

## REFERENCES

- 486  
487  
488 Riyaz Ahuja, Jeremy Avigad, Prasad Tetali, and Sean Welleck. Improver: Agent-based automated  
489 proof optimization. *arXiv preprint arXiv:2410.04753*, 2024.
- 490  
491 Rajeev Alur, Alessandro Annichini, Sanjit A. Seshia, et al. Modeling and verification of a dual-  
492 chamber implantable cardiac pacemaker. In *International Conference on Tools and Algorithms for*  
493 *the Construction and Analysis of Systems (TACAS)*, 2010. URL <https://www.cis.upenn.edu/~alur/Tacas12.pdf>.
- 494  
495 Leni Aniva, Chuyue Sun, Brando Miranda, Clark Barrett, and Sanmi Koyejo. Pantograph: A machine-  
496 to-machine interaction interface for advanced theorem proving, high level reasoning, and data  
497 extraction in lean 4. In *International Conference on Tools and Algorithms for the Construction*  
498 *and Analysis of Systems*, pp. 104–123. Springer, 2025.
- 499  
500 Jialun Cao, Yaojie Lu, Meiziniu Li, Haoyang Ma, Haokun Li, Mengda He, Cheng Wen, Le Sun,  
501 Hongyu Zhang, Shengchao Qin, et al. From informal to formal—incorporating and evaluating llms  
502 on natural language requirements to verifiable formal proofs. In *ACL*, 2025.
- 503  
504 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared  
505 Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large  
506 language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- 507  
508 Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next autodiff: Generative  
509 optimization with rich feedback, execution traces, and llms. *arXiv preprint arXiv:2406.16218*,  
510 2024.
- 511  
512 Thomas Claburn. Ai coding assistants keep spitting out buggy or insecure code, studies  
513 warn. URL [https://www.theregister.com/2022/12/21/ai\\_assistants\\_bad\\_code/](https://www.theregister.com/2022/12/21/ai_assistants_bad_code/). The Register.
- 514  
515 MIT CSAIL. 6.858 computer systems security—lab materials, 2024. URL <https://csslab.csail.mit.edu/6.858/>.
- 516  
517 Quinn Dougherty and Ronak Mehta. Proving the coding interview: A benchmark for formally verified  
518 code generation. *arXiv preprint arXiv:2502.05714*, 2025.
- 519  
520 GitHub Engineering. The ai wave continues to grow on software development teams, 2023. URL  
521 <https://github.blog/news-insights/research/survey-ai-wave-grows/>.  
522 GitHub blog post reporting 97% developer adoption of AI code assistants.
- 523  
524 Tarun Gupta. Python code instructions 18k (alpaca style). [https://huggingface.co/datasets/iamtarun/python\\_code\\_instructions\\_18k\\_alpaca%7D%7D](https://huggingface.co/datasets/iamtarun/python_code_instructions_18k_alpaca%7D%7D), 2023.  
525 Dataset, accessed 2025-06-21.
- 526  
527 Gerard J. Holzmann and Willem H. H. James. Using spin model checking for flight software  
528 verification. Technical Report 20060032122, Jet Propulsion Laboratory, NASA, 2006. URL  
529 <https://ntrs.nasa.gov/citations/20060032122>.
- 530  
531 Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée  
532 Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem  
533 provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022. URL <https://arxiv.org/abs/2210.12283>.
- 534  
535 Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts,  
536 and Matei Zaharia. Demonstrate-search-predict: Composing retrieval and language models for  
537 knowledge-intensive NLP. *arXiv preprint arXiv:2212.14024*, 2022.
- 538  
539 Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vard-  
540 hamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller,  
541 Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into  
542 self-improving pipelines. 2024.

- 540 Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou  
541 Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover: A frontier model for open-source  
542 automated theorem proving, 2025. URL <https://arxiv.org/abs/2502.07640>.
- 543 Ilkka Linnosmaa, Ari Virtanen, and Jukka Karjalainen. Model-based formal verification of safety-  
544 critical i&c logic in the finnish nuclear industry. In *NPIC & HMIT*, 2023. URL [https://](https://safer2028.fi/wp-content/uploads/LinnosmaaNPICHMIT2023.pdf)  
545 [safer2028.fi/wp-content/uploads/LinnosmaaNPICHMIT2023.pdf](https://safer2028.fi/wp-content/uploads/LinnosmaaNPICHMIT2023.pdf).
- 546 Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish  
547 Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. Dafnybench: A benchmark for  
548 formal software verification. *arXiv preprint arXiv:2406.08467*, 2024.
- 549 Anh Tuan Nguyen, Tien N. Nguyen, Sarah Nadi, and Weiyi Shang. An empirical evaluation of  
550 github copilot’s code suggestions. In *19th IEEE/ACM International Conference on Mining Soft-*  
551 *ware Repositories (MSR)*, 2022. URL [https://sarahnadi.org/assets/pdf/pubs/](https://sarahnadi.org/assets/pdf/pubs/NguyenMSR22.pdf)  
552 [NguyenMSR22.pdf](https://sarahnadi.org/assets/pdf/pubs/NguyenMSR22.pdf). MSR ’22.
- 553 Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia  
554 Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*,  
555 2025.
- 556 Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri.  
557 Asleep at the keyboard? assessing the security of github copilot’s code contributions. In *43rd*  
558 *IEEE Symposium on Security and Privacy (S&P)*, 2022. doi: 10.48550/arXiv.2108.09293. URL  
559 <https://arxiv.org/abs/2108.09293>.
- 560 Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure  
561 code with ai assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer*  
562 *and Communications Security, CCS ’23*, pp. 2785–2799. ACM, November 2023. doi: 10.1145/  
563 3576915.3623157. URL <http://dx.doi.org/10.1145/3576915.3623157>.
- 564 Alastair F. Reid. End-to-end verification of arm® processors with ISA-Formal. In *Computer*  
565 *Aided Verification (CAV)*, 2016. URL [https://alastairreid.github.io/papers/](https://alastairreid.github.io/papers/cav2016_isa_formal.pdf)  
566 [cav2016\\_isa\\_formal.pdf](https://alastairreid.github.io/papers/cav2016_isa_formal.pdf).
- 567 Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanxia Zhao, Liyue Zhang,  
568 Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu,  
569 Wenjun Gao, Daya Guo, and Chong Ruan. Deepseek-prover-v2: Advancing formal mathematical  
570 reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*,  
571 2025. URL <https://arxiv.org/abs/2504.21801>.
- 572 Kalahasti Ganesh Srivatsa, Sabyasachi Mukhopadhyay, Ganesh Katrapati, and Manish Shrivastava.  
573 A survey of using large language models for generating infrastructure as code, 2024. URL  
574 <https://arxiv.org/abs/2404.00227>.
- 575 Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Clo sed-loop ver ifiable code  
576 generation. In *International Symposium on AI Verification*, pp. 134–155. Springer, 2024.
- 577 Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais,  
578 and Giuliano Antoniol. Bugs in large language models generated code: An empirical study, 2024.  
579 URL <https://arxiv.org/abs/2403.08937>.
- 580 Halborn Security Research Team. A guide to formal verification of smart  
581 contracts, 2023. URL [https://www.halborn.com/blog/post/](https://www.halborn.com/blog/post/a-guide-to-formal-verification-of-smart-contracts)  
582 [a-guide-to-formal-verification-of-smart-contracts](https://www.halborn.com/blog/post/a-guide-to-formal-verification-of-smart-contracts). Blog post.
- 583 Pillar Security Research Team. New vulnerability in github copilot and cursor: How hack-  
584 ers can weaponize code agents, 2025. URL [https://www.pillar.security/blog/](https://www.pillar.security/blog/new-vulnerability-in-github-copilot-and-cursor-how-hackers-can-weaponize-code-agents)  
585 [new-vulnerability-in-github-copilot-and-cursor-how-hackers-can-weaponize-code-agents](https://www.pillar.security/blog/new-vulnerability-in-github-copilot-and-cursor-how-hackers-can-weaponize-code-agents)  
586 Blog post.
- 587 Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzche,  
588 Greg Durrett, Yisong Yue, and Swarat Chaudhuri. Clever: A curated benchmark for formally  
589 verified code generation. *arXiv preprint arXiv:2505.13938*, 2025.

594 Ruida Wang, Jipeng Zhang, Yizhen Jia, Rui Pan, Shizhe Diao, Renjie Pi, and Tong Zhang.  
 595 TheoremLlama: Transforming general-purpose llms into lean4 experts, 2024. URL <https://arxiv.org/abs/2407.03203>.  
 596  
 597

598 Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu,  
 599 Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F.  
 600 Wu, Fuli Luo, and Chong Ruan. Deepseek-prover-v1.5: Harnessing proof assistant feedback for  
 601 reinforcement learning and monte-carlo tree search. *arXiv preprint arXiv:2408.08152*, 2024. URL  
 602 <https://arxiv.org/abs/2408.08152>.  
 603

604 Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina:  
 605 Benchmarking verifiable code generation, 2025. URL [https://arxiv.org/abs/2505.](https://arxiv.org/abs/2505.23135)  
 606 [23135](https://arxiv.org/abs/2505.23135).  
 607

608  
 609 Jin Peng Zhou, Sébastien MR Arnold, Nan Ding, Kilian Q Weinberger, Nan Hua, and Fei Sha.  
 610 Graders should cheat: privileged information enables expert-level automated evaluations. *arXiv*  
 611 *preprint arXiv:2502.10961*, 2025.  
 612

## 613 A RELATED WORK (CONT.)

614  
 615  
 616 **Autoformalization.** Complementing this line of work, Wang et al. (2024) present THEOREML-  
 617 LAMA, a framework aimed at enhancing LLM translation into Lean 4. Drawing on over 100K proof  
 618 examples from the Mathlib4 library, TheoremLlama employs a novel natural-language-to-formal-  
 619 language (NL-FL) bootstrapping strategy and iterative proof synthesis. This enables the reuse of  
 620 verified examples as templates for future translations. The framework achieves 36.48% and 33.61%  
 621 accuracy on the MiniF2F-Valid and MiniF2F-Test benchmarks, respectively—surpassing GPT-4 by  
 622 more than ten percentage points on both.

