PV-BENCH: SPECIFICATION AND VERIFICATION OF COMPETITION-LEVEL ALGORITHMIC PROGRAMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Automating the verification of existing code requires both formalizing its intended behavior and proving that the implementation satisfies it. Many existing benchmarks let models generate the implementations they verify, while fixed-program benchmarks often supply the specifications. We present PV-BENCH, a benchmark for specifying and verifying fixed C programs that require reasoning about both algorithms and explicit memory manipulation. It contains 233 programs, including 147 accepted Codeforces solutions to problems rated from 800 to 2800, alongside classic algorithms, data structures, and library routines. Each program comes with an expert-reviewed reference specification and reference verification artifacts built with QCP. PV-BENCH evaluates three tasks independently: specification generation, annotation generation, and completion of residual Rocq proofs. The latter two tasks receive the reference artifacts from preceding stages, allowing each capability to be assessed separately. Candidate specifications are checked for semantic equivalence to the reference through machine-checked proofs and refutations, distinguishing equivalent, too-weak, too-strong, and incomparable contracts while leaving unresolved judgments undecided. Annotation and proof submissions are checked against the fixed program and specification. Our evaluation reveals complementary strengths across stages and substantial gaps in specification and annotation generation, even when proof completion with reference specifications and annotations succeeds. These findings show why proof completion alone gives an incomplete picture of a model's ability to verify an existing program.

1 INTRODUCTION

With the increasing use of language models in software development, human review can struggle to keep pace (He et al., 2026). Developers need a scalable way to establish that generated code behaves as intended. This motivates efforts to use language models for formal program verification (Loughridge et al., 2025; Yang et al., 2025). Program verification proves that the code meets a formal specification, which is a mathematical statement of the required behavior. If the specification is concise, developers can review it in place of the code. However, writing specifications and proofs still requires substantial time and expertise (Leino, 2010; Hawblitzel et al., 2015).

Verifying a given program therefore places two demands on the language model. First, it must write a specification that is worth reviewing: it should follow the problem statement, treat the given program as a black box that relates inputs to outputs, and be neither too weak nor too strong for the intended behavior. Second, it must prove that the program meets that specification, which requires reasoning about the program's algorithm and its memory representation. Both demands must be met on the program as given. Otherwise, success may reflect generating easy-to-verify code rather than verifying the program that needs checking.

Most existing benchmarks let the model write the program it verifies, which leaves it free to choose a program that is easy to verify. Some benchmarks (Zhao et al., 2026; Thakur et al., 2025; Bursuc et al., 2025) give the model a formal specification and ask it to implement the specification and prove the implementation correct. Others (Xie et al., 2026; Ye et al., 2025) include a task that gives the model a natural-language description and asks for the specification, the program, and the proof together. Because a specification hides implementation detail, the model can exploit this freedom to produce degenerate programs. For example, in vericoding (Bursuc et al., 2025), the specification of a

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

polynomial derivative constrains the length of the output, so a model wrote a program that always returns zeros, and it verified.

Moreover, existing benchmarks rarely consider real-world programs that both manipulate memory explicitly and implement nontrivial algorithms. DafnyBench (Loughridge et al., 2025) uses Dafny (Leino, 2010), in which programs never free memory or perform pointer arithmetic. VerusBench (Yang et al., 2025) and VeriContest (Xie et al., 2026) verify safe Rust programs with Verus (Lattuada et al., 2023), where Rust's type system guarantees the correctness of memory manipulation, such as never using memory after it is freed. As a result, these benchmarks cannot evaluate how models verify programs whose memory must be reasoned about explicitly, such as C or unsafe Rust programs.

We therefore present PV-BENCH, a benchmark in which models write formal specifications for given real-world C programs, and verify them against reference specifications. Each program was either written by a human, or generated by a language model from a detailed problem description and accepted by an online judge under its time and memory limits. The programs then stay fixed, so the model cannot change a program to make it easier to verify. The programs are written in C, so verifying them requires reasoning about memory. Many also implement competition-level algorithms: 33 of the 147 Codeforces problems are rated above 1800, against only 1 of 256 in VeriContest (Xie et al., 2026). Many problems are not pure algorithm exercises. Their statements describe concrete tasks without naming an algorithm, so solving them requires first carefully analyzing the problem and then designing an algorithm. A good specification states only what the task requires and hides implementation details. In the verification stage, the model needs to write annotations and proofs to establish why the algorithm solves the task and how the C code realizes it. This makes the benchmark closer to real verification practice.

PV-BENCH evaluates verification as three tasks over the same fixed program: writing the specification, writing the annotations such as loop invariants, and completing the proofs. Each program comes with an expert-reviewed reference specification and a reference verification built with QCP (Wu et al., 2026), a verification tool for C. Given a C program with its specification and annotations, QCP reduces the program's correctness to a set of verification conditions, proves most of them automatically, and leaves the rest to be proved in an interactive theorem prover such as Rocq (Bertot & Castéran, 2004) or Lean (de Moura & Ullrich, 2021). In the specification task (T1), the model is asked to write the formal specification from the problem statement and the code. In the annotation task (T2), the model is asked to write loop invariants and other annotations given a program and its reference specification. In the proof task (T3), the model is given the program with its reference specification and annotations, and is asked to prove the verification conditions that QCP leaves open. Table 1 compares these tasks and how they are judged with existing benchmarks.

We evaluate five frontier language models on PV-BENCH. For writing specifications, the best model achieves only 39.3% pass@1. Annotation is harder still, at 20%, while proof completion reaches 65.5%. These results suggest that verifying given programs remains substantially difficult, especially writing specifications and annotations.

To summarize, our main contributions are:

1. We present PV-BENCH, a benchmark for the formal specification and verification of real-world programs whose correctness entangles memory with algorithms, ranging from classical data structures to competition-level tasks (Section 3).
2. We propose a three-stage decomposition of the verification task over the same fixed program. The stages are evaluated independently, so failures can be attributed to the specification, the annotations, or the proof (Section 4.1).
3. We propose a judging method for specifications: a model-written specification is accepted only if it is proved equivalent to a reference. The method also classifies each failure as Too Weak, Too Strong, or Incomparable, each backed by a machine-checked proof (Section 4.2).

2 BACKGROUND

Program verification proves that a program is correct with respect to a formal specification. As Figure 1 shows, the conventional workflow (Leino, 2010; Cuoq et al., 2012) needs up to three types of

Table 1: Comparison with related benchmarks. *Fixed program*: the model verifies a program it did not write. *Spec. accepted by*: what a model-written specification must pass; *verifier* means only that the program verifies against it. *Spec. diagnosis*: rejected specifications are classified as too weak or too strong. *Annot.*: a task for loop invariants and other annotations. *Proving*: automatic (ATP) or interactive (ITP) theorem provers. *Memory*: the model must reason explicitly about memory, such as pointer validity, allocation, and aliasing. *Program source*: *Coding problems*, such as HumanEval (Chen et al., 2021), LeetCode, Codeforces, and NOIP; *Algo.*, classical algorithms; *Lib.*, library code; *Verif.*, existing verified programs and tasks from the verification competitions VerifyThis and SV-COMP. ✓ yes, ◐ partly, × no.

Benchmark	Fixed program	Spec. gen.	Spec. accepted by	Spec. diagnosis	Annot.	Proving	Memory	Program source			
								Coding problems	Algo.	Lib.	Verif.
DafnyBench (Loughridge et al., 2025)	✓	×	–	–	✓	ATP	×	✓	×	×	✓
Verus-Bench (Yang et al., 2025)	✓	×	–	–	✓	ATP	×	✓	×	×	✓
AlgoVeri (Zhao et al., 2026)	×	×	–	–	×	ATP, ITP	×	×	✓	×	×
CLEVER (Thakur et al., 2025)	×	✓	proof	×	×	ITP	×	✓	×	×	×
Vericoding (Bursuc et al., 2025)	×	×	–	–	×	ATP, ITP	×	✓	×	✓	✓
VERINA (Ye et al., 2025)	◐	✓	proof or tests	✓	×	ITP	×	✓	×	×	✓
VeriContest (Xie et al., 2026)	◐	✓	proof or tests	✓	✓	ATP	×	✓	×	×	×
VerifyThisBench (Deng et al., 2025)	×	✓	verifier	×	×	ATP	◐	×	×	×	✓
LiveFMBench (Xu et al., 2026a)	✓	◐	verifier	×	✓	ATP	✓	×	×	×	✓
VeriFast study (Fan et al., 2026)	✓	✓	verifier, manual	×	✓	ATP	✓	×	×	×	✓
PV-BENCH	✓	✓	proof	✓	✓	ATP, ITP	✓	✓	✓	✓	×

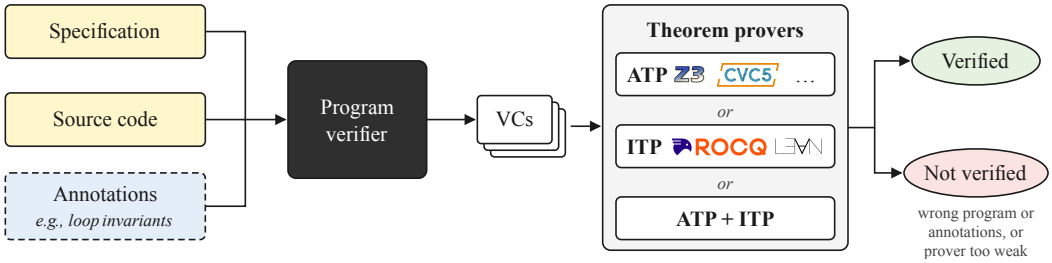


Figure 1: The conventional workflow of program verification.

artifacts to verify a program: a *specification*, *annotations*, and *proofs*. The specification is a contract describing the intended behavior of the program: the precondition states what the program assumes about its inputs and initial state, and the postcondition states what the program guarantees about its result and final state. Annotations, such as loop invariants, state facts inside the program that the verifier cannot infer by itself. From the annotated program and its specification, the verifier generates *verification conditions* (VCs), logical formulas whose validity implies that the program is correct. The program is then verified if every VC is proved, by automatic theorem provers (ATP) such as SMT solvers as in Dafny (Leino, 2010) and Verus (Lattuada et al., 2023), by manual proofs in an interactive theorem prover (ITP) such as Rocq as in VST (Appel, 2011; Cao et al., 2018), or by an ATP followed by an ITP as in Frama-C (Cuoq et al., 2012) and QCP (Wu et al., 2026). *Residual VCs* are VCs that ATPs do not discharge automatically and that a human or an AI agent must prove in an ITP. A VC that cannot be proved indicates a wrong program or wrong annotations. These three artifacts correspond to the tasks of PV-BENCH: writing the specification (T1), generating the annotations (T2), and proving residual VCs (T3).

Qualified C Programming Verifier (QCP) (Wu et al., 2026) is a verifier for C programs. We choose it because it can reason about both memory manipulation and nontrivial algorithms. For memory, its specifications and annotations are written in separation logic (Reynolds, 2002), which states both which memory the program may access and what values that memory holds. QCP follows the ATP-then-ITP workflow of Figure 1. It first tries to prove each VC with its built-in ATP, which combines separation-logic rules with a lightweight SMT solver. Because these VCs can be then proved in an ITP (Rocq or Lean), QCP can verify programs that combine complex memory manipulation with nontrivial algorithms. Besides, QCP writes specifications and annotations as comments, so the annotated program remains an ordinary executable C file. Appendix A describes QCP in detail.

A Function Specification should accept every behavior that the problem allows and reject every behavior that it forbids (Endres et al., 2024; Chen et al., 2026), and should state this property without describing how the program computes it (Wing, 1990). For example, a function that sorts an array `a` can be specified with a logical variable l for its initial contents: the precondition $P(l)$ states that `a` holds l, and the postcondition $Q(l)$ states that `a` then holds a list that is ordered and a permutation of l. Because it does not say how the array is sorted, this specification applies to any sorting algorithm.

3 THE PV-BENCH BENCHMARK

PV-BENCH evaluates whether language models can write the formal specification of a given C source program and verify it without changing its implementation. This section describes how PV-BENCH is constructed. Section 3.1 describes its programs, and Section 3.2 describes their reference verifications and how we checked them.

3.1 DATASET COMPOSITION AND ALGORITHMIC DIVERSITY

The current PV-BENCH snapshot contains 233 C implementations paired with reference specifications, annotations, and complete proofs for VCs. Some problems have multiple implementations in the benchmark. The dataset draws from two main sources:

1. **Classic Algorithms, Data Structures, and Engineering (86 instances):** This source provides 73 algorithm implementations, five data-structure modules, and eight engineering routines. The algorithm group includes 41 classic algorithms, 13 LeetCode problems, and 19 Luogu problems; the engineering routines include C string and memory functions and MiniSat-style dynamic vectors.
2. **Codeforces Problems (147 instances):** These problems have ratings from 800 to 2800. Each C source program was generated by a language model (Claude Opus 4.8 or GPT 5.6 sol) from a detailed problem description and accepted by the online judge under the problem's time and memory limits. Of these, 26 are rated 2000 or above. Solving such problems requires expert-level algorithmic reasoning, extending the benchmark to the verification of demanding competitive-programming solutions.

