Complexity Classes

Chapter 27 of the forthcoming *CRC Handbook on Algorithms and Theory of Computation*

Eric Allender[1]

Rutgers University

Michael C. Loui[2]

University of Illinois at Urbana-Champaign

Kenneth W. Regan[3]

State University of New York at Buffalo

1 Introduction

The purposes of complexity theory are to ascertain the amount of computational resources required to solve important computational problems, and to classify problems according to their difficulty. The resource most often discussed is computational time, although memory (space) and circuitry (or hardware) have also been studied. The main challenge of the theory is to prove lower bounds, i.e., that certain problems cannot be solved without expending large amounts of resources. Although it is easy to prove that inherently difficult problems exist, it has turned out to be much more difficult to prove that any *interesting* problems are hard to solve. There has been much more success in providing strong evidence of intractability, based on plausible, widely-held conjectures. In both cases, the mathematical arguments of intractability rely on the notions of *reducibility* and *completeness*—which are the topics of the next chapter. Before one can understand reducibility and completeness, however, one must grasp the notion of a *complexity class*—and that is the topic of this chapter.

First, however, we want to demonstrate that complexity theory really can prove—to even the most skeptical practitioner—that it is hopeless to try to build programs or circuits that solve certain problems. As our example, we consider the manufacture and testing of logic circuits and communication protocols. Many problems in these domains are solved by building a logical formula

[1]Supported by the National Science Foundation under Grant CCR-9509603. Portions of this work were performed while a visiting scholar at the Institute of Mathematical Sciences, Madras, India.

[2]Supported by the National Science Foundation under Grant CCR-9315696.

[3]Supported by the National Science Foundation under Grant CCR-9409104.

over a certain vocabulary, and then determining whether the formula is logically valid, or whether counterexamples (that is, bugs) exist. The choice of vocabulary for the logic is important here, as the next paragraph illustrates.

One particular logic that was studied in [Stockmeyer, 1974] is called WS1S. (We need not be concerned with any details of this logic.) Stockmeyer showed that any circuit that takes as input a formula with up to 616 symbols and produces as output a correct answer saying whether the formula is valid, requires at least 10^{123} gates. According to [Stockmeyer, 1987],

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

Of course, Stockmeyer's theorem holds for one particular sort of circuitry, but the awesome size of the lower bound makes it evident that, no matter how innovative the architecture, no matter how clever the software, no computational machinery will enable us to solve the validity problem in this logic. For the practitioner testing validity of logical formulas, the lessons are (1) be careful with the choice of the logic, (2) use small formulas, and often (3) be satisfied with something less than full validity testing.

In contrast to this result of Stockmeyer, most lower bounds in complexity theory are stated asymptotically. For example, one might show that a particular problem requires time $\Omega(t(n))$ to solve on a Turing machine, for some rapidly-growing function t. For the Turing machine model, no other type of lower bound is possible, because Turing machines have the linear-speed-up property (see Chapter 24, Theorem 3.1). This property makes Turing machines mathematically convenient to work with, since constant factors become irrelevant, but it has the by-product—which some find disturbing—that for any n there is a Turing machine that handles inputs of length n in just n steps by looking up answers in a big table. Nonetheless, these asymptotic lower bounds essentially always can be translated into concrete lower bounds on, say, the number of components of a particular technology, or the number of clock cycles on a particular vendor's machine, that are required to compute a given function on a certain input size.[4]

[4] The skeptical practitioner can still argue that these lower bounds hold only for the worst-case behavior of an algorithm, and that these bounds are irrelevant if the worst case arises very rarely in practice. There is a complexity theory of problems that are hard on average (as a counterpoint to the average case analysis of algorithms considered in Chapter 14), but to date only a small number of natural problems have been shown to be hard in this sense, and this theory is beyond the scope of this volume. See **Further Information** at the end of this chapter.

Sadly, to date, few general complexity-theoretic lower bounds are known that are interesting enough to translate into concrete lower bounds in this sense. Even worse, for the vast majority of important problems that are believed to be difficult, no nontrivial lower bound on complexity is known today. Instead, complexity theory has contributed (1) a way of dividing the computational world up into *complexity classes,* and (2) evidence suggesting that these complexity classes are probably distinct. If this evidence can be replaced by mathematical proof, then we will have an abundance of interesting lower bounds.

1.1 What is a Complexity Class?

Typically, a complexity class is defined by (1) a model of computation, (2) a resource (or collection of resources), and (3) a function known as the *complexity bound* for each resource.

The models used to define complexity classes fall into two main categories: (a) machine-based models, and (b) circuit-based models. Turing machines (TMs) and random-access machines (RAMs) are the two principal families of machine models; they were described in Chapter 24. We describe circuit-based models later, in Section 3. Other kinds of (Turing) machines were also introduced in Chapter 24, including deterministic, nondeterministic, alternating, and oracle machines.

When we wish to model real computations, deterministic machines and circuits are our closest links to reality. Then why consider the other kinds of machines? There are two main reasons.

The most potent reason comes from the computational problems whose complexity we are trying to understand. The most notorious examples are the hundreds of natural NP-complete problems (see [Garey and Johnson, 1988]). To the extent that we understand anything about the complexity of these problems, it is because of the model of *nondeterministic* Turing machines. Nondeterministic machines do not model physical computation devices, but they do model real computational problems. There are many other examples where a particular model of computation has been introduced in order to capture some well-known computational problem in a complexity class. This phenomenon is discussed at greater length in Chapter 29.

The second reason is related to the first. Our desire to understand real computational problems has forced upon us a repertoire of models of computation and resource bounds. In order to understand the relationships between these models and bounds, we combine and mix them and attempt to discover their relative power. Consider, for example, nondeterminism. By considering the complements of languages accepted by nondeterministic machines, researchers were naturally led to the notion of alternating machines. When alternating machines and deterministic machines were compared, a surprising virtual identity of deterministic space and alternating time emerged. Subsequently, alternation was found to be a useful way to model efficient parallel computation. (See Sections 2.8 and 3.4 below.) This phenomenon, whereby models of computation are generalized and modified in order to clarify their relative complexity, has occurred often through the brief history of complexity theory, and has generated some of the most important new insights.

Other underlying principles in complexity theory emerge from the major theorems showing *relationships* between complexity classes. These theorems fall into two broad categories. **Simulation theorems** show that computations in one class can be simulated by computations that meet the defining resource bounds of another class. The containment of nondeterministic logarithmic space (NL) in polynomial time (P), and the equality of the class P with alternating logarithmic space, are simulation theorems. **Separation theorems** show that certain complexity classes are distinct. Complexity theory currently has precious few of these. The main tool used in those separation theorems we have is called **diagonalization**. We illustrate this tool by giving proofs of some separation theorems in this chapter. In the next chapter, however, we show some apparently severe limitations of this tool. This ties in to the general feeling in computer science that *lower bounds are hard to prove*. Our current inability to separate many complexity classes from each other is perhaps the greatest challenge to our intellect posed by complexity theory.

2 Time and Space Complexity Classes

We begin by emphasizing the fundamental resources of time and space for deterministic and nondeterministic Turing machines. We concentrate on resource bounds between logarithmic and exponential, because those bounds have proved to be the most useful for understanding problems that arise in practice.

Time complexity and **space complexity** were defined in Chapter 24, Definition 3.1. We repeat Definition 3.2 of that chapter to define the following **fundamental time classes** and **fundamental space classes**, given functions $t(n)$ and $s(n)$:

- $\mathsf{DTIME}[t(n)]$ is the class of languages decided by deterministic Turing machines of time complexity $t(n)$.
- $\mathsf{NTIME}[t(n)]$ is the class of languages decided by nondeterministic Turing machines of time complexity $t(n)$.
- $\mathsf{DSPACE}[s(n)]$ is the class of languages decided by deterministic Turing machines of space complexity $s(n)$.
- $\mathsf{NSPACE}[s(n)]$ is the class of languages decided by nondeterministic Turing machines of space complexity $s(n)$.

We sometimes abbreviate $\mathsf{DTIME}[t(n)]$ to $\mathsf{DTIME}[t]$ (and so on) when t is understood to be a function, and when no reference is made to the input length n.

2.1 Canonical Complexity Classes

The following are the **canonical complexity classes**:

- $\mathsf{L} = \mathsf{DSPACE}[\log n]$ (deterministic log space)
- $\mathsf{NL} = \mathsf{NSPACE}[\log n]$ (nondeterministic log space)
- $\mathsf{P} = \mathsf{DTIME}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{DTIME}[n^k]$ (polynomial time)
- $\mathsf{NP} = \mathsf{NTIME}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{NTIME}[n^k]$ (nondeterministic polynomial time)
- $\mathsf{PSPACE} = \mathsf{DSPACE}[n^{O(1)}] = \bigcup_{k\geq 1} \mathsf{DSPACE}[n^k]$ (polynomial space)
- $\mathsf{E} = \mathsf{DTIME}[2^{O(n)}] = \bigcup_{k\geq 1} \mathsf{DTIME}[k^n]$
- $\mathsf{NE} = \mathsf{NTIME}[2^{O(n)}] = \bigcup_{k\geq 1} \mathsf{NTIME}[k^n]$
- $\mathsf{EXP} = \mathsf{DTIME}[2^{n^{O(1)}}] = \bigcup_{k\geq 1} \mathsf{DTIME}[2^{n^k}]$ (deterministic exponential time)
- $\mathsf{NEXP} = \mathsf{NTIME}[2^{n^{O(1)}}] = \bigcup_{k\geq 1} \mathsf{NTIME}[2^{n^k}]$ (nondeterministic exponential time)
- $\mathsf{EXPSPACE} = \mathsf{DSPACE}[2^{n^{O(1)}}] = \bigcup_{k\geq 1} \mathsf{DSPACE}[2^{n^k}]$ (exponential space)

The space classes PSPACE and $\mathsf{EXPSPACE}$ are defined in terms of the DSPACE complexity measure. By Savitch's Theorem (see Theorem 2.3 in Section 2.4), the NSPACE measure with polynomial bounds also yields PSPACE, and with $2^{n^{O(1)}}$ bounds yields $\mathsf{EXPSPACE}$.

2.2 Why Focus on These Classes?

The class P contains many familiar problems that can be solved efficiently, such as finding shortest paths in networks, parsing context-free grammars, sorting, matrix multiplication, and linear programming. By definition, in fact, P contains all problems that can be solved by (deterministic) programs of reasonable worst-case time complexity.