623 **Techniques for Code Verification.** IMPROVER (Ahuja et al., 2024) introduces a Lean-aware Chain-  
 624 of-States prompting loop that integrates retrieval, best-of- $n$  sampling, and iterative correction to  
 625 rewrite formal proofs with improved properties. By optimizing for metrics such as brevity and  
 626 readability, ImProver reduces the number of tactics by half, doubles proof readability, and boosts  
 627 theorem prover acceptance rates by over 80%. In addition, CLOVER (Sun et al., 2024) implements  
 628 a closed-loop pipeline in which an LLM first generates code, docstrings, and formal annotations,  
 629 then uses reconstruction-based prompting to enforce consistency across these outputs, and finally  
 630 applies SMT-based verification to validate correctness. Evaluated on the CloverBench suite of Dafny  
 631 programs, Clover accepts 87% of correct solutions, rejects 100% of flawed ones, and even uncovers  
 632 bugs in human-written code—demonstrating the power of hybrid generation-verification pipelines.  
 633 In a complementary direction, Zhou et al. (2025) improve general-purpose LLM-based graders by  
 634 augmenting them with “privileged” information such as gold-standard solutions, grading rubrics,  
 635 and detailed annotations. When necessary, the system provides targeted hints back to candidate  
 636 models. This approach achieves grading performance on par with or exceeding that of specialized  
 637 systems—and even expert humans—on difficult programming benchmarks.

638 **Agentic Frameworks and Tools for Code Verification.** TRACE (Cheng et al., 2024) proposes  
 639 *generative optimization*, tuning entire computational workflows—including code, prompts, tool  
 640 calls, and error signals—by treating execution traces as gradients in the OPTO framework. With  
 641 a PyTorch-like API and the LLM-based optimizer OptoPrime, it supports diverse tasks such as  
 642 prompt tuning, debugging, and robot control, rivaling specialized optimizers. Building on modular  
 643 composition, DSPY (Khatab et al., 2024) treats LLM calls as declarative modules in a computational  
 644 graph. Users define concise input–output signatures, and DSPy’s compiler automatically bootstraps  
 645 or fine-tunes pipelines using built-in “teleprompters.” This enables a few-line programs to outperform  
 646 expert-crafted prompts in math, QA, and agent workflows. Extending agentic capabilities to formal  
 647 reasoning, PANTOGRAPH (Aniva et al., 2025) offers a programmatic interface to Lean 4 with support  
 for advanced proof search. It exposes internal proof states and tactics for integration with learning  
 agents, replacing human-facing interfaces with API-level control.

## B DISCUSSION (CONT.)

**Why Lean 4?** Lean 4 is a full fledged programming language and lets VeriBench contain full runnable code and machine-checked proofs in the same dependently-typed language. Ahead-of-time compilation with the Lake toolchain produces fast native binaries, while first-class `Task` primitives, async I/O, and a thread-pool scheduler enable genuine concurrent programs inside the prover. Lean’s C-level foreign-function interface (FFI) lets those binaries call out to high-performance libraries when needed. On the proof side, Lean ships a powerful metaprogramming system written in Lean itself, giving researchers a programming interface to access its internals. This led to the creation of tools such as Pantograph and Aesop. Finally, the community-maintained `mathlib4` and `std4` libraries supply thousands of reusable theorems and data structures, and they are expanding quickly thanks to an active Zulip and GitHub ecosystem. Lean FRO also has plans to create libraries for verifying monadic programs. Lean 4 is backed by an unusually vibrant open-source community: hundreds of contributors refine `mathlib4` on GitHub each week, the public Zulip sees expert discussion around the clock, and even Fields-Medalist Terence Tao has chosen Lean to formalise portions of his current research—clear testimony to the ecosystem’s accessibility and intellectual depth. Unlike the unit tests, fuzzers, and static-analysis pipelines common in industry—tools that sample inputs or rely on heuristics—Lean 4 supplies machine-checked proofs that a property holds for *all* executions. Its dependent type system can encode deep invariants (e.g., length-indexed arrays, bounded integers), so programs that violate them fail to compile, eliminating whole classes of bugs such as buffer overflows or integer wrap-around. Code, specification, and proof reside in the same file and are compiled by Lake into the shipped native binary, preventing the drift that arises when verification artifacts live outside the build. In short, Lean turns informal “best-effort” checks into formal, end-to-end guarantees without sacrificing performance or interoperability.

**Lean 4’s limitations.** Lean’s toolchain is still younger than Coq’s or Isabelle’s, making its standard libraries and automation smaller, and thus some formalizations demand extra groundwork. While Lake delivers native executables, Lean’s runtime has not been stress-tested at the scale of mainstream systems languages, meaning large-scale or safety-critical deployments may require additional vetting. Acknowledging these gaps clarifies that VeriBench chooses Lean 4 for its unique unified programming-plus-proving model and modern automation hooks, not because it already matches the decades-old industrial maturity of older theorem provers. *Static vs. runtime caveat:* Lean’s guarantees are *static* and specification-relative: they certify the Lean program meets the stated properties, but they do not by themselves detect emergent runtime faults outside the model (e.g., I/O (Input/Output) failures, environment misconfiguration, FFI (Foreign Function Interface) unsoundness, resource exhaustion, or undefined behavior in linked C code). Such behaviors must either be modeled and proved, or mitigated with production safeguards (monitoring, sandboxing, input validation).

**Lean 4 versus Dafny.** Unlike Dafny, whose verifier translates each program into the Boogie intermediate language and then discharges first-order verification conditions with an SMT solver such as Z3, Lean 4 reasons natively in a dependently-typed calculus. Because Lean 4’s types can mention run-time data and the very same source code is ahead-of-time compiled to a native binary via Lake, we can both state and prove value-indexed, higher-order properties (e.g., length-indexed vectors) and run the verified program itself—an end-to-end, fidelity that Dafny’s SMT-centred, Boogie-to-Z3 workflow cannot natively match. This gives Lean the expressive power to specify and prove higher-order, data-dependent properties—precisely the kind of semantic guarantees VeriBench seeks to test—while still yielding runnable binaries compiled by the same toolchain. Dafny’s SMT-centric workflow offers impressive push-button automation for imperative code but cannot natively encode the richer specifications (e.g., length-indexed vectors, algebraic invariants) that Lean handles directly.

**Lean 4 versus TLA+.** TLA+ excels at high-level specification of concurrent and distributed protocols, with correctness checked by the TLC model-checker and the TLAPS proof system that dispatches first-order obligations to external provers. However, TLA+ specifications are not executable programs; a separate implementation step is required, and state-space explosion can limit model-checking scalability. VeriBench instead needs a prover where the specification, proof, and runnable code live in the same language. Lean 4’s dependently-typed core lets us capture fine-grained, data-dependent invariants and then compile the very same artifacts to fast native binaries—capabilities outside TLA+’s scope.

Model	VeriBench	Alpaca
LLaMA 3.1–8B (DSPy)	0/8	1/40
LLaMA 70B (DSPy)	0/8	2/40
WattAI 70B (DSPy)	0/8	5/40
Claude 3.5 Sonnet v1 (DSPy)	8/8	24/40
Claude 3.5 Sonnet v2 (DSPy)	4/8	1/40
Claude 3.7 Sonnet (DSPy)	1/8	7/40
Claude 3.5 Sonnet v1 (Baseline)	8/116	22/40
Claude 3.5 + Self-Debug + Judge (Trace)	42/116	—

Table 4: Number of problems resolved by language models on Mini-VeriBench and Alpaca (Gupta, 2023) benchmarks.

Model	HumanEval	EasySet	CSSet	SecuritySet	Average
Baseline	4/51	3/41	0/10	1/11	8/113
Trace v1	31/51	30/41	3/10	4/11	67/113
Trace v2	8/51	24/41	1/10	7/11	40/113

Table 5: Performance on VeriBench (in %). For Trace, we used Claude 3.5 v1. For the baseline, we called Claude 3.5 v1 without access to the compiler and LLM judge feedback. "Trace v1" means just self-debug by the language models, while "Trace v2" indicates self-debug plus LLM-judge.

## C VERIBENCH-V0 RESULTS

### C.1 MINI-VERIBENCH RESULTS

Table 4 We evaluate a range of prominent language models on VeriBench, including both open-source and proprietary systems. Specifically, our evaluation covers LLaMA 3.1–8B, LLaMA 70B, WattAI (LLaMA 70B trained for tool use), Claude 3.5 Sonnet, Claude 3.5 Sonnet (v1 and v2), and Claude 3.7. This diverse selection provides a balanced view of current model capabilities in code translation to formal code under different inference configurations.

#### C.1.1 MAIN RESULTS VERIBENCH-V0.1

Table 4 summarizes the compilation success across multiple settings and models, including Trace-agents variants. These results highlight the challenges of formal code verification via LLMs and the benefits of reward-guided, self-optimizing agent architectures.

Despite comprising just 8 HumanEval translations, current frontier models demonstrate significant limitations: Claude 3.7 Sonnet achieves compilation on only 1 of 8 test examples (12.5%), while Claude 3.5 Sonnet achieves compilation on only 4 of 8 tasks (50%), and LLaMA-70B fails to compile any programs in the Lean 4 HumanEval subset, even with 50 feedback-guided attempts. Note on the easier Alpaca Code subset, the WattAI 70B (a LLaMA 70B tool use model) got barely 11 out of 40 compilation successes (27.5%).

Notably, of the approaches evaluated, our experiments reveal that only a self-optimizing agent architecture achieves meaningful compilation rates approaching 90%. VeriBench establishes a rigorous foundation for developing AI systems capable of synthesizing provably correct, bug-free code, thereby advancing the trajectory toward more secure and dependable software infrastructure.

## D FUTURE DIRECTIONS

We envision VeriBench as a launching pad for multilingual code verification. Many HumanEval problems exist in MultiPL-E’s corpus of 47 languages, allowing future extensions from Python to C, C++, Rust, OCaml, and more. This opens the door for evaluating how models generalize verification strategies across language boundaries.

## E WHY VERIBENCH DOES *not* PROMISE A “COMPLETE” THEOREM LIST

### E.1 MOTIVATION

It is tempting to publish VeriBench together with a fixed theorem set  $\Sigma$  and claim:

*“If your tool can derive every statement in  $\Sigma$ , then the analysed program is completely correct.”*

In practice that promise is unattainable once programs are written in a Turing-complete language like Python, C, and Java. The obstacle is not a lack of ingenuity but an established negative result from computability theory.

