# Basic Complexity

*Eric Allender*[*]
*Catherine McCartin*[†]

**Abstract.** This paper summarizes a series of three lectures the first author was invited to present at the NZMRI summer 2000 workshop, held in Kaikoura, New Zealand. Lecture 1 presents the goals of computational complexity theory. We discuss (a) what complexity provably can never deliver, (b) what it hopes to deliver but thus far has not, and finally (c) where it has been extremely successful in providing useful theorems. In so doing, we introduce nondeterministic Turing machines. Lecture 2 presents alternation, a surprisingly-useful generalization of nondeterminism. Using alternation, we define more complexity classes, and inject clarity into a confusing situation. In Lecture 3 we present a few of the most beautiful results in computational complexity theory. In particular, we discuss (a) the algebraic approach to circuit complexity, (b) circuit lower bounds, and (c) derandomization.

## 1. Lecture 1

> **Warning:** This brief survey cannot take the place of a comprehensive textbook. Readers looking for a more detailed account of the topics introduced here may wish to consult books such as [HU79, BDG95, BDG90, Vol99, Pap94, DK00] or survey chapters such as [ALR99, BS90].

To illustrate what we would like complexity theory to do, it may be best to start by considering a "dream result" that complexity theory *cannot* prove (yet). Consider the following elusive goal, currently far beyond our capabilities.

> *Prove:* Any circuit of NAND gates that will factor 600 digit numbers must have $\geq 2^{100}$ gates.

If this could be proved, surely it would place public-key cryptography on a firm foundation, and would show that factoring is hard with *any* computing technology.

Or would it? Note that factoring is *easy* using "quantum circuits" [Sho97].

This example forces us to consider the following questions:

[*]Supported in part by NSF grant CCR-9734918.

[†]This paper was prepared from notes taken by the second author of the first author's lectures at Kaikoura 2000.

- Are *any* functions this complex?

  **Answer:** Yes! Just count.
  The number of functions on 600 bits is exactly $2^{2^{600}}$.
  A circuit of size $2^{100}$ can be described in $300 \times 2^{100}$ bits.
  Thus, the number of functions with circuits of size $2^{100}$ will be less than the number of descriptions of size $300 \cdot 2^{100}$, and $2^{2^{100 \cdot 300}} <<<<<< 2^{2^{600}}$.

- Are any *interesting* functions this complex? (The function shown to exist in the preceding paragraph is probably something that nobody would *want* to compute, anyway!)

  **Answer:** Consider the following theorem of Stockmeyer [Sto74].

  **Theorem 1.1** (Stockmeyer). *Any circuit that takes as input a formula (in the language of WS1S, the weak second-order theory of one successor) with up to 616 symbols and produces as output a correct answer saying whether the formula is valid or not, requires at least* $10^{123}$ *gates.*

  To quote Stockmeyer:

  > Even if the gates were the size of a proton and were connected by infinitely thin wires, the network would densely fill the known universe.

  The validity problem (even for formulae in the language WS1S) *is* a fairly interesting problem. It frequently arises in work in the computer-aided verification community.

- Aren't all theorems of this sort meaningless? Theorems of this sort all depend on a particular technology. For instance, factoring is easy on quantum circuits, which shows that theorems about NAND circuits are probably irrelevant. In fact, *any* function $f$ is easy with $f$-gates.

  **Two answers:**

  - For every technology there is a complexity theory. Many theorems are invariant under change of technology. (In fact, even Stockmeyer's theorem can be re-proved for quantum circuits, with only slightly different constants.)
  - Complexity theory is (in part) an empirical study. All observations so far show that existing computations can be implemented "efficiently" with NAND gates. Similarly, all existing programming languages are roughly "equivalent". Theorems about NAND circuits will become irrelevant only after someone builds a computer that cannot be efficiently simulated by NAND circuits. (More to the point, theorems about NAND circuits will be interesting as long as most computation is done by computers that can be implemented with NAND circuitry.)

To obtain concrete results, it helps to have a theoretical framework. As a foundation for this framework, we define classes of easy functions.

A complication here is that some functions are "hard" to compute, simply because the output is long, although each bit is easy. (For instance, the exponentiation function $x \mapsto 2^x$ has an exponentially long output, although the $i$th bit of $2^x$ can be computed in time linear in the length of the binary representation of $i$, and hence in many regards it is an "easy" function.) As a solution, we focus on functions with a single bit of output $f : \{0,1\}^* \longrightarrow \{0,1\}$. Equivalently, we focus on languages $\{x : f(x) = 1\}$.

We define *Dtime*$(t)$ as the set of all languages whose characteristic functions can be computed in time $t(n)$ on inputs of length $n$. Similarly, we can define *Dspace*$(t)$ to be the set of all languages whose characteristic functions can be computed using at most $t(n)$ memory locations on inputs of length $n$.

Another complication that arises is the question of what kind of computer is doing the computation. What kind of programming language do we use?

We choose to measure time on a multi-tape Turing machine. (We'll use the abbreviation "*TM*" for "Turing machine".) While this may seem to be an absurd choice, at first glance, we should note that *any* choice of technology would be arbitrary and would soon be obsolete. Turing machines have a very simple architecture (they're the original "RISC" machine), which makes some proofs more simple. Also, for any program implemented on any machine ever built, run-time $t(n)$ can be simulated in time $\leq t(n)^3$ on a Turing machine. Fine distinctions may be lost, but the big picture remains the same.

Finally, now, we have the basic objects of complexity theory, *Dtime*$(t)$ and *Dspace*$(t)$. Note that we have chosen *worst-case* running time in the definition of our complexity classes. This is a reasonable choice, although certainly there are also good reasons for considering average-case complexity. The complexity theory for average-case computation is considerably more complicated. For a survey, consult [Wan97].

A reasonable goal might be to first try to prove "If we have more resources, we can compute more". That is, if $t << T$, then *Dtime*$(t) \subsetneq$ *Dtime*$(T)$.

So, how much bigger must $T$ be?

- $t + 1 \leq T$?
- $t = o(T)$?
- $t \cdot \log t = o(T)$?
- $2^t \leq T$?
- $2^{2^{2^{2^{2^t}}}} \leq T$?

Surprisingly, *none* of these work in general. Consider the following theorem.

**Theorem 1.2** (Gap Theorem[1]). *Let $r$ be* any *computable function. Then there exist computable functions $t$ such that:*

$$Dtime(t) = Dtime(r(t))$$

So we have "weird" functions $t$ such that $[t, r(t)]$ is a "no-man's land", i.e. every Turing Machine runs in time $t(n)$, or takes time more than $r(t(n))$, for all large input lengths $n$.

In response to the Gap Theorem, we are going to consider only "reasonable" time bounds, using the following definition.

**Definition 1.1.** A function $t$ is *time-constructible* if there is a *TM* that, on all inputs of length $n$, runs for exactly $t(n)$ steps.

Every time bound you'd ever care about is time-constructible: $n \log n$, $n^2$, $n^{2.579} \log n$, $2^n$, $n^{\log n}$, ...

**Theorem 1.3.** *Let $t$ and $T$ be time-constructible functions such that $T$ is "a little bigger than $t$". Then:*

$$Dtime(t) \subsetneq Dtime(T)$$

*Proof.* (by diagonalization)

We will build a *TM*, $M$, running in time $T$, such that $\forall\, i$ ($M_i$ runs in time $t \Rightarrow \exists\, x :\ M_i(x) \neq M(x)$).

$M$: on input $x$,
count the number of 1's at the start; that is, $x = 1^i 0 x'$,
compute $n = |x|$,
compute $T(n)$,
simulate as many moves of $M_i(x)$ as possible in time $T(n)$.
If this simulation runs to completion and $M_i(x) = 1$,
then halt and output 0, otherwise halt and output 1.

The time required for the computation above is $\leq n + n + T(n) + T(n) \leq 4T(n)$, so $L(M) \in Dtime(4T(n)) = Dtime(T(n))$.

(The last equality follows from a weird fact about Turing Machines: "constant factors don't matter". This is not realistic, but it is convenient and doesn't hurt the relevance of the theory.)

