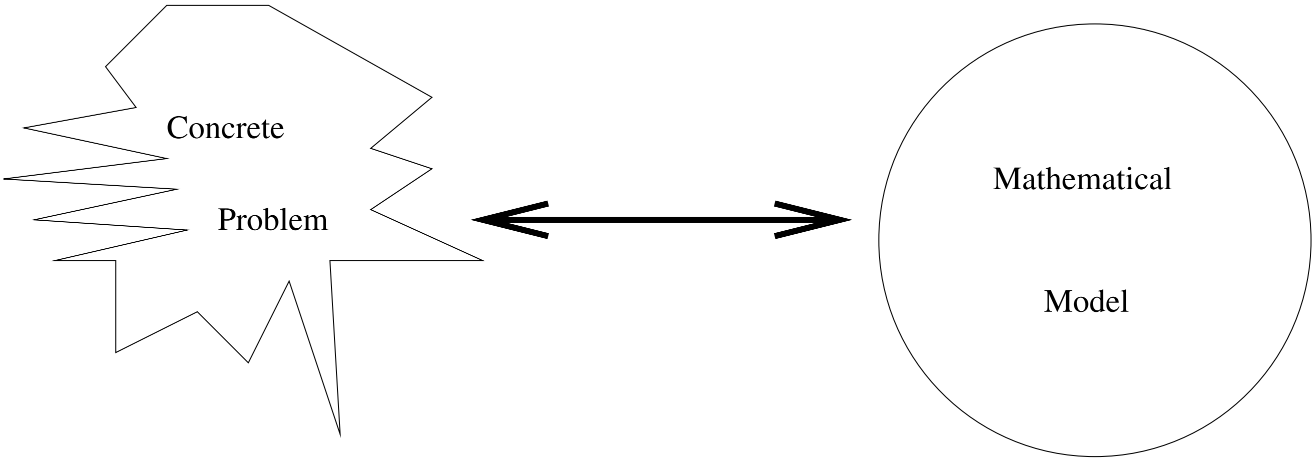
Summary & Conclusions

We've studied the main models and concepts of the theory of computation:

- **Computability**: what can be computed in principle
- **Logic**: how can we express our requirements
- **Complexity**: what can be computed in practice



- How is the input organized?
- What computational operations are allowed?
- Do we have internal memory, and how much?

An answer to these questions gives us a formal model of computation.

Some Formal Models of Computation:

- **Boolean:**
- **Formal Language Theory:**
- **First-Order Logic:**
- **Recursive Function Theory:**
- **Abstract RAM:** (as in an algorithms course)

Boolean Computation:

Data Type: Booleans, true or false, 0 or 1

Operations: AND, OR (inclusive), NOT, can define more

Variables: Inputs $x_1, \ldots, x_n$, others $y_1, \ldots, y_m$

Expressions: $(x_1 \vee \neg x_2) \wedge x_3$, etc.

Straight-Line Program:

```
y_1 = NOT x_1;
y_2 = y_1 OR x_2;
y_3 = x_3 AND x_4;
y_4 = y_2 AND y_3;
y_5 = y_2 OR x_1;
return NOT y_5;
```

A straight-line program is equivalent to a *boolean circuit*, about which more later. Each line of the program (gate of the circuit) has a truth value that is a *function of the input variables*. Given a function of the inputs (a *property*), we will consider among other things *how long a program* is needed to compute it in this way.

Boolean Logic:

Chapters 1-8 of [BE] dealt with propositional or boolean logic. The main topics were:

- *Syntax* and *semantics* for boolean expressions
- Rules for proving that an expression is a *tautology* or that one expression *follows logically* from another
- A formal proof system encapsulating these rules and implemented as interactive software, included with the book

The basics of boolean logic should have been review, but formal proof may not have been. We studied the system from the book and software starting on HW#2. We then considered two important properties of the system:

- Soundness: everything provable is true
- Completeness: everything true is provable

Here “everything” refers only to statements that can be made within the system.

