July 1995

UILU-ENG-95-2221
ACT-134

Applied Computation Theory

Complexity Theory

Michael C. Loui

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Complexity Theory

Michael C. Loui[1]

Department of Electrical and Computer Engineering, and
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

July 7, 1995

Revised November 4, 1995

Abstract

This manuscript presents the rudiments of computational complexity theory. The manuscript will appear as a chapter in the forthcoming *CRC Handbook of Computer Science and Engineering*, and it refers to other chapters and sections in the *Handbook*.

Key words: computational complexity theory, Turing machines, complexity classes, reducibility, NP-completeness.

[1]Supported by the National Science Foundation under grant CCR-9315696.

1 Introduction

Computational complexity is the study of the difficulty of solving computational problems, in terms of the required computational resources, such as time and space (memory). The formal theoretical study of computational complexity began with the paper of Hartmanis and Stearns [1965], who introduced the basic concepts and proved the first results. For their achievement, Hartmanis and Stearns received the 1993 Turing Award of the ACM.

Whereas the analysis of algorithms focuses on the time or space required by an algorithm to solve a specific computational problem, such as sorting, complexity theory focuses on the **complexity class**, which consists of all problems solvable within the same amount of time or space. Two important complexity classes are P, the set of problems that can be solved in polynomial time, and NP, the set of problems whose solutions can be verified in polynomial time. Complexity theorists have discovered that most common computational problems fall into a small number of complexity classes.

By quantifying the resources required to solve a problem, complexity theory has profoundly affected our thinking about computation. Computability theory establishes the existence of undecidable problems, which cannot be solved in principle, regardless of the amount of time invested. In contrast, complexity theory establishes the existence of decidable problems that, although solvable in principle, cannot be solved in practice, because the time and space required would be larger than the age and size of the known universe [Stockmeyer and Chandra, 1979]. Thus, complexity theory characterizes the computationally feasible problems.

The quest for the boundaries of the set of feasible problems has led to the most important unsolved question in all of computer science: Is P different from NP? Hundreds of fundamental problems, including many ubiquitous optimization problems of operations research, are NP-complete—they are the hardest problems in NP. If someone could find a polynomial-time algorithm for any one NP-complete problem, then there would be polynomial-time algorithms for all of them. Despite the concerted efforts of many scientists over several decades, no polynomial-time algorithm has been found for any NP-complete problem. Although we do not yet know whether P is different from NP, showing that a problem is NP-complete provides strong evidence that the problem is computationally infeasible, and justifies the use of heuristics for solving the problem.

In this chapter, we define P, NP, and related complexity classes. We illustrate the use of diagonalization and padding techniques to prove relationships between classes. Next, we define NP-completeness, and we show how to prove that a problem is NP-complete. Finally, we define complexity classes for probabilistic and interactive computations.

Throughout this chapter, all numeric functions take integer arguments and produce integer values. All logarithms are taken to base 2. In particular, $\log n$ means $\lceil \log_2 n \rceil$.

2 Models of Computation

> By a theory, I mean the result of removing ambiguity and uncertainty in the statement of a problem so that precise, rigorous statements about it can be made and verified. Abstraction, the process of eliminating unnecessary detail, ... permits the

problem to be reduced to a tractable and comprehensible size from which significant new insights can be obtained.

—*M. J. Fischer* [1980]

To develop a theory of the difficulty of computational problems, we need to specify precisely what a problem is, what an algorithm is, and what a measure of difficulty is. For simplicity, complexity theorists have chosen to represent problems as languages, to model algorithms by off-line multitape Turing machines, and to measure computational difficulty by the time and space required by a Turing machine. To justify these choices, some theorems of complexity theory show how to translate statements about, say, the time complexity of language recognition by Turing machines into statements about computational problems on more realistic models of computation. These theorems imply that the principles of complexity theory are not artifacts of Turing machines, but intrinsic properties of computation.

This section defines different kinds of Turing machines. The deterministic Turing machine models actual computers. The nondeterministic Turing machine is not a realistic model, but it helps classify the complexity of important computational problems. The alternating Turing machine models a form of parallel computation, and it helps elucidate the relationship between time and space.

2.1 Computational Problems and Languages

Computer scientists have invented many elegant formalisms for representing data and control structures. Fundamentally, all representations are patterns of symbols. Therefore, we represent an instance of a computational problem as a sequence of symbols. (See Chapter 4, on formal models and computability.)

Let Σ be a finite set, called the **alphabet.** A **word** over Σ is a finite sequence of symbols from Σ. Sometimes a word is called a "string." Let Σ^* denote the set of all words over Σ. For example, if $\Sigma = \{0, 1\}$, then

$$\Sigma^* = \{\lambda, 0, 1, 00, 01, 10, 11, 000, \ldots\}$$

is the set of all binary words, including the empty word λ. (Binary representations of numbers and data, such as ASCII, are pervasive in computing.) The **length** of a word w, denoted $|w|$, is the number of symbols in w. A **language** over Σ is a subset of Σ^*.

A **decision problem** is a computational problem whose answer is simply "yes" or "no." For example: "Is the input graph connected?" "Is the input a sorted list of integers?" A decision problem can be expressed as a membership problem for a language L: for an input x, does x belong to L? For a language L that represents connected graphs, the input word x might represent an input graph G, and $x \in L$ if and only if G is connected.

For every decision problem, the representation should allow for easy parsing, to determine whether a word represents a legitimate instance of the problem. Furthermore, the representation should be concise. In particular, it would be unfair to encode the answer to the problem into the representation of an instance of the problem; for example, for the problem of deciding whether an input graph is connected, the representation should not have an extra bit that tells whether the graph is connected. A set of integers $S = \{x_1, \ldots, x_m\}$ is represented by listing the binary representation of each x_i, with the representations of

consecutive integers in S separated by a nonbinary symbol. A graph is naturally represented by giving either its adjacency matrix or a set of adjacency lists, where the list for each vertex v specifies the vertices adjacent to v.

Whereas the solution to a decision problem is "yes" or "no," the solution to an optimization problem is more complicated—for example, "Determine the shortest path from vertex u to vertex v in an input graph G." Nevertheless, for every optimization (minimization) problem, with objective function g, there is a corresponding decision problem that asks whether there exists a feasible solution x such that $g(x) \leq k$, where k is a given target value. Clearly, if there is an algorithm that solves an optimization problem, then that algorithm can be used to solve the corresponding decision problem. Conversely, if algorithm A solves the decision problem, then with a binary search on the range of values of g, we can determine the optimal value. A fortiori, for many optimization problems, with multiple calls to the decision algorithm A, we can even construct an optimal solution. Therefore, there is little loss of generality in considering only decision problems, represented as language membership problems.

2.2 Turing machines

> Tape memory as an architectural feature of a modern computer is quaint, at best.
>
> — *W. L. Ruzzo* [1981]

This subsection and the next three give precise, formal definitions of Turing machines and their variants. These subsections are intended for reference. For the rest of this chapter, from Section 3 on, the reader need not understand these definitions in detail, but may generally substitute "program" or "computer" for each reference to "Turing machine."

A **k-tape Turing machine** M consists of the following:

- A finite set of states Q, with special states q_0 (initial state), q_A (accept state), and q_R (reject state).
- A finite alphabet Σ, and a special blank symbol $\square \notin \Sigma$.
- $k+1$ linear tapes, each divided into cells. Tape 0 is the input tape, and tapes $1, \ldots, k$ are the worktapes. Each tape is infinite to the left and to the right. Each cell holds a single symbol from $\Sigma \cup \{\square\}$. By convention, the input tape is read-only. Each tape has an access head, and at every instant, each access head scans one cell. See Figure 1.
- A finite transition table δ, which comprises tuples of the form

$$(q, s_0, s_1, \ldots, s_k, q', s'_1, \ldots, s'_k, d_0, d_1, \ldots, d_k)$$

  where $q, q' \in Q$, each $s_i, s'_i \in \Sigma \cup \{\square\}$, and each $d_i \in \{-1, 0, +1\}$.

  A tuple specifies a step of M: if the current state is q, and $s_0, s_1, \ldots, s_k$ are the symbols in the cells scanned by the access heads, then M replaces s_i by s'_i for $i = 1, \ldots, k$ simultaneously, change state to q', and move the head on tape i one cell to the left

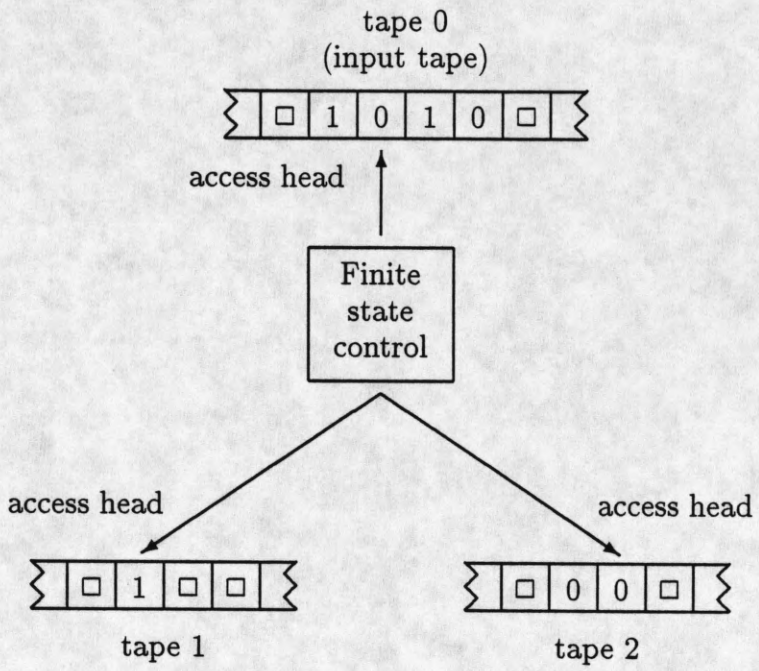

Figure 1: A 2-tape Turing machine.

$(d_i = -1)$ or right $(d_i = +1)$ or not at all $(d_i = 0)$ for $i = 0, \ldots, k$. Note that M cannot write on tape 0, that is, M may write only on the worktapes, not on the input tape.

- In a tuple, no s'_i can be the blank symbol $\square$. Since M may not write a blank, the worktape cells that its access heads previously visited are nonblank.
- No tuple contains q_A or q_R as its first component. Thus, once M enters state q_A or state q_R, it stops.
- Initially, M is in state q_0, an input word in Σ^* is inscribed on contiguous cells of the input tape, the access head on the input tape is on the leftmost symbol of the input word, and all other cells of all tapes contain the blank symbol $\square$.

