

## Complexitylib: selected formal abstracts

Selected proved results, with their definitions and formal statements. Proof bodies are omitted from the display and remain in the linked Lean sources.

- [Cook–Levin: SAT is NP-complete](#). Every efficiently checkable bit-string problem reduces in polynomial time to satisfiability.
- [The PCP theorem](#). NP proofs can be checked with logarithmic randomness and constantly many nonadaptive queries.
- [Fagin: logic captures NP](#). Existential second-order logic captures precisely the isomorphism-invariant NP properties of finite relational structures.
- [Savitch: NPSPACE equals PSPACE](#). Nondeterminism does not enlarge polynomial space.
- [Reingold: undirected reachability is in L](#). Undirected graph connectivity can be decided with logarithmic work space.
- [Shamir: IP equals PSPACE](#). Interaction and randomness give a polynomial-time verifier the power of polynomial space.
- [Toda: a counting oracle decides the polynomial hierarchy](#). Every problem in the polynomial hierarchy is decidable in polynomial time with an exact counting oracle.
- [Barrington: five states capture  \$NC^1\$](#) . Polynomial-length permutation programs on five states capture logarithmic-depth polynomial-size Boolean circuits.
- [Håstad and Razborov–Smolensky: parity resists shallow circuits](#). At fixed depth, parity requires more than polynomially many gates, including over an odd-prime modular basis.
- [Formula lower bounds: parity and Andreev](#). Tree-shaped computation pays for repeated work: parity requires quadratic leaf size, and Andreev gives a stronger explicit bound.
- [One-way functions yield pseudorandom generators and adaptive functions](#). One-wayness yields pseudorandomness, from an extra output bit to functions secure against adaptive oracle queries.
- [Haken: pigeonhole formulas require long resolution proofs](#). The pigeonhole principle has short CNF statements but requires exponentially many resolution lines.
- [Communication: disjointness and inner product resist randomness](#). Disjointness and parity of intersection require linear communication even with shared randomness and bounded error.
- [Sipser–Gács–Lautemann: BPP lies in the second polynomial level](#). Bounded-error randomized polynomial time admits both existential–universal and universal–existential polynomial descriptions.

## Cook–Levin: SAT is NP-complete

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [sat\\_npComplete](#), [unsat\\_coNPComplete](#)

Every efficiently checkable bit-string problem reduces in polynomial time to satisfiability.

*Scope.* Explicit CNF encoding, total native-machine reductions, and UNSAT coNP-completeness.

A proposed satisfying truth assignment is easy to check. Cook–Levin says that every problem with short, efficiently checkable witnesses can be translated into satisfiability in polynomial time. The translation describes a verifier’s computation by Boolean constraints: a satisfying assignment records an accepting computation. Thus SAT is NP-complete, and its complement UNSAT is coNP-complete.

The definitions below specify CNF formulas, their binary encoding, total polynomial-time reductions and native finite-state machine bounds. The claims are about asymptotic complexity, not a fast practical SAT solver. Lean checks the bridge proofs; `#check_abstract` checks that the statements use the definitions here, the displayed shared foundations, and upstream mathematics.

### References

- S. Arora, B. Barak, *Computational Complexity: A Modern Approach* (AroraBarak09), Section 2.3.
- C. H. Papadimitriou, *Computational Complexity* (Papadimitriou94), Theorem 8.2.

Shared foundation definitions appear in the appendix: [ComplexityAbstracts.Encoding](#), [ComplexityAbstracts.Computation](#).

### Propositional CNF formulas and satisfiability

A literal  $(v, b)$  pairs a natural-number variable index  $v : \mathbb{N}$  with a Boolean polarity  $b : \text{Bool}$ . A clause is a finite list of literals interpreted as a disjunction, and a CNF formula is a finite list of clauses interpreted as a conjunction. Lists are used for finite collections so that algorithms can process them directly; duplicate literals or clauses do not affect truth values.

A literal  $(v, b)$ : variable  $v$  with polarity  $b$  (true for  $v$ , false for  $\neg v$ ).

`abbrev Literal :=  $\mathbb{N} \times \text{Bool}$`

A clause: the disjunction of its literals.

`abbrev Clause := List Literal`

A CNF formula: the conjunction of its clauses.

`abbrev Formula := List Clause`

A truth assignment  $\tau : \mathbb{N} \rightarrow \text{Bool}$  satisfies a CNF formula  $\varphi$  if every clause  $c \in \varphi$  contains at least one literal  $(v, b) \in c$  with  $\tau v = b$ .

```
def Satisfiable ( $\varphi : \text{Formula}$ ) : Prop :=
   $\exists \tau : \mathbb{N} \rightarrow \text{Bool}, \forall c \in \varphi, \exists l \in c, \tau l.1 = l.2$ 
```

SAT is the set of satisfiable CNF formulas.

```
def SAT : Set Formula := { $\varphi \mid \text{Satisfiable } \varphi$ }
```

UNSAT is the set of unsatisfiable CNF formulas. Because instances are formulas rather than raw bitstrings, UNSAT is the exact set-theoretic complement of SAT.

```
def UNSAT : Set Formula := { $\varphi \mid \neg \text{Satisfiable } \varphi$ }
```

ExactThreeSAT is the set of satisfiable CNF formulas in which every clause has length 3 (repeated literals within a clause are permitted).

```
def ExactThreeSAT : Set Formula :=
  { $\varphi \mid (\forall c \in \varphi, c.\text{length} = 3) \wedge \text{Satisfiable } \varphi$ }
```

## Efficient verification and reductions

`Computation.PolyTimeVerifier enc V` means that one deterministic machine halts on every instance/certificate pair in polynomial time in their combined bit length. The machine and polynomial bound are fixed for the problem. We use certificates of exactly  $p(|\text{enc } a|)$  bits; this is the fixed-length certificate convention. `Computation.PolyTime` is the same total computation contract for general input and output encodings. Their definitions are in the foundations appendix.

A function  $V : \alpha \times \text{List Bool} \rightarrow \text{Bool}$  verifies a decision problem  $A$  with certificate length polynomial  $p$  if an instance  $a$  belongs to  $A$  exactly when there is a binary certificate  $w$  of length  $p(|\text{enc } a|)$  accepted by  $V$ .

```
def IsVerifier { $\alpha : \text{Type}$ } (enc :  $\alpha \hookrightarrow \text{List Bool}$ ) (p : Polynomial  $\mathbb{N}$ )
  (V :  $\alpha \times \text{List Bool} \rightarrow \text{Bool}$ ) (A : Set  $\alpha$ ) : Prop :=
   $\forall a, a \in A \leftrightarrow \exists w : \text{List Bool}, w.\text{length} = p.\text{eval } (\text{enc } a).\text{length} \wedge V(a, w) = \text{true}$ 
```

The class NP with respect to instance encoding `enc` consists of decision problems that have a polynomial-time verifier with polynomially long binary certificates.

```
def NP { $\alpha : \text{Type}$ } (enc :  $\alpha \hookrightarrow \text{List Bool}$ ) : Set (Set  $\alpha$ ) :=
  {A  $\mid \exists (p : Polynomial \mathbb{N}) (V : \alpha \times \text{List Bool} \rightarrow \text{Bool}),$ 
    Computation.PolyTimeVerifier enc V  $\wedge$  IsVerifier enc p V A}
```

The class coNP consists of decision problems whose complements belong to NP.

```
def coNP { $\alpha : \text{Type}$ } (enc :  $\alpha \hookrightarrow \text{List Bool}$ ) : Set (Set  $\alpha$ ) :=
  {A  $\mid A^c \in \text{NP enc}$ }
```

Problem A (with encoding  $\text{encA}$ ) many-one reduces in polynomial time to problem B (with encoding  $\text{encB}$ ) if a polynomial-time computable map  $f$  preserves yes/no membership on every instance.

```
def PolyTimeReducible {α β : Type} (encA : α ↦ List Bool) (encB : β ↦ List Bool)
  (A : Set α) (B : Set β) : Prop :=
  ∃ f : α → β, Computation.PolyTime f encA encB ∧ ∀ a, a ∈ A ↔ f a ∈ B
```

A problem B is NP-complete if it belongs to NP and every NP problem (over any instance type  $\alpha$  and injective binary encoding  $\text{encA}$ ) reduces to B in polynomial time.

```
def NPComplete {β : Type} (encB : β ↦ List Bool) (B : Set β) : Prop :=
  B ∈ NP encB ∧
  ∀ (α : Type) (encA : α ↦ List Bool) (A : Set α),
    A ∈ NP encA → PolyTimeReducible encA encB A B
```

A problem B is coNP-complete if it belongs to coNP and every coNP problem reduces to B in polynomial time.

```
def coNPComplete {β : Type} (encB : β ↦ List Bool) (B : Set β) : Prop :=
  B ∈ coNP encB ∧
  ∀ (α : Type) (encA : α ↦ List Bool) (A : Set α),
    A ∈ coNP encA → PolyTimeReducible encA encB A B
```

## Main results

**Theorem (Cook–Levin):** SAT is NP-complete under polynomial-time many-one reductions.

```
theorem sat_npComplete : NPComplete encCNF SAT
```

**Corollary:** UNSAT is coNP-complete under polynomial-time many-one reductions.

```
theorem unsat_coNPComplete : coNPComplete encCNF UNSAT
```

## Appendix: representation details

CNF formulas are lists of clauses with binary variable indices. The following fixed encoding uses the common framing conventions in the appendix.

Encode a CNF formula as a list of clauses, each a list of literals ( $v$ ,  $b$ ) encoded as the pair of  $\text{Nat.bits } v$  and the one-bit word  $[b]$ .

```
def encodeCNF : Formula → List Bool :=
  Encoding.encodeList (Encoding.encodeList (Encoding.encodePair Nat.bits (fun b => [b])))
```

Injective framed binary encoding of CNF formulas.

```
def encCNF : Formula ↦ List Bool :=
  ⟨encodeCNF, encodeCNF_injective⟩
```

## The PCP theorem

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [pcp\\_theorem](#)

NP proofs can be checked with logarithmic randomness and constantly many nonadaptive queries.

*Scope.* Perfect completeness, soundness at most one half, and explicit native efficiency; construction constants are large.

For every problem with certificates checkable in polynomial time, proofs can be presented so that verification reads only constantly many bits. The verifier chooses those positions using only logarithmically many random bits. A correct instance has a proof accepted for every random choice; for an incorrect instance, every purported proof is rejected with probability at least one half. Dinur’s construction repeatedly magnifies the fraction of violated constraints while returning to a fixed alphabet, so a constant number of local checks can detect global inconsistency.

Queries are nonadaptive: their positions are chosen before any proof bits are read. Native machine bounds charge the complete construction, the binary query addresses and verification. The construction is asymptotically polynomial, with enormous fixed constants.

Lean checks the bridge to the complete construction, and `#check_abstract` audits the public statement’s dependency boundary.

