  all_goals try (
    simp only [List.mem_append, List.mem_singleton] at h
    rcases h with hfactors | heq
    · exact ‹∀ p ∈ factors, p ≤ factor› p hfactors
    · rw [heq]
  )
  -- Handle remaining / factor % p ≠ 0 for primes p < factor
  all_goals try (
    have hdiv : factor ∣ remaining := Nat.dvd_of_mod_eq_zero ‹remaining % factor = 0›
    have hno := ‹∀ q, Nat.Prime q → q < factor → remaining % q ≠ 0› p ‹Nat.Prime p› ‹p < factor›
    intro hdivp
```

```
    have hdvd : p ∣ remaining / factor := Nat.dvd_of_mod_eq_zero hdivp
    have hdvd2 : p ∣ remaining := Nat.dvd_trans hdvd (Nat.div_dvd_of_dvd hdiv)
    have hmod0 : remaining % p = 0 := Nat.mod_eq_zero_of_dvd hdvd2
    contradiction
  )

#print axioms factorize_correct
```

H.3. Example: Kadane's algorithm

To give a full example of a standard algorithm that was verified by WybeCoder using GPT-5 using the subgoal decomposition pipeline described in Section 4.1, consider the maximum subarray problem: Given a list of numbers, the goal is to find the maximal sum of a contiguous sublist. Kadane's algorithm solves this task in $\mathcal{O}(n)$ time complexity. It is defined as follows:

Listing 6. Kadane's algorithm

```
def max_subarray(numbers):
    best = 0
    current = 0
    for x in numbers:
        current = max(0, current + x)
        best = max(best, current)
    return best
```

For the correctness proof, we argue as follows. The variable `current` holds the maximal subarray sum for subarrays that end at the index before `x`. The variable `best` holds the maximal subarray sum for the subarray that ends before `x`, i.e. the running maximum over the values of `current` with the straightforward update rule. The update rule for `current` considers two cases: If the maximal subarray contains `x`, it can be extended to the left using the maximal subarray that ends at the position before `x`. Otherwise, it is the empty subarray and `current` must be set to zero. Which of the two happens to be the case depends on whether `current + x` is positive or not. We leave the details to the reader to appreciate the complexity of writing a complete formal argument.

Listing 7 shows the full formal proof obtained with our pipeline. The method implementation uses *ghost variables* such as `curStart` which are only required to formulate the invariants as hinted at by the above informal argument. The actual algorithm matches the pseudo-code above exactly, while there is significant "formalization overhead" from invariant annotations and ghost variables updates. The proof consists of several lemmas generated by the prover agents (one of which for dealing with Loom/Velvet internals), followed by the main correctness theorem with the `loom_solve` tactic for generating verification conditions and discharging them to CVC5, and the reassembled subgoal proofs obtained from the subgoal proof agents. We note that the proving pipeline does not optimize proof length, maintainability or elegance, and that a minimal proof might be significantly shorter.

Listing 7. Implementation and correctness proof for Kadane's algorithm (`verina_advanced_46`)

```
import Auto
import Lean
import Mathlib

import CaseStudies.Velvet.Std
import CaseStudies.TestingUtil

set_option loom.semantics.termination "total"
set_option loom.semantics.choice "demonic"
set_option loom.solver "cvc5"
set_option auto.smt.timeout 3
set_option maxHeartbeats 100000
set_option auto.smt.trust true



method maxSubarraySum (numbers : List Int) return (result : Int)
  ensures let subArraySums :=
      List.range (numbers.length + 1) |>.flatMap (fun start =>
        List.range (numbers.length - start + 1) |>.map (fun len =>
          numbers.drop start |>.take len |>.sum))
    subArraySums.contains result ∧ subArraySums.all (· ≤ result)
  do
    let mut i : Nat := 0
    let mut rest : List Int := numbers
```

```
    let mut cur : Int := 0
    let mut max : Int := 0
    let mut curStart : Nat := 0
    let mut bestStart : Nat := 0
    let mut bestLen : Nat := 0
    while rest ≠ []
      invariant hi_len : i + rest.length = numbers.length
      invariant hrest_eq : rest = List.drop i numbers
      invariant hi_bounds : 0 ≤ i ∧ i ≤ numbers.length
      invariant hcurStart_le : curStart ≤ i
      invariant hcur_nonneg : 0 ≤ cur
      invariant hcur_sum : cur = (List.take (i - curStart) (List.drop curStart numbers)).sum
      invariant hcur_suffix_max :
        forall (start : Nat),
          start ≤ i →
          (List.take (i - start) (List.drop start numbers)).sum ≤ cur
      invariant hmax_nonneg : 0 ≤ max
      invariant hbest_sum : (List.take bestLen (List.drop bestStart numbers)).sum = max
      invariant hbest_start_in_prefix : bestStart ≤ i
      invariant hbest_end_in_prefix : bestStart + bestLen ≤ i
      invariant hprefix_max :
        forall (start : Nat) (len : Nat),
          start ≤ i ∧ start + len ≤ i →
          (List.take len (List.drop start numbers)).sum ≤ max
      done_with hdone : rest = []
      decreasing d : rest.length
    do
      let x := rest.head!
      let tail := rest.tail
      if cur + x ≤ 0 then
        cur := 0
        curStart := i + 1
      else
        cur := cur + x
      if max < cur then
        max := cur
        bestStart := curStart
        bestLen := (i + 1) - curStart
      i := i + 1
      rest := tail
    return max


@[simp] lemma WithName_erase_length {α} (l : List α) (n : Lean.Name) :
  (WithName.mk' l n).erase.length = l.length := by
  rfl

lemma take_succ_eq_append_head! {α} [Inhabited α]
    (m : List α) (k : Nat) (h : List.drop k m ≠ []) :
  List.take (k+1) m = List.take k m ++ [ (List.drop k m).head! ] := by
  revert m
  induction k with
  | zero =>
    intro m h
    cases m with
    | nil =>
      simp at h
    | cons x xs =>
      simp [List.drop, List.take, List.head!]
  | succ k ih =>
    intro m h
    cases m with
    | nil =>
      simp at h
    | cons x xs =>
      have hx : List.drop k xs ≠ [] := by
        simpa [List.drop] using h
      have ih' := ih xs hx
      simpa [List.drop, List.take, ih', List.cons_append]

lemma sum_take_succ_eq_sum_take_add_head! {α} [Inhabited α] [AddCommMonoid α]
  (l : List α) (n : Nat) (h : l.drop n ≠ []) :
  (l.take (n + 1)).sum = (l.take n).sum + (l.drop n).head! := by
  have hrepr := take_succ_eq_append_head! l n h
  calc
    (List.take (n+1) l).sum
        = (List.take n l ++ [ (List.drop n l).head! ]).sum := by simpa [hrepr]
    _ = (List.take n l).sum + [ (List.drop n l).head! ].sum := by
          simpa [List.sum_append]
    _ = (List.take n l).sum + (List.drop n l).head! := by simp
```

```
lemma sum_take_one_head! (l : List ℤ) (h : l ≠ []) : (List.take 1 l).sum = l.head! := by
  cases l with
  | nil => cases h rfl
  | cons a t => simp [List.take]

lemma sum_take_succ_of_drop_ne_nil (l : List ℤ) (n : ℕ) (hne : List.drop n l ≠ []) :
  (List.take (Nat.succ n) l).sum = (List.take n l).sum + (List.drop n l).head! := by
  revert l
  induction n with
  | zero =>
    intro l hne
    cases l with
    | nil =>
      cases hne rfl
    | cons x xs =>
      simp [List.take, List.drop, List.sum_cons, List.head!]
  | succ n ih =>
    intro l hne
    cases l with
    | nil =>
      simp at hne
    | cons x xs =>
      have hne' : List.drop n xs ≠ [] := by
        simpa [List.drop] using hne
      have ih' := ih xs hne'
      simp [List.take, List.drop, List.sum_cons, ih', add_comm, add_left_comm, add_assoc]

set_option maxHeartbeats 1000000 in
prove_correct maxSubarraySum by
  loom_solve
  case hi_len.loop_1 => (
  rcases List.exists_cons_of_ne_nil (l := rest) if_pos with ⟨x, xs, hrest⟩
  have hLHS :
      i + 1 + (WithName.mk' rest.tail (Lean.Name.anonymous.mkStr "rest")).erase.length
        = i + 1 + xs.length := by
    simp [WithName_erase_length, hrest]
  have hlen :
      i + xs.length + 1 = numbers.length := by
    simpa [hrest, Nat.add_comm, Nat.add_left_comm, Nat.add_assoc] using hi_len
  have hreassoc : i + 1 + xs.length = i + xs.length + 1 := by
    simpa [Nat.add_comm, Nat.add_left_comm, Nat.add_assoc]
  exact by
    rw [hLHS, hreassoc, hlen]
  )
  case hrest_eq.loop_1 => (
  try simp_all ; try grind
  )
  case hi_bounds.loop_3 => (
  cases rest with
  | nil =>
      cases if_pos rfl
  | cons x tail =>
      have hlen_pos : 0 < (x :: tail).length := by simp
      have hstep : i + 1 ≤ i + (x :: tail).length :=
        Nat.add_le_add_left (Nat.succ_le_of_lt hlen_pos) i
      simpa using (hi_len ▷ hstep)
  )
  case hcur_sum.loop_1 => (
  try simp_all ; try grind
  )
  case hcur_suffix_max.loop_1 => (
  classical
  by_cases hstart : start = i + 1
  · subst hstart
    simp
  ·
    have hlt : start < i + 1 := lt_of_le_of_ne a_2 hstart
    have hle_start_i : start ≤ i := Nat.le_of_lt_succ hlt
    have eqi : start + (i - start) = i := by
      simpa [Nat.add_comm] using (Nat.sub_add_cancel hle_start_i)
    have hlen_eq : i + 1 - start = (i - start) + 1 := by
      calc
        i + 1 - start
            = (start + (i - start) + 1) - start := by
                simpa [eqi, Nat.add_comm, Nat.add_left_comm, Nat.add_assoc]
        _ = ((i - start) + 1) := by
                simpa [Nat.add_comm, Nat.add_left_comm, Nat.add_assoc]
                  using (Nat.add_sub_cancel start ((i - start) + 1))
    have hdecomp :
        List.take (i + 1 - start) (List.drop start numbers)
          = List.take (i - start) (List.drop start numbers)
```

