Discrete Mathematics

(for Computer Science)

Zhaoguo Wang, Zeyu Mi

IPADS

Why Choose CDM?

What makes you distinguished?

Understand the “laws of physics” in computer science

Answer the fundamental questions behind computation

Tools change.
Fundamental principles last.

Questions To Be Answered In CDM

What problems can computers solve?

How can we know that a program is correct?

Coding Agents and Human Programmers

Can Coding Agents Replace Programmers?

CODING AGENT

Reads the repository

Writes and edits code

Runs tests

Opens a pull request

+

HUMAN PROGRAMMER

Defines the goal

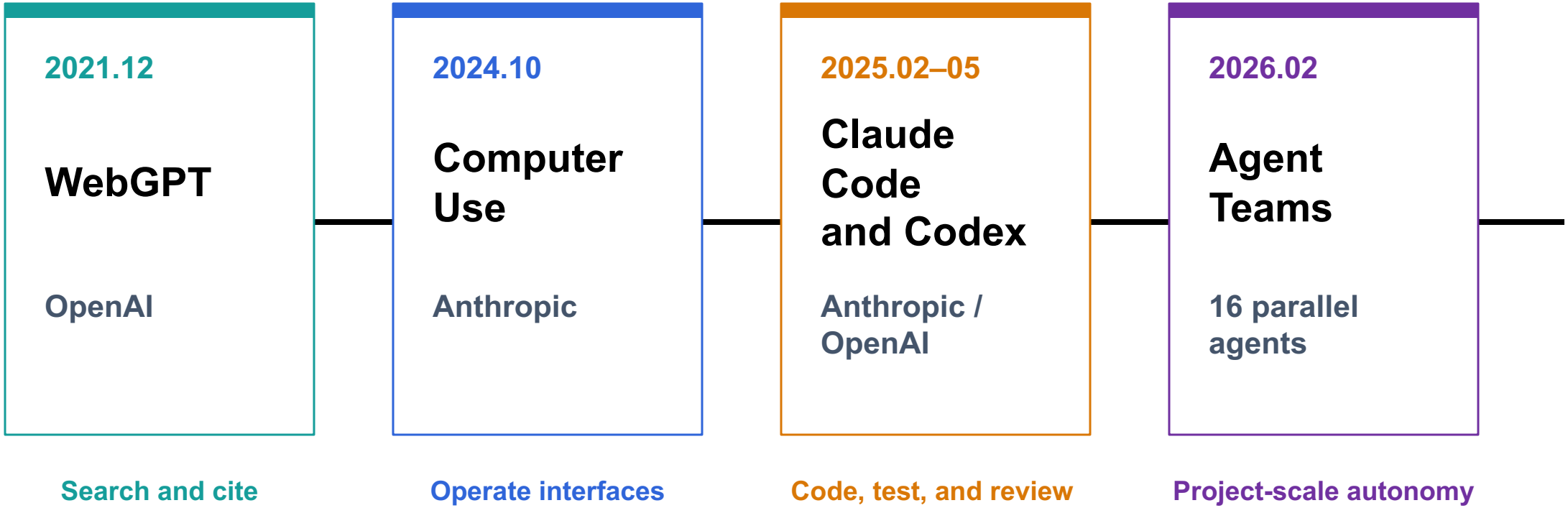
Supplies context and constraints

Reviews evidence

Accepts responsibility

The question is changing from replacement to division of responsibility.

Roadmap of Coding Agent



Agents progressed from retrieving information to changing entire software projects.

Why Did Coding Agents Improve So Quickly?

01

Training on code

Models learn from code, documentation, issues, reviews, and test logs.

02

Larger context

Agents can read more of a repository and keep relevant files in view.

03

Context engineering

Goals, constraints, errors, and test results remain available during the task.

04

Tools and skills

Agents run commands and reuse documented workflows instead of only producing text.

Programming provides immediate feedback: build, run, test, revise.

A C Compiler Built by an Agent Team

16	2 weeks	nearly $20,000	~100K
parallel agents			lines of code

The compiler can build Linux 6.9 on x86, ARM, and RISC-V.

It passes about 99% of most compiler test suites.

This is an impressive engineering result.

It Passes the Tests. Is It Correct?

WHAT THE TESTS SHOW

Linux can be built

Most test suites pass

Many real programs compile

WHAT USERS STILL FIND

Unsupported language features

Incorrect edge-case behavior

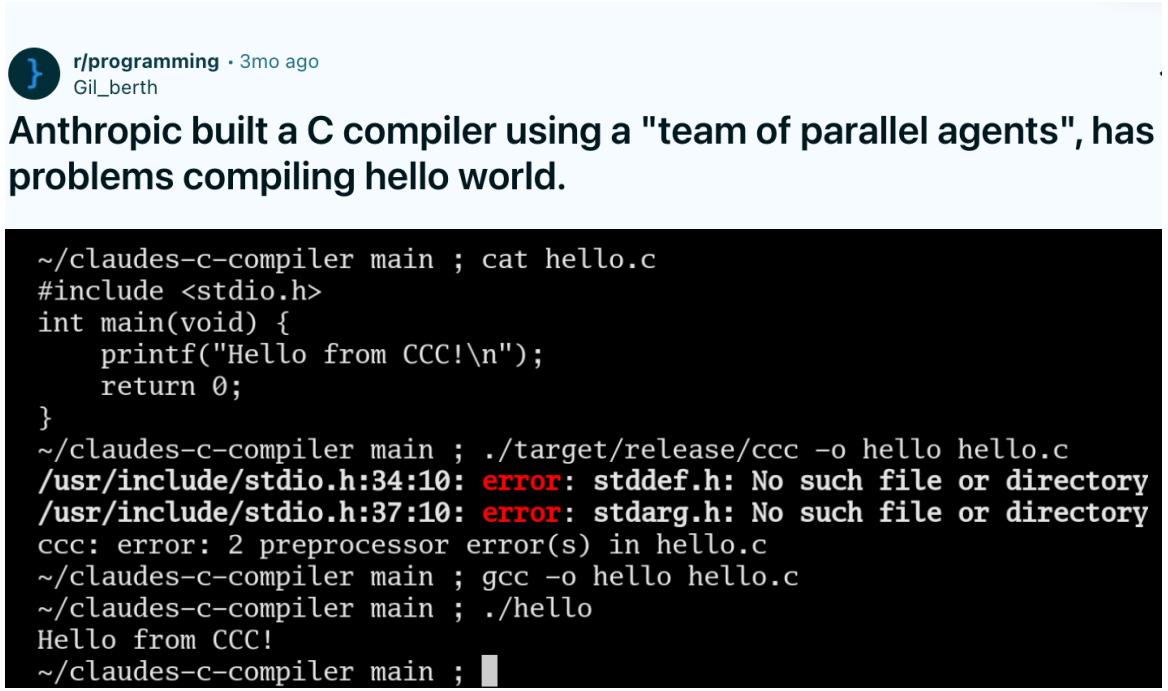
New bug reports and fixes

Testing can reveal bugs. Passing tests does not prove their absence.