### References

- I. Dinur, *The PCP theorem by gap amplification* (Dinur2007)
- S. Arora, B. Barak, *Computational Complexity: A Modern Approach* (AroraBarak09)

Shared foundation definitions appear in the appendix: [ComplexityAbstracts.Encoding](#), [ComplexityAbstracts.Computation](#).

### Efficient verification

`Computation.PolyTimeVerifier enc V` is one deterministic verifier that halts on every pair in polynomial time in  $|enc\ a| + |w|$ . Its code and bound are fixed before either input is chosen. Certificates here have exactly the polynomial length  $p(|enc\ a|)$ .

NP consists of problems with a polynomial-time deterministic verifier and binary certificates of an exact polynomial length. The verifier halts on every input pair.

```
def NP {α : Type} (enc : α ↪ List Bool) : Set (Set α) :=
  {A | ∃ (p : Polynomial ℕ) (V : α × List Bool → Bool),
    Computation.PolyTimeVerifier enc V ∧
    ∀ a, a ∈ A ↔ ∃ w : List Bool, w.length = p.eval (enc a).length ∧ V (a, w) = true}
```

Read a finite proof at the queried addresses. Positions beyond its end read as false.

```
def proofAnswers (proof : List Bool) (positions : List ℕ) : List Bool :=
  positions.map (fun i => proof.getD i false)
```

The number of  $n$ -bit random strings on which the verifier accepts a fixed proof.

```
def acceptCount {α : Type} (queries : α × List Bool → List ℕ)
  (verdict : (α × List Bool) × List Bool → Bool) (a : α) (proof : List Bool) (n : ℕ) : ℕ :=
  (Finset.univ.filter fun r : Fin n → Bool =>
    verdict ((a, List.ofFn r), proofAnswers proof (queries (a, List.ofFn r))) = true).card
```

## Probabilistically checkable proofs

Randomness is a uniformly chosen word of exactly  $\text{coins } a$  bits. Thus the acceptance probability is  $\text{acceptCount} / 2^{(\text{coins } a)}$ . The final inequality below says it is at most one half, without division or rounding. EfficientVerifier charges coin counting, query selection and the final verdict.

Nonadaptive probabilistically checkable proofs: one fixed polynomial-time algorithm computes the number of random bits, the list of binary proof addresses and the final verdict. For constants  $c, d$ , it uses at most  $c \cdot r(n)$  random bits and  $d \cdot q(n)$  queries. A yes-instance has a finite proof accepted for every random string; on a no-instance every finite proof is accepted on at most half of the random strings.

```
def PCP {α : Type} (r q : ℕ → ℕ) (enc : α ↦ List Bool) : Set (Set α) :=
  {A | ∃ (coins : α → ℕ) (queries : α × List Bool → List ℕ)
    (verdict : (α × List Bool) × List Bool → Bool) (c d : ℕ),
    EfficientVerifier enc coins queries verdict ∧
    (∀ a, coins a ≤ c * r (enc a).length) ∧
    (∀ a ρ, ρ.length = coins a → (queries (a, ρ)).length ≤ d * q (enc a).length) ∧
    (∀ a ∈ A, ∃ proof : List Bool, ∀ ρ, ρ.length = coins a →
      verdict ((a, ρ), proofAnswers proof (queries (a, ρ))) = true) ∧
    ∀ a ∉ A, ∀ proof, 2 * acceptCount queries verdict a proof (coins a) ≤ 2 ^ coins a}
```

NP equals logarithmic-randomness, constant-query PCP, with perfect completeness and soundness at most one half. Queries are nonadaptive; their binary addresses and all computation are charged, including the full gap-graph construction.

```
theorem pcg_theorem {α : Type} (enc : α ↦ List Bool) :
  NP enc = PCP (fun n => Nat.log 2 n + 1) (fun _ => 1) enc
```

## Appendix: representation details

All three verifier routines run in polynomial time. Query positions are binary integers, so constructing their addresses is charged; the proof itself is accessed only at those positions.

Native polynomial-time realization of the coin count, query list and verdict.

```
def EfficientVerifier {α : Type} (enc : α ↪ List Bool) (coins : α → ℕ)
  (queries : α × List Bool → List ℕ)
  (verdict : (α × List Bool) × List Bool → Bool) : Prop :=
  Computation.PolyTime coins enc Encoding.encNat ∧
  Computation.PolyTime queries (Encoding.encPair enc Encoding.encBits)
    (Encoding.encList Encoding.encNat) ∧
  Computation.PolyTime verdict
    (Encoding.encPair (Encoding.encPair enc Encoding.encBits) Encoding.encBits) Encoding.encBool
```

## Fagin: logic captures NP

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [fagin](#)

Existential second-order logic captures precisely the isomorphism-invariant NP properties of finite relational structures.

*Scope.* Finite relational signatures, explicit encodings, empty universes and nullary relations.

An existential second-order sentence guesses finite relations and then checks a first-order condition. Fagin's theorem says that this is exactly polynomial-time verification for properties of finite structures that do not depend on how their elements are named. The guessed relations can describe an accepting computation, with local consistency enforced by the formula. Conversely, their truth tables have polynomial size and a fixed first-order condition can be checked in polynomial time.

The vocabulary is finite and relational, with no function symbols or built-in order. Universe size is encoded in unary, followed by framed relation truth tables; incorrect table shapes are excluded from the query language. Empty and singleton universes and nullary relations are included.

Lean checks the explicit logical and computational correspondence through the bridge, and `#check_abstract` audits the public dependency boundary.

### References

- R. Fagin, *Generalized first-order spectra and polynomial-time recognizable sets* (Fagin1974)
- N. Immerman, *Descriptive Complexity* (Immerman1999)

Shared foundation definitions appear in the appendix: [ComplexityAbstracts.Encoding](#), [ComplexityAbstracts.Computation](#).

Relation variables with prescribed arities, used for existential second-order quantifiers. There are no function symbols; a relation symbol is its index together with its arity equation.

```
def relationVariables {ι : Type} (arity : ι → ℕ) : FirstOrder.Language where
  Functions _ := Empty
  Relations n := {i : ι // arity i = n}
```

Interpret a relation variable by the chosen relation on the universe.

```
@[instance_reducible] def relationStructure {ι : Type} {arity : ι → ℕ} {M : Type}
  (R : ∀ i, Set (Fin (arity i) → M)) : (relationVariables arity).Structure M where
  funMap f := isEmptyElim (show Empty from f)
  RelMap r x := x ∘ Fin.cast r.2 ∈ R r.1
```

A class is ESO-definable if one first-order sentence, preceded by finitely many existential relation quantifiers, defines its members among finite structures. Infinite structures are outside the contract. Nullary relation variables and the empty universe are permitted.

```
def ESODefinable {L : FirstOrder.Language}
  (K : Set (CategoryTheory.Bundled.{0} L.Structure)) : Prop :=
  ∃ (ι : Type) (ι : Finite ι) (arity : ι → ℕ)
    (matrix : (L.sum (relationVariables arity)).Sentence),
  ∀ (M : CategoryTheory.Bundled.{0} L.Structure) [Finite M], M ∈ K ↔
    ∃ R : ∀ i, Set (Fin (arity i) → M), letI := relationStructure R;
      FirstOrder.Language.Sentence.Realize M matrix
```

Membership is invariant under isomorphism of finite structures.

```
def IsoClosed {L : FirstOrder.Language} (K : Set (CategoryTheory.Bundled.{0} L.Structure)) :
  Prop :=
  ∀ {{M N : CategoryTheory.Bundled.{0} L.Structure}} [Finite M],
    (FirstOrder.Language.Equiv L M N) → (M ∈ K ↔ N ∈ K)
```

## Efficient verification

Computation.PolyTimeVerifier enc V means one deterministic verifier running in polynomial time in the combined encoded length of its instance and certificate, on every such pair. The fixed certificate length is the polynomial p evaluated at the instance bit length. The machine and encoding conventions are collected in the foundations appendix.

NP consists of problems with a polynomial-time deterministic verifier and binary certificates of an exact polynomial length. The verifier halts on every input pair.

```
def NP {α : Type} (enc : α ↔ List Bool) : Set (Set α) :=
  {A | ∃ (p : Polynomial ℕ) (V : α × List Bool → Bool),
    Computation.PolyTimeVerifier enc V ∧
    ∀ a, a ∈ A ↔ ∃ w : List Bool, w.length = p.eval (enc a).length ∧ V (a, w) = true}
```

Parsed input data: cardinality and one Boolean truth table for each relation symbol.

```
abbrev Input := ℕ × List (List Bool)
```

## The query language

Parsed data is well formed when every relation symbol has a table of exactly  $n^{\text{arity}}$  bits, including one bit for every nullary relation even when  $n=0$ .

```
def WellFormed {L : FirstOrder.Language} {count : ℕ}
  (symbols : Fin count ≈ ∑ n, L.Relations n) (a : Input) : Prop :=
  a.2.length = count ∧ ∀ i : Fin count, (a.2.getD i.val []).length = a.1 ^ (symbols i).1
```

The encoded query contains precisely well-formed data whose decoded finite structure belongs to the class. Incorrect table shapes are excluded.

```
def queryLanguage {L : FirstOrder.Language} [L.IsRelational] {count : ℕ}
  (symbols : Fin count ≈ Σ n, L.Relations n)
  (K : Set (CategoryTheory.Bundled.{0} L.Structure)) : Set Input :=
  {a | WellFormed symbols a ∧
    (⟨Fin a.1, decode symbols a⟩ : CategoryTheory.Bundled L.Structure) ∈ K}
```

Over a finite relational vocabulary, ESO-definability is exactly isomorphism closure on finite structures together with NP membership of the encoded query language.

```
theorem fagin {L : FirstOrder.Language} [L.IsRelational] {count : ℕ}
  (symbols : Fin count ≈ Σ n, L.Relations n) (K : Set (CategoryTheory.Bundled.{0} L.
  Structure)) :
  ESODefinable K ↔ IsoClosed K ∧ queryLanguage symbols K ∈ NP encInput
```

## Appendix: representation details

An input gives the universe size in unary and one truth table per relation symbol. Unary size ensures that input length is at least the universe size, even for an empty vocabulary. A `CategoryTheory.Bundled L.Structure` is simply a universe together with an interpretation of the symbols of `L`. The decoder below uses the universe `Fin a.1`.

Encode the unary cardinality and the framed list of relation truth tables.

```
def encInput : Input ↔ List Bool :=
  Encoding.encPair Encoding.encUnary (Encoding.encList Encoding.encBits)
```

Decode input data according to a finite enumeration of all relation symbols, with arities. The first tuple coordinate is least significant (`finFunctionFinEquiv`); missing entries read false. The theorem's query language separately rejects incorrect table sizes.

```
@[instance_reducible] def decode {L : FirstOrder.Language} [L.IsRelational] {count : ℕ}
  (symbols : Fin count ≈ Σ n, L.Relations n) (a : Input) : L.Structure (Fin a.1) where
  funMap f := isEmptyElim f
  RelMap r x := (a.2.getD (symbols.symm ⟨_, r⟩).val []).getD
    (finFunctionFinEquiv x).val false = true
```

## Savitch: NPSPACE equals PSPACE

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [savitch](#)

Nondeterminism does not enlarge polynomial space.

