Submission guidelines:
- Present complete, self-contained solutions.
- For proofs: use Fitch notation (numbered steps, subproofs, rule citations).
- For tableaux/resolution: show the full derivation tree.
- For semantic arguments: provide explicit interpretations and show evaluations.
- For counterexamples: give a concrete model and demonstrate the failure.
- For transformations: show all intermediate steps.

Problem 1: Natural deduction (Fitch proofs)

Construct a natural deduction proof for each sequent below. Use Fitch notation: numbered steps, proper subproof indentation, explicit rule citations.

Strategy hints:
- To prove $\alpha \to \beta$, use $\to$i: assume α, derive β, discharge.
- To prove $\neg\alpha$, use $\neg$i: assume α, derive $\bot$, discharge.
- To use $\alpha \lor \beta$, apply $\lor$e: derive the goal from each disjunct separately.
- For contradictions, $\bot$e lets you derive anything; RAA lets you derive α from $\neg\alpha \vdash \bot$.

(a) $A \to C, B \to C, A \lor B \vdash C$ *(disjunction elimination)*

(b) $A \to B, A \to \neg B \vdash \neg A$ *(reductio ad absurdum)*

(c) $\vdash (A \to B) \to ((\neg A \to \bot) \to B)$ *(nested implications)*

(d) $\vdash P \lor \neg P$ *(law of excluded middle — derive it, don't assume it)*

(e) $\vdash ((A \to B) \to A) \to A$ *(Peirce's law)*

(f) $\neg A \to \neg B \vdash B \to A$ *(contrapositive, classical)*

(g) $\vdash (A \to (B \to C)) \to ((A \to B) \to (A \to C))$ *(distributivity of $\to$)*

(h) $(A \to B), (\neg A \to B) \vdash B$ *(case analysis)*

(i) $A \lor (B \to A) \vdash \neg A \to \neg B$ *(combining disjunction and negation)*

(j) $\vdash (\neg\neg A \to A) \to ((A \to \neg\neg A) \to (A \leftrightarrow \neg\neg A))$ *(double negation equivalence)*

Problem 2: Semantics: validity, entailment, and countermodels

For each claim below, determine whether it holds.
- If **valid/entails**, justify with a brief semantic argument or a truth table.
- If **invalid**, give a concrete counterexample: an interpretation ν (specifying $\nu(p), \nu(q), \ldots$) and show the evaluation that falsifies the claim.

(a) $A \to B, B \to C \vDash A \to C$ *(transitivity)*

(b) $A \lor B, \neg A \vDash B$ *(disjunctive syllogism)*

(c) $A \to B \vDash \neg B \to \neg A$ *(contraposition)*

(d) $A \to B \vDash B \to A$ *(converse — suspicious!)*

(e) $\vDash (A \land B) \to A$ *(conjunction elimination)*

(f) $\vDash ((A \to B) \land (A \to \neg B)) \to \neg A$ *(proof by contradiction)*

(g) $A \leftrightarrow B, B \leftrightarrow C \vDash A \leftrightarrow C$ *(equivalence transitivity)*

(h) $\vDash (A \to B) \lor (B \to A)$ *(linearity of implication — tricky!)*

Problem 3: Normal forms and Tseitin transformation

SAT solvers operate on CNF. This problem practices the two main conversion strategies: direct transformation (via distributivity) and Tseitin encoding (equisatisfiability with fresh variables).

(a) **Direct CNF conversion.** Convert $(\neg A \lor B) \land (\neg B \lor C) \to (\neg A \lor C)$ to CNF by:
- Eliminating $\to$ (rewrite using $\neg$, $\lor$).
- Pushing negations to atoms (De Morgan, double negation).
- Distributing $\lor$ over $\land$ to obtain CNF.
- State whether the result is satisfiable (give a model or show unsatisfiability).

(b) **Tseitin transformation.** Convert $(P_1 \leftrightarrow P_2) \leftrightarrow (P_3 \leftrightarrow P_4)$ to *equisatisfiable* CNF.
- Introduce fresh variable N_i for each complex subformula α.
- Write definitional clauses $N_i \leftrightarrow \alpha$.
- Convert each $\leftrightarrow$ definition to CNF.
- Write the final clausal form.
- **Compare:** How many clauses does your Tseitin encoding have? How many would the *direct* distributive CNF have?

(c) **NNF without CNF.** Convert $\neg((A \to B) \land (C \lor \neg D))$ to Negation Normal Form (NNF) — negations only on atoms, but *no* requirement for CNF, *i.e.* do not distribute.

Problem 4: Proof system comparison: Tableaux and Resolution

Both semantic tableaux and resolution are *refutation systems*: to prove $\Gamma \vDash \alpha$, they attempt to derive a contradiction from $\Gamma \cup \{\neg\alpha\}$.