```
          ++ List.take 1 (List.drop i numbers) := by
    have := (List.take_add (l := List.drop start numbers) (i := i - start) (j := 1))
    simpa [hlen_eq, List.drop_drop, eqi, Nat.add_comm, Nat.add_left_comm, Nat.add_assoc] using this
  have hsum_eq :
      (List.take (i + 1 - start) (List.drop start numbers)).sum
        = (List.take (i - start) (List.drop start numbers)).sum
          + (List.take 1 (List.drop i numbers)).sum := by
    calc
      _ = ((List.take (i - start) (List.drop start numbers)) ++ List.take 1 (List.drop i numbers)).sum := by
            simp [hdecomp]
      _ = (List.take (i - start) (List.drop start numbers)).sum
            + (List.take 1 (List.drop i numbers)).sum := by
            simpa using (List.sum_append
              (List.take (i - start) (List.drop start numbers))
              (List.take 1 (List.drop i numbers)))
  have hsum_take1 :
      (List.take 1 (List.drop i numbers)).sum = (List.drop i numbers).head! := by
    have : (List.take 1 rest).sum = rest.head! := by
      cases rest <;> simp
    simpa [hrest_eq] using this
  have hprev_le_cur :
      (List.take (i - start) (List.drop start numbers)).sum ≤ cur :=
    hcur_suffix_max start hle_start_i
  have hcur_head_le0 :
      cur + (List.drop i numbers).head! ≤ 0 := by
    simpa [hrest_eq] using if_pos_1
  calc
    (List.take (i + 1 - start) (List.drop start numbers)).sum
        = (List.take (i - start) (List.drop start numbers)).sum
            + (List.take 1 (List.drop i numbers)).sum := hsum_eq
    _ = (List.take (i - start) (List.drop start numbers)).sum
            + (List.drop i numbers).head! := by simpa [hsum_take1]
    _ ≤ cur + (List.drop i numbers).head! := by
          exact add_le_add_right hprev_le_cur _
    _ ≤ 0 := hcur_head_le0
)
case hprefix_max.loop_1 => (
classical
cases Nat.lt_or_eq_of_le a_3 with
| inl hlt =>
  have hle_i : start + len ≤ i := Nat.lt_succ_iff.mp hlt
  have hstart_le_i : start ≤ i := Nat.le_trans (Nat.le_add_right _ _) hle_i
  exact hprefix_max start len hstart_le_i hle_i
| inr heq =>
  cases Nat.eq_or_lt_of_le a_2 with
  | inl hstart_eq =>
    have : (i + 1) + len = i + 1 := by simpa [hstart_eq] using heq
    have hlen0 : len = 0 := Nat.add_left_cancel this
    simp [hlen0, hmax_nonneg]
  | inr hstart_lt =>
    have hstart_le_i : start ≤ i := Nat.lt_succ_iff.mp hstart_lt
    have hlen_sub : (i + 1) - start = len := by
      have := congrArg (fun t => t - start) heq.symm
      simpa [Nat.add_sub_cancel] using this
    have hlen_succ : (i + 1) - start = (i - start) + 1 := by
      have hi_eq : start + (i - start) = i := Nat.add_sub_of_le hstart_le_i
      calc
        (i + 1) - start
            = ((start + (i - start)) + 1) - start := by simpa [hi_eq]
        _ = (start + ((i - start) + 1)) - start := by
              simpa [Nat.add_assoc]
        _ = (i - start) + 1 := by
              simpa using (Nat.add_sub_cancel start ((i - start) + 1))
    have hsum_decomp :
        (List.take len (List.drop start numbers)).sum
          =
        (List.take (i - start) (List.drop start numbers)).sum
          +
        (List.take 1 (List.drop i numbers)).sum := by
      calc
        (List.take len (List.drop start numbers)).sum
            = (List.take ((i + 1) - start) (List.drop start numbers)).sum := by
                simpa [hlen_sub]
        _ = (List.take ((i - start) + 1) (List.drop start numbers)).sum := by
                simpa [hlen_succ]
        _ = (List.take (i - start) (List.drop start numbers)).sum
              + (List.take 1 (List.drop (i - start) (List.drop start numbers))).sum := by
                simp [List.take_add, List.sum_append]
        _ = (List.take (i - start) (List.drop start numbers)).sum
              + (List.take 1 (List.drop i numbers)).sum := by
                have hi_eq : start + (i - start) = i := Nat.add_sub_of_le hstart_le_i
```

```
                have h1 := List.drop_drop (l := numbers) (i := i - start) (j := start)
                have hdrop_eq :
                    List.drop (i - start) (List.drop start numbers) = List.drop i numbers := by
                  simpa [Nat.add_comm, hi_eq] using h1
                simpa [hdrop_eq]
    have hdrop_ne_nil : List.drop i numbers ≠ [] := by simpa [hrest_eq] using if_pos
    have htake1_eq_head : (List.take 1 (List.drop i numbers)).sum = (List.drop i numbers).head! :=
      sum_take_one_head! _ hdrop_ne_nil
    have hold_le_cur :
        (List.take (i - start) (List.drop start numbers)).sum ≤ cur :=
      hcur_suffix_max start hstart_le_i
    have hsum_le_zero :
        (List.take (i - start) (List.drop start numbers)).sum
          + (List.drop i numbers).head!
          ≤ 0 := by
      have h1 :
          (List.take (i - start) (List.drop start numbers)).sum
            + (List.drop i numbers).head!
            ≤
          cur + (List.drop i numbers).head! :=
        add_le_add_right hold_le_cur _
      have h2 : cur + (List.drop i numbers).head! ≤ 0 := by
        simpa [hrest_eq, htake1_eq_head] using if_pos_1
      exact le_trans h1 h2
    have hnonneg_max : 0 ≤ max := hmax_nonneg
    have hsum_le_max :
        (List.take (i - start) (List.drop start numbers)).sum
          + (List.drop i numbers).head!
          ≤ max :=
      le_trans hsum_le_zero hnonneg_max
    simpa [hsum_decomp, htake1_eq_head]
      using hsum_le_max
)
case decreasing.loop_1 => (
classical
cases rest with
| nil =>
    cases if_pos rfl
| cons x xs =>
    change ((WithName.mk' xs (Lean.Name.anonymous.mkStr "rest")).erase.length) < Nat.succ xs.length
    have h : (WithName.mk' xs (Lean.Name.anonymous.mkStr "rest")).erase = xs := rfl
    simpa [h] using Nat.lt_succ_self xs.length
)
case hi_len.loop_2 => (
cases rest with
| nil =>
  cases if_pos rfl
| cons x xs =>
  have h_erase :
      (WithName.mk' ((x :: xs).tail) (Lean.Name.anonymous.mkStr "rest")).erase
      = (x :: xs).tail := rfl
  have hi_len' : i + (List.length (x :: xs)) = numbers.length := hi_len
  simpa [h_erase, List.tail, List.length, Nat.add_assoc, Nat.add_comm, Nat.add_left_comm] using hi_len'
)
case hrest_eq.loop_2 => (
try simp_all ; try grind
)
case hi_bounds.loop_5 => (
cases rest with
| nil =>
  cases if_pos rfl
| cons x xs =>
  have hi_len' : i + (xs.length + 1) = numbers.length := by
    simpa [List.length_cons, Nat.add_comm, Nat.add_assoc] using hi_len
  have h : i + 1 ≤ (i + 1) + xs.length := Nat.le_add_right (i + 1) xs.length
  have h' : i + 1 ≤ i + (xs.length + 1) := by
    simpa [Nat.add_assoc, Nat.add_comm] using h
  simpa [hi_len'] using h'
)
case hcur_sum.loop_2 => (
classical
have hrestne : List.drop i numbers ≠ [] := by simpa [hrest_eq] using if_pos
set xs := List.drop curStart numbers
set k := i - curStart
have hdrop_eq :
    List.drop k xs = List.drop i numbers := by
  simpa [xs, k, Nat.add_sub_of_le hcurStart_le] using
    (List.drop_drop (l := numbers) (n := curStart) (m := i - curStart))
have hdrop_ne : List.drop k xs ≠ [] := by
  intro h
  exact hrestne (by simpa [hdrop_eq] using h)
```