But P also contains problems whose best algorithms have time complexity $n^{10^{500}}$. It seems ridiculous to say that such problems are computationally feasible. Nevertheless, there are four important reasons to include these problems:

1. For the main goal of proving lower bounds, it is sensible to have an overly generous notion of the class of feasible problems. That is, if we show that a problem is *not* in P, then we have shown in a very strong way that solution via deterministic algorithms is impractical.

2. The theory of complexity-bounded reducibility (Chapter 28) is predicated on the simple notion that if functions f and g are both easy to compute, then the composition of f and g should also be easy to compute. If we want to allow algorithms of time complexity n^2 to be considered feasible (and certainly many algorithms of this complexity are used daily), then we are immediately led to regard running times $n^4, n^8, \ldots$ as also being feasible. Put another way, the choice is either to lay down an arbitrary and artificial limit on feasibility (and to forgo the desired property that the composition of easy functions be easy), or to go with the natural and overly-generous notion given by P.

3. Polynomial time has served well as the intellectual boundary between feasible and infeasible problems. Empirically, problems of time complexity $n^{10^{500}}$ do not arise in practice, while problems of time complexity $O(n^4)$, and those proved or believed to be $\Omega(2^n)$, occur often. Moreover, once a polynomial-time algorithm for a problem is found, the foot is in the door, and an armada of mathematical and algorithmic techniques can be used to improve the algorithm. Linear programming may be the best known example. The breakthrough $O(n^8)$ time algorithm of [Khachiyan, 1979], for $n \times n$ instances, was impractical in itself, but it prompted an innovation by [Karmarkar, 1984] that produced an algorithm whose running time of about $O(n^3)$ on all cases competes well commercially with the simplex method, which

runs in $O(n^3)$ time in most cases but takes 2^n time in some. Of course, if it should turn out that the Hamiltonian circuit problem (or some other NP-complete problem) has complexity $n^{10^{500}}$, then the theory would need to be overhauled. For the time being, this seems unlikely.

4. We would like our fundamental notions to be independent of arbitrary choices we have made in formalizing our definitions. There is much that is arbitrary and historically accidental in the prevalent choice of the Turing machine as the standard model of computation. This choice does not affect the class P itself, however, because the natural notions of polynomial time for essentially all models of sequential computation that have been devised yield the same class. The random-access and pointer machine models described in Section 4 of Chapter 24 can be simulated by Turing machines with at most a cubic increase in time. Many feel that our "true" experience of running time on real sequential computers falls midway between Turing machines and these more-powerful models, but this only bolsters our conviction that the class P gives the "true" notion of polynomial time.

By analogy to the famous *Church-Turing thesis* (see Chapter 26, Section 4), which states that the definition of a (partial) recursive function captures the intuitive notion of a computable process, several authorities have proposed the following

"Polynomial-Time Church-Turing Thesis." The class P captures the true notion of those problems that are computable in polynomial time by sequential machines, and is the same for any physically relevant model and minimally reasonable time measure of sequential computation that will ever be devised.

This thesis extends also to parallel models if "time" is replaced by the technologically important notion of parallel *work* (see Chapter 45, on parallel computation).

Another way in which the concept of P is robust is that P is characterized by many concepts from logic and mathematics that do not mention machines or time. Some of these characterizations are surveyed in Chapter 29.

The class NP can also be defined by means other than nondeterministic Turing machines. NP equals the class of problems whose solutions can be *verified* quickly, by deterministic machines in polynomial time. Equivalently, NP comprises those languages whose membership proofs can be checked quickly.

For example, one language in NP is the set of composite numbers, written in binary. A proof that a number z is composite can consist of two factors $z_1 \geq 2$ and $z_2 \geq 2$ whose product $z_1 z_2$ equals z. This proof is quick to check if z_1 and z_2 are given, or guessed. Correspondingly, one can design a nondeterministic Turing machine N that on input z branches to write down "guesses" for z_1 and z_2, and then deterministically multiplies them to test whether $z_1 z_2 = z$. Then $L(N)$, the language accepted by N, equals the set of composite numbers, since there exists an accepting computation path if and only if z really is composite. Note that N does not really solve the problem—it just checks the candidate solution proposed by each branch of the computation.

Another important language in NP is the set of satisfiable Boolean formulas, called SAT. A Boolean formula ϕ is satisfiable if there exists a way of assigning `true` or `false` to each variable such that under this truth assignment, the value of ϕ is `true`. For example, the formula $x \wedge (\overline{x} \vee y)$ is satisfiable, but $x \wedge \overline{y} \wedge (\overline{x} \vee y)$ is not satisfiable. A nondeterministic Turing machine N, after checking the syntax of ϕ and counting the number n of variables, can nondeterministically write down an n-bit 0-1 string a on its tape, and then deterministically (and easily) evaluate ϕ for the truth assignment denoted by a. The computation path corresponding to each individual a accepts if and only if $\phi(a) = \texttt{true}$, and so N itself accepts ϕ if and only if ϕ is satisfiable; i.e., $L(N) = \text{SAT}$. Again, this checking of given assignments differs significantly from trying to find an accepting assignment.

The above characterization of NP as the set of problems with easily verified solutions is formalized as follows: $A \in \mathsf{NP}$ if and only if there exist a language $A' \in \mathsf{P}$ and a polynomial p such that for every x, $x \in A$ if and only if there exists a y such that $|y| \leq p(|x|)$ and $(x, y) \in A'$. Here, whenever x belongs to A, y is interpreted as a positive solution to the problem represented by x, or equivalently, as a proof that x belongs to A. The difference between P and NP is that between solving and checking, or between finding a proof of a mathematical theorem and testing whether a candidate proof is correct. In essence, NP represents all sets of theorems with proofs that are short (i.e., of polynomial length), while P represents those statements that can proved or refuted quickly from scratch.

The theory of NP-completeness, together with the many known NP-complete problems, is perhaps the best justification for interest in the classes P and NP. All of the other canonical complexity classes listed above have natural and important problems that are complete for them (under various reducibility relations, the subject of the next chapter). Further motivation for studying L, NL, and PSPACE, comes from their relationships to P and NP. Namely, L and NL are the largest space-bounded classes known to be contained in P, and PSPACE is the smallest space-bounded class known to contain NP. (It is worth mentioning here that NP does not stand for "non-polynomial time"; the class P is a subclass of NP.)

Similarly, EXP is of interest primarily because it is the smallest deterministic time class known to contain NP. The closely-related class E is not known to contain NP; we will see in Section 2.7 the main reason for interest in E.

2.3 Constructibility

Before we go further, we need to introduce the notion of *constructibility.* Without it, no meaningful theory of complexity is possible.

The most basic theorem that one should expect from complexity theory would say, "If you have more resources, you can do more." Unfortunately, if we aren't careful with our definitions, then this claim is false:

Theorem 2.1 (Gap Theorem) *There is a computable time bound* $t(n)$ *such that* $\mathsf{DTIME}[t(n)] = \mathsf{DTIME}[2^{2^{t(n)}}]$.

That is, there is an empty gap between time $t(n)$ and time doubly-exponentially greater than $t(n)$, in the sense that anything that can be computed in the larger time bound can already be computed in the smaller time bound. That is, even with much more time, you can't compute more. This gap can be made much larger than doubly-exponential; for any computable r, there is a computable time bound t such that $\mathsf{DTIME}[t(n)] = \mathsf{DTIME}[r(t(n))]$. Exactly analogous statements hold for the NTIME, DSPACE, and NSPACE measures.

Fortunately, the gap phenomenon cannot happen for time bounds t that anyone would ever be interested in. Indeed, the proof of the Gap Theorem proceeds by showing that one can define a time bound t such that no machine has a running time that is between $t(n)$ and $2^{2^{t(n)}}$. This theorem indicates the need for formulating only those time bounds that actually describe the complexity of some machine.

A function $t(n)$ is **time-constructible** if there exists a deterministic Turing machine that halts after exactly $t(n)$ steps for every input of length n. A function $s(n)$ is **space-constructible** if there exists a deterministic Turing machine that uses exactly $s(n)$ worktape cells for every input of length n. (Most authors consider only functions $t(n) \geq n + 1$ to be time-constructible, and many limit attention to $s(n) \geq \log n$ for space bounds. There do exist sub-logarithmic space-constructible functions, but we prefer to avoid the tricky theory of $o(\log n)$ space bounds.)

For example, $t(n) = n + 1$ is time-constructible. Furthermore, if $t_1(n)$ and $t_2(n)$ are time-constructible, then so are the functions t_1+t_2, $t_1 t_2$, $t_1^{t_2}$, and c^{t_1} for every integer $c > 1$. Consequently, if $p(n)$ is a polynomial, then $p(n) = \Theta(t(n))$ for some time-constructible polynomial function $t(n)$. Similarly, $s(n) = \log n$ is space-constructible, and if $s_1(n)$ and $s_2(n)$ are space-constructible, then so are the functions $s_1 + s_2$, $s_1 s_2$, $s_1^{s_2}$, and c^{s_1} for every integer $c > 1$. Many common functions are space-constructible: e.g., $n \log n$, n^3, 2^n, $n!$.

Constructibility helps eliminate an arbitrary choice in the definition of the basic time and space classes. For general time functions t, the classes $\mathsf{DTIME}[t]$ and $\mathsf{NTIME}[t]$ may vary depending on whether machines are required to halt within t steps on all computation paths, or just on those paths that accept. If t is time-constructible and s is space-constructible, however, then $\mathsf{DTIME}[t]$, $\mathsf{NTIME}[t]$, $\mathsf{DSPACE}[s]$, and $\mathsf{NSPACE}[s]$ can be defined without loss of generality in terms of Turing machines that always halt.

As a general rule, any function $t(n) \geq n+1$ and any function $s(n) \geq \log n$ that one is interested in as a time or space bound, is time- or space-constructible, respectively. As we have seen, little of interest can be proved without restricting attention to constructible functions. This restriction still leaves a rich class of resource bounds.

The Gap Theorem is not the only case where intuitions about complexity are false. Most people also expect that a goal of algorithm design should be to arrive at an optimal algorithm for a given problem. In some cases, however, no algorithm is remotely close to optimal.