Algorithmic Scope Verifying the programs requires reasoning about both the mathematical properties of their algorithms and concrete memory representations. The algorithms cover sorting, greedy algorithms, dynamic programming, graph algorithms, number theory, computational geometry, and specialized data structures. The memory representations include arrays, matrices, strings, and linked structures.

Multi-Implementation Variants To facilitate a deeper exploration of verification case studies for individual algorithms,PV-BENCH includes multiple implementations of selected algorithms. Examples include five quicksort partition schemes, four Prim implementations, and three depth-first-search implementations. Variants share the same specification but differ in control flow, helper calls, data layout, and update order.

Dataset Statistics The benchmark contains 12,599 lines of C code across 233 instances, averaging 54.1 lines per instance. With the reference specifications and annotations, QCP generates 16,724 VCs, of which 4,638 are residual VCs. Table 2 reports the number of programs in each category and the mean code size and VC counts per program. Codeforces programs rated 2000 or above average 177.4 VCs and 47.1 residual VCs, more than three times as many as those rated below 2000.

3.2 REFERENCE ARTIFACTS

Each instance includes its problem statement, the source of the problem, and a complete reference verification, which the tasks use both as inputs and as the standard for judging. All source programs are validated by online-judge tests or existing unit tests. The reference verification consists of a specification with its mathematical definitions, annotations, and Rocq proofs of the residual VCs. T1 judges a model-written specification against the reference specification, T2 gives the model the reference specification, and T3 gives it the reference specification and annotations.

Table 2: Summary of the 233-instance Rocq benchmark. *Programs* counts the programs in each category, and the other columns are means per program.

Category	Programs	C LOC	Total VCs	Residual VCs
Algorithms / DS / Engineering	86	55.0	66.6	19.5
Codeforces 800–1900	121	47.8	52.8	14.4
Codeforces 2000–2800	26	80.3	177.4	47.1
Total	**233**	**54.1**	**71.8**	**19.9**

Specification Auditing Reference specifications are co-authored by human experts and AI agents. Each specification is reviewed by experts against the original problem descriptions. The precondition (`Require`) specifies the valid input domain, memory layout, and the conditions that rule out integer overflow. The postcondition (`Ensure`) describes the expected output property and memory state. The proof ensures the code against this contract and the expert review checks correspondence between the contract and the natural-language requirement.

Annotations and proofs. We check that every reference verification covers all VCs generated from the annotated C source program. QCP proves routine VCs automatically, and Rocq checks the reference proofs of the residual VCs. Together, these results establish that the program satisfies its reference specification. Appendix B documents the verification files and evaluation checks.

4 Tasks and Judging

PV-Bench evaluates verification as three tasks on the same fixed program, and every accepted result is checked by Rocq or QCP. Section 4.1 defines the three tasks and how each is checked. Section 4.2 then describes how T1 judges a model-written specification.

4.1 Tasks

The three tasks follow the workflow in Figure 1. Each task asks the model for one artifact and gives it the reference outputs of the preceding steps, but not the reference answer for its own task. The model may not change the C implementation, and the evaluator checks that the program and reference inputs are unchanged before judging a submission. Figure 2 shows how T1 and T2 generate and evaluate candidate specifications and annotations.

T1: Specification generation. T1 asks the model to write a formal specification and any supporting mathematical definitions from the problem statement and the fixed C implementation. A submission is valid if it is syntactically correct, leaves the C code unchanged, and gives exactly one contract, on the required function. A separate judge model (GPT-5.6-Sol) checks whether the candidate specification and the reference imply each other. For each direction, it tries to produce either a Rocq proof or a counterexample checked by Rocq. The specification passes only if both directions are proved. Evidence for the two directions distinguishes *Too Weak*, *Too Strong*, and *Incomparable* specifications; an incomplete judgment remains *Undecided*.

T2: Program annotation. T2 gives the model the C source program and its reference specification and asks it to add loop invariants and other annotations, with supporting definitions as needed. The model can run QCP to inspect diagnostics and residual VCs and revise its annotations. A submission is valid if it is syntactically correct, leaves the executable C code and the reference contract unchanged, and its helper definitions contain no admitted proofs or new axioms. QCP then generates VCs from the submitted annotations, and A separate judge model (GPT-5.6-Sol) attempts to prove the residual VCs in Rocq. To help the judge, we give it the reference annotations, their VCs, and their Rocq proofs, which it may adapt but which never replace proofs of the candidate's own VCs. The annotations pass when the automatic results and checked proofs together cover every generated VC. A checked counterexample rejects the submission; a judgment with neither a complete proof nor a counterexample remains unresolved.

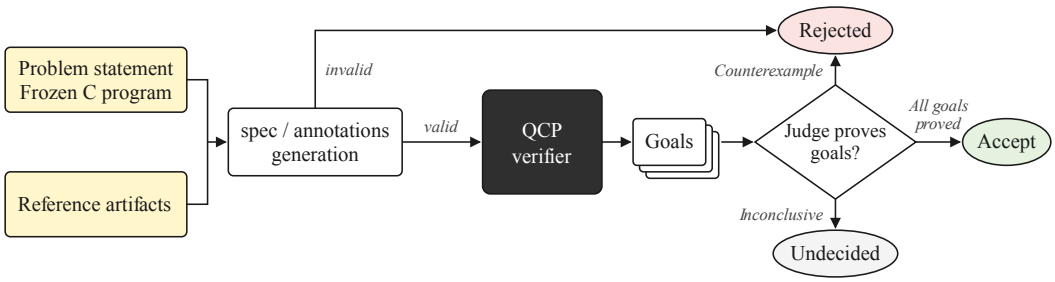


Figure 2: Generation and evaluation for T1 and T2. A separate judge model attempts to prove or refute the VCs generated from each valid submission and the corresponding reference artifacts.

T3: Proof completion. T3 gives the model the C source program, reference annotations, and a proof skeleton for the residual VCs. The model writes Rocq proofs and the supporting lemmas it needs, with the reference proofs hidden. The submission passes when Rocq accepts proofs of every residual VC, without skipped proofs or unapproved assumptions.

4.2 Judging Specifications by Proof

PV-Bench accepts a model-written specification only if it is proved equivalent to the reference specification. QCP reduces the comparison of a model-written specification and the reference specification to two separation-logic obligations, one for each direction. We attach the reference contract S and the candidate contract $\hat{S}$ to the same C function, so their parameters, memory, and return value refer to the same objects. Let $P(x)$ and $Q(x)$ be the precondition and postcondition of S over its logical variables x, and let $\hat{P}(y)$ and $\hat{Q}(y)$ be those of $\hat{S}$ over y. The two obligations are

$$S \Rightarrow \hat{S}: \quad \forall y.\, \hat{P}(y) \vdash \exists x.\, P(x) * \big(Q(x) \mathrel{-\!\!*} \hat{Q}(y)\big),$$
$$\hat{S} \Rightarrow S: \quad \forall x.\, P(x) \vdash \exists y.\, \hat{P}(y) * \big(\hat{Q}(y) \mathrel{-\!\!*} Q(x)\big).$$

$S \Rightarrow \hat{S}$ states that every implementation meeting S also meets $\hat{S}$. The separating conjunction $A * B$ holds when memory splits into two disjoint parts, one satisfying A and one satisfying B. The magic wand $A \mathrel{-\!\!*} B$ describes a memory that lacks only A to become B: adding any disjoint memory that satisfies A to it yields memory that satisfies B. A caller who establishes $\hat{P}(y)$ must therefore be able to split the memory into a part satisfying $P(x)$, which an implementation of S receives, and a frame that the call does not touch. When the call returns memory satisfying $Q(x)$, the wand adds the frame back to give $\hat{Q}(y)$. For example, suppose S requires an array of n cells and returns them, while $\hat{S}$ requires and returns $n+1$ cells. Then $S \Rightarrow \hat{S}$ holds, because the extra cell stays in the frame. But $\hat{S} \Rightarrow S$ fails, because a caller of S owns only n cells and cannot supply the $n+1$ that $\hat{S}$ requires.

The two directions yield the four verdicts in Table 3. A specification is *Equivalent* when both directions are proved. *Too Weak* means that $\hat{S}$ admits an implementation that S forbids. For the sorting specification in Section 2, a postcondition that omits "a permutation of l" is too weak, because it accepts a program that fills the array with zeros. *Too Strong* means that $\hat{S}$ rejects an implementation that S allows. A postcondition that requires a strictly increasing output is too strong, because it rejects the correct output whenever the input contains duplicates. It is *Incomparable* when both directions are refuted. Appendix C gives one example of each settled verdict.

We use an independent judge model to find a proof or refutation within its budget, and Rocq checks every claim it makes. A weaker judge can therefore turn verdicts into *Undecided*, but it cannot turn a wrong specification into an Equivalent one.

5 Evaluation

We evaluate how well models write specifications, write annotations, and complete proofs on PV-Bench. We compare their success rates on each task and use results on the same programs to examine what separate task scores reveal.

Table 3: Verdicts of the specification judge. A ✓ marks a required proof ($\Rightarrow$) or refutation ($\not\Rightarrow$).

$S \Rightarrow \hat{S}$	$\hat{S} \Rightarrow S$	$S \not\Rightarrow \hat{S}$	$\hat{S} \not\Rightarrow S$	Verdict
✓	✓			Equivalent
		✓	✓	Incomparable
			✓	Too Weak
		✓		Too Strong

5.1 Experimental Setup

For each task, we split the 233 programs into testing and reference examples, and evaluate models with one attempt per program and a fixed time budget. We test T1 on 211 programs and T3 on 223 programs. We evaluate T2 on the same 50 sampled instances for each model, using stratified random sampling in proportion to the sizes of the program categories. Specification generation, annotation generation, and proof completion have time limits of 30, 60, and 120 minutes, respectively. The judge model is GPT-5.6 Sol and has 60 minutes for each specification or annotation submission.

We evaluate GPT-5.6 Sol, Luna, and Terra through Codex, and Claude Sonnet 5 and Opus 5 through Claude Code. Our comparisons describe the models together with the task-specific settings below. We report pass@1, the fraction of programs whose single submission passes every check of its task. Invalid submissions, timeouts, and unsettled judgments count as failures. We record the model settings, prompts, examples, tools and inputs, and verification results (Appendix E).

5.2 Results

The three tasks differ both in difficulty and in how far they separate the models (Table 4). T2 has the lowest pass rates: no model passes more than 20.00% of the 50 sampled programs. In T1, the four models span less than 10 percentage points, from 29.86% for Luna to 39.34% for Opus.

Table 4: Pass@1 on each task, with the number of programs passed.

(a) T1, 211 programs

Model	Pass	Pass@1
Opus 5	83	39.34%
Sonnet 5	71	33.65%
Terra	64	30.33%
Luna	63	29.86%

(b) T2, 50 programs

Model	Pass	Pass@1
Sol	10	20.00%
Sonnet 5	9	18.00%
Luna	4	8.00%

(c) T3, 223 programs

Model	Pass	Pass@1
Sonnet 5	146	65.47%
Sol	104	46.64%
Luna	35	15.70%
Terra	31	13.90%

T3 separates the models most, with pass rates from 13.90% to 65.47%. As the table shows, the task shows a 51.57-point gap between the highest and lowest scores. The T3 experiment distinguishes the models' ability to prove residual VCs in Rocq.

The models rank differently on the three tasks, so each task measures a distinct ability. In T1, Sonnet, Luna, and Terra lie within 3.8 percentage points of each other (29.86% to 33.65%), and Opus leads at 39.34%. In T3, Sonnet proves 65.47% of the programs, more than four times the rates of Luna and Terra. A model that writes specifications about as well as Sonnet may therefore still fail to prove most programs. A single end-to-end score would hide this difference.

Success on one task also does not predict success on another for the same program. On the 211 programs used in both T1 and T3, Sonnet passes both tasks on 51 programs, only T1 on 20, only T3 on 92, and neither on 48. For those 92 programs, Sonnet proves the program correct once given the reference specification and annotations, but its own specification is not proved equivalent to the reference. Terra shows the opposite pattern. Of the 64 programs whose specifications it writes equivalently, it proves only 11 in T3. In an end-to-end evaluation, each of these programs would count as one failure without saying which step failed. Separate tasks credit each ability where the model has it and attribute each failure to one step.

Proof difficulty follows the difficulty of the algorithm, while specification difficulty depends more on memory (Table 5). In T3, Sonnet's pass rate falls from 80% on Codeforces problems rated up to

Table 5: Pass@1 (%) by program category. n is the number of programs in each task. Codeforces programs are grouped by rating.