Let $M_i$ run in time $t$.

The time required to simulate $M_i(x) \approx t(|x|) \cdot$ ("penalty $(i)$") $\ll T(|x|)$ for large $x = 1^i 0 x'$. Thus, the simulation can run to completion, and $M_i(x) \neq M(x)$. □

[1] In this brief write-up, we will not worry about providing careful citations to the original articles where fundamental results were proved. (In this case, it was [Bor72a].) Rather, the reader who wants more details should consult one of the standard texts on the subject, as listed in the introduction.

The above is an example of a diagonalization argument. Diagonalization is good at creating monsters (things that have an unpleasant property, use huge resources, etc.). An amazing fact: Interesting problems have monsters sitting inside them!

A case in point:

**Theorem 1.4** (Part of the proof of Stockmeyer's Theorem). *$\exists A \in Dspace(2^n)$, $A$ requires circuits of size $\geq 2^{n/2}$ on every input length.*

*Proof.* Here is the outline of an algorithm that runs in exponential space, and differs from any function having a small circuit.

On input $x$ of length $n$:
For each bit-string $y$ of length $2^n$ (representing a possible truth-table for $A \cap \Sigma^n$)
    For each circuit $C$ of size $2^{n/2}$
        If $C(z) = y(z)$ for all $z \in \Sigma^n$
            then ($y$ is easy)
                Get next $y$
    EndFor
At this point, $y$ represents a function computed by no circuit of size $\leq 2^{n/2}$.
Output the $x^{th}$ bit of $y$. □

The preceding theorem shows that there is a monster living in $Dspace(2^n)$. The rest of Stockmeyer's theorem involves showing that this monster can be found lurking inside the validity problem.

More precisely, there is an efficient reduction from $A$ to WS1S, i.e. an easy function $f$ such that $x \in A \Leftrightarrow f(x) \in$ WS1S.

If WS1S had small circuits, then so would $A$. In fact, such a reduction exists for every $B \in Dspace(2^n)$, so WS1S is "harder than" everything in $Dspace(2^n)$.

The truly unexpected and fundamental observation is the following:

*Most "natural" problems are "hard" for some complexity class in this sense.*

Since this is such an important notion, it is worthwhile spending some time defining it properly.

To formalize the notion of a "reduction", we need to revisit the notion of an "easy function".

**Desiderata:**

- If $f$ and $g$ are easy, then so is $f \circ g$.
- If $f$ is computable in time $n^2$, then $f$ is easy.

Unfortunately, these two seemingly harmless desiderata have the unpleasant implication that there are "easy" functions requiring time $n^{1000}$. This forces us to consider some tough choices:

- give up on our desiderata, or

- try to justify our (ridiculous) definition of "easy".

We choose the second option. The justification is that we are not really interested in the notion of "easy functions", but rather we are interested in functions that are "difficult" to compute. Note that if a function is *not* "easy" (i.e. if it's not computable in time $n^k$ for any $k$) then it really is "difficult" in an intuitively appealing sense.

**Definition 1.2** (m-reduction). $A \leq_m^P B$ if $\exists\, k\ \exists\, f$, computable in time $n^k + k$, such that $x \in A \Leftrightarrow f(x) \in B$.

The relation $\leq_m^P$ defines a partial order on equivalence classes (where $A \equiv_m^P B$ if $A \leq_m^P B \leq_m^P A$).

Intuitively, this partial order corresponds to the "is no harder than" relation, in the sense that "$A \leq_m^P B$" should roughly mean the same thing as "$A$ is no harder than $B$". It is slightly more precise to translate this as "$A$ is not too much harder than $B$", as is made precise below:

**Theorem 1.5.** *Let $A \leq_m^P B$. Then $B \in Dtime(t(n)) \Rightarrow \exists\, k\ :\ A \in Dtime(n^k + t(k + n^k))$, and $B \in Dspace(t(n)) \Rightarrow \exists\, k\ :\ A \in Dspace(n^k + t(k + n^k))$.*

We now face a new problem. We have two useful tools ($\leq_m^P$ reducibility, and *Dtime* and *Dspace* classes), but the notions of "easiness" they give don't mesh perfectly. What is needed is to modify the time and space complexity classes, in order to obtain classes with some nice *closure* properties.

**Definition 1.3** (Closure). Let $\mathcal{C}$ be a class of languages. $\mathcal{C}$ is *closed* w.r.t. $\leq_m^P$ if:

$$A \leq_m^P B \text{ and } B \in \mathcal{C} \Rightarrow A \in \mathcal{C}$$

Some classes with nice closure properties:

- $P = \bigcup_k Dtime(n^k)$
- $PSPACE = \bigcup_k Dspace(n^k)$
- $EXP = \bigcup_k Dtime(2^{n^k})$
- $EXPSPACE = Dspace(2^{n^{O(1)}})$

Now it is time to give a formal definition of the notion of "hardness" that we introduced earlier.

**Definition 1.4** (Hardness). $A$ is *hard* for $\mathcal{C}$, under $\leq_m^P$, if:

$$\forall\, B \in \mathcal{C}\,,\ B \leq_m^P A$$

Hardness may be viewed as a *lower bound* on the complexity of $A$.

**Definition 1.5** (Completeness). $A$ is *complete* for $\mathcal{C}$, under $\leq_m^P$, if:

- $A \in \mathcal{C}$ and
- $A$ is hard for $\mathcal{C}$ under $\leq_m^P$.

Completeness may be viewed as a *tight lower bound* on the complexity of $A$. A surprise! Nice complexity classes have interesting complete problems.

| Complexity class | Complete problems |
|---|---|
| $P$ | anything with a reasonable algorithm |
| $PSPACE$ | $\{(M,\varphi) : M \vDash \varphi\}$ (Here, $M$ is a finite structure and $\varphi$ is a first-order logic statement. Equivalently, $M$ is a database, and $\varphi$ is a question that is being asked about the database.)<br>RegExp $(\cup\,,\cdot\,,*)$ (This is the problem, given regular expressions $r$ and $s$ using the operations $(\cup\,,\cdot\,,*)$, of determining if $r$ and $s$ are equivalent, in the sense that $L(r) = L(s)$.) |
| $EXP$ | $n \times n$ checkers [Rob84]<br>$n \times n$ Go [Rob84] |
| $EXPSPACE$ | RegExp $(\cup\,,\cdot\,,*\,,^2)$ where $\alpha^2 = \alpha \cdot \alpha$ |

This is starting to look promising! Taking as our starting point a natural formalization of the venerable mathematical concept of "reducibility", we have focused on complexity classes that are closed under this reducibility, and discovered that there are natural complete problems for these complexity classes, and thus we have "tight" lower bounds on the complexity of many of these problems. It is natural to wonder if this approach can be pushed further, to give a better understanding of even more computational problems.

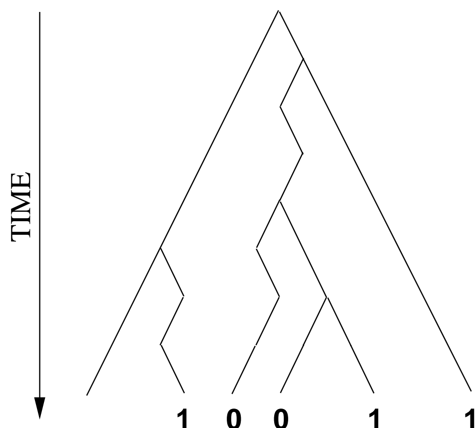
At this point, however, we encounter a disappointment. There are *many* equivalence classes under $\leq_m^P$ that seem *not* to correspond to *Dtime* and *Dspace* classes.

Shock! They correspond to time and space classes on "fantasy" machines. That is, in order to use the tools of reducibility and completeness to understand the complexity of a wider range of problems, we will need to define some very "unrealistic" models of computation.

**Nondeterministic** machines have $\geq 1$ "legal" moves at any given time.

An *NDTM* (*N*on*D*eterministic *T*uring *M*achine) is said to *accept* $x$ in time $t$ if there *exists* a sequence of $\leq t$ legal moves leading to "accept". (Similarly, the machine is said to accept in space $s$ if there *exists* a sequence of legal moves leading to "accept", where no more than $s$ memory locations are accessed **along this sequence of moves**.)

