# Bounded Arithmetic, Part I

Raheleh Jalali

Proof Society 2025

# Outline

1. Motivation
2. A very short course on complexity
3. Syntax of bounded arithmetic
4. (Induction) axioms for bounded arithmetic
5. Theories of bounded arithmetic

# Motivation: capturing feasible reasoning/computation

- *Effective computation*: an intuitive/informal notion.

# Motivation: capturing feasible reasoning/computation

- *Effective computation*: an intuitive/informal notion.
- Church–Turing thesis: a problem is effectively computable iff it is recursively computable iff can be decided by a Turing machine.

# Motivation: capturing feasible reasoning/computation

- *Effective computation*: an intuitive/informal notion.
- Church–Turing thesis: a problem is effectively computable iff it is recursively computable iff can be decided by a Turing machine.
- However, it may not be *feasibly* computable, since it may need enormous computational resources (time or space).

# Motivation: capturing feasible reasoning/computation

- *Effective computation*: an intuitive/informal notion.
- Church–Turing thesis: a problem is effectively computable iff it is recursively computable iff can be decided by a Turing machine.
- However, it may not be *feasibly* computable, since it may need enormous computational resources (time or space).
- By feasible (also an intuitive notion) we mean computable in practice or in real world.

# Motivation: capturing feasible reasoning/computation

- *Effective computation*: an intuitive/informal notion.
- Church–Turing thesis: a problem is effectively computable iff it is recursively computable iff can be decided by a Turing machine.
- However, it may not be *feasibly* computable, since it may need enormous computational resources (time or space).
- By feasible (also an intuitive notion) we mean computable in practice or in real world.
- It's natural to look for a feasible analog of Church's thesis to provide a formal mathematical model for feasible computation.
  An advantage: mathematically investigating the power and limits of feasible computation.

# Motivation: capturing feasible reasoning/computation

- *Effective computation*: an intuitive/informal notion.
- Church–Turing thesis: a problem is effectively computable iff it is recursively computable iff can be decided by a Turing machine.
- However, it may not be *feasibly* computable, since it may need enormous computational resources (time or space).
- By feasible (also an intuitive notion) we mean computable in practice or in real world.
- It's natural to look for a feasible analog of Church's thesis to provide a formal mathematical model for feasible computation.
  An advantage: mathematically investigating the power and limits of feasible computation.
- It is widely believed that polynomial time computation is the correct mathematical model for feasible computation.

# Example: Algorithms for multiplication

Let's see two algorithms for the multiplication of two integers; the first is effective but not feasible, while the second is effective and feasible.

- Suppose integers $x$ and $y$ are given in binary notation.

# Example: Algorithms for multiplication

Let's see two algorithms for the multiplication of two integers; the first is effective but not feasible, while the second is effective and feasible.

- Suppose integers $x$ and $y$ are given in binary notation.
- Let $|x|$ denote the length of $x$, the number of bits in the binary representation of $x$.

# Example: Algorithms for multiplication

Let's see two algorithms for the multiplication of two integers; the first is effective but not feasible, while the second is effective and feasible.

- Suppose integers $x$ and $y$ are given in binary notation.
- Let $|x|$ denote the length of $x$, the number of bits in the binary representation of $x$.
- Take a feasible algorithm for addition (which we will use as a subroutine in our algorithms). The algorithm computes the addition of $x$ and $y$ in $|x| + |y|$ steps, where a step is a constant number of bit operations.

# Example: Algorithms for multiplication

Let's see two algorithms for the multiplication of two integers; the first is effective but not feasible, while the second is effective and feasible.

- Suppose integers $x$ and $y$ are given in binary notation.
- Let $|x|$ denote the length of $x$, the number of bits in the binary representation of $x$.
- Take a feasible algorithm for addition (which we will use as a subroutine in our algorithms). The algorithm computes the addition of $x$ and $y$ in $|x| + |y|$ steps, where a step is a constant number of bit operations.

**Example (Algorithm 1 for computing $x \cdot y$)**

Add $x$ to itself $y - 1$ times. Runtime: $(y - 1) \cdot (|x| + |y|)$ steps.

# Example Cont'

## Example (Algorithm 2)

The usual long multiplication (also called grade-school or standard). For instance, multiplying 6 and 5 in binary notation:

$$\begin{array}{rl} 110 & (=6) \\ \underline{\times\, 101} & (=5) \\ 110 & \\ 000\phantom{0} & \\ \underline{+\,110\phantom{00}} & \\ 11110 & (=30) \end{array}$$

The runtime: approximately $|x| \cdot |y|$ steps.

# Comparison of two algorithms

- Feasibility vs infeasibility: let $x, y$ be 20-digit numbers and each step takes one microsecond time.

# Comparison of two algorithms

- Feasibility vs infeasibility: let $x, y$ be 20-digit numbers and each step takes one microsecond time.
- As $x \approx 10^{20}$ , we have $|x| \approx 70$. Likewise, $|y| \approx 70$.