The Turing machine M that we have defined is **nondeterministic**: δ may have several tuples with the same combination of state q and symbols $s_0, s_1, \ldots, s_k$ as the first $k + 2$ components, so that M may have several possible next steps. A machine M is **deterministic** if for every combination of state q and symbols $s_0, s_1, \ldots, s_k$, at most one tuple in δ contains the combination as its first $k + 2$ components. A deterministic machine always has at most one possible next step.

A **configuration** of a Turing machine M specifies a state, contents of all tapes, and positions of all access heads.

A **computation path** is a sequence of configurations $C_0, C_1, \ldots, C_t, \ldots$, where C_0 is the initial configuration of M, and each C_{j+1} follows from C_j in one step according to the changes specified by a tuple in δ. If no tuple is applicable to C_t, then C_t is **terminal**, and the computation path is **halting.** If M has no infinite computation paths, then M **always halts.** Without loss of generality, we may assume that in every terminal configuration, the state is q_A or q_R.

A halting computation path is **accepting** if the state in the last configuration C_t is q_A, **rejecting** if the state in C_t is q_R.

M **accepts** an input word x if there exists an accepting computation path that starts from the initial configuration in which x is on the input tape. M **rejects** x if the only halting computation paths are rejecting; a machine may reject x by failing to halt. If M is deterministic, then there is at most one halting computation path, hence at most one accepting path.

The **language accepted by** M, written $\mathcal{L}(M)$, is the set of words accepted by M. If $L = \mathcal{L}(M)$, and M always halts, then M **decides** L.

Generally, in this chapter, we consider only Turing machines that always halt. When we define complexity classes, we consider only time- and space-constructible functions (see Section 3.2 below) as bounds, and Turing machines that accept within those time or space bounds can be converted into machines that always halt.

In addition to deciding languages, deterministic Turing machines can compute functions. Designate tape 1 to be the **output tape.** If M halts on input word x, then the nonblank word on tape 1 in the final configuration is the output of M. A function f is **total recursive** if there exists a deterministic Turing machine M that always halts such that for each input word x, the output of M is the value of $f(x)$.

Almost all results in complexity theory are insensitive to minor variations in the underlying computational models. For example, we could have chosen Turing machines whose tapes are restricted to be only one-way infinite, or whose alphabet is restricted to $\{0, 1\}$. It is straightforward to simulate a Turing machine as defined above by one of these restricted Turing machines, one step at a time: each step of the original machine can be simulated by $O(1)$ steps of the restricted machine. The simulation of one Turing machine by a Turing machine with a different structure is analogous to the implementation of virtual machines in multilayer computer systems. (See Section IX of this *Handbook,* on operating systems and networks.)

2.3 Universal Turing Machines

A **k-tape universal Turing machine** is a Turing machine U with two input tapes and k worktapes that interprets the word i on one input tape as an encoding of a k-tape Turing machine M_i, and the word x on the other input tape as the input to M_i. Machine U simulates the operation of M_i on x, consulting the transition table encoded in i to determine each next step. (See Chapter 4, on formal models and computability, for a universal GOTO program.)

Each symbol in the alphabet of M_i is encoded by $O(1)$ symbols of the alphabet of U. Thus, to simulate the writing of a cell on a worktape of M_i, machine U writes on $O(1)$ of its cells on the corresponding worktape. Overall, provided that the encoding i is reasonable, U takes $O(|i|)$ steps to simulate each step of M_i.

We can think of U with a fixed i as a machine U_i, and define $\mathcal{L}(U_i) = \{x : U \text{ accepts } \langle i, x \rangle\}$. Then $\mathcal{L}(U_i) = \mathcal{L}(M_i)$.

2.4 Alternating Turing Machines

By definition, a nondeterministic Turing machine M accepts its input word x if there exists an accepting computation path, starting from the initial configuration with x on the input tape. Let us call a configuration C **accepting** if there is a computation path of M that starts in C and ends in a configuration whose state is q_A. Equivalently, a configuration C is accepting if either the state in C is q_A, or there exists an accepting configuration C' reachable from C by one step of M. Then M accepts x if the initial configuration with input word x is accepting.

The **alternating Turing machine** generalizes this notion of acceptance. In an alternating Turing machine M, each state is labeled either existential or universal. (Do not confuse the universal state in an alternating Turing machine with the universal Turing machine.) A nonterminal configuration C is existential (respectively, universal) if the state in C is labeled existential (universal). A terminal configuration is accepting if its state is q_A. A nonterminal existential configuration C is accepting if there exists an accepting configuration C' reachable from C by one step of M. A nonterminal universal configuration C is accepting if for every configuration C' reachable from C by one step of M, the configuration C' is accepting. Finally, M accepts x if the initial configuration with input word x is an accepting configuration.

A nondeterministic Turing machine is a special case of an alternating Turing machine in which every state is existential.

The computation of an alternating Turing machine M alternates between existential states and universal states. Intuitively, from an existential configuration, M guesses a step that leads toward acceptance; from a universal configuration, M checks whether each possible next step leads toward acceptance—in a sense, M checks all possible choices in parallel. An alternating computation captures the essence of a two-player game: Player 1 has a winning strategy if there exists a move for Player 1 such that for every move by Player 2, there exists a subsequent move by Player 1, etc., such that Player 1 eventually wins.

2.5 Oracle Turing Machines

Some computational problems remain difficult even when solutions to instances of a particular, different decision problem are available for free. When we study the complexity of a problem **relative** to a language L, we assume that answers about membership in L have been precomputed and stored in a (possibly infinite) table, and that there is no cost to obtain an answer to a membership query "Is w in L?" The language L is called an **oracle**. Conceptually, an algorithm queries the oracle whether a word w is in L, and it receives the correct answer.

An **oracle Turing machine** is a Turing machine M with a special **oracle tape** and special states QUERY, YES, and NO. The computation of the oracle Turing machine M^L, with oracle language L, is the same as that of an ordinary Turing machine, except that when M enters the QUERY state with a word w on the oracle tape, in one step, M enters either

the YES state if $w \in L$, or the NO state if $w \notin L$. Furthermore, during this step, the oracle tape is erased, so that the time for setting up each query is accounted separately.

3 Resources and Complexity Classes

In this section, we define the measures of difficulty of solving computational problems: time and space. We introduce complexity classes, which enable us to classify problems according to the difficulty of their solution.

3.1 Time and Space

> PROFESSOR: But how did you know that, if you don't know the principles of arithmetical reasoning?
>
> PUPIL: It's easy. Not being able to rely on my reasoning, I've memorized all the products of all possible multiplications.
>
> PROFESSOR: That's pretty good.
>
> — *Eugene Ionesco, "The Lesson"*

We measure the difficulty of a computational problem by the running time and the space (memory) requirements of an algorithm that solves the problem. Clearly, in general, a finite algorithm cannot have a table of all answers to infinitely many instances of the problem, although an algorithm could look up precomputed answers to a finite number of instances; in terms of Turing machines, the finite answer table is built into the set of states and the transition table. For these instances, the running time is negligible—just the time needed to read the input word. Consequently, our complexity measure should consider a whole problem, not only specific instances.

We express the complexity of a problem in terms of the growth of the required time or space, as a function of the length n of the input word that encodes a problem instance. As in Chapter 5, on algorithm analysis, we consider the worst case complexity, that is, for each n, the maximum time or space required among all inputs of length n.

The **time** taken by a Turing machine M on input word x, denoted $\text{Time}_M(x)$, is defined as follows:

- If M accepts x, then $\text{Time}_M(x)$ is the number of steps in the shortest accepting computation path for x.
- If M rejects x, then $\text{Time}_M(x)$ is the number of steps in the longest computation path for x, or $+\infty$ if M has an infinite computation path for x.

For a deterministic machine M, for every input x, there is at most one halting computation path, and $\text{Time}_M(x)$ is unambiguous. For a nondeterministic machine M, if $x \in \mathcal{L}(M)$, then M can "guess" the correct steps to take toward an accepting configuration, and $\text{Time}_M(x)$ measures the length of the path on which M always makes the best guess.

The **space** used by a Turing machine M on input x, denoted $\text{Space}_M(x)$, is defined as follows. The space used by a halting computation path is the number of nonblank worktape cells in the last configuration; this is the number of different cells ever written by the worktape

heads of M during the computation path, since M never writes the blank symbol. Because the space occupied by the input word is not counted, a machine can use a sublinear ($o(n)$) amount of space.

- If M accepts x, then $\text{Space}_M(x)$ is the minimum space used among all accepting computation paths for x.
- If M rejects x, then $\text{Space}_M(x)$ is the maximum space used among all computation paths for x, possibly $+\infty$.

The **time complexity** of a machine M is the function

$$t(n) = \max\{\text{Time}_M(x) : |x| = n\}$$

We assume that M reads all of its input word, and the blank symbol after the right end of the input word, so $t(n) \geq n + 1$. The **space complexity** of M is the function

$$s(n) = \max\{\text{Space}_M(x) : |x| = n\}$$

Because few interesting languages can be decided by machines of sublogarithmic space complexity, we henceforth assume that $s(n) \geq \log n$.

A function $f(x)$ is **computable in polynomial time** if there exists a deterministic Turing machine M of polynomial time complexity such that for each input word x, the output of M is $f(x)$.

3.2 Constructibility

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 1-tape deterministic Turing machine that uses exactly $s(n)$ worktape cells for every input of length n.

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., $n \log n, n^3, 2^n, n!$. Essentially every natural function is $\Theta(s)$ for a space-constructible s.

The following theorem characterizes the time- and space-constructible functions.

Theorem 1 *Let $t(n)$ be a function such that for some constant $\epsilon > 0$, $t(n) \geq (1+\epsilon)n$ for all n sufficiently large. Then $t(n)$ is time-constructible if and only if there exists a deterministic Turing machine M_t of time complexity $O(t(n))$ such that on every input word of length n, machine M_t computes the unary representation of $t(n)$, i.e., the word consisting of $t(n)$ occurrences of the symbol* 1. [Kobayashi, 1985].

A function $s(n)$ is space-constructible if and only if there exists a deterministic Turing machine M_s of space complexity $O(s(n))$ such that on every input word of length n, machine M_s computes the unary representation of $s(n)$.

Suppose $t(n)$ is time-constructible by a deterministic Turing machine M_t. For every Turing machine M that may have infinite computation paths, we can convert M into a machine M' of time complexity $O(t(n))$ that always halts, as follows: first, make a copy of the input word on a separate worktape; then concurrently run M with M_t, and halt as soon as M_t halts. The total time taken is $2n + t(n) + O(1)$, which is $O(t(n))$, because $t(n) \geq n + 1$. In particular, for a deterministic (respectively, nondeterministic) Turing machine M that accepts each word in $\mathcal{L}(M)$ in $t(n)$ time, but may reject words by not halting, this construction produces a deterministic (nondeterministic) Turing machine M' of time complexity $O(t(n))$ that accepts the same language—i.e., $\mathcal{L}(M') = \mathcal{L}(M)$—but M' always halts.