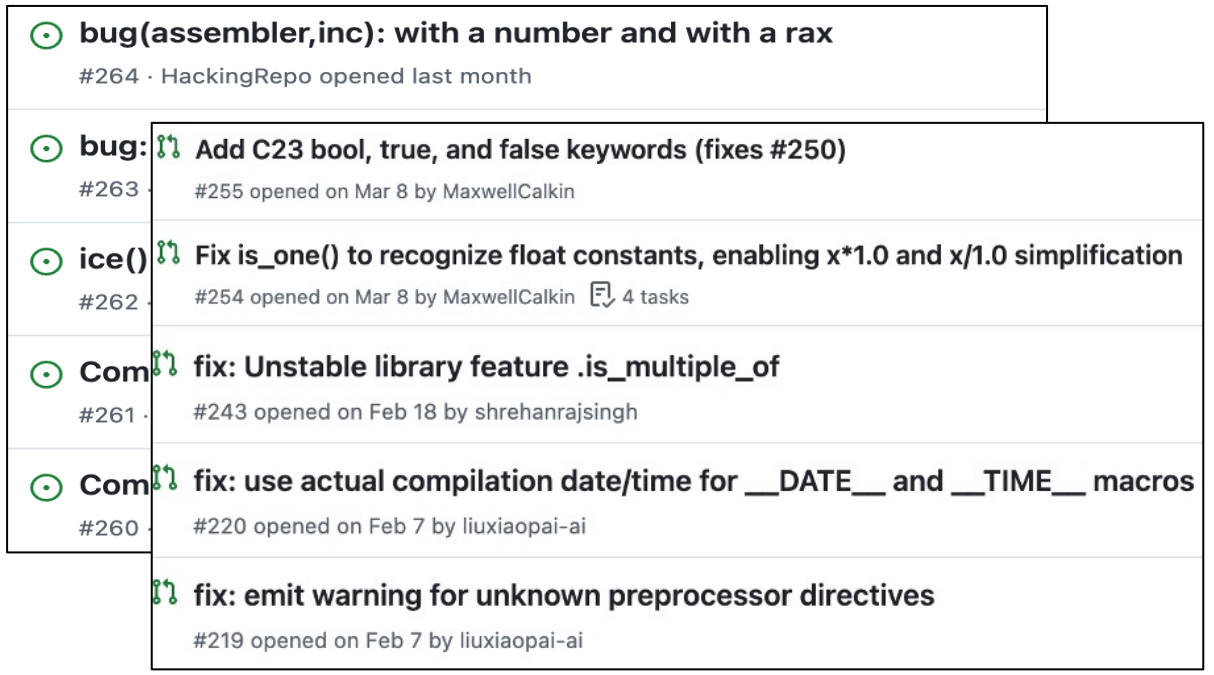
It Passes the Tests. Is It Correct?

Claude C Compiler passes many test suites and even compile Linux correctly

However, there are still many bugs



CCC cannot compile Hello World



Many bug Issues and PRs in CCC repo

https://www.reddit.com/r/programming/comments/1qwzyu4/anthropic_built_a_c_compiler_using_a_team_of/
https://github.com/anthropics/claudes-c-compiler/issues/1

Coding Agents Change What Programmers Do

Write

Produce most code by hand

>

Delegate

Describe tasks and supervise agents

>

Specify and verify

State what must hold and check the evidence

Producing code becomes easier.

Specifying and checking it becomes more important.

The Correctness Problem Predates AI

1949 **Alan Turing** *"Checking a Large Routine"*

```
int factorial(int n) {
  int u = 1;
  for (int r = 0; r < n; r++) {
    int v = u;
    for (int s = 1; s <= r; s++)
      u = u + v;
  }
  return u;
}
```

Turing's idea

Write assertions at key program points.

Check each local step.

Use the assertions to establish the final result.

What must be true here?

A Familiar Challenge Returns in the AI Era

THE SOFTWARE CRISIS

1960s

Hardware advanced rapidly.
Software grew larger.
Manual debugging did not scale.

THE AGENT ERA

2020s

Agents generate code rapidly.
Software changes faster.
Manual review does not scale.

When code generation scales, correctness checking must scale too.

Questions To Be Answered In CDM

What problems can computers solve?

How can we know that a program is correct?

⇨ **How to ensure the program generated by coding agent is correct?**

Why Choose CDM?

V.S.

GPT-4

What makes you distinguished?

I can write programs.

I can use coding agents to build programs faster.

I can specify what programs should do and verify whether they do it.

大纲

- 图论
  - 图的基本概念
  - 道路与回路
  - 树
- 逻辑
  - 命题逻辑
  - 谓词逻辑
  - 公理系统
- 集合论
  - 集合
  - 关系
  - 函数

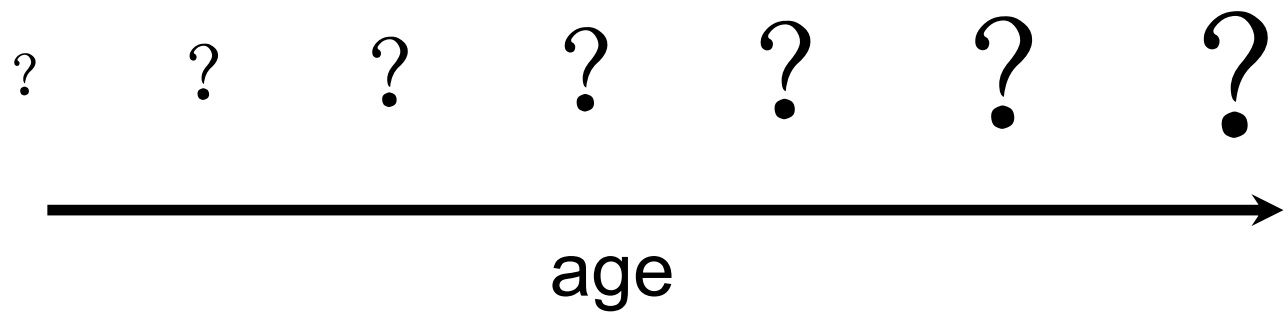
大纲

- 图论
  - 图的基本概念
  - 道路与回路
  - 树
- 逻辑
  - 命题逻辑
  - 谓词逻辑
  - 公理系统
- 集合论
  - 集合
  - 关系
  - 函数

?