Theorem 2.2 (Speed-Up Theorem) *There is a decidable language A such that for every machine M that decides A, with running time $u(n)$, there is another machine M' that decides A much faster: its running time $t(n)$ satisfies $2^{2^{t(n)}} \leq u(n)$ for all but finitely many n.*

This statement, too, holds with any computable function $r(t)$ in place of 2^{2^t}. Put intuitively, the program M' running on an old IBM PC is better than the program M running on the fastest hardware to date. Hence A has no best algorithm, and no well-defined time-complexity function. Unlike the case of the Gap Theorem, the speed-up phenomenon may hold for languages and time bounds of interest. For instance, a problem of time complexity bounded by $t(n) = n^{\log n}$, which is just above polynomial time, may have arbitrary polynomial speed-up—i.e., may have algorithms of time complexity $t(n)^{1/k}$ for all $k > 0$.

One implication of the Speed-Up Theorem is that the complexities of some problems need to be sandwiched between upper and lower bounds. Actually, there is a sense in which every problem has a well defined lower bound on time. For every language A there is a computable function t_0 such that for every time-constructible function t, there is some machine that accepts A within time t if and only if $t = \Omega(t_0)$ [Levin, 1996]. A catch, however, is that t_0 itself may not be time-constructible.

2.4 Basic Relationships

Clearly, for all time functions $t(n)$ and space functions $s(n)$, $\mathsf{DTIME}[t(n)] \subseteq \mathsf{NTIME}[t(n)]$ and $\mathsf{DSPACE}[s(n)] \subseteq \mathsf{NSPACE}[s(n)]$, because a deterministic machine is a special case of a nondeterministic machine. Furthermore, $\mathsf{DTIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$ and $\mathsf{NTIME}[t(n)] \subseteq \mathsf{NSPACE}[t(n)]$, because at each step, a k-tape Turing machine can write on at most $k = O(1)$ previously unwritten cells. The next theorem presents additional important relationships between classes.

Theorem 2.3 *Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function, $s(n) \geq \log n$.*

(a) $\mathsf{NTIME}[t(n)] \subseteq \mathsf{DTIME}[2^{O(t(n))}]$.

(b) $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DTIME}[2^{O(s(n))}]$.

(c) $\mathsf{NTIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$.

(d) **(Savitch's Theorem)** $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DSPACE}[s(n)^2]$.

As a consequence of the first part of this theorem, $\mathsf{NP} \subseteq \mathsf{EXP}$. No better general upper bound on deterministic time is known for languages in NP, however. See Figure 2 for other known inclusion relationships between canonical complexity classes—the classes AC^0, TC^0, and NC^1 are defined in Section 3.4.

Although we do not know whether allowing nondeterminism strictly increases the class of languages decided in polynomial time, Savitch's Theorem says that for space classes, nondeterminism does not help by more than a polynomial amount.

2.5 Complementation

For a language A over an alphabet Σ, define $\overline{A}$ to be the complement of A in the set of words over Σ: $\overline{A} = \Sigma^* - A$. For a class of languages $\mathcal{C}$, define co-$\mathcal{C} = \{\, \overline{A} : A \in \mathcal{C} \,\}$. If $\mathcal{C} =$ co-$\mathcal{C}$, then $\mathcal{C}$ is **closed under complementation.**

In particular, co-NP is the class of languages that are complements of languages in NP. For the language SAT of satisfiable Boolean formulas, $\overline{\text{SAT}}$ is the set of unsatisfiable formulas, whose value is `false` for every truth assignment, together with the syntactically incorrect formulas. A closely related language in co-NP is the set of Boolean tautologies, namely, those formulas whose value is `true` for every truth assignment. The question of whether NP equals co-NP comes down to whether every tautology has a short (i.e., polynomial-sized) proof. The only obvious general way to prove a tautology ϕ in m variables is to verify all 2^m rows of the truth table for ϕ, taking exponential time. Most complexity theorists believe that there is no general way to reduce this time to polynomial, hence that NP $\neq$ co-NP.

Questions about complementation bear directly on the P vs. NP question. It is easy to show that P is closed under complementation (see the next theorem). Consequently, if NP $\neq$ co-NP, then P $\neq$ NP.

Theorem 2.4 (Complementation Theorems) *Let t be a time-constructible function, and let s be a space-constructible function, with $s(n) \geq \log n$ for all n. Then*

(a) DTIME$[t]$ *is closed under complementation.*

(b) DSPACE$[s]$ *is closed under complementation.*

(c) **(Immerman-Szelepcsényi Theorem)** NSPACE$[s]$ *is closed under complementation.*

The Complementation Theorems are used to prove the Hierarchy Theorems in the next section.

2.6 Hierarchy Theorems and Diagonalization

Diagonalization is the most useful technique for proving the existence of computationally difficult problems. In this section, we will see examples of two rather different types of arguments, both of which can be called "diagonalization," and we will see how these are used to prove hierarchy theorems in complexity theory.

A hierarchy theorem is a theorem that says "If you have more resources, you can compute more." As we saw in Section 2.3, this theorem is possible only if we restrict attention to constructible time and space bounds. Next, we state hierarchy theorems for deterministic and nondeterministic time and space classes. In the following, $\subset$ denotes *strict* inclusion between complexity classes.

Theorem 2.5 (Hierarchy Theorems) *Let t_1 and t_2 be time-constructible functions, and let s_1 and s_2 be space-constructible functions, with $s_1(n), s_2(n) \geq \log n$ for all n.*

(a) If $t_1(n) \log t_1(n) = o(t_2(n))$, then $\mathsf{DTIME}[t_1] \subset \mathsf{DTIME}[t_2]$.

(b) If $t_1(n+1) = o(t_2(n))$, then $\mathsf{NTIME}[t_1] \subset \mathsf{NTIME}[t_2]$.

(c) If $s_1(n) = o(s_2(n))$, then $\mathsf{DSPACE}[s_1] \subset \mathsf{DSPACE}[s_2]$.

(d) If $s_1(n) = o(s_2(n))$, then $\mathsf{NSPACE}[s_1] \subset \mathsf{NSPACE}[s_2]$.

As a corollary of the Hierarchy Theorem for DTIME,

$$\mathsf{P} \subseteq \mathsf{DTIME}[n^{\log n}] \subset \mathsf{DTIME}[2^n] \subseteq \mathsf{E};$$

hence we have the strict inclusion $\mathsf{P} \subset \mathsf{E}$. Although we do not know whether $\mathsf{P} \subset \mathsf{NP}$, there exists a problem in E that cannot be solved in polynomial time. Other consequences of the Hierarchy Theorems are $\mathsf{NE} \subset \mathsf{NEXP}$ and $\mathsf{NL} \subset \mathsf{PSPACE}$.

In the Hierarchy Theorem for DTIME, the hypothesis on t_1 and t_2 is $t_1(n) \log t_1(n) = o(t_2(n))$, instead of $t_1(n) = o(t_2(n))$, for technical reasons related to the simulation of machines with multiple worktapes by a single universal Turing machine with a fixed number of worktapes. Other computational models, such as random access machines, enjoy tighter time hierarchy theorems.

All proofs of the Hierarchy Theorems use the technique of **diagonalization**. For example, the proof for DTIME constructs a Turing machine M of time complexity t_2 that considers all machines

$M_1, M_2, \ldots$ whose time complexity is t_1; for each i, the proof finds a word x_i that is accepted by M if and only if $x_i \notin L(M_i)$, the language decided by M_i. Consequently, $L(M)$, the language decided by M, differs from each $L(M_i)$, hence $L(M) \notin \mathsf{DTIME}[t_1]$. The diagonalization technique resembles the classic method used to prove that the real numbers are uncountable, by constructing a number whose j^{th} digit differs from the j^{th} digit of the j^{th} number on the list. To illustrate the diagonalization technique, we outline proofs of the Hierarchy Theorems for DSPACE and for NTIME. In this subsection, $\langle i, x\rangle$ stands for the string $0^i 1x$, and $\mathit{zeroes}(y)$ stands for the number of 0's that a given string y starts with. Note that $\mathit{zeroes}(\langle i, x\rangle) = i$.

Proof. (of the DSPACE Hierarchy Theorem)

We construct a deterministic Turing machine M that decides a language A such that $A \in \mathsf{DSPACE}[s_2] - \mathsf{DSPACE}[s_1]$.

Let U be a deterministic universal Turing machine, as described in Chapter 26, Section 2.2. On input x of length n, machine M performs the following:

1. Lay out $s_2(n)$ cells on a worktape.
2. Let $i = \mathit{zeroes}(x)$.
3. Simulate the universal machine U on input $\langle i, x\rangle$. Accept x if U tries to use more than s_2 worktape cells. (We omit some technical details, such as interleaving multiple worktapes onto the fixed number of worktapes of M, and the way in which the constructibility of s_2 is used to ensure that this process halts.)
4. If U accepts $\langle i, x\rangle$, then reject; if U rejects $\langle i, x\rangle$, then accept.

Clearly, M always halts and uses space $O(s_2(n))$. Let $A = L(M)$.

Suppose $A \in \mathsf{DSPACE}[s_1(n)]$. Then there is some Turing machine M_j accepting A using space at most $s_1(n)$. The universal Turing machine U can easily be given the property that its space needed to simulate a given Turing machine M_j is at most a constant factor higher than the space used by M_j itself. More precisely, there is a constant k depending only on j (in fact, we can take $k = |j|$), such that U, on inputs z of the form $z = \langle j, x\rangle$, uses at most $ks_1(|x|)$ space.

Since $s_1(n) = o(s_2(n))$, there is an n_0 such that $ks_1(n) \leq s_2(n)$ for all $n \geq n_0$. Let x be a string of length greater than n_0 such that the first $j+1$ symbols of x are $0^j 1$. Note that the universal

Turing machine U, on input $\langle j, x\rangle$, simulates M_j on input x and uses space at most $ks_1(n) \leq s_2(n)$. Thus, when we consider the machine M defining A, we see that on input x the simulation does not stop in step 3, but continues on to step 4, and thus $x \in A$ if and only if U rejects $\langle j, x\rangle$. Consequently, M_j does not accept A, contrary to our assumption. Thus $A \notin \mathsf{DSPACE}[s_1(n)]$. □

A more sophisticated argument is required to prove the Hierarchy Theorem for NTIME. To see why, note that it is necessary to diagonalize against *nondeterministic* machines, and thus it is necessary to use a nondeterministic universal Turing machine as well. In the deterministic case, when we simulated an accepting computation of the universal machine, we would reject, and if we simulated a rejecting computation of the universal machine, we would accept. That is, we would do exactly the opposite of what the universal machine does, in order to "fool" each simulated machine M_i. If the machines under consideration are *nondeterministic*, then M_i can have both an accepting path and a rejecting path on input x, in which case the universal nondeterministic machine would accept input $\langle i, x\rangle$. If we simulate the universal machine on an input and accept upon reaching a rejecting leaf and reject if upon reaching an accepting leaf, then this simulation would still accept (because the simulation that follows the rejecting path now accepts). Thus, we would fail to do the opposite of what M_i does.