*Scope.* Native machines and polynomial space; this abstract states class equality rather than a parameterized quadratic-space bound.

A nondeterministic computation can choose among exponentially many paths while using little memory. Savitch's theorem says that a deterministic machine can decide whether an accepting path exists without leaving polynomial space. The key is to test reachability between configurations by trying possible midpoints and checking the two halves in turn, reusing storage between recursive calls. The deterministic computation may take far longer, but its memory remains polynomial.

The statement concerns binary multi-tape machines on encoded inputs. Every nondeterministic branch terminates (a stuck branch rejects), and every computation prefix obeys its space bound. Read-only input and write-only output are not charged. We state equality of the polynomial-space classes, rather than the fully parameterized quadratic-space simulation bound.

Lean checks the transfer through the bridge, and `#check_abstract` audits the public dependencies.

### References

- M. Sipser, *Introduction to the Theory of Computation*, Corollary 8.6 (Sipser2013)
- S. Arora, B. Barak, *Computational Complexity: A Modern Approach*, Section 4.2 (AroraBarak09)

A Boolean is encoded by a single bit.

```
def encBool : Bool ↔ List Bool := ⟨fun b => [b], _⟩
```

PSPACE is total deterministic decision with polynomially bounded work space. There is no time-rate restriction. Read-only input and write-only output are not charged.

```
def PSPACE {α : Type} (enc : α ↔ List Bool) : Set (Set α) :=
  {A | ∃ (f : α → Bool) (t : α → ℕ) (s : ℕ → ℕ),
    Cslib.PolynomiallyBounded s ∧
    Turing.MultiTapeTM.ComputableInTimeAndSpace f enc encBool t (fun a => s (enc a).length) ∧
    ∀ a, a ∈ A ↔ f a = true}
```

A nondeterministic machine accepts if an actual computation path ends halted with output 1.

```
def Accepts {k : ℕ} {State : Type} (M : Turing.MultiTapeNTM k Bool State)
  (input : List Bool) : Prop :=
  ∃ p : M.ComputationPath input, p.last.Halted ∧ p.last.output = [true]
```

Every computation prefix respects the workspace bound, membership is accepting-path existence, and on each input some finite time bounds all paths. A stuck branch rejects.

```
def DecidesInSpace {α : Type} {k : ℕ} {State : Type}
  (M : Turing.MultiTapeNTM k Bool State) (enc : α ↪ List Bool) (A : Set α) (s : α → ℕ) :
Prop :=
  (∀ a, M.RunsInSpace (enc a) (s a) ∧ (a ∈ A ↔ Accepts M (enc a))) ∧
  ∀ a, ∃ t, M.RunsInTime (enc a) t
```

Nondeterministic polynomial space, with one fixed finite-state machine and all branches terminating; the space bound holds along every computation prefix, including rejecting branches.

```
def NPSPACE {α : Type} (enc : α ↪ List Bool) : Set (Set α) :=
  {A | ∃ (k : ℕ) (State : Type) (_ : Finite State) (M : Turing.MultiTapeNTM k Bool State)
    (s : α → ℕ), (∃ b : ℕ → ℕ, Cslib.PolynomiallyBounded b ∧ ∀ a, s a ≤ b (enc a).length) ∧
    DecidesInSpace M enc A s}
```

Savitch's equality of polynomial-space classes. Every nondeterministic computation prefix obeys its space bound, and every branch terminates (a stuck branch rejects). This does not assert the fully parameterized  $\text{NSPACE}(s) \subseteq \text{DSPACE}(s^2)$  theorem.

```
theorem savitch {α : Type} (enc : α ↪ List Bool) : NPSPACE enc = PSPACE enc
```

# Reingold: undirected reachability is in L

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [ustcon\\_mem\\_1](#)

Undirected graph connectivity can be decided with logarithmic work space.

*Scope.* Explicit unary-size header and adjacency-matrix encoding; native logspace, without a reduction-defined SL corollary.

Reingold's theorem decides whether two vertices are joined by a path using too little memory even to mark the vertices already visited: only logarithmic work space. It replaces the graph by a bounded-degree expander construction in which connectivity can be detected by short walks. Evaluating the transformed graph's rotation map in place lets the algorithm enumerate those walks while repeatedly rereading the input instead of retaining the graph.

Instances are simple undirected graphs given by adjacency matrices, with unary headers for the vertex count and the two endpoints. The native machine bound applies to valid encoded instances, including a single vertex joined to itself by the empty walk; malformed-input recognition is outside the statement. Read-only input and write-only output are excluded from work space.

Lean checks the encoding-preserving transfer through the bridge; `#check_abstract` audits its public dependency boundary.

## References

- O. Reingold, *Undirected connectivity in log-space* (Reingold2008)
- H. R. Lewis, C. H. Papadimitriou, *Symmetric space-bounded computation* (LewisPapadimitriou1982)

A Boolean is encoded as one bit.

```
def encBool : Bool ↔ List Bool := ⟨fun b => [b], _⟩
```

A simple graph on  $\text{Fin } n$ , with its symmetric zero-diagonal Boolean adjacency matrix and two distinguished vertices. The sigma type records the common vertex count.

```
abbrev Instance := Σ n : ℕ,
  {adj : Fin n → Fin n → Bool //
    (∀ a b, adj a b = adj b a) ∧ (∀ a, adj a a = false)} × Fin n × Fin n
```

The number of vertices of an instance.

```
def Instance.vertexCount (I : Instance) : ℕ := I.1
```

The symmetric, zero-diagonal adjacency matrix of an instance.

```
def Instance.adjacency (I : Instance) : Fin I.vertexCount → Fin I.vertexCount → Bool :=
  I.2.1.val
```

The source vertex. Its existence ensures that the vertex count is positive.

```
def Instance.source (I : Instance) : Fin I.vertexCount := I.2.2.1
```

The target vertex.

```
def Instance.target (I : Instance) : Fin I.vertexCount := I.2.2.2
```

The graph determined by the adjacency matrix.

```
def graph (I : Instance) : SimpleGraph (Fin I.vertexCount) where
  Adj a b := I.adjacency a b = true
  symm.symm a b h := (I.2.1.property.1 b a).trans h
  loopless.irrefl a h := by simp only [Instance.adjacency, I.2.1.property.2 a,
    Bool.false_eq_true] at h
```

Undirected s–t connectivity: a finite walk connects the distinguished vertices.

```
def USTCON : Set Instance := {I | (graph I).Reachable I.source I.target}
```

Encode the unary header  $1^n 0 1^s 0 1^t 0$ , then the adjacency matrix row by row. `List.ofFn` lists coordinates in increasing order; flattening the list of rows gives exactly  $n^2$  matrix bits. This is the same binary encoding used by the machine theorem.

```
def encode (I : Instance) : List Bool :=
  List.replicate I.vertexCount true ++ false :: (List.replicate I.source.val true ++ false ::
    (List.replicate I.target.val true ++ false ::
      (List.ofFn fun row => List.ofFn (I.adjacency row)).flatten))
```

Injective adjacency-matrix instance encoding.

```
def encInstance : Instance  $\hookrightarrow$  List Bool := ⟨encode, encode_injective⟩
```

One fixed finite-state binary multi-tape machine computes a total function within  $c * (\text{floor}(\log_2(\text{input length})) + 1)$  work-tape cells and halts on every encoded input. Read-only input and write-only output are excluded.

```
def ComputableInLogSpace {α β : Type} (f : α  $\rightarrow$  β)
  (a : α  $\hookrightarrow$  List Bool) (b : β  $\hookrightarrow$  List Bool) : Prop :=
   $\exists$  t c, Turing.MultiTapeTM.ComputableInTimeAndSpace f a b t
  (fun x => c * (Nat.log2 (a x).length + 1))
```

Deterministic logarithmic-space decision.

```
def L {α : Type} (enc : α  $\hookrightarrow$  List Bool) : Set (Set α) :=
  {A |  $\exists$  f : α  $\rightarrow$  Bool, ComputableInLogSpace f enc encBool  $\wedge$   $\forall$  a, a  $\in$  A  $\leftrightarrow$  f a = true}
```

Undirected s-t connectivity is decided in logarithmic work space.

**theorem** `ustcon_mem_1` : `USTCON`  $\in$  L `encInstance`

# Shamir: IP equals PSPACE

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [ip\\_eq\\_pspace](#)

Interaction and randomness give a polynomial-time verifier the power of polynomial space.

*Scope.* All prover strategies are quantified explicitly, with a uniform native clock and specified completeness and soundness.

Interaction lets a verifier challenge a prover about a computation far larger than the verifier could inspect directly. Shamir turns quantified Boolean formulas into arithmetic expressions: the prover supplies polynomial claims, and the verifier checks their consistency at random field points. These local checks suffice to verify every problem decidable in polynomial space, using only polynomial time on the verifier's side. Conversely, the best success probability of such an interaction can be analyzed in polynomial space, giving equality of the two classes.

The verifier is one finite fair-coin oracle machine with a polynomial clock uniform over all stateful randomized provers. Completeness is at least two thirds and soundness at most one third; bounds and correctness apply to encoded instances. The explicit foundational model below distinguishes finite-uniform requests in program syntax from their efficient fair-coin realization.

The definitions and theorem statement below specify the claim. Lean checks the bridge, and the audit permits only the seven listed foundational program and machine modules, with no complexity-class or headline-result allowance.