(a) **Tableaux.** Use the semantic tableaux method to prove:

$$\vDash (P \to Q) \to (\neg Q \to \neg P)$$

Negate the formula, apply decomposition rules (α-rules extend, β-rules branch), and show that all branches close. If you find an open branch, give the countermodel it represents.

(b) **Resolution.** Use resolution refutation to prove:

$$\{P \lor Q, \neg P \lor R, \neg Q \lor R\} \vDash R$$

Convert $\Gamma \cup \{\neg r\}$ to CNF (one clause per formula), then derive the empty clause $\square$ by repeated resolution steps. Annotate each resolvent with the parent clauses and the pivot literal.

(c) **Comparison.** For the formula $(A \to B) \land (B \to C) \land A \land \neg C$:
- Show it is *unsatisfiable* using tableaux (exhibit the closed tree).
- Show it is *unsatisfiable* using resolution (derive $\square$).
- Which method required fewer steps? Briefly explain why.

Problem 5: Modeling with propositional logic

Consider a simplified access control system with four propositional atoms:
- P: production mode is active
- D: debug mode is active
- L: verbose logging is enabled
- S: strict security checks are enforced

The system specification states:
1. Production mode requires strict security.
2. Debug mode requires verbose logging.
3. Strict security is incompatible with debug mode.
4. The system is always in exactly one mode: production or debug.
5. If logging is disabled, the system cannot be in debug mode.

(a) Express the specification as a single formula Φ_{spec} (a conjunction of the five constraints).

(b) Determine whether $\Phi_{\text{spec}} \models (P \to \neg D)$. Justify or provide a countermodel.

(c) Determine whether $\Phi_{\text{spec}} \models (\neg L \to \neg P)$. Justify or provide a countermodel.

(d) Show that Φ_{spec} is satisfiable by giving *two* distinct models (interpretations satisfying all constraints). What are the only two valid system configurations?

Problem 6: First-order logic and metatheory

This problem explores quantifiers, structures, and fundamental limitations of FOL.

(a) **Formalization.** Express the following statements in first-order logic over the signature $\{E(\cdot,\cdot), \cdot \leq \cdot\}$ (where $E(X, Y)$ means "X is an edge to Y" and $\leq$ is a partial order on vertices):
- Every vertex has at most one outgoing edge.
- There exists a vertex from which all other vertices are reachable (directly or transitively).
- The graph is acyclic.

(b) **Quantifier equivalences.** Determine whether the following are *logically valid* (true in all structures). If valid, justify; if not, give a countermodel.
- $\exists X.\, \forall Y.\, R(X, Y) \quad \vdash \quad \forall Y.\, \exists X.\, R(X, Y)$
- $\forall Y.\, \exists X.\, R(X, Y) \quad \vdash \quad \exists X.\, \forall Y.\, R(X, Y)$

(c) **Prenex Normal Form.** Convert to prenex normal form (all quantifiers at the front):

$$(\forall x.\, P(x)) \to (\exists y.\, Q(y))$$

Hint: Rename bound variables to avoid capture (note: variables x, y are bound), then move quantifiers outward.

(d) **Natural deduction in FOL.** Prove:

$$\forall x.\,(P(x) \to Q(x)), \forall x.\,P(x) \quad \vdash \quad \forall x.\,Q(x)$$

Use Fitch notation with quantifier rules. Recall: ∀e instantiates to any term; ∀i requires an arbitrary (fresh) variable.

(e) **Compactness and infinity.** Let $\Gamma = \{\exists x.\, x \neq c\} \cup \{\exists x \exists y.\,(x \neq y \land x \neq c \land y \neq c)\} \cup \ldots$ (*i.e.*, "there are at least n elements distinct from c" for every $n \in \mathbb{N}$).
- Show that every *finite* subset $\Gamma_0 \subseteq \Gamma$ is satisfiable.
- By compactness, what does this imply about Γ itself?
- Conclude: "the domain is infinite" can be expressed by an infinite set of FOL sentences, but not by a *single* sentence (or any finite set). Why?

Problem 7: Programming Challenge — Logic in Practice

Implementation context:

This is a *programming project* where you implement core logic concepts from scratch and explore their real-world applications. Choose your language: Python, Rust, OCaml, Haskell, or any language with algebraic data types. For formal verification tasks, use Lean 4 or Coq.

Submission: Code repository (GitHub/GitLab) with README explaining your design choices, plus a brief report (2-3 pages) documenting what you implemented, challenges encountered, and insights gained.

Grading: Core tasks (50%), code quality & documentation (30%), extensions (20%).

Part A: Formula Engine (Core Implementation)

Build a *propositional logic toolkit* from the ground up.