First-Order Logic:

We can think of a sequence of boolean variables $x_0, \ldots, x_{n-1}$ as a single object X, a *function* from the set $\{0, \ldots, n-1\}$ to $\{0, 1\}$. Thus for any number i in the range, $X(i)$ is the boolean variable we used to call x_i.

X is also called a *unary relation*, a *property* that each number in the range either has or doesn't have. We can also have *binary*, *ternary*, or k-ary relations.

For example, in a directed graph where the vertices are named $\{0, \ldots, n-1\}$ we have a binary relation E, where the boolean $E(i, j)$ is 1 iff there is an edge from i to j.

More First-Order Logic:

A *first-order structure*, roughly speaking, is a universe of base objects and a set of relations over them. If one of these relations is the *input*, we can use *first-order logic* to define properties of the input.

For example the first-order sentence

$$\forall x : \exists y : E(x, y)$$

is true for the graph with edge relation E iff every vertex has out-degree at least one.

In Chapters 9-13 of [BE] we saw syntax and semantics for first-order logic, a set of proof rules for it, and software that lets you play with structures consisting of sets of blocks of different types on a grid. We proved (using later chapters of [BE]) that this proof system is also sound and complete.

Recursive Function Theory:

Bits can be used to represent *numbers* (by which I generally mean *non-negative integers* in binary). General computations may be coded as functions from numbers (or tuples of numbers) to numbers.

Kleene's *recursive function theory* defines a set of *general recursive functions* by induction. There are *base functions* and rules for creating new functions from old ones.

It turns out that "general recursive" corresponds to "algorithmically computable". (This is a theorem, not a definition.)

Primitive Recursion and Bloop

An important subset of the partial recursive functions is the *primitive recursive functions*. We will define these in terms of a simple programming language called Bloop. (My Bloop is based on the language of the same name in Hofstadter's *Gödel, Escher, Bach* but has a different syntax.)

Variables represent numbers (non-negative integers). They are declared and initialized to 0 when they first appear.

Statements of Bloop consist of:

- assignment statements `x = ...`
- the increment operator `++`
- function declarations and calls (call-by-value, no recursion)
- bounded loops `for (x) B`, where `x` is a variable and `B` is a block of code in which `x` is not modified. This code executes `B` exactly `x` times.

A function is *primitive recursive* iff it can be implemented by a Bloop program.

Relations Among The Models:

Let's look at a single problem. Given n input bits, we want to know whether *exactly two* of them are ones. This question can be posed in each of our models:

- **Boolean:** There are various ways to build an SLP or circuit, which we explored on HW#1.
- **Finite-State Machine:** Sweep the input string left-to-right, remembering whether we've seen zero, one, two, or more than two ones.
- **First-Order Logic:**

$$\exists x : \exists y : \neg(x = y) \wedge \forall z : I(z) \leftrightarrow (z = x \vee z = y)$$

- **Numerical Input:** Is the input the sum of two distinct powers of two? On HW#1 you wrote a Bloop program to decide this.
- **Abstract RAM:** The problem probably defaults to one of the others once we decide on our data representation.

Kleene's Theorem: Let $A \subseteq \Sigma^\star$ be any language. Then the following are equivalent:

1. $A = \mathcal{L}(D)$, for some DFA D.
2. $A = \mathcal{L}(N)$, for some NFA N without ϵ transitions
3. $A = \mathcal{L}(N)$, for some NFA N.
4. $A = \mathcal{L}(e)$, for some regular expression e.

Myhill-Nerode Theorem: The language A is regular iff $\sim_A$ has a finite number of equivalence classes. Furthermore, this number of equivalence classes is equal to the number of states in the minimum-state DFA that accepts A.