We can think of an *NDTM* computation as a computation tree, as shown here:




Note that, because of the existential quantifier in the definition of what it means for an *NDTM* to accept an input $x$, nondeterministic Turing machines can quickly solve search problems that *seemingly* require exponential time for a deterministic machine to solve. Now, just as for "ordinary" deterministic machines, we can also define complexity classes for nondeterministic machines.

- $Ntime(t)$
- $Nspace(t)$
- $NP = \bigcup_k Ntime(n^k)$

If you have doubts about the wisdom of introducing wildly-unrealistic models of computing such as the *NDTM*, we now argue that more-than-adequate justification is provided by the following list of *NP*-complete problems:

- Traveling Salesperson Problem (see page 9)
- SAT (Boolean Satisfiability)
- Clique
- Hamiltonian path
- 3-Colorability
- ... hundreds more ...

## Review - Lecture 1

- Reducibility is a tool to expose unexpected relationships among seemingly unrelated problems.

- Some equivalence classes $\equiv_M^P$ correspond to *Dtime* and *Dspace* classes, but many seem *not* to.

- *Non-determinism* provides a "computational" view of some of these "problematic" $\equiv_M^P$ classes, and helps explain their (perceived) computational intractability.

**A historical reality check:** It didn't quite happen this way. Theoreticians *like* non-deterministic *TM*'s, and had studied them for years before it turned out that they were useful in practice. *NP*-completeness was initially seen as a "cute" translation of some notions from recursive function theory to complexity theory.

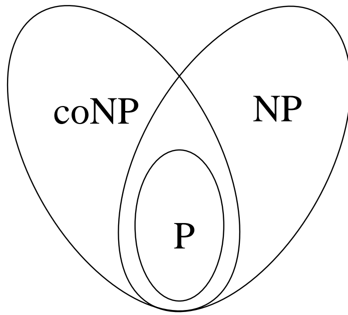
Complexity classes *seem* to differ from each other.

For example: $A \in Dtime(t) \Rightarrow \overline{A} \in Dtime(t)$.
In contrast: $A \in Ntime(t) \Rightarrow \overline{A} \in Ntime(2^{O(t)})$ seems optimal.

In particular: $coNP = \{A : \overline{A} \in NP\}$. The standard *coNP*-complete problem is $TAUT = \{\varphi : \varphi \text{ is a tautology}\}$. Tautologies have *proofs*. It *seems* as if a proof of a tautology on $n$ variables needs to be of size $\approx 2^n$. The question of whether tautologies have short proofs is equivalent to the question of the *Ntime* complexity of *coNP*.

$coNP \subseteq Ntime(t(n^{O(1)})) \Leftrightarrow$ tautologies have "proofs" of size $t(n)$ [CR79].

The conjectured situation is illustrated here:




An example: **Traveling Salesperson Problem**

$$TSP = \{(G, k) : \exists \text{ a path of length } \leq k \text{ visiting all cities on the map } G\}$$

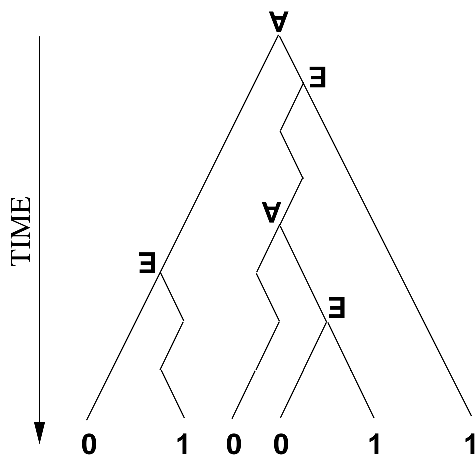
An *NDTM* guesses the path, then checks that the path is valid, and computes the length of the path.

This illustrates the typical way to "program" an *NDTM* via the "guess and check" method (verifying the guess using a deterministic computation). Thus, for problems in *NP* we can check solutions easily. It seems more difficult to *find* solutions.

# 2. Lecture 2

## 2.1. Alternation

*NDTM*'s are like ∃ quantifiers; an *NDTM* accepts if there exists an accepting path. Similarly, we can define ∀ machines, where *all* paths must be accepting. *Alternating* machines allow both ∃ and ∀ states. In the accompanying figure, note that the universal node at the root of the computation tree is *accepting*, since for *all* of the existential nodes that are its children, there *exist* descendents labeled "1".




As for nondeterministic machines, we can define complexity classes $Atime(t)$ and $Aspace(t)$ for alternating machines.

Why do we consider these notions? *Alternation* clarifies the relationship between time and space. To see this, first consider the following chains of inclusions (some of which are trivial, but some of which require proof).

$$Dtime(t) \subseteq Ntime(t) \subseteq Atime(t) \subseteq Dspace(t) \subseteq Nspace(t) \subseteq Aspace(t) \subseteq Dtime(2^{O(t)})$$

$$Nspace(t) \subseteq Atime(t^2) \subseteq Dspace(t^2) \quad \text{[Savitch's Theorem]}$$

$$Dtime(t) \subseteq Aspace(\log t)$$

As a consequence, we see that deterministic time classes correspond to alternating space classes, and vice versa. That is, we have the following corollaries:

- $P = Aspace(\log n)$
- $PSPACE = Atime(n^{O(1)})$

- $EXP = Aspace(n^{O(1)})$
- $EXPSPACE = Atime(2^{n^{O(1)}})$

The equality $Dspace(t^{O(1)}) = Atime(t^{O(1)})$ holds even for bounds $t$ as small as $\log n$ (although we need to modify the definition of Turing machine to allow "random access" to the input, in order to obtain a useful notion of a machine whose running time is significantly less than the length of its input).

It is natural to wonder if there is an even stronger equivalence between deterministic space and alternating time. In particular, the question

$$Dspace(\log n) \overset{???}{=} Atime(\log n)$$

is of great interest. Deterministic logspace is usually denoted $L$, and alternating log time is usually denoted $NC^1$. Thus this question can be restated as:

$$L \overset{???}{=} NC^1$$

With regard to this question, note that we know $Atime(\log n) \subseteq Dspace(\log n)$, but whether $Dspace(\log n) \subseteq Atime(\log n)$ is unknown (and it is not widely believed to hold).

Not only is it unknown if $P = NP$, but it is also unknown if $P = Atime(n^{O(1)})$. Similarly, it is not known if alternating space-bounded classes are more powerful than deterministic space-bounded classes. However, it is useful to note that alternation adds power to *either* time- or space-bounded computation. That is,

**Theorem 2.1. Either** *Atime is more powerful than Dtime* **or** *Aspace is more powerful than Dspace.*

To see why this is true, note that if $Aspace(n^{O(1)}) = Dspace(n^{O(1)})$ and $Dtime(n^{O(1)}) = Atime(n^{O(1)})$, then it follows that $P = EXP$, which we know isn't true.

It is little wonder that most complexity theoreticians conjecture that alternation adds power to **both** time- and space-bounded computation.

Note however that nondeterminism does *not* add much power to space-bounded computation. The inclusion

$$Nspace(s) \subseteq Atime(s^2) \subseteq Dspace(s^2)$$

is usually referred to as Savitch's Theorem. (More precisely, the simulation of nondeterministic space-bounded Turing machines by deterministic ones using only quadratically-more space is called by this name. Alternation provides a useful intermediate step.) It is instructive to see how the proof goes, because this gives you some sense of how to program an alternating machine.

*Proof of Savitch's Theorem ($Nspace(s) \subseteq Atime(s^2)$).* A *configuration* of a machine consists of:

- input head position

- work head(s) position(s)
- work tape contents

Note: the input is not part of the configuration.

We'll assume $s(n) \geq \log n$. Thus a configuration has $\leq O(s(n))$ bits of information. If an *NDTM* has an accepting path, it has an accepting path with no repeated configurations, which therefore has length $\leq 2^{O(s(n))}$.