```
cases h : List.drop k xs with
| nil =>
    cases hdrop_ne (by simpa [h])
| cons y ys =>
    have hcons : List.drop k xs = y :: ys := h
    have hkadd : i + 1 - curStart = k + 1 := by
      have h' : curStart + (i - curStart) = i := Nat.add_sub_of_le hcurStart_le
      have step1 :
        i + 1 - curStart = (curStart + (i - curStart) + 1) - curStart := by
        simpa [Nat.add_assoc] using (congrArg (fun t => t + 1 - curStart) h').symm
      have step2 :
        (curStart + (i - curStart) + 1) - curStart
          = (curStart + ((i - curStart) + 1)) - curStart := by
        simpa [Nat.add_assoc]
      have step3 :
        (curStart + ((i - curStart) + 1)) - curStart
          = (i - curStart) + 1 := by
        simpa using (Nat.add_sub_cancel curStart ((i - curStart) + 1))
      exact by
        simpa [k] using (step1.trans (step2.trans step3))
    have hhead' :
      (List.drop k xs).head! = y := by
      simp [hcons]
    have hhead :
      (List.drop i numbers).head! = y := by
      simpa [hdrop_eq] using hhead'
    have h1 :
      ((List.take k xs).sum) + (List.drop i numbers).head!
        = ((List.take k xs).sum) + y := by
      simpa [hhead]
    have h2 :
      ((List.take k xs).sum) + y
        = ((List.take k xs) ++ [y]).sum := by
      simpa using (List.sum_append (l₁ := List.take k xs) (l₂ := [y])).symm
    have h3 :
      ((List.take k xs) ++ [y]).sum
        = ((List.take k xs) ++ List.take 1 (List.drop k xs)).sum := by
      simpa [hcons]
    have h4 :
      ((List.take k xs) ++ List.take 1 (List.drop k xs)).sum
        = (List.take (k + 1) xs).sum := by
      have htakeadd :
          List.take (k + 1) xs
            = List.take k xs ++ List.take 1 (List.drop k xs) := by
        simpa using (List.take_add (l := xs) (i := k) (j := 1))
      simpa [htakeadd]
    calc
      cur + (rest).head!
          = cur + (List.drop i numbers).head! := by simpa [hrest_eq]
      _ = ((List.take k xs).sum) + (List.drop i numbers).head! := by
        simpa [hcur_sum, xs, k]
      _ = ((List.take k xs).sum) + y := h1
      _ = ((List.take k xs) ++ [y]).sum := h2
      _ = ((List.take k xs) ++ List.take 1 (List.drop k xs)).sum := h3
      _ = (List.take (k + 1) xs).sum := h4
      _ = (List.take (i + 1 - curStart) (List.drop curStart numbers)).sum := by
        simpa [xs, k, hkadd]
)
case hcur_suffix_max.loop_2 => (
have hdne_drop_i : List.drop i numbers ≠ [] := by
  simpa [hrest_eq] using if_pos
have hhead_eq : (rest).head! = (List.drop i numbers).head! := by
  simpa [hrest_eq]
have hcases : start ≤ i ∨ start = i + 1 := by
  cases lt_or_ge i start with
  | inl hi_lt_start =>
    have h1 : i + 1 ≤ start := Nat.succ_le_of_lt hi_lt_start
    have h2 : start ≤ i + 1 := a_2
    exact Or.inr (le_antisymm h2 h1)
  | inr hge =>
    exact Or.inl hge
cases hcases with
| inl hstart_le_i =>
  have hdrop_eq :
      List.drop (i - start) (List.drop start numbers) = List.drop i numbers := by
    have hadd : start + (i - start) = i := by
      simpa [Nat.add_comm] using (Nat.sub_add_cancel hstart_le_i)
    simpa [List.drop_drop, hadd]
  have hlen : i + 1 - start = (i - start) + 1 := by
    simpa [Nat.succ_eq_add_one] using (Nat.succ_sub hstart_le_i)
  have hsplit :
```

```
      (List.take (i + 1 - start) (List.drop start numbers)).sum
        = (List.take (i - start) (List.drop start numbers)).sum + (List.drop i numbers).head! := by
    have hx : List.drop (i - start) (List.drop start numbers) ≠ [] := by
      exact fun h0 => hdne_drop_i (by simpa [hdrop_eq] using h0)
    have hsum := sum_take_succ_eq_sum_take_add_head! (l := List.drop start numbers) (n := i - start) (h := hx)
    simpa [hlen, hdrop_eq] using hsum
  have hpre_le : (List.take (i - start) (List.drop start numbers)).sum ≤ cur := by
    exact hcur_suffix_max start hstart_le_i
  calc
    (List.take (i + 1 - start) (List.drop start numbers)).sum
        = (List.take (i - start) (List.drop start numbers)).sum + (List.drop i numbers).head! := by
          simpa [hsplit]
    _ ≤ cur + (List.drop i numbers).head! := by
          exact add_le_add_right hpre_le _
    _ = cur + (rest).head! := by
          simpa [hhead_eq]
| inr hstart_eq_succ =>
  have : i + 1 - start = 0 := by
    simpa [hstart_eq_succ, Nat.sub_self]
  have pos : 0 < cur + (rest).head! := by
    exact lt_of_not_ge if_neg
  have nonneg : 0 ≤ cur + (rest).head! := le_of_lt pos
  simpa [this] using nonneg
)
case hbest_sum.loop_2 => (
have hdrop_ne' : List.drop i numbers ≠ [] := by
  simpa [hrest_eq] using if_pos
have idx_eq : curStart + (i - curStart) = i := by
  simpa [Nat.add_comm] using Nat.sub_add_cancel hcurStart_le
have hdrop_ne :
    List.drop (i - curStart) (List.drop curStart numbers) ≠ [] := by
  simpa [List.drop_drop, idx_eq] using hdrop_ne'
have hsum :=
  sum_take_succ_of_drop_ne_nil (List.drop curStart numbers) (i - curStart) hdrop_ne
have hlen_eq : i - curStart + 1 = i + 1 - curStart := by
  simpa [Nat.succ_eq_add_one] using (Nat.succ_sub hcurStart_le).symm
simpa [hlen_eq, List.drop_drop, idx_eq, hcur_sum, hrest_eq] using hsum
)
case hprefix_max.loop_2 => (
have hdrop_ne : List.drop i numbers ≠ [] := by
  simpa [hrest_eq] using if_pos
have hpos_curx : 0 < cur + (List.drop i numbers).head! := by
  exact not_le.mp (by
    simpa [hrest_eq] using if_neg)
by_cases hend_le_i : start + len ≤ i
·
  have hstart_le_i : start ≤ i := Nat.le_trans (Nat.le_add_right _ _) hend_le_i
  have hsum_le_max :
      (List.take len (List.drop start numbers)).sum ≤ max :=
    hprefix_max start len hstart_le_i hend_le_i
  have hmax_lt_curx_drop : max < cur + (List.drop i numbers).head! := by
    simpa [hrest_eq] using if_pos_1
  have hmax_le_curx_rest : max ≤ cur + rest.head! := by
    simpa [hrest_eq] using (le_of_lt hmax_lt_curx_drop)
  exact le_trans hsum_le_max hmax_le_curx_rest
·
  have hi_le_sum : i ≤ start + len := by
    cases Nat.le_total i (start + len) with
    | inl h => exact h
    | inr h =>
      exact (False.elim (hend_le_i h))
  have hi_lt_sum : i < start + len := Nat.lt_of_le_of_ne hi_le_sum (by
    intro h
    have hle : start + len ≤ i := by simpa [h] using le_of_eq (Eq.symm h)
    exact (hend_le_i hle).elim)
  have hge_succ : i + 1 ≤ start + len := Nat.succ_le_of_lt hi_lt_sum
  have hEq_end : start + len = i + 1 := Nat.le_antisymm a_3 hge_succ
  have hstart_le_or_eq := Nat.lt_or_eq_of_le a_2
  cases hstart_le_or_eq with
  | inr hstart_eq =>
    have hlen0 : len = 0 := by
      have := congrArg (fun t => t - start) hEq_end
      simpa [Nat.add_sub_cancel, hstart_eq] using this
    have : (List.take len (List.drop start numbers)).sum = 0 := by
      simpa [hlen0]
    have hnonneg_curx_rest : 0 ≤ cur + rest.head! := by
      simpa [hrest_eq] using (le_of_lt hpos_curx)
    simpa [this] using hnonneg_curx_rest
  | inl hstart_lt_succ =>
    have hstart_le_i : start ≤ i := Nat.lt_succ_iff.mp hstart_lt_succ
    have hlen_eq : len = i + 1 - start := by
```

```
  have := congrArg (fun t => t - start) hEq_end
  simpa [Nat.add_sub_cancel] using this
have hS_le_cur :
    (List.take (i - start) (List.drop start numbers)).sum ≤ cur :=
  hcur_suffix_max start hstart_le_i
have h_take_take :
    List.take (i - start)
      (List.take (i + 1 - start) (List.drop start numbers))
      = List.take (i - start) (List.drop start numbers) := by
  have hle : i - start ≤ i + 1 - start :=
    Nat.sub_le_sub_right (Nat.le_succ i) start
  simpa [List.take_take, Nat.min_eq_left hle]
have h_drop_take :
    List.drop (i - start)
      (List.take (i + 1 - start) (List.drop start numbers))
      = List.take ((i + 1 - start) - (i - start))
          (List.drop (i - start) (List.drop start numbers)) := by
  simpa [List.drop_take]
have hadd_eq : start + (i - start) = i := by
  have := Nat.sub_add_cancel hstart_le_i
  simpa [Nat.add_comm] using this
have hdiff_one :
    (i + 1 - start) - (i - start) = 1 := by
  calc
    (i + 1 - start) - (i - start)
        = (i + 1) - (start + (i - start)) := by
            simpa [Nat.sub_sub]
      _ = (i + 1) - i := by simpa [hadd_eq]
      _ = 1 := by simpa [Nat.add_comm] using Nat.add_sub_cancel i 1
have hdropdrop :
    List.drop (i - start) (List.drop start numbers) = List.drop i numbers := by
  simpa [Nat.add_comm, hadd_eq] using (List.drop_drop (i - start) start numbers)
have hdecomp :
    (List.take (i + 1 - start) (List.drop start numbers))
    = List.take (i - start)
        (List.take (i + 1 - start) (List.drop start numbers))
      ++
      List.drop (i - start)
        (List.take (i + 1 - start) (List.drop start numbers)) := by
  have := List.take_append_drop (i - start)
      (List.take (i + 1 - start) (List.drop start numbers))
  exact this.symm
have hsum_decomp :
    (List.take (i + 1 - start) (List.drop start numbers)).sum
    =
    (List.take (i - start) (List.take (i + 1 - start) (List.drop start numbers))).sum
    +
    (List.drop (i - start) (List.take (i + 1 - start) (List.drop start numbers))).sum := by
  simpa [List.sum_append] using congrArg List.sum hdecomp
have hsum_first :
    (List.take (i - start) (List.take (i + 1 - start) (List.drop start numbers))).sum
    = (List.take (i - start) (List.drop start numbers)).sum := by
  simpa [List.take_take, Nat.min_eq_left (Nat.sub_le_sub_right (Nat.le_succ i) start)]
    using congrArg List.sum h_take_take
have hsum_second :
    (List.drop (i - start) (List.take (i + 1 - start) (List.drop start numbers))).sum
    = (List.take 1 (List.drop i numbers)).sum := by
  have := congrArg List.sum h_drop_take
  simpa [hdiff_one, hdropdrop] using this
have hsum_take1_head :
    (List.take 1 (List.drop i numbers)).sum = (List.drop i numbers).head! := by
  cases hdi : List.drop i numbers with
  | nil =>
    cases hdrop_ne (by simpa [hdi])
  | cons y ys =>
    simp [hdi]
have hsum_eq :
    (List.take (i + 1 - start) (List.drop start numbers)).sum
    =
    (List.take (i - start) (List.drop start numbers)).sum
    + (List.drop i numbers).head! := by
  calc
    (List.take (i + 1 - start) (List.drop start numbers)).sum
        =
        (List.take (i - start) (List.take (i + 1 - start) (List.drop start numbers))).sum
        + (List.drop (i - start) (List.take (i + 1 - start) (List.drop start numbers))).sum := by
          exact hsum_decomp
    _ = (List.take (i - start) (List.drop start numbers)).sum
        + (List.take 1 (List.drop i numbers)).sum := by
          simpa [hsum_first, hsum_second]
    _ = (List.take (i - start) (List.drop start numbers)).sum
```