Task A.1: Abstract Syntax Tree

Design and implement an AST representation for propositional formulas.

Required:
- Data type/class for formulas: atoms (p, q, …), ¬, ∧, ∨, →, ↔, constants (⊤, ⊥)
- Constructor functions or smart constructors
- Structural equality and hashing (for sets/maps)

Example (Rust):

```rust
enum Formula {
    Atom(String),
    Not(Box<Formula>),
    And(Box<Formula>, Box<Formula>),
    // ... complete the rest
}
```

Example (Python):

```python
@dataclass(frozen=True)
class Atom:
    name: str

@dataclass(frozen=True)
class Not:
    operand: Formula
# ... complete using Union or inheritance
```

Open question: Should $\to$ and $\leftrightarrow$ be primitive or derived? Justify your choice.

Task A.2: Pretty Printer and Parser

Required:

- `to_string(formula)`: Convert AST to human-readable string with minimal parentheses. Use precedence: $\neg > \land > \lor > \to > \leftrightarrow$
- Handle associativity correctly

Optional:

- Parser: `parse(string)` $\longrightarrow$ AST. Use a parser combinator library (*e.g.*, `pyparsing`, `nom`, `parsec`) or write a recursive descent parser.
- Round-trip test: `parse(to_string(f)) == f`

Test case:

```
((p ∧ q) → r) ∨ (¬p ∧ s)  should print with minimal parens
```

Task A.3: Evaluator and Truth Tables

Required:

- `eval(formula, interpretation)`: Evaluate formula under a given variable assignment.
- `truth_table(formula)`: Generate complete truth table as a list/table of `(valuation, result)` pairs.
- `is_tautology(formula)`, `is_satisfiable(formula)`, `is_contradiction(formula)`

Output format:

```
p | q | r | (p ∧ q) → r
--|---|---|-------------
T | T | T |      T
T | T | F |      F
...
```

Open task: Implement *early termination*: stop generating the truth table for `is_satisfiable` as soon as you find one satisfying assignment.

Task A.4: Normal Forms

Required:
- `to_nnf(formula)`: Convert to Negation Normal Form (push negations to atoms).
- `to_cnf(formula)`: Convert to Conjunctive Normal Form using distributivity or Tseitin transformation.

Open choice: Should `to_cnf` always use Tseitin (polynomial blowup) or try direct conversion first (exponential worst case, but often smaller)? Implement both and compare.

Test: Verify that your CNF conversion preserves satisfiability (or equivalence, if not using Tseitin).

Task A.5: Equivalence and Properties

Required:
- `equivalent(f1, f2)`: Check logical equivalence using truth tables or SAT solving.
- Implement and test De Morgan's laws:
  - $\neg(p \land q) \equiv (\neg p) \lor (\neg q)$
  - $\neg(p \lor q) \equiv (\neg p) \land (\neg q)$
- Test distributivity: $p \land (q \lor r) \equiv (p \land q) \lor (p \land r)$

Extension:
- Implement a *random formula generator* for property-based testing.
- Use it to test commutativity, associativity, absorption laws.

Part B: Proof Systems (Advanced Implementation)

Implement proof checking and (optionally) proof search.

Task B.1: Fitch Proof Representation

Required:
- Data structure for Fitch proofs: list of steps, each with:
  - Line number
  - Formula
  - Justification (rule name + references to earlier lines)
  - Indentation level (for subproofs)

Example structure (pseudocode):

```
Step = {
  line: int,
  formula: Formula,
  rule: Rule,
  references: List[int],
```

```
  level: int,  // subproof depth
}
```

Open design question: How do you represent assumptions vs. derived steps? How do you track subproof scope?

Task B.2: Proof Checker

Required: Implement a checker that validates Fitch proofs step-by-step. Support at minimum:
- Premise (assumption at depth 0)
- Assumption (start subproof, increase depth)
- $\wedge$i, $\wedge$e, $\vee$i, $\vee$e, $\rightarrow$i, $\rightarrow$e, $\neg$i, $\neg$e, $\bot$e
- Reiteration (repeat earlier line from valid scope)

Checker requirements:
- Verify each step references valid earlier lines
- Check subproof scoping (can only reference lines from current or outer scopes)
- Verify rule applications are correct
- Report specific errors with line numbers

Test case: Validate the proof from Problem 1(a) in HW1.

Extension:
- Add RAA (reductio ad absurdum) and LEM (law of excluded middle)
- Support FOL quantifier rules ($\forall$i/e, $\exists$i/e) with eigenvariable checking

Task B.3: Automated Proof Search (Optional)

Challenge: Implement a simple automated prover.