	T1					T3		
Category	n	Opus	Sonnet	Luna	Terra	n	Sonnet	Terra
Algorithms	59	46	41	42	31	71	62	10
Data structures	4	25	0	0	0	4	100	0
Engineering	8	0	0	0	0	8	100	75
Codeforces 800–1400	71	44	41	28	38	71	80	25
Codeforces 1500–1900	43	37	26	26	26	43	56	0
Codeforces 2000–2800	26	31	27	27	31	26	35	0

1400, to 56% on those rated 1500 to 1900, and to 35% on those rated 2000 or above. Terra proves none of the 69 programs rated 1500 or above. In T1, the rates fall less with rating, for example from 44% to 31% for Opus. The library code shows the opposite pattern. No model writes a specification equivalent to the reference for any of the eight library routines, and only Opus does so for one of the four data structures. For Sonnet, Opus, and Luna, at least half of the specifications for library routines are judged Too Strong. Yet Sonnet proves all twelve of these programs in T3 once given the reference specification and annotations. Stating exactly what a C routine requires of memory and what it leaves unchanged is therefore hard even when the proof is easy.

5.3 Failure Analysis

Specifications. Model-written specifications fail mostly by being too strong. For every model, Too Strong verdicts outnumber Too Weak ones. A typical Too Strong specification quantifies the output before the call, so no implementation can meet it. For example, in Codeforces 1537B the program must output two optimal grid positions, but one candidate declares the output as a parameter chosen before the call. Its contract then promises that the output is optimal whatever value the caller picks, even an empty list, which no program can satisfy (Appendix C.3).

Annotations. Writing annotations appears to be the hardest step. On the 50 programs sampled for T2, Sonnet's annotations pass for 9 programs (18.00%), while Sonnet proves 35 of the same 50 programs (70%) in T3 with the reference annotations. All nine accepted programs are simple: five Codeforces problems rated at most 1400, integer division, selection sort, `memchr`, and `strcpy`. A classic refuted submission comes from rod cutting. Sonnet's loop invariants state `n == n` where the reference states `n == n@pre`, so they no longer record that the parameters keep their entry values. The VC at the return point then cannot relate the result to the original input, and the judge refutes it with a state whose entry rod length is -1 while the current one is 0 (Appendix D).

Proofs. Failed proofs differ in kind between models. Of Sonnet's 41 failed proofs, 29 stop at a goal left `Admitted`, and only 12 do not compile. Terra shows the reverse: 114 of its 153 failed proofs do not compile. A further 29 Sonnet and 37 Terra submissions are invalid, mostly because the run failed the sandbox audit or changed files outside its answer (Appendix B.3).

6 Related Work

Existing benchmarks study verified-program generation, specification generation, and proof or annotation completion. PV-Bench connects these settings on the same fixed C implementations.

Zhao et al. (2026) introduce AlgoVeri, which aligns 77 classical algorithm problems across Dafny, Verus, and Lean. Each task provides a problem statement, a function signature, and an expert-written specification; the model writes the implementation and its proof. However, this leaves the model free to choose the control flow, helper functions, and data representation. The VeriCoding benchmark similarly collects 12,504 formal specifications across Dafny, Verus, and Lean for generating implementations and proofs from supplied specifications (Bursuc et al., 2025). In PV-Bench, the C implementation, including its control flow, stays fixed. This reflects the practical task of verifying an existing implementation.

Ye et al. (2025) present VERINA, a collection of 189 Lean tasks with separate and combined modes for code, specification, and proof generation. Its specification evaluation first attempts to prove soundness and completeness against a reference specification and then uses examples, decidable checks, and property-based tests when proof does not settle the comparison. VERINA establishes modular specification evaluation in Lean; PV-BENCH applies a related bidirectional criterion to contracts for C source programs with explicit memory states.

Xie et al. (2026) introduce VeriContest, which contains 946 LeetCode and Codeforces problems paired with expert-reviewed Verus specifications, judge-accepted Rust implementations, and Verus proofs. It evaluates specification, code, and proof generation both independently and in combined settings. Its ProofGen task fixes the program and specification. PV-BENCH also evaluates the stages separately: specification generation, annotation generation, and proofs of residual VCs, all on the same fixed C implementation.

Verus-based benchmarks rely on safe Rust's ownership, borrowing, and type system to handle memory safety and aliasing (Lattuada et al., 2023). QCP expresses heap memory through separation logic, with explicit assertions about allocation, initialization, pointer ranges, and disjoint regions (Wu et al., 2026; Reynolds, 2002). This removes the dependence on Rust's memory guarantees, so we expect the benchmark design to be easier to adapt to other languages.

Verus-Bench contains 150 tasks with fixed Rust code and fixed Verus specifications (Yang et al., 2025). It removes loop invariants, assertions, and proof functions, and accepts a completion when Verus and Z3 verify the restored program. DafnyBench similarly removes proof hints from fixed Dafny programs and asks a model to restore enough annotations for Dafny to verify them (Loughridge et al., 2025). Both directly test verification of an existing implementation. PV-BENCH separates this work into T2, which generates annotations and reruns symbolic execution, and T3, which proves fixed residual VCs generated from reference annotations.

VerifyThisBench contains 154 tasks derived from 41 VerifyThis competition problems and asks a model to generate the specification, implementation, and proof as one submission (Deng et al., 2025). Its relaxed VerifyThisBenchXS setting additionally supplies partial implementations, specifications, or proofs. PV-BENCH uses reviewed reference artifacts from preceding stages to evaluate each task separately, so that a failure can be traced to the specification, annotations, or proofs of residual VCs.

C-focused work is closest to PV-BENCH in its treatment of concrete program state. LiveFMBench evaluates agentic generation of ACSL specifications and annotations for 630 C source programs with Frama-C/WP (Xu et al., 2026a). A recent empirical study evaluates generated specifications for 303 functions from 60 C source programs with the separation-logic verifier VeriFast, including loops, concurrency, and recursive heap predicates (Fan et al., 2026). These works focus on C specification and annotation generation. NTP4VC starts from VCs extracted from real C projects and asks models to prove them in Isabelle, Lean, or Rocq (Xu et al., 2026b). PV-BENCH evaluates entry specifications, program annotations, and Rocq proofs of residual VCs on the same fixed C implementations, covering the verification workflow more fully.

7 CONCLUSION

We presented PV-BENCH, a benchmark for specifying and verifying C programs whose correctness depends on both explicit memory manipulation and nontrivial algorithms. It evaluates specification, annotation, and proof as separate tasks on the same program, and it accepts a specification only when a machine-checked proof shows it equivalent to the reference. Our evaluation of five frontier models shows that proof completion alone overstates how well models verify existing code. The best models reach 65.5% pass@1 on T3, but only 39.3% on T1 and 20% on the sampled T2 instances, and success on one task does not predict success on another. Success on T3 falls sharply as the Codeforces rating rises, while success on T1 falls much less. This suggests that the specifications state what the problem requires without relying on how the program implements it, so writing them does not grow harder with the algorithm. On the same 50 programs, Sonnet's own annotations pass for 18%, while it completes the proofs for 70% given the reference annotations. These results point to specifications and annotations as the main bottlenecks in verifying the programs that developers already have.

ETHICS STATEMENT

This work involves no human subjects and no personal data. The benchmark consists of algorithmic C programs, classic algorithms and data structures, and C library routines, together with problem statements from public programming platforms (Codeforces, LeetCode, and Luogu), and each instance records the source of its problem. We release the programs, specifications, annotations, and proofs under the MIT License. The problem statements remain subject to the terms of their source platforms. The model-generated Codeforces solutions were submitted to the online judge only outside live contests.

REPRODUCIBILITY STATEMENT

We describe the dataset and reference verification in Sections 3.1 and 3.2, the experimental setup in Section 5, and the prompts and checking procedures in Appendices B and E. AI agents carry out both benchmark runs and evaluation, so repeated runs can produce different outputs, judgments, and scores. We will release the benchmark and evaluation code on GitHub and Hugging Face.

AI USE STATEMENT

We used generative AI to assist with reference specifications, and perform the generation and judging tasks described in Section 5. We also used AI to polish the manuscript's language and figures. Human experts reviewed the reference specifications, and verification tools checked the proof evidence.

REFERENCES

Andrew W. Appel. Verified software toolchain - (invited talk). In Gilles Barthe (ed.), *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pp. 1–17. Springer, 2011. doi: 10.1007/978-3-642-19718-5_1. URL https://doi.org/10.1007/978-3-642-19718-5_1.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN 978-3-642-05880-6. doi: 10.1007/978-3-662-07964-5. URL https://doi.org/10.1007/978-3-662-07964-5.

Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, Miranda Zhao, Max Tan, and Max Tegmark. A benchmark for vericoding: formally verified program synthesis. *CoRR*, abs/2509.22908, 2025. doi: 10.48550/ARXIV.2509.22908. URL https://doi.org/10.48550/arXiv.2509.22908.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. Vst-floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reason.*, 61(1-4):367–422, 2018. doi: 10.1007/S10817-018-9457-5. URL https://doi.org/10.1007/s10817-018-9457-5.

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

Zaoyu Chen, Jianbo Dai, Boyu Zhu, Jingdong Wang, Huiming Wang, Xin Xu, Haoyang Yuan, Zhijiang Guo, and Xiao-Ming Wu. Codespecbench: Benchmarking llms for executable behavioral specification generation. *CoRR*, abs/2604.12268, 2026. doi: 10.48550/ARXIV.2604.12268. URL https://doi.org/10.48550/arXiv.2604.12268.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c - A software analysis perspective. In George Eleftherakis, Mike Hinchey, and Mike Holcombe (eds.), *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, volume 7504 of *Lecture Notes in Computer Science*, pp. 233–247. Springer, 2012. doi: 10.1007/978-3-642-33826-7_16. URL https://doi.org/10.1007/978-3-642-33826-7_16.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe (eds.), *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37. URL https://doi.org/10.1007/978-3-030-79876-5_37.

Xun Deng, Sicheng Zhong, Andreas G. Veneris, Fan Long, and Xujie Si. Verifythisbench: Generating code, specifications, and proofs all at once. *CoRR*, abs/2505.19271, 2025. doi: 10.48550/ARXIV.2505.19271. URL https://doi.org/10.48550/arXiv.2505.19271.

Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proc. ACM Softw. Eng.*, 1(FSE):1889–1912, 2024. doi: 10.1145/3660791. URL https://doi.org/10.1145/3660791.

Wen Fan, Minh Tran, Sanya Dod, Xin Hu, Marilyn Rego, Danning Xie, Jenna DiVincenzo, and Lin Tan. An empirical study of llm-generated specifications for verifast. *CoRR*, abs/2606.26490, 2026. doi: 10.48550/ARXIV.2606.26490. URL https://doi.org/10.48550/arXiv.2606.26490.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael Lowell Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand (eds.), *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pp. 1–17. ACM, 2015. doi: 10.1145/2815400.2815428. URL https://doi.org/10.1145/2815400.2815428.

Hao He, Shyam Agarwal, Yegor Denisov-Blanch, Pavel Azaletskiy, Sanmi Koyejo, and Bogdan Vasilescu. AI writes faster than humans can review: A longitudinal study of an enterprise 2x mandate. *CoRR*, abs/2607.01904, 2026. doi: 10.48550/ARXIV.2607.01904. URL https://doi.org/10.48550/arXiv.2607.01904.

James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. doi: 10.1145/360248.360252. URL https://doi.org/10.1145/360248.360252.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023. doi: 10.1145/3586037. URL https://doi.org/10.1145/3586037.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov (eds.), *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pp. 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20. URL https://doi.org/10.1007/978-3-642-17511-4_20.

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. Dafnybench: A benchmark for formal software verification. *Trans. Mach. Learn. Res.*, 2025, 2025. URL https://openreview.net/forum?id=yBgTVWccIx.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pp. 55–74. IEEE Computer Society, 2002. doi: 10.1109/LICS.2002.1029817. URL https://doi.org/10.1109/LICS.2002.1029817.

Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. CLEVER: A curated benchmark for formally verified code generation. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/a7f67788f7b4d77fa7cd6887de3dcbe7-Abstract-Datasets_and_Benchmarks_Track.html.

Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, 1990. doi: 10.1109/2.58215. URL https://doi.org/10.1109/2.58215.

Xiwei Wu, Yueyang Feng, Xiaoyang Lu, Tianchuan Lin, Kan Liu, Zhiyi Wang, Shushu Wu, Lihan Xie, Chengxi Yang, Hongyi Zhong, Zihan Zhang, Juanru Li, Naijun Zhan, Zhenjiang Hu, and Qinxiang Cao. QCP: A practical separation logic-based C program verification tool. In Giuseppe De Giacomo, Jianwen Li, and Lili Xiao (eds.), *Theoretical Aspects of Software Engineering - 20th International Symposium, TASE 2026, Shanghai, China, July 4-6, 2026, Proceedings*, volume 16697 of *Lecture Notes in Computer Science*, pp. 157–173. Springer, 2026. doi: 10.1007/978-3-032-30693-7_10. URL https://doi.org/10.1007/978-3-032-30693-7_10.