### E.2 RICE’S THEOREM

[Rice, 1953, informal] Let  $P$  range over all programs in a Turing-complete language, and let  $\mathcal{S}$  be a property that is

1. **semantic**: depends only on the input–output behaviour of  $P$ ,
2. **non-trivial**: is true for some programs and false for others.

Then no algorithm can decide for *every* program whether it satisfies  $\mathcal{S}$ .

Thus, already for the Halting Problem—“does  $P$  terminate on input  $x$ ?”—a universal decision procedure cannot exist; Rice’s theorem extends this impossibility to *any* meaningful behaviour-oriented property.

### E.3 IMPACT ON A “COMPLETE” THEOREM LIST

Assume, for contradiction, that a recursively enumerable and sound collection  $\Sigma$  captured *every* true semantic statement about a reference program  $P$ . Because  $\Sigma$  is enumerable, we could mechanically search its proof stream: the first appearance of either  $\psi$  or  $\neg\psi$  would decide  $\psi$ . For many interesting  $\psi$  (e.g. total termination, sorted output) this would yield a universal algorithm contradicting Rice’s theorem. Therefore any practical, machine-enumerable  $\Sigma$  must remain incomplete—some true statements are unavoidably absent.

### E.4 VERIBENCH DESIGN CHOICE

Rice’s theorem bars us from publishing a *single* theory that is both recursively enumerable and complete for *all* benchmarks. Yet many benchmarks in practice—especially mathematically-defined, total algorithms over finite or structurally decreasing data—*do* admit a fully exhaustive specification. VeriBench therefore pursues **instance-level completeness** by structuring the gold lean 4 reference files in a rigorous manner as stated in Appendix F and following two-step human curation processes.

## F GOLD STANDARD FOR LEAN 4 BENCHMARK FILES

**Purpose.** Each gold file specifies the intended Lean implementation by: (i) showing functional and (optionally) imperative code; (ii) declaring a *Pre-condition* predicate; (iii) stating algebraic/semantic properties as separate theorems; (iv) defining the *Post-condition* as their conjunction; (v) proving a *Correctness* theorem ( $\rightarrow$ ) by combining those theorems; and (vi) if an imperative variant exists, proving an *Equivalence* theorem between functional and imperative code. This design supports partial-credit grading and consistent comparison even under different theorem factorizations.

### Required order.

1. **Implementation.** Functional definition of *Prog*.
2. **Unit tests.** Include edge, positive, and negative cases: – Positive: valid input–output pairs satisfying *Pre* and all properties. – Negative: inputs/claims violating *Pre* or *Post* (e.g., fabricated equalities, out-of-bounds, overflow).

810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863

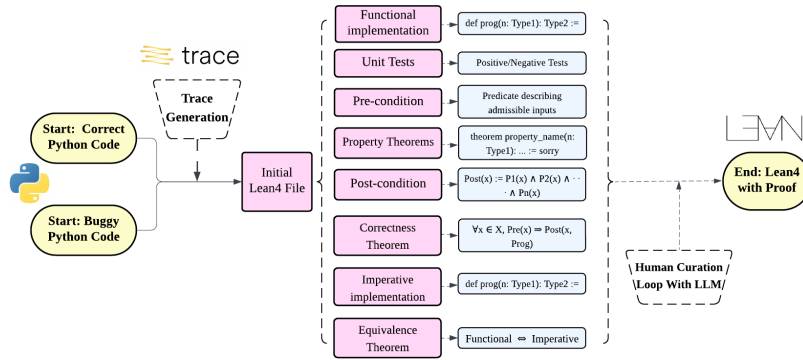


Figure 3: Benchmark Creation Procedure and Structure.

3. **Pre-condition.** Predicate :  $Input \rightarrow$ . Use  $\equiv \text{True}$  if the type already enforces constraints. No proof required.
4. **Properties.** Each property (e.g., identity, commutativity, associativity, distributivity, safety) stated as its own theorem—these are the atomic proof obligations.
5. **Post-condition.** Conjunction of properties, preserving order:
 
$$Post(x) := P_1(x) \wedge P_2(x) \wedge \dots \wedge P_n(x).$$
6. **Correctness.** For all valid  $x$ ,  $(x) \Rightarrow Post(x, Prog)$ . Proof is a direct combination of the property theorems.
7. **Imperative equivalence.** Provide an imperative version, add tests, and prove equivalence with the functional implementation.

## G CONSTRUCTION PIPELINE AND VALIDITY GUARANTEES

A common objection to LLM-generated benchmarks states that the model which creates the data may already have the capability to solve, compromising test integrity and more (?). We address this concern directly.

**Manual audit with public provenance.** After  $\circ 3$  drafts each Lean4 artefact, a curator opens a GitHub issue that is inspected by a 2nd human reviewers, edits the patch when needed, and merges only after the issues resolved (e.g., no comprehensive set of theorems). Every change, comment, and decision is preserved in the repository history, providing reproducible evidence of human oversight.

**Kernel-enforced correctness.** A pull request must compile to be accepted. Because the Lean kernel is a proof checker, compilation implies that every implementation, unit test, theorem, and proof is logically sound. Frontier models struggle to compile at 59% therefore the final tasks necessarily exceed the generator’s capabilities.

**No leakage of final solutions.** Curator edits routinely alter types, theorem statements, or proof strategies—changes the originating LLM cannot anticipate. The published tasks thus differ from the raw LLM output and are not trivially solvable by the same model.

**Benchmark Leaderboard Rankings are Robust to noise in Benchmarks.** Model *rankings* are stable even on noisy datasets (?). Therefore it is known that imperfections would not distort model rankings.

Key benefits of VeriBench Lean Gold File structure.

864 **Key benefits of the structure.**

- 865
- 866
- 867 • **Granular proof evaluation.** Splitting the post-condition into separate theorems provides
- 868 independent proof obligations and partial credit when only a subset is generated or proved.
- 869
- 870
- 871 • **Robust equivalence via a summary.** Even if a generated file formats or partitions prop-
- 872 erties differently from the gold file, its *Correctness* theorem (the conjunction of its own
- 873 property theorems) offers a uniform target. The judge checks that the candidate’s bundled
- 874 statement implies (or is equivalent to) the gold (*Post*), recognizing semantically correct but
- 875 syntactically different decompositions.
- 876
- 877
- 878 • **Clear separation of roles.** *Pre* is a declarative predicate (*Prop*) stating admissible in-
- 879 puts. Each post-condition component is a **theorem** requiring proofs. *Post* is the ordered
- 880 conjunction of these properties; **Correctness** bundles their proofs.
- 881
- 882
- 883 • **Imperative cross-check.** Providing an imperative implementation and an *Equivalence*
- 884 theorem strengthens trust: it guards loop→recursion translation errors, enables testing on a
- 885 realistic stateful version, and lets evaluators accept candidates whose functional code differs
- 886 internally as long as it is provably equivalent to the imperative reference.
- 887

888

889

890 **H STANDARD FOR CORRECT PYTHON INPUT FILES**

891

892

893 **Scope.** Each correct Python file accompanies the Lean gold file with a simple, executable ref-

894 erence. It declares a pre-condition, enforces it *at the start* of the implementation, and includes

895 positive/negative unit tests.

896

897

898

899 **Required structure (in order).**

- 900
- 901
- 902 1. **Docstring.** One–two sentences describing the function’s intent and edge cases.
- 903
- 904
- 905 2. **Python pre-condition.** A pure predicate `pre(...)`  $\rightarrow$  `bool` encoding admissible inputs.
- 906
- 907
- 908 3. **Implementation with early pre-check.** `prog(...)` must call `pre(...)` first and *raise*
- 909 *AssertionError* if it fails; otherwise perform the computation.
- 910
- 911
- 912 4. **Unit tests.** A small suite with:
- 913
- 914
- 915 • *Positive tests*: valid inputs with expected outputs (include edge cases).
- 916
- 917 • *Negative tests*: invalid inputs that must raise `AssertionError`.

## I UNIT TEST ACCURACY

Agent	Eval	Split					Overall
		Easy	CS	Real	HE	Security	
DSPY REACT	Functional	0.366	0.400	0.000	0.393	0.364	0.407
	Imperative	0.268	0.300	0.800	0.393	0.788	0.457
	<b>Average</b>	<b>0.317</b>	<b>0.350</b>	<b>0.400</b>	<b>0.393</b>	<b>0.576</b>	<b>0.432</b>
Trace Baseline	Functional	0.341	0.000	0.000	0.607	0.273	0.429
	Imperative	0.293	0.000	0.800	0.625	0.871	0.543
	<b>Average</b>	<b>0.317</b>	<b>0.000</b>	<b>0.400</b>	<b>0.616</b>	<b>0.572</b>	<b>0.486</b>
TRACE+ (Self-Debug)	Functional	0.756	0.300	0.200	0.589	0.318	0.586
	Imperative	0.659	0.300	0.800	0.607	0.955	0.671
	<b>Average</b>	<b>0.707</b>	<b>0.300</b>	<b>0.500</b>	<b>0.598</b>	<b>0.637</b>	<b>0.629</b>
TRACE++ (Self-Improve)	Functional	0.610	0.200	0.000	0.536	0.364	0.521
	Imperative	0.537	0.200	0.800	0.571	0.955	0.614
	<b>Average</b>	<b>0.573</b>	<b>0.200</b>	<b>0.400</b>	<b>0.554</b>	<b>0.659</b>	<b>0.568</b>

Table 6: Consolidated unit test accuracy on VERIBENCH. Rows group agents, and within each agent, we report *Functional*, *Imperative*, and their *Average*. Columns correspond to benchmark splits: Easy Set (Easy), CS Set (CS), Real Python Code (Real), HumanEval Set (HE), and Security Set (Security). The rightmost column reports the overall mean across splits for each evaluation type.

## J APPENDIX: JUDGING METHODS

Our evaluation pipeline has three simple stages: (1) *extract* the intended solution function(s) from each generated Lean file; (2) *align* VeriBench’s unit tests to those functions by renaming calls as needed; and (3) *verify* with the Lean compiler whether each test passes. We implement two extraction strategies: a lightweight heuristic parser and a more robust LLM-based extractor. We stress-test both approaches on original files, on versions with obfuscated identifiers, and on versions with small induced errors. Across these settings, the LLM-based judge is more reliable and roughly  $5\times$  faster to stand up in practice. The result is a scalable, black-box evaluation of agentic program synthesis that focuses on behavior—not just compilability—via unit-test accuracy.