# Comparison of two algorithms

- Feasibility vs infeasibility: let $x, y$ be 20-digit numbers and each step takes one microsecond time.
- As $x \approx 10^{20}$ , we have $|x| \approx 70$. Likewise, $|y| \approx 70$.
- The runtime of Algorithm 1:

$$\approx 140 * 10^{20} \mu s \approx 1000 \times \text{(age of the universe)}.$$

# Comparison of two algorithms

- Feasibility vs infeasibility: let $x, y$ be 20-digit numbers and each step takes one microsecond time.
- As $x \approx 10^{20}$ , we have $|x| \approx 70$. Likewise, $|y| \approx 70$.
- The runtime of Algorithm 1:

$$\approx 140 * 10^{20} \mu s \approx 1000 \times (\text{age of the universe}).$$

- The runtime of Algorithm 2:

$$\approx 70^2 \mu s = 4900 s \approx \frac{1}{100} s$$

# Comparison of two algorithms

- Feasibility vs infeasibility: let $x, y$ be 20-digit numbers and each step takes one microsecond time.
- As $x \approx 10^{20}$ , we have $|x| \approx 70$. Likewise, $|y| \approx 70$.
- The runtime of Algorithm 1:

$$\approx 140 * 10^{20} \mu s \approx 1000 \times (\text{age of the universe}).$$

- The runtime of Algorithm 2:

$$\approx 70^2 \mu s = 4900s \approx \frac{1}{100} s$$

- It is hopeless to try running Algorithm 1, but Algorithm 2 is very quick on modern computers.

# Bounded Arithmetic, feasible reasoning and Complexity

Bounded arithmetic (BA) shifts the study of complexity from computations to reasoning.

- Bounded arithmetic: A collective name for a family of weak fragments/subtheories of Peano Arithmetic (PA); contains induction schemas only for restricted formulas; do not prove the totality of the exponentiation function.

# Bounded Arithmetic, feasible reasoning and Complexity

Bounded arithmetic (BA) shifts the study of complexity from computations to reasoning.

- Bounded arithmetic: A collective name for a family of weak fragments/subtheories of Peano Arithmetic (PA); contains induction schemas only for restricted formulas; do not prove the totality of the exponentiation function.
- Initiated by Parikh (1971), developed by Buss (1986).

# Bounded Arithmetic, feasible reasoning and Complexity

Bounded arithmetic (BA) shifts the study of complexity from computations to reasoning.

- Bounded arithmetic: A collective name for a family of weak fragments/subtheories of Peano Arithmetic (PA); contains induction schemas only for restricted formulas; do not prove the totality of the exponentiation function.
- Initiated by Parikh (1971), developed by Buss (1986).
- Has relations to the polynomial time hierarchy and gives another viewpoint to open questions such as P vs NP.

# Outline

1. Motivation
2. A very short course on complexity
3. Syntax of bounded arithmetic
4. (Induction) axioms for bounded arithmetic
5. Theories of bounded arithmetic

# Some complexity classes

- With $n$ as input size, polynomial time means runtime $\leqslant n^c$, while exponential time means runtime $\leqslant 2^{n^c}$, for some constant $c$. (So, polynomial time means in terms of the length $|x|$ of the input $x$.)

# Some complexity classes

- With $n$ as input size, polynomial time means runtime $\leqslant n^c$, while exponential time means runtime $\leqslant 2^{n^c}$, for some constant $c$. (So, polynomial time means in terms of the length $|x|$ of the input $x$.)
- P: the collection of decision problems (problems with a "yes"/"no" answer) solvable by a deterministic machine in polynomial time.

# Some complexity classes

- With $n$ as input size, polynomial time means runtime $\leqslant n^c$, while exponential time means runtime $\leqslant 2^{n^c}$, for some constant $c$. (So, polynomial time means in terms of the length $|x|$ of the input $x$.)
- P: the collection of decision problems (problems with a "yes" / "no" answer) solvable by a deterministic machine in polynomial time.
- A nondeterministic Turing machine makes "choices" or "guesses": the machine accepts its input iff there is at least one sequence of choices that leads to an accepting state.

# Some complexity classes

- With $n$ as input size, polynomial time means runtime $\leqslant n^c$, while exponential time means runtime $\leqslant 2^{n^c}$, for some constant $c$. (So, polynomial time means in terms of the length $|x|$ of the input $x$.)
- P: the collection of decision problems (problems with a "yes" / "no" answer) solvable by a deterministic machine in polynomial time.
- A nondeterministic Turing machine makes "choices" or "guesses": the machine accepts its input iff there is at least one sequence of choices that leads to an accepting state.
- NP: the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

# Some complexity classes

- With $n$ as input size, polynomial time means runtime $\leqslant n^c$, while exponential time means runtime $\leqslant 2^{n^c}$, for some constant $c$. (So, polynomial time means in terms of the length $|x|$ of the input $x$.)
- P: the collection of decision problems (problems with a "yes" / "no" answer) solvable by a deterministic machine in polynomial time.
- A nondeterministic Turing machine makes "choices" or "guesses": the machine accepts its input iff there is at least one sequence of choices that leads to an accepting state.
- NP: the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.
- coNP: If a problem $X$ is in NP, then its complement $X^c$ is in coNP.