Pumping Lemma for Regular Sets: Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Let $n = |Q|$. Let $w \in \mathcal{L}(D)$ s.t. $|w| \geq n$. Then $\exists x, y, z \in \Sigma^\star$ s.t. the following all hold:

- $xyz = w$
- $|xy| \leq n$
- $|y| > 0$, and
- $(\forall k \geq 0) xy^k z \in \mathcal{L}(D)$

Closure Theorem for Context Free Languages: Let $A, B \subseteq \Sigma^\star$ be context-free languages, let $R \subseteq \Sigma^\star$ be a regular language, and let $h : \Sigma^\star \to \Gamma^\star$ and $g : \Gamma^\star \to \Sigma^\star$ be homomorphisms. Then the following languages are context-free:

1. $A \cup B$
2. AB
3. $A \cap R$
4. $h(A)$
5. $g^{-1}(A)$

CFL Pumping Lemma: Let A be a CFL. Then there is a constant n, depending only on A such that if $z \in A$ and $|z| \geq n$, then there exist strings u, v, w, x, y such that $z = uvwxy$, and,

- $|vx| \geq 1$,
- $|vwx| \leq n$, and
- for all $k \in \mathbf{N}$, $uv^kwx^ky \in A$

A (partial) function is *recursive* iff it is computed by some TM M.

Let $S \subseteq \{0,1\}^\star$ or $S \subseteq \mathbf{N}$.

S is a *recursive set* iff the function χ_S is a (total) recursive function, where

$$\chi_S(x) \;=\; \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$$

S is a *recursively enumerable set* (S is r.e.) iff the function p_S is a (partial) recursive function, where

$$p_S(x) \;=\; \begin{cases} 1 & \text{if } x \in S \\ \nearrow & \text{otherwise} \end{cases}$$

Theorem: Recursive = r.e. $\cap$ co-r.e.

Define the *primitive recursive functions* to be the smallest class of functions that

- contains the Initial functions: ζ, σ, and π_i^n, $n = 1, 2, \ldots$, $1 \leq i \leq n$, and
- is closed under **Composition**, and
- is closed under **Primitive Recursion**

Define the *general recursive functions* to be the smallest class of functions that

- contains the Initial functions, and
- is closed under **Composition**, and
- is closed under **Primitive Recursion**, and
- is closed under **Unbounded Mimimalization**

Theorem: [Kleene] COMP(n, x, c, y) is a primitive recursive predicate.

Theorem: A (partial) function is recursive iff it is general recursive.

Cantor's Theorem: $\wp(\mathbf{N})$ is not countable!

Proof: Suppose it were. Let $f : \mathbf{N} \overset{1:1}{\underset{onto}{\rightarrow}} \wp(\mathbf{N})$. Define the diagonal set,

$$D \quad = \quad \{n \mid n \notin f(n)\}$$

Thus $D = f(k)$ for some $k \in \mathbf{N}$.

$$k \in D \quad \Leftrightarrow \quad k \notin f(k) \quad \Leftrightarrow \quad k \notin D$$

$\Rightarrow\Leftarrow$ Therefore, $\wp(\mathbf{N})$ is not countable! ♠

n	0	1	2	3	4	5	6	7	8	$\cdots$	$f(n)$
0	$\boxed{0}$	0	0	0	0	0	0	0	0	$\cdots$	$f(0)$
1	1	$\boxed{1}$	1	1	1	1	1	1	1	$\cdots$	$f(1)$
2	1	0	$\boxed{1}$	0	1	0	1	0	1	$\cdots$	$f(2)$
3	0	1	0	$\boxed{1}$	0	1	0	1	0	$\cdots$	$f(3)$
4	1	0	0	0	$\boxed{0}$	0	0	0	0	$\cdots$	$f(4)$
5	0	1	1	0	1	$\boxed{0}$	0	0	1	$\cdots$	$f(5)$
6	1	0	0	1	0	0	$\boxed{1}$	0	0	$\cdots$	$f(6)$
7	1	1	0	0	0	0	0	$\boxed{0}$	0	$\cdots$	$f(7)$
8	0	1	0	0	0	0	0	0	$\boxed{0}$	$\cdots$	$f(8)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
	1	0	0	0	1	1	0	1	1	$\cdots$	D

$$K \quad = \quad \{n \mid M_n(n) = 1\}$$

Theorem: $\overline{K}$ is not r.e.

Know how to prove this!

Hierarchy Theorems: Let $f(n)$ be a well behaved function, and $\mathcal{C}$ one of DSPACE, NSPACE, DTIME, NTIME.