The following careful argument guarantees that each machine M_i is fooled on some input. It draws on a result of [Book *et al.*, 1970] that every language in $\mathsf{NTIME}[t(n)]$ is accepted by a two-tape nondeterministic Turing machine that runs in time $t(n)$.

Proof. (of the NTIME Hierarchy Theorem)

Let $M_1, M_2, \ldots$ be an enumeration of two-tape nondeterministic Turing machines running in time $t_1(n)$. Let f be a rapidly-growing function such that time $f(i, n, s)$ is enough time for a deterministic machine to compute the function

$$(i, n, s) \mapsto \begin{cases} 1 & \text{if } M_i \text{ accepts } 1^n \text{ in } \leq s \text{ steps} \\ 0 & \text{otherwise} \end{cases}$$

Letting $f(i, n, s)$ be greater than $2^{2^{i+n+s}}$ is sufficient.

Now divide Σ^* into regions, so that in region $j = \langle i, y\rangle$, we try to "fool" machine M_i. Note that each M_i is considered infinitely often. The regions are defined by functions $start(j)$ and $end(j)$, defined as follows: $start(1) = 1$, $start(j+1) = end(j) + 1$, where taking $i = zeroes(j)$, we have

$end(j) = f(i, start(j), t_2(start(j)))$. The important point is that, on input $1^{end(j)}$, a deterministic machine can, in time $t_2(end(j))$, determine whether M_i accepts $1^{start(j)}$ in at most $t_2(start(j))$ steps.

By picking f appropriately easy to invert, we can guarantee that, on input 1^n, we can in time $t_2(n)$ determine which region j contains n.

Now it is easy to verify that the following routine can be performed in time $t_2(n)$ by a nondeterministic machine. (In the pseudo-code below, U is a "universal" nondeterministic machine with 4 tapes, which is therefore able to simulate one step of machine M_i in $O(i^3)$ steps.)

1. On input 1^n, determine which region j contains n. Let $j = \langle i, y \rangle$.

2. If $n = end(j)$, then accept if and only if M_i does *not* accept $1^{start(j)}$ within $t_2(start(j))$ steps.

3. Otherwise, accept if and only if U accepts $\langle i, 1^{n+1} \rangle$ within $t_2(n)$ steps. (Here, it is important that we are talking about $t_2(n)$ steps of U, which may be only about $t_2(n)/i^3$ steps of M_i.)

Let us call the language accepted by this procedure A. Clearly $A \in \mathsf{NTIME}[t_2(n)]$. We now claim that $A \notin \mathsf{NTIME}[t_1(n)]$.

Assume otherwise, and let M_i be the nondeterministic machine accepting A in time $t_1(n)$. Recall that M_i has only two tapes. Let c be a constant such that $i^3 t_1(n+1) < t_2(n)$ for all $n \geq c$. Let y be a string such that $|y| \geq c$, and consider stage $j = \langle i, y \rangle$. Then for all n such that $start(j) \leq n < end(j)$, we have $1^n \in A$ if and only if $1^{n+1} \in A$. However this contradicts the fact that $1^{start(j)} \in A$ if and only if $1^{end(j)} \notin A$. $\Box$

Although the diagonalization technique successfully separates some pairs of complexity classes, diagonalization does not seem strong enough to separate P from NP. (See Theorem 7.1 in Chapter 28.)

2.7 Padding Arguments

A useful technique for establishing relationships between complexity classes is the **padding argument.** Let A be a language over alphabet Σ, and let $\#$ be a symbol not in Σ. Let f be a numeric function. The **f-padded version of** A is the language

$$A' = \{x\#^{f(n)} : x \in A \text{ and } n = |x|\}$$

That is, each word of A' is a word in A concatenated with $f(n)$ consecutive $\#$ symbols. The padded version A' has the same information content as A, but because each word is longer, the computational complexity of A' is smaller!

The proof of the next theorem illustrates the use of a padding argument.

Theorem 2.6 *If* $\mathsf{P} = \mathsf{NP}$, *then* $\mathsf{E} = \mathsf{NE}$.

Proof. Since $\mathsf{E} \subseteq \mathsf{NE}$, we prove that $\mathsf{NE} \subseteq \mathsf{E}$.

Let $A \in \mathsf{NE}$ be decided by a nondeterministic Turing machine M in at most $t(n) = k^n$ time for some constant integer k. Let A' be the $t(n)$-padded version of A. From M, we construct a nondeterministic Turing machine M' that decides A' in linear time: M' checks that its input has the correct format, using the time-constructibility of t; then M' runs M on the prefix of the input preceding the first $\#$ symbol. Thus, $A' \in \mathsf{NP}$.

If $\mathsf{P} = \mathsf{NP}$, then there is a deterministic Turing machine D' that decides A' in at most $p'(n)$ time for some polynomial p'. From D', we construct a deterministic Turing machine D that decides A, as follows. On input x of length n, since $t(n)$ is time-constructible, machine D constructs $x\#^{t(n)}$, whose length is $n+t(n)$, in $O(t(n))$ time. Then D runs D' on this input word. The time complexity of D is at most $O(t(n)) + p'(n + t(n)) = 2^{O(n)}$. Therefore, $\mathsf{NE} \subseteq \mathsf{E}$. $\square$

A similar argument shows that the $\mathsf{E} = \mathsf{NE}$ question is equivalent to the question of whether $\mathsf{NP} - \mathsf{P}$ contains a subset of 1^*, that is, a language over a single-letter alphabet.

Padding arguments sometimes can be used to give tighter hierarchies than can obtained by straightforward diagonalization. For instance, Theorem 2.5 leaves open the question of whether, say, $\mathsf{DTIME}[n^3 \log^{1/2} n] = \mathsf{DTIME}[n^3]$. We can show that these classes are not equal, by using a padding argument. We will need the following lemma, whose proof is similar to that of Theorem 2.6.

Lemma 2.7 (Translational Lemma) *Let* t_1, t_2, *and* f *be time-constructible functions. If* $\mathsf{DTIME}[t_1(n)] = \mathsf{DTIME}[t_2(n)]$, *then* $\mathsf{DTIME}[t_1(f(n))] = \mathsf{DTIME}[t_2(f(n))]$.

Theorem 2.8 *For any real number* $a > 0$ *and natural number* $k \geq 1$, $\mathsf{DTIME}[n^k] \subset \mathsf{DTIME}[n^k \log^a n]$.

Proof. Suppose for contradiction that $\mathsf{DTIME}[n^k] = \mathsf{DTIME}[n^k \log^a n]$. For now let us also suppose that $a > 1/2$. Taking $f(n) = 2^{n/k}$, and using the linear speed-up property, we obtain from the Translational Lemma the identity $\mathsf{DTIME}[2^n n^a] = \mathsf{DTIME}[2^n]$. This does not yet give the desired contradiction to the DTIME Hierarchy Theorem—but it is close. We'll need to use the Translational Lemma twice more.

Assume that $\mathsf{DTIME}[2^n n^a] = \mathsf{DTIME}[2^n]$. Using the Translational Lemma with $f(n) = 2^n$ yields $\mathsf{DTIME}[2^{2^n} 2^{an}] = \mathsf{DTIME}[2^{2^n}]$. Applying the Lemma once again on the classes $\mathsf{DTIME}[2^n n^a] = \mathsf{DTIME}[2^n]$, this time using $f(n) = 2^n + an$, we obtain $\mathsf{DTIME}[2^{2^n} 2^{an} f(n)^a] = \mathsf{DTIME}[2^{2^n} 2^{an}]$. Combining these two equalities yields $\mathsf{DTIME}[2^{2^n} 2^{an} f(n)^a] = \mathsf{DTIME}[2^{2^n}]$. Since $f(n)^a > 2^{an}$, we have that $2^{an} f(n)^a > 2^{2an} = 2^n 2^{bn}$ for some $b > 0$ (since $a > 1/2$). Thus $\mathsf{DTIME}[2^{2^n} 2^n 2^{bn}] = \mathsf{DTIME}[2^{2^n}]$, and this result contradicts the DTIME Hierarchy Theorem, since $2^{2^n} \log 2^{2^n} = o(2^{2^n} 2^n 2^{bn})$.

Finally, for any fixed $a > 0$, not just $a > 1/2$, we need to apply the Translational Lemma several more times. $\square$

One consequence of this theorem is that within P, there can be no "complexity gaps" of size $(\log n)^{\Omega(1)}$.

2.8 Alternating Complexity Classes

In this section, we define time and space complexity classes for alternating Turing machines, and we show how these classes are related to the classes introduced already. Alternating Turing machines and their *configurations* are defined in Chapter 24, Section 2.4.

The possible computations of an alternating Turing machine M on an input word x can be represented by a tree T_x in which the root is the initial configuration, and the children of a nonterminal node C are the configurations reachable from C by one step of M. For a word x in $L(M)$, define an **accepting subtree** S of T_x as follows:

- S is finite.
- The root of S is the initial configuration with input word x.
- If S has an existential configuration C, then S has exactly one child of C in T_x; if S has a universal configuration C, then S has all children of C in T_x.

- Every leaf is a configuration whose state is the accepting state q_A.

Observe that each node in S is an accepting configuration.

We consider only alternating Turing machines that always halt. For $x \in L(M)$, define the time taken by M to be the height of the shortest accepting tree for x, and the space to be the maximum number of non-blank worktape cells among configurations in the accepting tree that minimizes this number. For $x \notin L(M)$, define the time to be the height of T_x, and the space to be the maximum number of non-blank worktape cells among configurations in T_x.

Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function. Define the following complexity classes:

- $\mathsf{ATIME}[t(n)]$ is the class of languages decided by alternating Turing machines of time complexity $O(t(n))$.

- $\mathsf{ASPACE}[s(n)]$ is the class of languages decided by alternating Turing machines of space complexity $O(s(n))$.

Because a nondeterministic Turing machine is a special case of an alternating Turing machine, for every $t(n)$ and $s(n)$, $\mathsf{NTIME}(t) \subseteq \mathsf{ATIME}(t)$ and $\mathsf{NSPACE}(s) \subseteq \mathsf{ASPACE}(s)$. The next theorem states further relationships between computational resources used by alternating Turing machines, and resources used by deterministic and nondeterministic Turing machines.