Analogously, though by a different argument, if $s(n)$ is space-constructible and $s(n) \geq \log n$, then for every deterministic (respectively, nondeterministic) Turing machine M that accepts each word in $\mathcal{L}(M)$ in $s(n)$ space, there is a deterministic (nondeterministic) Turing machine M' of space complexity $O(s(n))$ that accepts the same language, but M' always halts. On input word x of length n, machine M' simulates one step of M at a time, counting the number of steps and the number of worktape cells used by M. If M accepts x, then M has an accepting computation path whose number of steps is at most the total number of distinct configurations of M that use $s(n)$ space, which is $O(nc^{s(n)})$ for some constant c. Thus, if the number of steps taken by M exceeds this bound, then this computation path is not accepting—machine M might be in an infinite loop—and M' halts in its rejecting state. If the number of worktape cells used by M exceeds $s(n)$, then M' rejects and halts. The space used by M' is $s(n)$ to simulate M, plus

$$\log(nc^{s(n)}) = O(\log n + s(n)) = O(s(n))$$

tape cells for its step counter, and $\log s(n)$ cells for its cell counter.

3.3 Complexity Classes

Having defined the time complexity and space complexity of individual Turing machines, we now define classes of languages with particular complexity bounds. These definitions will lead to definitions of P and NP.

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

- DTIME($t(n)$) is the class of languages decided by deterministic Turing machines of time complexity $O(t(n))$.
- NTIME($t(n)$) is the class of languages decided by nondeterministic Turing machines of time complexity $O(t(n))$.
- DSPACE($s(n)$) is the class of languages decided by deterministic Turing machines of space complexity $O(s(n))$.
- NSPACE($s(n)$) is the class of languages decided by nondeterministic Turing machines of space complexity $O(s(n))$.

Defining each class asymptotically (with the big-O) has several advantages. First, the definition of each complexity class is insensitive to the alphabet used by the Turing machines; without loss of generality, we may assume that every Turing machine uses alphabet $\Sigma_0 = \{0, 1\}$, because every step of a Turing machine with a larger alphabet can be simulated by $O(1)$ steps of a Turing machine whose alphabet is Σ_0. Second, the big-O allows us to concentrate on the high order term of the complexity function, ignoring smaller order terms, such as terms that arise from the time-constructibility maneuver above. Third, because the big-O bound holds for all n sufficiently large, languages L and L' that differ by a finite number of words belong to the same class: a machine M that decides L can be converted into a machine that decides L' by adding to M a table of answers for inputs up to a certain length.

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

- $\mathsf{DLOG} = \mathsf{DSPACE}(\log n)$ (deterministic log space)
- $\mathsf{NLOG} = \mathsf{NSPACE}(\log n)$ (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)
- $\mathsf{E} = \mathsf{DTIME}(O(1)^n) = \bigcup_{k \geq 1} \mathsf{DTIME}(k^n)$
- $\mathsf{NE} = \mathsf{NTIME}(O(1)^n) = \bigcup_{k \geq 1} \mathsf{NTIME}(k^n)$
- $\mathsf{EXP} = \mathsf{DTIME}(2^{n^{O(1)}}) = \bigcup_{k \geq 1} \mathsf{DTIME}(2^{n^k})$ (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 classes DLOG and NLOG are often denoted by L and NL, respectively. The space classes PSPACE and EXPSPACE are defined in terms of DSPACE; by Savitch's Theorem (see Theorem 2 in Section 4.1), PSPACE and EXPSPACE could also be defined by NSPACE classes.

Each of these classes contains important computational problems, some of which are listed here and in Sections 5.3 and 5.4.

The class P contains many familiar problems that can be solved efficiently, such as finding shortest paths in networks, parsing for context-free languages, sorting, matrix multiplication, and linear programming. Consequently, P has become accepted as representing the set of computationally feasible problems. Although one could legitimately argue that a problem whose best algorithm has time complexity $\Theta(n^{99})$ is really infeasible, in practice, the time complexities of the vast majority of known polynomial-time algorithms have low degrees—they run in $O(n^4)$ time or less. Moreover, P is a robust class: though defined by Turing machines, P remains the same when defined by other models of sequential computation. For example, random access machines (RAMs) (a more realistic model of computation defined in Chapter 4, on models and computability) can be used to define P, because Turing machines

and RAMs can simulate each other with polynomial-time overhead [Cook and Reckhow, 1973].

The class NP also enjoys alternative characterizations. In one characterization, NP comprises the problems whose solutions can be verified quickly, by a deterministic Turing machine in polynomial time; equivalently, NP comprises the 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 would consist of two factors $z_1 \geq 2$ and $z_2 \geq 2$ whose product $z_1 z_2 = z$. A nondeterministic Turing machine for composite numbers takes a computation path on which it guesses z_1 and z_2, and then deterministically computes the product to check whether $z_1 z_2 = z$. The machine accepts z if there exists a computation path on which $z_1 z_2 = z$. Another important language in NP is the set of satisfiable boolean formulas, SAT. A boolean formula ϕ is satisfiable if there exists a truth assignment of `true` or `false` to each of its variables such that under this truth assignment, the value of ϕ is `true`. For example, $x \wedge (\overline{x} \vee y)$ is satisfiable, but $x \wedge \overline{y} \wedge (\overline{x} \vee y)$ is not satisfiable. A nondeterministic Turing machine takes a computation path that guesses a truth assignment, and then deterministically evaluates ϕ for this truth assignment, to check whether this truth assignment proves that ϕ is satisfiable. The machine accepts ϕ if and only if there exists a computation path that finds a satisfying truth assignment.

The characterization of NP as the set of problems with easily verified solutions is formalized as follows: $L \in \mathsf{NP}$ if and only if there exist a language $L' \in \mathsf{P}$ and a polynomial p such that for every x, $x \in L$ if and only if there exists a y such that $|y| \leq p(|x|)$ and $(x, y) \in L'$. Here, y is interpreted as the solution to the problem represented by x, or equivalently, as a proof that x belongs to L.

The difference between P and NP is the difference between finding a proof of a mathematical theorem and checking that a proof is correct. In essence, P comprises the theorems that can be proved quickly from scratch (in polynomial time), and NP comprises the theorems whose proofs are short (of polynomial length).

4 Relationships Between Complexity Classes

The P vs. NP question asks about the relationship between these complexity classes: Is P a proper subset of NP, or does P = NP? Much of complexity theory focuses on the relationships between complexity classes, because these relationships have implications for the difficulty of solving computational problems. In this section, we summarize important known relationships. We demonstrate two techniques for proving relationships between classes: diagonalization and padding.

4.1 Basic Relationships

Clearly, for every $t(n)$ and $s(n)$, $\mathsf{DTIME}(t) \subseteq \mathsf{NTIME}(t)$ and $\mathsf{DSPACE}(s) \subseteq \mathsf{NSPACE}(s)$, because a deterministic machine is a special case of a nondeterministic machine. Furthermore, $\mathsf{DTIME}(t) \subseteq \mathsf{DSPACE}(t)$ and $\mathsf{NTIME}(t) \subseteq \mathsf{NSPACE}(t)$, 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.

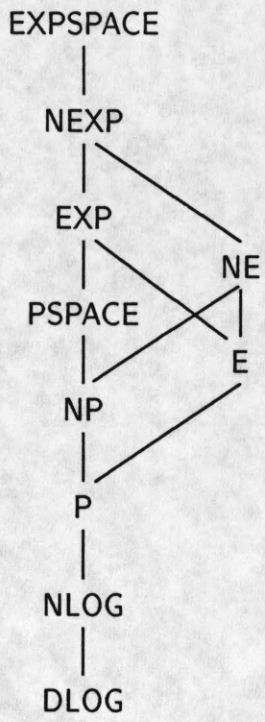

Figure 2: Inclusion relationships between the canonical complexity classes.

Theorem 2 *Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function, $s(n) \geq \log n$.*

1. $\mathsf{NTIME}(t) \subseteq \mathsf{DTIME}(O(1)^t)$.
2. $\mathsf{NSPACE}(s) \subseteq \mathsf{DTIME}(n + O(1)^s)$.
3. $\mathsf{DTIME}(t) \subseteq \mathsf{DSPACE}(t/\log t)$. [Hopcroft et al., 1977].
4. $\mathsf{NTIME}(t) \subseteq \mathsf{DSPACE}(t)$.
5. **(Savitch's Theorem)** $\mathsf{NSPACE}(s) \subseteq \mathsf{DSPACE}(s^2)$. [Savitch, 1970].

As a consequence of the first part of this theorem, NP $\subseteq$ 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 the canonical complexity classes.

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. Savitch's Theorem implies that both PSPACE and EXPSPACE can be defined by either deterministic or nondeterministic Turing machines.

4.2 Complementation

For a language L over an alphabet Σ, define $\overline{L}$ to be the complement of L in the set of words over Σ: $\overline{L} = \Sigma^* - L$. For a class of languages $\mathcal{C}$, define co-$\mathcal{C} = \{\overline{L} : L \in \mathcal{C}\}$. 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, formulas whose value is `true` for every truth assignment. Most complexity theorists believe that NP $\neq$ co-NP: it is unclear how to check that a formula on m variables is a tautology without checking all 2^m possible truth assignments, which would take exponential time.

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

1. DTIME(t) *is closed under complementation.*
2. DSPACE(s) *is closed under complementation.*
3. NSPACE(s) *is closed under complementation.* [Immerman, 1988; Szelepcsényi, 1988].

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

4.3 Hierarchy Theorems and Diagonalization

Intuitively, with more time (or more space), we should be able to solve more problems. More precisely, the class of languages decided by Turing machines with a large time complexity should strictly include the class decided by machines with a small time complexity. The next theorem confirms this intuition. In following, $\subset$ denotes strict inclusion between complexity classes.

Theorem 4 (Hierarchy Theorems) *Let $t_1(n)$ and $t_2(n)$ be time-constructible functions, and let $s_1(n)$ and $s_2(n)$ be space-constructible functions, with $s_1(n), s_2(n) \geq \log n$.*

1. *If* $t_1(n) \log t_1(n) = o(t_2(n))$, *then* DTIME($t_1$) $\subset$ DTIME(t_2).
2. *If* $t_1(n+1) = o(t_2(n))$, *then* NTIME(t_1) $\subset$ NTIME(t_2). [Seiferas et al., 1978].
3. *If* $s_1(n) = o(s_2(n))$, *then* DSPACE(s_1) $\subset$ DSPACE(s_2).
4. *If* $s_1(n) = o(s_2(n))$, *then* NSPACE(s_1) $\subset$ 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 don't 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 $\mathsf{DLOG} \subset \mathsf{PSPACE}$.