Our goal: determine if there is a computation path of length $2^{c \cdot s(n)}$ from the initial configuration to the accepting configuration. We write $C \vdash^k D$ to mean there is a path of length less than or equal to $k$ from configuration $C$ to configuration $D$.

Path$(C, D, T)$
Begin
  If $T \leq 1$
    then accept if $C = D$ or $C \vdash^1 D$
    else (we want to accept if there is some $B$ such that $C \vdash^{T/2} B \vdash^{T/2} D$)
    $\exists$ "guess" configuration $B$
    $\forall$ verify:
      Path$(C, B, T/2)$
      Path$(B, D, T/2)$
End □

At this point we have given proofs of almost all of the nontrivial relations among the deterministic and alternating time- and space-bounded classes. One interesting inclusion remains: the simulation of deterministic time by alternating space.

*Proof of Dtime*$(t) \subseteq$ *Aspace*$(\log t)$. First, observe that if $A \in$ *Dtime*$(t)$, then $A$ is accepted by a 1-tape *TM* in time $t^2$.

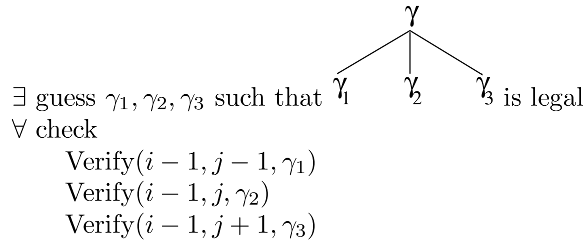
Since $\log(t^2) \in O(\log t)$ we may assume wlog that $A$ is accepted in time $t$ by a 1-tape *TM*. (That is, the proof would be the same if we used a machine running in time $t^2$, but notationally it would be messier.)

We write the machine execution as a table. Consider a table with $t$ rows (one row for each of $t$ time steps), and $t$ columns (one column for each memory location). Thus the $i$th row contains a "picture" of the *TM* at time $i$, and the total size of the table is $t^2$. Everything we do here is very *local*. The *TM* head moves at most one step, writes in one cell, and/or changes state at each time step. Thus, the $j$th cell in row $i$ of the table is determined by locations ($j-1$, $j$, and $j+1$) of row $i-1$.

Our goal is to verify that entry $(t, 1)$ is equal to "$(b, q_{acc})$", where $q_{acc}$ is the accept state of the *TM*.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $t$ | $b, q_{acc}$ | $b$ | $\dots\dots$ | $b$ | $b$ | $\dots$ | $b$ |
| $t-1$ | $\alpha, q_b$ | $b$ | $\dots\dots$ | $b$ | $b$ | $\dots$ | $b$ |
| . | . | . | | . | . | | . |
| . | . | . | | . | . | | . |
| . | . | . | | . | . | | . |
| 2 | $\sigma$ | $x_2, q_1$ | $\dots\dots$ | $x_n$ | $b$ | $\dots$ | $b$ |
| **time** 1 | $x_1, q_0$ | $x_2$ | $\dots\dots$ | $x_n$ | $b$ | $\dots$ | $b$ |
| | **cell** 1 | 2 | $\dots\dots$ | $n$ | $n+1$ | $\dots$ | $t$ |

Verify($i, j, \gamma$) (cell $(i, j)$ of table contains $\gamma$)
If $i = 1$
then (accept $\Leftrightarrow \gamma$ is correct)
else




$\exists$ guess $\gamma_1, \gamma_2, \gamma_3$ such that [tree] is legal
$\forall$ check
Verify($i-1, j-1, \gamma_1$)
Verify($i-1, j, \gamma_2$)
Verify($i-1, j+1, \gamma_3$)
End □

The preceding theorems give some motivation for studying alternation; it clarifies the relationship between time and space. Later on, we will see some additional motivation, but first we should discuss some of the fundamental properties of complexity classes.

## 2.2. Fundamental Properties; Hierarchy Theorems

Complexity classes *seem* to differ from each other. For example, $A \in \mathit{Dtime}(t) \Rightarrow \overline{A} \in \mathit{Dtime}(t)$, but in contrast, $A \in \mathit{Ntime}(t) \Rightarrow \overline{A} \in \mathit{Ntime}(2^{O(t)})$ seems optimal. How about nondeterministic space-bounded classes? Are they closed under complementation?

Note that Savitch's Theorem implies that $A \in \mathit{Nspace}(s) \Rightarrow \overline{A} \in \mathit{Nspace}(s^2)$. People had *believed* that this quadratic overhead might be optimal. Surprisingly, Immerman [Imm88] and Szelepcsényi [Sze88] proved that *Nspace*($s$) *is closed under complementation*!

Another problem related to the complementation of *NDTM*'s is that direct diagonalization doesn't work. To see why, consider the diagonalization argument we gave in Lecture 1, to prove the time hierarchy theorem. The crucial step there was to carry out a simulation of a machine $M_i$ on input $x$, and to *accept if and only if the simulation does NOT accept.* Determining if an *NDTM* does *not* accept seems to be difficult for an *NDTM*, in that it seems to involve simulating a $\forall$ quantifier with an $\exists$ quantifier. (For space-bounded Turing machines, the

Immerman-Szelepcsényi theorem allows us to prove an *Nspace* hierarchy theorem using fairly straightforward diagonalization.)

It turns out that it is possible to prove an *Ntime* hierarchy theorem, too. Before we state the theorem and give the proof, let us first demonstrate that there really is a fundamental difference between the *Dtime* and *Ntime* hierarchies. We know:

$$Dtime(n^2) \subsetneq Dtime(n^3)$$
$$Dtime(2^{2^n}) \subsetneq Dtime((2^{2^n})^{1.5})$$

using the *Dtime* hierarchy theorem. In contrast, the corresponding questions for *Ntime* are much more interesting:

$$Ntime(n^2) \subsetneq Ntime(n^3)\textbf{?} \qquad \text{YES}$$
$$Ntime(2^{2^n}) \subsetneq Ntime((2^{2^n})^{1.5})\textbf{?} \qquad \text{open question}$$

Furthermore, the open question mentioned in the preceding paragraph will not yield to any straightforward attack. To illustrate why, it is necessary to introduce a useful way to modify the model of computation, by providing certain functions at *no cost*. This gives rise to "Oracle Turing Machines", and the classes $Dtime^B(t)$, $Ntime^B(t)$, $PSPACE^B$, $P^B$, $NP^B$, etc. These classes have access to an "oracle" $B$, which means that machines can write a string $z$ on a "query tape" and in one step receive an answer to the question "Is $z \in B$?" The time and space hierarchy theorems that we proved carry over unchanged to these "oracle" classes. In fact, most simulation and diagonalization proofs carry over unchanged from ordinary Turing machines to "oracle" Turing machines.

Now the hierarchy theorem and the open question posed above can be stated more forcefully [RS81]:

$$\forall B \;\; Ntime^B(n^2) \subsetneq Ntime^B(n^3)\textbf{?} \qquad \text{YES!}$$
$$\forall B \;\; Ntime^B(2^{2^n}) \subsetneq Ntime^B((2^{2^n})^{1.5})\textbf{?} \qquad \text{NO! } \exists\, B \text{ where these are equal!}$$

Now, let us state and prove the nondeterministic time hierarchy theorem.

**Theorem 2.2** ([SFM78, Ž83]). *Let $t$ and $T$ be time-constructible, such that $t(n+1) = o(T(n))$. Then $Ntime(t) \subsetneq Ntime(T)$.*

*Proof.* Partition $\mathbb{N}$ into intervals

$$[\text{start}(i_1, y_1)\ ,\ \text{end}(i_1, y_1)]\ [\text{start}(i_2, y_2)\ ,\ \text{end}(i_2, y_2)] \ldots$$

such that $\text{end}(i, y)$ is exponentially bigger than $\text{start}(i, y)$, and $(i_1, y_1), (i_1, y_2), \ldots$ is an enumeration of $\mathbb{N} \times \{0,1\}^*$.