# Some complexity classes

- With $n$ as input size, polynomial time means runtime $\leqslant n^c$, while exponential time means runtime $\leqslant 2^{n^c}$, for some constant $c$. (So, polynomial time means in terms of the length $|x|$ of the input $x$.)
- P: the collection of decision problems (problems with a "yes" / "no" answer) solvable by a deterministic machine in polynomial time.
- A nondeterministic Turing machine makes "choices" or "guesses": the machine accepts its input iff there is at least one sequence of choices that leads to an accepting state.
- NP: the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.
- coNP: If a problem $X$ is in NP, then its complement $X^c$ is in coNP.
- Example: SAT $\in$ NP and TAUT $\in$ coNP.

# Polynomial time hierarchy

A hierarchy of complexity classes that generalize the classes NP and co-NP.

Define:

$$\Delta_0^P := \Sigma_0^P := \Pi_0^P := P$$

and

$$\Delta_{i+1}^P := P^{\Sigma_i^P}$$
$$\Sigma_{i+1}^P := NP^{\Sigma_i^P}$$
$$\Pi_{i+1}^P := coNP^{\Sigma_i^P}$$

where $P^A$ is the set of decision problems solvable in polynomial time by a Turing machine augmented by an oracle for some problem in class A. For instance, $\Sigma_1^P = NP$, $\Pi_1^P = coNP$, and $\Delta_2^P = P^{NP}$.

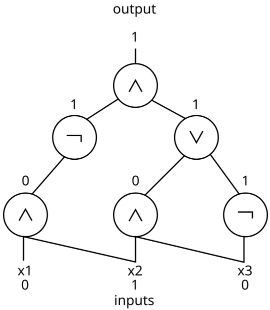
$$
\begin{tikzcd}
 & \vdots & \\
\Sigma_3^{\mathrm{P}} & & \Pi_3^{\mathrm{P}} \\
 & \Delta_3^{\mathrm{P}} \arrow[ul] \arrow[ur] & \\
\Sigma_2^{\mathrm{P}} \arrow[uu] \arrow[ur] & & \Pi_2^{\mathrm{P}} \arrow[uu] \arrow[ul] \\
 & \mathrm{P}^{\mathrm{NP}} = \Delta_2^{\mathrm{P}} \arrow[ul] \arrow[ur] & \\
\mathrm{NP} = \Sigma_1^{\mathrm{P}} \arrow[uu] \arrow[ur] & & \Pi_1^{\mathrm{P}} = \mathrm{coNP} \arrow[uu] \arrow[ul] \\
 & \Delta_0^{\mathrm{P}} = \Sigma_0^{\mathrm{P}} = \mathrm{P} = \Pi_0^{\mathrm{P}} = \Delta_1^{\mathrm{P}} \arrow[ul] \arrow[ur] &
\end{tikzcd}
$$

**Open**: P $=$ NP? NP $=$ coNP? Is the polynomial hierarchy proper?

If any two of the classes are equal, then the hierarchy collapses to them.

# Boolean circuits

Boolean circuit: Non-uniform model of computation.

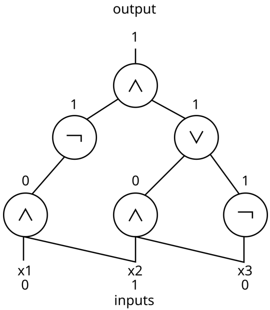



## Boolean circuits

Directed acyclic graph; computes Boolean functions; contains $\wedge, \vee, \neg$ gates. takes a fixed number of bits as input, outputs a single bit.

# Boolean circuits

Boolean circuit: Non-uniform model of computation.




## Boolean circuits

Directed acyclic graph; computes Boolean functions; contains $\wedge, \vee, \neg$ gates. takes a fixed number of bits as input, outputs a single bit.

## P/poly

P/poly is the class of decision problems solvable by a sequence of small, i.e., polynomial size circuits $(C_0, C_1, \ldots)$, where $C_n$ acts on inputs of length $n$.

# Cobham's definition of *FP*

FP is the set of polynomial time computable functions.

Cobham provided a machine independent characterization of polynomial time functions.

**Base functions**: 0, $S$ (successor), $\lfloor \frac{1}{2}x \rfloor$, $2 \cdot x$,

$$\chi_{\leqslant}(x, y) = \begin{cases} 1 & \text{if } x \leqslant y \\ 0 & \text{otherwise} \end{cases}$$

$$C(x, y, z) = \begin{cases} y & \text{if } x > 0 \\ z & \text{otherwise} \end{cases}$$

# Cobham's definition of *FP*

## Definition

Let $q$ be a polynomial. $f$ is defined from $g$ and $h$ by *(bounded) limited recursion on notation* with space bound $q$ iff

$$f(\bar{x},0) = g(\bar{x})$$
$$f(\bar{x},y) = h(\bar{x},y,f(\bar{x},\lfloor\frac{1}{2}y\rfloor))$$

provided $|f(\bar{x},y)| \leqslant q(|\bar{x}|,|y|)$ for all $\bar{x},y$.