大纲

- 图论
  - 图的基本概念
  - 道路与回路
  - 树
- 逻辑
  - 命题逻辑
  - 谓词逻辑
  - 公理系统
- 集合论
  - 集合
  - 关系
  - 函数



大纲

图论

- 图的基本概念
- 道路与回路
- 树

• 逻辑

- 命题逻辑
- 谓词逻辑
- 公理系统

• 集合论

- 集合
- 关系
- 函数

这将不是一门普通的离散数学

? ? ? ? ? ? ?

age

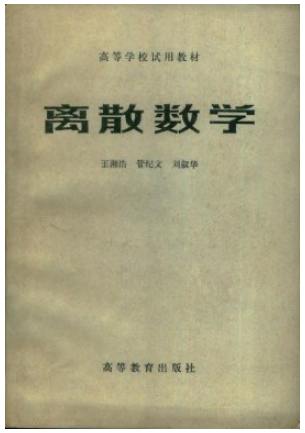
Be Practical than Attractive

Is it solvable?

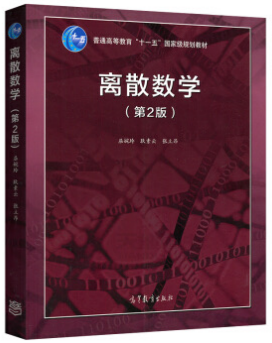

```
bool is_power2(unsigned int v)
{
    unsigned int i;
    for(i = 0; i <= 31; ++i)
    {
       if(v == (1 << i))
           return true;
    }
    return false;
}
```

GPT-4

How to prove its correctness?



1983



2015

离散数学 =

- 图论
- 数理逻辑
- 集合论
- 组合数学
- ……

What we teach: Discrete Mathematics for **Computer Science**

NYU RICE Unconventional Wisdom

- Automata
- Turing machine
- Computation theory

How to module a program

- Satisfiability problem
- Satisfiability modulo theory

How to prove a program

Three Parts of CDM

Part I. Logic Reasoning. (*~10 Weeks*)

How to prove the correctness?

Part II. Computability. (~ *5 Weeks*)

Is the problem computable (solvable)?

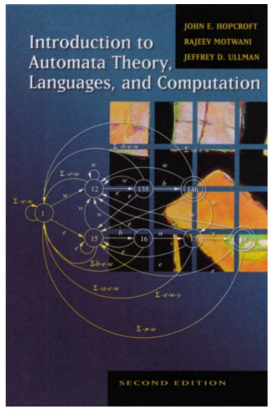
Part III. Applications. (~ *1 Weeks*)

Solve the problem in real systems.

References

Textbooks

《数理逻辑与集合论》第2版
石纯一 著，清华大学出版社



John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation, Pearson, 2001

To Be Distinguished, You Need To

Take

- ✓ Lectures
- ✓ Recitation (Optional)

Do

- ✓ 3 Labs
- ✓ Homework

Pass

- ✓ Quiz
- ✓ Final exam

Grading Breakdown

- **30%： Lab + Lecture + Homework**
- **20%: Quiz**
- **50%： Final**

Policies

You must work alone on all assignments

- You may post questions on Canvas.
- You are encouraged to answer others' questions, but refrain from explicitly giving away solutions.

Labs & Exercises

- Preview assignments due at course starting time (10:00am on Mon. & 14:00am on Thu.) on the due date
- Review assignments due at 11:59pm on the due date
- Everybody has 5 grace days
    - Must make the claim before the due
- Zero score after the due

Integrity and Collaboration Policy

We will enforce the policy strictly.

1. The work that you turn in must be yours
2. You must acknowledge your influences
3. You must not look at, or use, solutions from prior years or the Web, or seek assistance from the Internet
4. You must take reasonable steps to protect your work

   You must not publish your solutions
5. If there are inexplicable discrepancies between exam and lab performance, we will over-weight the exam and interview you.

Faculty Information

Lecturer
Zhaoguo Wang (王肇国)
zhaoguowang@sjtu.edu.cn

Lecturer
Zeyu Mi (糜泽羽)
yzmizeyu@sjtu.edu.cn

TA
Yuexuan Zhang(张岳轩)
zyx2024@sjtu.edu.cn

TA
Wei Huang (黄炜)
boogiepop1230@sjtu.edu.cn

Three Parts of CDM

Part I. Logic Reasoning. (*~10 Weeks*)

How to prove the correctness?

Part II. Computability. (~ *5 Weeks*)

Is the problem computable (solvable)?

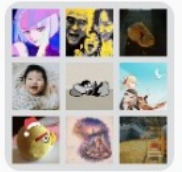
Part III. Applications. (~ *1 Weeks*)

Solve the problem in real systems.

Class Info

Website:
http://ipads.se.sjtu.edu.cn/courses/cdm

Canvas:
https://oc.sjtu.edu.cn/courses/93251



群聊：2026 离散数学课程群

该二维码 7 天内（9 月 20 日前）有效

What is Discrete Mathematics?

Definition by LLM

What is the most formal definition of Discrete Mathematics?

Formal Definition

deepseek

Discrete Mathematics is the branch of mathematics concerned with the study of mathematical structures that are fundamentally **countable** or **distinct** and **separable**. Unlike continuous mathematics, which deals with objects that can vary smoothly (such as real numbers and calculus), discrete mathematics deals with objects that can assume only distinct, separated values.

Its primary focus is on:

1. **Finite** or **countably infinite** sets.
2. Structures that emphasize **relationships** and **connectivity** between distinct elements.
3. Processes that involve **finite-state** changes and **step-by-step** procedures (algorithms).

The field is characterized by its emphasis on **precision**, **rigorous proof**, and **methodical reasoning** about discrete objects.

Continuous and Discrete Models

CONTINUOUS

Time

Distance

Temperature

Differential equations

DISCRETE

Bits

Program states

Graph nodes

Execution steps

Distinct objects **Explicit relationships** **Step-by-step processes**

Computers represent and manipulate discrete states.

Example I – Number Theory

Definition: Number theory is a branch of pure mathematics devoted primarily to the study of **the integers and arithmetic functions**.