```
            + (List.drop i numbers).head! := by
              simpa [hsum_take1_head]
    have hS_le_cur' :
        (List.take (i - start) (List.drop start numbers)).sum ≤ cur := hS_le_cur
    have : (List.take (i + 1 - start) (List.drop start numbers)).sum ≤ cur + (List.drop i numbers).head! := by
      have hadded := add_le_add_right hS_le_cur' ((List.drop i numbers).head!)
      simpa [hsum_eq] using hadded
    simpa [hlen_eq, hrest_eq] using this
)
case decreasing.loop_2 => (
cases rest with
| nil =>
  cases if_pos rfl
| cons x xs =>
  have h :
    (WithName.mk' xs (Lean.Name.anonymous.mkStr "rest")).erase.length = xs.length := by
    rfl
  simpa [List.length, h] using Nat.lt_succ_self xs.length
)
case hi_len.loop_3 => (
have hpos : 0 < rest.length := List.length_pos_of_ne_nil if_pos
have hge1 : 1 ≤ rest.length := Nat.succ_le_of_lt hpos
have hlen_tail :
    (WithName.mk' rest.tail (Lean.Name.anonymous.mkStr "rest")).erase.length
      = rest.length - 1 := by
  have heq : (WithName.mk' rest.tail (Lean.Name.anonymous.mkStr "rest")).erase.length
               = rest.tail.length := rfl
  simpa [heq, List.length_tail]
have hsub : rest.length - 1 + 1 = rest.length := Nat.sub_add_cancel hge1
calc
  i + 1 + (WithName.mk' (rest).tail (Lean.Name.anonymous.mkStr "rest")).erase.length
      = i + 1 + (rest.length - 1) := by simpa [hlen_tail]
  _   = i + ((rest.length - 1) + 1) := by
          simpa [Nat.add_assoc] using
            (Nat.add_right_comm i 1 (rest.length - 1))
  _   = i + rest.length := by
          simpa [hsub]
  _   = numbers.length := by
          simpa [hi_len]
)
case hrest_eq.loop_3 => (
try simp_all ; try grind
)
case hi_bounds.loop_7 => (
have hpos : 0 < rest.length := List.length_pos_of_ne_nil if_pos
have hone : 1 ≤ rest.length := Nat.succ_le_of_lt hpos
have hle : i + 1 ≤ i + rest.length := Nat.add_le_add_left hone i
simpa [hi_len] using hle
)
case hcur_sum.loop_3 => (
have hrest_ne : List.drop i numbers ≠ [] := by
  simpa [hrest_eq] using if_pos
have hadd : curStart + (i - curStart) = i := Nat.add_sub_of_le hcurStart_le
have hdropdrop :
    List.drop (i - curStart) (List.drop curStart numbers)
      = List.drop i numbers := by
  simpa [hadd, Nat.add_comm, Nat.add_left_comm, Nat.add_assoc] using
    (List.drop_drop (l := numbers) (n := curStart) (m := i - curStart)).symm
have hdropne2 :
    List.drop (i - curStart) (List.drop curStart numbers) ≠ [] := by
  simpa [hdropdrop] using hrest_ne
have hsucc : i + 1 - curStart = (i - curStart) + 1 := by
  simpa [Nat.succ_eq_add_one] using (Nat.succ_sub hcurStart_le)
calc
  cur + (rest).head!
      = ((List.take (i - curStart) (List.drop curStart numbers))).sum
        + ((List.drop i numbers).head!) := by
        simpa [hcur_sum, hrest_eq]
  _ = ((List.take (i - curStart) (List.drop curStart numbers))).sum
        + ((List.drop (i - curStart) (List.drop curStart numbers)).head!) := by
        simpa [hdropdrop]
  _ = ((List.take ((i - curStart) + 1) (List.drop curStart numbers))).sum := by
        have hsum :=
          sum_take_succ_eq_sum_take_add_head!
            (l := List.drop curStart numbers) (n := i - curStart) (h := hdropne2)
        simpa using hsum.symm
  _ = (List.take (i + 1 - curStart) (List.drop curStart numbers)).sum := by
        simpa [hsucc]
)
case hcur_suffix_max.loop_3 => (
have hrestNE : List.drop i numbers ≠ [] := by simpa [hrest_eq] using if_pos
```

```
cases hdi : List.drop i numbers with
| nil =>
  cases hrestNE (by simpa [hdi])
| cons x xs =>
  have hhead_drop : (List.drop i numbers).head! = x := by simp [hdi]
  have hhead_rest : rest.head! = x := by simpa [hrest_eq, hdi] using hhead_drop
  have hcase : start ≤ i ∨ start = i + 1 := by
    cases lt_or_eq_of_le a_2 with
    | inl hlt =>
      exact Or.inl (Nat.lt_succ_iff.mp hlt)
    | inr heq =>
      exact Or.inr heq
  cases hcase with
  | inl hsi =>
    set L := List.drop start numbers
    set n1 := i - start
    have hLlen : L.length = numbers.length - start := by
      simpa [L] using List.length_drop start numbers
    have hn1_le_L : n1 ≤ L.length := by
      have : i - start ≤ numbers.length - start := Nat.sub_le_sub_right a_1 _
      simpa [hLlen, n1] using this
    have hlen_take : (List.take n1 L).length = n1 := by
      simpa [List.length_take, Nat.min_eq_left hn1_le_L]
    have hsub_eq : i + 1 - start = n1 + 1 := by
      have : (i - start) + start = i := Nat.sub_add_cancel hsi
      have : i + 1 = start + (n1 + 1) := by
        have : i + 1 = ((i - start) + start) + 1 := by simpa [this]
        simpa [n1, Nat.add_comm, Nat.add_left_comm, Nat.add_assoc] using this
      simpa [this, Nat.add_sub_cancel, n1]
    have hdropdrop : List.drop n1 L = List.drop i numbers := by
      have : List.drop n1 (List.drop start numbers) = List.drop (n1 + start) numbers := by
        simpa [Nat.add_comm] using (List.drop_drop numbers start n1)
      simpa [L, n1, Nat.sub_add_cancel hsi] using this
    have ht_decomp :
      List.take (n1 + 1) L
        = List.take n1 L ++ List.take ((n1 + 1) - (List.take n1 L).length) (List.drop n1 L) := by
      have h := List.take_append (l₁ := List.take n1 L) (l₂ := List.drop n1 L) (i := n1 + 1)
      simpa [List.take_append_drop, List.take_take, Nat.min_eq_right (Nat.le_succ n1)] using h
    have hsum_eq :
      (List.take (i + 1 - start) (List.drop start numbers)).sum
        = (List.take (i - start) (List.drop start numbers)).sum
          + (List.take 1 (List.drop i numbers)).sum := by
      calc
        (List.take (i + 1 - start) (List.drop start numbers)).sum
            = (List.take (n1 + 1) L).sum := by
              simp [L, n1, hsub_eq]
        _ = (List.take n1 L ++ List.take ((n1 + 1) - (List.take n1 L).length) (List.drop n1 L)).sum := by
              simpa [L, n1] using (congrArg List.sum ht_decomp)
        _ = (List.take n1 L).sum + (List.take ((n1 + 1) - (List.take n1 L).length) (List.drop n1 L)).sum := by
              simpa using (List.sum_append (List.take n1 L) (List.take ((n1 + 1) - (List.take n1 L).length)
   (List.drop n1 L)))
        _ = (List.take n1 L).sum + (List.take 1 (List.drop n1 L)).sum := by
              simp [hlen_take, Nat.add_comm, Nat.add_left_comm, Nat.add_assoc]
        _ = (List.take (i - start) (List.drop start numbers)).sum
            + (List.take 1 (List.drop i numbers)).sum := by
              simp [L, n1, hdropdrop]
    have hpref_le : (List.take (i - start) (List.drop start numbers)).sum ≤ cur :=
      hcur_suffix_max start hsi
    calc
      (List.take (i + 1 - start) (List.drop start numbers)).sum
          = (List.take (i - start) (List.drop start numbers)).sum
            + (List.take 1 (List.drop i numbers)).sum := hsum_eq
      _ = (List.take (i - start) (List.drop start numbers)).sum + x := by
            simp [hdi]
      _ ≤ cur + x := by
            exact add_le_add_right hpref_le x
      _ = cur + (rest).head! := by
            simpa [hhead_rest]
  | inr heq =>
    have hsum0 : (List.take (i + 1 - start) (List.drop start numbers)).sum = 0 := by
      simp [heq]
    have hpos : 0 < cur + (rest).head! := by
      exact lt_of_not_ge if_neg
    have hle : 0 ≤ cur + (rest).head! := le_of_lt hpos
    simpa [hsum0]
)
case hprefix_max.loop_3 => (
classical
-- From the loop condition, rest = numbers.drop i is nonempty
have hrest_nonempty : List.drop i numbers ≠ [] := by
  simpa [hrest_eq] using if_pos
```