## Theorem (Cobham '64)

*FP is equal to the set of functions which can be obtained from the above base functions by using composition and limited recursion on notation.*

# Outline

# Language of bounded arithmetic

Language of bounded arithmetic includes the predicates $=, \leqslant$ and functions

$$0, S, +, \cdot, |x|, \left\lfloor \frac{1}{2}x \right\rfloor, x\#y$$

where:

- variables $x, y, z, \ldots$ range over non-negative integers;

# Language of bounded arithmetic

Language of bounded arithmetic includes the predicates $=, \leqslant$ and functions

$$0, S, +, \cdot, |x|, \left\lfloor \frac{1}{2}x \right\rfloor, x\#y$$

where:

- variables $x, y, z, \ldots$ range over non-negative integers;
- $|x| :=$ length of binary representation of $x$, equivalently, $\lceil \log_2(x+1) \rceil$.

# Language of bounded arithmetic

Language of bounded arithmetic includes the predicates $=, \leqslant$ and functions

$$0, S, +, \cdot, |x|, \left\lfloor \frac{1}{2}x \right\rfloor, x\#y$$

where:

- variables $x, y, z, \ldots$ range over non-negative integers;
- $|x| :=$ length of binary representation of $x$, equivalently, $\lceil \log_2(x+1) \rceil$.
- $|0| = 0$.

# Language of bounded arithmetic

Language of bounded arithmetic includes the predicates $=, \leqslant$ and functions

$$0, S, +, \cdot, |x|, \left\lfloor \tfrac{1}{2}x \right\rfloor, x\#y$$

where:

- variables $x, y, z, \ldots$ range over non-negative integers;
- $|x| :=$ length of binary representation of $x$, equivalently, $\lceil \log_2(x+1) \rceil$.
- $|0| = 0$.
- $x\#y := 2^{|x|\cdot|y|}$; so $|x\#y| = |x|\cdot|y| + 1$ (because $|2^i| = i+1$)

# The syntax

Let $t$ be a term not containing the variable $x$. Define:

- Bounded quantifiers $(\exists x \leqslant t), (\forall x \leqslant t)$,
- Sharply bounded quantifiers $(\exists x \leqslant |t|), (\forall x \leqslant |t|)$.

# The syntax

Let $t$ be a term not containing the variable $x$. Define:

- Bounded quantifiers $(\exists x \leqslant t), (\forall x \leqslant t)$,
- Sharply bounded quantifiers $(\exists x \leqslant |t|), (\forall x \leqslant |t|)$.

In a sharply bounded quantifier the quantifiers are stricter.
In real life the length function is surjective on the non-negative integers (given $|0| = 0$):

$$f : x \to |x| \quad f \text{ is surjective on } \mathbb{N}$$

or

$$g : x \to 2^x \quad g \text{ is total on } \mathbb{N}$$

However, in theories of abounded arithmetic these are **not** provable. Thus, $\forall x \leqslant |t|$ is a very different thing than $\forall x \leqslant t$.

# (Sharply) Bounded formulas

A formula is *bounded* if every quantifier in it is bounded:

$$(\exists x \leqslant t)\psi(x) := \exists x(x \leqslant t \wedge \psi(x)),$$

$$(\forall x \leqslant t)\psi(x) := \forall x(x \leqslant t \rightarrow \psi(x)),$$

where $x$ is not free in $t$.

A formula is *sharply bounded* when all its quantifiers are sharply bounded.

## Example

Using $\#, \lfloor \frac{1}{2}x \rfloor, \cdot$, we can write the term $2^{p(|x|)}$, for any polynomial $p$ with non-negative coefficients. Basically by using:

1. $1 \# x = 2^{|x|}$
2. $|\lfloor \frac{1}{2}(x \# y) \rfloor| = |x| \cdot |y|$

## Example

Using $\#, \lfloor \frac{1}{2}x \rfloor, \cdot$, we can write the term $2^{p(|x|)}$, for any polynomial $p$ with non-negative coefficients. Basically by using:

1. $1\#x = 2^{|x|}$
2. $|\lfloor \frac{1}{2}(x\#y) \rfloor| = |x| \cdot |y|$

For instance,

$$2^{|x|^2+y} = (x\#x) \cdot (1\#y)$$

## Example

Using $\#, \lfloor \frac{1}{2}x \rfloor, \cdot$, we can write the term $2^{p(|x|)}$, for any polynomial $p$ with non-negative coefficients. Basically by using:

1. $1\#x = 2^{|x|}$
2. $|\lfloor \frac{1}{2}(x\#y) \rfloor| = |x| \cdot |y|$

For instance,

$$2^{|x|^2+y} = (x\#x) \cdot (1\#y)$$

Another example, $2^{|x|^2 \cdot |y|}$ can be written as

$$\begin{aligned} \lfloor \frac{x\#x}{2} \rfloor \# y &= 2^{|\lfloor \frac{x\#x}{2} \rfloor| \cdot |y|} \\ &= 2^{(|x\#x|-1) \cdot |y|} && (*) \\ &= 2^{|x|^2 \cdot |y|} && (\dagger) \end{aligned}$$

$*$: $|\lfloor \frac{1}{2}x \rfloor| = |x| - 1$
$\dagger$: $|x\#x| = 2^{|x| \cdot |x|} = |x|^2 + 1$

# Benefits of $\#$

Gives terms of polynomial growth rate.