1. **Finite** or **countably infinite** sets.

$$1, 2, 3, 4, 5, \ldots$$

2. Structures that emphasize **relationships and connectivity** between distinct elements**.**

$$2 > 1, 4 > 5$$

3. Processes that involve f**inite-state changes and step-by-step procedures** (algorithms).

$$1 + 1 = 2$$

$$2 + 1 = 3$$

Example I – Number Theory

0, 1, 2, 3, ...

MATHEMATICAL INTEGERS

..., −2, −1, 0, 1, 2, ...

Unbounded
Exact mathematical objects

32-BIT UNSIGNED INTEGERS

$0, 1, ..., 2^{32} - 1$

Finite
Arithmetic may overflow

A program may behave differently from its mathematical model.

Example II – Set Theory

Definition: Its core subject of study is sets — that is, the **totality of objects with specific properties or satisfying certain conditions**.

1. **Finite** or **countably infinite** sets.

$$\{2\},\ \{2,\ 4\},\ \{2,\ 4,\ 6,\ 8\},\ \ldots$$

2. Structures that emphasize **relationships and connectivity** between distinct elements**.**

$$\{2\} \subseteq \{2,4\}$$

3. Processes that involve f**inite-state changes and step-by-step procedures** (algorithms).

$$\{2\} \cup \{4\} = \{2,4\}$$

Example II – Set Theory

A set is a collection of distinct objects treated as one mathematical object.

$A = \{2, 4, 6, 8\}$

$x \in A$ $\quad A \subseteq B$ $\quad A \cup B$

SETS IN PROGRAMS

- Files modified by an agent
- Functions reachable from main
- Inputs accepted by a program
- States that violate an assertion

A specification often defines a set of acceptable behaviors.

Example III – Logic Reasoning

Definition: Its provides the formal system for manipulating **discrete truth values**, establishing the validity of **relationships between discrete statements.**

1. **Finite** or **countably infinite** sets.

   True, False, Statements …

2. Structures that emphasize **relationships and connectivity** between distinct elements**.**

   "Discrete Math is Math" = True

3. Processes that involve f**inite-state changes and step-by-step procedures** (algorithms).

"Discrete Math is Math" ∧ ***"I Teach Discrete Math"*** → ***"I Teach Math"***

Example IV – Graph

GRAPH

```
main → parse → check
         ↓
       report
```

Objects and relationships
Paths describe reachability

STATE MACHINE

```
Idle → Running → Completed
          ↓
        Failed
```

States and transitions
Executions describe behavior

Both models describe discrete computation.

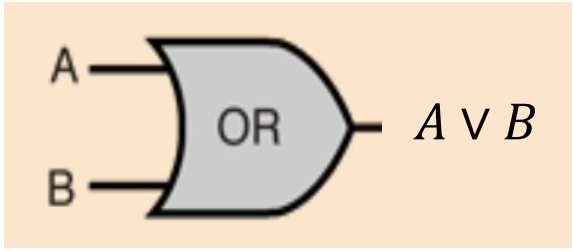
Three Parts of CDM

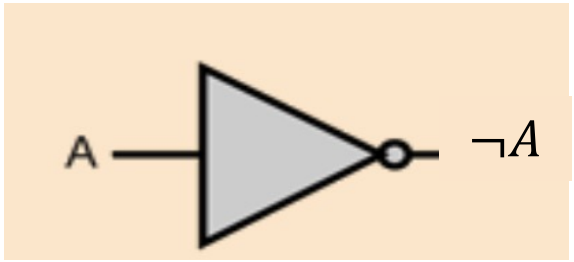
Part I. Logic Reasoning.

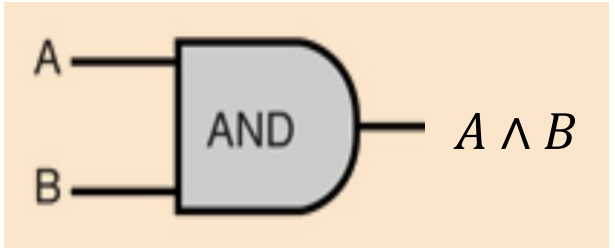
Part II. Computability.

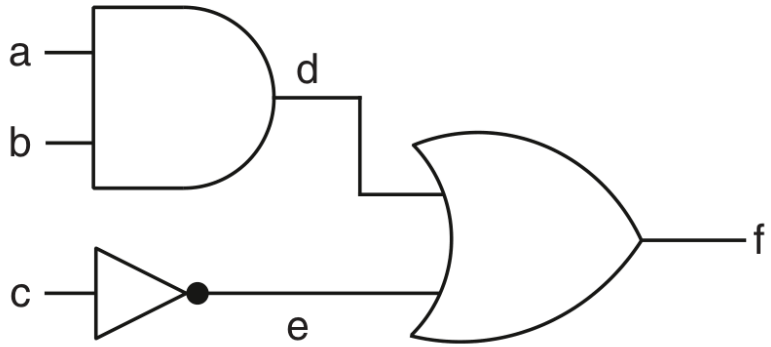
Part III. Applications.

Part I: Logic Reasoning – Integrated Circuits









Use Logic Reasoning to

- Check the logic of a designed circuit
- Verify if an optimized circuit is equivalent to the original one

Logic and Digital Circuits

AND	$A \wedge B$
OR	$A \vee B$
NOT	$\neg A$

$$\neg(A \vee B)$$

$$\neg A \wedge \neg B$$

Are these two circuits equivalent?

Logic and Programs

deepseek

Write a C function to determine whether an integer is a power of 2?

c

Copy Download

```c
#include <stdbool.h>

bool isPowerOfTwo(unsigned int n) {
    if (n == 0) {
        return false;
    }
    int count = 0;
    while (n) {
        count += n & 1; // Check if the least significant bit is 1
        n >>= 1;        // Shift bits to the right
    }
    // If only one bit was set, it's a power of two.
    return (count == 1);
}
```

Logic and Programs

deepseek

Can you simplify the generated C code?

Here's a simple C function to check if an integer is a power of 2:

c

Copy Download

```c
#include <stdbool.h>

bool isPowerOfTwo(int n) {
    if (n <= 0) {
        return false;
    }
    return (n & (n - 1)) == 0;
}
```

Logic and Programs

deepseek

can you further simplify it?

Here's the simplest C function to determine if an integer is a power of 2:

c

Copy Download

```c
#include <stdbool.h>

bool isPowerOfTwo(int n) {
    return n > 0 && (n & (n - 1)) == 0;
}
```

Logic and Programs

deepseek

How could you ensure the most simplified version is correct?