```
-- Branch on whether start = i+1
by_cases hstart_eq_i1 : start = i + 1
· subst hstart_eq_i1
  have hle : i + 1 + len ≤ i + 1 := by simpa using a_3
  have hlen_le_zero : len ≤ 0 := by simpa using hle
  have hlen_zero : len = 0 := Nat.le_zero.mp hlen_le_zero
  simp [hlen_zero, hmax_nonneg]
·
  have hstart_le_i : start ≤ i := by
    have hlt : start < i + 1 := Nat.lt_of_le_of_ne a_2 hstart_eq_i1
    simpa [Nat.lt_succ_iff] using hlt
  by_cases h_le_i : start + len ≤ i
  · exact hprefix_max start len hstart_le_i h_le_i
  · -- so start + len = i + 1
    have hi1_le : i + 1 ≤ start + len := Nat.succ_le_of_lt (Nat.lt_of_not_ge h_le_i)
    have h_eq : start + len = i + 1 := Nat.le_antisymm a_3 hi1_le
    -- set k = i - start
    let k := i - start
    have hk : k + start = i := Nat.sub_add_cancel hstart_le_i
    -- len = k + 1
    have hlen_eq2 : len = k + 1 := by
      have h1 : start + len = start + (k + 1) := by
        have htmp : i = start + k := by
          have : i = k + start := by simpa using hk.symm
          simpa [Nat.add_comm] using this
        simpa [htmp, Nat.add_assoc] using h_eq
      exact Nat.add_left_cancel h1
    -- show drop k (drop start numbers) is nonempty
    have dd : List.drop k (List.drop start numbers) = List.drop (start + k) numbers := by
      simpa [Nat.add_comm, Nat.add_left_comm, Nat.add_assoc] using List.drop_drop k start numbers
    have hdrop_nonempty : List.drop k (List.drop start numbers) ≠ [] := by
      have hk_comm : start + k = i := by simpa [Nat.add_comm] using hk
      have : List.drop (start + k) numbers ≠ [] := by simpa [hk_comm] using hrest_nonempty
      simpa [dd] using this
    -- cur + head! ≤ max (since this branch didn't update max)
    have hcurhead_le_max0 :
        cur + rest.head! ≤ max := not_lt.mp if_neg_1
    have hcurhead_le_max :
        cur + (List.drop i numbers).head! ≤ max := by
      simpa [hrest_eq] using hcurhead_le_max0
    -- Bind old part by cur
    have hprefix_end_le_cur :
        (List.take k (List.drop start numbers)).sum ≤ cur := by
      simpa [k] using hcur_suffix_max start hstart_le_i
    -- Combine using the generic take-succ sum lemma
    have : (List.take len (List.drop start numbers)).sum ≤ max := by
      calc
        (List.take len (List.drop start numbers)).sum
            = (List.take (k + 1) (List.drop start numbers)).sum := by
              simpa [hlen_eq2]
        _ = (List.take k (List.drop start numbers)).sum +
            (List.drop k (List.drop start numbers)).head! := by
              simpa using (sum_take_succ_eq_sum_take_add_head!
                            (l := List.drop start numbers) (n := k) (h := hdrop_nonempty))
        _ ≤ cur + (List.drop k (List.drop start numbers)).head! :=
              add_le_add_right hprefix_end_le_cur _
        _ ≤ max := by
              -- rewrite the head via dd and hk
              have hk_comm : start + k = i := by simpa [Nat.add_comm] using hk
              have : (List.drop k (List.drop start numbers)).head!
                      = (List.drop (start + k) numbers).head! := by simpa [dd]
              have : cur + (List.drop (start + k) numbers).head! ≤ max := by
                simpa [hk_comm] using hcurhead_le_max
              simpa [this]
    exact this
)
case decreasing.loop_3 => (
cases rest with
| nil =>
  cases if_pos rfl
| cons x xs =>
  have h : (WithName.mk' xs (Lean.Name.anonymous.mkStr "rest")).erase = xs := rfl
  simpa [h] using Nat.lt_succ_self (List.length xs)
)
case hcur_suffix_max.entry => (
try simp_all ; try grind
)
case hprefix_max.entry => (
try simp_all ; try grind
)
case «ensures» => (
```

```
  rcases (by simpa using i_7) with ⟨hbLenEq, hbStartEq, hcurEq, hcurStartEq, hiEqI, hmaxEq, hrest1Eq⟩
  have hiEq : i = numbers.length := by
    simpa [hdone] using hi_len
  have hStartLE : bestStart ≤ numbers.length := by simpa [hiEq] using hbest_start_in_prefix
  have hEndLE' : bestStart + bestLen ≤ numbers.length := by simpa [hiEq] using hbest_end_in_prefix
  simp [hmaxEq, hbLenEq, hbStartEq, hiEq]
  refine ⟨bestStart, Nat.lt_succ_of_le hStartLE, ⟨bestLen, ?_, ?_⟩⟩
  have hRHSrewrite : bestStart + (numbers.length - bestStart) = numbers.length := by
    simpa [Nat.add_comm] using Nat.sub_add_cancel hStartLE
  have hineq : bestStart + bestLen ≤ bestStart + (numbers.length - bestStart) := by
    simpa [hRHSrewrite] using hEndLE'
  have hLenLE : bestLen ≤ numbers.length - bestStart :=
    Nat.le_of_add_le_add_left hineq
  exact Nat.lt_succ_of_le hLenLE
  simpa [hmaxEq] using hbest_sum
  )
  case ensures_1 => (
  have hi_eq : i = numbers.length := by
    simpa [hdone] using hi_len
  have hmax_eq : max = i_6 := by
    have := congrArg (fun (p : _) => p.snd.snd.snd.snd.snd.fst) i_7
    simpa using this
  refine (List.all_eq_true.mpr ?H)
  intro x hx
  rcases List.mem_flatMap.1 hx with ⟨start, hstart_mem, hx_mem⟩
  rcases List.mem_map.1 hx_mem with ⟨len, hlen_mem, rfl⟩
  have hstart_lt : start < numbers.length + 1 := List.mem_range.1 hstart_mem
  have hlen_lt   : len < numbers.length - start + 1 := List.mem_range.1 hlen_mem
  have hstart_le : start ≤ numbers.length := Nat.lt_succ_iff.mp hstart_lt
  have hlen_le   : len ≤ numbers.length - start := Nat.lt_succ_iff.mp hlen_lt
  have hstart_len_le : start + len ≤ numbers.length := by
    have h := Nat.add_le_add_left hlen_le start
    have hsum : start + (numbers.length - start) = numbers.length := Nat.add_sub_of_le hstart_le
    simpa [hsum] using h
  have hsum_le_max : (List.take len (List.drop start numbers)).sum ≤ max := by
    have h1 : start ≤ i := by simpa [hi_eq] using hstart_le
    have h2 : start + len ≤ i := by simpa [hi_eq] using hstart_len_le
    exact hprefix_max start len h1 h2
  have hsum_le_i6 : (List.take len (List.drop start numbers)).sum ≤ i_6 := by
    simpa [hmax_eq] using hsum_le_max
  exact (decide_eq_true_iff).2 hsum_le_i6
  )
```

I. Example proof: auto-active style

The following proof found by GPT-5 in a sequential agent scaffolding demonstrates the *auto-active* proving style where the user provides invariants, lemmas and solver hints and then hands over the remainder of the proof to SMT solvers (CVC5 via `loom_solve` and Lean's automation tools (`simp_all`).

Listing 8. Implementation and correctness proof for a power of two check algorithm (`verina_advanced_23`)

```
import Auto
import Lean
import Mathlib

import CaseStudies.Velvet.Std
import CaseStudies.TestingUtil

set_option loom.semantics.termination "total"
set_option loom.semantics.choice "demonic"
set_option loom.solver "cvc5"
set_option auto.smt.timeout 3
set_option maxHeartbeats 100000
set_option auto.smt.trust true

def pow (base : Int) (exp : Nat) : Int :=
  match exp with
  | 0 => 1
  | n+1 => base * pow base n

method isPowerOfTwo (n : Int) return (result : Bool)
  ensures if result then ∃ (x : Nat), (pow 2 x = n) ∧ (n > 0)
    else ¬ (∃ (x : Nat), (pow 2 x = n) ∧ (n > 0))
  do
    -- If n ≤ 0, it's not a power of two.
    if n ≤ 0 then
      return false
```

```
    else
      -- We iterate candidate powers p = 2^i while p < n and i < toNat n.
      -- Bounding i by toNat n gives a simple decreasing measure (toNat n - i).
      let nNat := Int.toNat n
      let mut p : Int := 1
      let mut i : Nat := 0
      while p < n ∧ i < nNat
        invariant n > 0
        invariant i ≤ nNat
        -- p tracks exactly 2^i
        invariant p = pow 2 i
        -- All earlier powers are strictly below n
        invariant ∀ j, j < i → pow 2 j < n
        -- Upon termination, we will have n ≤ p
        done_with n ≤ p
        decreasing nNat - i
      do
        -- Move to the next power: 2^(i+1) = 2 * 2^i
        p := 2 * p
        i := i + 1
      -- After the loop, return whether p equals n.
      return p = n

-- Helper lemmas for rewriting and monotonicity of pow with base 2.