## References

- C. Lund, L. Fortnow, H. Karloff, N. Nisan, *Algebraic methods for interactive proof systems* (LundFortnowKarloffNisan1992)
- A. Shamir, *IP = PSPACE* (Shamir1992)
- A. Shen, *IP = SPACE: simplified proof* (Shen1992)
- S. Goldwasser, S. Micali, C. Rackoff, *The knowledge complexity of interactive proof systems* (GoldwasserMicaliRackoff1989)

A Boolean is encoded by a single bit.

```
def encBool : Bool ↔ List Bool := ⟨fun b => [b], _⟩
```

A terminating return/request tree. A private request indexed by  $n$  draws a fresh uniform  $F_{in} (n+1)$ ; an external request sends  $((), \text{word})$  and receives a word. Private draws do not change the oracle's state. Bare syntax imposes no efficiency or fair-coin implementability: those are required separately by `StrictOraclePPT`.

```
abbrev InteractiveProgram (β : Type) :=
  Cslib.ProbOracleComp (fun _ : Unit × List Bool => List Bool) β
```

One fixed finite fair-coin machine and one polynomial clock realize the entire joint law of the program's encoded result and the oracle's final state, against every stateful oracle. The clock charges computation, query writing, reply access and output writing. Each instruction costs one unit; the prover's computation is external. The bounded run returns its written output on halting or clock exhaustion; it does not require the unlocked machine to halt.

```
def StrictOraclePPT {α β : Type} (enc : α ↪ List Bool) (out : β ↪ List Bool)
  (program : α → InteractiveProgram β) : Prop :=
  ∃ (k : ℕ) (Control : Type) (_ : Finite Control)
  (machine : Turing.MultiTapePTM k Bool Control Unit) (c d : ℕ),
  ∀ a (State : Type)
  (oracle : Unit × List Bool → StateT State PMF (List Bool)) s,
  Cslib.ProbOracleComp.runWith oracle (out <$> program a) s =
  Cslib.ProbOracleComp.runWith oracle
    (machine.runOracle (c * ((enc a).length + 1) ^ d) (enc a)) s
```

Acceptance probability against a prover with arbitrary private state and probabilistic responses. The response function sees the message and its own state, not the verifier's coins.

```
noncomputable def acceptProbability (program : InteractiveProgram Bool)
  (State : Type)
  (respond : Unit × List Bool → StateT State PMF (List Bool)) (initial : State) : ENNReal :=
  (Cslib.ProbOracleComp.runWith respond program initial).map Prod.fst true
```

Private-coin interactive proofs: strict polynomial verifier, completeness at least 2/3 for some unbounded prover on each member, soundness at most 1/3 against every unbounded stateful randomized prover on every nonmember. The efficiency clock is uniform over all those provers.

```
def IP {α : Type} (enc : α ↪ List Bool) : Set (Set α) :=
  {A | ∃ verifier : α → InteractiveProgram Bool, StrictOraclePPT enc encBool verifier ∧
    (∀ a ∈ A, ∃ (State : Type) (initial : State)
      (respond : Unit × List Bool → StateT State PMF (List Bool)),
      (2 / 3 : ENNReal) ≤ acceptProbability (verifier a) State respond initial) ∧
    ∀ a ∉ A, ∀ (State : Type) (initial : State)
      (respond : Unit × List Bool → StateT State PMF (List Bool)),
      acceptProbability (verifier a) State respond initial ≤ (1 / 3 : ENNReal)}
```

Total deterministic polynomial-space decision, using plain upstream Turing machines. The input is read-only, output write-only and the time bound is unrestricted.

```
def PSPACE {α : Type} (enc : α ↪ List Bool) : Set (Set α) :=
  {A | ∃ (f : α → Bool) (t : α → ℕ) (s : ℕ → ℕ), Cslib.PolynomiallyBounded s ∧
    Turing.MultiTapeTM.ComputableInTimeAndSpace f enc encBool t (fun a => s (enc a).length) ∧
    ∀ a, a ∈ A ↔ f a = true}
```

Private-coin interactive proofs with strict polynomial-time verifiers capture deterministic polynomial space. Provers may be stateful, randomized and unbounded.

```
theorem ip_eq_pspace {α : Type} (enc : α ↪ List Bool) : IP enc = PSPACE enc
```

## Appendix: representation details

The fair-coin machine uses one independent fair bit at each live instruction to choose between two transitions. Ordinary instructions perform an upstream tape action, append at most one bit to a query buffer and move a read-only reply head by at most one cell. A query instruction submits the buffer, installs the reply, resets the reply head and clears the buffer. Query construction and reply access therefore consume the verifier's clock, while the prover's computation does not. Private sampling leaves the prover's state unchanged. The required equality preserves both the verifier's output and the final prover state for every initial state and response strategy.

# Toda: a counting oracle decides the polynomial hierarchy

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [toda](#)

Every problem in the polynomial hierarchy is decidable in polynomial time with an exact counting oracle.

*Scope.* Witness counts and oracle bit answers are explicit; correctness holds for every extension of valid oracle answers.

Counting accepting certificates can be much more powerful than asking whether even one exists. Toda shows that a deterministic algorithm with an exact counting oracle can decide every problem specified by a fixed finite alternation of polynomially bounded existential and universal choices. The proof first expresses these choices through randomized parity tests, then encodes their majority verdict in a binary digit of a witness count. This gives containment of the polynomial hierarchy in polynomial time with counting access, without asserting the reverse containment.

The oracle counts certificates accepted by a deterministic verifier in polynomial time and answers queries about individual binary digits. One native polynomial clock governs the entire stateful interaction, including constructing queries and reading replies. Correctness holds for every extension of valid bit answers to words outside the query encoding's image. The decision program is deterministic. Its resource bound is expressed by exact realization on a clocked fair-coin machine; the same interaction interface also supports randomized programs.

The definitions and theorem statement below specify the claim. Lean checks the bridge, and the audit permits only the listed foundational program and machine modules, with no complexity-class or Toda-theorem allowance.

## References

- S. Toda, *PP is as hard as the polynomial-time hierarchy* (Toda1991)
- S. Arora, B. Barak, *Computational Complexity: A Modern Approach* (AroraBarak09)

Shared foundation definitions appear in the appendix: [ComplexityAbstracts.Encoding](#), [ComplexityAbstracts.Computation](#).

Count the accepted binary strings of exactly  $n$  bits.

```
def acceptCount {α : Type} (V : α × List Bool → Bool) (a : α) (n : ℕ) : ℕ :=
  (Finset.univ.filter fun w : Fin n → Bool => V (a, List.ofFn w) = true).card
```

A #P function counts witnesses of a fixed polynomial length accepted by one total polynomial-time deterministic verifier. Counts are natural numbers in binary.

```
def SharpP {α : Type} (enc : α ↪ List Bool) : Set (α → ℕ) :=
  {f | ∃ (p : Polynomial ℕ) (V : α × List Bool → Bool),
    Computation.PolyTimeVerifier enc V ∧
    ∀ a, f a = acceptCount V a (p.eval (enc a).length)}
```

Alternating quantifiers over fixed-length blocks of bits, with an existential first quantifier when  $q=\text{true}$ . Each subsequent block has the opposite quantifier.

```
def Alternation : Bool → List ℕ → (List Bool → Prop) → Prop
| _, [], V => V []
| true, n :: ns, V => ∃ w : List Bool, w.length = n ∧ Alternation false ns (fun v => V (w ++ v))
| false, n :: ns, V => ∀ w : List Bool, w.length = n → Alternation true ns (fun v => V (w ++ v))
```

The polynomial hierarchy: a fixed finite number of alternating polynomial-length certificate blocks followed by a total polynomial-time verifier. The number of alternations and the length polynomials are fixed for the language, not supplied as part of an instance.

```
def PH {α : Type} (enc : α ↔ List Bool) : Set (Set α) :=
{A | ∃ (ps : List (Polynomial ℕ)) (V : α × List Bool → Bool),
  Computation.PolyTimeVerifier enc V ∧
  ∀ a, a ∈ A ↔ Alternation true (ps.map (fun p => p.eval (enc a).length))
  (fun w => V (a, w) = true)}
```

The counting oracle is accessed by a deterministic program: it can return an answer or make a word query, and later queries may depend on earlier replies. The probabilistic interface below is used only to state its machine resource contract; it adds no random operation to this program.

Deterministic word-query programs; the syntax has queries but no random operation.

```
abbrev QueryProgram (β : Type) := Cslib.OracleComp (fun _ : Unit × List Bool => List Bool) β
```

A terminating return/request tree. A private request indexed by  $n$  draws a fresh uniform  $\text{Fin } (n+1)$ ; an external request sends  $((), \text{word})$  and receives a word. Private draws do not change the oracle's state. Bare syntax imposes no efficiency or fair-coin implementability: those are required separately by `StrictOraclePPT`.

```
abbrev InteractiveProgram (β : Type) :=
  Cslib.ProbOracleComp (fun _ : Unit × List Bool => List Bool) β
```

One fixed finite fair-coin machine and one polynomial clock realize the entire joint law of the program's encoded result and the oracle's final state, against every stateful oracle. The clock charges computation, query writing, reply access and output writing. Each instruction costs one unit; oracle computation is external. The bounded run returns its written output on halting or clock exhaustion; it does not require the unlocked machine to halt.

```
def StrictOraclePPT {α β : Type} (enc : α ↔ List Bool) (out : β ↔ List Bool)
  (program : α → InteractiveProgram β) : Prop :=
  ∃ (k : ℕ) (Control : Type) (_ : Finite Control)
  (machine : Turing.MultiTapePTM k Bool Control Unit) (c d : ℕ),
  ∀ a (State : Type)
  (oracle : Unit × List Bool → StateT State PMF (List Bool)) s,
  Cslib.ProbOracleComp.runWith oracle (out <$> program a) s =
    Cslib.ProbOracleComp.runWith oracle
      (machine.runOracle (c * ((enc a).length + 1) ^ d) (enc a)) s
```

An oracle language replies to each requested word with its one-bit membership verdict.

```
noncomputable def languageOracle (B : Set (List Bool)) (q : Unit × List Bool) : List Bool :=
  [@decide (q.2 ∈ B) (Classical.propDecidable _)]
```

An oracle language answers the bits of  $f$ : every valid query  $(b, i)$  returns the  $i$ th binary digit. Answers to words outside the encoding image are unrestricted.

```
def AnswersBits {β : Type} (enc : β ↦ List Bool) (f : β → ℕ)
  (B : Set (List Bool)) : Prop :=
  ∀ b i, Encoding.encPair enc Encoding.encNat (b, i) ∈ B ↔ (f b).testBit i
```

Deterministic polynomial time with a #P oracle, via bit queries. The program must decide the language against every extension of the valid bit answers to malformed query words. Its native machine clock is polynomial uniformly over every stateful oracle. Interpreting each query through `ProbOracleComp.query` embeds the deterministic program into the shared interaction model without drawing coins. The machine must realize that program's entire joint law exactly.

```
def PWithSharpPOracle {α : Type} (enc : α ↦ List Bool) : Set (Set α) :=
  {A | ∃ (β : Type) (b : β ↦ List Bool) (f : β → ℕ), f ∈ SharpP b ∧
    ∃ program : α → QueryProgram Bool,
      StrictOraclePPT enc Encoding.encBool
        (fun a => Cslib.OracleComp.interpret Cslib.ProbOracleComp.query (program a)) ∧
    ∀ B : Set (List Bool), AnswersBits b f B →
      ∀ a, a ∈ A ↔ Cslib.OracleComp.interpret (m := Id) (languageOracle B) (program a) = true}
```

Every problem in the polynomial hierarchy is decidable by a deterministic polynomial-time oracle program with bit access to a #P function, with uniform native efficiency.

```
theorem toda {α : Type} (enc : α ↦ List Bool) : PH enc ⊆ PWithSharpPOracle enc
```

## Appendix: representation details

The fair-coin machine uses one independent fair bit at each live instruction to choose between two transitions. Ordinary instructions perform an upstream tape action, append at most one bit to a query buffer and move a read-only reply head by at most one cell. A query instruction submits the buffer, installs the reply, resets the reply head and clears the buffer. Thus constructing a query and accessing its answer are charged even though the oracle's own computation is external. The output and final oracle state must agree for every handler, not just for the counting oracle.

# Barrington: five states capture $NC^1$

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [barrington](#)

Polynomial-length permutation programs on five states capture logarithmic-depth polynomial-size Boolean circuits.