**Definition**

A function $f : \mathbb{N}^k \to \mathbb{N}$ has *polynomial growth rate* iff there is a suitable polynomial $p$ such that for all $\bar{x}$, we have $|f(\bar{x})| \leqslant p(|\bar{x}|)$.

All terms $t(x)$ have polynomial growth rate. Because $\#$ has polynomial growth rate ($|x\#y| = |x| \cdot |y| + 1$) and all basic functions $(+, S, \cdot, \lfloor \frac{1}{2}x \rfloor, |x|)$ are polynomially bounded by the lengths of the arguments to them.

$$\forall t \exists p \; |t(\bar{x})| \leqslant p(|\bar{x}|)$$

On the other hand, by the previous example, $\#$ is powerful enough to allow us form $2^{p(|\bar{x}|)}$, for any polynomial $p$:

$$\forall p \exists t \; |t(\bar{x})| = p(|\bar{x}|)$$

# A hierarchy of formulas

- We classify the bounded formulas in a hierarchy by counting alternations of bounded quantifiers, ignoring sharply bounded ones.

# A hierarchy of formulas

- We classify the bounded formulas in a hierarchy by counting alternations of bounded quantifiers, ignoring sharply bounded ones.
- This is analogous to the arithmetic hierarchy: counting alternations of unbounded quantifiers, ignoring bounded quantifiers.

# A hierarchy of formulas

- We classify the bounded formulas in a hierarchy by counting alternations of bounded quantifiers, ignoring sharply bounded ones.
- This is analogous to the arithmetic hierarchy: counting alternations of unbounded quantifiers, ignoring bounded quantifiers.
- In BA, roles of bounded and sharply bounded quantifiers are analogous to unbounded and bounded quantifiers, respectively, in PA.

# A hierarchy of formulas

- We classify the bounded formulas in a hierarchy by counting alternations of bounded quantifiers, ignoring sharply bounded ones.
- This is analogous to the arithmetic hierarchy: counting alternations of unbounded quantifiers, ignoring bounded quantifiers.
- In BA, roles of bounded and sharply bounded quantifiers are analogous to unbounded and bounded quantifiers, respectively, in PA.

**Definition (Quantifier alternation classes)**

The hierarchy of $\Sigma_i^b$ and $\Pi_i^b$ formulas is defined inductively:

- $\Delta_0^b = \Sigma_0^b = \Pi_0^b$ is the set of sharply bounded formulas.

# A hierarchy of formulas

- We classify the bounded formulas in a hierarchy by counting alternations of bounded quantifiers, ignoring sharply bounded ones.
- This is analogous to the arithmetic hierarchy: counting alternations of unbounded quantifiers, ignoring bounded quantifiers.
- In BA, roles of bounded and sharply bounded quantifiers are analogous to unbounded and bounded quantifiers, respectively, in PA.

## Definition (Quantifier alternation classes)

The hierarchy of $\Sigma_i^b$ and $\Pi_i^b$ formulas is defined inductively:

- $\Delta_0^b = \Sigma_0^b = \Pi_0^b$ is the set of sharply bounded formulas.
- $\Sigma_{i+1}^b$: closure of $\Pi_i^b$ under $\wedge, \vee$, sharply bounded quantifiers, and **bounded existential** quantifiers.

# A hierarchy of formulas

- We classify the bounded formulas in a hierarchy by counting alternations of bounded quantifiers, ignoring sharply bounded ones.
- This is analogous to the arithmetic hierarchy: counting alternations of unbounded quantifiers, ignoring bounded quantifiers.
- In BA, roles of bounded and sharply bounded quantifiers are analogous to unbounded and bounded quantifiers, respectively, in PA.

## Definition (Quantifier alternation classes)

The hierarchy of $\Sigma_i^b$ and $\Pi_i^b$ formulas is defined inductively:

- $\Delta_0^b = \Sigma_0^b = \Pi_0^b$ is the set of sharply bounded formulas.
- $\Sigma_{i+1}^b$: closure of $\Pi_i^b$ under $\wedge, \vee$, sharply bounded quantifiers, and **bounded existential** quantifiers.
- $\Pi_{i+1}^b$: closure of $\Sigma_i^b$ under $\wedge, \vee$, sharply bounded quantifiers, and **bounded universal** quantifiers.

# Some generic examples

**Example**

What is the complexity of the following formulas, for sharply bounded $\phi$?

- $(\exists x \leqslant t)(\exists y \leqslant t')(\forall z \leqslant s)\phi$.

# Some generic examples

## Example

What is the complexity of the following formulas, for sharply bounded $\phi$?

- $(\exists x \leqslant t)(\exists y \leqslant t')(\forall z \leqslant s)\phi$. $\Sigma_2^b$

# Some generic examples

**Example**

What is the complexity of the following formulas, for sharply bounded $\phi$?

