# Autoformalization with Numina-Lean-Agent

## Euler's partition theorem: from blueprint to a Lean proof, and a choice-free reproof

June 2026

# One theorem, one prover loop

Numina-Lean-Agent turns the natural-language proof in `blueprint_verified.md` into Lean code the Lean kernel checks — the same job as Archon, a different harness.


- You hand Claude the blueprint and one prompt — **you write no Lean**.
- Claude sets up the project, seeds the statement, and runs the loop; the **Lean kernel** is the judge — it cannot be talked into a wrong proof.

Same loop — generate → verify → repair → accept — the kernel decides.

# Euler's partition theorem

A **partition** of $n$ writes $n$ as a sum of positive integers, order ignored.

- **distinct parts:** all summands different. $D(n)$ counts them.
- **odd parts:** every summand odd. $O(n)$ counts them.

Theorem (Euler). For every $n$, $D(n) = O(n)$.

Worked case $n = 6$ (both counts equal $4$):

| | | | | |
|---|---|---|---|---|
| **distinct** | $6$ | $5{+}1$ | $4{+}2$ | $3{+}2{+}1$ |
| **odd** | $5{+}1$ | $3{+}3$ | $3{+}1{+}1{+}1$ | $1{+}1{+}1{+}1{+}1{+}1$ |

Mathlib proves $D(n) = O(n)$ through generating functions — not a bijection and not computable.

The goal here is the **constructive** proof.

# The bijection: binary multiplicities and 2-adic factorization

Write a partition into odd parts by the multiplicity $m(a)$ of each odd part $a$.

- $\Phi$ (odd $\to$ distinct): expand each multiplicity in binary, $m(a) = \sum_{r \geq 0} \varepsilon_r 2^r$; for every set bit $r$, emit the distinct part $2^r \cdot a$.
- $\Psi$ (distinct $\to$ odd): every positive $x$ factors uniquely as $x = 2^r \cdot a$ with $a$ odd; replace $x$ by $2^r$ copies of $a$.

**Example** ($n = 6$): six $1$'s, multiplicity $6 = 110_2$, set bits at $r = 1, 2 \to$ distinct parts $2^1 \cdot 1 = 2$ and $2^2 \cdot 1 = 4$. So $\Phi$ gives $\{2, 4\}$ (sum $6$), and $\Psi$ recovers $\{1, 1, 1, 1, 1, 1\}$.

$\Phi$ and $\Psi$ are inverse: that is the whole proof.

# Stage 1 · Drive Numina with a prompt

# Numina: `claude -p` in a loop, checked by the Lean kernel

Numina-Lean-Agent is a loop (`scripts.run_claude`) that launches headless Claude Code to edit one `.lean` file, then reads the Lean kernel through `lean-lsp-mcp` to check it.

```
each round:  claude -p   (edits the .lean; calls lean_diagnostic_messages)
   END_REASON:LIMIT     -> claude -c -p continue        (next round)
   END_REASON:COMPLETE  -> check: lake env lean -> 0 sorry/0 error ? done
```

- **Bounded:** `--max-rounds N` caps the loop.
- **Recoverable:** `git_commit` after every round — an interrupted run resumes by re-running.
- **You give Claude one prompt**; Claude sets up the project, seeds the statement, runs the loop, and monitors — you write no Lean and never call the tool yourself.

# The prompt you give Claude

You drive the whole run with one natural-language prompt — the blueprint plus what you want. Claude does the rest.

**Prompt — give to Claude:**

```
Use Numina-Lean-Agent to formalize blueprint_verified.md (workspace numina_test).
It may take up to ~100 rounds of editing; recover if it is interrupted.
Make the formalization as computable as possible.
```

Claude then sets up the Lean project, seeds the theorem statement with `sorry`, writes the loop's task prompt and config, runs `run_claude`, and watches the rounds. You read the log.

# What Claude sets up for the loop — you touch none of this

From that one prompt, Claude prepares the run.

**Seeds the statement** (with `sorry`; keeps the target faithful):

```
theorem euler_partition (n : ℕ) :
    (Nat.Partition.distincts n).card = (Nat.Partition.odds n).card := by sorry
```

**Writes the loop's task prompt + config** (constructive, no generating functions, computable):

```
FOLLOW blueprint_verified.md; FORBIDDEN: Glaisher, PowerSeries; COMPUTABLE.
target: Theorem.lean   max_rounds: 100   allow_sorry: false   git_commit: true
$ python -m scripts.run_claude batch numina_config.yaml
```

# Stage 2 · The Euler run

# One round: `euler_partition` via `Finset.card_nbij'`

The run:

- read the blueprint
- defined $\Phi, \Psi$ on `Multiset` $\mathbb{N}$
- proved the 2-adic lemma, well-definedness, both round trips
- assembled the bijection

```
theorem euler_partition (n : ℕ) :
    (distincts n).card = (odds n).card := by
  refine Finset.card_nbij' psiP phiP
    ?_ ?_ ?_ ?_
  · ...   -- Ψ : distinct → odd
  · ...   -- Φ : odd → distinct
  · ...   -- Φ ∘ Ψ = id
  · ...   -- Ψ ∘ Φ = id
```

1 round, $\sim$29 min (Opus); `sorry`: $2 \to 0$.

`Finset.card_nbij'` = equal size from a forward map, an explicit inverse, and proofs each lands in the other set. The statement uses Mathlib's own `distincts`/`odds` — genuinely Euler's theorem; only the proof is new.