## K MORE SIMPLIFIED BENCHMARK EXAMPLES

In this section we provide more simplified examples from VERIBENCH.

```

# Implementation
def my_max(a: int, b: int) -> int:
    """
    Return the larger of two non-negative integers.
    """
    return b if a <= b else a

# Tests
from typing import Callable

def check(candidate: Callable[[int, int], int]) -> bool:
    assert candidate(7, 3) == 7, f"expected 7 from (7,3) but got {candidate(7, 3)}"
    ... (other tests) ...
    return True

if __name__ == "__main__":
    assert check(my_max), f"Failed: {__file__}"
    print(f'All tests passed: {__file__}!')
```

Listing 3: An exemplar input Python code of VeriBench-EasySet (simplified for showcase).

```
namespace MyMax
```

```

972
973 -- Implementation
974 def myMax (a b : Nat) : Nat :=
975   if _ : a ≤ b then b else a
976
977 infixl:70 "⊔" => myMax -- left-associative, precedence 70
978
979 -- Tests
980 #eval myMax 7 3 -- expect 7
981 example : myMax 7 3 = 7 := by native_decide
982 #eval myMax 0 0 -- expect 0
983 example : myMax 0 0 = 0 := by native_decide
984
985 -- Theorems
986 @[simp] theorem max_left_identity (n : Nat) : myMax 0 n = n := sorry
987
988 -- Theorem Right Identity
989 @[simp] theorem max_right_identity (n : Nat) : myMax n 0 = n := sorry
990
991 ... (other theorems) ...
992
993 end MyMax

```

Listing 4: An exemplar golden output Lean 4 code of VeriBench-EasySet.

```

990 # Implementation
991 from typing import List, Optional
992 def binary_search(arr: List[int], target: int) -> Optional[int]:
993     """
994     Binary search implementation that searches for a target value in a sorted list.
995     Returns the index if found, None if not found.
996     """
997     if not arr:
998         return None
999
1000     left, right = 0, len(arr) - 1
1001
1002     while left <= right:
1003         mid = (left + right) // 2
1004         mid_val = arr[mid]
1005
1006         if mid_val == target:
1007             return mid
1008         elif mid_val < target:
1009             left = mid + 1
1010         else:
1011             right = mid - 1
1012
1013     return None
1014
1015 # Tests
1016 from typing import Callable
1017 def check(candidate: Callable[[List[int], int], Optional[int]]) -> bool:
1018     assert candidate([1, 2, 3, 4, 5], 1) == 0
1019     assert candidate([1, 2, 3, 4, 5], 3) == 2
1020     ... (more tests) ...
1021
1022     print("Pass: all correct!")
1023     return True
1024
1025 if __name__ == "__main__":
1026     assert check(binary_search), f"Failed: {__file__}"

```

Listing 5: An exemplar input Python code of VeriBench-CSSet (simplified for showcase).

```

1018 import Mathlib.Data.List.Sort
1019 import Mathlib.Data.List.Basic
1020
1021 namespace BinarySearch
1022 open List
1023
1024 -- Implementation
1025 partial def binarySearchAux (arr : List Nat) (target : Nat) (left right : Nat) : Option Nat :=
1026   if left > right then
1027     none
1028   else
1029     let mid := (left + right) / 2
1030     if mid >= arr.length then

```

```

1026     none
1027   else
1028     let midVal := arr.get <mid, by sorry>
1029     if midVal = target then
1030       some mid
1031     else if midVal < target then
1032       binarySearchAux arr target (mid + 1) right
1033     else
1034       binarySearchAux arr target left (mid - 1)
1035
1036 def binarySearch (arr : List Nat) (target : Nat) : Option Nat :=
1037   if arr.isEmpty then
1038     none
1039   else
1040     binarySearchAux arr target 0 (arr.length - 1)
1041
1042 /-- Linear search for comparison and verification -/
1043 def linearSearch (arr : List Nat) (target : Nat) : Option Nat :=
1044   arr.findIdx? (· = target)
1045
1046 -- Theorem: If binarySearch returns Some i, then arr[i] = target
1047 theorem correctness_binarySearch (arr : List Nat) (target : Nat) (i : Nat) :
1048   binarySearch arr target = some i → arr[i]? = some target := by
1049   sorry
1050
1051 -- Theorem: If target is in the sorted array, then binarySearch finds it
1052 theorem completeness_binarySearch (arr : List Nat) (target : Nat) :
1053   List.Sorted (fun x y => x ≤ y) arr → target ∈ arr →
1054   ∃ i, binarySearch arr target = some i := by
1055   sorry
1056
1057 ... (more theorems) ...
1058
1059 end BinarySearch

```

Listing 6: An exemplar golden output Lean 4 code of VeriBench-CSSet (simplified for showcase).

1051

```

1052 # Implementation
1053 def unsafe_copy(dst: bytearray, src: bytearray) -> None:
1054   """
1055   Copy bytes from 'src' into 'dst' at the same indices, without any bounds checking.
1056   If 'len(src) > len(dst)', this will raise an IndexError (buffer overflow).
1057   """
1058   for i, b in enumerate(src):
1059     dst[i] = b
1060
1061 # Tests
1062 def check(candidate) -> bool:
1063   # 1) Safe copy: src fits in dst
1064   d = bytearray(3)
1065   s = bytearray(b'abc')
1066   candidate(d, s)
1067   assert bytes(d) == b'abc'
1068
1069   ... (other tests) ...
1070
1071   return True
1072
1073 assert check(unsafe_copy), "Candidate failed buffer-overflow tests"
1074 print("Pass!")

```

Listing 7: An exemplar input Python code of VeriBench-SecuritySet (simplified for showcase).

1068

1069

1070

1071

1072

1073

1074

1075

1076

1077

1078

1079

```

namespace BufferOverflow

-- Implementation
def unsafeCopy (dst src : List UInt8) : Option (List UInt8) :=
  let n := dst.length
  -- fold over enumerated bytes with their indices
  src.enum.foldl1 (fun o (i, b) =>
    o.bind fun acc =>
      if h : i < n then
        some (acc.set i b)
      else
        none
  ) (some dst)

-- Tests

```

```

1080 example : unsafeCopy [0, 0, 0] [1,2] = some [1,2,0] := by rfl
1081 example : unsafeCopy [0, 0] [1,2,3] = none := by rfl
1082
1083 ... (other tests) ...
1084
1084 -- Theorem: safety precondition
1085 theorem copy_safe {dst src : List UInt8}
1086   (h : src.length ≤ dst.length) :
1087   ∃newDst, unsafeCopy dst src = some newDst := by
1088     unfold unsafeCopy
1089
1088 -- Theorem: overflow detection
1089 theorem copy_overflow {dst src : List UInt8}
1090   (h : dst.length < src.length) :
1091   unsafeCopy dst src = none := by
1092     unfold unsafeCopy
1093     admit
1094
1093 end BufferOverflow

```

Listing 8: An exemplar golden output Lean 4 code of VeriBench-SecuritySet (simplified for showcase).

```

1097 # -- Implementation --
1098 # source: https://github.com/python/cpython/blob/3.13/Lib/heapq.py
1099
1099 # 'heap' is a heap at all indices >= startpos, except possibly for pos.
1100 # Restore the heap invariant by bubbling the new item up.
1101 def _siftdown(heap, startpos, pos):
1102     newitem = heap[pos]
1103     while pos > startpos:
1104         parentpos = (pos - 1) >> 1
1105         parent = heap[parentpos]
1106         if newitem < parent:
1107             heap[pos] = parent
1108             pos = parentpos
1109             continue
1110         break
1111     heap[pos] = newitem
1112
1113 def heappush(heap, item):
1114     """Push item onto heap, maintaining the heap invariant."""
1115     heap.append(item)
1116     _siftdown(heap, 0, len(heap) - 1)
1117
1118 # -- Tests (simplified) --
1119 from typing import Callable
1120
1121 def _check_invariant(h):
1122     for i in range(1, len(h)):
1123         p = (i - 1) >> 1
1124         assert h[p] <= h[i], "heap invariant violated"
1125
1126 def check(candidate: Callable[[list, int], None]) -> bool:
1127     # Basic sanity
1128     h = []
1129     candidate(h, 3); _check_invariant(h)
1130     candidate(h, 1); _check_invariant(h)
1131     candidate(h, 2); _check_invariant(h)
1132     assert h[0] == 1
1133
1134     # Equal elements
1135     h = []
1136     candidate(h, 5); candidate(h, 5); _check_invariant(h)
1137     assert h[0] == 5
1138
1139     # ... (additional randomized/property tests omitted) ...
1140     return True
1141
1142 assert check(heappush)

```

Listing 9: An exemplar input Python code of VeriBench-RealCode (simplified for showcase).

```

1132 /-!
1133 # VeriBench Heappush (min-heap, simplified)