- $(\exists x \leqslant t)(\exists y \leqslant t')(\forall z \leqslant s)\phi$. $\Sigma_2^b$
- $(\exists x \leqslant t)(\forall u \leqslant |r|)(\exists y \leqslant t')(\forall z \leqslant s)\phi$.

# Some generic examples

**Example**

What is the complexity of the following formulas, for sharply bounded $\phi$?

- $(\exists x \leqslant t)(\exists y \leqslant t')(\forall z \leqslant s)\phi$. $\Sigma_2^b$
- $(\exists x \leqslant t)(\forall u \leqslant |r|)(\exists y \leqslant t')(\forall z \leqslant s)\phi$. $\Sigma_2^b$

## Example (Real life examples)

- The primality of a number $x$ can be defined by a $\Pi_1^b$-formula:

$$\mathrm{Prime}(x) := x > 1 \wedge (\forall y \leqslant x)(y \mid x \rightarrow (y = 1 \vee y = x)).$$

## Example (Real life examples)

- The primality of a number $x$ can be defined by a $\Pi_1^b$-formula:

$$\mathrm{Prime}(x) := x > 1 \wedge (\forall y \leqslant x)(y \mid x \to (y = 1 \vee y = x)).$$

- The predicate "$a$ is a power of 2":

$$Pow(a) := 2a = (a\#1)$$

## Example (Real life examples)

- The primality of a number $x$ can be defined by a $\Pi_1^b$-formula:

$$\mathrm{Prime}(x) := x > 1 \wedge (\forall y \leqslant x)(y \mid x \to (y = 1 \vee y = x)).$$

- The predicate "$a$ is a power of 2":

$$Pow(a) := 2a = (a\#1)$$

- The function $i$th bit of $a$, $bit(a, i)$:

$$\exists u, v, w \leqslant a\ (u + v + 2vw = a \wedge Pow(v) \wedge |v| = i + 1)$$

or

$$\forall u, v, w \leqslant a\ (u + v + 2vw = a \wedge Pow(v) \wedge |u| \leqslant i \to |v| = i + 1)$$

# Connections with the polynomial time hierarchy

One of the primary justifications for defining $\Sigma_i^b$ and $\Pi_i^b$-formulas:

**Theorem**

1. *Any $\Delta_0^b = \Sigma_0^b = \Pi_0^b$ formula expresses a polynomial time property.*
2. *For $i \geqslant 1$, $\Sigma_i^b$- and $\Pi_i^b$-formulas define exactly the predicates in $\Sigma_i^p$ and $\Pi_i^p$ at the $i$-th level of the polynomial hierarchy, respectively.*

Special cases:

$\Sigma_1^b$-formulas define exactly NP properties.

$\Pi_1^b$-formulas define exactly coNP properties.

# Proof

## Proof of 1

1. Any sharply bounded formula (e.g. $(\exists x \leqslant |a|)(x \cdot x = a)$) can be evaluated in polynomial time. There are polynomially many (in the input size) possible values for $x$ to check. As the inner formula is a polynomial time property, the entire process runs in polynomial time. (But $\exists y \leqslant a$: exp. many values to check.)

## Proof of 2

2. $\Sigma_1^b$ formulas can be evaluated in non-deterministic polynomial time: one can non-deterministically guess values for the existentially quantified variables, while the sharply bounded quantifiers can be evaluated deterministically by exhaustively checking all possible values within the polynomial bounds.

## Proof of 2

2. $\Sigma_1^b$ formulas can be evaluated in non-deterministic polynomial time: one can non-deterministically guess values for the existentially quantified variables, while the sharply bounded quantifiers can be evaluated deterministically by exhaustively checking all possible values within the polynomial bounds.

   **The other direction:** An NP property is expressed by a $\Sigma_1^b$ formula by encoding the entire computation of an NP Turing machine: the formula $\exists w \leqslant |a|^k \, Com(w, a)$ states that there exists a polynomially-bounded computation history $w$, and the $\Delta_0^b$ formula $Com$ checks that $w$ is a valid and accepting computation path for the input $a$, verifying each step using local, sharply bounded checks. □

Defining $\Sigma_i^b$ and $\Pi_i^b$ makes sense. We are expressing P, NP and others within a language, syntactically, so that we can talk about them.