If $g(n)$ is sufficiently smaller than $f(n)$ then $\mathcal{C}[g(n)]$ is strictly contained in $\mathcal{C}[f(n)]$.

"$g(n)$ sufficiently smaller than $f(n)$" means

$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$	$\lim_{n\to\infty} \frac{g(n)\log(g(n))}{f(n)} = 0$
$\mathcal{C} = $ **DSPACE**, **NSPACE**, **NTIME**	$\mathcal{C} = $ **DTIME**

Hence **P** $\neq$ **EXPTIME**, **L** $\neq$ **PSPACE**.

But these are the *only* separations of classes we know! (Except at the p.r. and above level, and for **REG** and **CFL**).

Theorem: The busy beaver function is eventually larger than any total, recursive function.

Theorem: Let $S \subseteq \mathbf{N}$. T.F.A.E.

1. S is the domain of a partial, recursive function.
2. $S = \emptyset$ or S is the range of a total, recursive function.
3. S is the range of a partial, recursive function.
4. $S = W_n$, some $n = 0, 1, 2, \ldots$ where

$$W_n \quad = \quad \{m \mid M_n(m) = 1\}$$

Definitions of **Formula, Structure,** and **Truth**

Fitch Proof Rules

- **Propositional (12):** Intro and Elim rules for $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\neg$, and $\perp$.
- **Equality (2):** Intro and Elim for $=$.
- **Quantifier (4):** Intro and Elim for $\exists$ and $\forall$.

$$\text{FO-THEOREMS} \quad = \quad \{\varphi \mid \vdash \varphi\}$$

$$\text{FO-VALID} \quad = \quad \{\varphi \mid \models \varphi\}$$

Soundness Theorem: If $\vdash \varphi$ then $\models \varphi$.

$$\text{FO-THEOREMS} \quad \subseteq \quad \text{FO-VALID}$$

Completeness Theorem: If $\models \varphi$ then $\vdash \varphi$.

$$\text{FO-THEOREMS} \quad \supseteq \quad \text{FO-VALID}$$

Corollary:

$$\vdash \;=\; \models; \qquad \text{FO-THEOREMS} = \text{FO-VALID}$$

Compactness Theorem: If every finite subset of Γ has a model, then Γ has a model.

Gödel's Incompleteness Theorem:

Theory($\mathbf{N}$) (true arithmetic) is not r.e. and thus not axiomatizable. (If Γ is an r.e. set of axioms that are all true in $\mathbf{N}$, then there exists a formula in Theory($\mathbf{N}$) that is not provable from Γ.)

Theorem: For $t(n) \geq n, s(n) \geq \log n$,

$$\textbf{DTIME}[t(n)] \;\subseteq\; \textbf{NTIME}[t(n)] \;\subseteq\; \textbf{DSPACE}[t(n)]$$

$$\textbf{DSPACE}[s(n)] \quad\subseteq\quad \textbf{DTIME}[2^{O(s(n))}]$$

Savitch's Theorem:

For $s(n) \geq \log n$,

$$\textbf{NSPACE}[s(n)] \;\subseteq\; \textbf{ATIME}(s(n))^2 \;\subseteq\; \textbf{DSPACE}[(s(n))^2]$$

Immerman-Szelepcsényi Theorem:

For $s(n) \geq \log n$,

$$\textbf{NSPACE}[s(n)] \quad=\quad \text{co-}\textbf{NSPACE}[s(n)]$$

Definition: $A \leq B$ means there exists $f \in F(\mathbf{L})$ such that for any x, $x \in A$ iff $f(x) \in B$. This is also called a **logspace many-one reduction**.

Theorem: Let $\mathcal{C}$ be one of the following complexity classes: L, NL, P, NP, co-NP, PSPACE, EXPTIME, Primitive-Recursive, RECURSIVE, r.e., co-r.e.

$$\text{Suppose } A \leq B.$$
$$\text{If } B \in \mathcal{C} \quad \text{Then} \quad A \in \mathcal{C}$$

All these complexity classes are **closed downward under reductions.**

Lower Bounds: If A is hard then B is hard.