On input $1^n$:
Find region $(i, y)$ containing $n$
  $m :=$ start$(i, y)$
  $z :=$ end$(i, y)$
If $n = z$
  then accept $\Leftrightarrow M_i(1^m)$ does not accept in $T(m)$ time

else accept $\Leftrightarrow M_i(1^{n+1})$ accepts in $T(n)$ time
End

The running time of this algorithm is within $O(T(n))$. We show that the set it accepts is not in the smaller running time as follows: Assume, for contradiction, that $M_i$ accepts this set in *Ntime*$(t)$. Let $m = \text{start}(i, y)$ for some large $y$.

$$\begin{aligned} 1^m \in A \Leftrightarrow & \quad M_i(1^{m+1}) \text{ accepts in time } T(m) \\ \Leftrightarrow & \quad 1^{m+1} \in A \\ \Leftrightarrow & \quad 1^{m+2} \in A \\ & \vdots \\ \Leftrightarrow & \quad 1^{\text{end}(i,y)} \in A \\ \Leftrightarrow & \quad M_i \text{ does not accept } 1^m \\ \Leftrightarrow & \quad 1^m \notin A \end{aligned}$$

□

## 2.3. Alternation and Circuit Complexity

Consider a logspace bounded *ATM*. There are $n^{O(1)}$ configurations, and they have a natural graph structure. Assume wlog that the input is consulted only at *halting* configurations. This is a circuit!

- $\exists$ configurations are $\vee$ gates.
- $\forall$ configurations are $\wedge$ gates.
- Halting configurations are input gates (or constants).

That is, an *ATM* gives, for all $n$, a description of a circuit $C_n$ for the *ATM*, on inputs of length $n$.

A circuit family $\{C_n : n \in \mathbb{N}\}$ that is "easy to describe" is called a *uniform* circuit family.

Below, we list some characterizations of important complexity classes in terms of alternating machines, and (equivalently) in terms of uniform circuits. In so doing, we introduce a new complexity measure for alternating machines: *Alts*, which counts the number of times the machine "alternates" between $\exists$ and $\forall$ states along any path.

$$
\begin{aligned}
P &= \mathit{Aspace}(\log n)\\
&= \text{uniform poly-size circuits}\\[1ex]
AC^1 &= \mathit{Aspace}(O(\log n))\ \mathit{Alts}(O(1))\\
&= \text{uniform log-depth poly-size unbounded fan-in } \wedge,\ \vee \text{ circuits}\\
&= O(\log n) \text{ time on a } \mathit{PRAM} \text{ with } n^{O(1)} \text{ processors}\\[1ex]
NC^1 &= \mathit{Atime}(\log n)\\
&= \text{uniform log-depth fan-in 2 circuits}\\[1ex]
AC^0 &= \mathit{Atime}(O(\log n))\ \mathit{Alts}(O(1))\\
&= \text{unbounded fan-in } \wedge,\ \vee \text{ circuits of poly size and } O(1) \text{ depth}\\
&= \text{First-order logic } (+, \times, >)
\end{aligned}
$$

By "First-order logic $(+, \times, >)$" we mean the class of languages for which there exists a first-order logic formula with predefined function symbols for addition and multiplication, and an order relation. As an example of how a first-order logic formula can define a language, please consider the simple regular set $0^*1^*$ consisting of all strings $x$ such that there is some position $i$ with the property that all bits of $x$ after $i$ are 1, and all bits of $x$ before $i$ are zero. Equivalently, it is the set

$$\{x:\ x \models\ \exists i\, \forall j\ (j > i \rightarrow x[j])\ \wedge\ (i > j \rightarrow \neg x[j])\}$$

For many more reasons than merely the connection to first-order logic, $AC^0$ is a fundamental complexity class (even if it *is* a very *small* subset of $P$).

One advantage of a very small subset of $P$ is that it provides us a tool for "looking inside" $P$. Note for instance that we have already encountered the complexity classes $L$ and $NC^1$. Do these classes correspond to natural computational problems, in the same way that $P$ and $NP$ and $PSPACE$ do? In order to formulate a notion of "completeness" to talk about subclasses of $P$, it is first necessary to formulate a notion of reducibility under which these complexity classes will be closed. $AC^0$ gives an ideal notion of reducibility for this purpose. That is, we define $\leq_m^{AC^0}$ reducibility by analogy to $\leq_m^P$ reducibility; $A \leq_m^{AC^0} B$ means that there is a function $f$ computable in $AC^0$ such that $x \in A$ iff $f(x) \in B$.

*Almost every natural computational problem is complete for some complexity class under $AC^0$ reducibility.* It seems that nature presents us with computational problems corresponding in deep ways to notions of non-determinism, counting, and circuits, and $AC^0$ reducibility helps elucidate this structure.

Below is a short list of some important complexity classes, along with some standard complete problems. We won't present definitions or complete references here. For more details, you can consult [GHR95, ABO99, CM87, Ete97, Bus93].

| Complexity Class | Complete Problems under $\leq_m^{AC^0}$ |
|---|---|
| $P$ | linear programming<br>circuit evaluation<br>least fixed point evaluation |
| $C_=L$ | matrix singularity<br>many questions in linear algebra (rank etc.) |
| $NL$ | finding shortest paths<br>transitive closure |
| $L$ | graph acyclicity<br>tree isomorphism<br>Is $A$ before $B$?[2] |
| $NC^1$ | formula evaluation<br>regular sets |

Two of the classes listed in this table need to be defined. $NL$ is the class of languages accepted by nondeterministic machines using space $O(\log n)$. The related class $C_=L$ is defined in terms of *counting the number of accepting paths* of $NL$ machines. More precisely, a language $A$ is in $C_=L$ if there is a nondeterministic logspace-bounded machine $M$ with the property that $x$ is in $A$ if and only if *exactly half* of the computation paths of $M$ on input $x$ are accepting. It is not hard to show that

$$NC^1 \subseteq L \subseteq NL \subseteq C_=L \subseteq P$$

Advantages of using $\leq_m^{AC^0}$:

- interesting structure is revealed
- completeness $\Rightarrow$ lower bounds

To illustrate what is meant by "lower bounds", consider:

$$\begin{array}{rcl} A \text{ complete for } PSPACE \text{ under } \leq_m^{AC^0} & \Rightarrow & A \notin NL \\ \text{under } \leq_m^{P} & \Rightarrow & [\text{nothing}] \\ A \text{ complete for } NP \text{ under } \leq_m^{AC^0} & \Rightarrow & A \notin AC^0 \\ \text{under } \leq_m^{P} & \Rightarrow & [\text{nothing}] \end{array}$$

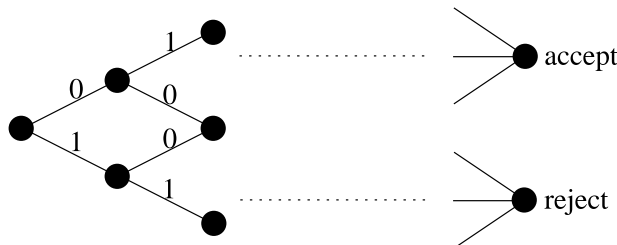
The fact that no language in $AC^0$ can be complete for $NP$ is not entirely easy to prove! A much more interesting fact is that Parity $\notin AC^0$. That is, an $AC^0$ computation cannot tell if the number of 1's in an input string is odd or even.

[3]This is a frustratingly simple problem. The input is a graph that wlog consists of a sequence of edges forming a simple path (but in random order), and two points $A$ and $B$. The question is: does point $A$ come before point $B$ on the line?

# 3. Lecture 3

## 3.1. Branching Programs

We have another way to view $L$ ($Dspace(\log n)$). We use branching programs.




An easy observation ...

$$L = \{A : A \text{ has uniform branching programs of polynomial size}\}$$

Since we need only remember where we are at each step of the execution, a poly-size branching program can be simulated in log space.