-- Unfolding step: 2^(i+1) = 2 * 2^i
theorem pow_two_succ (i : Nat) : pow 2 (i+1) = 2 * pow 2 i := by
  simp [pow]

-- Multiplicative law for our Int-based pow: b^(i+k) = b^i * b^k
theorem pow_int_add (b : Int) (i k : Nat) : pow b (i + k) = pow b i * pow b k := by
  induction k with
  | zero =>
    simp [pow, Nat.add_comm, Nat.add_left_comm, Nat.add_assoc]
  | succ k hk =>
    -- LHS: b^(i+k+1) = b * b^(i+k)
    -- RHS: b^i * b^(k+1) = b^i * (b * b^k)
    -- Using IH to rewrite b^(i+k), commute/associate multiplications.
    simpa [pow, Nat.add_comm, Nat.add_left_comm, Nat.add_assoc, hk] using
      show b * (pow b i * pow b k) = pow b i * (b * pow b k) from by
        calc
          b * (pow b i * pow b k)
              = (b * pow b i) * pow b k := by simpa [mul_assoc]
          _   = (pow b i * b) * pow b k := by simpa [mul_comm]
          _   = pow b i * (b * pow b k) := by simpa [mul_assoc]

-- For base 2, powers are at least 1.
theorem pow_two_ge_one (i : Nat) : (1 : Int) ≤ pow 2 i := by
  induction i with
  | zero => simp [pow]
  | succ i hi =>
    -- 2 * (2^i) ≥ 2 ≥ 1
    have hmul : (2 : Int) ≤ 2 * pow 2 i := by
      -- If 1 ≤ pow 2 i and 0 ≤ 2 then 2*1 ≤ 2*(pow 2 i).
      have := mul_le_mul_of_nonneg_left hi (by simp : (0 : Int) ≤ 2)
      simpa using this
    exact le_trans (by simp : (1 : Int) ≤ 2) hmul

-- Monotonicity: if i ≤ j then 2^i ≤ 2^j
theorem pow_two_le_of_le (i j : Nat) (h : i ≤ j) : pow 2 i ≤ pow 2 j := by
  rcases Nat.exists_eq_add_of_le h with ⟨k, hk⟩
  have hi_nonneg : (0 : Int) ≤ pow 2 i := le_trans (by simp : (0 : Int) ≤ 1) (pow_two_ge_one i)
  have hk_ge1 : (1 : Int) ≤ pow 2 k := pow_two_ge_one k
  have hmul : pow 2 i ≤ pow 2 i * pow 2 k := by
    have := mul_le_mul_of_nonneg_left hk_ge1 hi_nonneg
    simpa using this
  simpa [hk, pow_int_add] using hmul

-- Growth lower bound: for all n, 2^n ≥ n (as Int)
theorem pow_two_ge_nat (n : Nat) : (n : Int) ≤ pow 2 n := by
  induction n with
  | zero => simp [pow]
  | succ k hk =>
    -- k + 1 ≤ 2^k + 2^k = 2 * 2^k = 2^(k+1)
    have hsum : (k + 1 : Int) ≤ pow 2 k + pow 2 k := by
      exact add_le_add hk (pow_two_ge_one k)
    simpa [pow, two_mul] using hsum

-- Main postcondition helper: given the facts at loop exit,
-- decide the existence of x with pow 2 x = n based on whether p = n.
theorem post_from_facts
```

```
  (n : Int) (i : Nat) (p : Int)
  (hpos : 0 < n) (hp : p = pow 2 i) (hprev : ∀ j, j < i → pow 2 j < n)
  (hle : n ≤ p)
  : (if p = n then ∃ x, pow 2 x = n else ∀ x, pow 2 x ≠ n) := by
  by_cases h : p = n
  · simp [h]
    exact ⟨i, hp.symm.trans h⟩
  · have hlt : n < p := lt_of_le_of_ne hle (Ne.symm h)
    have hn : n ≠ p := ne_of_lt hlt
    simp [h]
    intro x
    by_cases hx : x < i
    · intro hxeq
      have hlt' : pow 2 x < n := hprev x hx
      have : pow 2 x < pow 2 x := by simpa [hxeq] using hlt'
      exact lt_irrefl _ this
    · have hix : i ≤ x := le_of_not_lt hx
      have hmon : pow 2 i ≤ pow 2 x := pow_two_le_of_le i x hix
      intro hxeq
      have : n < pow 2 x := lt_of_lt_of_le hlt (by simpa [hp] using hmon)
      have : n < n := by simpa [hxeq] using this
      exact lt_irrefl _ this

attribute [local solverHint] pow_two_succ pow_int_add pow_two_ge_one pow_two_le_of_le pow_two_ge_nat
attribute [local solverHint] post_from_facts
attribute [local solverHint] Int.toNat_of_nonneg

prove_correct isPowerOfTwo by
  -- SMT will handle most obligations; remaining ones are discharged by simp_all,
  -- and the solver can use the helper lemma via solverHint.
  loom_solve <;> simp_all
```

I.1. Termination in the Clever benchmark

In the Clever benchmark, termination is formulated using a "the output exists" formulation such as in the following example.

```
def problem_spec
  (implementation: List Int → Int → Bool)
  (q: List Int) (w: Int) :=
  let spec (result : Bool) :=
    result ↔ (List.Palindrome q) ∧ (List.sum q ≤ w)
  ∃ result, implementation q w = result ∧ spec result
```

We note here that this is unnecessary because functions such as `implementation: List Int → Int → Bool` in Lean are always total (proved to terminate by the equation compiler), as non-terminating functions would lead to inconsistencies in the logic (example: a non-terminating function which produces proofs of `False`). The existential statement `∃ result, f x = result` can always trivially be proved with `⟨f x, rfl⟩`.

On the other hand, for imperative code, proof of termination is typically more involved, and covered by `decreasing` annotations in Velvet's syntax and the corresponding proof obligations.

J. Prompt

We layout the procedure of the evolution of the prompts used in our framework. We provide Gemini 2.5 Pro with the Velvet documentation[2] and instruct it to write a prompt for an agentic system which do not have prior knowledge of it. We use this version of prompt for experiments during the development of the framework. The prompt is refined by us through manually looking at early failure cases. This version of prompt gives 53.44% of pass@4 on Verina with GPT-5 sequential agent.

In later iterations, we provide Claude Code with the whole trajectories directory and instruct it to further enhance the prompt. Claude Code is able to dynamically explore trajectory files, write scripts to gather statistics and spot several failing patterns, such as the usage of sorry and non-existing lemma, and add extra instruction, e.g., retry different strategies after consistent failures. With the enhanced prompt, pass@4 on Verina with GPT-5 sequntial agent increases to 56.08%.

However, this step also introduces some inconsistencies and misconceptions about Loom/Lean. We conclude the final step by looking at the prompt manually and cleaning up these inconsistencies. The final version of the prompt, shown in Listing 9, results in 58.73% pass@4 on Verina with the same setup.

[2] https://github.com/verse-lab/loom/blob/master/CaseStudies/Velvet/velvet_documentation.md

Listing 9. Final prompt for sequential agent, with highlights showing the diff compared with the first version

```
You are a world-class expert in formal verification and an assistant specializing in the Lean 4 theorem prover.
    Your task is to write and prove the correctness of programs written in **Velvet**, a Dafny-like language
    shallowly embedded in Lean 4 via the **Loom** framework.

Your primary goal is to take a given program specification and produce a complete, verifiable Velvet program,
    including the necessary pre-conditions, post-conditions, loop invariants, and a `prove_correct` block that
    successfully verifies the implementation.

## 1. Core Concepts of Loom and Velvet

* **Loom Framework**: Loom is a framework built in Lean 4 for creating program verifiers. It automatically
    generates weakest preconditions to create proof obligations (Verification Conditions or VCs).
* **Velvet Language**: Velvet is a specific verifier built with Loom. It provides a syntax inspired by Dafny for
    writing imperative programs with specifications.
* **Shallow Embedding**: Velvet is not a separate language with its own compiler. Instead, its constructs
    (`method`, `while`, etc.) are macros that translate directly into Lean 4 monadic computations. This means you
    can use the full power of Lean within your proofs.
* **Hybrid Verification**: The main verification strategy is a hybrid one:
    1.  **Automated SMT Solving**: The primary tactic, `loom_solve`, attempts to automatically discharge all
    proof obligations by translating them into SMT queries (using `cvc5`).
    2.  **Interactive Proving**: If `loom_solve` fails, you can look at the remaining goals, prove theorems and
    use them to complete the proof interactively using standard Lean tactics (`simp`, `grind`, `rw`, `intros`,
    etc.).

---

## 2. Velvet Language Syntax

You must adhere to the following syntax for writing Velvet programs and proofs.

### Method Definition

A program is defined as a `method`. It includes typed arguments, named return values, pre-conditions (`require`),
    and post-conditions (`ensures`).

**Syntax:**
```lean
method <method_name> (<arg1>: <Type1>) (mut <arg2>: <Type2>) return (<ret_val>: <Type>)
  require <precondition_1>
  require <precondition_2>
  ensures <postcondition_1>
  ensures <postcondition_2>
  do
    -- method body
```
* Use `mut` for parameters that are modified by the method (like an array being sorted in-place). In
    post-conditions, the original value is `<arg2>` and the final value is `<arg2>New`.
* Lean's logical symbols are used: `∧` (and), `∨` (or), `→` (implies), `∀` (forall), `∃` (exists).