Upper Bounds: If B is easy then A is easy.

Complete for NL: REACH, EMPTY-DFA, EMPTY-NFA, 2-SAT

Complete for P: CVP, MCVP, EMPTY-CFL, Horn-SAT, REACH_a

Complete for NP: TSP, SAT, 3-SAT, 3-COLOR, CLIQUE, Subset Sum, Knapsack

Complete for PSPACE: QSAT, GEOGRAPHY, SUCCINCT-REACH, REG-EXP-$\Sigma^\star$

Complete for r.e.: K, HALT, $A_{0,17}$, FO-VALID

Complete for co-r.e.: $\overline{K}$, $\Sigma^\star$CFL, EMPTY, FO-SAT

Theorem:

$$\begin{aligned}
\textbf{r.e.} &= \mathrm{FO}\exists(\mathbf{N}) \\
\text{co-}\textbf{r.e.} &= \mathrm{FO}\forall(\mathbf{N}) \\
\\
\mathrm{PH} &= \mathrm{SO} \\
\mathbf{NP} &= \mathrm{SO}\exists \\
\mathbf{P} &= \mathrm{SO}\exists\text{-Horn} \\
\mathbf{AC}^0 &= \mathbf{CRAM}[1] \;=\; \mathrm{LH} \;=\; \mathrm{FO}
\end{aligned}$$

One can understand the complexity of a problem as the richness of a logical language that is needed to describe the problem.

Theorem: For $s(n) \geq \log n$, and for $t(n) \geq n$,

$$\bigcup_{k=1}^{\infty} \textbf{ATIME}[(t(n))^k] \;=\; \bigcup_{k=1}^{\infty} \textbf{DSPACE}[(t(n))^k]$$

$$\textbf{ASPACE}[s(n)] \;=\; \bigcup_{k=1}^{\infty} \textbf{DTIME}[k^{s(n)}]$$

Corollary: In particular,

$$\textbf{ASPACE}[\log n] \;=\; \textbf{P}$$

$$\textbf{ATIME}[n^{O(1)}] \;=\; \textbf{PSPACE}$$

$$\textbf{ASPACE}[n^{O(1)}] \;=\; \textbf{EXPTIME}$$

Circuit Complexity

$$\text{depth} \quad = \quad \text{parallel time}$$

$$\text{width} \;=\; \text{hardware}$$

$$\text{number of gates} \;=\; \text{computational work} \;=\; \text{sequential time}$$

Theorem: For all i, $\quad \mathbf{CRAM}[(\log n)^i] \quad = \quad \mathbf{AC}^i$

$$\begin{array}{cccccccccccc}
\mathbf{AC}^0 & \subseteq & \mathbf{ThC}^0 & \subseteq & \mathbf{NC}^1 & \subseteq & \mathbf{L} & \subseteq & \mathbf{NL} & \subseteq & \mathbf{sAC}^1 & \subseteq \\
\mathbf{AC}^1 & \subseteq & \mathbf{ThC}^1 & \subseteq & \mathbf{NC}^2 & & & \subseteq & & & \mathbf{sAC}^2 & \subseteq \\
\vdots & \subseteq & \vdots & \subseteq & \vdots & & & \subseteq & & & \vdots & \subseteq \\
\mathbf{AC}^i & \subseteq & \mathbf{ThC}^i & \subseteq & \mathbf{NC}^{i+1} & & & \subseteq & & & \mathbf{sAC}^{i+1} & \subseteq \\
\vdots & \subseteq & \vdots & \subseteq & \vdots & & & \subseteq & & & \vdots & \subseteq \\
\mathbf{NC} & = & \mathbf{NC} & = & \mathbf{NC} & & & = & & & \mathbf{NC} & = \\
\mathbf{NC} & & \subseteq & & \mathbf{P} & & & \subseteq & & & \mathbf{NP} &
\end{array}$$

Alternation/Circuit Theorem:

Log-space ATM's with:

- $O(\log^i n)$ time give $\mathbf{NC}^i$ ($i \geq 1$)
- $O(\log^i n)$ alternations give $\mathbf{AC}^i$ ($i \geq 1$)

Alternating TM's are one good way to design uniform families of circuits. We used this method to prove $\mathbf{CFL} \subseteq \mathbf{sAC}^1$.