In combination with the relationship between DTIME and DSPACE in Theorem 2, the Hierarchy Theorem for DSPACE implies

$$\mathsf{DTIME}(t) \subseteq \mathsf{DSPACE}(t/\log t) \subset \mathsf{DSPACE}(t)$$

In other words, space is more valuable than time. Intuitively, Turing machines can decide more languages with t units of space than with t units of time, because tape cells can be reused.

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 machine with a fixed number of worktapes. Other computational models, such as random access machines, enjoy tighter time hierarchy theorems [Cook and Reckhow, 1973].

The Gap Theorem [Borodin, 1972] shows that the constructibility hypotheses are necessary. The Gap Theorem implies, for example, that there exists a function $t(n)$ such that $\mathsf{DTIME}(t) = \mathsf{DTIME}(2^t)$; if $t(n)$ were time-constructible, then this would be a contradiction of the Hierarchy Theorem for DTIME.

The proofs of the Hierarchy Theorems use the **diagonalization** technique. 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 $x_i \notin \mathcal{L}(M_i)$. Consequently, $\mathcal{L}(M)$ differs from each $\mathcal{L}(M_i)$, hence $\mathcal{L}(M) \notin \mathsf{DTIME}(t_1)$. 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 a proof of the Hierarchy Theorem for DSPACE.

Proof: We construct a deterministic Turing machine M that decides a language L such that $L \in \mathsf{DSPACE}(s_2) - \mathsf{DSPACE}(s_1)$. Let U be a deterministic universal Turing machine. On input x of length n, machine M performs the following:

1. Lay out $s_2(n)$ cells on a worktape.
2. On another worktape, copy the first $\min\{n, s_2(n)\}$ symbols of x, and let i be this word.
3. Simulate 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 using the constructibility of s_2 to ensure that this process halts.)
4. If U_i accepts x, then reject; if U_i rejects x, then accept.

Clearly, M always halts and uses space $O(s_2(n))$. Let $L = \mathcal{L}(M)$.

Suppose $L \in \mathsf{DSPACE}(s_1(n))$. By carefully constructing U, we can ensure that there exists a word y such that U_y decides L, and for every word z, U_{yz} also decides L in space $cs_1(n)$, where the constant c depends only on L; in essence, y is a complete description of a machine that decides L, and U_{yz} ignores symbols beyond the end of y.

Since $s_1(n) = o(s_2(n))$, there is an n_0 such that $cs_1(n) \leq s_2(n)$ for all $n \geq n_0$. Choose z so that $|y| \leq s_2(|yz|)$ and $|yz| \geq n_0$, so that $cs_1(|yz|) \leq s_2(|yz|)$. On input yz, machine M has enough space to simulate U_y on input yz. By construction, M accepts yz if and only if U_y rejects yz. Contradiction! ∎

Although the diagonalization technique successfully separates some pairs of complexity classes, diagonalization does not seem strong enough to separate P from NP. (See Theorem 9 in Section 6.)

4.4 Padding Arguments

A useful technique for establishing relationships between complexity classes is the **padding argument.** Let L 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 L** is the language

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

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

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

Theorem 5 *If* $\mathsf{P} = \mathsf{NP}$, *then* $\mathsf{E} = \mathsf{NE}$. [Book, 1974].

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

Let $L \in \mathsf{NE}$ be decided by a nondeterministic Turing machine M in at most $t(n) = k^n$ time for some constant integer k. Let L' be the $t(n)$-padded version of L. From M, we construct a nondeterministic Turing machine M' that decides L' 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, $L' \in \mathsf{NP}$.

If $\mathsf{P} = \mathsf{NP}$, then there is a deterministic Turing machine D' that decides L' in at most $p'(n)$ time for some polynomial p'. From D', we construct a deterministic Turing machine D that decides L, 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 $O(t(n)) + p'(n + t(n)) = O(1)^n$. Therefore, $\mathsf{NE} \subseteq \mathsf{E}$. ∎

5 Reducibility and Completeness

In this section, we discuss relationships between problems: informally, if one problem reduces to another problem, then in a sense, the second problem is harder than the first. The hardest

problems in NP are the NP-complete problems. We define NP-completeness precisely, and we show how to prove that a problem is NP-complete. Finally, we list some problems that are complete for NP and other complexity classes.

5.1 Resource-Bounded Reducibilities

In mathematics, as in everyday life, a typical way to solve a new problem is to reduce it to a previously solved problem. Frequently, an instance of the new problem is expressed completely in terms of an instance of the previous problem, and the solution is then interpreted in the terms of the new problem. For example, the maximum weighted matching problem for bipartite graphs reduces to the network flow problem. (See Chapter 10, on graph and network problems.) This kind of reduction is called **many-one reducibility**, and is defined below.

A different way to solve the new problem is to use a subroutine that solves the previous problem. For example, we can solve an optimization problem whose solution is feasible and maximizes the value of an objective function g by repeatedly calling a subroutine that solves the corresponding decision problem of whether there exists a feasible solution x whose value $g(x)$ satisfies $g(x) \geq k$. This kind of reduction is called **Turing reducibility**, and is defined below.

Let L_1 and L_2 be languages. L_1 is **many-one reducible** to L_2, written $L_1 \leq_m L_2$, if there exists a total recursive function f such that for all x, $x \in L_1$ if and only if $f(x) \in L_2$. The function f is called the **transformation function.** L_1 is **Turing reducible** to L_2, written $L_1 \leq_T L_2$, if L_1 can be decided by a deterministic oracle Turing machine M using L_2 as its oracle, i.e., $L_1 = \mathcal{L}(M^{L_2})$.

A reduction between problems may not be helpful if it takes too much time. To study complexity classes defined by bounds on time and space resources, it is natural to consider resource-bounded reducibilities.

Let L_1 and L_2 be languages. L_1 is **Karp reducible** to L_2, written $L_1 \leq_m^P L_2$, if L_1 is many-one reducible to L_2 via a transformation function that is computable deterministically in polynomial time. Karp reducibility is also called "polynomial-time reducibility."

L_1 is **log-space reducible** to L_2, written $L_1 \leq_m^{\log} L_2$, if L_1 is many-one reducible to L_2 via a transformation function that is computable by a deterministic Turing machine in $O(\log n)$ space.

L_1 is **Cook reducible** to L_2, written $L_1 \leq_T^P L_2$, if L_1 is Turing reducible to L_2 via a deterministic oracle Turing machine of polynomial time complexity.

A reduction from a language L_1 to a language L_2, together with a method for deciding membership in L_2, yields a method for deciding L_1. Suppose L_1 is Karp reducible to L_2 via the transformation f. If machine M_2 decides L_2, and machine M_f computes f, then to decide whether an input word x is in L_1, first use M_f to compute $f(x)$, then run M_2 on input $f(x)$. A fortiori, if the time complexity of M_2 is a polynomial t_2, and the time complexity of M_f is a polynomial t_f, then on inputs x of length $|x| = n$, the time taken by this method for deciding membership in L_1 is at most $t_f(n) + t_2(t_f(n))$, which is also a polynomial in n. In summary, if L_2 is feasible, and there is an efficient reduction from L_1 to L_2, then L_1 is feasible. We formally state this property of Karp reducibility for P in Theorem 8 after stating other properties of these reducibilities.

Log-space reducibility is useful for complexity classes within P, such as NLOG, for which Karp reducibility allows too many reductions. By definition, for every nontrivial language L_0, (i.e., $L_0 \neq \emptyset$ and $L_0 \neq \Sigma^*$), and for every L in P, necessarily $L \leq_m^P L_0$ via a transformation that simply runs a deterministic Turing machine that decides L in polynomial time. It is not known whether log-space reducibility is different from Karp reducibility, however: all transformations for known Karp reductions can be computed in $O(\log n)$ space. Even for decision problems, DLOG is not known to be a proper subset of P.

Theorem 6 *Log-space reducibility implies Karp reducibility, which implies Cook reducibility:*

1. *If* $L_1 \leq_m^{\log} L_2$, *then* $L_1 \leq_m^P L_2$.
2. *If* $L_1 \leq_m^P L_2$, *then* $L_1 \leq_T^P L_2$.

Theorem 7 *Log-space reducibility, Karp reducibility, and Cook reducibility are transitive:*

1. *If* $L_1 \leq_m^{\log} L_2$ *and* $L_2 \leq_m^{\log} L_3$, *then* $L_1 \leq_m^{\log} L_3$. [Jones, 1975].
2. *If* $L_1 \leq_m^P L_2$ *and* $L_2 \leq_m^P L_3$, *then* $L_1 \leq_m^P L_3$.
3. *If* $L_1 \leq_T^P L_2$ *and* $L_2 \leq_T^P L_3$, *then* $L_1 \leq_T^P L_3$.

A class of languages $\mathcal{C}$ is **closed under a reducibility** $\leq$ if for all languages L_1 and L_2, whenever $L_1 \leq L_2$ and $L_2 \in \mathcal{C}$, necessarily $L_1 \in \mathcal{C}$.

Theorem 8

1. P *is closed under log-space reducibility, Karp reducibility, and Cook reducibility.*
2. NP *is closed under log-space reducibility and Karp reducibility.*
3. DLOG *and* NLOG *are closed under log-space reducibility.*

We shall see the importance of closure under a reducibility in conjunction with the concept of completeness, which we define in the next section.

5.2 Complete Languages

Let $\mathcal{C}$ be a class of languages that represent computational problems. A language L_0 is **$\mathcal{C}$-hard** under a reducibility $\leq$ if for all L in $\mathcal{C}$, $L \leq L_0$. A language L_0 is **$\mathcal{C}$-complete** under $\leq$ if L_0 is $\mathcal{C}$-hard, and $L_0 \in \mathcal{C}$. Informally, if L_0 is $\mathcal{C}$-hard, then L_0 represents a problem that is at least as difficult to solve as any problem in $\mathcal{C}$. If L_0 is $\mathcal{C}$-complete, then in a sense, L_0 is one of the most difficult problems in $\mathcal{C}$.

Unless stated otherwise, Karp reducibility is generally assumed. Thus, a language L_0 is **NP-hard** if L_0 is NP-hard under Karp reducibility. L_0 is **NP-complete** if L_0 is NP-complete under Karp reducibility.

Let L_0 be NP-complete. If there exists a deterministic Turing machine that decides L_0 in polynomial time—that is, if $L_0 \in$ P—then because P is closed under Karp reducibility

(Theorem 8 in Section 5.1), it would follow that NP $\subseteq$ P, hence P = NP. In essence, the question of whether P is the same as NP reduces to whether any particular NP-complete language is in P.