Method Body

The method body is a `do` block containing imperative statements.

* **Variable Declaration**: `let mut <name> := <value>`
* **Assignment**: `<name> := <new_value>`
* **Conditionals**: Standard `if/then/else`. **Important**: Avoid `else if`. Instead, nest the `if` inside the
    `else` block.
    ```lean
    -- Correct nesting
    if cond1 then
      ...
    else
      if cond2 then
        ...
      else
        ...
    ```
* **Return**: `return <value>`

Loops

Velvet only has `while` loops. They must be annotated with invariants.

Syntax:
```lean
while <condition>
  invariant <invariant_1>
  invariant <invariant_2>
  done_with <optional_termination_condition>
```
```

```
  do
    -- loop body
  ```
  * **`invariant`**: A property that is true at the beginning of every loop iteration. It is essential for the
      proof.
  * **`done_with`**: A condition that is true upon loop termination. If omitted, it defaults to the negation of the
      loop condition.

  ### Data Types and Operations

  Velvet uses standard Lean data types. `Int`, `Nat` (or `ℕ`), and `Array <Type>` are common.

  * **Array Size**: `<array>.size`
  * **Array Access**: `<array>[<index>]!` (the `!` is crucial).
  * **Array Update**: `arr := Array.set! arr i value`. This is functional; it returns a *new* array.
  * **Array Creation**: `Array.replicate <size> <default_value>`

  ## 3. Common Mistakes

  * **Never use `return` inside a `while` loop body.** Loop bodies must signal continuation/termination of the *
      loop*, not the entire method. Instead, use a boolean flag pattern: initialize a flag variable, include it in
      the loop condition (`while i < n ∧ all_ok`), add an invariant linking the flag to the checked property, and
      set the flag instead of returning early. See Example 3 in Section 5 for a complete example.
  * **Empty else branches**: Don't write `else ()` or `else skip`. Simply omit the `else` branch. In other cases,
      perform a benign assignment like `x := x` if needed.
  * **Chained inequalities**: Write `a < b ∧ b < c` instead of `a < b < c`.

  ## 4. The Verification Process

  Every `method` must be followed by a `prove_correct` block.

  Syntax:
  ```lean
  prove_correct <method_name> by
    <tactics>
  ```

  ### A. The Main Tactic: `loom_solve`

  Always start with `loom_solve`. This tactic generates the verification conditions and attempts to solve all
      verification conditions automatically using SMT.
+ Otherwise, it will leave the remaining unproven goals. It is thus often a good idea to chain `<;> try grind`
+ after `loom_solve` which might further close easy goals.

  ```lean
  prove_correct my_method by
-   loom_solve
+   loom_solve <;> try grind
  ```

  ### B. Strategies for Handling Proof Failures

  If `loom_solve` fails, use the following strategies:

  #### Provide Solver Hints

  The SMT solver may lack knowledge of specific mathematical lemmas. Provide these using the `attribute [local
      solverHint]` command *before* the `prove_correct` block.

  Common Necessary Hints:
  * For modular arithmetic: `attribute [local solverHint] Nat.mod_lt`
  * For array size properties: `attribute [local solverHint] Array.size_replicate Array.size_set`
  * For array element access after updates: `attribute [local solverHint] Array.get_set_c`
  * For array swaps: `attribute [local solverHint] Array.multiset_swap`

  #### Use Interactive Tactics

  If hints are not enough, `loom_solve` will leave the remaining unproven goals. Use standard Lean tactics to solve
      them. A common pattern is to use finishing tactics such as `simp_all` or `grind` after `loom_solve`.

  Example:
  ```lean
  prove_correct complex_method by
-   loom_solve <;> simp_all -- try simp on all generated subgoals
+   loom_solve <;> try grind
```

```
    -- Now, handle remaining goals manually
    { intros k hk
      have := my_theorem k hk
      by_cases h : k = i <;> simp_all }
```
```

+ **Important:** If a finishing tactic does not succeed, try other finishing tactics too, for example:
+ - `grind`: general purpose automation, includes domain solvers, can handle logical connectives and quantifiers.
+ - `omega`: for arithmetic goals over `Int`/`Nat` with linear arithmetic if `grind` fails.
+ - `decide`: for decidable propositions, especially concrete equalities.
+ - `native_decide`: fast version of `decide`.
+ - `simp_all`: simplifies all goals and hypotheses recursively.

+ Never rely on one of them alone as your primary tactic for closing goals. Choose tactics based on the goal
+ structure.

+ **If automatic finishing tactics are not sufficient**, analyze the unsolved goals and fall back to a full
+ interactive proof.

+ #### Prove a Theorem

+ You can prove a helper theorem separately and use it in an interactive proof of the main correctness theorem or
+ add it as a solver hint.

+ **Example:**
+ ```lean
+ @[local solverHint]
+ theorem my_get_set_c [Inhabited α] (i j: Nat) (val: α) (arr: Array α):
+   i < arr.size → (arr.set! j val)[i]! = if i = j then val else arr[i]! := by
+   simp [Array.get_set_c]
+ ```

+ #### Adapt Invariants

+ Reason whether the loop invariants are actually correct based on the failed goals.

+ * **Invariant too strong**: If an invariant fails to hold during the loop:
+   - The invariant may be claiming more than the loop actually maintains.
+   - Weaken the invariant or add preconditions to when it holds.
+ * **Hypotheses too weak**: If a goal fails to be proved, it may be because the hypotheses that stem from other
+     invariants are too weak.
+   - The hypothesis invariants may not capture enough information about the loop state.
+   - Check: Do the hypothesis invariants mention ALL mutable variables in the loop?
+   - Check: Do the hypothesis invariants relate the current state to what you'll need in the conclusion?

#### Handle Timeouts

If the proof times out, you can increase resource limits using scoped options. Use these options only when
    encountering timeout errors, not preemptively.

**Syntax:**
```lean
set_option maxHeartbeats 1000000 in
set_option auto.smt.timeout 8 in
prove_correct method_name by
  loom_solve
```

Common timeout options:
* `maxHeartbeats`: Increases computation budget (default: 200000)
* `auto.smt.timeout`: Increases SMT solver timeout in seconds (default: 4)

+ **Critical**: When escalating resources, increase by **at least 10x**, not 2x:
+ * First timeout: 100k → 1M (10x)
+ * Second timeout: 1M → 10M (10x)
+ * If you hit 3 timeouts at progressively higher limits with the same proof code, the proof strategy is
+ fundamentally wrong. Change your approach rather than just increasing resources further.

+ ### C. Troubleshooting Proof Failures
```

+ When `loom_solve` leaves unsolved goals, follow this systematic approach:

+ #### Step 1: Analyze the Goal Type

+ Read the unsolved goal carefully. Different goal patterns require different tactics:

+ | Goal Pattern | Recommended Tactic |
+ |--------------|--------------------|
+ | Arithmetic over Int/Nat (contains `+`, `-`, `<`, `≤`, `*`, `%`) | `omega` |
+ | Decidable equality (`x == y` for concrete values) | `decide` |
+ | Case analysis needed (disjunction, match on structure) | `cases h` or `rcases h with ...` |
+ | Array indexing properties (`arr[i]!`) | Add `solverHint` for `Array.get_set_c`, etc. |
+ | Quantifier goals (`∀`, `∃`) | `intro`, `use`, or `grind` |
+ | Cleanup/simplification | `simp_all` (only AFTER other tactics) |

+ #### Step 2: If the Same Error Repeats 3+ Times

+ **STOP** using the same tactic. This indicates the approach is not working. Instead:

+ 1. Re-read the unsolved goal to understand what property is missing
+ 2. Check if an invariant is too weak → strengthen it to capture more information
+ 3. Try a fundamentally different tactic from Step 1 based on the goal structure
+ 4. Consider if a helper lemma is needed to bridge the gap

+ Do NOT just retry `loom_solve <;> simp_all` repeatedly when it keeps failing on the same goal.

+ ### D. Available Solver Hints

+ When adding `attribute [local solverHint]` declarations, **only use lemma names that actually exist**. Here is a
+ reference list of commonly available hints in the Loom/Velvet framework:

+ #### Array Operations
+ * `Array.size_set` - size of array after set operation
+ * `Array.size_swap` - size of array after swap operation
+ * `Array.get_set_c` - accessing array after set at same/different index
+ * `Array.multiset_swap` - swap preserves multiset (permutation property)
+ * `Array.size_replicate` - size of array created by replicate

+ #### Arithmetic
+ * `Nat.mod_lt` - modular arithmetic bounds
+ * `Nat.sub_add_cancel` - subtraction/addition identity
+ * `Int.add_comm`, `Int.mul_comm` - commutativity properties

+ #### DO NOT USE (these do not exist)
+ * ✗ `Array.get_swap_c` - does NOT exist (confusion with `Array.get_set_c`)
+ * ✗ `<MethodName>_correct` - not a standard naming convention in this framework
+ * ✗ `List.qsort` - Lean 4 uses different sorting functions

+ **If you need a lemma and aren't sure of its name**: Check whether you can just quickly prove it. Guessing is a
+ last resort but costs time and likely leads to "unknown constant" errors.

5. Complete Examples

Here are complete, correct examples demonstrating the required structure and syntax.

Example 1: Square Root

This example shows a simple loop, invariants, and a successful proof using `loom_solve`.