Theorem 2.9 (Alternation Theorems) *Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function, $s(n) \geq \log n$.*

(a) $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{ATIME}[s(n)^2]$

(b) $\mathsf{ATIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)]$

(c) $\mathsf{ASPACE}[s(n)] \subseteq \mathsf{DTIME}[2^{O(s(n))}]$

(d) $\mathsf{DTIME}[t(n)] \subseteq \mathsf{ASPACE}[\log t(n)]$

In other words, space on deterministic and nondeterministic Turing machines is polynomially related to time on alternating Turing machines. Space on alternating Turing machines is exponentially related to time on deterministic Turing machines. The following corollary is immediate.

Theorem 2.10

(a) $\mathsf{ASPACE}[O(\log n)] = \mathsf{P}$.

(b) $\mathsf{ATIME}[n^{O(1)}] = \mathsf{PSPACE}$.

(c) $\mathsf{ASPACE}[n^{O(1)}] = \mathsf{EXP}$.

Note that Theorem 2.9(a) says, for instance, that NL is contained in $\mathsf{ATIME}(\log^2(n))$. For this to make sense, it is necessary to modify the definition of alternating Turing machines to allow them to read individual bits of the input in constant time, rather than requiring n time units to traverse the entire input tape. This has become the standard definition of alternating Turing machines, because it is useful in establishing relationships between Turing machine complexity and circuit complexity, as explained in the upcoming section.

3 Circuit Complexity

Up to now, this chapter has been concerned only with complexity classes that were defined in order to understand the nature of sequential computation. Although we called them "machines," the models discussed here and in Chapter 24 are closer in spirit to software, namely to sequential algorithms or to single-processor machine-language programs. Circuits were originally studied to model hardware. The hardware of electronic digital computers is based on digital gates, connected into combinational and sequential networks. Whereas a software program can branch and even modify itself while running, hardware components on today's typical machines are fixed and cannot reconfigure themselves. Also, circuits capture well the notion of non-branching, straight-line computation.

Furthermore, circuits provide a good model of parallel computation. Many machine models, complexity measures, and classes for parallel computation have been devised, but the circuit complexity classes defined here coincide with most of them. Chapter 45 in this volume surveys parallel models and their relation to circuits in more detail.

3.1 Kinds of Circuits

A *circuit* can be formalized as a directed graph with some number n of sources, called *input nodes* and labeled $x_1, \ldots, x_n$, and one sink, called the *output node*. The edges of the graph are called *wires*.

Every non-input node v is called a *gate*, and has an associated *gate function* g_v that takes as many arguments as there are wires coming into v. In this survey we limit attention to Boolean circuits, meaning that each argument is 0 or 1, although arithmetical circuits with numeric arguments and $+, *$ (etc.) gates have also been studied in complexity theory. Formally g_v is a function from $\{0,1\}^r$ to $\{0,1\}$, where r is the *fan-in* of v. The value of the gate is transmitted along each wire that goes out of v. The *size* of a circuit is the number of nodes in it.

We restrict attention to circuits C in which the graph is acyclic, so that there is no "feedback." Then every Boolean assignment $x \in \{0,1\}^n$ of values to the input nodes determines a unique value for every gate and wire, and the value of the output gate is the output $C(x)$ of the circuit. The circuit *accepts* x if $C(x) = 1$.

The sequential view of a circuit is obtained by numbering the gates in a manner that respects the edge relation, meaning that for all edges (u, v), g_u has a lower number than g_v. Then the gate functions in that order become a sequence of basic instructions in a straight-line program that computes $C(x)$. The size of the circuit becomes the number of steps in the program. However, this view presumes a single processing unit that evaluates the instructions in sequence, and ignores information that the graphical layout provides. A more powerful view regards the gates as simple processing units that can act in parallel. Every gate whose incoming wires all come from input nodes can act and compute its value at step 1, and every other gate can act and transmit its value at the first step after all gates on its incoming wires have computed their values. The number of steps for this process is the *depth* of the circuit. Depth is a notion of *parallel time complexity*. A circuit with small depth is a fast circuit. The circuit *size* in this view is the amount of hardware needed. Chapter 45 gives much more information on the the correspondence between circuits and parallel machines, and gives formal definitions of size and depth.

A *circuit family* $\mathbf{C}$ consists of a sequence of circuits $\{C_1, C_2, \ldots\}$, where each C_n has n input nodes. The language accepted by the family is $L(\mathbf{C}) = \{x : C_{|x|} \text{ accepts } x\}$. (Circuit families computing functions $f : \{0,1\}^* \to \{0,1\}^*$ are defined in Chapter 45.)

The *size complexity* of the family is the function $z(n)$ giving the number of nodes in C_n. The *depth complexity* is the function $d(n)$ giving the depth of C_n.

Another aspect of circuits that must be specified in order to define complexity classes is the underlying technology. By technology we mean the types of gates that are used as components in the circuits. Three types of technology are considered in this chapter:

(1) Bounded fan-in gates, usually taken to be the "standard basis" of binary $\wedge$ (AND), binary $\vee$ (OR), and unary $\neg$ (NOT) gates. A notable alternative is to use NAND gates.

(2) Unbounded fan-in $\wedge$ and $\vee$ gates (together with unary $\neg$ gates).

(3) Threshold gates. For our purposes, it suffices to consider the simplest kind of threshold gate, called the $\mathsf{MAJORITY}$ gate, which also uses the Boolean domain. A $\mathsf{MAJORITY}$ gate outputs 1 if and only if at least $r/2$ of its r incoming wires have value 1. These gates can simulate unbounded fan-in $\wedge$ and $\vee$ with the help of "dummy wires." Threshold circuits also have unary $\neg$ gates.

The difference between (1) and (2) corresponds to general technological issues about high-bandwidth connections, whether they are feasible and how powerful they are. Circuits of type (1) can be converted to equivalent circuits that also have bounded fan-*out*, with only a constant-factor penalty in size and depth. Thus the difference also raises issues about one-to-many broadcast and all-to-one reception.

Threshold gates model the technology of *neural networks*, which were formalized in the 1940s. The kind of threshold gate studied most often in neural networks uses Boolean arguments and values, with '1' for "firing" and '0' for "off." It has numerical *weights* $w_1, \ldots, w_r$ for each of the r incoming wires and a *threshold* t. Letting $a_1, \ldots, a_r$ stand for the incoming 0-1 values, the gate outputs 1 if $\sum_{i=1}^{r} a_i w_i \geq t$, 0 otherwise. Thus the $\mathsf{MAJORITY}$ gate is the special case with $w_1 = \ldots = w_r = 1$ and $t = r/2$. A depth-2 (sub-)circuit of $\mathsf{MAJORITY}$ gates can simulate this general threshold gate.

3.2 Uniformity and Circuit Classes

One tricky aspect of circuit complexity is the fact that many functions that are *not computable* have trivial circuit complexity! For instance, let K be a non-computable set of numbers, such as the indices of halting Turing machines, and let A be the language $\{x : |x| \in K\}$. For each n, if $n \in K$, then define C_n by attaching a $\neg$ gate to input x_1 and an OR gate whose two wires come from the $\neg$ gate and x_1 itself. If $n \notin K$, then define C_n similarly but with an AND gate in place of the OR. The circuit family $[C_n]$ so defined accepts A and has size and depth 2. The rub, however, is that there is no algorithm to tell *which* choice for C_n to define for each n. A related anomaly is that there are uncountably many circuit families. Indeed, every language is accepted by some circuit family $[C_n]$ with size complexity $2^{O(n)}$ and depth complexity 3 (unbounded fan-in) or $O(n)$ (bounded fan-in). Consequently, for general circuits, size complexity is at most exponential, and depth complexity is at most linear.

The notion of **uniform circuit complexity** avoids both anomalies. A circuit family $[C_n]$ is *uniform* if there is an easy algorithm Q that, given n, outputs an encoding of C_n. Either the adjacency-matrix or the edge-list representation of the graphs of the circuits C_n, together with the gate type of each node, may serve for our purposes as the *standard encoding scheme* for circuit families. If Q runs in polynomial time, then the circuit family is P-*uniform*, and so on.

P-uniformity is natural because it defines those families of circuits that are feasible to construct. However, we most often use circuits to model computation in *subclasses* of P. Allowing powerful computation to be incorporated into the step of *building* $C_{|x|}$ may overshadow the computation done by the circuit $C_{|x|}$ itself. The following much more stringent condition has proved to be most useful for characterizing these subclasses, and also works well for circuit classes at the level of polynomial time.

Definition 3.1. A circuit family $[C_n]$ is $\mathsf{DLOGTIME}$-*uniform* if there is a Turing machine M that can answer questions of the forms "Is there a path of edges from node u to node v in C_n?" and "What gate type does node u have?" in $O(\log n)$ time.

This uniformity condition is sufficient to build an encoding of C_n in sequential time roughly proportional to the size of C_n, and even much faster in parallel time. We will not try to define $\mathsf{DLOGTIME}$ as a complexity class, but note that since the inputs u, v to M can be presented by strings of length $O(\log n)$, the computation by M takes linear time in the (scaled down) input length. This definition presupposes that the size complexity $z(n)$ of the family is polynomial, which will be our chief interest here. The definition can be modified for $z(n)$ more than polynomial by changing the time limit on M to $O(\log z(n))$. Many central results originally proved using L-uniformity extend without change to $\mathsf{DLOGTIME}$-uniformity, as explained later in this section. Unless otherwise stated, "uniform" means $\mathsf{DLOGTIME}$-uniform throughout this and the next two chapters. We define the following circuit complexity classes:

Definition 3.2. Given complexity functions $z(n)$ and $d(n)$,

- $\mathsf{SIZE}[z(n)]$ is the class of all languages accepted by $\mathsf{DLOGTIME}$-uniform bounded fan-in circuit families whose size complexity is at most $z(n)$;
- $\mathsf{DEPTH}[d(n)]$ is the class of all languages accepted by $\mathsf{DLOGTIME}$-uniform bounded fan-in circuit families whose depth complexity is at most $d(n)$;
- $\mathsf{SIZE,DEPTH}[z(n), d(n)]$ is the class of all languages accepted by $\mathsf{DLOGTIME}$-uniform bounded fan-in circuit families whose size complexity is at most $z(n)$ and whose depth complexity is at most $d(n)$.