1. Test with Key Values

c Copy Download

```c
#include <stdio.h>

int isPowerOfTwo(int n) { return n > 0 && !(n & (n - 1)); }

int main() {
    // Powers of 2 (should return 1)
    printf("1: %d\n", isPowerOfTwo(1));     // 2^0
    printf("2: %d\n", isPowerOfTwo(2));     // 2^1
    printf("4: %d\n", isPowerOfTwo(4));     // 2^2
    printf("8: %d\n", isPowerOfTwo(8));     // 2^3
    printf("16: %d\n", isPowerOfTwo(16));   // 2^4
    printf("1024: %d\n", isPowerOfTwo(1024)); // 2^10

    // Non-powers of 2 (should return 0)
    printf("0: %d\n", isPowerOfTwo(0));     // edge case
    printf("3: %d\n", isPowerOfTwo(3));
    printf("6: %d\n", isPowerOfTwo(6));
    printf("10: %d\n", isPowerOfTwo(10));
    printf("15: %d\n", isPowerOfTwo(15));

    // Negative numbers (should return 0)
    printf("-1: %d\n", isPowerOfTwo(-1));
    printf("-4: %d\n", isPowerOfTwo(-4));
    printf("-8: %d\n", isPowerOfTwo(-8));

    return 0;
}
```

2. Mathematical Verification

Let's trace through the logic:

- For n = 8 (1000 in binary): `8 > 0 && !(8 & 7)` → `true && !(1000 & 0111)` → `true && !(0000)` → `true && true` → **1** ✓
- For n = 6 (0110 in binary): `6 > 0 && !(6 & 5)` → `true && !(0110 & 0101)` → `true && !(0100)` → `true && false` → **0** ✓
- For n = 0: `0 > 0 && ...` → `false && ...` → **0** ✓
- For n = -4: `-4 > 0 && ...` → `false && ...` → **0** ✓

3. Property Verification

Powers of 2 have exactly one '1' bit in their binary representation:

- 1: 0001 ✓
- 2: 0010 ✓
- 4: 0100 ✓
- 8: 1000 ✓

The expression `n & (n - 1)` clears the least significant '1' bit. If only one '1' bit exists (power of 2), this results in 0.

Logic and Programs

deepseek

how could you ensure your test cases are correct and cover all branches?

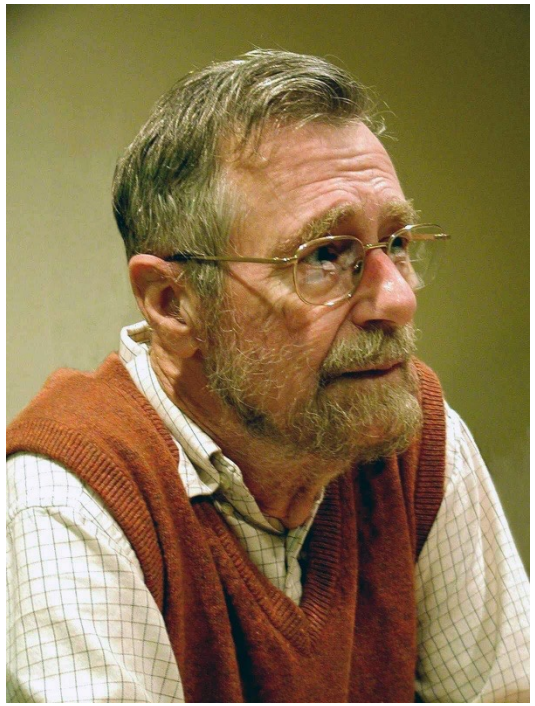
2. Comprehensive Test Suite

c

Copy Downlc

```c
#include <stdio.h>
#include <limits.h>

int isPowerOfTwo(int n) { return n > 0 && !(n & (n - 1)); }

void test_case(int n, int expected, const char *description) {
    int result = isPowerOfTwo(n);
    printf("%s: %d -> %d (expected %d) %s\n",
           description, n, result, expected,
           result == expected ? "✓" : "✗");
}

int main() {
    printf("=== Testing Power of 2 Function ===\n");

    // Branch 1: n > 0 → false (n ≤ 0)
    test_case(0, 0, "Zero");
    test_case(-1, 0, "Negative one");
    test_case(-4, 0, "Negative four");
    test_case(-8, 0, "Negative eight");
    test_case(INT_MIN, 0, "Minimum integer");
    test_case(3, 0, "Three (not power of 2)");
    test_case(5, 0, "Five (not power of 2)");
    test_case(6, 0, "Six (not power of 2)");
    test_case(7, 0, "Seven (not power of 2)");
    test_case(9, 0, "Nine (not power of 2)");
    test_case(10, 0, "Ten (not power of 2)");
    test_case(15, 0, "Fifteen (not power of 2)");
    test_case(1023, 0, "1023 (not power of 2)");
    test_case(INT_MAX, 0, "Maximum integer (not power of 2)");

    // Branch 4: !(n & (n-1)) → true (power of 2)
    test_case(1, 1, "One (2^0)");
    test_case(2, 1, "Two (2^1)");
    test_case(4, 1, "Four (2^2)");
    test_case(8, 1, "Eight (2^3)");
    test_case(16, 1, "Sixteen (2^4)");
    test_case(32, 1, "Thirty-two (2^5)");
    test_case(64, 1, "Sixty-four (2^6)");
    test_case(128, 1, "One twenty-eight (2^7)");
    test_case(256, 1, "Two fifty-six (2^8)");
    test_case(512, 1, "Five twelve (2^9)");
    test_case(1024, 1, "Ten twenty-four (2^10)");
    test_case(2048, 1, "Two forty-eight (2^11)");
    test_case(4096, 1, "Four ninety-six (2^12)");

    // Edge cases around powers of 2
    test_case(1023, 0, "1024-1");
    test_case(1025, 0, "1024+1");
    test_case(2047, 0, "2048-1");
    test_case(2049, 0, "2048+1");
```

Logic and Programs



Edsger Wybe Dijkstra

Testing shows the presence not the absence of bugs.

Logic and Programs

deepseek

Could you prove the correctness of the code?

Theorem:

The function `isPowerOfTwo(n)` returns `true` **if and only if** `n` is a positive integer and $n = 2^k$ for some integer $k \geq 0$.

Proof:

1. (⇒) If `isPowerOfTwo(n)` returns `true`, then `n` is a power of two.