First-order logic gives us another way to design uniform families of circuits. We've used this to construct $\mathbf{AC}^0$ circuits by showing a problem to be in FO.

We need uniformity definitions on our circuit classes to relate them to ordinary classes. For example, poly-size circuit families compute languages in **P** only if they are at least **P**-uniform.

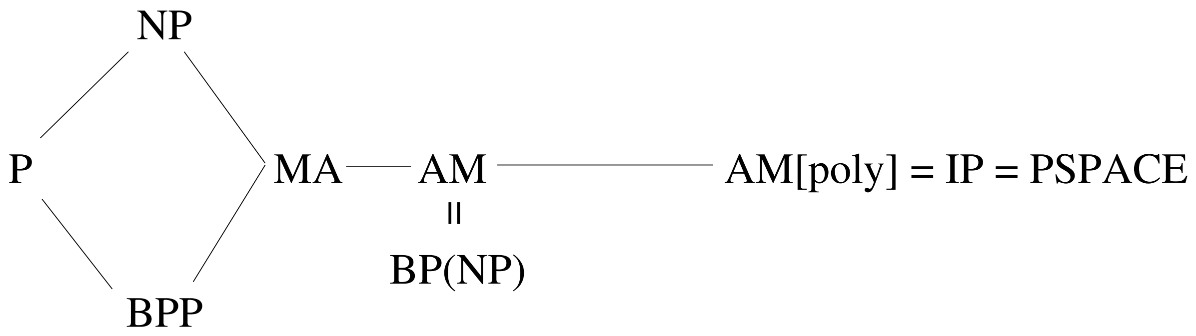
Theorem: PRIME and Factoring are in **NP** $\cap$ co-**NP**. (PRIME is now in **P** as well.)

Theorem: [Solovay-Strassen, Miller]