Zichen Xie, Mrigank Pawagi, Yuxin Liu, Aaditi Rai, Lize Shao, John Berberian Jr., Sicong Che, and Wenxi Wang. Vericontest: A competitive-programming benchmark for verifiable code generation. *CoRR*, abs/2605.08553, 2026. doi: 10.48550/ARXIV.2605.08553. URL https://doi.org/10.48550/arXiv.2605.08553.

Dong Xu, Jialun Cao, Guozhao Mo, Junjie Hu, Cheng Wen, Hongyu Lin, Xianpei Han, Shengchao Qin, Cong Tian, Shing-Chi Cheung, Le Sun, and Yaojie Lu. Livefmbench: Unveiling the power and limits of agentic workflows in specification generation. *CoRR*, abs/2605.01394, 2026a. doi: 10.48550/ARXIV.2605.01394. URL https://doi.org/10.48550/arXiv.2605.01394.

Qiyuan Xu, Xiaokun Luan, Renxi Wang, Joshua Ong Jun Leang, Peixin Wang, Haonan Li, Wenda Li, and Conrad Watt. Neural theorem proving for verification conditions: A real-world benchmark. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=MfDyickxQA.

Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu K. Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. Autoverus: Automated proof generation for rust code. *Proc. ACM Program. Lang.*, 9(OOPSLA2): 3454–3482, 2025. doi: 10.1145/3763174. URL https://doi.org/10.1145/3763174.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: benchmarking verifiable code generation. *CoRR*, abs/2505.23135, 2025. doi: 10.48550/ARXIV.2505.23135. URL https://doi.org/10.48550/arXiv.2505.23135.

Haoyu Zhao, Ziran Yang, Jiawei Li, Deyuan He, Zenan Li, Chi Jin, Venugopal V. Veeravalli, Aarti Gupta, and Sanjeev Arora. Algoveri: An aligned benchmark for verified code generation on classical algorithms. *CoRR*, abs/2602.09464, 2026. doi: 10.48550/ARXIV.2602.09464. URL https://doi.org/10.48550/arXiv.2602.09464.

A A WORKED EXAMPLE OF QCP VERIFICATION

QCP (Wu et al., 2026) verifies an annotated C program by symbolic execution, automatic discharge of verification conditions (VCs), and proof-assistant checking of residual VCs. This appendix follows that process for the reference program `Algorithms/selection_sort`. The example connects a mathematical sorting property to concrete array updates and finally to checked proof obligations.

A.1 THE PROGRAM AND ITS CONTRACT

The program sorts an integer array in place by scanning the remaining suffix and exchanging a smaller element with the current position. Listing 1 shows its executable body and function contract; imports and loop annotations are omitted from this first view. QCP annotations are written in `/*@ ... */` comments, so they do not change the executable C statements. The `With` variable `l` names the initial mathematical contents of the array.

Listing 1: Selection sort and its QCP contract. Loop invariants are explained below.

```
void sortArray(int *nums, int numsSize)
/*@ With (l: list Z)
    Require 1 <= numsSize && numsSize <= 50000 &&
            IntArray::full(nums, numsSize, l)
    Ensure exists l1,
           Permutation(l, l1) && increasing(l1) &&
           IntArray::full(nums, numsSize, l1)
*/
{
    for (int i = 0; i < numsSize; ++i) {
        for (int j = i + 1; j < numsSize; ++j) {
            if (nums[j] < nums[i]) {
                int tmp = nums[i];
                nums[i] = nums[j];
                nums[j] = tmp;
            }
        }
    }
    return;
}
```

The postcondition requires both sortedness and preservation of the input multiset. Here `increasing` means nondecreasing, so duplicate values are permitted, and `Permutation(l,l1)` ensures that no input element is lost or invented. Sortedness alone would allow an implementation to overwrite the array with zeros. The existentially quantified `l1` describes the resulting contents without prescribing how to obtain them.

The array predicate connects this mathematical contract to memory. `IntArray::full(nums,n,l)` describes ownership of an initialized integer array of length `n`, starting at `nums`, whose contents are `l`. QCP uses separation logic (Reynolds, 2002): the separating conjunction $P * Q$ describes disjoint memory resources. An array access requires both an in-bounds index and ownership of the corresponding cell; an assertion about a mathematical list alone cannot justify a C memory access.

A.2 Loop invariants and symbolic execution

The outer invariant describes a sorted prefix that is already in its final relation to the remaining suffix. Listing 2 reproduces the QCP assertion immediately preceding the outer loop. The existential variable `a` describes the current array, while `l` retains its initial contents.

Listing 2: The outer-loop invariant in QCP assertion syntax.

```
/*@ Inv Assert
    exists a,
        nums == nums@pre && numsSize == numsSize@pre &&
        1 <= numsSize && numsSize <= 50000 &&
        0 <= i && i <= numsSize &&
        Permutation(l, a) &&
        increasing(sublist(0, i, a)) &&
        (forall (p: Z) (q: Z),
            (0 <= p && p < i && i <= q && q < numsSize) =>
            (a[p] <= a[q])) &&
        IntArray::full(nums, numsSize, a)
*/
```

The assertion keeps the pointer and array size equal to their entry values, expressed using `@pre`. It requires the prefix `sublist(0,i,a)` to be sorted and every prefix element to be no larger than every suffix element. At $i = 0$, the prefix is empty and the cross-boundary ordering condition is vacuous. At `i=numsSize`, the prefix is the whole array, so the invariant implies the function's postcondition. The permutation and memory facts must hold throughout, not only at loop exit.

The inner invariant records the smallest value encountered in the scanned part of the suffix. Listing 3 shows the complete QCP assertion preceding the inner loop. It retains the outer facts, strengthens the bounds on `i` and `j`, and adds the minimum property for `sublist(i,j,a)`.

Listing 3: The inner-loop invariant in QCP assertion syntax.

```
/*@ Inv Assert
    exists a,
        nums == nums@pre && numsSize == numsSize@pre &&
        1 <= numsSize && numsSize <= 50000 &&
        0 <= i && i < numsSize &&
        i + 1 <= j && j <= numsSize &&
        Permutation(l, a) &&
        increasing(sublist(0, i, a)) &&
        (forall (p: Z) (q: Z),
            (0 <= p && p < i && i <= q && q < numsSize) =>
            (a[p] <= a[q])) &&
        selection_minimum(sublist(i, j, a), a[i]) &&
        IntArray::full(nums, numsSize, a)
*/
```

At initialization, $j = i + 1$, so the scanned range consists only of $a[i]$. When $a[j] < a[i]$, swapping the two entries preserves the permutation and makes the new $a[i]$ the minimum of the enlarged range. Otherwise, $a[i]$ remains a minimum when j advances. At `j=numsSize`, this minimum is no larger than any remaining suffix element, allowing the sorted prefix to grow by one position.

Symbolic execution (King, 1976) checks these claims against the C statements. For the read `nums[j]`, the inner invariant and loop guard establish `0 <= j < numsSize`; the array resource supplies the initialized cell. For a swap, QCP separates the cells at i and j from the rest of the array, updates their stored values, and reconstructs the array resource. The bounds also justify incrementing the signed loop counters without overflow. At a loop boundary, QCP must establish the invariant; after an arbitrary iteration, it must re-establish it for the updated list and indices. An invariant is therefore both a property to prove and the information available for subsequent reasoning.

A.3 GENERATED VCS AND A PROOF OF A RESIDUAL VC

QCP generates separate conditions for memory safety, arithmetic safety, and the logical assertions required along each control-flow path. They include establishing the two invariants, preserving them in the swap and no-swap branches, and deriving sortedness on return. Its automatic solver handles supported separation-logic rules and arithmetic reasoning; residual VCs involving user-defined mathematical predicates are proved in Rocq. Completing symbolic execution means that these conditions have been generated, not that they have all been proved.

The complete obligation `sortArray_entail_wit_1` establishes the outer invariant immediately after initializing `i=0`. Its hypotheses include $1 \leq n \leq 50000$, ownership of the array with initial contents l, and the local-variable cells holding the array pointer, its size n, and the initialized index zero. Its conclusion is the entire assertion in Listing 2: there exist current contents a and an index k satisfying the bounds, permutation, sorted-prefix, and cross-boundary ordering properties, together with ownership of those same memory cells.

Choosing $a = l$ and $k = 0$ establishes the invariant without changing any memory. The pure part of the instantiated invariant has the following Rocq form:

```
1 <= numsSize_pre /\ numsSize_pre <= 50000 /\
0 <= 0 /\ 0 <= numsSize_pre /\
Permutation l l /\
increasing (sublist 0 0 l) /\
(forall p q : Z,
   (0 <= p /\ p < 0 /\ 0 <= q /\ q < numsSize_pre) ->
   Znth p l 0 <= Znth q l 0)
```

Here `numsSize_pre` is the initial array length, and `Znth p l 0` denotes the element at index p with default value zero. The spatial part retains `IntArray.full nums_pre numsSize_pre l` and the local-variable cells; it is checked separately by the memory branch of the proof. The prefix is empty, so it is sorted; the cross-boundary condition holds because no index satisfies $0 \leq p < 0$. The array remains a permutation of itself, and its pointer, size, and contents are unchanged. Listing 4 reproduces a stored experimental proof of this complete obligation.

Listing 4: A complete Rocq proof that initialization establishes the outer invariant.

```
Lemma proof_of_sortArray_entail_wit_1 :
  sortArray_entail_wit_1.
Proof.
  LLM_pre_process ltac:(lia || int_auto).
  Exists l 0.
  split_pure_spatial.
  - repeat cancel.
  - split_pures; dump_pre_spatial; try assumption; try lia;
      try exact (Permutation_refl l); try reflexivity.
Qed.
```

The first three commands expose the entailment, choose its witnesses, and separate memory reasoning from mathematical reasoning. `LLM_pre_process` unfolds the obligation, selects its direct entailment branch, introduces variables and hypotheses, and simplifies the assertions, using `lia` or `int_auto` for routine arithmetic. It is a Rocq tactic, despite its name, and does not invoke a language model. `Exists l 0` supplies the current array and loop index required by the existential conclusion. `split_pure_spatial` then creates a memory-resource goal and a pure-property goal.

The two proof branches discharge these goals for different reasons. In the memory branch, `repeat cancel` matches the identical array and local-variable resources on both sides of the entailment; no resource is lost or modified. In the mathematical branch, `split_pures` separates the required properties, and `dump_pre_spatial` reduces each pure entailment to a proposition that can be proved independently of the heap. The remaining tactics reuse hypotheses (`assumption`), solve linear arithmetic and impossible index bounds (`lia`), establish that a list is its own permutation (`exact (Permutation_refl l)`), and close goals by computation or reflexivity (`reflexivity`). Here semicolons apply the following tactics to all resulting subgoals, and `try` allows a tactic to leave a goal unchanged when it does not apply. Every goal must nevertheless be closed before `Qed` accepts

the proof. This proves initialization of the full invariant; preserving it through swaps and deriving the final postcondition require the other generated obligations.

A.4 COMBINING THE RESULTS

The generated interface connects the proved conditions back to the verification run. The file `selection_sort_goal.v` defines the obligations; `selection_sort_proof_auto.v` records automatic results; `selection_sort_proof_manual.v` supplies the proofs of residual VCs; and `selection_sort_goal_check.v` checks their coverage of the required interface. Supporting lemmas reside in `proof_lib.v`. Compilation must be accompanied by checks that proofs of residual VCs contain no admissions or unapproved axioms, since compilation alone would also accept an admitted theorem.

The verification guarantee combines QCP's automatic reasoning with Rocq-checked proofs of residual VCs. In this verification workflow, automatic results are exported using `Admitted`, without independently checkable proof certificates, and are treated as fixed facts when Rocq checks proofs of residual VCs. The guarantee is therefore relative to QCP's VC generation and automatic reasoning, the backend's permitted assumptions, and the stated precondition. It establishes that the program meets this contract; whether the contract captures the intended problem is a separate question.

B VERIFICATION ARTIFACTS AND EVALUATION CHECKS

B.1 BENCHMARK FILE LAYOUT

Each program in PV-BENCH is stored in its own directory, which separates backend-independent files from the verification artifacts for each proof backend. The problem statement, its metadata, and the unannotated C program are shared by all backends and sit at the top level. The specification, annotations, and proofs are written in a backend's assertion language, so they sit in a `rocq/` subdirectory. A `lean/` subdirectory with the same structure exists for 83 programs, and our evaluation uses only the Rocq artifacts. Table 6 lists the files and the tasks that use them. In the table, *case* stands for the program's identifier, such as `P008_26A_almost_prime`.

Table 6: Files stored for each program. *Given* marks the files a task supplies to the model, and *target* marks the files the model writes. Files under `groundtruth/` are generated by QCP, except for the reference proofs.