Non-uniform circuit classes can be approached by an alternative view introduced by [Karp and Lipton, 1982], by counting the number of bits of information needed to set up the preprocessing. For integer-valued functions t, a, define $\mathsf{DTIME}[t(n)]/\mathsf{ADV}[a(n)]$ to be the class of languages accepted by Turing machines M as follows: for all n there is a word y_n of length at most $a(n)$ such that for all x of length n, on input (x, y_n), M accepts if and only if $x \in L$, and M halts within $t(n)$ steps. Here y_n is regarded as "advice" on how to accept strings of length n. The class $\mathsf{DTIME}[n^{O(1)}]/\mathsf{ADV}[n^{O(1)}]$ is called $\mathsf{P/poly}$. Karp and Lipton observed that $\mathsf{P/poly}$ is equal to the class of languages accepted by polynomial-sized circuits. Indeed, $\mathsf{P/poly}$ is now the standard name for this class.

3.3 Circuits and Sequential Classes

The importance of $\mathsf{P/poly}$ and uniformity is shown by the following basic theorem. We give the proof since it is used often in the next chapter.

Theorem 3.1 *Every language in* P *is accepted by a family of polynomial-sized circuits that is* $\mathsf{DLOGTIME}$*-uniform. Conversely, every language with* P*-uniform polynomial-sized circuits belongs to* P.

Proof. Let $A \in \mathsf{P}$. By Theorem 2.4 of Chapter 24, A is accepted by a Turing machine M with just one tape and tape head that runs in polynomial time $p(n)$. Let δ be the transition function of M, whereby for all states q of M and characters c in the worktape alphabet Γ of M, $\delta(q, c)$ specifies the character written to the current cell, the movement of the head, and the next state of M. We build a circuit of "δ-gates" that simulates M on inputs x of a given length n as follows, and then show how to simulate δ-gates by Boolean gates.

Lay out a $p(n) \times p(n)$ array of cells. Each cell (i, j) $(0 \leq i, j \leq p(n))$ is intended to hold the character on tape cell j after step i of the computation of M, and if the tape head of M is in that cell, also the state of M after step i. Cells $(0, 0)$ through $(0, n{-}1)$ are the input nodes of C_n, while cells $(0, n)$ through $(0, p(n))$ can be treated as "dummy wires" whose value is the blank B in the alphabet Γ. The key idea is that the value in cell (i, j) for $i \geq 1$ depends only on the values in cells $(i{-}1, j{-}1)$, $(i{-}1, j)$, and $(i{-}1, j{+}1)$. Cell $(i{-}1, j{-}1)$ is relevant in case its value includes the component for the tape head being there, and the head moves right at step i; cell $(i{-}1, j{+}1)$ similarly for a left move.

When the boundary cases $j = 0$ or $j = p(n)$ are handled properly, each cell value is computed by the same finite function of the three cells above, and this function defines a "δ-gate" for each cell. (See Figure 1.) Finally, we may suppose that M is coded to signal acceptance by moving its tape head to the left end and staying in a special state q_a. Thus node $(i, 0)$ becomes the output gate of the circuit, and the accepting output values are those with q_a in the state component. Since in $p(n)$ steps M can visit at most $p(n)$ tape cells, the array is large enough to hold all the computations of M on inputs of length n.

Since each argument and value of a δ-gate comes from a finite domain, we may take an (arbitrary) binary encoding of the domain, and replace all δ-gates by identical fixed-size sub-circuits of Boolean gates that compute δ under the encoding. If the alphabet Σ over which A is defined is $\{0, 1\}$ then no re-coding need be done at the inputs; otherwise, we similarly adopt a binary encoding of Σ. The Boolean circuits C_n thus obtained accept A. They also are $\mathsf{DLOGTIME}$-uniform, intuitively by the very regular structure of the identical δ-gates.

Conversely, given a P-uniform family $\mathbf{C}$, a Turing machine can accept $L(\mathbf{C})$ in polynomial time given any input x by first constructing $C_{|x|}$ in polynomial time, and then evaluating $C_{|x|}(x)$. $\square$

A caching strategy that works for Turing machines with any fixed number of tapes yields the following improvement:

Theorem 3.2 *If $t(n)$ is a time-constructible function, then* $\mathsf{DTIME}(t) \subseteq \mathsf{SIZE}(t \log t)$.

Connections between space complexity and circuit depth are shown by the next result.

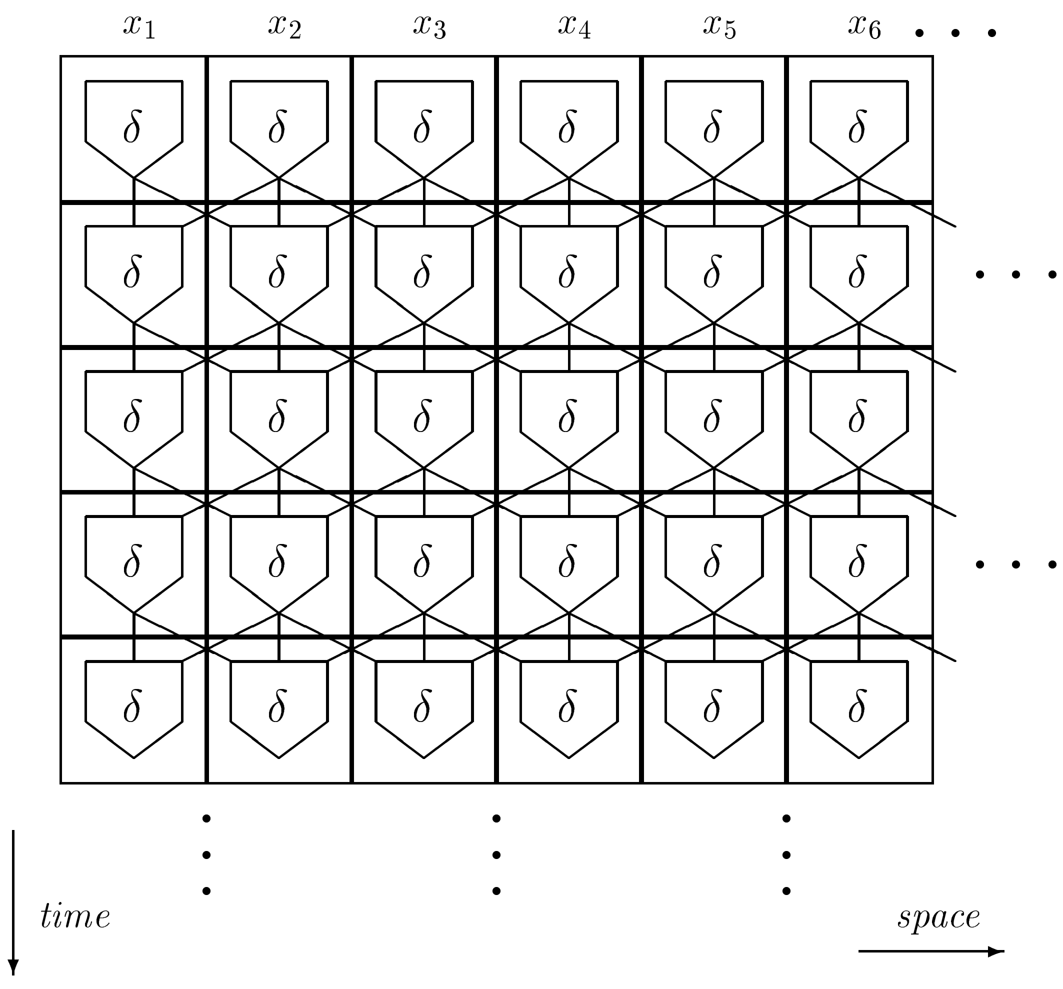

Figure 1: Conversion from Turing machine to Boolean circuits

Theorem 3.3 *(a) If $d(n) \geq \log n$, then* $\mathsf{DEPTH}[d(n)] \subseteq \mathsf{DSPACE}[d(n)]$.

(b) If $s(n)$ is a space-constructible function and $s(n) \geq \log n$, then $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DEPTH}[s(n)^2]$.

3.4 Circuits and Parallel Classes

Since the 1970s, research on circuit complexity has focused on problems that can be solved quickly in parallel, with feasible amounts of hardware—circuit families of polynomial size and depth as small as possible. Note, however, that the meaning of the phrase "as small as possible" depends on the technology used. With unbounded fan-in gates, depth $O(1)$ is sufficient to carry out interesting computation, whereas with fan-in two gates, depth less than $\log n$ is impossible if the value at the output gate depends on all of the input bits. In any technology, however, a circuit with depth nearly logarithmic is considered to be very fast. This observation motivates the following definitions. Let $\log^k n$ stand for $(\log n)^k$.

Definition 3.3. For all $k \geq 0$,

(a) NC^k denotes the class of languages accepted by DLOGTIME-uniform bounded fan-in circuit families of polynomial size and $O(\log^k n)$ depth. In other words, $\mathsf{NC}^k = \mathsf{SIZE,DEPTH}[n^{O(1)}, O(\log^k n)]$. NC denotes $\cup_{k\geq 0}\mathsf{NC}^k$.

(b) AC^k denotes the class of languages accepted by DLOGTIME-uniform families of circuits of unbounded fan-in $\wedge$, $\vee$, and $\neg$ gates, again with polynomial size and $O(\log^k n)$ depth.

(c) TC^k denotes the class of languages accepted by DLOGTIME-uniform families of circuits of MAJORITY and $\neg$ gates, again with polynomial size and $O(\log^k n)$ depth.

The case $k = 0$ in these definitions gives constant-depth circuit families. A function f is said to belong to one of these classes if the language $A_f = \{ \langle x, i, b\rangle : 1 \leq i \leq |f(x)| \text{ and bit } i \text{ of } f(x) \text{ is } b\}$ belongs to the class. NC^0 is not studied as a language class in general, since the output gate can depend on only a constant number of input bits, but NC^0 is interesting as a function class.

Some notes on the nomenclature are in order. Nicholas Pippenger was one of the first to study polynomial-size, polylog-depth circuits in the late 1970s, and NC was dubbed "Nick's Class." There is no connotation of nondeterminism in NC. The "A" in AC^k connotes both alternating circuits and alternating Turing machines for reasons described below. The "T" in TC^k stands for the presence of threshold gates.

The following theorem expresses the relationships at each level of the hierarchies defined by these classes.