$$\text{PRIME} \ \in \ \mathbf{BPP}$$

Fact: $\text{REACH}_u \ \in \ \text{BPL}$

Interactive Proofs



Fact: PCP[$\log n, 1$] = NP

A is an *optimization problem* iff

For each instance x, $F(x)$ is the set of *feasible solutions*

Each $s \in F(x)$ has a cost $c(s) \in \mathbf{Z}^+$

For minimization problems,

$$\text{OPT}(x) \quad = \quad \min_{s \in F(x)} c(s)$$

For maximization problems,

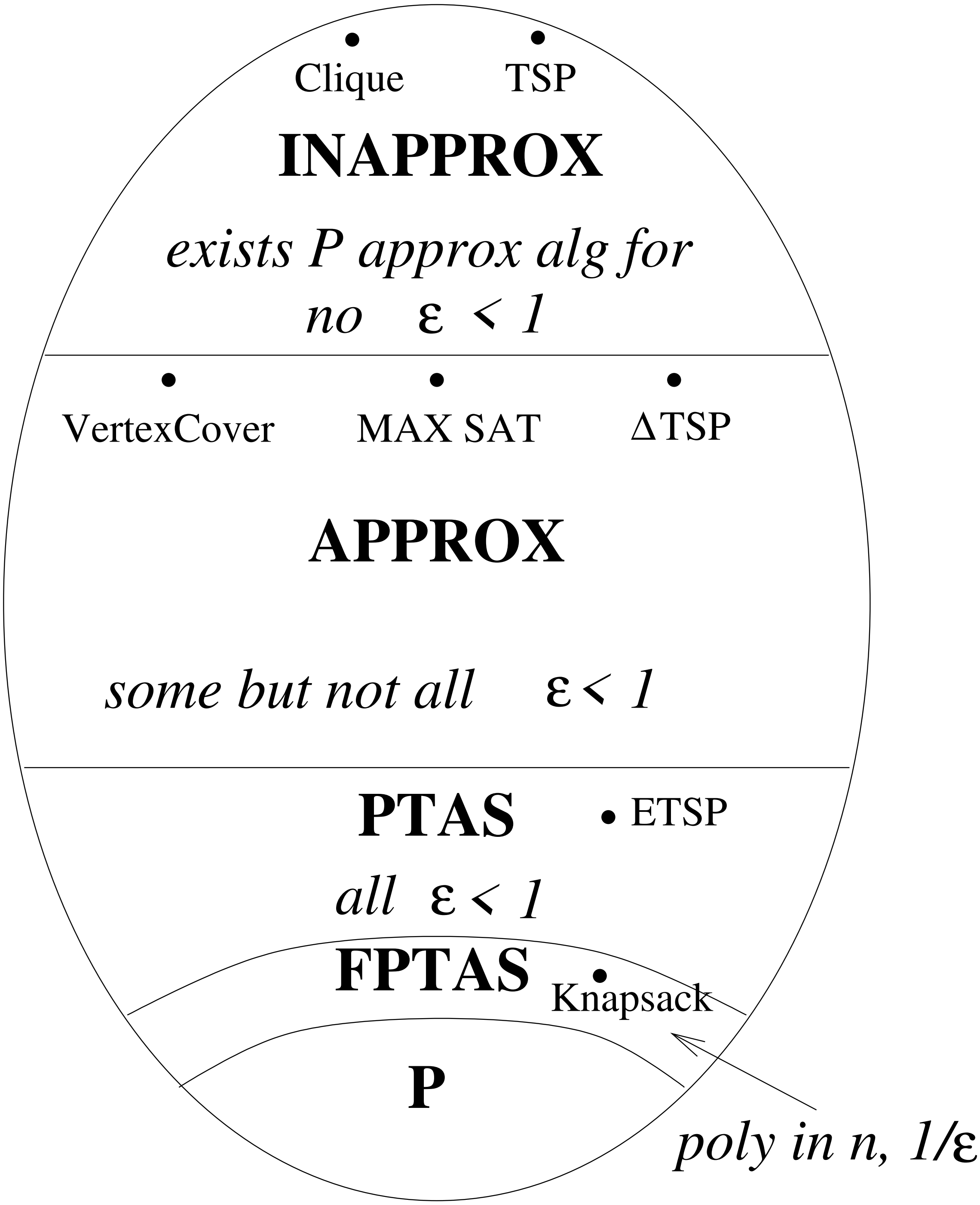
$$\text{OPT}(x) \quad = \quad \max_{s \in F(x)} c(s)$$