# Verification: `#print axioms` and `#eval`

```
lake env lean Theorem.lean      -> 0 sorry, 0 error
#print axioms euler_partition   -> {propext, Classical.choice, Quot.sound}
#eval phi {1,1,1,1,1,1}         -> {2, 4}          -- Φ of 1^6
#eval psi {2,4}                 -> {1,1,1,1,1,1}   -- Ψ inverts it
```

No `sorryAx`. `Classical.choice` comes from the *proof* (as in most Mathlib lemmas); the *maps* are plain `def`s and `#eval`.

**Can the proof avoid `Classical.choice` entirely?**

# Stage 3 · A proof with no `Classical.choice`

# The prompt you give Claude

You do not design the choice-free proof or the gate — you ask one question.

**Prompt — give to Claude:**

```
Can the proof avoid Classical.choice? If so, use Numina-Lean-Agent to
redo it with no Classical.choice -- only {propext, Quot.sound}.
```

Claude finds where the choice axiom enters, builds a compile-time gate that rejects it, and reruns Numina until the proof is choice-free. The next slides are what Claude worked out — you watch.

# `Classical.choice` is in the foundations, not our proof

Not while the statement rests on Mathlib's `Nat.Partition`: the choice axiom is baked into the libraries underneath.

```
Nat.Partition.distincts / odds / Fintype   -> Classical.choice
Nat.factorization  (under ordProj/ordCompl) -> Classical.choice
Finset.card_bij' / card_nbij'               -> Classical.choice
Multiset.bind / dedup / toFinset            -> Classical.choice  (in the DEFs)
-- choice-free: Nat.bitIndices, Finset.sum, Multiset.count/map/sum, List.*
```

So a choice-free proof must drop that infrastructure: a custom 2-adic valuation by recursion; `flatMap`/`dedup` via `Quotient.lift` of `List` ops; and deliver an `Equiv` (`Finset.card` forces choice). Floor: `{propext, Quot.sound}` — `Multiset` is a quotient.

# The gate: make the loop enforce "no `Classical.choice`"

Numina's checker only gates on `sorry`/errors. Add an in-file command that turns a forbidden axiom into a **compile error** — the loop then cannot stop until the proof is both `sorry`-free *and* choice-free.

```
open Lean Elab Command in
elab "#assert_choice_free " id:ident : command => do
  let axs ← liftCoreM <| Lean.collectAxioms id.getId
  if axs.contains ``Classical.choice then
    throwError "✗ {id.getId} depends on Classical.choice : {axs}"
  else logInfo m!"✓ {id.getId} is choice-free"

#assert_choice_free EulerCF.euler_equiv   -- errors until choice-free
```

One intermediate version was `sorry`-free but still used choice; the gate rejected it and the loop kept going until it was purged.

# The result: `euler_equiv`, axioms `{propext, Quot.sound}`

```
def OddPartition (n : ℕ) :=
  { l : Multiset ℕ // (∀ i ∈ l, 0 < i) ∧ l.sum = n ∧ ∀ i ∈ l, Odd i }

def euler_equiv (n : ℕ) : OddPartition n ≃ DistinctPartition n where
  toFun  p := ... phi p.1 ...      -- Φ  (+ well-definedness)
  invFun q := ... psi q.1 ...      -- Ψ
  left_inv  := ... psi_phi ...     -- Ψ ∘ Φ = id
  right_inv := ... phi_psi ...     -- Φ ∘ Ψ = id

#print axioms euler_equiv          -> {propext, Quot.sound}   -- NO choice
#eval (v2 24, oddPart 24)          -> (3, 3)      -- 24 = 2^3·3, choice-free
```

An `Equiv` is the constructive content of "equinumerous", so this *is* $O(n) = D(n)$ — with no `Classical.choice`. (1 round, $\sim$57 min, 761 lines.)

# Same theorem, two artifacts — from a few prompts

```
Theorem.lean    : (distincts n).card = (odds n).card
                  Mathlib-idiomatic; {propext, Classical.choice, Quot.sound}
ChoiceFree.lean : OddPartition n ≃ DistinctPartition n
                  from scratch;     {propext, Quot.sound}
```

Each artifact came from one natural-language prompt to Claude:

```
1. Formalize blueprint_verified.md with Numina-Lean-Agent; up to ~100 rounds;
   recover if interrupted; as computable as possible.       -> Theorem.lean
2. Can the proof avoid Classical.choice? Use Numina to do it. -> ChoiceFree.lean
```

Claude sets up the project, seeds the statement, writes the config, runs the loop, and monitors; you read the round log and the final `#print axioms`.

# Appendix: one Mathlib build, shared to save disk

A compiled Mathlib `.lake` is several GB. Reuse one prebuilt copy across projects instead of rebuilding per project — you ask Claude, Claude wires it up.

**Prompt — give to Claude:**

```
Set this Lean project to reuse my prebuilt shared Mathlib instead of
building its own, to save disk and skip the rebuild.
```

**What Claude does:**

```
ln -s ~/lean-shared/.lake .lake    # reuse the build; match toolchain + rev
```

$N$ projects share one build: $\sim N\times$ less disk, setup $\sim 40$s not hours. Caveat: same `lean-toolchain` + Mathlib rev; give modules distinct names so projects do not clobber each other's `.olean`s. Tool-agnostic (Rethlas / Archon / Numina).