*Scope.* Nonuniform circuits and permutation programs; no uniform generator or compiler runtime is asserted.

How much computation can be expressed by repeatedly permuting a set of only five elements? Each instruction reads one selected input bit and chooses one of two permutations; the program composes these choices and tests the resulting permutation against an acceptance set. Barrington's theorem says that polynomial-length programs of this form capture exactly the languages computed by polynomial-size Boolean circuits of logarithmic depth, the class  $NC^1$ .

Both circuit families and program families are nonuniform: a separate object may be chosen at each input length. The theorem asserts no uniform generator or compiler running-time bound. The elementary monoid-program model is the only explicitly allowed implementation module; Lean checks the bridge proof and `#check_abstract` audits the statement boundary.

## References

- D. Barrington, *Bounded-width polynomial-size branching programs recognize exactly those languages in  $NC^1$*  (Barrington1989)
- D. Barrington, D. Thérien, *Finite monoids and the fine structure of  $NC^1$*  (BarringtonTherien1988)

A nonuniform  $NC^1$  language has a polynomial-size De Morgan circuit at each input length, with depth at most a constant times  $\text{floor}(\log_2 n) + 1$ . Every gate, constants included, costs one. Here  $L : \text{Language Bool}$  is a set of finite bit strings.  $L.\text{slice } n \times$  tests whether the word  $\text{List.ofFn } n \times$  belongs to  $L$ ; `single` makes that Boolean the sole circuit output, and `Computes` requires agreement on every input.

```
def InNC1 (L : Language Bool) : Prop :=
  ∃ (c : ℕ) (size : ℕ → ℕ), Cslib.PolynomiallyBounded size ∧ ∀ n,
    ∃ C : Cslib.Circuits.Circuit Cslib.Circuits.Boolean.signature n 1,
      C.Computes Cslib.Circuits.Boolean.interpretation
        (Cslib.Circuits.single (L.slice n)) ∧
        C.depth ≤ c * (Nat.log 2 n + 1) ∧ C.size ≤ size n
```

A width-five permutation program reads one input bit per instruction, selects one of two permutations, and multiplies the selected permutations in instruction order. The elementary `MonoidProgram` model is explicitly allowed; it is a list of (position, selection function) instructions. Its length is polynomial. Both the program and the acceptance set may depend on input length; acceptance tests the entire final permutation.

```
def HasWidthFivePrograms (L : Language Bool) : Prop :=
  ∃ size : ℕ → ℕ, Cslib.PolynomiallyBounded size ∧ ∀ n,
    ∃ (P : Cslib.BranchingPrograms.MonoidProgram (Fin n) Bool (Equiv.Perm (Fin 5)))
      (accept : Set (Equiv.Perm (Fin 5))),
      (∀ x, L.slice n x = true ↔ P.eval x ∈ accept) ∧ P.length ≤ size n
```

**Barrington.** The same language has logarithmic-depth, polynomial-size De Morgan circuits if and only if it has polynomial-length permutation programs on five states.

```
theorem barrington (L : Language Bool) : InNC1 L ↔ HasWidthFivePrograms L
```

# Håstad and Razborov–Smolensky: parity resists shallow circuits

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [hastad](#), [razborov\\_smolensky](#)

At fixed depth, parity requires more than polynomially many gates, including over an odd-prime modular basis.

*Scope.* Explicit finite bounds, gate semantics and depth conventions; nonuniform lower bounds.

An AND or OR gate can inspect arbitrarily many input bits at once. Even with that freedom, a fixed number of layers cannot compute parity with polynomially many gates: as input length grows, every polynomial gate budget eventually fails. Håstad’s argument uses random restrictions; Razborov and Smolensky show that adding modular counting gates of any fixed odd prime modulus still does not make parity easy.

The statements give finite numerical bounds and expose their different depth conventions. These are nonuniform size lower bounds. The elementary gate basis and weighted-depth recursion are the two allowed model modules; their meanings are explained below. Lean checks the bridge proofs, and `#check_abstract` audits the remaining dependency boundary.

## References

- J. Håstad, *Almost optimal lower bounds for small depth circuits* (Hastad1986)
- A. Razborov, *Lower bounds on the size of bounded depth circuits* (Razborov1987)
- R. Smolensky, *Algebraic methods in the theory of lower bounds* (Smolensky1987)
- S. Jukna, *Boolean Function Complexity* (Jukna2012)

Parity is true when an odd number of bits are true.

```
def parity (n : ℕ) (x : Fin n → Bool) : Bool :=
  decide ((Finset.univ.filter fun i => x i = true).card % 2 = 1)
```

The elementary unbounded-fan-in basis has unary NOT and AND/OR of every arity. Nullary AND and OR are the two constants. Its model module is explicitly allowed below.

```
abbrev ACBasis := Cslib.Circuits.Boolean.Unbounded.signature
```

Ordinary Boolean meaning of unbounded AND/OR/NOT.

```
def meaning : Cslib.Circuits.Interpretation ACBasis Bool :=
  Cslib.Circuits.Boolean.Unbounded.interpretation
```

Depth charges one for AND/OR, including their nullary constant cases, and zero for NOT. This weight affects depth only: `C.size` counts every operation gate, including NOT and constants.

```
def andOrCost : Cslib.Circuits.Boolean.Unbounded.Op → ℕ
| .not => 0
| .and _ | .or _ => 1
```

Add one gate of each arity, true when its number of true inputs is nonzero modulo  $p$ .

```
def modularMeaning (p : ℕ) :
  Cslib.Circuits.Interpretation Cslib.Circuits.Boolean.Unbounded.extSignature Bool :=
  Cslib.Circuits.Boolean.Unbounded.extInterpretation
  (fun _ x => decide ((Finset.univ.filter fun i => x i = true).card % p ≠ 0))
```

**Håstad.** A parity circuit of AND/OR depth at most  $d$  needs  $2^s$  gates whenever  $s \geq 2$  and  $17s(16s+1)^{(d-2)} \leq n$ . Subtraction is in naturals. This gives the sharp exponent  $1/(d-1)$  for every fixed  $d \geq 2$ . Negations cost zero in depth and one gate in size.

```
theorem hastad {n d s : ℕ} (C : Cslib.Circuits.Circuit ACBasis n 1)
  (correct : ∀ x, C.eval meaning x 0 = parity n x) (hs : 2 ≤ s)
  (depth : C.depthWith andOrCost ≤ d) (budget : 17 * s * (16 * s + 1) ^ (d - 2) ≤ n) :
  2 ^ s ≤ C.size
```

**Razborov–Smolensky.** For an odd prime  $p$ , even circuits with MOD $_p$  gates need more than polynomially many gates for parity at every fixed depth. This bound counts every gate and every depth level, including NOT and constants; the modulus is explicit and fixed.

```
theorem razborov_smolensky {p n ℓ : ℕ} (prime : p.Prime) (odd : p ≠ 2)
  (C : Cslib.Circuits.Circuit Cslib.Circuits.Boolean.Unbounded.extSignature n 1)
  (correct : ∀ x, C.eval (modularMeaning p) x 0 = parity n x) (hℓ : 1 ≤ ℓ)
  (budget : 100 * (((p - 1) * ℓ) ^ C.depth + 1) ^ 2 ≤ n + 1) :
  2 * 2 ^ ℓ ≤ 5 * C.size
```

## Formula lower bounds: parity and Andreev

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [parity\\_lower\\_bound](#), [andreev\\_lower\\_bound](#)

Tree-shaped computation pays for repeated work: parity requires quadratic leaf size, and Andreev gives a stronger explicit bound.

*Scope.* De Morgan formulas, literal leaves, explicit block indexing and finite quantitative bounds; no general-circuit lower bound.

A circuit may reuse a computed value. A formula is a tree, so using an intermediate result twice requires copying its computation. This restriction makes strong lower bounds possible: parity on  $n$  bits needs at least  $n^2$  input occurrences. Andreev's function asks for a truth-table entry addressed by block parities, giving a stronger bound for an explicit Boolean function.

The model uses binary AND/OR, unary NOT and constants. Size counts input leaves with multiplicity; constants cost zero leaves. The statements concern these De Morgan formulas, not circuits with shared subcomputations or the runtime of constructing a formula. Only the elementary formula-tree model is allowed from the implementation; its constructors and semantics are explained below. Lean checks the bridge proofs and `#check_abstract` audits the statement boundary.

### References

- P. Spira, *On time-hardware complexity tradeoffs for Boolean functions* (Spira1971)
- V. Khrapchenko, *Method of determining lower bounds for the complexity of Pi-schemes* (Khrapchenko1971b)
- S. Yablonskii, *On superpositions of functions of the algebra of logic* (Yablonskii1954)
- B. Subbotovskaya, *Realizations of linear functions by formulas* (Subbotovskaya1961)
- A. Andreev, *On a method of obtaining more than quadratic effective lower bounds* (Andreev1987)
- S. Jukna, *Boolean Function Complexity* (Jukna2012)

A De Morgan expression is a tree of binary AND/OR, unary NOT, constants, and input leaves. Only the elementary tree datatype/evaluator/costs in the explicitly allowed Formula module are used; the lower-bound and balancing implementations are not allowed dependencies.

```
abbrev Expression (n : ℕ) := Cslib.Circuits.Formula Cslib.Circuits.Boolean.signature n
```

The Boolean function computed by an expression under the standard interpretation.

```
def evaluate {n : ℕ} (F : Expression n) : (Fin n → Bool) → Bool :=
  F.eval Cslib.Circuits.Boolean.interpretation
```

Input occurrences are counted with multiplicity. Constants contribute zero leaves.

```
def leaves {n : ℕ} (F : Expression n) : ℕ := F.leafSize
```

Count AND/OR gates on the longest path; NOT and constant gates cost zero.

```
def andOrDepth {n : ℕ} (F : Expression n) : ℕ :=
  F.depthWith (fun op : Cslib.Circuits.Boolean.Op =>
    match op with | .and | .or => 1 | .const _ | .not => 0)
```

The least number of input occurrences in an expression computing  $f$ . Every Boolean function has such an expression; this minimum is a size measure, not a search algorithm.

```
noncomputable def minimumLeaves {n : ℕ} (f : (Fin n → Bool) → Bool) : ℕ :=
  sInf {L | ∃ F : Expression n, (∀ x, evaluate F x = f x) ∧ leaves F = L}
```

Parity is true exactly when an odd number of input bits are true.

```
def parity (n : ℕ) (x : Fin n → Bool) : Bool :=
  decide ((Finset.univ.filter fun i => x i = true).card % 2 = 1)
```

The binary number represented by a word, with coordinate zero least significant: the value is  $\sum_i (if \text{ bits } i \text{ then } 1 \text{ else } 0) * 2^i$ .

```
def tableIndex (n : ℕ) : (Fin n → Bool) ≈ Fin (2 ^ n) :=
  (Equiv.arrowCongr (Equiv.refl (Fin n)) finTwoEquiv.symm).trans finFunctionFinEquiv
```