- By the function's definition, $n > 0$ and $(n \,\&\, (n-1)) = 0$.
- The condition $(n \,\&\, (n-1)) = 0$ implies that n has a single 1 bit in its binary representation. This is because subtracting 1 flips the least significant 1 bit and all subsequent 0s, and the bitwise AND is zero only if no higher bits remain.
- A positive integer with exactly one 1 bit is, by definition, a power of two. ■

2. (⇐) If `n` is a positive power of two, then `isPowerOfTwo(n)` returns `true`.

- Let $n = 2^k$ for some $k \geq 0$. Then $n > 0$.
- The binary representation of n is a 1 followed by k 0s.
- The binary representation of $n - 1$ is k 1s.
- Therefore, $n \,\&\, (n-1) = 0$ (the 1 in n aligns with a 0 in $n-1$, and all other bits are 0 in n).
- Thus, both conditions $n > 0$ and $(n \,\&\, (n-1)) = 0$ are satisfied. ■

By proving both directions, we conclude the function is correct.

Logic and Programs

deepseek

How could you ensure your proof is correct?

1. Verify the Logical Structure

The proof has a standard and robust structure: an **if-and-only-if** statement proven in two parts (⇒ and ⇐). This structure ensures no gaps in logic.

- **Part 1 (Soundness):** Proves the function has no **false positives**. If it returns `true`, the input *must be* a power of two.
- **Part 2 (Completeness):** Proves the function has no **false negatives**. If the input *is* a power of two, the function *will* return `true`.

This two-part approach is a recognized best practice for validating equivalence.

2. Validate the Core Insight

The entire proof hinges on one key insight: **For `n > 0`, `n & (n - 1) == 0` is equivalent to `n` having a single `1` in its binary representation.**

- **Justification:** This is a well-known and proven property in bitwise arithmetic. It can be found in computer science literature and textbooks (e.g., *Hacker's Delight* by Henry S. Warren, Jr.). Relying on such an established lemma strengthens the proof's credibility.
- **Sanity Check with Examples:**
  - `n = 8` (1000). `n-1 = 7` (0111). `1000 & 0111 = 0000`. ✓
  - `n = 6` (0110). `n-1 = 5` (0101). `0110 & 0101 = 0100 != 0`. ✓
  - `n = 0`. `n-1 = -1`. `0 & (-1) = 0`, but `n > 0` is false, so the function correctly returns `false`. This validates the necessity of the `n > 0` check.

Testing the core logic with concrete examples confirms the abstract reasoning.

3. Check for Edge Cases

A correct proof must account for all possible inputs, especially edge cases.

- **n = 0:** Addressed. `0` is not positive, so the function returns `false`. The proof explicitly states `n > 0`.
- **n = 1 (2^0):** `1 > 0` is true. `1 & 0 = 0` is true. The function returns `true`. The proof holds for the smallest power of two.
- **Negative Numbers:** The function's first condition `n > 0` filters them out correctly. The proof only concerns itself with positive integers.
- **Large Numbers:** The proof is based on the properties of binary representation, which hold for all integers within the language's limits (e.g., 32-bit or 64-bit `int`). The logic is scale-invariant.

4. Peer Review and Formalization

- **Explain it to someone else (or a rubber duck):** Articulating the proof's steps forces you to clarify your reasoning and exposes hidden assumptions or leaps in logic. The proof presented is clear and easy to follow.
- **Formalize the bit-level argument:** The proof can be made more formal by denoting the position of the least significant `1` bit.
  - Let `n` be a positive integer. Let `i` be the position of its least significant `1` bit. Then `n` can be represented as: `n = ...a_{i+1}a_i 00...0` where `a_i = 1`.
  - Then `n-1 = ...a_{i+1}0 11...1`.
  - Therefore, `n & (n-1) = ...a_{i+1}0 00...0`.
  - For `n & (n-1)` to be `0`, there must be no `1` bits to the left of position `i` (i.e., all `a_{i+1}...` must be `0`). This means the `1` at position `i` was the only `1` in the number.

5. Contrast with Incorrect Alternatives

A good way to ensure a proof is correct is to see why common *incorrect* solutions fail and confirm your proof accounts for this.

- **Incorrect alternative:** `return (n & (n - 1)) == 0;`
  - **Why it's wrong:** Fails for `n=0`. `0 & (-1) = 0`, so it would return `true` for `0`, which is not a power of two.
  - **Does our proof cover this?** Yes. The proof explicitly requires `n > 0` as a necessary condition in both directions.

More difficult to ensure the correctness

Logic and Programs

deepseek

How to ensure the proof of your proof is correct?

1. The Social Process: Peer Review

This is the most common and practical method in both academia and industry.

- **How it works:** You present your proof to other experts. They critique the logic, identify hidden assumptions, challenge steps, and attempt to find counterexamples.
- **Why it works:** It leverages the collective intelligence and skepticism of a community. Errors that one person overlooks are often caught by another.
- **Analogy:** It's like having multiple senior engineers review a piece of critical code.
- **For our power-of-two proof:** Explaining the two-part if-and-only-if structure and the bit-level argument to another programmer or mathematician is a form of peer review. If they agree it's sound, your confidence increases.

2. The Mechanical Process: Exhaustive Testing (for finite domains)

While not a proof itself, testing can provide extreme confidence for functions with limited input spaces.

- **How it works:** For a 32-bit integer, there are only 32 positive powers of two (2^0 to 2^{31}). You can write a test harness that iterates over all ~4.3 billion possible `int` values and verifies that the function returns `true` **only** for those 32 values and `false` for all others.
- **Why it works:** It provides a complete, empirical verification for the given domain (e.g., 32-bit integers). It can catch off-by-one errors and missed edge cases that a logical proof might have overlooked.
- **Limitation:** It only works for finite domains. It doesn't constitute a general proof for all possible integers (e.g., in arbitrary-precision arithmetic), but for a C `int`, it's effectively perfect.

3. The Formal Process: Constructing a Formal Proof

This is the gold standard. You break down the proof into such small, precise steps that validity can be verified automatically.