Approach 1 (Semantic Tableaux):
- Implement a tableau prover: try to build a countermodel by systematic case analysis.
- If all branches close, the formula is a tautology.

Approach 2 (Resolution):
- Convert to CNF, apply resolution until deriving $\square$ or saturating.

Approach 3 (Sequent Calculus):
- Bottom-up proof search in sequent calculus.

Open exploration: Which approach finds proofs fastest for the examples in Problem 1? Can you find formulas where one method outperforms the others dramatically?

Part C: Real-World Applications

Connect theory to practical software engineering.

Task C.1: Configuration Validation

Scenario: You're building a deployment system with configuration constraints (like Problem 5: production mode, debug mode, logging, security).

Implementation:

- Define a configuration schema as propositional formulas (constraints).
- Implement `validate_config(constraints, config)`: check if a configuration satisfies all constraints.
- Implement `find_valid_configs(constraints)`: enumerate *all* valid configurations using your SAT solver or truth table generator.
- Implement `explain_conflict(constraints, config)`: if a config is invalid, report which constraints are violated and suggest fixes.

Test: Use the access control system from Problem 5.

Extension:

- Minimal correction: given an invalid config, find the *smallest* set of variables to flip to make it valid.

Task C.2: SMT Solver Integration

Tool: Use Z3 (Python/C++/Java bindings) or CVC5.

Task:

- Translate your propositional formulas to SMT-LIB or use the API.
- Solve satisfiability: `z3_solve(formula)` ⟶ SAT/UNSAT + model.
- Compare performance with your truth table implementation on large formulas (100+ variables).

Research direction:

- Generate random 3-SAT instances with varying clause/variable ratios.
- Plot satisfiability probability vs. ratio. Observe the *phase transition* around ratio ≈ 4.26.
- Document your findings.

Task C.3: Symbolic Execution (Bonus)

Challenge: Implement a *toy symbolic executor* for a simple imperative language.

Language (example):

```
x := E          // assignment
if B then S1 else S2
assert(B)       // fails if B is false
while B do S    // bounded unrolling
```

Symbolic execution:

- Execute with *symbolic* inputs (variables, not concrete values).
- Track path condition (formula representing choices made).
- At `assert(B)`, check if `path_condition ∧ ¬B` is satisfiable:
  - If SAT ⟶ assertion can fail (counterexample).
  - If UNSAT ⟶ assertion always holds on this path.

Deliverable:
- Implement symbolic executor for assertion checking.
- Test on 2-3 small programs (*e.g.*, array bounds check, login validation).

Open question: How do you handle loops? (Bounded unrolling? Loop invariants?)

Task C.4: Type Checking as Logic (Bonus)

Insight: Type systems are logical systems (Curry-Howard correspondence).

Task:
- Design a simple typed lambda calculus or a subset of a real language (*e.g.*, Simply Typed Lambda Calculus with booleans and integers).
- Encode typing judgments $\Gamma \vdash e : \tau$ as logical formulas or inference rules.
- Implement a type checker using your proof representation from Part B.
- Show that type checking ≡ proof search in a specific logic.

Extension:
- Implement in Lean or Coq: prove type safety (progress + preservation theorems).
- Compare the formal proof to your implementation.

Part D: Formal Verification Track (Optional)

Use Lean 4 or Coq to *prove properties* about your implementations.

Task D.1: Verified Evaluator (Lean/Coq)

Task:
- Define propositional formulas in Lean/Coq as an inductive type.
- Implement `eval : Formula → Valuation → Bool`.
- **Prove:** `eval` is deterministic: `∀ f v, eval f v = eval f v` (trivial, warmup).
- **Prove:** Double negation: `∀ f v, eval (¬¬f) v = eval f v`.
- **Prove:** De Morgan: `∀ f g v, eval (¬(f ∧ g)) v = eval (¬f ∨ ¬g) v`.

Resources:
- Lean 4: Theorem Proving in Lean
- Coq: Software Foundations (Vol. 1, *Logical Foundations*)

Task D.2: Verified CNF Conversion (Lean/Coq)

Challenge:
- Implement NNF conversion in Lean/Coq.
- **Prove correctness:** `∀ f, equivalent f (to_nnf f)` where `equivalent f g := ∀ v, eval f v = eval g v`.
- Prove termination (structural recursion or well-founded relation).

Extension:
- Prove Tseitin transformation preserves *satisfiability* (not equivalence).

Task D.3: Soundness of Proof Checker (Lean/Coq)

Advanced challenge:
- Formalize Fitch-style natural deduction in Lean/Coq.
- Implement proof checking.
- **Prove soundness:** If `check_proof Γ φ proof = true`, then `Γ ⊢ φ` (semantic entailment).

This is research-level work — partial results are valuable. Document your approach and any obstacles.