A common misconception is that this property of NP-complete languages is actually their definition: that is, if $L \in$ NP, and $L \in$ P implies P = NP, then L is NP-complete. This "definition" is wrong. It is known that P $\neq$ NP if and only if there exists a language L^* in NP $-$ P such that L^* is not NP-complete [Ladner, 1975]. Thus, if P $\neq$ NP, then L^* is a counterexample to the "definition."

We have noted that an NP-complete language L_0 is unlikely to belong to P. It is also unlikely to belong to co-NP, because, by an elementary argument, if $L_0 \in$ co-NP, then NP = co-NP.

5.3 Proving NP-Completeness

> Proving NP-completeness is an important ingredient of our methodology for studying computational problems. It is also something of an art form.
>
> — *C. H. Papadimitriou* [1994]

After one language has been proved complete for a class, others can be proved complete by constructing transformations. For NP, if L_0 is NP-complete, then to prove that another language L_1 is NP-complete, it suffices to prove that $L_1 \in$ NP, and to construct a polynomial-time transformation that establishes $L_0 \leq_m^P L_1$. Since L_0 is NP-complete, for every language L in NP, $L \leq_m^P L_0$, hence by transitivity (Theorem 7 in Section 5.1), $L \leq_m^P L_1$.

Cook [1971] defined NP-completeness and proved that SAT, the language of satisfiable boolean formulas defined in Section 3.3, is NP-complete. Consequently, if deciding SAT is easy (in polynomial time), then factoring integers is easy—a surprising connection between ostensibly unrelated problems.

Beginning with Cook [1971] and Karp [1972], hundreds of computational problems in many fields of science and engineering have been proved to be NP-complete, almost always by reduction from a problem that was previously known to be NP-complete. The following NP-complete decision problems are frequently used in these reductions. (The language corresponding to each problem is the set of instances whose answers are "yes.")

3-SATISFIABILITY (3SAT)
Instance: A boolean expression ϕ in conjunctive normal form with three literals per clause (e.g., $(w \vee x \vee \overline{y}) \wedge (\overline{x} \vee y \vee z)$).
Question: Is ϕ satisfiable?

VERTEX COVER
Instance: A graph G and an integer k.
Question: Does G have a set W of k vertices such that every edge in G is incident on a vertex of W?

CLIQUE
Instance: A graph G and an integer k.

Question: Does G have a set K of k vertices such that every two vertices in K are adjacent in G?

HAMILTONIAN CIRCUIT
Instance: A graph G.
Question: Does G have a circuit that includes every vertex exactly once?

3-DIMENSIONAL MATCHING
Instance: Sets W, X, Y with $|W| = |X| = |Y| = q$ and a subset $S \subseteq W \times X \times Y$.
Question: Is there a subset $S' \subseteq S$ of size q such that no two triples in S' agree in any coordinate?

PARTITION
Instance: A set S of positive integers.
Question: Is there a subset $S' \subseteq S$ such that the sum of the elements of S' equals the sum of the elements of $S - S'$?

Here is an example of an NP-completeness proof, for the following decision problem:

TRAVELING SALESMAN PROBLEM (TSP)
Instance: A set of m cities $C_1, \ldots, C_m$, with a distance $d(i, j)$ between every pair of cities C_i and C_j, and an integer D.
Question: Is there a tour of the cities whose total length is at most D, i.e., a permutation $c_1, \ldots, c_m$ of $\{1, \ldots, m\}$, such that

$$d(c_1, c_2) + \cdots + d(c_{m-1}, c_m) + d(c_m, c_1) \leq D?$$

First, it is easy to see that TSP is in NP: a nondeterministic Turing machine simply guesses a tour and checks that the total length is at most D.

Next, we construct a reduction from HAMILTONIAN CIRCUIT to TSP. (The reduction goes from the known NP-complete problem, HAMILTONIAN CIRCUIT, to the new problem, TSP, not vice versa!)

From a graph G on m vertices $v_1, \ldots, v_m$, define the distance function d as follows:

$$d(i, j) = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge in } G \\ m + 1 & \text{otherwise} \end{cases}$$

Set $D = m$. Clearly, d and D can be computed in polynomial time from G. Each vertex of G corresponds to a city in the constructed instance of TSP.

If G has a hamiltonian circuit, then the length of the tour that corresponds to this circuit is exactly m. Conversely, if there is a tour whose length is at most m, then each step of the tour must have distance 1, not $m + 1$. Thus, each step corresponds to an edge of G, and the corresponding sequence of vertices in G is a hamiltonian circuit.

5.4 Complete Problems for Other Classes

Besides NP, the following canonical complexity classes have natural complete problems. The three problems listed below are complete for their respective classes under log-space reducibility.

NLOG: GRAPH ACCESSIBILITY PROBLEM
Instance: A directed graph G with nodes $1, \ldots, N$.
Question: Does G have a directed path from node 1 to node N?

P: CIRCUIT VALUE PROBLEM
Instance: A boolean circuit (see Section 9) with output node u, and an assignment I of $\{0, 1\}$ to each input node.
Question: Is 1 the value of u under I?

PSPACE: QUANTIFIED BOOLEAN FORMULAS
Instance: A boolean expression with all variables quantified with either $\forall$ or $\exists$ (e.g., $\forall x \forall y \exists z(x \wedge (\overline{y} \vee z))$).
Question: Is the expression `true`?

The theory of P-completeness, analogous to the theory of NP-completeness, is explained in Chapter 14, on parallel algorithms.

Stockmeyer and Meyer [1973] defined a natural decision problem that they proved to be complete for NE. If this problem were in P, then by closure under Karp reducibility (Theorem 8 in Section 5.1), we would have $\mathsf{NE} \subseteq \mathsf{P}$, a contradiction of the Hierarchy Theorems (Theorem 4 in Section 4.3). Therefore, this decision problem is infeasible: it has no polynomial-time algorithm. In contrast, decision problems in $\mathsf{NE} - \mathsf{P}$ constructed by diagonalization are unnatural.

6 Relativization of the P vs. NP Problem

Let L be a language. Define P^L (respectively, NP^L) to be the class of languages decided in polynomial time by deterministic (nondeterministic) oracle Turing machines with oracle L.

Theorem 9 *There exist languages A and B such that $\mathsf{P}^A = \mathsf{NP}^A$, and $\mathsf{P}^B \neq \mathsf{NP}^B$.* [Baker et al., 1975].

This theorem suggests that resolving the P vs. NP question demands techniques that do not relativize, i.e., that do not apply to oracle Turing machines too. Proofs that use the diagonalization technique on Turing machines without oracles generally relativize to oracle Turing machines; thus, diagonalization is unlikely to succeed in separating P from NP. The only major nonrelativizing proof technique in complexity theory appears to be the technique used to prove that IP = PSPACE (see Section 11.1).

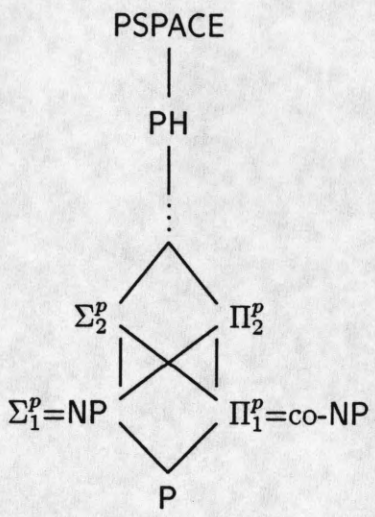

Figure 3: The polynomial hierarchy.

7 The Polynomial Hierarchy

The oracle B in Theorem 9 is an ad hoc language. Let us explore what classes we can define with oracle languages from known complexity classes.

Let $\mathcal{C}$ be a class of languages. Define

$$\mathsf{NP}^{\mathcal{C}} = \bigcup_{L \in \mathcal{C}} \mathsf{NP}^{L}$$

Define

$$\Sigma_0^P = \Pi_0^P = \mathsf{P}$$

For $k \geq 0$, define

$$\Sigma_{k+1}^P = \mathsf{NP}^{\Sigma_k^P}$$
$$\Pi_{k+1}^P = \text{co-}\Sigma_{k+1}^P$$

Observe that $\Sigma_1^P = \mathsf{NP}^{\mathsf{P}} = \mathsf{NP}$, because each of polynomially many queries to an oracle language in P can be answered directly by a (nondeterministic) Turing machine in polynomial time. Consequently, $\Pi_1^P = \text{co-NP}$. For each k, $\Sigma_k^P \subseteq \Sigma_{k+1}^P$, and $\Pi_k^P \subseteq \Sigma_{k+1}^P$, but these inclusions are not known to be strict. See Figure 3.

The classes Σ_k^P and Π_k^P constitute the **polynomial hierarchy**. Define

$$\mathsf{PH} = \bigcup_{k \geq 0} \Sigma_k^P$$

It is straightforward to prove that $\mathsf{PH} \subseteq \mathsf{PSPACE}$, but it is not known whether the inclusion is strict. In fact, if $\mathsf{PH} = \mathsf{PSPACE}$, then the polynomial hierarchy collapses to some level, i.e., $\mathsf{PH} = \Sigma_m^P$ for some m.

In the next section, we define the polynomial hierarchy in two other ways, one in terms of alternating Turing machines.

8 Alternating Complexity Classes

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 $\mathcal{L}(M)$, 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.

See Figure 4. Observe that each node in S is an accepting configuration (see Section 2.4).