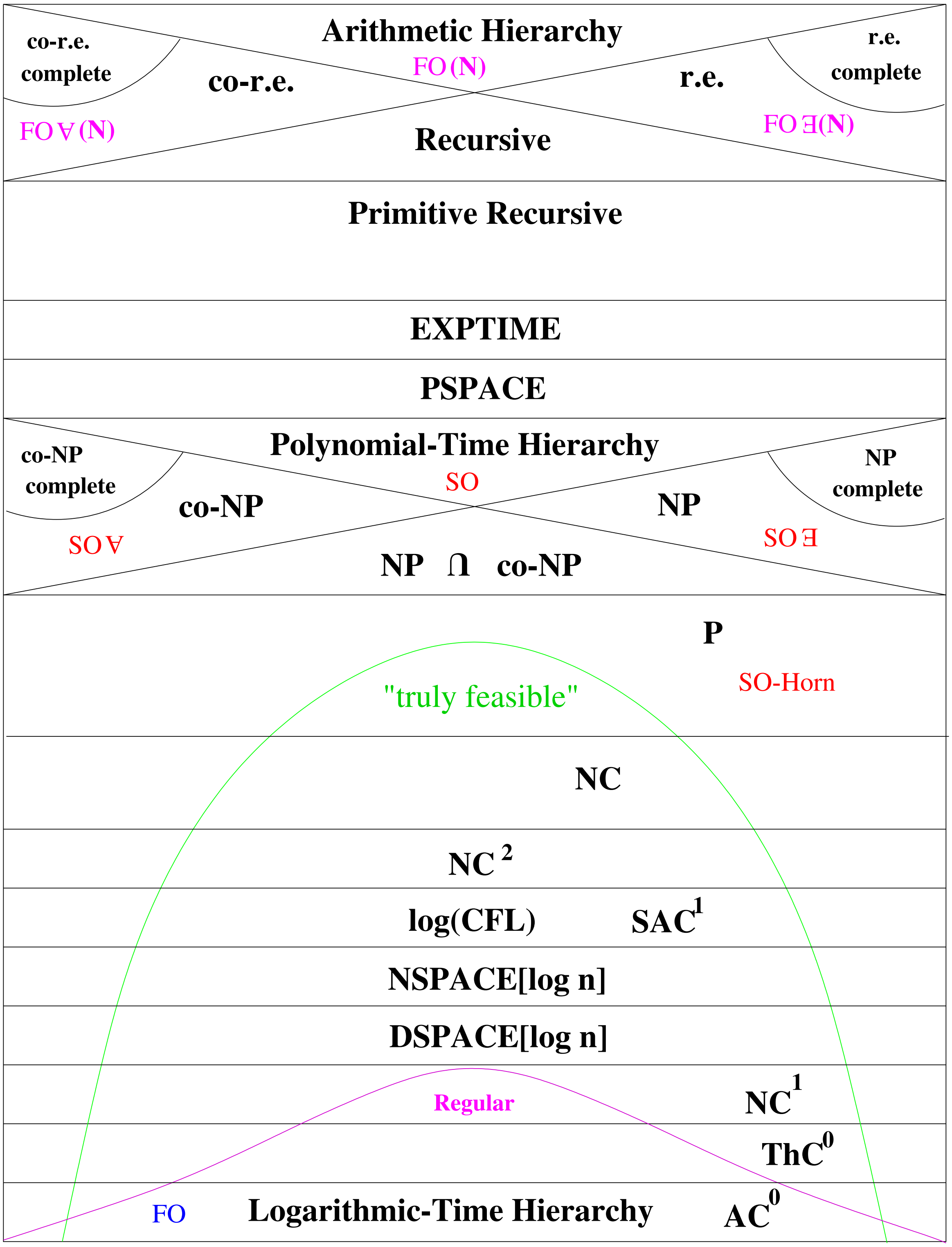
Let M be an algorithm s.t. on any instance x,

$$M(x) \in F(x)$$

M is an *ϵ-approximation algorithm* iff for all x,

$$\frac{|c(M(x)) - \text{OPT}(x)|}{\max(\text{OPT}(x), c(M(x)))} \le \epsilon$$ ♠


Clique
TSP
INAPPROX
exists P approx alg for
no ε < 1
VertexCover
MAX SAT
ΔTSP
APPROX
some but not all ε < 1
PTAS
ETSP
all ε < 1
FPTAS
Knapsack
P
poly in n, 1/ε


Arithmetic Hierarchy
co-r.e. complete
co-r.e.
FO (N)
r.e.
r.e. complete
FO ∀ (N)
Recursive
FO ∃(N)
Primitive Recursive
EXPTIME
PSPACE
Polynomial-Time Hierarchy
co-NP complete
co-NP
SO
NP
NP complete
SO ∀
NP ∩ co-NP
SO ∃
P
"truly feasible"
SO-Horn
NC
NC 2
log(CFL)
SAC1
NSPACE[log n]
DSPACE[log n]
Regular
NC1
ThC0
FO
Logarithmic-Time Hierarchy
AC0

Why are the following so hard to prove?

- $\mathbf{P} \neq \mathbf{NP}$
- $\mathbf{P} \neq \mathbf{PSPACE}$
- $\mathbf{ThC}^0 \neq \mathbf{NP}$
- $\mathbf{BPP} = \mathbf{P}$

We do know a lot about computation. Reductions and complete problems are a key tool. So is the equivalence of apparently different models and methods. Yet much remains unknown.