The first  $2^b$  input bits form the truth table, in address order.

```
def tableInput {b m : ℕ} (x : Fin (2 ^ b + b * m) → Bool) : Fin (2 ^ b) → Bool :=
  fun i => x (Fin.castAdd (b * m) i)
```

The remaining input is  $b$  consecutive blocks of  $m$  bits. Block  $j$ , position  $t$ , is input position  $2^b + m*j + t$ .

```
def blockInput {b m : ℕ} (x : Fin (2 ^ b + b * m) → Bool) (j : Fin b) : Fin m → Bool :=
  fun t => x (Fin.natAdd (2 ^ b) (finProdFinEquiv (j, t)))
```

Andreev's function reads the truth-table entry addressed by the  $b$  block parities. The table is part of the input, not a hardwired oracle. Its bit at address zero is used when there are no blocks ( $b = 0$ ), or when every block is empty ( $m = 0$ ).

```
def andreev (b m : ℕ) (x : Fin (2 ^ b + b * m) → Bool) : Bool :=
  tableInput x (tableIndex b fun j => parity m (blockInput x j))
```

Parity on  $n$  bits requires at least  $n^2$  input leaves, including  $n = 0$ .

```
theorem parity_lower_bound {n : ℕ} (F : Expression n)
  (hF : ∀ x, evaluate F x = parity n x) : n ^ 2 ≤ leaves F
```

**Andreev.** Beyond one fixed cutoff in  $b$ , every  $m \geq \text{floor}(\log_2 b) + 2$  gives the following quantitative formula-size bound. Taking  $m = 2^b$  yields the familiar  $n^{(5/2-o(1))}$  bound, with  $n = (b+1)2^b$ . The theorem does not concern general circuits.

```
theorem andreev_lower_bound :  
  ∃ B : ℕ, ∀ b ≥ B, ∀ m ≥ Nat.log 2 b + 2,  
    2 ^ (2 * b) * m ^ 3 ≤  
      144 * b ^ 2 * (Nat.log 2 b + 2) ^ 3 * minimumLeaves (andreev b m) ^ 2
```

# One-way functions yield pseudorandom generators and adaptive functions

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [hill](#), [adaptive\\_functions](#)

One-wayness yields pseudorandomness, from an extra output bit to functions secure against adaptive oracle queries.

*Scope.* Uniform probabilistic polynomial time with explicit native evaluators, negligible advantage and full stateful oracle semantics; asymptotic security, not a concrete implementation audit.

A one-way function is easy to compute, but finding any preimage of its output on a random input is hard for every efficient randomized algorithm. Assuming such a function, the Håstad–Impagliazzo–Levin–Luby construction produces a generator whose output is one bit longer than its seed and computationally indistinguishable from uniform bits. The Goldreich–Goldwasser–Micali construction then produces a keyed function that looks like one consistently chosen random function even when each query depends on earlier answers. Existence of a one-way function remains a hypothesis; efficiency and security use uniform finite-control machines with strict polynomial bounds and independent fair coins. The definitions and theorem statements below specify the claim; Lean checks their proofs through CryptographyBridge. The audit allows only the listed foundational program, interpretation and machine modules across the implementation boundary, without importing a security definition.

## References

- J. Håstad, R. Impagliazzo, L. Levin, M. Luby, *A Pseudorandom Generator from Any One-Way Function* (HILL1999)
- T. Holenstein, *Pseudorandom Generators from One-Way Functions* (Holenstein2006), Sections 4–5.
- S. Arora, B. Barak, *Computational Complexity* (AroraBarak09), Chapter 9.
- D. Boneh, V. Shoup, *A Graduate Course in Applied Cryptography* (BonehShoup2023), Chapter 3.
- O. Goldreich, S. Goldwasser, S. Micali, *How to Construct Random Functions* (GGM1986)

The typed oracle-program syntax follows VCVio, by Devon Tuma, Quang Dao and collaborators, as credited in `Complexity.Languages.Probabilistic.Basic`.

Shared foundation definitions appear in the appendix: [ComplexityAbstracts.Encoding](#), [ComplexityAbstracts.Computation](#).

Exactly  $n$  independent fair bits, as a word of length  $n$ .

```
noncomputable def uniformBits (n : ℕ) : PMF (List Bool) :=
  (Cslib.Probability.PMF.uniformOfFintype (Fin n → Bool)).map List.ofFn
```

A uniformly strict polynomial-time probability law: one deterministic evaluator, supplied with polynomially many independent fair bits, realizes every input's law exactly. Its time includes reading the coin tape and writing the encoded output. Both evaluator and polynomial are fixed before the input. This also bounds every execution, rather than its expectation.

```
def StrictPPT {α β : Type} (a : α ↪ List Bool) (b : β ↪ List Bool)
  (law : α → PMF β) : Prop :=
  ∃ (c d : ℕ) (evaluate : α → List Bool → β),
    Computation.PolyTime (fun p : α × List Bool => evaluate p.1 p.2)
      (Encoding.encPair a Encoding.encBits) b ∧
      ∀ x, law x = (uniformBits (c * ((a x).length + 1) ^ d)).map (evaluate x)
```

A nonnegative error is negligible if multiplication by every fixed power of the security parameter still tends to zero. The threshold and decay bound may depend on the whole adversary.

```
def Negligible (ε : ℕ → NNReal) : Prop :=
  ∀ k : ℕ, Filter.Tendsto (fun n : ℕ => (n : NNReal) ^ k * ε n)
    Filter.atTop (nhds 0)
```

Probability that a Boolean experiment returns true.

```
noncomputable def probability (p : PMF Bool) : NNReal := (p true).toNNReal
```

Every one uniform strict PPT test has negligible distinguishing advantage. The test receives the unary seed length and a sample. bind samples the word and then runs the test with independent coins; nndist is the absolute difference of success probabilities.

```
def Indistinguishable (X Y : ℕ → PMF (List Bool)) : Prop :=
  ∀ test : ℕ → List Bool → PMF Bool,
    StrictPPT (Encoding.encPair Encoding.encUnary Encoding.encBits) Encoding.encBool
      (fun p => test p.1 p.2) →
    Negligible (fun n => nndist
      (probability (do let sample ← X n; test n sample))
      (probability (do let sample ← Y n; test n sample)))
```

A polynomial-time function is one-way if every strict PPT inverter, given unary  $n$  and  $f(x)$  for a uniform  $n$ -bit  $x$ , finds any preimage only with negligible probability. The candidate's length is unrestricted, and recovering the sampled  $x$  is unnecessary.

```
def OneWay (f : List Bool → List Bool) : Prop :=
  Computation.PolyTime f Encoding.encBits Encoding.encBits ∧
  ∀ inverter : ℕ → List Bool → PMF (List Bool),
    StrictPPT (Encoding.encPair Encoding.encUnary Encoding.encBits) Encoding.encBits
      (fun p => inverter p.1 p.2) →
    Negligible (fun n => probability (do
      let x ← uniformBits n
      (inverter n (f x)).map fun candidate => f candidate == f x))
```

A deterministic polynomial-time generator has exact seed-dependent output length, strictly expands every length, and its uniform-seed outputs are indistinguishable from the uniform distribution at that output length. The security parameter in both games is the seed length. The function need not be injective.

```
def PseudorandomGenerator (g : List Bool → List Bool) (length : ℕ → ℕ) : Prop :=
  Computation.PolyTime g Encoding.encBits Encoding.encBits ∧
  (∀ seed, (g seed).length = length seed.length) ∧ (∀ n, n < length n) ∧
  Indistinguishable (fun n => (uniformBits n).map g) (fun n => uniformBits (length n))
```

**Håstad–Impagliazzo–Levin–Luby:** any uniform one-way function yields a deterministic uniform polynomial-time generator stretching every  $n$ -bit seed to  $n+1$  bits. The proof follows Holenstein’s pseudoentropy route. No entropy estimate or nonuniform advice is an extra hypothesis; the theorem does not assert optimized seed-length exponents or practical efficiency.

```
theorem hill {f : List Bool → List Bool} (hf : OneWay f) :
  ∃ g : List Bool → List Bool, PseudorandomGenerator g (fun n => n + 1)
```

## Adaptive oracle tests and pseudorandom functions

A program is a terminating interaction tree. It can return a result, request an independent uniform sample, or submit a word and branch on the oracle’s reply. Its next requests may depend on earlier replies. Termination alone gives no efficiency: the separate machine certificate below charges ordinary computation, query writing, reply access and output writing.

Programs with private finite-uniform samples and one word-query channel. A request is  $((), \text{word})$  and its reply is a word. Sampling is independent of the external oracle; strict fair-coin realizability is a separate requirement in OraclePPT.

```
abbrev OracleProgram (β : Type) :=
  Cslib.ProbOracleComp (fun _ : Unit × List Bool => List Bool) β
```

Interpret a program against any shared stateful probabilistic handler. Fresh random draws are independent of that handler and leave its private state unchanged; external requests can update it. The result retains both the program’s output and the final handler state.

```
noncomputable def execute {State β : Type} (program : OracleProgram β)
  (oracle : Unit × List Bool → StateT State PMF (List Bool)) (s : State) : PMF (β × State) :=
  Cslib.ProbOracleComp.runWith oracle program s
```

One uniform finite-control oracle machine realizes the entire interaction, at a strict polynomial clock in the encoded unary parameter and auxiliary input. Equality is required for every handler and starting state, preserving the final state as well as the output. The machine cannot return an arbitrarily long reply for free: access and output writing are charged. Each instruction costs one unit; oracle computation is external. The clocked run returns its written output on halting or clock exhaustion. The definition does not require the unclocked machine to halt; this is not a mere bound on the number of oracle queries.

```
def OraclePPT (program : ℕ → List Bool → OracleProgram Bool) : Prop :=
  ∃ (k : ℕ) (Control : Type) (_ : Finite Control)
    (machine : Turing.MultiTapePTM k Bool Control Unit) (c d : ℕ),
  ∀ n aux (State : Type) (oracle : Unit × List Bool → StateT State PMF (List Bool)) s,
  let input := Encoding.encPair Encoding.encUnary Encoding.encBits (n, aux)
  execute (Encoding.encBool <$> program n aux) oracle s =
    execute (machine.runOracle (c * (input.length + 1) ^ d) input) oracle s
```

Run a program against a fixed deterministic word oracle, with independent private samples. The handler has no changing state: Unit is its single state, discarded from the result.

```
noncomputable def evaluateOracle {β : Type} (program : OracleProgram β)
  (oracle : List Bool → List Bool) : PMF β :=
  (Cslib.ProbOracleComp.runWith
    (fun request => StateT.lift (PMF.pure (oracle request.2))) program []).map Prod.fst
```

The real experiment samples one uniform key and consistently answers every valid query with that key's function. Wrong-length queries return the empty word. The adversary receives unary  $n$  and an empty auxiliary input.

```
noncomputable def realPRF (family : List Bool → List Bool → List Bool)
  (adversary : ℕ → List Bool → OracleProgram Bool) (n : ℕ) : PMF Bool := do
  let key ← uniformBits n
  evaluateOracle (adversary n [])
  (fun query => if query.length = n then family key query else [])
```