```

```

1134 Order:
1135 1. Implementation ('_siftdown', 'heappush')
1136 2. Minimal checks
1137 3. Pre-condition
1138 4. Properties
1139 5. Post-condition
1140 6. Correctness
1141 -/
1142 namespace HeapPush
1143
1144 /-- Move the leaf at `pos` up until the min-heap invariant holds. -/
1145 def _siftdown (heap : Array Int) (startpos pos : Nat) : Array Int :=
1146   let newitem := heap[pos]!
1147   let rec loop (h : Array Int) (pos : Nat) : Array Int :=
1148     if pos > startpos then
1149       let parentpos := (pos - 1) >>> 1
1150       let parent := h[parentpos]!
1151       if newitem < parent then
1152         loop (h.set! pos parent) parentpos
1153       else
1154         h.set! pos newitem
1155     loop heap pos
1156
1157 /-- Push an element and restore the heap invariant. -/
1158 def heappush (heap : Array Int) (item : Int) : Array Int :=
1159   let h := heap.push item
1160   _siftdown h 0 (h.size - 1)
1161
1162 /-! ## Minimal checks (others omitted) -/
1163 example : (heappush (heappush (heappush (#[[] : Array Int) 3) 1) 2)[0]! = 1 := by native_decide
1164 example :
1165   let h1 := heappush (#[[] : Array Int) 5
1166   let h2 := heappush h1 5
1167   h1[0]! = 5 ∧ h2[0]! = 5 := by native_decide
1168
1169 /-- Boolean heap invariant: for every child, parent ≤ child. -/
1170 def checkInvariant (h : Array Int) : Bool :=
1171   let n := h.size
1172   let rec go (i : Nat) : Bool :=
1173     if i ≥ n then true else
1174     if i = 0 then go (i+1) else
1175     let p := (i - 1) >>> 1
1176     (h[p]! ≤ h[i]!) && go (i+1)
1177   go 0
1178
1179 /-! ## Pre-condition -/
1180 def Pre (heap : Array Int) : Prop :=
1181   checkInvariant heap = true
1182
1183 /-! ## Properties -/
1184 def prop_invariant (heap : Array Int) (item : Int) : Prop :=
1185   checkInvariant (heappush heap item) = true
1186
1187 def prop_size (heap : Array Int) (item : Int) : Prop :=
1188   (heappush heap item).size = heap.size + 1
1189
1190 def prop_multiset (heap : Array Int) (item : Int) : Prop :=
1191   List.Perm (heappush heap item).toList (item :: heap.toList)
1192
1193 /-! ## Theorems (sketched; proofs omitted) -/
1194 @[simp] theorem invariant_thm (heap : Array Int) (item : Int) (hPre : Pre heap) :
1195   prop_invariant heap item := by
1196   sorry
1197
1198 @[simp] theorem size_thm (heap : Array Int) (item : Int) :
1199   prop_size heap item := by
1200   sorry
1201
1202 @[simp] theorem multiset_thm (heap : Array Int) (item : Int) :
1203   prop_multiset heap item := by
1204   sorry
1205
1206 /-! ## Post-condition -/
1207 def Post_prop (heap : Array Int) (item : Int) : Prop :=
1208   prop_invariant heap item ∧ prop_size heap item ∧ prop_multiset heap item
1209
1210 /-! ## Correctness -/
1211 theorem correctness_thm (heap : Array Int) (item : Int) (hPre : Pre heap) :

```

```

1188 Post_prop heap item := by
1189 exact And.intro
1190   (invariant_thm heap item hPre)
1191   (And.intro (size_thm heap item) (multiset_thm heap item))
1192 end HeapPush

```

Listing 10: An exemplar golden output Lean 4 code of VeriBench-RealCode (simplified for showcase).

## L FULL BENCHMARK EXAMPLES

In this section we provide full examples from VERIBENCH.

```

1200 # -- Implementation --
1201 def my_max(a: int, b: int) -> int:
1202     """
1203     Return the larger of two non-negative integers.
1204
1205     >>> my_max(7, 3)
1206     7
1207     >>> my_max(0, 0)
1208     0
1209     """
1210     return b if a <= b else a
1211
1212 # -- Tests --
1213 from typing import Callable
1214
1215 def check(candidate: Callable[[int, int], int]) -> bool:
1216     print(f'Running tests for {candidate.__name__}...')
1217     # Basic unit tests
1218     assert candidate(7, 3) == 7, f"expected 7 from (7,3) but got {candidate(7, 3)}"
1219     # Edge unit tests
1220     assert candidate(0, 0) == 0, f"expected 0 from (0,0) but got {candidate(0, 0)}"
1221
1222     # Property checks on a small domain
1223     for x in range(6):
1224         # idempotence
1225         assert candidate(x, x) == x, f"idempotence violated for {x}"
1226
1227         for y in range(6):
1228             # commutativity
1229             lhs = candidate(x, y)
1230             rhs = candidate(y, x)
1231             assert lhs == rhs, (
1232                 f"commutativity violated for ({x},{y}): {lhs} != {rhs}"
1233             )
1234
1235             # upper-bound property
1236             assert x <= candidate(x, y), f"left bound violated for ({x},{y})"
1237             assert y <= candidate(x, y), f"right bound violated for ({x},{y})"
1238
1239             for z in range(6):
1240                 # associativity
1241                 left_assoc = candidate(candidate(x, y), z)
1242                 right_assoc = candidate(x, candidate(y, z))
1243                 assert left_assoc == right_assoc, (
1244                     f"associativity violated for "
1245                     f"({x},{y},{z}): {left_assoc} != {right_assoc}"
1246                 )
1247
1248         return True
1249
1250 if __name__ == "__main__":
1251     assert check(my_max), f"Failed: {__file__}"
1252     print(f'All tests passed: {__file__}!')

```

Listing 11: An exemplar input Python code of VeriBench-EasySet.

```

1238 /-!
1239 # Implementation
1240 -/
1241
1242 namespace MyMax
1243 /--

```

```

1242 Implementation of my custom maximum function.
1243
1244 `myMax a b` returns the larger of two natural numbers.
1245
1246 ## Examples
1247 #eval myMax 7 3 -- expect 7
1248 #eval myMax 0 0 -- expect 0
1249 -/
1250 def myMax (a b : Nat) : Nat :=
1251   if _ : a ≤ b then b else a
1252
1253 infixl:70 "⌈" => myMax -- left-associative, precedence 70
1254
1255 /-!
1256 # Tests
1257 We use `#eval` to print results, then nameless `example` to confirm correctness
1258 (especially in cases where `native_decide` is used to prove the example).
1259 -/
1260
1261 -- Functional tests
1262 #eval myMax 7 3 -- expect 7
1263 example : myMax 7 3 = 7 := by native_decide
1264
1265 #eval myMax 0 0 -- expect 0
1266 example : myMax 0 0 = 0 := by native_decide
1267
1268 /-!
1269 # Theorems
1270 -/
1271
1272 /-- Theorem Left Identity: Taking max with zero on the left acts as the identity. -/
1273 @[simp] theorem max_left_identity (n : Nat) : myMax 0 n = n := sorry
1274
1275 /-- Theorem Right Identity: Taking max with zero on the right acts as the identity. -/
1276 @[simp] theorem max_right_identity (n : Nat) : myMax n 0 = n := sorry
1277
1278 /-- Theorem Commutativity: The order of the arguments does not affect the maximum. -/
1279 @[simp] theorem max_commutativity (a b : Nat) : myMax a b = myMax b a := sorry
1280
1281 /-- Theorem Idempotence: Taking max of a number with itself yields that number. -/
1282 @[simp] theorem max_idempotent (a : Nat) : myMax a a = a := sorry
1283
1284 /-- Theorem Left Bound: The first argument never exceeds the maximum. -/
1285 theorem max_left_bound (a b : Nat) : a ≤ myMax a b := sorry
1286
1287 /-- Theorem Right Bound: The second argument never exceeds the maximum. -/
1288 theorem max_right_bound (a b : Nat) : b ≤ myMax a b := sorry
1289
1290 /--
1291 Imperative implementation of `myMax`.
1292
1293 `myMaxImp a b` computes the same maximum using mutable state:
1294 start with `m := a`, then overwrite with `b` if `b` is larger.
1295
1296 ## Examples
1297 #eval myMaxImp 7 3 -- expect 7
1298 #eval myMaxImp 0 0 -- expect 0
1299 -/
1300 def myMaxImp (a b : Nat) : Nat :=
1301   Id.run do
1302     let mut m : Nat := a
1303     for x in [a, b] do -- loop over both inputs
1304       if m ≤ x then
1305         m := x
1306     return m
1307
1308 -- Imperative tests
1309 #eval myMaxImp 7 3 -- expect 7
1310 example : myMaxImp 7 3 = 7 := by native_decide
1311
1312 #eval myMaxImp 0 0 -- expect 0
1313 example : myMaxImp 0 0 = 0 := by native_decide
1314
1315 /--
1316 Theorem Equivalence of Functional and Imperative Maximum:
1317 Both implementations produce identical results for all inputs.
1318 -/
1319 theorem myMax_equiv_functional_imperative (a b : Nat) :
1320   myMax a b = myMaxImp a b := sorry

```

```
1296
1297 end MyMax
```

Listing 12: An exemplar golden output Lean 4 code of VeriBench-EasySet.

```
1300 # -- Implementation --
1301 from typing import List, Optional
1302
1303 def binary_search(arr: List[int], target: int) -> Optional[int]:
1304     """
1305     Binary search implementation that searches for a target value in a sorted list.
1306     Returns the index if found, None if not found.
1307
1308     >>> binary_search([1, 2, 3, 4, 5], 3)
1309     2
1310     >>> binary_search([1, 2, 3, 4, 5], 6)
1311
1312     >>> binary_search([], 1)
1313
1314     """
1315     if not arr:
1316         return None
1317
1318     left, right = 0, len(arr) - 1
1319
1320     while left <= right:
1321         mid = (left + right) // 2
1322         mid_val = arr[mid]
1323
1324         if mid_val == target:
1325             return mid
1326         elif mid_val < target:
1327             left = mid + 1
1328         else:
1329             right = mid - 1
1330
1331     return None
1332
1333 # -- Tests --
1334 from typing import Callable
1335 def check(candidate: Callable[[List[int], int], Optional[int]]) -> bool:
1336     # Basic functionality tests
1337     assert candidate([1, 2, 3, 4, 5], 1) == 0
1338     assert candidate([1, 2, 3, 4, 5], 3) == 2
1339     assert candidate([1, 2, 3, 4, 5], 5) == 4
1340     assert candidate([1, 2, 3, 4, 5], 6) is None
1341     assert candidate([1, 2, 3, 4, 5], 0) is None
1342
1343     # Edge cases
1344     assert candidate([], 1) is None
1345     assert candidate([5], 5) == 0
1346     assert candidate([5], 3) is None
1347
1348     # Larger arrays
1349     assert candidate([1, 3, 5, 7, 9], 3) == 1
1350     assert candidate([1, 3, 5, 7, 9], 7) == 3
1351     assert candidate([1, 3, 5, 7, 9], 4) is None
1352     assert candidate([10, 20, 30, 40, 50, 60], 60) == 5
1353     assert candidate([10, 20, 30, 40, 50, 60], 10) == 0
1354
1355     # Test with duplicates (binary search may return any valid index)
1356     test_arr = [1, 2, 3, 3, 3, 4, 5]
1357     result = candidate(test_arr, 3)
1358     assert result is not None and test_arr[result] == 3 and 2 <= result <= 4
1359
1360     # Large sorted array test
1361     large_arr = list(range(100))
1362     assert candidate(large_arr, 49) == 49
1363     assert candidate(large_arr, 99) == 99
1364     assert candidate(large_arr, 100) is None
1365
1366     # Two element arrays
1367     assert candidate([1, 2], 1) == 0
1368     assert candidate([1, 2], 2) == 1
1369     assert candidate([1, 2], 3) is None
1370
1371     print("Pass: all correct!")
1372     return True
1373
1374 if __name__ == "__main__":
```

```
1350     assert check(binary_search), f"Failed: {__file__}"
```