It is also interesting to consider **non-uniform** circuits and branching programs. That is, instead of studying what can be computed by small circuits or branching programs *that are easy to build*, we simply concentrate on small circuits/branching programs. Most of the complexity classes that we have seen thus far come in both uniform and non-uniform flavors. For instance, we have the classes:

- $P/poly$
- $NL/poly$
- $L/poly$

Where, for a class $\mathcal{C}$, we define $\mathcal{C}/poly$ to be:

$$\begin{aligned}\{A : &\exists B \in \mathcal{C} \\ &\exists\{\alpha_n : n \in \mathbb{N}\}\ |\alpha_n| = n^{O(1)} \\ &\forall x \ \ x \in A \Leftrightarrow (x, \alpha_{|x|}) \in B\}\end{aligned}$$

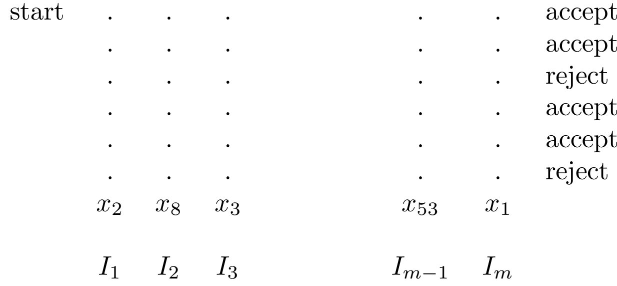
Similarly, one could define $NC^1/poly$, $AC^0/poly$, and so on, but for sociological reasons these classes are usually referred to simply as $NC^1$, $AC^0$, etc., and if one wants to focus on the uniform classes one typically specifies *uniform* $NC^1$, etc. (The fact that there are different versions of uniformity is yet another complication that it is best to simply avoid here in this discussion.)

Just as there has been important work concentrating on circuits of depth $O(1)$, there has been a parallel line of research investigating branching programs of width 1, 2, etc. One of the most lovely theorems about branching programs describes what happens when you consider width $O(1)$.

**Theorem 3.1** (Barrington's Theorem [Bar89]). *(A shock!) $NC^1$ is equal to $\{A :$ $A$ has uniform branching programs of polynomial size and width $O(1)\}$.*

In fact, width 5 suffices.

*Proof.* $\supseteq$ This direction is easy; just use divide and conquer.




The diagram above shows a width-5 branching program. (For simplicity, the edges are not shown.) The next-to-last row indicates which input variable is queried in each column. For instance, at each node in the first column, the second input bit is read; in the second column, the eighth bit is read, etc. The "instruction" $I_j$ in the $j$th column is given by the edges leading from the $j$th column to column $j+1$, saying where to go if the bit is 1 and where to go if the bit is 0.

Note that an input $x$ determines a relation $\Theta_j$ in $\{\Theta_{j_0}, \Theta_{j_1}\}$ for each instruction $I_j$, where $\Theta_{j_b}$ is the subset of $\{1,2,3,4,5\} \times \{1,2,3,4,5\}$ corresponding to the edges that are followed if the input bit read in instruction $j$ is $b$.

Input $x$ is accepted $\Leftrightarrow$ $\Theta_1 \circ \Theta_2 \circ \ldots \circ \Theta_{m-1} \circ \Theta_m$ maps "start" to "accept". Note that each $\Theta_j$ can be encoded by $O(1)$ bits, and computing the composition of two adjacent $\Theta$'s can be done by a constant amount of circuitry; thus in $O(1)$ depth the sequence of $m$ relations can be replaced by $m/2$ relations of the form $\Theta_{2j-1} \circ \Theta_{2j}$. Continuing in this way for $O(\log m)$ steps is thus sufficient to determine if $x$ is accepted or not.

$\subseteq$ We will show how to simulate circuits by restricted branching programs, so that every $\Theta_j$ is a *permutation*, and

- $x$ is accepted $\Leftrightarrow$ $\Theta_1 \circ \Theta_2 \circ \ldots \circ \Theta_{m-1} \circ \Theta_m$ is a 5-cycle
- $x$ is rejected $\Leftrightarrow$ $\Theta_1 \circ \Theta_2 \circ \ldots \circ \Theta_{m-1} \circ \Theta_m$ is the identity

It is not too hard to see that all 5-cycles are equivalent, in the sense that if there is a restricted branching program for a problem using one 5-cycle $\pi$, then for any other 5-cycle $\rho$ there is an equivalent branching program of the same size, using $\rho$.

First note that (via DeMorgan's laws) we can simulate OR gates by AND and NOT gates. Thus the theorem follows easily from the following lemma:

**Lemma 3.2.** *If $A$ is recognized by a circuit of $\{\wedge, \neg\}$ gates of depth $d$, then $A$ has a (restricted) branching program of length $4^d$.*

Basis $d = 1$ (trivial)
Inductive step (2 cases)

- case 1. output gate is $\neg$ (easy)
- case 2. output is $C_1 \wedge C_2$
  Let $P_1$ have size $4^{d-1}$ and accept $C_1$ with $\pi = (1, 2, 3, 4, 5)$.
  Let $P_2$ accept $C_2$ with $\rho = (1, 3, 5, 4, 2)$.
  Let $P_3$ accept $C_1$ with $\pi^{-1}$.
  Let $P_4$ accept $C_2$ with $\rho^{-1}$.
  Then $P_1P_2P_3P_4$ accepts $C_1 \wedge C_2$ with $\pi\rho\pi^{-1}\rho^{-1}$ = (1,3,2,5,4).

If $C_1$ rejects, then $P_1, P_3$ are the identity and $P_2, P_4$ cancel; and similarly for $C_2$. If both $C_1$ and $C_2$ evaluate to 1, then $\pi\rho\pi^{-1}\rho^{-1}$ is a 5-cycle, and in particular is not the identity. □

Although Barrington's theorem has a very simple and elegant proof, there is still not a good intuitive understanding of how a width-5 branching program computes the MAJORITY function (the problem of determining if there are more 1's than 0's in an input string $x$).

## 3.2. The Algebraic Approach to Circuit Complexity

Elements such as $\pi$ and $\rho$ exist only in non-solvable groups (and monoids). It is natural to ask what happens if we consider branching programs built from *solvable* algebras.

We won't present the formal definitions here (but see [MPT91]), but this can be formalized, and a pleasant outcome is that computation over solvable algebras turns out to be exactly what one obtains when one augments $AC^0$ with modular counting gates. More precisely:

- $NC^1$ = "poly-size branching programs over non-solvable monoids"
- $ACC^0$ = "poly-size branching programs over solvable monoids"
  $= \bigcup_m AC^0(m)$
- $AC^0(m)$ = poly-size circuits of depth $O(1)$ with $\wedge, \vee, Mod_m$ gates

$AC^0(m)$ is the current frontier for proofs of circuit lower bounds. For $AC^0(p)$, where $p$ is *prime*, we can prove things:

**Theorem 3.3** ([Smo87] (see [BS90])). *Let $p$ be prime. Let $m$ not be a power of $p$. Then $Mod_m \notin AC^0(p)$.*

The restriction that $p$ be prime is important, as the following list of open questions illustrates.

Open questions:

- Is $NP =$ uniform $AC^0(6)$, depth-3?
- Is $Ntime(2^n) \subseteq$ non-uniform $AC^0(6)$, depth-3?

*Proof Sketch of the lower bound for $AC^0(2)$.* The proof consists of two main steps:

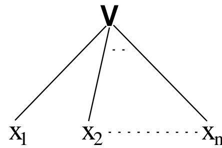
1st step: $A \in AC^0(2) \Rightarrow A$ is recognized by a probabilistic circuit of
size $2^{\log^{O(1)} n}$
depth 2
gates: $Mod_2$ (at output), $\wedge$ of fan-in $\log^{O(1)} n$

2nd step: $Mod_3$ is *not* computed by a circuit of this sort.