# Outline

**BASIC**:
Finite set of open axioms defining simple properties of function and relation symbols.

(1) $y \leq x \supset y \leq Sx$
(2) $x \neq Sx$
(3) $0 \leq x$
(4) $x \leq y \wedge x \neq y \leftrightarrow Sx \leq y$
(5) $x \neq 0 \supset 2 \cdot x \neq 0$
(6) $y \leq x \vee x \leq y$
(7) $x \leq y \wedge y \leq x \supset x = y$
(8) $x \leq y \wedge y \leq z \supset x \leq z$
(9) $|0| = 0$
(10) $x \neq 0 \supset |2 \cdot x| = S(|x|) \wedge |S(2 \cdot x)| = S(|x|)$
(11) $|S0| = S0$
(12) $x \leq y \supset |x| \leq |y|$
(13) $|x \# y| = S(|x| \cdot |y|)$
(14) $0 \# y = S0$
(15) $x \neq 0 \supset 1 \#(2 \cdot x) = 2(1 \# x) \wedge 1 \#(S(2 \cdot x)) = 2(1 \# x)$
(16) $x \# y = y \# x$
(17) $|x| = |y| \supset x \# z = y \# z$
(18) $|x| = |u| + |v| \supset x \# y = (u \# y) \cdot (v \# y)$
(19) $x \leq x + y$
(20) $x \leq y \wedge x \neq y \supset S(2 \cdot x) \leq 2 \cdot y \wedge S(2 \cdot x) \neq 2 \cdot y$
(21) $x + y = y + x$
(22) $x + 0 = x$
(23) $x + Sy = S(x + y)$
(24) $(x + y) + z = x + (y + z)$
(25) $x + y \leq x + z \leftrightarrow y \leq z$
(26) $x \cdot 0 = 0$
(27) $x \cdot (Sy) = (x \cdot y) + x$
(28) $x \cdot y = y \cdot x$
(29) $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
(30) $x \geq S0 \supset (x \cdot y \leq x \cdot z \leftrightarrow y \leq z)$
(31) $x \neq 0 \supset |x| = S(|\lfloor \frac{1}{2} x \rfloor|)$
(32) $x = \lfloor \frac{1}{2} y \rfloor \leftrightarrow (2 \cdot x = y \vee S(2 \cdot x) = y)$

# Axioms for bounded arithmetic

**Induction Axioms**:
Let $\Phi$ be a class of formulas and $A$ range over $\Phi$-formulas:

- $\Phi$-IND: $A(0) \wedge (\forall x)(A(x) \to A(x+1)) \to (\forall x)A(x)$
- $\Phi$-PIND: $A(0) \wedge (\forall x)(A(\lfloor \frac{1}{2}x \rfloor) \to A(x)) \to (\forall x)A(x)$
- $\Phi$-LIND: $A(0) \wedge (\forall x)(A(x) \to A(x+1)) \to (\forall x)A(|x|)$

Recall: $|0| := 0$.

$\Phi$-PIND and $\Phi$-LIND are "polynomially feasible" versions of induction.

# Polynomial induction (PIND)

$$\Phi\text{-PIND} : A(0) \wedge (\forall x)(A(\lfloor \tfrac{1}{2}x \rfloor) \rightarrow A(x)) \rightarrow (\forall x)A(x)$$

**Question**

Is P-induction valid? Is it even a true statement?

# Polynomial induction (PIND)

$$\Phi\text{-PIND} : A(0) \wedge (\forall x)(A(\lfloor \tfrac{1}{2}x \rfloor) \to A(x)) \to (\forall x)A(x)$$

**Question**

Is P-induction valid? Is it even a true statement?

- Looks a bit strange, as we're going from $\lfloor \frac{1}{2}x \rfloor$ to $x$. PIND is valid because if the hypotheses are true, there can't be any least $x$ where $A(x)$ is false.

string mindset: You want to prove $A(x_k \dots x_1 x_0)$.

## Example

$A(0)$ holds. Goal: show $A(100)$.

- $\Phi$-IND: $A(0) \mapsto A(1) \mapsto \ldots \mapsto A(100)$ (100 steps)

## Example

$A(0)$ holds. Goal: show $A(100)$.

- $\Phi$-IND: $A(0) \mapsto A(1) \mapsto \ldots \mapsto A(100)$ (100 steps)
- $\Phi$-PIND:
  $A(0) \mapsto A(1) \mapsto A(3) \mapsto A(6) \mapsto A(12) \mapsto A(25) \mapsto A(50) \mapsto A(100)$
  ($7 = |100|$ steps)

# Length induction (LIND)

$$\Phi\text{-LIND} : A(0) \wedge (\forall x)(A(x) \rightarrow A(x+1)) \rightarrow (\forall x)A(|x|)$$

**Question**

Is $\Phi$-LIND stronger/weaker than the usual $\Phi$-IND induction or the same?

# Length induction (LIND)

$$\Phi\text{-LIND} : A(0) \wedge (\forall x)(A(x) \rightarrow A(x+1)) \rightarrow (\forall x)A(|x|)$$

## Question

Is $\Phi$-LIND stronger/weaker than the usual $\Phi$-IND induction or the same?

## Answer

- $\Phi$-IND $\Longrightarrow$ $\Phi$-LIND. Because if $\forall x A(x)$ then certainly $\forall x A(|x|)$.

# Length induction (LIND)

$$\Phi\text{-LIND} : A(0) \wedge (\forall x)(A(x) \to A(x+1)) \to (\forall x)A(|x|)$$

**Question**

Is $\Phi$-LIND stronger/weaker than the usual $\Phi$-IND induction or the same?