Listing 13: An exemplar input Python code of VeriBench-SSet.

```
1354 import Mathlib.Data.List.Sort
1355 import Mathlib.Data.List.Basic
1356
1357 /-!
1358 # Implementation
1359 -/
1360 namespace BinarySearch
1361 open List
1362
1363 /-- Binary search implementation using recursive approach with bounds -/
1364 partial def binarySearchAux (arr : List Nat) (target : Nat) (left right : Nat) : Option Nat :=
1365   if left > right then
1366     none
1367   else
1368     let mid := (left + right) / 2
1369     if mid >= arr.length then
1370       none
1371     else
1372       let midVal := arr.get <mid, by sorry>
1373       if midVal = target then
1374         some mid
1375       else if midVal < target then
1376         binarySearchAux arr target (mid + 1) right
1377       else
1378         binarySearchAux arr target left (mid - 1)
1379
1380 /-- Binary search that searches for a target value in a sorted list.
1381 Returns Some index if found, None if not found. -/
1382 def binarySearch (arr : List Nat) (target : Nat) : Option Nat :=
1383   if arr.isEmpty then
1384     none
1385   else
1386     binarySearchAux arr target 0 (arr.length - 1)
1387
1388 /-- Linear search for comparison and verification -/
1389 def linearSearch (arr : List Nat) (target : Nat) : Option Nat :=
1390   arr.findIdx? (· = target)
1391
1392 /-!
1393 # Theorems
1394 -/
1395 /--
1396 **Correctness**: If `binarySearch` returns `Some i`, then `arr[i] = target`.
1397 -/
1398 theorem correctness_binarySearch (arr : List Nat) (target : Nat) (i : Nat) :
1399   binarySearch arr target = some i → arr[i]? = some target := by
1400   sorry
1401
1402 /--
1403 **Completeness**: If `target` is in the sorted array, then `binarySearch` finds it.
1404 -/
1405 theorem completeness_binarySearch (arr : List Nat) (target : Nat) :
1406   List.Sorted (fun x y => x ≤ y) arr → target ∈ arr →
1407   ∃ i, binarySearch arr target = some i := by
1408   sorry
1409
1410 /--
1411 **Valid Index**: If `binarySearch` returns `Some i`, then `i` is a valid index.
1412 -/
1413 theorem valid_index_binarySearch (arr : List Nat) (target : Nat) (i : Nat) :
1414   binarySearch arr target = some i → i < arr.length := by
1415   sorry
1416
1417 /--
1418 **Not Found**: If `binarySearch` returns `None`, then `target` is not in the array
1419 (assuming the array is sorted).
1420 -/
1421 theorem not_found_binarySearch (arr : List Nat) (target : Nat) :
1422   List.Sorted (fun x y => x ≤ y) arr →
1423   binarySearch arr target = none → target ∉ arr := by
1424   sorry
```

```

1404 /--
1405 **Equivalence with Linear Search**: On sorted arrays, binary search and linear search
1406 find the same elements (though possibly different indices for duplicates).
1407 -/
1408 theorem equiv_linearSearch_binarySearch (arr : List Nat) (target : Nat) :
1409   List.Sorted (fun x y => x ≤ y) arr →
1410   (binarySearch arr target).isSome ↔ (linearSearch arr target).isSome := by
1411     sorry
1412 /-!
1413 # Imperative Tests
1414 -/
1415 /-- expected: some 0 -/
1416 example : binarySearch [1, 2, 3, 4, 5] 1 = some 0 := by native_decide
1417 #eval binarySearch [1, 2, 3, 4, 5] 1 -- expected: some 0
1418 /-- expected: some 2 -/
1419 example : binarySearch [1, 2, 3, 4, 5] 3 = some 2 := by native_decide
1420 #eval binarySearch [1, 2, 3, 4, 5] 3 -- expected: some 2
1421 /-- expected: some 4 -/
1422 example : binarySearch [1, 2, 3, 4, 5] 5 = some 4 := by native_decide
1423 #eval binarySearch [1, 2, 3, 4, 5] 5 -- expected: some 4
1424 /-- expected: none -/
1425 example : binarySearch [1, 2, 3, 4, 5] 6 = none := by native_decide
1426 #eval binarySearch [1, 2, 3, 4, 5] 6 -- expected: none
1427 /-- expected: none -/
1428 example : binarySearch [] 1 = none := by native_decide
1429 #eval binarySearch [] 1 -- expected: none
1430 /-- expected: some 0 -/
1431 example : binarySearch [5] 5 = some 0 := by native_decide
1432 #eval binarySearch [5] 5 -- expected: some 0
1433 /-- expected: none -/
1434 example : binarySearch [5] 3 = none := by native_decide
1435 #eval binarySearch [5] 3 -- expected: none
1436 /-- expected: some 1 -/
1437 example : binarySearch [1, 3, 5, 7, 9] 3 = some 1 := by native_decide
1438 #eval binarySearch [1, 3, 5, 7, 9] 3 -- expected: some 1
1439 /-- expected: some 3 -/
1440 example : binarySearch [1, 3, 5, 7, 9] 7 = some 3 := by native_decide
1441 #eval binarySearch [1, 3, 5, 7, 9] 7 -- expected: some 3
1442 /-- expected: none -/
1443 example : binarySearch [1, 3, 5, 7, 9] 4 = none := by native_decide
1444 #eval binarySearch [1, 3, 5, 7, 9] 4 -- expected: none
1445 /-- expected: some 5 -/
1446 example : binarySearch [10, 20, 30, 40, 50, 60] 60 = some 5 := by native_decide
1447 #eval binarySearch [10, 20, 30, 40, 50, 60] 60 -- expected: some 5
1448 /-- expected: some 0 -/
1449 example : binarySearch [10, 20, 30, 40, 50, 60] 10 = some 0 := by native_decide
1450 #eval binarySearch [10, 20, 30, 40, 50, 60] 10 -- expected: some 0
1451 /-- Test with duplicates: expected: some 2 (could be any of the valid indices) -/
1452 example : binarySearch [1, 2, 3, 3, 3, 4, 5] 3 = some 2 := by native_decide
1453 #eval binarySearch [1, 2, 3, 3, 3, 4, 5] 3 -- expected: some 2
1454 /-- Large sorted array test: expected: some 49 -/
1455 example : binarySearch (List.range 100) 49 = some 49 := by native_decide
1456 #eval binarySearch (List.range 100) 49 -- expected: some 49
1457 end BinarySearch

```

Listing 14: An exemplar golden output Lean 4 code of VeriBench-CSSet.

```

# -- Implementation --
from typing import List

```

```

1458
1459 def has_close_elements(numbers: List[float], threshold: float) -> bool:
1460     """
1461     Check if in given list of numbers, are any two numbers closer to each other
1462     than given threshold.
1463     >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
1464     False
1465     >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
1466     True
1467     """
1468     for idx, elem in enumerate(numbers):
1469         for idx2, elem2 in enumerate(numbers):
1470             if idx != idx2:
1471                 distance = abs(elem - elem2)
1472                 if distance < threshold:
1473                     return True
1474     return False
1475
1476 # -- Tests --
1477 from typing import Callable
1478 def check(candidate: Callable[[List[float], float], bool]) -> bool:
1479     # Original tests
1480     assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
1481     assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
1482     assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
1483     assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
1484     assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
1485     assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
1486     assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False
1487
1488     # Additional tests to cover edge/corner cases:
1489
1490     # 1. Empty list -> no pairs, so we expect False.
1491     assert candidate([], 0.1) == False
1492
1493     # 2. Single element -> no pairs to compare, so should be False.
1494     assert candidate([1.5], 0.1) == False
1495
1496     # 3. Two identical elements -> distance = 0 < threshold => True if threshold > 0.
1497     assert candidate([3.14, 3.14], 0.1) == True
1498     # But if threshold == 0, that can't be "closer" than 0:
1499     assert candidate([3.14, 3.14], 0.0) == False
1500
1501     # 4. Large threshold -> any pair is "close" if we have >= 2 elements
1502     # so [100, 200] with threshold=999.9 => True
1503     assert candidate([100, 200], 999.9) == True
1504
1505     # 5. Distinct elements that are still quite close
1506     # e.g. [1.0, 1.000000 1] with threshold=1e-5 => distance=1e-7 < 1e-5 => True
1507     assert candidate([1.0, 1.00000001], 1e-5) == True
1508
1509     # 6. Distinct elements that are not that close
1510     # e.g. [1.0, 1.0002] with threshold=1e-5 => distance=2e-4 => False
1511     assert candidate([1.0, 1.0002], 1e-5) == False
1512
1513     print("Pass: all coorrect!")
1514
1515     return True
1516
1517 if __name__ == "__main__":
1518     assert check(has_close_elements), f"Failed: {__file__}"

```

Listing 15: An exemplar input Python code of VeriBench-HumanEval.

```

1501 /-!
1502 # Implementation
1503
1504 ## Has Close Elements
1505
1506 Implements `hasCloseElements`, which checks whether any two elements of a list
1507 are closer than a threshold, plus an imperative variant `hasCloseElementsImp`
1508 and a collection of small-to-medium theorems that together mimic the
1509 multi-lemma style of real-world code verification.
1510 -/
1511
1512 namespace HasCloseElements
1513 open List -- brings the `` permutation notation into scope
1514
1515 /--
1516 Recursive implementation.