We consider only alternating Turing machines that always halt. For $x \in \mathcal{L}(M)$, 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 nonblank worktape cells among configurations in the accepting tree that minimizes this number. For $x \notin \mathcal{L}(M)$, define the time to be the height of T_x, and the space to be the maximum number of nonblank 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 10 [Chandra et al., 1981]. *Let $t(n)$ be a time-constructible function, and let $s(n)$ be a space-constructible function, $s(n) \geq \log n$.*

1. $\mathsf{NSPACE}(s) \subseteq \mathsf{ATIME}(n + s^2)$
2. $\mathsf{ATIME}(t) \subseteq \mathsf{DSPACE}(t)$
3. $\mathsf{ASPACE}(s) \subseteq \mathsf{DTIME}(n + O(1)^s)$
4. $\mathsf{DTIME}(t) \subseteq \mathsf{ASPACE}(\log t)$

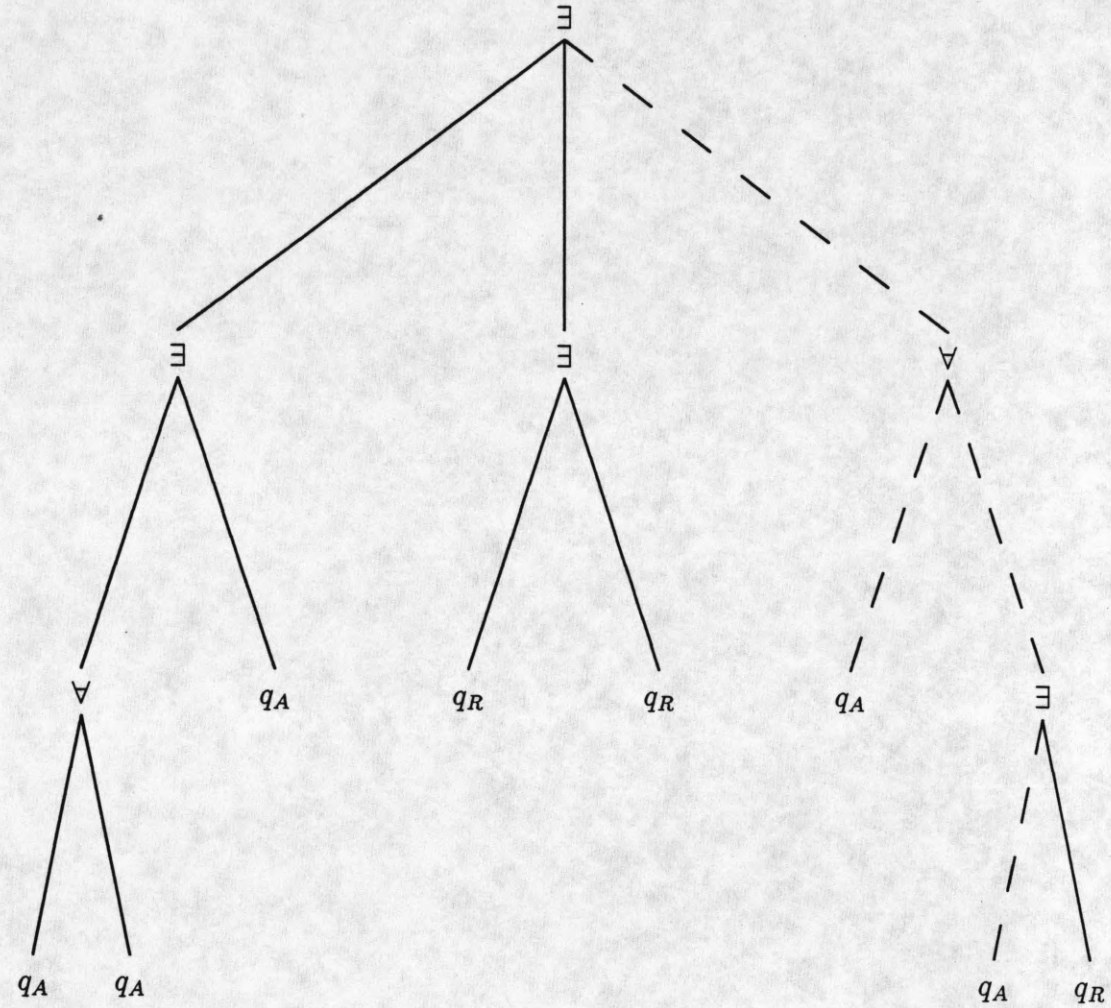

Figure 4: A computation tree of an alternating Turing machine. Each ∃ marks an existential configuration; each ∀ marks a universal configuration. The edges of one accepting subtree are drawn with dashed lines.

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. In particular, logarithmic space on alternating Turing machines corresponds to P. Polynomial time on alternating Turing machines corresponds to PSPACE. Polynomial space on alternating Turing machines corresponds to EXP.

In Section 7, we defined the classes of the polynomial hierarchy in terms of oracles, but we can also define them in terms of alternating Turing machines with restrictions on the number of alternations between existential and universal states. Define a **k-alternating Turing machine** to be a machine such that on every computation path, the number of changes from an existential state to universal state, or from a universal state to an existential state, is at most $k-1$. Thus, a nondeterministic Turing machine, which stays in existential states, is a 1-alternating Turing machine.

Theorem 11 [Stockmeyer, 1976; Wrathall, 1976]. *The following are equivalent:*

1. $L \in \Sigma_k^P$.
2. *L is decided in polynomial time by a k-alternating Turing machine that starts in an existential state.*
3. *There exists a language L' in* P *and a polynomial p such that $x \in L$ if and only if*

$$(\exists y_1 : |y_1| \leq p(|x|))(\forall y_2 : |y_2| \leq p(|x|)) \cdots (Qy_k : |y_k| \leq p(|x|))[(x, y_1, \ldots, y_k) \in L']$$

where the quantifier Q is $\exists$ if k is odd, $\forall$ if k is even.

Alternating Turing machines are closely related to boolean circuits, which are defined in the next section.

9 Circuit Complexity

The hardware of electronic digital computers is based on digital logic gates, connected into combinational networks. (See Chapter 17, on architecture components.) Here, we specify a model of computation that formalizes the (bounded fan-in) combinational network.

A **boolean circuit** on n input variables $x_1, \ldots, x_n$ is a directed acyclic graph with exactly n input nodes of indegree 0 labeled $x_1, \ldots, x_n$, and other nodes of indegree 1 or 2, called **gates**, labeled with the boolean operators in $\{\wedge, \vee, \neg\}$. One node is designated as the output of the circuit. See Figure 5. Without loss of generality, we assume that there are no extraneous nodes: there is a directed path from each node to the output node.

An input assignment I is a function that maps each variable x_i to either 0 or 1. The value of each gate g under I is obtained by applying the boolean operation that labels g to the values of the immediate predecessors of g. The function computed by the circuit is the value of the output node for each input assignment.

A boolean circuit computes a finite function: a function of only n binary input variables. To decide membership in a language, we need a circuit for each input length n.

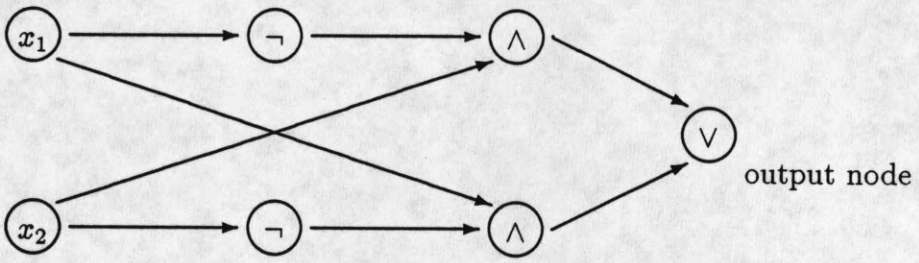

Figure 5: A boolean circuit.

A **circuit family** is an infinite set of circuits $C = \{c_1, c_2, \ldots\}$ in which each c_n is a boolean circuit on n inputs. C **decides** a language $L \subseteq \{0,1\}^*$ if for every n and every assignment $a_1, \ldots, a_n$ of $\{0,1\}$ to the n inputs, the value of the output node of c_n is 1 if and only if the word $a_1 \cdots a_n \in L$. The **size complexity** of C is the function $z(n)$ that specifies the number of nodes in each c_n. The **depth complexity** of C is the function $d(n)$ that specifies the length of the longest directed path in c_n. Clearly, since the fan-in of each gate at most 2, $d(n) \geq \log z(n) \geq \log n$.

With a different circuit for each input length, a circuit family could solve an undecidable problem such as the Halting Problem (see Chapter 4, on models and computability)! For each input length, a table of all answers for machine descriptions of that length could be encoded into the circuit. Thus, we need to restrict our circuit families. The most natural restriction is that all circuits in a family should have a concise, uniform description, to disallow a different answer table for each input length. Several uniformity conditions have been studied, and the following is the most convenient.

A circuit family $\{c_1, c_2, \ldots\}$ of size complexity $z(n)$ is **log-space uniform** if there exists a deterministic Turing machine M such that on each input of length n, machine M produces a description of c_n, using space $O(\log z(n))$.

Now we define complexity classes for uniform circuit families and relate these classes to previously defined classes. Define the following complexity classes:

- $\mathsf{SIZE}(z(n))$ is the class of languages decided by log-space uniform circuit families of size complexity $O(z(n))$.
- $\mathsf{DEPTH}(d(n))$ is the class of languages decided by log-space uniform circuit families of depth complexity $O(d(n))$.

Theorem 12

1. *If $t(n)$ is a time-constructible function, then* $\mathsf{DTIME}(t) \subseteq \mathsf{SIZE}(t \log t)$. [Pippenger and Fischer, 1979].
2. $\mathsf{SIZE}(z) \subseteq \mathsf{DTIME}(z^{O(1)})$.

3. *If $s(n)$ is a space-constructible function and $s(n) \geq \log n$, then* $\mathsf{NSPACE}(s) \subseteq \mathsf{DEPTH}(s^2)$. [Borodin, 1977].

4. $\mathsf{DEPTH}(d) \subseteq \mathsf{DSPACE}(d)$. [Borodin, 1977].

The next theorem shows that size and depth on boolean circuits are closely related to space and time on alternating Turing machines, provided that we permit sublinear running times for alternating Turing machines, as follows. We augment alternating Turing machines with a random-access input capability. To access the cell at position j on the input tape, M writes the binary representation of j on a special tape, in $\log j$ steps, and enters a special reading state to obtain the symbol in cell j.

Theorem 13 *Let $t(n) \geq \log n$ and $s(n) \geq \log n$.*

1. *Every language decided by an alternating Turing machine of simultaneous space complexity $s(n)$ and time complexity $t(n)$ can be decided by a uniform circuit family of simultaneous size complexity $O(1)^{s(n)}$ and depth complexity $O(t(n))$.* [Ruzzo, 1981].

2. *If $d(n) \geq (\log z(n))^2$, then every language decided by a uniform circuit family of simultaneous size complexity $z(n)$ and depth complexity $d(n)$ can be decided by an alternating Turing machine of simultaneous space complexity $O(\log z(n))$ and time complexity $O(d(n))$.* [Ruzzo, 1981].

In a sense, the boolean circuit family is a model of parallel computation, because all gates compute independently, in parallel. A fortiori, boolean circuits (or equivalently, alternating Turing machines) can be used to define the parallel complexity classes NC^k. (See Chapter 14, on parallel algorithms.)

10 Probabilistic Complexity Classes

Since the 1970s, with the development of randomized algorithms for computational problems (see Chapter 8, on randomized algorithms), complexity theorists have placed randomized algorithms on a firm intellectual foundation. Several definitions of randomness have been compared, and various kinds of errors distinguished. In this section, we outline some basic concepts in this area.

A **probabilistic Turing machine** M is a nondeterministic Turing machine with exactly two choices at each step. During a computation, M chooses each possible next step with independent probability $1/2$. Intuitively, at each step, M flips a fair coin to decide what to do next. The probability of a computation path of t steps is $1/2^t$. The probability that M accepts an input word x, denoted $p_M(x)$, is the sum of the probabilities of the accepting computation paths.