Even Harder

- **Step 1: Formal Specification:** Precisely define the function's behavior in a formal l our function, we would specify:
  - `isPowerOfTwo(n) ↔ (n > 0) ∧ (∃k : Nat, n = 2^k)`
- **Step 2: Use a Proof Assistant:** Use a tool like **Coq**, **Lean**, **Isabelle/HOL**, or **ACL2**. These are not just programming languages but systems for writing machine-checkable proofs.
  - You would **define** the function in the tool's language.
  - You would **state the theorem** you want to prove (`∀ n : int, isPowerOfTwo n = true ↔ n > 0 ∧ ∃ k, n = 2^k`).
  - You would **interactively construct the proof** within the tool. The tool breaks the goal into smaller subgoals. For each step (e.g., "rewrite using the definition of `&`", "case analysis on `n`"), the tool ensures the logical rule is applied correctly.
- **Why it works:** The proof assistant acts as an infallible, ultra-pedantic peer reviewer. It has a built-in kernel of logical rules and will not allow you to proceed unless each step is rigorously justified. It eliminates human error in the *va*
- **What it ensures:** The proof of your proof is the axioms to the final theorem, has been assumptions are the correctness of the pr

4. The Foundational Process: Minimizing Trust

Even with a proof assistant, you must trust its core logic ("kernel"). To ensure even *that* is correct, the field of **verified computing** takes it a step further.

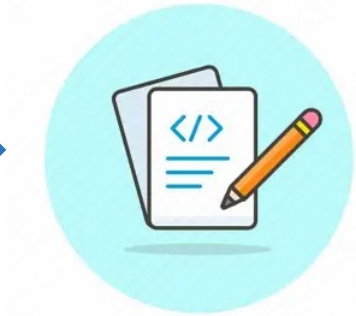
- **How it works:** The source code of the proof assistant's kernel itself is mathematically proven to be correct. This is called a **verified kernel**.
- **Example:** The **CompCert** C compiler is a famous project where the compiler itself is formally verified in Coq to ensure it translates source code correctly. You could, in theory, verify a proof assistant's kernel with another, more minimal, proof assistant.
- **This is the final step.** It minimizes the "trust base" to a very small set of axioms and the physical laws governing the computer hardware.

Logic and Coding Agents

Prompt: Code Requirement

Write a C function to determine whether an integer is a power of 2?

deepseek



Computer Aided Reasoning (Automatic Verifier)

✅

Logic and Coding Agents

```
if (logged_in && has_permission) {
    allow_access();
}
```

```
Condition:
logged_in ∧ has_permission

Safety property:
allow_access → has_permission
```

Program decisions can be represented as logical formulas.

CODING AGENT

Generates an implementation

Suggests test cases

Proposes an explanation

LOGIC

States the requirement precisely

Exposes missing assumptions

Checks whether conclusions follow

A Programming Task

Given a nonnegative integer n, compute the sum from 1 to n.

```
sum(0) = 0
sum(3) = 6
sum(5) = 15
```

What exactly should the program guarantee?

An Agent-Generated Program

```
unsigned int sum(unsigned int n) {
    unsigned int s = 0;
    unsigned int i = 0;
    while (i < n) {
        i = i + 1;
        s = s + i;
    }
    return s;
}
```

- The code compiles.
- The examples pass.
- The implementation looks reasonable.

Is that enough?

Testing the Program

```
assert(sum(0) == 0);

assert(sum(1) == 1);

assert(sum(2) == 3);

assert(sum(10) == 55);
```

- What about n = 100?
- What about the largest unsigned integer?
- What about overflow?

Tests provide concrete evidence.

Preconditions and Postconditions

THE SPECIFICATION For every allowed n, result = n(n + 1) / 2.

The specification states the result without prescribing the algorithm.

```
{ n ≥ 0 }   sum(n)   { result = n(n + 1) / 2 }
```

PRECONDITION

What must hold before execution

$n \geq 0$

Example: $4 \geq 0$

POSTCONDITION

What must hold after execution

result = n(n + 1) / 2

Example: result = 10

Part I: Computer Aided Reasoning

```
bool xor(unsigned int v)
{
    return v ^ v;
}
```

$$(((\neg v1) \wedge v1) \vee (v1 \wedge (\neg v1))) \vee$$
$$(((\neg v2) \wedge v2) \vee (v2 \wedge (\neg v2))) \vee$$
$$(((\neg v3) \wedge v3) \vee (v3 \wedge (\neg v3))) \vee$$
$$\ldots$$
$$(((\neg v32) \wedge v32) \vee (v32 \wedge (\neg v32)))$$

A Propositional Formula

1.1 Propositional Logic

Step 1. Convert it into mathematical formula.

Step 2. Ask the computer to solve the formula.

Part I: Computer Aided Reasoning

```
void swap(int a, int b)
{
    a ^= b;
    b ^= a;
    a ^= b;
}
```

$$
\begin{aligned}
&(a1' \leftrightarrow (((((a1 \wedge (\neg b1)) \vee ((\neg a1) \wedge b1)))\wedge\\
&(\neg((((((a1 \wedge (\neg b1)) \vee ((\neg a1) \wedge b1))) \wedge (\neg b1))\vee\\
&((\neg(((a1 \wedge (\neg b1)) \vee ((\neg a1) \wedge b1)))) \wedge b1)))))\vee\\
&((\neg(((a1 \wedge (\neg b1)) \vee ((\neg a1) \wedge b1))))\wedge\\
&((((((a1 \wedge (\neg b1)) \vee ((\neg a1) \wedge b1))) \wedge (\neg b1))\vee\\
&((\neg(((a1 \wedge (\neg b1)) \vee ((\neg a1) \wedge b1)))) \wedge b1))))))\wedge\\
&(b1' \leftrightarrow (((((a1 \wedge (\neg b1)) \vee ((\neg a1) \wedge b1))) \wedge (\neg b1))\vee\\
&((\neg(((a1 \wedge (\neg b1)) \vee ((\neg a1) \wedge b1)))) \wedge b1)))\wedge\\
&\ldots
\end{aligned}
$$

A Propositional Formula

1.1 Propositional Logic

Step 1. Convert it into mathematical formula.

Step 2. Ask the computer to solve the formula.

It is complicated for both human being and computer.