```

```

1512 Returns `true` iff there exist distinct elements in `numbers`
1513 whose absolute difference is less than `threshold`.
1514
1515 ## Examples
1516 #eval hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.3 -- expected: true
1517 #eval hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.05 -- expected: false
1518 #eval hasCloseElements [] 0.1 -- expected: false
1519 -/
1519 def hasCloseElements (numbers : List Float) (threshold : Float) : Bool :=
1520   match numbers with
1521   | [] => false
1522   | x :: xs =>
1523     if xs.any (fun y => Float.abs (x - y) < threshold) then
1524       true
1525     else
1526       hasCloseElements xs threshold
1527
1528 -/!
1529 # Tests
1530 -/
1531 /-- expected: true -/
1532 example : hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.3 = true := by
1533   native_decide
1534 #eval hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.3 -- expected: true
1535
1536 /-- expected: false -/
1537 example : hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.05 = false := by
1538   native_decide
1539 #eval hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.05 -- expected: false
1540
1541 -/!
1542 # Tests: Edge Cases
1543 -/
1544 /-- expected: false -/
1545 example : hasCloseElements [] 0.1 = false := by native_decide
1546 #eval hasCloseElements [] 0.1 -- expected: false
1547
1548 /-- expected: false -/
1549 example : hasCloseElements [42.0] 0.01 = false := by native_decide
1550 #eval hasCloseElements [42.0] 0.01 -- expected: false
1551
1552 -/!
1553 # Theorems
1554 -/
1555 /--
1556 **Specification**: `hasCloseElements numbers t = true`
1557 iff  $\exists$  distinct indices whose elements differ by  $< t$ .
1558 -/
1559 theorem hasCloseElements_iff
1560 (numbers : List Float) (t : Float) :
1561   hasCloseElements numbers t = true  $\leftrightarrow$ 
1562      $\exists i j : \text{Nat},$ 
1563      $i < \text{numbers.length} \wedge j < \text{numbers.length} \wedge$ 
1564      $i \neq j \wedge$ 
1565      $\text{Float.abs} (\text{numbers}[i]! - \text{numbers}[j]!) < t := by$ 
1566     sorry
1567
1568 /--
1569 **Monotone in `threshold`**: enlarging the tolerance preserves truth.
1570
1571 If  $t_1 \leq t_2$  and the predicate is `true` at  $t_1$ ,
1572 then it is also `true` at  $t_2$ .
1573 -/
1574 @[simp] theorem threshold_mono
1575 {numbers : List Float} {t1 t2 : Float}
1576 (hle : t1  $\leq$  t2)
1577 (h : hasCloseElements numbers t1 = true) :
1578   hasCloseElements numbers t2 = true := by
1579     sorry
1580
1581 /--
1582 **Duplicates  $\Rightarrow$  true**:
1583
1584 If the list contains a value appearing twice and `threshold > 0`,
1585 the result is `true` (distance  $0 < \text{threshold}$ ).
1586 -/
1587 @[simp] theorem duplicates_imply_true

```

```

1566     (numbers : List Float) (t : Float)
1567     (hpos : t > 0)
1568     (hdup :  $\exists i j, i < \text{numbers.length} \wedge j < \text{numbers.length} \wedge i \neq j \wedge$ 
1569         numbers[i]! = numbers[j]!) :
1570     hasCloseElements numbers t = true := by
1571     sorry
1572 /--
1573 **Non-positive threshold  $\Rightarrow$  false** :
1574 For `t ≤ 0` no pair can satisfy `|x - y| < t`, so the predicate is `false`.
1575 -/
1576 @[simp] theorem nonpos_threshold_false
1577   (numbers : List Float) (t : Float) (hle : t ≤ 0) :
1578   hasCloseElements numbers t = false := by
1579   sorry
1580 /--
1581 **Empty or singleton list  $\Rightarrow$  false** :
1582 The predicate needs at least two elements to succeed.
1583 -/
1584 @[simp] theorem length_le_one_false
1585   (numbers : List Float) (t : Float)
1586   (hlen : numbers.length ≤ 1) :
1587   hasCloseElements numbers t = false := by
1588   sorry
1589 /--
1590 **True  $\Rightarrow$  length ≥ 2** :
1591 Conversely, if the predicate is `true`, the list must have at least two elements.
1592 -/
1593 @[simp] theorem true_implies_length_ge_two
1594   (numbers : List Float) (t : Float)
1595   (h : hasCloseElements numbers t = true) :
1596   2 ≤ numbers.length := by
1597   sorry
1598 /--
1599 **Permutation invariance** :
1600 `hasCloseElements` depends only on the multiset of values,
1601 so it is stable under list permutations.
1602 -/
1603 @[simp] theorem perm_invariant
1604   (numbers numbers' : List Float) (t : Float)
1605   (hp : numbers ~ numbers') :
1606   hasCloseElements numbers t = hasCloseElements numbers' t := by
1607   sorry
1608 /--
1609 Imperative double-loop implementation (`hasCloseElementsImp`).
1610 -/
1611 def hasCloseElementsImp (numbers : List Float) (threshold : Float) : Bool :=
1612   Id.run do
1613     if numbers.length ≤ 1 then
1614       return false
1615     for i in [:numbers.length] do
1616       let x := numbers[i]!
1617       for j in [:numbers.length] do
1618         if i ≠ j then
1619           let y := numbers[j]!
1620           if Float.abs (x - y) < threshold then
1621             return true
1622       return false
1623 /-!
1624 # Imperative Tests
1625 -/
1626 /-- expected: true -/
1627 example : hasCloseElementsImp [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.3 = true := by
1628   native_decide
1629 #eval hasCloseElementsImp [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.3 -- expected: true
1630 /-!
1631 # Imperative Tests: Edge Cases
1632 -/
1633 /-- expected: false -/
1634 example : hasCloseElementsImp [] 1e-5 = false := by native_decide
1635 #eval hasCloseElementsImp [] 1e-5 -- expected: false

```

```

1620
1621 /--
1622 **Equivalence**: recursive and imperative implementations coincide.
1623 -/
1623 theorem hasCloseElements_equiv_functional_imperative
1624 (numbers : List Float) (threshold : Float) :
1625   hasCloseElements numbers threshold =
1626   hasCloseElementsImp numbers threshold := by
1627   sorry
1627 end HasCloseElements

```

Listing 16: An exemplar golden output Lean 4 code of VeriBench-HumanEval.

```

1631 def unsafe_copy(dst: bytearray, src: bytearray) -> None:
1632   """
1633   Copy bytes from `src` into `dst` at the same indices, without any bounds checking.
1634   If `len(src) > len(dst)`, this will raise an IndexError (buffer overflow).
1635   """
1636   for i, b in enumerate(src):
1637     dst[i] = b
1638
1639 def check(candidate) -> bool:
1640   # 1) Safe copy: src fits in dst
1641   d = bytearray(3)
1642   s = bytearray(b'abc')
1643   candidate(d, s)
1644   assert bytes(d) == b'abc'
1645
1646   # 2) Exact fit
1647   d2 = bytearray(2)
1648   s2 = bytearray(b'xy')
1649   candidate(d2, s2)
1650   assert bytes(d2) == b'xy'
1651
1652   # 3) Overflow: src longer than dst -> IndexError
1653   d3 = bytearray(2)
1654   s3 = bytearray(b'123')
1655   try:
1656     candidate(d3, s3)
1657     assert False, "Expected IndexError due to overflow"
1658   except IndexError:
1659     pass
1660
1661   # 4) Empty src -> no change
1662   d4 = bytearray(b'hello')
1663   candidate(d4, bytearray())
1664   assert bytes(d4) == b'hello'
1665
1666   # 5) Empty dst, nonempty src -> immediate overflow
1667   try:
1668     candidate(bytearray(), bytearray(b'z'))
1669     assert False, "Expected IndexError"
1670   except IndexError:
1671     pass
1672
1673   return True
1674
1675 assert check(unsafe_copy), "Candidate failed buffer-overflow tests"
1676 print("Pass!") # bell

```

Listing 17: An exemplar input Python code of VeriBench-SecuritySet (simplified for showcase).

```

1665 /-
1666 Description: A Lean 4 model of the unsafe copy routine that can overflow.
1667 We return `none` if an overflow (index out of bounds) would occur,
1668 and `some newDst` otherwise.
1669 -/
1670 namespace BufferOverflow
1671 /--
1672 `unsafeCopy dst src` attempts to overwrite the first `src.length` bytes of `dst`
1673 with those from `src`. Returns `some newDst` if `src.length ≤ dst.length`,
1674 otherwise `none`, modeling a buffer overflow.
1675 -/
1676 def unsafeCopy (dst src : List UInt8) : Option (List UInt8) :=

```

```

1674   let n := dst.length
1675   -- fold over enumerated bytes with their indices
1676   src.enum.foldl (fun o (i, b) =>
1677     o.bind fun acc =>
1678       if h : i < n then
1679         some (acc.set i b)
1680       else
1681         none
1682     ) (some dst)
1683
1684   /-! ## Examples / Unit Tests -/
1685
1686   #eval unsafeCopy [0x00,0x00,0x00] [0x41,0x42] -- some [0x41,0x42,0x00]
1687   #eval unsafeCopy [0x00,0x00] [0x61,0x62,0x63] -- none
1688
1689   example : unsafeCopy [0, 0, 0] [1,2] = some [1,2,0] := by rfl
1690   example : unsafeCopy [0, 0] [1,2,3] = none := by rfl
1691   example : unsafeCopy [0x68,0x69] [] = some [0x68,0x69] := by rfl
1692   example : unsafeCopy [] [0x7A] = none := by rfl
1693
1694   /-!
1695   # Theorem: safety precondition
1696
1697   If `src.length ≤ dst.length`, then `unsafeCopy dst src = some newDst` for some `newDst`.
1698   ## Proof:
1699   By construction, each index `i < src.length` satisfies `i < dst.length` → tail calls always
1700   succeed.
1701   Thus the fold never returns `none`, yielding `some` of the fully-updated buffer.
1702   -/
1703   theorem copy_safe {dst src : List UInt8}
1704     (h : src.length ≤ dst.length) :
1705     ∃ newDst, unsafeCopy dst src = some newDst := by
1706     unfold unsafeCopy
1707     -- For now, we admit this theorem since formalizing the foldl behavior
1708     -- requires more complex lemmas about foldl with guaranteed bounds
1709     admit
1710
1711   /-!
1712   # Theorem: overflow detection
1713
1714   If `src.length > dst.length`, then `unsafeCopy dst src = none`.
1715   ## Proof:
1716   At the first position `i = dst.length`, the check `i < dst.length` fails,
1717   causing the fold to return `none` immediately.
1718   -/
1719   theorem copy_overflow {dst src : List UInt8}
1720     (h : dst.length < src.length) :
1721     unsafeCopy dst src = none := by
1722     unfold unsafeCopy
1723     -- For now, we admit this theorem since formalizing the foldl behavior
1724     -- requires more complex lemmas about foldl with guaranteed bounds
1725     admit
1726
1727   end BufferOverflow