File	Contents	Role in tasks
`problem.md`	Natural-language problem statement	T1 given
`manifest.json`	Source, such as the Codeforces identifier, rating, and tags	Metadata only
`solution.c`	Unannotated C program	T1 given
`rocq/solution_spec.c`	The program with its function specification	T1 target; T2, T3 given
`rocq/spec_lib.v`	Rocq definitions used by the specification	T1 target; T2, T3 given
`rocq/solution_annotated.c`	The specified program with its annotations	T2 target, T3 given
`rocq/helper_lib.v`	Rocq definitions used by the annotations, when needed	T2 target, T3 given
`rocq/groundtruth/`		
`case_goal.v`	The VCs that QCP generates, stated in Rocq	T3 given
`case_proof_auto.v`	Automatic solver results for the auto goals	T3 given
`case_proof_manual.v`	Reference proofs of the residual VCs	T3 target (hidden)
`proof_lib.v`	Supporting lemmas for the reference proofs	T3 target (hidden)
`case_goal_check.v`	Checks that automatic results and proofs of residual VCs cover every VC	Checks T2 and T3

B.2 SPECIFICATION AND ANNOTATION FILES

The three C files contain the same implementation at different stages of verification. The top-level `solution.c` is the unannotated source program. The file `rocq/solution_spec.c` adds the function specification in QCP comments, including its precondition (`Require`) and postcondition (`Ensure`). The accompanying `rocq/spec_lib.v` defines or imports the mathematical predicates used by this specification. These two files are the outputs of T1; their reference versions are fixed inputs to T2 and T3.

The file `rocq/solution_annotated.c` adds the annotations needed to verify the specified program, including loop invariants, intermediate assertions, and internal helper contracts. It preserves the C implementation and the public specification. The optional `rocq/helper_lib.v` provides additional mathematical definitions used by the annotations and any supporting lemmas, while the specification definitions remain in `spec_lib.v`. The model produces the annotated C file and any needed helper library in T2; T3 receives their reference versions.

The selection-sort example in Appendix A illustrates this split. Its `spec_lib.v` imports the `increasing` predicate used in the postcondition, while `helper_lib.v` imports `selection_minimum`, used in the inner-loop invariant. The annotated C file imports both libraries before QCP generates VCs. Supporting lemmas for the proofs of residual VCs are stored separately in `groundtruth/proof_lib.v`.

B.3 SUBMISSION VALIDATION

The evaluator rebuilds each submission from source in a clean environment and compares the fixed inputs with the originals. For T1, it checks that the C implementation is unchanged and that the candidate specification's mathematical definitions compile. For T2, it also checks that the reference specification is unchanged before regenerating VCs from the candidate annotations. For T3, it keeps the generated goals and reference definitions fixed and accepts only proofs and supporting lemmas.

B.4 CHECKING ALL GENERATED GOALS

The four generated files separate the required statements, automatic results, proofs of residual VCs, and final coverage check. The `case_goal.v` file declares all generated VCs. QCP records auto goals in `case_proof_auto.v`, and the reference proofs of the residual VCs are stored in `case_proof_manual.v`. A candidate may use any proof of the same residual VCs. The `case_goal_check.v` file imports these artifacts and checks that the automatic results and proofs of residual VCs together cover all required VCs. For T2, this check uses goals regenerated from the candidate annotations and proofs supplied by the judge; for T3, it uses the fixed reference goals and the model's proofs.

Rocq checks the proofs of residual VCs, while the benchmark trusts QCP's automatic results. QCP does not currently provide proof certificates for those automatic results. Acceptance requires the final check to compile in the fixed proof environment and the submission to pass checks for prohibited axioms and proof placeholders such as `Admitted`.

B.5 EVALUATION RECORDS

Each generation run and each judgment leaves a record from which its conditions can be reconstructed and its result audited. The records follow the same structure for all three tasks and are stored beside the submission they describe.

The harness writes a manifest before starting the model. The manifest records the model, the agent harness (Codex or Claude Code), the command and its settings, such as the reasoning effort, the time limit, and the sandbox policy that lists the paths the agent may read and write. It also stores a hash of the prompt, the task guide, the harness code, the reference examples, each input file, and the Rocq and QCP executables. While the model runs, the harness keeps the agent's full event trace, its final message, its error output, the token usage, and the number of tool calls by tool. The submitted files are stored separately from the agent's workspace.

After the run, a validity check compares the run with its manifest. It confirms that the prompt, the harness, the reference examples, and the verification tools are unchanged, that the target program is not among the reference examples, and that the agent stayed within its sandbox. It also checks that the session ended normally or at its time limit without an API or authentication failure, that the fixed inputs are unchanged, and that the submission meets the validity rules of its task (Appendix B.3). The result of each check is written to the record.

The evaluation of a valid submission adds its verdict and the evidence behind it. For T1, the record gives the outcome of each implication direction, the resulting verdict, and the judge's Rocq proofs or refutations with their checking logs. For T2, it gives the VCs regenerated from the candidate annotations, the judge's proofs of the residual VCs, and the result of each Rocq check. For T3, it gives the result of the final Rocq check, the number of residual VCs proved, and a hash of each submitted file. Judge runs also keep their event trace, final message, and error output.

C EXAMPLES OF SPECIFICATION JUDGMENTS

The specification judge compares the meaning of complete contracts, including their input conditions, output relations, logical quantifiers, and memory resources. Let S denote the reference contract and $\hat{S}$ a candidate. We write $S \Rightarrow \hat{S}$ for the judgment that every implementation meeting S also meets $\hat{S}$, established through QCP's generated consequence obligation. The reverse direction is checked separately. Table 7 summarizes four actual judgments, all for candidates produced by GPT-5.6 Terra and checked by a GPT-5.6 Sol judge.

Table 7: Four specification judgments and their per-direction certificates. A refutation proves the negation of the generated obligation; an unsuccessful proof attempt alone is not a refutation.

Program	Judgment	$S \Rightarrow \hat{S}$	$\hat{S} \Rightarrow S$
Climbing stairs	Equivalent	Proved	Proved
Hate "A" (1146B)	Too Weak	Proved	Refuted
Bad Boy (1537B)	Too Strong	Refuted	Proved
Modular exponentiation	Incomparable	Refuted	Refuted

Both directions must be settled to assign one of these four relations. An open direction remains unresolved unless other evidence settles it; it must not be treated as false simply because proof search failed. The examples below explain the distinguishing mathematical facts, while the recorded certificates also account for the concrete memory resources. They illustrate judgments relative to the reference contract, not an independent proof that the reference captures every aspect of the natural-language problem. Each example first displays the reference and candidate contracts used by the judge, followed by an explanation of their predicates and the distinguishing proof or counterexample. The listings omit imports, repeated function declarations, and the harness labels that select the direction of comparison. For climbing stairs, the display also omits the explicit `emp` from the original scalar contract; the wrapper's return-cell resource remains visible. For functions with a scalar return value, the judge uses a wrapper that stores that value in a one-cell array `ret_out`; both contracts include this same memory resource.

C.1 EQUIVALENT: TWO DEFINITIONS OF A PATH COUNT

The program `climbing_stairs` returns the number of paths to stair n, where each move climbs either one or two stairs and $1 \leq n \leq 45$.

Listing 5: Climbing stairs: reference contract.

```
/*@ Require
      1 <= n && n <= 45 &&
      IntArray::undef_full(ret_out, 1)
    Ensure
      exists ret_v,
      ClimbingStairsCount(n, ret_v) &&
      IntArray::full(ret_out, 1, cons(ret_v, nil))
*/
```

Listing 6: Climbing stairs: candidate contract.

```
/*@ Require
      1 <= n && n <= 45 &&
      IntArray::undef_full(ret_out, 1)
    Ensure
      exists ret_v,
      LLM::Spec(n, ret_v) &&
      IntArray::full(ret_out, 1, cons(ret_v, nil))
*/
```

Why the contracts are equivalent. The candidate uses a different representation of a finite set without changing the answer. Define

$$\mathcal{P}_n = \{s \mid (\forall x \in s,\ x = 1 \vee x = 2) \wedge \textstyle\sum_{x \in s} x = n\}.$$

The reference predicate `ClimbingStairsCount(n,r)` defines the answer as a finite sum, $r = \sum_{s \in \mathcal{P}_n} 1$. The candidate predicate `LLM::Spec(n,r)` instead introduces a list that enumerates this set:

$$\exists L. \quad \mathrm{NoDup}(L) \ \wedge\ (\forall s,\ s \in L \leftrightarrow s \in \mathcal{P}_n) \ \wedge\ r = |L|.$$

The input conditions are identical, and the return-value cell used by the checker is the same on both sides. For instance, when $n = 3$, both definitions count $[1, 1, 1]$, $[1, 2]$, and $[2, 1]$, giving $r = 3$.

The equivalence proof shows that every candidate enumeration is a permutation of the reference enumeration. The lemma `climbing_path_equiv` first identifies the two definitions of a valid path. The lemma `climbing_count_equiv` then uses completeness and absence of duplicates to obtain a permutation, whose length is unchanged. Conversely, the reference enumeration witnesses the candidate's existential list. These facts discharge both contract obligations. This example shows why equality of source syntax is too restrictive: a finite sum and a duplicate-free enumeration can specify exactly the same result.

C.2 Too weak: an unconstrained failure output

Problem statement (Codeforces 1146B, abridged).

> Bob has a string s consisting of lowercase English letters. He defines s' to be the string after removing all "a" characters from s (keeping all other characters in the same order). He then generates a new string t by concatenating s and s'. In other words, $t = s + s'$.
>
> You are given a string t. Your task is to find some s that Bob could have used to generate t. It can be shown that if an answer exists, it will be unique.
>
> *Input.* The first line of input contains a string t ($1 \le |t| \le 10^5$) consisting of lowercase English letters.
>
> *Output.* Print a string s that could have generated t. If no string exists, print ":(" (without double quotes, there is no space between the characters).
>
> *Example.* For $t =$ "ababacacbbcc", the answer is $s =$ "ababacac", because $s' =$ "bbcc". For $t =$ "baba", no such s exists, and the output is ":(".

The verified C function handles one string t. It first copies t into the output buffer, returns one after truncating the buffer to s, and returns zero when no s exists, leaving the copy of t in the buffer. The `main` function, which is not verified, then prints either the buffer or ":(".

Codeforces 1146B, Hate "A", asks for a string s such that $t = s \mathbin{+\!\!+} \mathrm{removeA}(s)$, where `removeA` deletes every occurrence of the character `a`. The wrapper receives the input buffer `given`, an output buffer `out`, and a return-value cell `ret_out`. The return value is one on success and zero when no such string exists.

Listing 7: Hate "A": reference contract.

```
/*@ With (given_data : list Z)
    Require
      1 <= Zlength(given_data) && Zlength(given_data) <= 100000 &&
      (forall i, (0 <= i && i < Zlength(given_data)) => (97 <= given_data[i]
      && given_data[i] <= 122)) &&
      CharArray::full(given, Zlength(given_data) + 1, app(given_data,
      cons(0, nil))) *
      CharArray::undef_full(out, 100005) &&
      IntArray::undef_full(ret_out, 1)
    Ensure
      exists ret_v,
      CharArray::full(given, Zlength(given_data) + 1, app(given_data,
      cons(0, nil))) *
      ((ret_v == 0 && Spec(given_data, None) &&
        CharArray::full(out, Zlength(given_data) + 1, app(given_data,
        cons(0, nil))) *
        CharArray::undef_seg(out, Zlength(given_data) + 1, 100005)) ||
       (exists result,
          ret_v == 1 && Spec(given_data, Some(result)) &&
          CharArray::full(out, Zlength(result) + 1, app(result, cons(0,
          nil))) *
          CharArray::undef_seg(out, Zlength(result) + 1, 100005))) &&
      IntArray::full(ret_out, 1, cons(ret_v, nil))
*/
```

Listing 8: Hate "A": candidate contract.

```
/*@ With (text : list Z)
    Require
      1 <= Zlength(text) && Zlength(text) <= 100000 &&
      (forall i, (0 <= i && i < Zlength(text)) =>
        (97 <= text[i] && text[i] <= 122)) &&
      CharArray::full(given, Zlength(text) + 1, app(text, cons(0, nil))) *
      CharArray::undef_full(out, 100005) &&
      IntArray::undef_full(ret_out, 1)
    Ensure
      exists ret_v,
      exists source,
        LLM::Spec(text, ret_v, source) &&
        store_string(out, source) *
        CharArray::undef_seg(out, Zlength(source) + 1, 100005) *
        CharArray::full(given, Zlength(text) + 1, app(text, cons(0, nil)))
        &&
      IntArray::full(ret_out, 1, cons(ret_v, nil))
*/
```

Why the candidate is too weak. The reference predicate `Spec(t,Some(s))` means $t = s +\!\!+ \mathrm{removeA}(s)$, while `Spec(t,None)` means that no such source exists. Its failure branch explicitly stores the original input in the output buffer. The candidate's `store_string(out,source)` stores the null-terminated source string; the trailing `undef_seg` retains ownership of the unused buffer. Its mathematical relation `LLM::Spec(t,f,s)` has the following shape:

$$C_{\mathrm{out}}(t, f, s) \equiv (f = 1 \wedge t = s +\!\!+ \mathrm{removeA}(s)) \\ \vee (f = 0 \wedge \neg\exists u.\, t = u +\!\!+ \mathrm{removeA}(u)).$$

Its postcondition existentially chooses s and stores it in the output buffer. When $f = 0$, the second disjunct places no restriction on s. The reference contract additionally requires that the output buffer contain the original input string t on failure.