The ideal experiment samples one entire uniform function from  $n$  bits to  $n$  bits and reuses it across all adaptive calls. This is a probability specification; a lazy cache in the implementation realizes the same law without constructing its exponential truth table. Malformed queries are rejected in exactly the same way as in the real experiment.

```
noncomputable def idealPRF (adversary : ℕ → List Bool → OracleProgram Bool) (n : ℕ) : PMF
  Bool := do
  let table ← Cslib.Probability.PMF.uniformOfFintype ((Fin n → Bool) → Fin n → Bool)
  evaluateOracle (adversary n []) (fun query =>
    if query.length = n then List.ofFn (table fun i => query.getD i.val false) else [])
```

A uniform adaptive pseudorandom function family has a deterministic polynomial-time keyed evaluator,  $n$ -bit answers to valid  $n$ -bit queries under  $n$ -bit keys, and negligible advantage between the preceding games for every one strict uniform adaptive oracle-PPT adversary.

```
def PseudorandomFunction (family : List Bool → List Bool → List Bool) : Prop :=
  Computation.PolyTime (fun p : List Bool × List Bool => family p.1 p.2)
    (Encoding.encPair Encoding.encBits Encoding.encBits) Encoding.encBits ∧
  (∀ key query, query.length = key.length → (family key query).length = key.length) ∧
  ∀ adversary : ℕ → List Bool → OracleProgram Bool, OraclePPT adversary →
    Negligible (fun n => nndist (probability (realPRF family adversary n))
      (probability (idealPRF adversary n))))
```

**One-way functions imply adaptive pseudorandom functions:** HILL, polynomial stretching and the Goldreich–Goldwasser–Micali tree construction give an efficient family satisfying the full adaptive contract above. No fixed query-list restriction or nonuniform advice is assumed.

```
theorem adaptive_functions {f : List Bool → List Bool} (hf : OneWay f) :
  ∃ family : List Bool → List Bool → List Bool, PseudorandomFunction family
```

## Appendix: representation details

The binary multitape machine has finite control and one oracle channel with a write-only query buffer and a read-only reply tape. An ordinary instruction performs an upstream tape action, appends at most one bit to the query buffer and moves the reply head by at most one cell. A query instruction submits the buffer, installs the reply, resets the reply head and clears the buffer. Each live instruction uses one independent fair bit to select between two transitions. The interpreter starts with blank work and communication tapes. Its clock bounds every run, including query construction and reply access, while the handler’s computation is external.

The underlying response family, retained for clients inspecting the program representation: `.inl n` requests a fresh uniform  $\text{Fin } (n+1)$ ; `.inr ((), word)` requests an external word reply.

```
abbrev OracleResponse : Sum ℕ (Unit × List Bool) → Type :=
  Cslib.probOracleSpec (fun _ : Unit × List Bool => List Bool)
```

# Haken: pigeonhole formulas require long resolution proofs

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [haken](#), [pigeonhole\\_satisfiable](#)

The pigeonhole principle has short CNF statements but requires exponentially many resolution lines.

*Scope.* The non-onto pigeonhole formula and a line-based resolution rule, with bound  $2^{\lfloor n/24 \rfloor}$ ; proof size is not solver runtime.

The pigeonhole principle with  $n + 1$  pigeons and  $n$  holes has a CNF of polynomial size in  $n$ , but its resolution refutations need exponentially many lines in  $n$ . This file defines the exact line-proof rule and pigeonhole formula and states Haken’s lower bound, following the Beame–Pitassi bottleneck argument.

A reader can inspect the definitions and theorem statements here without reading proof implementations: `#check_abstract` audits their dependencies, and Lean checks the proofs through `ResolutionBridge`. No internal library model is allowlisted. Proof size counts certificate lines, including input-clause lines; it does not measure solver runtime.

## References

- J. A. Robinson, *A machine-oriented logic based on the resolution principle* (Robinson1965)
- A. Haken, *The intractability of resolution* (Haken1985)
- P. Beame, T. Pitassi, *Simplified and improved resolution lower bounds* (BeamePitassi1996)
- S. Jukna, *Boolean Function Complexity*, Chapter 18 (Jukna2012)

## CNF formulas and resolution certificates

A literal is a variable paired with its Boolean polarity. A clause is a disjunction of its literals, and a formula is a conjunction of clauses. Lists preserve occurrences and order.

A clause over variables  $V$ .

```
abbrev Clause (V : Type) := List (V × Bool)
```

A formula in conjunctive normal form.

```
abbrev CNF (V : Type) := List (Clause V)
```

An assignment satisfies every clause when each clause contains a matching literal.

```
def Satisfiable {V : Type} (φ : CNF V) : Prop :=
  ∃ σ : V → Bool, ∀ c ∈ φ, ∃ l ∈ c, σ l.1 = l.2
```

A resolution proof is listed last line first. A new line is an input clause or is obtained from two earlier lines by deleting all positive occurrences of a pivot in the first and all negative occurrences in the second, then concatenating. Both pivot occurrences are required. There is no weakening rule.

```
def IsProof {V : Type} [DecidableEq V] (φ : CNF V) : List (Clause V) → Prop
| [] => True
| c :: ls => IsProof φ ls ∧ (c ∈ φ ∨ ∃ v c₁ c₂,
  c₁ ∈ ls ∧ c₂ ∈ ls ∧ (v, true) ∈ c₁ ∧ (v, false) ∈ c₂ ∧
  c = c₁.filter (· ≠ (v, true)) ++ c₂.filter (· ≠ (v, false)))
```

A refutation derives the empty clause using at most  $s$  lines.

```
def RefutableInSize {V : Type} [DecidableEq V] (φ : CNF V) (s : ℕ) : Prop :=
  ∃ ls, IsProof φ ls ∧ [] ∈ ls ∧ ls.length ≤ s
```

## The non-onto pigeonhole formula

Variables  $(i, j)$  assert that pigeon  $i$  uses hole  $j$ . There is one positive clause per pigeon and a negative clause for each pair of distinct pigeons and each hole. No clause requires a hole to be occupied, and no clause prohibits one pigeon from occupying several holes.

The formula for  $m$  pigeons and  $n$  holes.

```
def pigeonhole (m n : ℕ) : CNF (Fin m × Fin n) :=
  (List.finRange m).map (fun i => (List.finRange n).map fun j => ((i, j), true)) ++
  (List.finRange n).flatMap fun j => (List.finRange m).flatMap fun i =>
    ((List.finRange m).filter (i < ·)).map fun i' => [((i, j), false), ((i', j), false)]
```

Pigeonhole instances are satisfiable exactly when there are at least as many holes as pigeons. In particular, the refuted family below is unsatisfiable for every  $n$ , including zero.

```
theorem pigeonhole_satisfiable (m n : ℕ) : Satisfiable (pigeonhole m n) ↔ m ≤ n
```

**Haken:** every resolution refutation with  $s$  lines of the formula for  $n + 1$  pigeons and  $n$  holes satisfies  $2^{\lfloor n/24 \rfloor} \leq s$ . The exponential parameter is the number of holes, not the length of an encoding of the CNF. The proof follows the Beame–Pitassi bottleneck argument.

```
theorem haken {n s : ℕ} (hs : RefutableInSize (pigeonhole (n + 1) n) s) :
  2 ^ (n / 24) ≤ s
```

## Communication: disjointness and inner product resist randomness

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [disjointness\\_lower\\_bound](#), [innerProduct\\_lower\\_bound](#), [equality](#)

Disjointness and parity of intersection require linear communication even with shared randomness and bounded error.

*Scope.* Binary protocol trees, worst-case cost and error at most one third on every input pair; explicit finite bounds and the final-answer convention.

Alice and Bob each hold an  $n$ -bit string and exchange bits to compute a joint answer. Disjointness and parity of the intersection both require linearly many communicated bits, even with shared randomness and error at most one third on every input pair.

Local computation is free. Cost is the worst case over inputs and coins; protocols may be chosen separately at each input length. Both players learn the answer, so exact deterministic equality costs  $n + 1$  bits for  $n > 0$ : Alice sends her input and Bob replies with the answer.

The only allowlisted model is the binary protocol tree described below. Cost and per-input error are defined here. Lean checks the bridge proofs; `#check_abstract` audits their statement dependencies. The disjointness constants are not optimized.

### References

- B. Kalyanasundaram, G. Schnitger, *The probabilistic communication complexity of set intersection* (KalyanasundaramSchnitger1992)
- A. Razborov, *On the distributional complexity of disjointness* (Razborov1992)
- E. Kushilevitz, N. Nisan, *Communication Complexity*, Chapters 1–4 (KushilevitzNisan1997)

A bit string is a function assigning a Boolean to each of  $n$  positions.

```
abbrev Input (n : ℕ) := Fin n → Bool
```

A finite binary tree: output  $b$  is a leaf announcing  $b$ ; `alice_send p0 p1` branches on the bit `send x` of Alice's input, and `bob` branches on Bob's input. The transcript determines the current node, so messages may depend on all earlier messages.

```
abbrev Protocol (n : ℕ) := Cslib.Communication.Protocol (Input n) (Input n) Bool
```

Follow the sent bits from the root to a leaf and return its announced answer.

```
def answer {n : ℕ} (p : Protocol n) (x y : Input n) : Bool := p.run x y
```

Tree depth: leaves cost zero, and sending one bit costs one plus the larger branch cost.

```
def cost {n : ℕ} (p : Protocol n) : ℕ := p.depth
```

Shared randomness selects a deterministic tree independently of the inputs. PMF is a probability distribution with total mass one.

```
abbrev RandomizedProtocol (n : ℕ) := PMF (Protocol n)
```

Every tree with positive probability has depth at most  $c$ , bounding all its branches.

```
def CostAtMost {n : ℕ} (μ : RandomizedProtocol n) (c : ℕ) : Prop :=
  ∀ p ∈ μ.support, cost p ≤ c
```

On each input pair separately, the probability of a wrong answer is at most  $\varepsilon$ . The measure sums the probabilities of trees giving a wrong answer; toReal returns a real.

```
def ErrorAtMost {n : ℕ} (μ : RandomizedProtocol n)
  (f : Input n → Input n → Bool) (ε : ℝ) : Prop :=
  ∀ x y, (μ.toOuterMeasure {p | answer p x y ≠ f x y}).toReal ≤ ε
```

The sets of true positions are disjoint exactly when no position is true in both inputs.

```
def disjoint {n : ℕ} (x y : Input n) : Bool :=
  decide (∀ i, (x i && y i) = false)
```

The inner product modulo two: true when an odd number of positions are true in both inputs. The sum below uses Boolean multiplication (AND) and addition (XOR).

```
def innerProduct {n : ℕ} (x y : Input n) : Bool := ∑ i, x i * y i
```

**Exact equality.** A deterministic protocol deciding equality within  $c$  bits exists exactly when  $n + 1 \leq c$ , for positive input length. At length zero, the constant true answer costs no communication.

```
theorem equality {n c : ℕ} (hn : 0 < n) :
  (∃ p : Protocol n, (∀ x y, answer p x y = decide (x = y)) ∧ cost p ≤ c) ↔ n + 1 ≤ c
```

**Kalyanasundaram–Schnitger / Razborov.** Even with shared randomness and error at most one third, disjointness needs linear communication:  $n \leq 65536 \cdot c + 4096$ . The proof uses Razborov’s corruption method, with a variant of his hard distribution.

```
theorem disjointness_lower_bound {n c : ℕ} (μ : RandomizedProtocol n)
  (hc : CostAtMost μ c) (he : ErrorAtMost μ disjoint (1 / 3)) :
  n ≤ 2 ^ 16 * c + 2 ^ 12
```

**Inner product.** Even with shared randomness and error at most one third, computing parity of the intersection requires  $n \leq 2 \cdot c + 3$ . The proof bounds the discrepancy of the inner-product sign matrix using its orthogonal rows.

```
theorem innerProduct_lower_bound {n c : ℕ} (μ : RandomizedProtocol n)
  (hc : CostAtMost μ c) (he : ErrorAtMost μ innerProduct (1 / 3)) :
  n ≤ 2 * c + 3
```

# Sipser–Gács–Lautemann: BPP lies in the second polynomial level

Source revision [247a4399206e](#) / [Lean source](#)

Statements: [sipserLautemann](#)

Bounded-error randomized polynomial time admits both existential–universal and universal–existential polynomial descriptions.