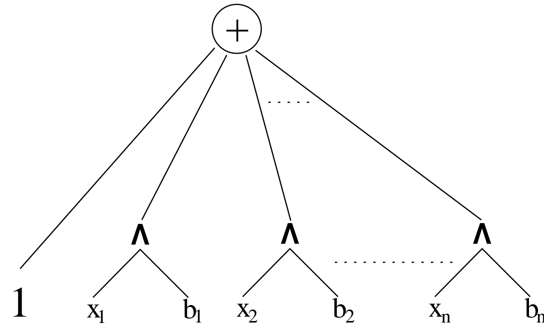
We have time only to say a few words about the first of these two steps. However, this step is interesting in its own right (and it introduces an interesting example of probabilistic computation). Note that it says that the algebraic structure of $AC^0(2)$ is such that it allows any depth $k$ circuit to be replaced by a probabilistic *depth two* circuit of "almost" polynomial size.

First, let us see how to replace a single OR gate by a probabilistic depth-two circuit of the desired form. (An AND gate can be simulated in a similar way.)

To simulate:




we will use:

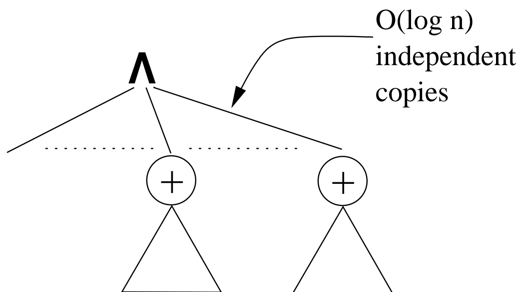



In this new circuit, the bits $b_1, \ldots, b_n$ are probabilistic bits, chosen at random. If any of the original bits $x_1, \ldots, x_n$ are 1, then with probability one-half an even number of those bits will "survive" being masked by the random bits. The output gate of the circuit (labeled "+" in the figure) is a $Mod_2$ gate (also known as a Parity gate).

Note that:
$\vee x_i = 0 \Rightarrow$ output $= 1$

$\vee x_i = 1 \Rightarrow \text{Prob(output } = 0) = 1/2$

Now take $O(\log n)$ independent copies of this and $\wedge$ together.




Now, $\vee x_i = 0 \Rightarrow$ output $= 1$
$\vee x_i = 1 \Rightarrow \text{Prob(output} = 0) \geq 1 - \frac{1}{n^{O(1)}}$

Replace all $\wedge$ and $\vee$ gates in an $AC^0(2)$ circuit by sub-circuits of this form. The output is correct with Prob $\geq 1 - \frac{1}{n^{O(1)}}$.

This circuit can be viewed as a polynomial over $GF(2)$. That is, a $\wedge$ gate is computing multiplication in $GF(2)$, and a Parity gate is computing addition. The degree of this polynomial is $\log^{O(1)} n$. Rewriting each polynomial in sum-of-product form gives us the desired family of probabilistic depth-2 circuits. □

$ACC^0$ probably does not have complete sets under $\leq_m^{AC^0}$. The preceding theorem shows that there could be no complete set for $ACC^0$ in $AC^0(p)$ for any prime $p$.

One other class that probably has no complete sets under $\leq_m^{AC^0}$ is

$$TC^0 = \text{ poly-size circuits, depth } O(1) \text{, of MAJORITY gates}$$

However, $TC^0$ *does* have complete sets, if we consider a slightly more general form of reducibility: $AC^0$ Turing reducibility $\leq_T^{AC^0}$.

**Definition 3.1** (Turing reducibility). $A \leq_T^{AC^0} B$ if there is a family of circuits $\{C_n\}$, of size $n^{O(1)}$, depth $O(1)$, with gates: $\wedge, \vee, \neg,$"$B$", recognizing $A$.

Why care about $TC^0$?

- It is a good theoretical model of "neural nets".
- Here is a sample of some complete problems under $\leq_T^{AC^0}$:
  - Multiplication
  - Sorting
  - MAJORITY

– Division[4]

**Aside:** You might be wondering what the superscripts "1" and "0" denote, in classes such as $NC^1$, $TC^0$, etc. There is actually a family of infinitely many complexity classes, with the following naming scheme:

$$\left.\begin{array}{ll} \text{(bounded fan-in)} & NC^k \\ \text{(unbounded fan-in)} & AC^k \\ \text{(unbounded fan-in, with mod gates)} & ACC^k \\ \text{(threshhold circuits)} & TC^k \end{array}\right\} \text{circuits of depth } \log^k n$$

For all $k$, $NC^k \subseteq AC^k \subseteq ACC^k \subseteq TC^k \subseteq NC^{k+1}$. The union of all of these classes is known as $NC$, and it is a way of formalizing the class of problems for which a feasible number of processors can provide massive speed-up through parallelism.

Earlier, we claimed that $TC^0$ is unlikely to have a complete set. Here is one reason why this seems unlikely.

**Theorem 3.4.** *$TC^0$ has a complete set under $\leq_m^{AC^0}$ $\Rightarrow$ $\exists\, k$ : every set in $TC^0$ has depth $k$ circuits of size $n^{\log^{O(1)} n}$.*

This theorem is an easy consequence of the following lemma.

**Lemma 3.5.** *Every $\leq_m^{AC^0}$ reduction is computed by a depth-3 $TC^0$ circuit of size $n^{\log^{O(1)} n}$.*

We already have seen how to start the proof of this lemma. Namely, any $AC^0$ circuit can be simulated by a probabilistic depth-two $AC^0$ circuit of the size indicated, where the probabilistic circuit has the property that, for every input $x$ of length $n$, the probability (over the choice of random bits) of computing an incorrect output bit is at most $1/4$. Thus if we take $10n$ independent copies of these circuits, and take the MAJORITY vote of the independent depth two circuits (which is easy to do in depth three with MAJORITY gates), then we have a probabilistic circuit that, on each input $x$, computes the correct output with probability at least $1-(1/2^{2n})$.

However, we need a deterministic circuit, and we have only a probabilistic circuit. Now we appeal to a standard trick: make the circuit *deterministic.* Consider the following (exponentially big) matrix, with a row for each possible $n$-bit input, and a column for each possible sequence of random bits. Put a 1 in column $y$ of row $x$ if probabilistic sequence $y$ causes the circuit to produce the correct output input $x$, and put a 0 in that entry otherwise.

---

[4] When these lectures were presented, division was known to be hard for $TC^0$, but it was still unknown whether division was in uniform $TC^0$. At the time, division was known only to lie in *nonuniform* $TC^0$ [RT92]. In the mean time, this has been resolved [Hes01].

| | $2^m$ random sequences | | | | |
|---|---|---|---|---|---|
| $2^n$ inputs $x$ | | | | | |
| $000\ldots0$ | 0 | 0 | $1\ldots$ | $\ldots$ | 1 |
| $000\ldots1$ | | | | | |
| . | | | | | |
| . | | | | | |
| . | | | | | |
| . | 1 | 1 | $0\ldots$ | $\ldots$ | 1 |
| . | | | | | |
| . | | | | | |
| . | | | | | |
| $111\ldots1$ | 0 | 1 | $0\ldots$ | $\ldots$ | 1 |

Each row has $\geq 2^m(1 - \frac{1}{2^{2n}})$ 1's (for "good" random sequences).

Total # of 0's $\leq (2^n \cdot \frac{2^m}{2^{2n}}) < 2^m$. That is, there are fewer 0's than there are columns, and thus some column is all 1's. We hardwire these bits in, instead of using "random" bits. This gives us a deterministic circuit.

Note: This is a *non-uniform* construction. It shows that a sequence exists, but provides no clue about how to find it, other than by brute-force search.

## 3.3. Derandomization

This sort of non-uniform construction of a deterministic algorithm from a randomized algorithm is not very satisfying. This leads to the important problem: can probabilistic algorithms be simulated by *uniform* deterministic algorithms?

*BPP* is the class of problems that can be solved by probabilistic polynomial-time algorithms with negligible error probability. It is useful to give a typical example of a problem in *BPP* but *not* known to be in *P*.

$$\{(f, g) : f \text{ and } g \text{ are arithmetic expressions for multivariate polynomials of degree } n^{O(1)} \text{ with } f = g\}$$

$$\begin{aligned} f = g \Leftrightarrow\ & f - g = 0 \\ \Leftrightarrow\ & \text{a random } x \text{ satisfies } (f - g)(x) = 0 \end{aligned}$$

That is, an easy probabilistic algorithm can determine if two multivariate polynomials are equal; no efficient deterministic algorithm for this task is known.