Theorem 3.4 *For each $k \geq 0$,*

$$\mathsf{NC}^k \subseteq \mathsf{AC}^k \subseteq \mathsf{TC}^k \subseteq \mathsf{NC}^{k+1}.$$

Proof. The first inclusion is immediate (for each k), and the second conclusion follows from the observation noted above that $\mathsf{MAJORITY}$ gates can simulate unbounded fan-in AND and OR gates. The interesting case is $\mathsf{TC}^k \subseteq \mathsf{NC}^{k+1}$. For this, it suffices to show how to simulate a single $\mathsf{MAJORITY}$ gate with a fan-in two circuit of logarithmic depth. To simulate $\mathsf{MAJORITY}(w_1, \ldots, w_r)$, we add up the one-bit numbers $w_1, \ldots, w_r$ and test whether the sum is at least $r/2$. We may suppose for simplicity that the fan-in r is a power of 2, $r = 2^m$. The circuit has m distinguished nodes that represent the sum written as an m-bit binary number. Then the sum is at least $r/2 = 2^{m-1}$ if and only if the node representing the most significant bit of the sum has value 1.

To compute the sum efficiently, we use a standard "carry-save" technique: There is a simple $O(1)$ depth fan-in two circuit that takes as input three b-bit binary numbers a_1, a_2, a_3 and produces as output two (b+1)-bit numbers b_1, b_2 such that $a_1 + a_2 + a_3 = b_1 + b_2$. Thus in one phase, the original sum of r bits is reduced to taking the sum of $\frac{2}{3}r$ numbers, and after $O(\log r)$ additional phases, the problem is reduced to taking the sum of two $\log r$-bit numbers, and this sum can be produced by a full carry-lookahead adder circuit of $O(\log r)$ depth. Finally, since the circuits have polynomial size, r is polynomial in n, and so $O(\log r) = O(\log n)$. $\Box$

Thus in particular, $\cup_k \mathsf{AC}^k = \cup_k \mathsf{TC}^k = \mathsf{NC}$. The only proper inclusion known, besides the trivial case $\mathsf{NC}^0 \subset \mathsf{AC}^0$, is $\mathsf{AC}^0 \subset \mathsf{TC}^0$, discussed in Section 3.5 below. For all we know at this time, TC^0 may be equal not only to NC, but even to NP!

Several relationships between complexity classes based on circuits and classes based on Turing machines are known:

Theorem 3.5 $\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{AC}^1$.

In fact, the connection with Turing machines is much closer than this theorem suggests. Using *alternating* Turing machines (see Section 2.8 above), we define the following complexity classes:

- $\mathsf{ASPACE,TIME}[s(n), t(n)]$ is the class of languages recognized by alternating Turing machines that use space at most $s(n)$ and also run in time at most $t(n)$.
- $\mathsf{ASPACE,ALTS}[s(n), a(n)]$ is the class of languages recognized by alternating Turing machines that use space at most $s(n)$ and make at most $a(n)$ alternations between existential and universal states.
- $\mathsf{ATIME,ALTS}[s(n), a(n)]$ is the class of languages recognized by alternating Turing machines that run in time $t(n)$ and make at most $a(n)$ alternations between existential and universal states.

Theorem 3.6 *(a) For all $k \geq 1$, $\mathsf{NC}^k = \mathsf{ASPACE,TIME}[O(\log n), O(\log^k n)]$.*

(b) For all $k \geq 1$, $\mathsf{AC}^k = \mathsf{ASPACE,ALTS}[O(\log n), O(\log^k n)]$.

(c) $\mathsf{NC}^1 = \mathsf{ATIME}[O(\log n)]$.

(d) $\mathsf{AC}^0 = \mathsf{ATIME,ALTS}[O(\log n), O(1)]$.

For AC^1 and the higher circuit classes, changing the uniformity condition to L-uniformity does not change the class of languages. However, it is not known whether L-uniform NC^1 differs from NC^1, nor L-uniform AC^0 from AC^0. Thus the natural extension (c,d) of the results in (a,b) is another advantage of $\mathsf{DLOGTIME}$-uniformity. Insofar as the containment of NC^1 in L is believed to be proper by many researchers, the definition of L-uniform NC^1 may allow more computing power to the "preprocessing stage" than to the circuits themselves. Avoiding this anomaly is a reason to adopt $\mathsf{DLOGTIME}$-uniformity.

As discussed in Chapter 45, many other models of parallel computation can be used to define NC. This robustness of NC supports the belief that NC is not merely an artifact of some arbitrary choices made in formulating the definitions, but instead captures a fundamental aspect of parallel computation. The criticism has been made that NC is overly generous in allowing polynomial size. Again, the justification in complexity theory is that the ultimate goal is to prove lower bounds, and a lower bound proved against a generous upper-bound notion is impervious to this criticism.

3.5 Why Focus on These Circuit Classes?

The class AC^0 is particularly important for the following reasons:

- It captures the complexity of important basic operations such as integer addition and subtraction.
- It corresponds closely to first-order logic, as described in Chapter 29, Section 4.
- Most important, it is one of the few complexity classes for which lower bounds are actually known, instead of merely being conjectured.

It is known that AC^0 circuits, even non-uniform ones, cannot recognize the language PARITY of strings that have an odd number of 1's. Consequently, constant depth unbounded fan-in AND/OR/NOT circuits for PARITY must have super-polynomial size. However, PARITY does have constant-depth polynomial-size threshold circuits; indeed, it belongs to TC^0.

Note that this also implies that AC^0 is somehow "finer" than the notion of constant space, because the class of regular languages, which includes PARITY, can be decided in constant space. There has been much progress on proving lower bounds for classes of constant-depth circuits. Still, the fact that TC^0 is not known to differ from NP is a wide gulf in our knowledge. Separating NC from P, or L from P, or L from NP would imply separating TC^0 from NP.

TC^0 is important because it captures the complexity of important basic operations such as integer multiplication and sorting. Further, integer division is known to be in P-uniform TC^0, and many suspect that $\mathsf{DLOGTIME}$-uniformity would also be sufficient. Also, TC^0 is a good complexity-theoretic counterpart to popular models of neural networks.

NC^1 is important because it captures the complexity of the basic operation of evaluating a Boolean formula on a given assignment. The problem of whether NC^1 equals TC^0 thus captures the question of whether basic calculations in logic are harder than basic operations in arithmetic, or harder than basic neural processes. Several other characterizations of NC^1 besides the one given for $\mathsf{ATIME}[O(\log n)]$ are known. NC^1 equals the class of languages definable by polynomial-size Boolean *formulas* (as opposed to polynomial-sized *circuits*; a formula is equivalent to a circuit of fan-out 1). Also, NC^1 equals the class of languages recognized by bounded-width *branching programs*, for which see [Barrington, 1989]. Finally, NC^1 captures the circuit complexity of regular expressions.

4 Research Issues and Summary

The complexity class is the fundamental notion of complexity theory. What makes a complexity class useful to the practitioner is the close relationship between complexity classes and real computational problems. The strongest such relationship comes from the concept of completeness, which is a chief subject of the next chapter. Even in the absence of lower bounds separating complexity classes, the apparent fundamental difference between models such as deterministic and nondeterministic Turing machines, for example, provides insight into the nature of problem solving on computers.

The initial goal when trying to solve a computational problem is to find an efficient polynomial-time algorithm. If this attempt fails, then one could attempt to prove that no efficient algorithm exists, but to date nobody has succeeded doing this for any problem in PSPACE. With the notion of a complexity class to guide us, however, we can attempt to discover the complexity class that exactly captures our current problem. A main theme of the next chapter is the surprising fact that most natural computational problems are *complete* for one of the canonical complexity classes. When viewed in the abstract setting provided by the model that defines the complexity class, the aspects of a problem that make an efficient algorithm difficult to achieve are easier to identify. Often this perspective leads to a redefinition of the problem in a way that is more amenable to solution.

Figure 2 shows the known inclusion relationships between canonical classes. Perhaps even more significant is what is currently not known. Although AC^0 differs from TC^0, TC^0 (let alone P!) is not known to differ from NP, nor NP from EXP, nor EXP from $\mathsf{EXPSPACE}$. The only other proper inclusions known are (immediate consequences of) $\mathsf{L} \neq \mathsf{PSPACE} \neq \mathsf{EXPSPACE}$, $\mathsf{P} \neq \mathsf{E} \neq \mathsf{EXP}$, and $\mathsf{NP} \neq \mathsf{NE} \neq \mathsf{NEXP}$—and these follow simply from the hierarchy theorems proved in this chapter.

We have given two examples of diagonalization arguments. Diagonalization is still the main tool for showing the existence of hard-to-compute problems inside a complexity class. Unfortunately, the languages constructed by diagonalization arguments rarely correspond to computational problems that arise in practice. In some cases, however, one can show that there is an efficient reduction from a difficult problem (shown to exist by diagonalization) to a more natural problem—with the consequence that the natural problem is also difficult to solve. Thus diagonalization inside a complexity class (the topic of this chapter) can work hand-in-hand with reducibility (the topic of the next chapter) to produce intractability results for natural computational problems.

5 Defining Terms

Canonical complexity classes: The classes defined by logarithmic, polynomial, and exponential bounds on time and space, for deterministic and nondeterministic machines. These are the most central to the field, and classify most of the important computational problems.

Circuit A network of input, output, and logic gates, contrasted with a **Turing machine** in that its hardware is static and fixed.

Circuit complexity The study of the size, depth, and other attributes of circuits that decide specified languages or compute specified functions.

Diagonalization: A proof technique for showing that a given language does not belong to a given complexity class, used in many **separation theorems**.

Padding argument: A method for transferring results about one complexity bound to another complexity bound, by padding extra dummy characters onto the inputs of the machines involved.

Polynomial-Time Church-Turing Thesis: An analogue of the classical **Church-Turing Thesis**, for which see Chapter 26, stating that the class P captures the true notion of feasible (polynomial time) sequential computation.

Separation theorems: Theorems showing that two complexity classes are distinct. Most known separation theorems have been proved by **diagonalization**.

Simulation theorems: Theorems showing that one kind of computation can be simulated by another kind within stated complexity bounds. Most known containment or equality relationships between complexity classes have been proved this way.

Space-constructible function: A function $s(n)$ that gives the actual space used by some Turing machine on all inputs of length n, for all n

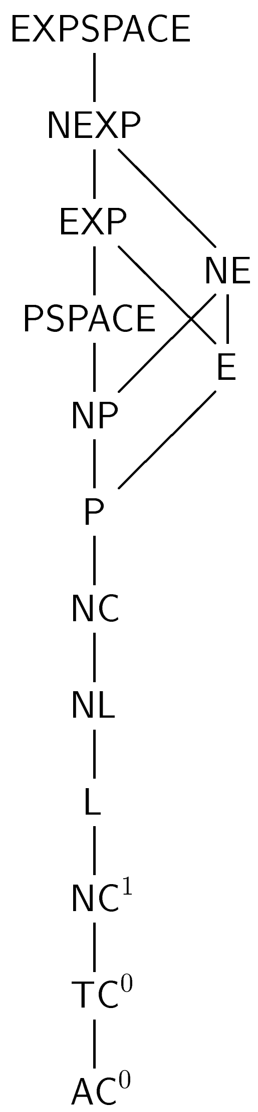

Figure 2: Inclusion relationships between the canonical complexity classes.