Throughout this section, we consider only machines whose time complexity $t(n)$ is time-constructible. Without loss of generality, we may assume that every halting computation path of the machine has exactly t steps, and terminates in either the accepting state q_A or the rejecting state q_R.

Let L be a language. A probabilistic Turing machine M decides L with

		for all $x \in L$	for all $x \notin L$
two-sided error	if	$p_M(x) > 1/2$	$p_M(x) \leq 1/2$
bounded two-sided error	if	$p_M(x) > 1/2 + \epsilon$ for some constant ϵ	$p_M(x) < 1/2 - \epsilon$
one-sided error	if	$p_M(x) > 1/2$	$p_M(x) = 0$

For example, the Solovay-Strassen primality testing algorithm of Chapter 8 (on randomized algorithms) makes one-sided errors: when the input x is a prime number, the algorithm always says "prime"; when x is composite, the algorithm usually says "composite," but may occasionally say "prime." (Actually, the Solovay-Strassen algorithm is a compositeness testing algorithm, because it errs only for inputs that are composite numbers.)

Define the following complexity classes:

- PP is the classes of languages decided by probabilistic Turing machines of polynomial time complexity with two-sided error.
- BPP is the classes of languages decided by probabilistic Turing machines of polynomial time complexity with bounded two-sided error.
- RP is the classes of languages decided by probabilistic Turing machines of polynomial time complexity with one-sided error.

In the literature, RP is also called R.

A probabilistic Turing machine M is a **PP-machine** (respectively, a **BPP-machine**, an **RP-machine**) if M has polynomial time complexity, and M decides with two-sided error (bounded two-sided error, one-sided error).

Through repeated Bernoulli trials, we can make the error probabilities of BPP-machines and RP-machines arbitrarily small:

Theorem 14 *If $L \in$ BPP, then for every polynomial $q(n)$, there exists a* BPP*-machine M such that $p_M(x) > 1 - 1/2^{q(n)}$ for every $x \in L$, and $p_M(x) < 1/2^{q(n)}$ for every $x \notin L$.*

If $L \in$ RP, then for every polynomial $q(n)$, there exists an RP*-machine M such that $p_M(x) > 1 - 1/2^{q(n)}$ for every x in L.*

Next, we define a class of problems that have probabilistic algorithms that make no errors. Define

$$\mathsf{ZPP} = \mathsf{RP} \cap \text{co-RP}$$

The letter Z in ZPP is for zero probability of error, as we now demonstrate. Suppose $L \in$ ZPP. Here is an algorithm that checks membership in L. Let M be an RP-machine that decides L, and let M' be an RP-machine that decides $\overline{L}$. For an input word x, alternately run M and M' on x, repeatedly, until a computation path of one machine accepts x. If M accepts x, then accept x; if M' accepts x, then reject x. This algorithm works correctly because when an RP-machine accepts its input, it does not make a mistake. This algorithm might not terminate, but with high probability, the algorithm terminates after a few iterations.

The next theorem expresses some known relationships between probabilistic complexity classes and other complexity classes, such as classes in the polynomial hierarchy (see Section 7).

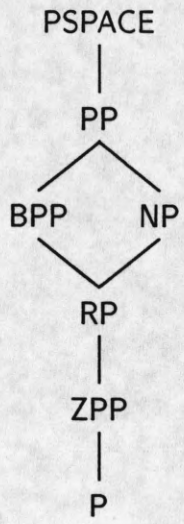

Figure 6: Probabilistic complexity classes.

Theorem 15 $\mathsf{P} \subseteq \mathsf{ZPP} \subseteq \mathsf{RP} \subseteq \mathsf{BPP} \subseteq \mathsf{PP} \subseteq \mathsf{PSPACE}$. [Gill, 1977].
$\mathsf{RP} \subseteq \mathsf{NP} \subseteq \mathsf{PP}$. [Gill, 1977].
$\mathsf{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$. [Sipser, 1983; Lautemann, 1983].
$\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$. [Toda, 1991].

See Figure 6.

11 Interactive Models and Complexity Classes

11.1 Interactive Proofs

In Section 3.3, we characterized NP as the set of languages whose membership proofs can be checked quickly, by a deterministic Turing machine M of polynomial time complexity. A different notion of proof involves interaction between two parties, a prover P and a verifier V, who exchange messages. In an **interactive proof system** [Goldwasser et al., 1989], the prover is an all-powerful machine, with unlimited computational resources, analogous to a teacher. The verifier is a computationally limited machine, analogous to a student. One of the original papers on interactive proof systems [Babai and Moran, 1988] called them "Arthur-Merlin games": the wizard Merlin corresponds to P, and the dim-witted Arthur corresponds to V.

Formally, an **interactive proof system** comprises the following:

- A read-only input tape on which an input word x is written.
- A **prover** P, whose behavior is not restricted.

- A **verifier** V, which is a probabilistic Turing machine augmented with the capability to send and receive messages. The running time of V is bounded by a polynomial in $|x|$.

- A tape on which V writes messages to send to P, and a tape on which P writes messages to send to V. The length of every message is bounded by a polynomial in $|x|$.

A computation of an interactive proof system (P, V) proceeds in rounds, as follows. For $j = 1, 2, \ldots$, in round j, V performs some steps, writes a message m_j, and temporarily stops. Then P reads m_j and responds with a message m'_j, which V reads in round $j + 1$. An interactive proof system (P, V) **accepts** an input word x if the probability of acceptance by V satisfies $p_V(x) > 1/2$.

In an interactive proof system, a prover can convince the verifier about the truth of a statement without exhibiting an entire proof. For example, consider the graph isomorphism problem: the input consists of two graphs G and H, and the decision is "yes" if and only if G is isomorphic to H. Here is an interactive proof system (P, V) for this problem. On the first round, V asks P whether the input graphs are isomorphic, but V does not immediately believe the response. If P says "yes" on the first round, then V repeatedly presents queries to P to try to construct an isomorphism. If P says "no" on the first round, then V challenges P by repeatedly asking further queries, as follows. In each round, V randomly chooses either G or H with equal probability; if V chooses G, then V computes a random permutation G' of G, presents G' to P, and asks P whether G' came from G or from H (and similarly if V chooses H). If P gave an erroneous answer on the first round, and G is isomorphic to H, then after k subsequent rounds, the probability that P answers all the subsequent queries correctly is $1/2^k$. Thus, in polynomial time, with high probability, V can check whether the original answer of P was correct.

The complexity class IP comprises the languages L for which there exists a verifier V and an ϵ such that

- there exists a prover $\hat{P}$ such that for all x in L, the interactive proof system $(\hat{P}, V)$ accepts x with probability greater than $1/2 + \epsilon$; and

- for every prover P and every $x \notin L$, the interactive proof system (P, V) rejects x with probability greater than $1/2 + \epsilon$.

By substituting random choices for existential choices in the proof that $\mathsf{ATIME}(t) \subseteq \mathsf{DSPACE}(t)$ (Theorem 10 in Section 8), it is straightforward to show that $\mathsf{IP} \subseteq \mathsf{PSPACE}$. As evidence of strict inclusion, Fortnow and Sipser [1988] constructed an oracle language A for which $\text{co-NP}^A - \mathsf{IP}^A \neq \emptyset$, and hence IP^A is strictly included in PSPACE^A. Using a proof technique that does not relativize, however, Shamir [1992] proved that in fact, IP and PSPACE are the same class.

Theorem 16 $\mathsf{IP} = \mathsf{PSPACE}$. [Shamir, 1992].

If NP is a proper subset of PSPACE, as is widely believed, then Theorem 16 says that interactive proof systems can decide a larger class of languages than NP.

11.2 Probabilistically Checkable Proofs

In an interactive proof system, the verifier does not need a complete conventional proof to become convinced about the membership of a word in a language, but uses random choices to query parts of a proof that the prover may know. A new notion of proof explicitly quantifies how much of the proof needs to be inspected.

A language L has a **probabilistically checkable proof** if there exists an oracle BPP-machine M such that

- for all $x \in L$, there exists an oracle language B_x such that M^{B_x} accepts x.
- for all $x \notin L$, and for every language B, machine M^B rejects x.

Intuitively, the oracle language B_x represents a proof of membership of x in L. Notice that B_x can be finite since the length of each possible query during a computation of M^{B_x} on x is bounded by the running time of M. The oracle language takes the role of the prover in an interactive proof system. The next theorem makes this role precise.

Theorem 17 *L has a probabilistically checkable proof if and only if L is decided by an interactive proof system with multiple provers.* [Fortnow et al., 1988].

Let $\mathsf{PCP}(r(n), q(n))$ denote the class of languages with probabilistically checkable proofs in which the probabilistic oracle Turing machine M makes $O(r(n))$ random binary choices, and queries its oracle $O(q(n))$ times. (For this definition, we assume that M has either one or two choices for each step.) By definition, $\mathsf{BPP} = \mathsf{PCP}(n^{O(1)}, 0)$, and $\mathsf{NP} = \mathsf{PCP}(0, n^{O(1)})$.

Theorem 18 $\mathsf{NP} = \mathsf{PCP}(\log n, 1)$. [Arora et al., 1992].

Theorem 18 asserts that for every language L in NP, a proof that $x \in L$ can be encoded so that the verifier can be convinced of the correctness of the proof (or detect an incorrect proof) by using only $O(\log n)$ random choices, and inspecting only a constant number of bits of the proof!

11.3 Computational Learning Theory

When we defined interactive proof systems, we compared the prover to a teacher, and the verifier to a student. In this section, we define formal models of learning, with a reliable teacher, and a computationally limited student (the learner).

Let X be a set, called the **domain**; an element of X is an **example.** A **concept class** C is a collection of subsets of X, i.e., $C \subseteq 2^X$. For a concept c in C, an example x is **positive** if $x \in c$, **negative** if $x \notin c$. The learner's task is to learn an initially unknown concept c in C.

For instance, the learner may be required to learn a boolean formula ϕ on n variables from a class of boolean formulas Φ. In this case, the domain is the set of binary n-tuples, $\{0,1\}^n$, and a concept is the set of n-tuples $(x_1, \ldots, x_n)$ on which $\phi(x_1, \ldots, x_n) = 1$. Each n-tuple example represents a truth assignment, with $0 = \texttt{false}$ and $1 = \texttt{true}$. An example $(x_1, \ldots, x_n)$ is positive if $\phi(x_1, \ldots, x_n) = 1$, negative if $\phi(x_1, \ldots, x_n) = 0$. The task of the

learner is to output a boolean formula in Φ that is equivalent to ϕ, and hence specifies the same concept. Each boolean formula is a particular representation of an underlying concept, which formally is a subset of $\{0,1\}^n$; learnability results often depend on the representation of concept classes.