*Scope.* Bit-string languages, fair independent random bits, correctness at least two thirds on every input, and total native polynomial-time tests; no derandomization assumption.

Every bounded-error randomized polynomial-time language admits both existential–universal and universal–existential descriptions by polynomially long binary blocks and a deterministic polynomial-time test:  $\text{BPP} \subseteq \Sigma_2 \cap \Pi_2$ .

Lautemann’s proof amplifies the randomized algorithm. For a yes-instance, polynomially many xor shifts of its accepting seeds cover every seed; for a no-instance, no such cover exists. Guessing the shifts and checking every seed yields the existential–universal description. Complement closure yields the reverse order.

Lean checks the proof bridge and audits the abstract’s statement dependencies.

## References

- M. Sipser, *A complexity theoretic approach to randomness* (Sipser1983).
- C. Lautemann, *BPP and the polynomial hierarchy* (Lautemann1983).
- S. Arora, B. Barak, *Computational Complexity: A Modern Approach* (AroraBarak09), Theorem 7.15.

## Bit strings and deterministic polynomial-time tests

Inputs are bit strings. A test reads an instance  $x$  and a second string  $w$  of random bits or certificates. Encode the pair by framing each bit  $b$  of  $x$  as  $[\text{true}, b]$ , terminating with  $[\text{false}]$ , then appending  $w$ . The separator preserves the boundary.

Encode a pair  $(x, w)$  by the self-delimiting frame of  $x$ , followed by  $w$ .

```
def encodeInput (z : List Bool × List Bool) : List Bool :=
  z.1.flatMap (fun b => [true, b]) ++ [false] ++ z.2
```

Injective binary encoding of the two input strings.

```
def encInput : (List Bool × List Bool) ↪ List Bool := ⟨encodeInput, encodeInput_injective⟩
```

A Boolean answer is encoded as its single bit.

```
def encAnswer : Bool ↪ List Bool := ⟨fun b => [b], _⟩
```

One deterministic binary multi-tape machine halts on every encoded pair and writes  $V$ 's answer, within time and work-space bounds polynomial in the pair's encoded length. `Cslib.PolynomiallyBounded` means bounded above by a polynomial.

```
def PolyTimePredicate (V : List Bool × List Bool → Bool) : Prop :=
  ∃ t s : ℕ → ℕ, Cslib.PolynomiallyBounded t ∧ Cslib.PolynomiallyBounded s ∧
    Turing.MultiTapeTM.ComputableInTimeAndSpaceOfLength V encInput encAnswer t s
```

## Randomness and two quantifier blocks

BPP languages have a polynomial-time test with  $p(|x|)$  fair independent random bits, correct on at least two thirds of seeds for each input. The inequalities count correct seeds out of  $2^{p(|x|)}$ . A function  $\text{Fin } m \rightarrow \text{Bool}$  represents an  $m$ -bit seed; `List.ofFn` lists its bits.

```
def BPP : Set (Set (List Bool)) :=
  {A | ∃ (p : Polynomial ℕ) (V : List Bool × List Bool → Bool),
    PolyTimePredicate V ∧ ∀ x,
      (x ∈ A → 2 * 2 ^ p.eval x.length ≤
        3 * (Finset.univ.filter fun r : Fin (p.eval x.length) → Bool =>
          V (x, List.ofFn r) = true).card) ∧
      (x ∉ A → 2 * 2 ^ p.eval x.length ≤
        3 * (Finset.univ.filter fun r : Fin (p.eval x.length) → Bool =>
          V (x, List.ofFn r) = false).card)}
```

$\Sigma_2$  uses an existential block  $u$  followed by a universal block  $v$ , with polynomial lengths and a total polynomial-time test of  $u ++ v$ . Block lengths depend only on the input length.

```
def SigmaTwo : Set (Set (List Bool)) :=
  {A | ∃ (p q : Polynomial ℕ) (V : List Bool × List Bool → Bool),
    PolyTimePredicate V ∧ ∀ x, x ∈ A ↔
      ∃ u : List Bool, u.length = p.eval x.length ∧
        ∀ v : List Bool, v.length = q.eval x.length → V (x, u ++ v) = true}
```

$\Pi_2$  reverses the order: every polynomial-length block  $u$  has a polynomial-length response  $v$  passing the test. The response may depend on the instance and  $u$ . The two classes may use different tests and length polynomials.

```
def PiTwo : Set (Set (List Bool)) :=
  {A | ∃ (p q : Polynomial ℕ) (V : List Bool × List Bool → Bool),
    PolyTimePredicate V ∧ ∀ x, x ∈ A ↔
      ∀ u : List Bool, u.length = p.eval x.length →
        ∃ v : List Bool, v.length = q.eval x.length ∧ V (x, u ++ v) = true}
```

**Sipser–Gács–Lautemann.** Every bounded-error randomized polynomial-time language has both polynomial two-block descriptions:  $\text{BPP} \subseteq \Sigma_2 \cap \Pi_2$ .

```
theorem sipserLautemann : BPP ⊆ SigmaTwo ∩ PiTwo
```

## Appendix: shared foundations

These checked definitions specify the representations and machine bounds used above. They are displayed once here; their Lean sources remain part of the document’s provenance.

### Binary input conventions

[Lean source](#). Namespaces: ComplexityTests.Abstracts.Encoding.

Declarations: `frame`, `encodePair`, `encodeList`, `encodePair_injective`, `encodeList_injective`, `bits_injective`, `encPair`, `encList`, `encNat`, `encBool`, `encBits`, `encUnary`.

An encoding is an injective map to finite bitstrings;  $\hookrightarrow$  records injectivity. These fixed conventions specify the inputs and outputs of the machines used in the abstracts. Framing makes component boundaries recoverable with linear overhead. Natural numbers use binary digits unless an abstract explicitly supplies a unary parameter. Changing binary to unary can change a complexity claim. The delimiter convention below fixes the precise machine input.

The conventions are those of Christian Reitwiessner’s native-machine encoding API. Their exact definitions are collected here so each theorem can be read before its serialization details.

Make a word self-delimiting: tag each bit by true and finish with false.

```
def frame (w : List Bool) : List Bool := w.flatMap (fun b => [true, b]) ++ [false]
```

Frame the first component, then append the second.

```
def encodePair {α β : Type} (a : α → List Bool) (b : β → List Bool) (p : α × β) :
  List Bool := frame (a p.1) ++ b p.2
```

Concatenate the framed elements of a list.

```
def encodeList {α : Type} (enc : α → List Bool) (l : List α) : List Bool :=
  l.flatMap (frame ∘ enc)
```

Injective encoding of a pair. Its length is  $2 * |a \ x| + 1 + |b \ y|$ .

```
def encPair {α β : Type} (a : α ↪ List Bool) (b : β ↪ List Bool) : α × β ↪ List Bool :=
  ⟨encodePair a b, encodePair_injective a b⟩
```

Injective encoding of a list.

```
def encList {α : Type} (enc : α ↪ List Bool) : List α ↪ List Bool :=
  ⟨encodeList enc, encodeList_injective enc⟩
```

Canonical binary digits, least significant first; zero has the empty encoding.

```
def encNat : ℕ ↪ List Bool := ⟨Nat.bits, bits_injective⟩
```

A Boolean is a single bit.

```
def encBool : Bool  $\hookrightarrow$  List Bool := ⟨fun b => [b], _⟩
```

A bitstring is its own encoding.

```
def encBits : List Bool  $\hookrightarrow$  List Bool := Function.Embedding.refl _
```

Unary size or security parameter: exactly  $n$  bits.

```
def encUnary :  $\mathbb{N}$   $\hookrightarrow$  List Bool := ⟨fun n => List.replicate n true, _⟩
```

## Polynomial-time functions and verifiers

[Lean source](#). Namespaces: ComplexityTests.Abstracts.Computation.

Declarations: [PolyTime](#), [PolyTimeVerifier](#).

One deterministic binary multi-tape Turing machine, with finitely many states, computes the function on every encoded input. Its time and work space are polynomially bounded in input bit length. Input is read-only and output is write-only. The machine is uniform: its code and its bounds are fixed before the input is chosen. No condition is imposed on words outside an encoding's image.

A verifier receives an instance and a certificate. Its running time is polynomial in their combined encoded length, not just the instance length. The pair encoding has length  $2 * |enc\ x| + 1 + |w|$ , so it gives the usual polynomial bound in  $|enc\ x| + |w|$ .

Total polynomial-time computation on encoded inputs, with explicit time and space bounds.

```
def PolyTime { $\alpha$   $\beta$  : Type} (f :  $\alpha \rightarrow \beta$ ) (a :  $\alpha \hookrightarrow$  List Bool) (b :  $\beta \hookrightarrow$  List Bool) : Prop :=
   $\exists$  t s :  $\mathbb{N} \rightarrow \mathbb{N}$ , Cslib.PolynomiallyBounded t  $\wedge$  Cslib.PolynomiallyBounded s  $\wedge$ 
    Turing.MultiTapeTM.ComputableInTimeAndSpaceOfLength f a b t s
```

One total polynomial-time deterministic verifier, on an instance and a binary certificate.

```
def PolyTimeVerifier { $\alpha$  : Type} (enc :  $\alpha \hookrightarrow$  List Bool) (V :  $\alpha \times$  List Bool  $\rightarrow$  Bool) : Prop :=
  PolyTime V (Encoding.encPair enc Encoding.encBits) Encoding.encBool
```