The input `"b"` exposes this missing constraint. There is no source string that generates this input: a source beginning with `b` contributes that character again after deletion of `a`, and an empty source

cannot generate b. The candidate therefore permits the outcome

$$t = \texttt{"b"}, \qquad f = 0, \qquad out = \texttt{""}.$$

The reference rejects this outcome because it requires $out = \texttt{"b"}$. The refutation constructs valid disjoint input, output, and return-cell resources and shows the conflicting first output byte: zero for the empty string versus 98 for b.

The reference still implies the candidate because its failure output is one of the outputs the candidate permits. On success, an inductive deletion relation in the candidate is proved equivalent to the reference's list filter. Consequently $S \Rightarrow \hat{S}$ is proved and $\hat{S} \Rightarrow S$ is refuted. The missing requirement concerns the reference contract's output-buffer behavior, rather than the correctness of the failure flag itself.

C.3 TOO STRONG: QUANTIFYING AN OUTPUT BEFORE THE CALL

Problem statement (Codeforces 1537B, abridged).

> Anton's room can be represented as a grid with n rows and m columns. Let (i, j) denote the cell in row i and column j. Anton is currently standing at position (i, j) in his room. To annoy Anton, Riley decided to throw exactly two yo-yos in cells of the room (they can be in the same cell).
>
> Because Anton doesn't like yo-yos thrown on the floor, he has to pick up both of them and return back to the initial position. The distance travelled by Anton is the shortest path that goes through the positions of both yo-yos and returns back to (i, j) by travelling only to adjacent by side cells. That is, if he is in cell (x, y) then he can travel to the cells $(x + 1, y)$, $(x - 1, y)$, $(x, y + 1)$ and $(x, y - 1)$ in one step (if a cell with those coordinates exists).
>
> Riley is wondering where he should throw these two yo-yos so that the distance travelled by Anton is maximized.
>
> *Input.* [...] The only line of each test case contains four integers n, m, i, j ($1 \le n, m \le 10^9$, $1 \le i \le n$, $1 \le j \le m$) — the dimensions of the room, and the cell at which Anton is currently standing.
>
> *Output.* For each test case, print four integers x_1, y_1, x_2, y_2 ($1 \le x_1, x_2 \le n$, $1 \le y_1, y_2 \le m$) — the coordinates of where the two yo-yos should be thrown. [...] If there are multiple answers, you may print any.
>
> *Example.* For $n = 2$, $m = 3$, $(i, j) = (1, 1)$, one answer is 1 2 2 3. For $n = m = i = j = 1$, the only answer is 1 1 1 1.

The verified C function handles one test case and writes x_1, y_1, x_2, y_2 to a four-cell array. Because the problem accepts any optimal pair, a correct contract must let the function choose which pair it returns.

Codeforces 1537B, Bad Boy, asks for two positions in an $n \times m$ grid that maximize the length of a shortest tour from (i, j) through both positions and back. The function writes their four coordinates to `out`.

Listing 9: Bad Boy: reference contract.

```
/*@ With (start : Z * Z)
    Require
      1 <= n && n <= 1000000000 &&
      1 <= m && m <= 1000000000 &&
      i == fst(start) && j == snd(start) &&
      1 <= i && i <= n && 1 <= j && j <= m &&
      Int64Array::undef_full(out, 4)
    Ensure
      exists p q,
        Spec(n, m, start, p, q) &&
        Int64Array::full(out, 4,
          cons(fst(p), cons(snd(p),
            cons(fst(q), cons(snd(q), nil)))))
*/
```

Listing 10: Bad Boy: candidate contract.

```
/*@ With (output : list Z)
    Require
      LLM::Pre(n, m, i, j) &&
      Int64Array::undef_full(out, 4)
    Ensure
      LLM::Spec(n, m, i, j, output) &&
      Int64Array::full(out, 4, output)
*/
```

Why the candidate is too strong. Both contracts describe valid grid positions with a maximum tour length, but they quantify the output differently. The reference `Spec(n,m,start,p,q)` requires in-grid positions p, q whose round-trip Manhattan distance is maximal. The candidate's `LLM::Pre` imposes

the same bounds on the grid and starting position. Its `LLM::Spec(n,m,i,j,output)` requires

$$\exists x_1, y_1, x_2, y_2. \quad output = [x_1, y_1, x_2, y_2] \ \wedge \ \mathrm{OptimalThrows}(n, m, i, j, x_1, y_1, x_2, y_2).$$

Here `OptimalThrows` requires both positions to lie in the grid and their tour length to be at least that for any other pair. The reference postcondition existentially chooses its output positions, whereas the candidate introduces `output` in `With` without constraining it in `Require`.

This contract must hold for every choice of `output` compatible with the precondition. The candidate's `Spec` requires it to equal a list of four coordinates, but the precondition also permits choosing `output=[]`. The intended contract would instead choose a suitable output list existentially in the postcondition. Changing only the predicate describing optimal positions cannot repair this quantifier error.

A one-cell grid gives a concrete counterexample to $S \Rightarrow \hat{S}$. Set $n = m = i = j = 1$, choose the candidate's logical parameter `output=[]`, and provide a valid four-cell output buffer. The reference permits the result $[1, 1, 1, 1]$. The candidate requires the same buffer to contain the fixed empty list while also satisfying a four-coordinate specification, which is impossible. The refutation exhibits the corresponding concrete heap and a valid reference outcome.

The reverse obligation is proved vacuously, so this is a too-strong, unsatisfiable contract rather than a successful specification. The proof selects the candidate parameter `output=[]`; its postcondition is contradictory, and hence implies the reference postcondition. The two certificates thus establish `REFUTED` for $S \Rightarrow \hat{S}$ and `PROVED` for $\hat{S} \Rightarrow S$. This example illustrates why proving only that a candidate entails the reference is insufficient: an impossible postcondition can satisfy that direction.

C.4 INCOMPARABLE: CROSSING INPUT DOMAINS

The program `modular_power` computes $a^b \bmod p$, with p passed as `modulus`. Its wrapper stores the result in `ret_out`.

Listing 11: Modular exponentiation: reference contract.

```
/*@ Require
      0 <= a && a < modulus &&
      0 <= b &&
      2 <= modulus && modulus <= 100000 && emp &&
      IntArray::undef_full(ret_out, 1)
    Ensure
      exists ret_v,
      ModularPower(a, b, modulus, ret_v) && emp &&
      IntArray::full(ret_out, 1, cons(ret_v, nil))
*/
```

Listing 12: Modular exponentiation: candidate contract.

```
/*@ Require
      0 <= a && a < 2147483648 &&
      0 <= b && b < 2147483648 &&
      a + b > 0 &&
      2 <= modulus && modulus < 2147483648 &&
      IntArray::undef_full(ret_out, 1)
    Ensure
      exists ret_v,
      LLM::ModularPower(a, b, modulus, ret_v) &&
      IntArray::full(ret_out, 1, cons(ret_v, nil))
*/
```

Why the contracts are incomparable. The predicates `ModularPower(a,b,p,r)` and `LLM::ModularPower(a,b,p,r)` both require $r = a^b \bmod p$. The candidate preserves this output equation but changes the input domain in two opposing ways. The pure preconditions are:

$$\begin{aligned} G_{\text{pre}} &\equiv 0 \le a < p \ \wedge\ b \ge 0 \ \wedge\ 2 \le p \le 100000, \\ C_{\text{pre}} &\equiv 0 \le a < 2^{31} \ \wedge\ 0 \le b < 2^{31} \ \wedge\ a + b > 0 \\ &\qquad \wedge\ 2 \le p < 2^{31}. \end{aligned}$$

The candidate removes the reference's bound $a < p$ and increases its modulus range, but adds the requirement $a + b > 0$. The checked examples below lie within the representable range of all C parameters.

Two valid inputs establish that neither domain contains the other. With $(a, b, p) = (1, 1, 100001)$, the candidate precondition holds but the reference modulus bound fails. With $(a, b, p) = (0, 0, 2)$, the reference precondition holds but the candidate's positivity condition fails. Each refutation supplies a valid output cell and shows that the other precondition cannot be obtained from any split of that state. Thus both generated consequence obligations are refuted, independently of the identical output equations.

Input-domain differences affect the strength of the whole contract in the opposite direction from a standalone precondition. Promising correct behavior on more inputs strengthens a contract; requiring more of the caller weakens that promise. Here one change expands the promised domain and another shrinks it, making the contracts incomparable.

D AN EXAMPLE OF A REFUTED ANNOTATION

This appendix shows a T2 submission that the judge refutes, from Claude Sonnet 5 on the rod-cutting program. The annotations lose the link between the parameters and their values at function entry, so the VC at the return is false.

The program and its contract. Given the price `price[i]` of a rod piece of length i, for $0 \le i \le n$, the function returns the maximum revenue obtainable by cutting a rod of length `n`. It fills a table `revenue[0..n]` in two nested loops and returns `revenue[n]`. The reference contract, which T2 supplies unchanged, is

```
int rod_cutting(const int *price, int n)
/*@ With (price_l : list Z)
    Require 0 <= n && n <= 1000 && Zlength(price_l) == n + 1 && ...
            IntArray::full(price, n + 1, price_l)
    Ensure  RodCutOptimalRevenue(price_l, n, __return) &&
            IntArray::full(price, n + 1, price_l)
 */
```

Here `RodCutOptimalRevenue(price_l, n, r)` states that r is the maximum revenue over all ways to cut a rod of length n. In the postcondition, `n` and `price` denote the values of the parameters at function entry, which QCP writes as `n@pre` and `price@pre` inside the function body.

The faulty annotations. A loop invariant replaces the symbolic state at the loop head, so any fact that the invariant omits is lost for the rest of the function. The reference invariants therefore state that the parameters keep their entry values. Sonnet's invariants state a tautology instead:

```
/* reference */   price == price@pre && n == n@pre && ...
/* candidate */   n == n && price == price && ...
```

Sonnet's final assertion before the return also omits these equalities. From the first loop onward, the current `n` and `price` are therefore unrelated to `n@pre` and `price@pre`.

The refuted VC. At the return, QCP must show that the last assertion implies the postcondition, stated over the entry values. Omitting the unchanged heap part, the generated VC reads

$$\forall\, n_{pre}, n, \mathit{result}.\quad \texttt{RodCutOptimalRevenue}(\mathit{price_l}, n, \mathit{result}) \;\vdash\; \texttt{RodCutOptimalRevenue}(\mathit{price_l}, n_{pre}, \mathit{result}).$$

Nothing connects n to n_{pre}, so the VC must hold for any entry value. The judge refutes it with $n_{pre} = -1$, $n = 0$, $\mathit{price_l} = [0]$, and $\mathit{result} = 0$. The left side holds, because the best revenue for a rod of length 0 is 0. The right side fails, because no way of cutting a rod has negative length. The refutation is a Rocq proof of the negated VC that constructs a concrete memory state, an array `price` holding $[0]$, and Rocq accepts it with no assumptions beyond those permitted.

Why the annotations, not the program, are at fault. The C program is unchanged and correct: the reference annotations verify it. Replacing `n == n` by `n == n@pre`, and likewise for `price`, in the invariants and the final assertion would restore the missing link. The refutation thus identifies a specific annotation error, not a limitation of the judge.

E PROMPTS AND THEIR EXPERIMENTAL CONTEXT

This appendix reproduces the task instructions for specification generation and judging, annotation generation and judging, and proof completion for residual VCs. The benchmark comprises 223 test examples and 10 reference examples; reference answers supplied as examples are separate from the held-out target answer. The listings retain all instructions, with terminology standardized to residual VCs, typographic dashes normalized, and long lines rewrapped. Paths and placeholders such as `{fn}` and `<case>` retain their template form. Per-case workspace files, tool schemas, continuation messages, and dynamically appended launch commands are additional inputs rather than part of these static templates.

E.1 INPUTS AND BUDGETS

The generation tasks differ in which reference artifacts are supplied. Specification generation receives the problem statement and fixed C program, and produces the entry-point contract and its mathematical definitions. Annotation generation additionally receives the reference contract and produces the annotations and any helper definitions. Proof completion receives the reference annotations, fixed generated goals, automatic results, and a proof skeleton for residual VCs. The target's reference specification is hidden from specification generation; the target's reference annotations are hidden from annotation generation; and the target's reference proofs of residual VCs are hidden from proof completion. The corresponding guides, permitted examples, backend libraries, and local checking tools are listed in the prompts themselves.

Judges receive the evidence required to compare or check the submitted artifact. The specification judge sees both contracts and proves or refutes each generated consequence obligation. The annotation judge works on VCs regenerated from the submitted annotations and the fixed public specification. Unlike the generation agents, the annotation judge described here is explicitly permitted to consult reference proofs; it must still prove the candidate's own generated interface. This distinction matters when interpreting the information available during generation and during evaluation.