**Answer**

- $\Phi$-IND $\Longrightarrow$ $\Phi$-LIND. Because if $\forall x A(x)$ then certainly $\forall x A(|x|)$.
- Converse: Does $\forall x A(|x|)$ imply $\forall x A(x)$?
  Why do we have a separate axiom if the answer is yes? :)

# Length induction (LIND)

$$\Phi\text{-LIND} : A(0) \wedge (\forall x)(A(x) \rightarrow A(x+1)) \rightarrow (\forall x)A(|x|)$$

**Question**

Is $\Phi$-LIND stronger/weaker than the usual $\Phi$-IND induction or the same?

**Answer**

- $\Phi$-IND $\Longrightarrow$ $\Phi$-LIND. Because if $\forall x A(x)$ then certainly $\forall x A(|x|)$.
- Converse: Does $\forall x A(|x|)$ imply $\forall x A(x)$?
  Why do we have a separate axiom if the answer is yes? :)

In BA, $x \mapsto 2^x$ need not be total, so $x \mapsto |x|$ may not be surjective on non-negative integers. Thus, $\forall x A(|x|)$ is weaker than $\forall x A(x)$.

# Length induction (LIND)

$$\Phi\text{-LIND} : A(0) \wedge (\forall x)(A(x) \to A(x+1)) \to (\forall x)A(|x|)$$

## Question

Is $\Phi$-LIND stronger/weaker than the usual $\Phi$-IND induction or the same?

## Answer

- $\Phi$-IND $\Longrightarrow$ $\Phi$-LIND. Because if $\forall x A(x)$ then certainly $\forall x A(|x|)$.
- Converse: Does $\forall x A(|x|)$ imply $\forall x A(x)$?
  Why do we have a separate axiom if the answer is yes? :)

In BA, $x \mapsto 2^x$ need not be total, so $x \mapsto |x|$ may not be surjective on non-negative integers. Thus, $\forall x A(|x|)$ is weaker than $\forall x A(x)$.
Not surprising that LIND and PIND are provably equivalent; they say the same thing but in two different ways.

# Outline

1. Motivation
2. A very short course on complexity
3. Syntax of bounded arithmetic
4. (Induction) axioms for bounded arithmetic
5. Theories of bounded arithmetic

# Hierarchy of theories

## Definition (Fragments of bounded arithmetic)

- $S_2^i$: Basic $+$ $\Sigma_i^b$-PIND
- $T_2^i$: Basic $+$ $\Sigma_i^b$-IND
- $S_2 = \bigcup_i S_2^i$ and $T_2 = \bigcup_i T_2^i$

# Hierarchy of theories

**Definition (Fragments of bounded arithmetic)**

- $S_2^i$: Basic + $\Sigma_i^b$-PIND
- $T_2^i$: Basic + $\Sigma_i^b$-IND
- $S_2 = \bigcup_i S_2^i$ and $T_2 = \bigcup_i T_2^i$

Most important for us: $S_2^1$. Since the $\Sigma_1^b$-formulas express exactly the NP properties, $\Sigma_1^b$-PIND is a PIND on NP predicates. So, we're doing "feasible length induction", i.e., P-induction, but on the NP predicates.

# Hierarchy of theories

**Definition (Fragments of bounded arithmetic)**

- $S_2^i$: Basic $+$ $\Sigma_i^b$-PIND
- $T_2^i$: Basic $+$ $\Sigma_i^b$-IND
- $S_2 = \bigcup_i S_2^i$ and $T_2 = \bigcup_i T_2^i$

Most important for us: $S_2^1$. Since the $\Sigma_1^b$-formulas express exactly the NP properties, $\Sigma_1^b$-PIND is a PIND on NP predicates. So, we're doing "feasible length induction", i.e., P-induction, but on the NP predicates.
Looking ahead:

**Theorem (Buss '85, '90)**

1. $S_2^1 \subseteq T_2^1 \subseteq S_2^2 \subseteq T_2^2 \subseteq S_2^3 \subseteq \dots$
2. *Thus,* $S_2 = T_2$.

# Useful references


- "Tutorial on Bounded Arithmetic", tutorial by Sam Buss, Workshop on Proof Complexity, St. Petersburg State University, May 15, 2016. video 1, video 2, video 3, video 4, slides
- "Bounded arithmetic and propositional proof complexity", San Buss in Logic of computation, 1997, paper
- "Bounded Arithmetic", Sam Buss, Ph.D. thesis, 1985, thesis
- "Bounded Arithmetic, Propositional Logic, and Complexity Theory", Jan Krajíček, Cambridge University Press, 1995, book
- "Logical foundations of proof complexity", Stephen Cook and Phuong Nguyen, Cambridge University Press, 2010, book

# Useful references


- "Tutorial on Bounded Arithmetic", tutorial by Sam Buss, Workshop on Proof Complexity, St. Petersburg State University, May 15, 2016. video 1, video 2, video 3, video 4, slides
- "Bounded arithmetic and propositional proof complexity", San Buss in Logic of computation, 1997, paper
- "Bounded Arithmetic", Sam Buss, Ph.D. thesis, 1985, thesis
- "Bounded Arithmetic, Propositional Logic, and Complexity Theory", Jan Krajíček, Cambridge University Press, 1995, book
- "Logical foundations of proof complexity", Stephen Cook and Phuong Nguyen, Cambridge University Press, 2010, book


**Thank you!**