There is a fundamental problem with implementing probabilistic algorithms; where do we get random bits? It might be possible to hook the computer up to a Geiger counter to extract some theoretically-random bits, but nobody uses truly random sources in real life. Rather, people make use of some off-the-shelf generator that "seems to work well enough". It would be nice to have a *deterministic* algorithm.

There is good news! Recent indications are that we can simulate a probabilistic algorithm deterministically with some "small" overhead.

**Theorem 3.6** ([IW97]). *If* $\exists A \in Dtime(2^{O(n)})$ *such that* $A$ *requires circuits of size* $2^{\varepsilon n}$ *then* $BPP = P$.

The hypothesis to this theorem seems *very* likely to be true. We know that there are classes only "slightly" larger than $Dtime(2^{O(n)})$ that contain problems that are this hard. Unfortunately, at this time, we know of no proof that there is anything in $Dtime(2^{O(n)})$ that does not have *linear*-size depth-three $AC^0(6)$ circuits!

However, under the very likely assumption that there *is* in fact something in $Dtime(2^{O(n)})$ that requires large circuits, then anything you can do with a probabilistic algorithm, can be done efficiently with a deterministic algorithm.

The proof is fairly complicated. The main idea is that a probabilistic algorithm gives us a statistical test. Typical random sequences make the algorithm produce the correct output; an algorithm that produces one answer for most truly random bits and produces a different answer for some "pseudorandom" bits therefore distinguishes the pseudorandom bits from truly random bits. The question boils down to: "Can you generate 'static' random sequences that are indistinguishable from truly random sequences using an easy-to-compute statistical test?"

If a function is hard to compute by a small circuit, then it is in some sense unpredictable and "random-looking". The proof proceeds by starting with this intuition and making it precise, by coming up with a specific way to use a "hard" function to produce a pseudorandom bit generator.[5]

## 3.4. Epilogue

In these three lectures, we have presented the theoretical framework that complexity theoreticians have developed to prove that certain functions are hard to compute. Rather than an unstructured collection of unrelated problems, it has emerged that real-world computational problems (with a few exceptions) naturally fall into a few fundamental equivalence classes corresponding to complete sets for complexity classes.

Many of the fundamental open questions in complexity can be posed as asking if there is a simple reduction from one problem to another. That is,

$$\text{Is } A \leq_m^{AC^0} B?$$

For instance:

$$L \neq NP \Leftrightarrow \text{Travelling Salesperson Problem} \not\leq_m^{AC^0} \text{Is } A \text{ before } B?$$

Showing that there is *not* a reduction from one problem to another frequently seems nearly impossible. But there are some examples of arguments that show *exactly* that.

[5] There is a large and active community working in derandomization; the paper [IW97] cited above builds on a great deal of earlier work, and other exciting developments have followed.

For instance, consider the result of [AAI+97] showing that

$$\{A : A \text{ is } NP\text{-complete under } \leq_m^{AC^0}\} \subsetneq \{A : A \text{ is } NP\text{-complete under } \leq_m^{P}\}$$

That is, there is a set that is complete for $NP$ under poly-time reductions that is not complete under $AC^0$ reducibility. The argument shows that Parity $\not\leq_m^{AC^0}$ Encoding(SAT), using a particular error-correcting encoding of SAT.

## Acknowledgments

The first author thanks Rod Downey for his invitation to come to New Zealand. He thanks Lance Fortnow and Christos Papadimitriou for the parts they played in his receiving this invitation.

## References

[AAI+97] M. Agrawal, E. Allender, R. Impagliazzo, R. Pitassi, and S. Rudich. Reducing the complexity of reductions. In *ACM Symposium on Theory of Computing (STOC)*, pages 730–738. ACM Press, 1997.

[ABO99] E. Allender, R. Beals, and M. Ogihara. The complexity of matrix rank and feasible systems of linear equations. *Computational Complexity*, 8:99–126, 1999.

[ALR99] E. Allender, M. Loui, and K. R. Regan. Three chapters: 27 (Complexity classes), 28 (Reducibility and completeness), and 29 (Other complexity classes and measures). In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[Bar89] D. A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in $NC^1$. *Journal of Computer and System Sciences*, 38:150–164, 1989.

[BDG90] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity II*. Monographs on Theoretical Computer Science. Springer Verlag, Berlin Heidelberg, 1990.

[BDG95] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I*. Texts in Theoretical Computer Science. Springer Verlag, Berlin Heidelberg, 2nd edition, 1995.

[Bor72a] A. Borodin. Computational complexity and the existence of complexity gaps. *Journal of the ACM*, 19:158–174, 1972. See also [Bor72b].

[Bor72b] A. Borodin. Corrigendum: "Computational complexity and the existence of complexity gaps". *Journal of the ACM*, 19:576–576, 1972.

[BS90] R. B. Boppana and M. Sipser. The complexity of finite functions. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 757–804. Elsevier, 1990.

[Bus93] S. R. Buss. Algorithm for boolean formula evaluation and for tree contraction. In P. Clote and J. Krajíček, editors, *Arithmetic, Proof Theory, and Computational Complexity*, pages 96–115. Clarendon Press, Oxford, 1993.

[CM87] S. A. Cook and P. McKenzie. Problems complete for deterministic logarithmic space. *Journal of Algorithms*, 8:385–394, 1987.

[CR79] S. A. Cook and R. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44:36–50, 1979.

[DK00] D.-Z. Du and K.-I. Ko. *Theory of Computational Complexity.* Wiley-Interscience, New York, 2000.

[Ete97] Kousha Etessami. Counting quantifiers, successor relations, and logarithmic space. *Journal of Computer and System Sciences*, 54:400–411, Jun 1997.

[GHR95] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, New York, 1995.

[Hes01] W. Hesse. Division is in uniform $\mathrm{TC}^0$. In *Proc. Twenty-Eighth International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science. Springer, 2001. to appear.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[Imm88] N. Immerman. Nondeterministic space is closed under complement. *SIAM Journal on Computing*, 17:935–938, 1988.

[IW97] R. Impagliazzo and A. Wigderson. $P = BPP$ if $E$ requires exponential circuits: Derandomizing the XOR lemma. In *ACM Symposium on Theory of Computing (STOC)*, pages 220–229. ACM Press, 1997.

[MPT91] Pierre McKenzie, Pierre Péladeau, and Denis Thérien. $\mathrm{NC}^1$: The automata-theoretic viewpoint. *Computational Complexity*, 1:330–359, 1991.

[Pap94] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, New York, 1994.

[Rob84] J. M. Robson. $N$ by $N$ checkers is Exptime complete. *SIAM Journal on Computing*, 13:252–267, 1984.

[RS81] C. W. Rackoff and J. I. Seiferas. Limitations on separating nondeterministic complexity classes. *SIAM Journal on Computing*, 10:742–745, 1981.

[RT92] J. Reif and S. Tate. On threshold circuits and polynomial computation. *SIAM Journal on Computing*, 21:896–908, 1992.

[SFM78] J. Seiferas, M. Fischer, and A. Meyer. Separating nondeterministic time complexity classes. *Journal of the ACM*, 25:146–167, 1978.

[Sho97] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26:1484–1509, 1997.

[Smo87] R. Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *ACM Symposium on Theory of Computing (STOC)*, pages 77–82. ACM Press, 1987.

[Sto74] L. J. Stockmeyer. The complexity of decision problems in automata theory and logic. Technical Report MIT/LCS/TR-133, Massachusetts Institute of Technology, Laboratory for Computer Science, 1974.

[Sze88] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.

[Vol99] H. Vollmer. *Introduction to Circuit Complexity*. Springer-Verlag, 1999.

[Ž83] S. Žàk. A Turing machine hierarchy. *Theoretical Computer Science*, 26:327–333, 1983.

[Wan97] J. Wang. Average-case computational complexity theory. In L. Hemaspaandra and A. Selman, editors, *Complexity Theory Retrospective II*, pages 295–328. Springer-Verlag, 1997.