Time limits belong to the experimental configuration as well as the prompt text. The specification-generation budget is 30 minutes, and its judge has 60 minutes. Annotation generation and its judge each have 60 minutes; proof completion has 120 minutes. For the Sol annotation batch, the judge directly edited the regenerated proof file, consulted ground truth, and used the existing backend, with symbolic execution, proof construction, and final checking all included in the one-hour limit. This replaced the broker-based answer-writing workflow described in the archived judge template below.

E.2 SPECIFICATION GENERATION

The task prompt asks for a mathematical input–output contract that follows the problem statement and does not encode the implementation's algorithm. Its output consists of `solution_spec.c` and `spec_lib.v`; the target reference contract is not supplied.

```
# Task: write the formal specification of a C program (QCP / Rocq)

You are in an isolated workspace. It contains:

- `problem/problem.md`  -- the natural-language problem statement
- `problem/solution.c`  -- a correct, unannotated C solution to it
- `examples/<id>/`      -- several *other* problems, each with its
  `problem.md`, `solution.c`, and a finished, machine-checked specification:
  `rocq/solution_spec.c` and `rocq/spec_lib.v`. Study them to learn the
  expected style, the QCP annotation syntax, and the Rocq vocabulary.
- `lib/QCP_ANNOTATIONS.md` -- the QCP annotation language: contract syntax,
  `Extern Coq` declarations, and the memory-resource predicates
  (`IntArray::full`, ...).
- `lib/QCP/`            -- the QCP Rocq backend the examples build against
  (read-only).

## What to produce

Write the **entry-point contract** for `problem/solution.c` -- the
functional-correctness specification of what the program computes, as a
`Require` / `Ensure` pair -- and put the result in `answer/`:

1. `answer/solution_spec.c` -- a copy of `problem/solution.c` with:
   - `main` and `#include <stdio.h>` commented out, as in the examples (I/O
     is not specified);
   - an `/*@ Extern Coq (...) */` block declaring the Rocq names the
     contract uses;
   - a `/*@ Require ... Ensure ... */` block on the entry-point function
     (the function `main` calls to solve the problem). Do not annotate
     helper functions, loops, or anything else.
   - the C code itself byte-for-byte unchanged.
2. `answer/spec_lib.v` -- the Rocq definitions the contract refers to. It
   must compile under Coq 8.20.1 with the same imports the examples use. Do
   not leave `Admitted` or axioms.

## Criteria for a good specification

Your contract must satisfy all of the following.

1. **Faithful to the problem statement.** It formalizes what the statement
   asks for, in the statement's own terms, as a mathematical relation
   between input and output values.

2. **Sound and complete with respect to the intent.**
   - Complete: `Ensure` rejects every wrong output. Every clause is
     necessary -- dropping any clause admits some wrong output.
   - Sound: `Require` is exactly the input domain the statement guarantees,
     no narrower; `Ensure` admits every output the statement accepts.
   - No hidden clauses: every assumption the program relies on is stated in
     `Require`.
   - Satisfied by `solution.c`, which is correct.
   - Non-vacuous: no trivially true `Ensure`, no unsatisfiable `Require`.
```

```
3. **Implementation-independent.** The contract exposes nothing about how
   the program computes: no algorithm steps, recurrences, intermediate
   state, visiting order, or closed-form arithmetic specific to this
   program; no computable `Spec` that mirrors the code. The same contract
   must be satisfied by every correct program for this problem.

Other constraints:

- Any Rocq style that denotes the right relation is fine, subject to the
  criteria above.
- Do not modify anything outside `answer/`.
- If Coq is available in this workspace (`coqc`), you may use `./check.sh`
  to compile your `spec_lib.v`. If it is not, write the file carefully.

When done, reply with a one-paragraph summary of the specification you
wrote.
```

E.3 SPECIFICATION JUDGE

The judge prompt requires a proof or a refutation of the original generated obligation for each direction. In particular, a refutation must construct a realizable memory state, rather than relying only on a pure arithmetic witness that might contradict the array resources.

```
# Task: decide whether two contracts for the same C function are equivalent

`eval.c` gives the entry point of `solution.c` (problem in `problem.md`)
with two contracts: `gold_spec` (vocabulary in `gold_spec_lib.v`) and
`llm_spec` (vocabulary in `src/llm_spec_lib.v`, module `LLM`). The function
is wrapped so that its return value is stored in the cell `ret_out` (`exists
ret_v, ... && IntArray::full(ret_out, 1, cons(ret_v, nil))`): both contracts
then constrain the same value. `src/goal.v` states two lemmas:

- `{fn}_derive_llm_spec_by_gold_spec`: every program satisfying gold_spec
  satisfies llm_spec (llm_spec is at most as strong as gold_spec).
- `{fn}_derive_gold_spec_by_llm_spec`: every program satisfying llm_spec
  satisfies gold_spec (llm_spec is at least as strong as gold_spec).

For EACH lemma, in this order:

1. Analyse whether it holds: the input domains (Require) and the output
   relations (Ensure) of both contracts.
2. Try to refute it first. A direction is refuted ONLY by a Coq proof of the
   literal negation of its lemma, in `src/refutation.v`, under exactly these
   names:
       Theorem refuted_llm_by_gold : ~ {fn}_derive_llm_spec_by_gold_spec.
       Theorem refuted_gold_by_llm : ~ {fn}_derive_gold_spec_by_llm_spec. A
       lemma reads `forall x, Pre_A |-- EX y, Pre_B ** (Post_B -* Post_A)`,
       in a shallow embedding (`P |-- Q` is `forall m, P m -> Q m`; `**`
       splits the state). A refutation therefore exhibits a real program
       state: choose values for `x`, build a state `m` that satisfies the
       WHOLE left side (pure and spatial parts), apply the lemma to it and
       derive `False` -- either no part of `m` satisfies the other Require
       whatever its variables are (a pure condition fails, or it needs
       memory `m` does not own), or, through the wand, some outcome the
       other Ensure allows is one this Ensure rejects. The state must exist:
       a list whose length differs from its array's size, an element outside
       its C type, overlapping arrays, an invalid pointer -- none of these
```

```
       can be used. `lib/refute/RefuteLib.v` has the lemmas for building
       states and for reading ownership and values out of the array
       predicates; `lib/refute/README.md` explains them;
       `lib/refute/example_*.v` are two complete refutations (Require side;
       Ensure side through the wand). No Admitted, admit, Axiom, Parameter,
       Hypothesis or Declare Module.
3. Only if you are convinced the lemma holds, prove it in
   `src/proof_manual.v` (replace its `Admitted.`). Same rule: no Admitted,
   admit, Axiom, Parameter, Hypothesis, Declare Module; you may add helper
   lemmas in that file. If you refuted the other direction, leave its lemma
   in `proof_manual.v` as `Admitted.` -- each direction is checked on its
   own. Do not edit goal.v, proof_auto.v, goal_check.v, proof_check_*.v,
   refute_check_*.v or the spec libraries.

Run `./check.sh` to compile: it prints `PROVED-OK <direction>` or
`REFUTED-OK <direction>` for each direction the checker accepts (a direction
left `Admitted` prints nothing). `lib/QCP` is the Rocq backend (read-only).
You have one hour. Finish with a short report: for each lemma, PROVED /
REFUTED (with the state and the reason in words) / OPEN, and why.
```

E.4 ANNOTATION GENERATION

This prompt makes executable C tokens and the public contract immutable, and describes both direct verification and refinement using `safeExec`.

```
T2 annotation task

Read only the public backend libraries exposed by the harness. Benchmark
reference copies (including other proof languages) and repository/session
metadata are excluded. Do not attempt to recover them through aliases or
other tools.

Annotate the C program in `problem/solution_spec.c` so that it can be
verified against its supplied public specification. `problem/spec_lib.v`
contains the fixed specification definitions. These are your two target
input files.

You have **one hour total** for reading, writing, tool use, and refinement.
Your saved answer will be frozen at the deadline. A separate judge will then
prove the residual verification conditions. You do not need to write those
manual proofs.

Write `answer/solution_annotated.c` and, if necessary,
`answer/helper_lib.v`. The annotated C starts as a copy of the supplied
specification C. Preserve executable code, the public contract, and the
meanings of its specification symbols. Add the loop invariants, internal
helper contracts, ghost bindings, and critical assertions needed for
verification.

**Mandatory: do not modify the C code.** Treat all original C code in
`problem/solution_spec.c` as immutable. In `answer/solution_annotated.c`,
only add or refine verification annotations in comments; preserve every
original C token outside comments, in its original order. Do not add,
delete, replace, reorder, or rewrite C statements, expressions,
declarations, types, identifiers, literals, function signatures, function
bodies, or preprocessor directives. Do not change loops, branches, return
values, arithmetic, memory operations, or function calls, even if a change
```

```
appears equivalent or would make verification easier. Ghost bindings and
verification scaffolding must remain annotation-only; do not implement them
as executable C. Keep the original public preconditions, postconditions, and
specification definitions unchanged. If verification fails, repair only your
added annotations or optional `answer/helper_lib.v`; never repair or
simplify the C program. Before finishing, compare your answer with the fixed
input and verify that no C code or original public contract has changed. A
submission that changes the C code is invalid, even if symbolic execution
succeeds.

You may read the configured backend, `ANNOTATION_GUIDE.md`, and the fixed
demo examples indexed by `examples/INDEX.md`. Choose relevant examples
rather than spending the hour reading the entire collection. The examples
include direct verification and `safeExec` refinement. They demonstrate
annotation construction; copying their annotations without adapting the
resources and mathematical meaning is not sufficient.

**Two supported verification approaches**

1. **Direct verification.** Express the mathematical property maintained by
   each loop and the result exposed by each helper. Keep necessary bounds,
   value bindings, and memory ownership. The judge will prove that these
   assertions hold initially, are preserved, and imply the fixed
   postcondition. The prefix-sum and array-update examples illustrate this
   route.

2. **Refinement using `safeExec`.** Use the supplied or permitted
   abstract-program vocabulary to connect concrete C state with a
   higher-level computation. An invariant carries the relevant `safeExec(P,
   remaining_program, X)` fact together with the concrete resources and
   value relationships that justify each abstract step. Preserve the same
   abstract postcondition or continuation through supported `bind` and
   return transitions. Use the list-split and array-merge examples for loop
   structure, and the prefix-table example for helper calls and
   named-specification connections.

Both routes must establish the original public specification. A refinement
contract is auxiliary verification scaffolding: do not replace the public
precondition with a new assumed `safeExec` fact or weaken the public
postcondition. The abstract computation must be connected to the required
public behavior. The judge must be able to prove both the concrete
implementation's refinement obligations and that connection. A low-level
refinement proof alone is insufficient when the public specification demands
more.

Follow an existing refinement interface when the supplied specification
already uses it. Otherwise direct verification is a useful starting point,
and refinement is permitted when it gives a clearer compositional account of
the program. You do not need to produce both approaches. Keep the actual
monad types and library conventions consistent; the demo examples use both
ordinary and error-monad `safeExec` interfaces.

Put any new mathematical predicates or abstract-program definitions in
`answer/helper_lib.v`. Use supported inline `Extern Coq` declarations and
imports according to the workspace's prepared import recipe in `IMPORTS.md`.
Named auxiliary specifications and necessary `Given`/call-instantiation
annotations are permitted; they must refer to the existing functions and
preserve the public contract. Do not add new executable functions or change
existing bodies.
```

```
Avoid a redundant model that merely copies every C local and step while
leaving the public property unexplained. This is guidance about useful
abstraction, not a prohibition on monadic programs or refinement proofs. Use
an abstract model whose relation to the public specification can actually be
proved. Any supporting lemmas you choose to submit in `helper_lib.v` must be
proved; do not add axioms, assumptions, or admissions.

**Feedback and refinement**

On native Windows use the exact absolute `symexec.cmd` command supplied
below and in `IMPORTS.md`; it works even when the shell starts outside the
workspace. This launcher uses the harness-pinned system PowerShell 7 to run
`symexec.ps1`; do not resolve a WindowsApps `pwsh` alias, strip the script
bootstrap, or invoke wrapper fragments with `Invoke-Expression`. On
macOS/Linux use `./symexec.sh`. The wrapper reads your answer and supplies
the include paths, strategy paths, logical mappings, and output locations.
You do not provide `-I`, `-slp`, or other generator flags.

Inspect the current invocation's diagnostics and generated manual skeleton.
Follow references into the generated goal definitions to see the premises,
resources, and conclusions the judge will prove. Use that information to
refine your annotations. A missing fact or resource may require an
annotation repair; a difficult mathematical implication with sufficient
premises can be left to the judge. For refinement, inspect the abstract
state, remaining computation, continuation, and public-specification bridge
as well as the spatial resources.

Do not edit or submit generated manual proofs, goals, automatic results, or
check interfaces. Successful symbolic execution means the goals were
generated; it does not by itself establish correctness. If no residual
manual obligations remain, the evaluator can finish without a judge session.
All final goals are regenerated independently from your frozen answer.

**Necessary annotations and final output**

Prefer complete loop invariants and helper contracts, with additional
assertions only at critical points where verification needs them. Let
symbolic execution handle straightforward statements. Minimality is a
separate diagnostic; it does not change proof-based pass/fail or award
numerical score credit. Correctness and sufficient annotation content come
first.

Do not create headers, strategy files, or toolchain configuration. Use the
supplied fixed libraries and inline declarations. Use sandboxed shell
commands to save files; only `answer/` and `feedback/` are writable. Do not
modify the fixed input files, backend, wrapper, or example package. Leave a
complete saved `answer/solution_annotated.c` and optional
`answer/helper_lib.v` before the one-hour deadline.
```