Part I: Computer Aided Reasoning

```
void swap(int a, int b)
{
    a ^= b;
    b ^= a;
    a ^= b;
}
```

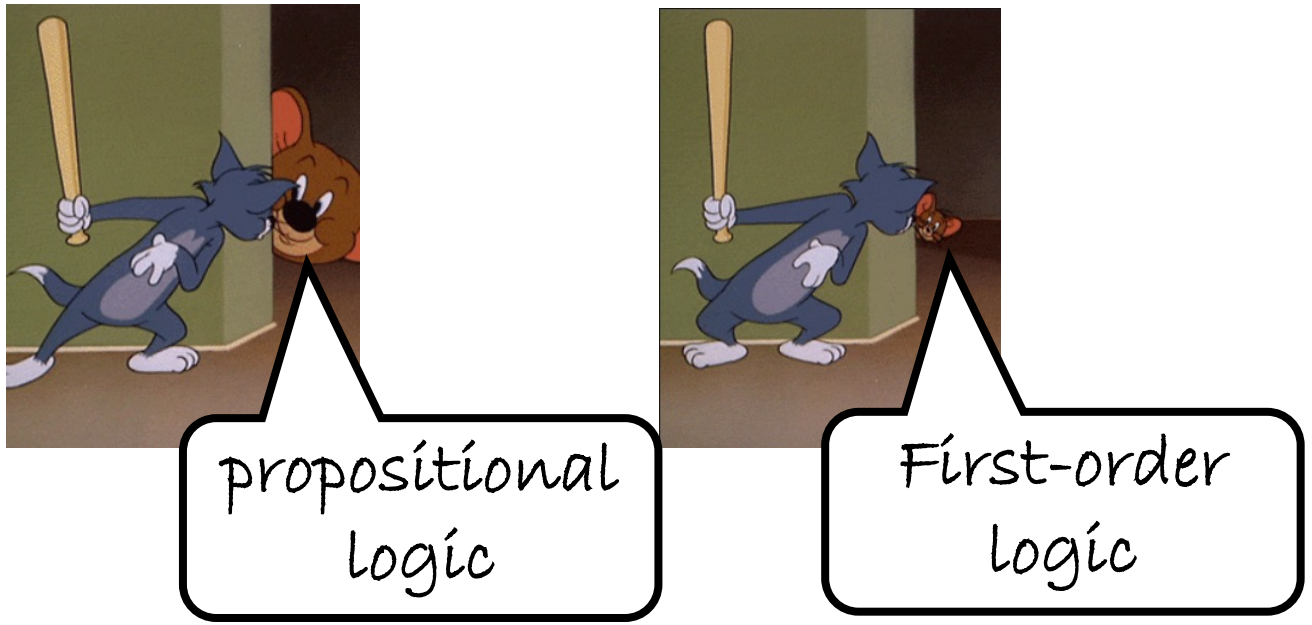
$$(a' = XOR(XOR(a,b), XOR(b, XOR(a,b))))\wedge$$
$$(b' = XOR(b, XOR(a,b)))$$

A Predicate Formula

1.1 Propositional Logic

Step 1. Convert it into mathematical formula.

Step 2. Ask the computer to solve the formula.



Part I: Computer Aided Reasoning

```
void swap(int a, int b)
{
    a ^= b;
    b ^= a;
    a ^= b;
}
```

$$(a' = XOR(XOR(a, b), XOR(b, XOR(a, b))))\wedge$$
$$(b' = XOR(b, XOR(a, b)))$$

A Predicate Formula

1.1 Propositional Logic

Step 1. Convert it into mathematical formula.

Step 2. Ask the computer to solve the formula.

1.2 First Order Logic

Step 1. Convert it into first logic formula.

Step 2. Ask the computer to solve the formula.

Part I: Computer Aided Reasoning

```
int clone(unsigned int n)
{
    int i = 0;
    while(i < n)
    {
        i = i + 1;
    }
    return i;
}
```

Loop can not be addressed by first-order logic.

Can we tell computer this is a loop?

1.1 Propositional Logic

Step 1. Convert it into mathematical formula.

Step 2. Ask the computer to solve the formula.

1.2 First Order Logic

Step 1. Convert it into first logic formula.

Step 2. Ask the computer to solve the formula.

Part I: Computer Aided Reasoning

```
int clone(unsigned int n)
{
    int i = 0;
    while(i < n)
    {
        i = i + 1;
    }
    return i;
}
```

```
method clone(n: nat) returns (b : nat)
  ensures b == n
{
   var i := 0;
   while i < n
      invariant 0 <= i
   {
      i := i + 1;
   }
   return i;
}
```

1.1 Propositional Logic

Step 1. Convert it into mathematical formula.

Step 2. Ask the computer to solve the formula.

1.2 First Order Logic

Step 1. Convert it into first logic formula.

Step 2. Ask the computer to solve the formula.

1.3 Auto-active Proof

Step 1. Axiom system

Step 2. Ask the computer to check the invariants

Part I: Computer Aided Reasoning

```
int clone(unsigned int n)
{
    int i = 0;
    while(i < n)
    {
        i = i + 1;
    }
    return i;
}
```

```
method clone(n: nat) returns (b : nat)
  ensures b == n
{
   var i := 0;
   while i < n
      invariant 0 <= i
   {
      i := i + 1;
   }
   return i;
}
```

		Description	Line	Column
⊗	1	A postcondition might not hold on this return path.	10	3
	2	This is the postcondition that might not hold.	2	12

Part I: Computer Aided Reasoning

```
int clone(unsigned int n)
{
    int i = 0;
    while(i < n)
    {
        i = i + 1;
    }
    return i;
}
```

```
method clone(n: nat) returns (b : nat)
  ensures b == n
{
   var i := 0;
   while i < n
      invariant 0 <= i && i <= n
   {
      i := i + 1;
   }
   return i;
}
```

```
Dafny 2.3.0.10506

Dafny program verifier finished with 1 verified, 0 errors
Program compiled successfully
```

Part I: Computer Aided Reasoning Overview

1.1 Propositional Logic

Step 0. Proof.

Step 1. Convert program into mathematical formula.

Step 2. Ask the computer to solve the formula.

1.2 First Order Logic

Step 1. Convert program into first logic formula.

Step 2. Ask the computer to solve the formula.

1.3 Auto-active Proof

Step 1. Axiom system

Step 2. Ask the computer to check the invariants

TA

Yuexuan Zhang(张岳轩)

zyx2024@sjtu.edu.cn

3 Labs in total.
Good luck to all!

Part I: Computer Aided Reasoning Overview

模型名称	模型参数	调用名	模式	侧重领域	上下文长度
DeepSeek V4 Flash（常规模式）	284B	deepseek-chat	非思考模式	通用文本处理	256k
DeepSeek V4 Flash（思考模式）	284B	deepseek-reasoner	深度思考模式	复杂逻辑深度推理	256k
MiniMax-M2.7	230B	minimax 或 minimax-m2.7	文本生成	智能体任务	192k
Qwen3.8-27B	27B	qwen 或 qwen3.8-27b	多模态处理	视觉与文本理解	256k

TA
Yuexuan Zhang(张岳轩)
zyx2024@sjtu.edu.cn

每个同学

- 每分钟限流300K TPM，10RPM
- 每周限额1B