Time-constructible function: A function $t(n)$ that is the actual running time of some Turing machine on all inputs of length n, for all n.

Uniform circuit family: A sequence of circuits, one for each input length n, that can be efficiently generated by a Turing machine.

Uniform circuit complexity The study of complexity classes defined by uniform circuit families.

References

[Ajtai, 1983] M. Ajtai. Σ_1^1 formulae on finite structures. *Annals of Pure and Applied Logic*, 24:1–48, 1983.

[Barrington *et al.*, 1990] D. Mix Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *J. Comp. Sys. Sci.*, 41:274–306, 1990.

[Barrington, 1989] D. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *J. Comp. Sys. Sci.*, 38:150–164, 1989.

[Berthiaume, 1997] A. Berthiaume. Quantum computation. In L. Hemaspaandra and A. Selman, editors, *Complexity Theory Retrospective II*, pages 23–51. Springer Verlag, 1997.

[Blum, 1967] M. Blum. A machine-independent theory of the complexity of recursive functions. *J. Assn. Comp. Mach.*, 14:322–336, 1967.

[Book *et al.*, 1970] R. Book, S. Greibach, and B. Wegbreit. Time- and tape-bounded turing acceptors and afls. *J. Comp. Sys. Sci.*, 4:606–621, 1970.

[Book, 1974] R. Book. Comparing complexity classes. *J. Comp. Sys. Sci.*, 9:213–229, 1974.

[Boppana and Sipser, 1990] R. 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 and MIT Press, 1990.

[Borodin, 1972] A. Borodin. Computational complexity and the existence of complexity gaps. *J. Assn. Comp. Mach.*, 19:158–174, 1972.

[Borodin, 1977] A. Borodin. On relating time and space to size and depth. *SIAM J. Comput.*, 6:733–744, 1977.

[Chandra *et al.*, 1981] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J. Assn. Comp. Mach.*, 28:114–133, 1981.

[Chandra *et al.*, 1984] A. Chandra, L. Stockmeyer, and U. Vishkin. Constant-depth reducibility. *SIAM J. Comput.*, 13:423–439, 1984.

[Cook, 1985] S. Cook. A taxonomy of problems with fast parallel algorithms. *Inform. and Control*, 64:2–22, 1985.

[Furst *et al.*, 1984] M. Furst, J. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Math. Sys. Thy.*, 17:13–27, 1984.

[Garey and Johnson, 1988] M. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1988. First edition was 1979.

[Gurevich, 1991] Y. Gurevich. Average case completeness. *J. Comp. Sys. Sci.*, 42:346–398, 1991.

[Hartmanis and Stearns, 1965] J. Hartmanis and R. Stearns. On the computational complexity of algorithms. *Transactions of the AMS*, 117:285–306, 1965.

[Håstad, 1989] J. Håstad. Almost optimal lower bounds for small-depth circuits. In S. Micali, editor, *Randomness and Computation*, volume 5 of *Advances in Computing Research*, pages 143–170. JAI Press, Greenwich, CT, USA, 1989.

[Hofmeister, 1996] T. Hofmeister. A note on the simulation of exponential threshold weights. In *Proc. 2nd International Computing and Combinatorics Conference (COCOON'96)*, volume 1090 of *Lect. Notes in Comp. Sci.*, pages 136–141. Springer Verlag, 1996.

[Hoover *et al.*, 1984] H. Hoover, M. Klawe, and N. Pippenger. Bounding fan-out in logical networks. *J. Assn. Comp. Mach.*, 31:13–18, 1984.

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

[Ibarra, 1972] O. Ibarra. A note concerning nondeterministic tape complexities. *J. Assn. Comp. Mach.*, 19:608–612, 1972.

[Immerman and Landau, 1995] N. Immerman and S. Landau. The complexity of iterated multiplication. *Inform. and Control*, 116:103–116, 1995.

[Immerman, 1988] N. Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17:935–938, 1988.

[Impagliazzo, 1995] R. Impagliazzo. A personal view of average-case complexity. In *Proc. 10th Annual IEEE Conference on Structure in Complexity Theory*, pages 134–147, 1995.

[Johnson, 1990] D.S. Johnson. A catalog of complexity classes. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 67–161. Elsevier and MIT Press, 1990.

[Karmarkar, 1984] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[Karp and Lipton, 1982] R. Karp and R. Lipton. Turing machines that take advice. *L'Enseignement Mathématique*, 28:191–210, 1982.

[Khachiyan, 1979] L. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20(1):191–194, 1979. English translation.

[Kurtz *et al.*, 1997] S. Kurtz, S. Mahaney, J. Royer, and J. Simon. Biological computing. In L. Hemaspaandra and A. Selman, editors, *Complexity Theory Retrospective II*, pages 179–195. Springer Verlag, 1997.

[Levin, 1996] Leonid A. Levin. Computational complexity of functions. *Theor. Comp. Sci.*, 157(2):267–271, 1996.

[Lewis II *et al.*, 1965] P. Lewis II, R. Stearns, and J. Hartmanis. Memory bounds for recognition of context-free and context-sensitive languages. In *Proceedings, Sixth Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 191–202, 1965.

[Papadimitriou, 1994] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Mass., 1994.

[Parberry, 1994] I. Parberry. *Circuit Complexity and Neural Networks.* M.I.T. Press, Cambridge, Mass., 1994.

[Pippenger and Fischer, 1979] N. Pippenger and M. Fischer. Relations among complexity measures. *J. Assn. Comp. Mach.*, 26:361–381, 1979.

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

[Ruby and Fischer, 1965] S. Ruby and P. Fischer. Translational methods and computational complexity. In *Proceedings, Sixth Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 173–178, 1965.

[Ruzzo, 1981] W. Ruzzo. On uniform circuit complexity. *J. Comp. Sys. Sci.*, 22:365–383, 1981.

[Savitch, 1970] W. Savitch. Relationship between nondeterministic and deterministic tape complexities. *J. Comp. Sys. Sci.*, 4:177–192, 1970.

[Seiferas *et al.*, 1978] J. Seiferas, M. Fischer, and A. Meyer. Separating nondeterministic time complexity classes. *J. Assn. Comp. Mach.*, 25:146–167, 1978.

[Sipser, 1983] M. Sipser. Borel sets and circuit complexity. In *Proc. 15th Annual ACM Symposium on the Theory of Computing*, pages 61–69, 1983.

[Stockmeyer, 1974] L. Stockmeyer. The complexity of decision problems in automata theory and logic. Technical Report MAC-TR-133, Project MAC, M.I.T., Cambridge, Mass., 1974.

[Stockmeyer, 1987] L. Stockmeyer. Classifying the computational complexity of problems. *J. Symb. Logic*, 52:1–43, 1987.

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

[Trakhtenbrot, 1964] B. Trakhtenbrot. Turing computations with logarithmic delay. *Algebra i Logika*, 3:33–48, 1964.

[van Emde Boas, 1990] P. van Emde Boas. Machine models and simulations. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 1–66. Elsevier and MIT Press, 1990.

[von zur Gathen, 1991] J. von zur Gathen. Efficient exponentiation in finite fields. In *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 384–391, 1991.

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

[Zak, 1983] S. Zak. A Turing machine time hierarchy. *Theor. Comp. Sci.*, 26:327–333, 1983.

Further Information

Primary sources for the results presented in this chapter are: Theorem 2.1 [Trakhtenbrot, 1964, Borodin, 1972]; Theorem 2.2 [Blum, 1967]; Theorems 2.3 and 2.4 [Hartmanis and Stearns, 1965, Lewis II *et al.*, 1965, Savitch, 1970, Immerman, 1988, Szelepcsényi, 1988]; Theorem 2.5 [Hartmanis and Stearns, 1965, Ibarra, 1972, Seiferas *et al.*, 1978]; Theorem 2.6 [Book, 1974]; Lemma 2.7 [Ruby and Fischer, 1965]; Theorems 2.9 and 2.10 [Chandra *et al.*, 1981]; Theorem 3.1 [Savitch, 1970]; Theorem 3.2 [Pippenger and Fischer, 1979]; Theorem 3.3 [Borodin, 1977]; Theorem 3.6 [Ruzzo, 1981, Chandra *et al.*, 1984, Sipser, 1983, Barrington *et al.*, 1990]. Theorems 3.4 and 3.5 are a combination of results in the last four papers; see also the influential survey by Cook [Cook, 1985]. Our proof of Theorem 2.5(b) follows [Zak, 1983].

For Section 3.1, a comparison of arithmetical circuits with Boolean circuits may be found in [von zur Gathen, 1991], the result that bounded fan-in circuits can be given bounded fan-out is due to [Hoover *et al.*, 1984], and the sharpest simulation of general weighted threshold gates by MAJORITY gates is due to [Hofmeister, 1996]. The theorem in Section 3.5 that PARITY is not in AC^0 is due to [Furst *et al.*, 1984, Ajtai, 1983], and the strongest lower bounds known on the size of constant-depth circuits for Parity are those in [Håstad, 1989]. The results mentioned for TC^0 may be found in [Barrington *et al.*, 1990, Reif and Tate, 1992, Immerman and Landau, 1995].

The texts [Hopcroft and Ullman, 1979] and [Papadimitriou, 1994] present many of these results in greater technical detail. Three chapters of the *Handbook of Theoretical Computer Science*, respectively [Johnson, 1990], [van Emde Boas, 1990], and [Boppana and Sipser, 1990], describe more complexity classes, compare complexity measures for more machine models, and present more information on circuit complexity. Relationships between circuits and parallel and neural models are covered very accessibly in [Parberry, 1994]. Average-case complexity is discussed by [Wang, 1997, Impagliazzo, 1995, Gurevich, 1991]. See also Chapter 29 and the notes at the end of that chapter for further sources.

Two important new research areas that challenge our arguments about feasible computation in Section 2.2 are *quantum computing* and *DNA computing*. Two new survey articles on these fields are [Berthiaume, 1997] and [Kurtz *et al.*, 1997].