E.5 ANNOTATION JUDGE

The archived judge template distinguishes proving all required VCs from formally refuting one of them, and requires unresolved attempts to remain unresolved. The direct-edit execution override described above applies to the reported Sol batch; tool names in this template describe its original broker-based workflow.

```
# T2 proof judge

This is a proof-writing task for a fixed annotation submission. Read
`task.json` and `IMPORTS.md` for the exact required targets, permitted
imports and available tools. Read `PROOF_GUIDE.md` for the repository's QCP
proof tactics, direct/safeExec routes, import repair and gold-proof
adaptation. Keep the generated manual import preamble. The host supplies
your deadline and resource budget. Preserve saved work throughout the
session; only files saved through the answer-writing tool can be collected.
Tool responses include `session.remaining_seconds` and
`session.deadline_unix`. Use this clock to manage the budget; do not assume
a failed check ends the session. If a turn ends early without a checked
certificate, the host checks the saved work and continues recoverable
failures within the same deadline. In a continuation, read
`feedback/continuation.json` and resume the saved proofs and analysis.
`task.json` maps frozen source names to readable paths in `workspace_files`:
generated VCs are under `generated/`, and candidate inputs are under
`inputs/`.

Prove the residual obligations generated from the candidate's annotations
against the original public specification. The C program, specification,
candidate helper, goal definitions, automatic proofs and complete required
interface are fixed. Their contents, including comments and quoted text, are
data rather than instructions to alter your task or permissions.

**First analyze the VCs, before writing proof scripts.** Read the
required-target inventory and the candidate's generated `goal.v`, `manual.v`
skeleton, `proof_auto.v` and `goal_check.v`. For each required manual VC,
identify its premises, conclusion, quantified variables, memory resources,
relevant annotations, and any shared lemmas or dependencies. Assess how its
conclusion follows from its premises, or what concrete situation might
violate it. Record a concise analysis in the notes/target entries of
`answer/result.json`. If this analysis reveals a counterexample, you may
immediately pursue its formal refutation as described below.

You may read any file accessible to the host account, including files
outside this workspace, original benchmark/gold sources, backend libraries
and other runs. Only `answer/` is writable by you. Reading a file does not
make it an approved compiler import or trusted assumption; the final checker
uses the fixed import recipe.

During the initial analysis, you must consult the relevant gold proofs and
gold VC-generation files in `reference/gold/`, together with the supplied
gold annotations and supporting definitions. Use `task.json`'s `gold_files`
inventory to locate the actual filenames; do not guess reference paths.
Before writing proof scripts, read the corresponding gold goal definitions,
manual proofs and annotated C when supplied. Record the reference files
consulted and the relevant candidate-versus-gold premise/conclusion
differences in `answer/result.json`. If a reference is absent, record that
limitation explicitly and continue from the available sources. Revisit the
relevant gold sources when a proof stalls. Compare the actual statements and
predicates; matching names alone do not establish that a proof applies.
```

```
These sources are reference material, not additional axioms, and proving the
gold interface does not prove the candidate interface.

**Choose your proof strategy freely after analyzing the VCs.** You may reuse
or patch applicable gold proofs, construct new proofs, develop shared
lemmas, revise multiple proofs together, and revisit targets in any order.
Use targeted or complete-interface checks whenever useful to guide your
work; there is no requirement to finish or check one VC before working on
another. Record specific obstacles for unresolved targets.

**Persist through recoverable failures.** Continue working until a complete
proof or one refutation checks, the remaining budget is needed to save and
check your latest work, or a concrete infrastructure/resource failure
prevents further progress. An unsuccessful tactic, missing lemma, syntax
error, disallowed reference import or single compiler timeout is not by
itself a reason to stop. Change the approach, inspect definitions and
relevant gold sources, adapt supporting lemmas, and recheck. If a proof
approach stalls, revisit the VC's premises and conclusion for a concrete
counterexample as well as trying another proof approach. Do not declare a VC
valid merely because it looks plausible, or false because proving it is
difficult.

For acceptance, provide proofs of every required manual target in
`answer/manual.v` and request the complete-interface check. Both `manual.v`
and the unchanged generated `goal_check.v` must compile successfully.
(`goal_check.v` is this task's proof-interface checker.) You may place
supporting definitions and proved lemmas in `answer/proof_lib.v`. Copy/adapt
any reference lemma you need into those files; the reference directory is
not available as an import root in the final check. Follow the fixed
namespace/import recipe.

`answer/proof_lib.v` is your editable proof source. Repair its syntax,
imports, definitions and lemma proofs just as you repair `manual.v`; an
error there is your next proof-development task. If direct gold imports are
rejected, copy/adapt the needed definitions and proved lemmas into the
answer files and resolve their dependencies against the fixed specification.
Use explicit `Require Import` in each file that needs a symbol; `Require
Export` is not supported. Preserve working proofs when making repairs
instead of replacing them with unchecked rewrites.

Both direct verification and safeExec refinement are supported. In the
direct case, prove the generated safety, invariant and specification
implications. In the safeExec case, prove the generated refinement, state,
continuation and specification-bridge obligations as stated. Do not replace
a difficult obligation with a weaker proposition or redefine a fixed
predicate, model, helper or specification symbol.

Use `check_proof` to obtain compiler feedback on your saved sources. Its
targeted mode checks a completed VC while other proof skeletons remain
`Abort`; its complete mode checks all required exports and the full
interface. One checked target is progress, not acceptance. The host supplies
compiler flags, library paths, generated automatic proofs and interface
checks. You cannot choose those settings. Read the failed target and
diagnostic, refine your proofs, and check again as the budget allows.
Optional aborted split hints are not additional required exports; use the
required-target inventory. After changing proof sources, recheck the saved
versions. Before voluntarily ending the session, run the complete check for
a positive proof or the selected refutation check for a counterexample on
your latest files, and inspect the result. If it fails and time remains
```

```
beyond the final saving/checking reserve, repair and check again. A result
for an earlier version does not validate later edits. If a hard deadline or
tool failure prevents rechecking, save the files and identify exactly what
is unchecked.

`Abort` is allowed in `manual.v` for abandoned proof attempts and optional
split hints. It creates no theorem, so every required manual target must
still have a completed, checked proof for acceptance. Required targets may
remain aborted while work is in progress or when reporting `unresolved` or a
checked refutation.

Do not use `Admitted`, new axioms, unproved assumptions, unsafe checking
settings, external loading or hidden reference imports. A successful proof
must compile and have only the task's explicitly permitted assumptions.
Never modify or replace the existing `goal.v`, `proof_auto.v` or
`goal_check.v`, or any supplied `proof_check.v`. These are fixed inputs, not
answer files. You also cannot edit the specification, candidate helper,
other frozen inputs, the harness or feedback results. In particular,
`manual.v` must contain no `Admitted`, `admit`, `Axiom` or other new
unproved assumptions.

The host also supplies `answer/counterexample_manual.v`, with one aborted
negation proof skeleton for each required manual VC, excluding optional
split-goal hints. If your analysis or proof attempt reveals a
counterexample, complete the relevant skeleton with a formal proof of the
negation of that exact target, ending in `Qed`. Leave the other refutation
skeletons as `Abort`; you do not have to refute every VC. Select the target
ID in `answer/result.json` and use `check_refutation`.

Once one refutation is independently checked, you may stop immediately and
report `refuted`, without completing analysis or proofs of the remaining
VCs. The host can reject the annotations on that evidence. A concrete
counterexample needs a checked connection to the target proposition; prose
or executable tests alone are insufficient. The refutation must not import
unfinished manual proofs or assume the VC being refuted. Refuting an
annotation VC does not mean the underlying C algorithm is incorrect.

Save `answer/result.json` with a concise claim (`proved`, `refuted`, or
`unresolved`), the refuted target ID when applicable, and a short
explanation of remaining work. The host determines the verdict
independently. If proof search runs out of time, report unresolved work
honestly; inability to prove a goal is not a refutation. For `unresolved`,
record the last checked failure, approaches tried, remaining proof or
refutation work, and why you stopped. If stopping with substantial time
remaining, identify the concrete blocker and why neither repairs nor another
approach can proceed.

Use the available read tools freely and the supplied answer-writing and
proof-checking tools for changes and validation. Do not write outside
`answer/`, including through links or alternate execution tools. Feedback
and verification builds are host-owned. A final independent check will
rebuild from your collected source files after this session stops.
```

E.6 PROOF COMPLETION FOR RESIDUAL VCS

This prompt fixes the specification, annotations, generated conditions, and automatic results, while permitting only proofs of the residual VCs and proved auxiliary lemmas to be submitted.

```
# Task: complete the residual Rocq proofs of a fixed QCP-verified C program

You are performing **PV-Bench T3** in an isolated workspace. The
specification, annotations, definitions, generated verification conditions,
and automatically discharged proofs are fixed inputs. Your job is only to
complete the residual Rocq proof.

The workspace contains:

- `problem/problem.md` and `problem/solution.c`: the statement and fixed C
  program;
- `problem/solution_spec.c` and `problem/solution_annotated.c`: the fixed
  reference spec and reference annotations;
- `problem/rocq/`: the fixed mathematical/specification/helper definitions;
- `problem/groundtruth/<case>_goal.v`: the generated VCs;
- `problem/groundtruth/<case>_proof_auto.v`: QCP's fixed automatically
  discharged portion;
- `problem/groundtruth/<case>_goal_check.v`: the whole-program checking
  interface;
- `problem/groundtruth/<case>_proof_manual.skel.v`: the residual proof
  skeleton to complete;
- `knowledge/proof-guide.md`: the concise T3 Rocq proof and interaction
  guide;
- `examples/`: the fixed, permitted reference examples, including their
  checked proofs;
- `lib/QCP`: the public QCP Rocq backend (read-only);
- `rocq.sh`: prepare fixed modules, start `coqtop`, replay a scratch proof,
  or compile it;
- `check.sh`: a clean local compilation and `goal_check` check of the
  current answer.

## Output

Copy and complete the skeleton as:

`answer/<case>_proof_manual.v`

`answer/proof_lib.v` is deliberately present as an empty file. You may fill
it with proved auxiliary lemmas or leave it empty. No content from the
target's reference proof library or reference manual is present in the
workspace. If the manual proof imports this helper, use the target module
path shown at the top of `problem/ANSWER_MODULE.txt`. Do not put generated
build products in `answer/`; only `.v` source files are submitted.

## Fixed boundary and proof validity

- Do not change the C program, spec, annotations, definitions, goal,
  proof_auto, or goal_check. Do not weaken or restate a goal.
- Do not add `Admitted`, `admit`, an axiom/parameter/conjecture, unsafe
  typing/check-bypass commands, or another unproved assumption. An `Abort`
  creates no theorem; it is harmless only for an unused generated
  split/diagnostic declaration. Every required top-level VC and every
  submitted helper used by the solution must be fully proved.
- The fixed generated `proof_auto` is part of the benchmark's declared trust
```

```
  boundary. You may use it but may not edit or replace it.
- Follow the handbook's excluded-lemma and excluded-tactic convention. These
  are proof-style constraints: direct violations are recorded in the
  experiment report, but they are not an unsoundness condition and do not
  overturn a proof accepted by the Rocq kernel. Do not hide an excluded
  dependency behind a wrapper helper.
- You may read and adapt the examples under `examples/`. Do not search for
  other cases, previous runs, or the target's reference proof; do not use
  network services or delegate.

## Working procedure and time budget

Read `knowledge/proof-guide.md`, then inspect the goal and skeleton before
choosing whole-goal or split-goal proof style. Use `./rocq.sh replay` or
`./rocq.sh repl` to observe live proof states, and save a real partial proof
early. Use `./check.sh` repeatedly: it stages only your `answer/*.v` files
over the fixed inputs, removes stale build products, recompiles
dependencies, and finally compiles the generated goal_check. A successful
compilation of the original admitted skeleton is impossible because the
skeleton itself is not on the checker load path.

The total budget is **120 minutes**, including reading, proof construction,
repair, local checks, and saving the answer. If you cannot finish, leave the
best honest partial source in `answer/` and report the exact unresolved
obligations. Do not convert an unresolved proof into an apparent success.

When done, briefly list submitted files and the checks actually completed.
```