```

Listing 18: An exemplar golden output Lean 4 code of VeriBench-SecuritySet (simplified for showcase).

```

1714   # -- Implementation --
1715   # source: https://github.com/python/cpython/blob/3.13/Lib/heapq.py
1716
1717   # 'heap' is a heap at all indices ≥ startpos, except possibly for pos. pos
1718   # is the index of a leaf with a possibly out-of-order value. Restore the
1719   # heap invariant.
1720   def _siftdown(heap, startpos, pos):
1721     newitem = heap[pos]
1722     # Follow the path to the root, moving parents down until finding a place
1723     # newitem fits.
1724     while pos > startpos:
1725       parentpos = (pos - 1) >> 1
1726       parent = heap[parentpos]
1727       if newitem < parent:
1728         heap[pos] = parent
1729         pos = parentpos
1730         continue
1731       break
1732     heap[pos] = newitem

```

```

1728
1729 def heappush(heap, item):
1730     """Push item onto heap, maintaining the heap invariant."""
1731     heap.append(item)
1732     _siftdown(heap, 0, len(heap) - 1)
1733
1734 # -- Tests --
1735 from typing import Callable
1736 import random
1737
1738 def _check_invariant(heap):
1739     # Check the min-heap invariant: for every node, its value <= each child's value.
1740     for pos, item in enumerate(heap):
1741         if pos: # pos 0 has no parent
1742             parentpos = (pos - 1) >> 1
1743             assert heap[parentpos] <= item, (
1744                 f"heap invariant violated at pos={pos}: "
1745                 f"parent {heap[parentpos]} > child {item}"
1746             )
1747
1748 def check(candidate: Callable[[list, int], None]) -> bool:
1749     # Basic unit tests
1750     h = []
1751     candidate(h, 3)
1752     _check_invariant(h)
1753     candidate(h, 1)
1754     _check_invariant(h)
1755     candidate(h, 2)
1756     _check_invariant(h)
1757     assert h[0] == 1, f"expected min at root to be 1 but got {h[0]}"
1758
1759     # Edge unit tests: push onto empty; push equal elements
1760     h = []
1761     candidate(h, 5)
1762     _check_invariant(h)
1763     candidate(h, 5)
1764     _check_invariant(h)
1765     assert min(h) == h[0] == 5, f"expected root 5 but got {h[0]}"
1766
1767     # Property check: push 256 random numbers; heap must be a permutation and satisfy invariant,
1768     # and its root must equal min(data).
1769     data = []
1770     h = []
1771     for _ in range(256):
1772         x = random.random()
1773         data.append(x)
1774         candidate(h, x)
1775         _check_invariant(h)
1776     assert len(h) == len(data), "heap size changed unexpectedly"
1777     assert sorted(h) == sorted(data), "heap does not contain same multiset of items"
1778     assert h[0] == min(data), f"root {h[0]} != min(data) {min(data)}"
1779
1780     # Error behavior checks (mirrors heap_test.py expectations):
1781     # Calling with missing arguments should raise TypeError
1782     try:
1783         heappush([]) # type: ignore[arg-type]
1784         assert False, "heappush([]) should raise TypeError (missing arg)"
1785     except TypeError:
1786         pass
1787
1788     # Passing None heap or None item should raise (AttributeError or TypeError)
1789     try:
1790         heappush(None, None) # type: ignore[arg-type]
1791         assert False, "heappush(None, None) should raise"
1792     except (AttributeError, TypeError):
1793         pass
1794
1795     return True
1796
1797 if __name__ == "__main__":
1798     assert check(heappush), f"Failed: {__file__}"

```

Listing 19: An exemplar input Python code of VeriBench-RealCode.

```

1781 /-!
1782 # VeriBench Heappush (min-heap)

```

```

1782
1783 File order:
1784
1785 1. Implementation ('_siftdown', 'heappush')
1786 2. Unit tests (basic, edge, small property checks)
1787 3. Pre-condition
1788 4. Property propositions
1789 5. Post-condition (same order as properties)
1790 6. Correctness theorem 'Pre →Post'
1791
1792 All nontrivial proofs are left as 'sorry' for the learner/model/agent.
1793 -/
1794 namespace HeapPush
1795
1796 /--
1797 '_siftdown'
1798
1799 Given:
1800 * 'heap' : a min-heap at all indices  $\geq$  'startpos' except possibly at 'pos'
1801 * 'startpos' : root of the sifted subtree (usually '0')
1802 * 'pos' : index of a leaf whose value may violate the heap invariant
1803
1804 Restores the heap invariant by moving parents down until a slot for 'newitem'
1805 is found, then writes 'newitem' once.
1806 -/
1807 def _siftdown (heap : Array Int) (startpos pos : Nat) : Array Int :=
1808   let newitem := heap[pos]!
1809   let rec loop (h : Array Int) (pos : Nat) : Array Int :=
1810     if pos > startpos then
1811       let parentpos := (pos - 1) >>> 1
1812       let parent := h[parentpos]!
1813       if newitem < parent then
1814         let h' := h.set! pos parent
1815         loop h' parentpos
1816       else
1817         h.set! pos newitem
1818     else
1819       h.set! pos newitem
1820   loop heap pos
1821
1822 /--
1823 'heappush'
1824
1825 Push 'item' onto 'heap' (append) and restore heap invariant via '_siftdown'.
1826 -/
1827 def heappush (heap : Array Int) (item : Int) : Array Int :=
1828   let heap1 := heap.push item
1829   _siftdown heap1 0 (heap1.size - 1)
1830
1831 /-!
1832 # Tests
1833 -/
1834
1835 /-- Boolean heap invariant checker: for every child, parent  $\leq$  child. -/
1836 def checkInvariant (h : Array Int) : Bool :=
1837   let n := h.size
1838   let rec go (i : Nat) : Bool :=
1839     if i  $\geq$  n then
1840       true
1841     else if i = 0 then
1842       go (i + 1)
1843     else
1844       let parentpos := (i - 1) >>> 1
1845       if h[parentpos]!  $\leq$  h[i]! then
1846         go (i + 1)
1847       else
1848         false
1849   go 0
1850
1851 /-- expected: root becomes 1 after pushing 3,1,2 -/
1852 example : (heappush (heappush (heappush (#[[] : Array Int) 3) 1) 2)[0]! = 1 := by native_decide
1853 #eval (heappush (heappush (heappush (#[[] : Array Int) 3) 1) 2)[0]! -- expected: 1
1854
1855 /-- expected: pushing 5 twice keeps root 5 -/
1856 example :
1857   let h1 := heappush (#[[] : Array Int) 5
1858   let h2 := heappush h1 5
1859   h1[0]! = 5  $\wedge$  h2[0]! = 5 := by native_decide

```

```

1836 #eval (let h1 := heappush ([] : Array Int) 5; let h2 := heappush h1 5; (h1[0]!, h2[0]!)) --
1837   expected: (5, 5)
1838
1839 /-- positive: invariant holds after each push in [3,1,2] -/
1840 example :
1841   let h1 := heappush ([] : Array Int) 3
1842   let h2 := heappush h1 1
1843   let h3 := heappush h2 2
1844   checkInvariant h1 ^checkInvariant h2 ^checkInvariant h3 := by native_decide
1845 #eval (let h1 := heappush ([] : Array Int) 3; let h2 := heappush h1 1; let h3 := heappush h2
1846   2; (checkInvariant h1, checkInvariant h2, checkInvariant h3)) -- expected: (true, true,
1847   true)
1848
1849 /-!
1850 # Pre-Condition
1851 -/
1852
1853 /-- **Pre-condition.** The input `heap` already satisfies the min-heap invariant. -/
1854 def Pre (heap : Array Int) : Prop :=
1855   checkInvariant heap = true
1856
1857 /-!
1858 # Property Theorems
1859 -/
1860
1861 /-- **Invariant property** after `heappush`, the heap invariant holds. -/
1862 def prop_invariant (heap : Array Int) (item : Int) : Prop :=
1863   checkInvariant (heappush heap item) = true
1864
1865 /-- **Invariant theorem** `heappush` preserves the heap invariant (given a heap input). -/
1866 @[simp] theorem invariant_thm (heap : Array Int) (item : Int) (hPre : Pre heap) :
1867   prop_invariant heap item := by
1868   sorry
1869
1870 /-- **Size property** pushing increases the size by one. -/
1871 def prop_size (heap : Array Int) (item : Int) : Prop :=
1872   (heappush heap item).size = heap.size + 1
1873
1874 /-- **Size theorem** `heappush` increases the size by one. -/
1875 @[simp] theorem size_thm (heap : Array Int) (item : Int) :
1876   prop_size heap item := by
1877   sorry
1878
1879 /-- **Multiset property** contents are preserved up to permutation. -/
1880 def prop_multiset (heap : Array Int) (item : Int) : Prop :=
1881   List.Perm (heappush heap item).toList (item :: heap.toList)
1882
1883 /-- **Multiset theorem** `heappush` preserves contents up to permutation. -/
1884 @[simp] theorem multiset_thm (heap : Array Int) (item : Int) :
1885   prop_multiset heap item := by
1886   sorry
1887
1888 /-!
1889 # Post-Condition
1890 -/
1891
1892 /-- **Post-condition** for `heappush`. -/
1893 def Post_prop (heap : Array Int) (item : Int) : Prop :=
1894   prop_invariant heap item ^
1895   prop_size heap item ^
1896   prop_multiset heap item
1897
1898 /-!
1899 # Correctness Theorem
1900 -/
1901
1902 /-- **Correctness theorem** the pre-condition implies the post-condition. -/
1903 theorem correctness_thm (heap : Array Int) (item : Int) (hPre : Pre heap) :
1904   Post_prop heap item := by
1905   exact And.intro
1906     (invariant_thm heap item hPre)
1907     (And.intro (size_thm heap item) (multiset_thm heap item))
1908
1909 end HeapPush

```

Listing 20: An exemplar golden output Lean 4 code of VeriBench-RealCode.

1887

1888

1889