```lean
import Auto
import Lean
import Mathlib
import Aesop

import CaseStudies.Velvet.Std
import CaseStudies.TestingUtil
```

```
set_option loom.semantics.termination "total"
set_option loom.semantics.choice "demonic"
set_option auto.smt.trust true
set_option loom.solver "cvc5"
set_option auto.smt.timeout 4
set_option maxHeartbeats 200000

-- Method definition with specification
method sqrt (x: ℕ) return (res: ℕ)
  ensures res * res ≤ x
  ensures ∀ i, i ≤ res → i * i ≤ x
  ensures ∀ i, i * i ≤ x → i ≤ res
  do
    if x = 0 then
      return 0
    else
      let mut i := 0
      -- Loop to find the integer square root
      while i * i ≤ x
      -- Invariant: for all numbers j checked so far, their square is <= x
      invariant ∀ j, j < i → j * j ≤ x
      -- because Lean's natural number substraction truncates at zero,
      -- add "padding" with `+ 1`
      decreasing x + 1 - i
      do
        i := i + 1
      -- The loop overshoots by one
      return i - 1

-- Verification of the method
prove_correct sqrt by
-  loom_solve
+  loom_solve <;> try grind

-- Verification check that no auxiliary axioms were used
#print axioms sqrt_correct
```

Example 2: Insertion Sort

This example is more complex, demonstrating mutable array parameters, nested loops, and the need for solver hints.

```lean
import Auto
import Lean
import Mathlib
import Aesop

import CaseStudies.Velvet.Std
import CaseStudies.TestingUtil

set_option loom.semantics.termination "total"
set_option loom.semantics.choice "demonic"
set_option auto.smt.trust true
set_option loom.solver "cvc5"
set_option auto.smt.timeout 4
set_option maxHeartbeats 200000

-- Method definition to sort an array in-place, with specification
method insertionSort
  (mut arr: Array Int) return (u: Unit)
  require 1 ≤ arr.size
  -- Postcondition: The array is sorted
  ensures forall i j, 0 ≤ i ∧ i ≤ j ∧ j < arr.size → arr[i]! ≤ arr[j]!
  -- Postcondition: The elements are a permutation of the original elements
  ensures arrOld.toMultiset = arr.toMultiset
  do
    let arr₀ := arr
    let arr_size := arr.size
    let mut n := 1
    -- Outer loop: considers prefixes of the array from left to right
    while n ≠ arr.size
    invariant arr.size = arr_size
    invariant 1 ≤ n ∧ n ≤ arr.size
    invariant forall i j, 0 ≤ i ∧ i < j ∧ j <= n - 1 → arr[i]! ≤ arr[j]!
    invariant arr.toMultiset = arr₀.toMultiset
    -- explicit decreasing measure for loop termination
    decreasing arr.size - n
    do
      let mut mind := n
```

```
      -- Inner loop: inserts the element at `n` into the sorted prefix
      while mind ≠ 0
      invariant arr.size = arr_size
      invariant mind ≤ n
      -- Invariant: The prefix is sorted except possibly at `mind`
      invariant forall i j, 0 ≤ i ∧ i < j ∧ j ≤ n ∧ j ≠ mind → arr[i]! ≤ arr[j]!
      invariant arr.toMultiset = arr₀.toMultiset
      decreasing mind
      do
        if arr[mind]! < arr[mind - 1]! then
          swap! arr[mind - 1]! arr[mind]!
        mind := mind - 1

      n := n + 1
    return

prove_correct insertionSort by
-   loom_solve
+   loom_solve <;> try grind

-- Verification check that no auxiliary axioms were used
#print axioms insertionSort_correct
```

Example 3: Boolean Flag Pattern for Early Exit

This example demonstrates the correct way to handle early loop termination using a boolean flag (avoiding the common mistake from Section 3).

```
import Auto
import Lean
import Mathlib
import Aesop

import CaseStudies.Velvet.Std
import CaseStudies.TestingUtil

set_option loom.semantics.termination "total"
set_option loom.semantics.choice "demonic"
set_option auto.smt.trust true
set_option loom.solver "cvc5"
set_option auto.smt.timeout 4
set_option maxHeartbeats 400000

def isOdd (n : Int) : Bool := n % 2 == 1

-- Method to check if all odd indices contain odd numbers
method isOddAtIndexOdd (a : Array Int) return (result : Bool)
  ensures result ↔ (∀ i, (hi : i < a.size) → isOdd i → isOdd (a[i]))
  do
    let n := a.size
    let mut i : Nat := 0
    let mut all_ok := true
    -- Loop continues while checking AND flag is still true
    while i < n ∧ all_ok
      invariant 0 ≤ i ∧ i ≤ n
      -- Key invariant: flag reflects property over scanned portion
      invariant all_ok ↔ (∀ j, (hj : j < i) → isOdd (j : Int) → isOdd (a[j]!))
      decreasing n - i
    do
      if isOdd (i : Int) ∧ !(isOdd (a[i]!)) then
        all_ok := false  -- Set flag instead of returning
      i := i + 1
    return all_ok  -- Single return after loop

prove_correct isOddAtIndexOdd by
-   loom_solve
+   loom_solve <;> try grind

-- Verification check that no auxiliary axioms were used
#print axioms isOddAtIndexOdd_correct
```

6. Important Notes and Hints

- **Do NOT add any guard statements**: You must not add any new `guard` statements in your generated Lean code. If the input or instructions contain any guard statements, you must exclude them from your final Lean code

```
    output.
- Due to a bug in Velvet, doc comments (`/- ... -/`) are not supported. Use `--` instead.
- `skip` is not a valid statement in Velvet. You can simply drop empty `else` clauses completely.
- Lean does not have inequality chaining (e.g., `a < b < c`). Instead, use `a < b ∧ b < c`.
- Do NOT change the header (the `import` clauses) unless strictly necessary.
- In particular, keep the full `import Mathlib` which makes all definitions, theorems and lemmas
  available. If you see `unknown constant` errors, the name of the constant is incorrect, not the `import`.
  There is NO point in adding additional finer-grained mathlib imports.
- If you see `unknown namespace` errors, it is likely that the error actually comes from a non-existing import.
+ - The named variable for the result is only available for post-conditions, not in the method body.
+   Use `return <value>` to return a value, or declare a `let mut` variable to hold tentative results.
+   You cannot assign to the result variable directly.
+ - Note that optional `done_with` statements come before `decreasing` measure statements.
+ - In total correctness semantics, all loops must be annotated with a `decreasing` measure to ensure termination.
+ - Recursive methods are not supported in Velvet. Write fully imperative code using loops and conditionals.
+ - Use `List.toArray` to convert lists to arrays if needed.
- You should NOT skip ANY proofs for brevity and you should NEVER use `sorry` as a placeholder for a proof.
+   **If you cannot complete a proof after multiple attempts, you MUST:**
+   1. Explain which specific goal is blocking you and what it requires
+   2. Suggest what lemma or invariant strengthening might be needed
+   3. Still provide code WITHOUT `sorry` that gets as far as possible with partial tactics
+   Do NOT introduce `sorry` to make compilation succeed. Leaving unsolved goals is better than using `sorry`.

## 7. Imperative Code Requirement

You **must** implement the algorithm in an **imperative** style suitable for Loom/Velvet verification.

### 1. Imperative core

- The asymptotically non-constant-time part of the algorithm (the "core": passes over arrays/lists, scans,
    sorting/merging, etc.) must be written with:
  - mutable variables (`let mut ...`),
  - loops (`while` / `for`),
  - explicit step-by-step updates in loop bodies.
- This core must **not** be implemented via higher-order functional traversals or library calls that hide such
    traversals.

### 2. What may and may not be functional in the implementation

In the actual implementation (`do` block, loop bodies, and helpers called from there):

- Functional style is allowed only for **O(1)** primitive operations:
  - arithmetic, comparisons,
  - basic tuple operations,
  - a single indexing operation (e.g. `a[i]!`),
  - simple conditionals.
- Functional style is **not** allowed for any **non-constant-time** work, including:
  - traversals, scans, cumulative computations,
  - sorting, merging, partitioning or reversing collections when used as part of the core computation,
  - higher-order traversals (`map`, `fold`, `filter`, `scan`, `any`, `all`, etc., on `List`, `Array`, or similar),
  - recursive helper functions that traverse data (e.g. recursing over a list to compute sums, maxima, merges,
    etc.) when this contributes to the main asymptotic cost.

If a computation conceptually iterates over or reorganizes data, it must be written with explicit loops and
    mutation.

### 3. No outsourcing the core to functional helpers

You must not take a conceptually expensive computation (e.g. "merge intervals", "compute prefix/suffix info",
    "sum coverage", "sort data") and:

- implement it via a `map`/`fold`/`sort`/`reverse`/recursive traversal, and
- call that from a thin imperative wrapper.

Any helper that contributes significantly to the algorithm's time complexity must itself follow the same
    imperative style (loops + mutable state) and the same restrictions.

### 4. Invariants and ghost code

- Loop invariants, ghost variables, and other **verification-only** code may use functional style (`map`, `fold`,
    `range`, `sort`, etc.) to **describe** what the imperative algorithm has achieved so far.
  - Example: an invariant like "`sum` equals the fold over the visited prefix" is allowed, even if `sum` is
    maintained by a loop.
- Such ghost code must **not** implement the algorithm in disguise; it may only characterize or relate the
    imperative state to the specification, not perform the actual computation.

---
```

```
## Your Task

You must now act as this verification assistant. Given a specific problem, the goal is to produce a complete
    Velvet `.lean` file containing:
1.  The necessary imports and options.
2.  Any required `solverHint` attributes.
3.  The full `method` definition with appropriate `require`, `ensures`, and `invariant` clauses.
4.  The final `prove_correct ... by loom_solve` block that successfully verifies the program, using additional
    theorems and proof tactic code as needed.
5.  Comments explaining your choice of complex invariants.
```