In computational learning theory, there are two basic models. In the Exact Learning Model, the learner (student) presents queries to an oracle (teacher), which provides the correct answers. In the PAC Model, the learner receives positive and negative examples, but has no choice about the examples.

Exact Learning Model. [Angluin, 1988]. In each round, the learning algorithm proposes a hypothesis c'. If $c' = c$, then the oracle responds "yes." If not, then the oracle provides a counterexample: an example on which c and c' differ. A concept class C is **learnable** if there is a learning algorithm that eventually outputs a correct hypothesis for each possible c in C, such that the total running time of the algorithm is a polynomial in the size $|c|$ of (the representation of) c.

PAC (Probably Approximately Correct) Model. [Valiant, 1984]. The learning algorithm receives a sequence of examples, generated according to an arbitrary probability distribution D on the domain. Each example is labeled correctly as positive or negative. A concept class C is **learnable** if for every c in C, for all ϵ and δ such that $0 < \epsilon, \delta < 1$, and for all probability distributions D, there is a learning algorithm with the following behavior: the algorithm runs in polynomial time $t(|c|, 1/\epsilon, 1/\delta)$ with probability at least $1 - \delta$, and the algorithm outputs (the representation of) a concept c' such that the probability measure of the examples on which c and c' differ, $D(c \oplus c')$, satisfies $D(c \oplus c') \leq \epsilon$.

The two models differ primarily in their criteria for success. The Exact Learning Model requires logical equivalence, whereas the PAC Model requires only approximate distribution-weighted equivalence.

Theorem 19 *Every concept class learnable under the Exact Learning Model is also learnable under the PAC Model.* [Angluin, 1987].

In both models, to learn a concept, a learning algorithm finds a hypothesis that is consistent with the examples that it has received. The "Occam's razor" principle of Blumer et al. [1987] asserts that if a sufficiently short hypothesis (not necessarily the shortest) consistent with sufficiently many examples can be computed in polynomial time, then the concept is PAC-learnable.

Many results on learnability and non-learnability (under complexity theoretic assumptions) have been proved for classes of boolean formulas. Here are two examples.

Theorem 20 *The following classes of boolean formulas are PAC-learnable:*

1. *Monomials: conjunctions of literals (e.g., $(w \wedge \overline{x} \wedge \overline{y} \wedge z)$).* [Valiant, 1984].
2. *k-DNF: formulas in disjunctive normal form with at most k literals per (monomial) term.*
3. *k-CNF: formulas in conjunctive normal form with at most k literals per clause (e.g., 3SAT).* [Valiant, 1984].

Theorem 21 [Pitt and Valiant, 1988]. *If* RP $\neq$ NP, *then the following classes of boolean formulas are not PAC-learnable for* $k \geq 2$*:*

1. *k-term-DNF: formulas in disjunctive normal form with at most k terms.*

2. *k-clause-CNF: formulas in conjunctive normal form with at most k clauses.*

Results such as Theorem 21 indicate that only simple kinds of concepts are learnable in the two basic models. Thus, to expand the classes of learnable concepts, learning theorists have devised numerous enhancements to the models. Both models can be augmented by allowing the learning algorithm to ask additional queries, such as requests for positive or negative examples. For a **membership query**, the learning algorithm presents an example x to the oracle, which tells whether x is a positive or negative example.

See Section IV of this *Handbook,* on artificial intelligence, for other work on models and algorithms for machine learning and concept acquisition.

12 Kolmogorov Complexity

Until now, we have considered only dynamic complexity measures, namely, the time and space used by running Turing machines. Kolmogorov complexity is a static complexity measure that captures the difficulty of describing a word. For example, the word consisting of three million zeroes can be described with fewer than three million symbols (as in this sentence). In contrast, for a word consisting of three million randomly generated bits, there is probably no shorter description than the word itself.

Let U be a universal Turing machine. Let λ denote the empty word. The **Kolmogorov complexity** of a binary word y with respect to U, denoted $K_U(y)$, is the length of the shortest binary word i such that on input $\langle i, \lambda \rangle$, machine U outputs y. In essence, i is a description of y, for it tells U how to generate y.

The next theorem states that different choices for the universal Turing machine affect the definition of Kolmogorov complexity in only a small way.

Theorem 22 (Invariance Theorem) *There exists a universal Turing machine U such that for every universal Turing machine U', there is a constant c such that for all y,*

$$K_U(y) \leq K_{U'}(y) + c$$

Henceforth, let K be defined by the universal Turing machine of Theorem 22. For every integer n and every binary word y of length n, because y can be described by giving itself explicitly, $K(y) \leq n + c'$ for a constant c'. Call y **incompressible** if $K(y) \geq n$. Since there are 2^n binary words of length n, and only $2^n - 1$ possible shorter descriptions, there exists an incompressible word for every length n.

Kolmogorov complexity has been used to prove many lower bounds on computational complexity. For example, Maass et al. [1987] constructed a language L_{SMT} (sparse matrix transposition) that can be decided by a 2-tape deterministic Turing machine of time complexity $O(n)$, but every 1-tape Turing machine that decides L_{SMT} requires $\Omega(n \log n / \log \log n)$ time.

13 Research Issues and Summary

The core research questions in complexity theory are expressed in terms of separating complexity classes:

- Is DLOG different from NLOG?
- Is P different from RP or BPP?
- Is P different from NP?
- Is NP different from PSPACE?

Motivated by these questions, much current research is devoted to efforts to understand the power of nondeterminism, randomization, and interaction. In these studies, researchers have gone well beyond the theory presented in this chapter:

- beyond Turing machines and boolean circuits, to restricted and specialized models in which nontrivial lower bounds on complexity can be proved;
- beyond Karp reducibility and Cook reducibility, to other kinds of reducibilities;
- beyond worst case complexity, to average case complexity;
- beyond decision problems, to enumeration problems and optimization problems.

Recent research in complexity theory has had direct applications to other areas of computer science and mathematics. Results on the existence of probabilistically checkable proofs imply that obtaining approximate solutions to NP-complete problems can be as difficult as solving them exactly. Complexity theory provides new tools for studying questions in finite model theory, a branch of mathematical logic. Some questions about logical expressibility are equivalent to open questions about relationships between complexity classes. Fundamental questions in complexity theory are intimately linked to practical questions about the use of cryptography for computer security, such as the existence of one-way functions and the strength of public key cryptosystems.

With precisely defined models and mathematically rigorous proofs, research in complexity theory will continue to provide sound insights into the difficulty of solving real computational problems.

14 Defining Terms

Complexity class: A set of languages that are decided within a particular resource bound. For example, $\mathsf{NTIME}(n^2 \log n)$ is the set of languages decided by nondeterministic Turing machines within $O(n^2 \log n)$ time. (See Section 3.3.)

Constructibility: A function $f(n)$ is time- (respectively, space-) constructible if there exists a deterministic Turing machine that halts after exactly $f(n)$ steps (after using exactly $f(n)$ worktape cells) for every input of length n. (See Section 3.2.)

Diagonalization: A technique for constructing a language L that differs from every $\mathcal{L}(M_i)$ for a list of machines $M_1, M_2, \ldots$. (See Section 4.3.)

NP-complete: A language L_0 is NP-complete if $L_0 \in \mathsf{NP}$ and $L \leq_m^P L_0$ for every L in NP; that is, for every L in NP, there exists a function f computable in polynomial time such that for every x, $x \in L$ if and only if $f(x) \in L_0$. (See Sections 5.1 and 5.2.)

Oracle: An oracle is a language L to which a machine presents queries of the form "Is w in L" and receives the correct answers at no cost. (See Sections 2.5, 6, and 7.)

Padding: A technique for establishing relationships between complexity classes that uses padded versions of languages, in which each word is padded out with multiple occurrences of a new symbol—the word x is replaced by the word $x\#^{f(n)}$—in order to artificially reduce the complexity of the language. (See Section 4.4.)

Reduction: A language L_1 reduces to a language L_2 if a machine that decides L_2 can be used to decide L_1 efficiently. (See Section 5.1.)

Time and space complexity: The time (respectively, space) complexity of a deterministic Turing machine M is the maximum number of steps taken (nonblank cells used) by M among all input words of length n. (See Section 3.1.)

Turing machine: A Turing machine M is a model of computation with a read-only input tape and multiple worktapes. At each step, M reads the tape cells on which its access heads are located, and depending on its current state and the symbols in those cells, M changes state, writes new symbols on the worktape cells, and moves each access head one cell left or right or not at all. (See Section 2.2.)

15 Further Information

Three contemporary textbooks on complexity theory are by Balcázar et al. [1998, 1990], by Bovet and Crescenzi [1994], and by Papadimitriou [1994]. The exhaustive survey of complexity theory by Wagner and Wechsung [1986] covers work published before 1986.

A good general reference is the *Handbook of Theoretical Computer Science* [van Leeuwen, 1990]. The following chapters in the *Handbook* are particularly relevant: Machine models and simulations, by P. van Emde Boas, pp. 1–66; A catalog of complexity classes, by D. S. Johnson, pp. 67–161; Machine-independent complexity theory, by J. I. Seiferas, pp. 163–186; Kolmogorov complexity and its applications, by M. Li and P. M. B. Vitányi, pp. 187–254; and The complexity of finite functions, by R. B. Boppana and M. Sipser, pp. 757–804, which covers circuit complexity.

A collection of articles edited by Hartmanis [1989] includes an overview of complexity theory, and chapters on sparse complete languages, on relativizations, on interactive proof systems, and on applications of complexity theory to cryptography.

For specific topics in complexity theory, the following references are helpful. Garey and Johnson [1979] explain NP-completeness thoroughly, with examples of NP-completeness proofs, and a collection of hundreds of NP-complete problems. Li and Vitányi [1993] provide a comprehensive scholarly treatment of Kolmogorov complexity, with many applications.

Angluin [1992] and Kearns and Vazirani [1994] give up-to-date introductions to computational learning theory.

For historical perspectives on complexity theory, see Hartmanis [1994], Sipser [1992], and Stearns [1990].

Research papers on complexity theory are presented at several annual conferences, including the annual ACM Symposium on Theory of Computing; the annual International Colloquium on Automata, Languages, and Programming, sponsored by the European Association for Theoretical Computer Science (EATCS); and the annual Symposium on Foundations of Computer Science, sponsored by the IEEE. The annual Conference on Computational Complexity (formerly Structure in Complexity Theory), also sponsored by the IEEE, is entirely devoted to complexity theory. Research articles on complexity theory regularly appear in the following journals, among others: *Computational Complexity, Information and Computation, Journal of the ACM, Journal of Computer and System Sciences, Mathematical Systems Theory, SIAM Journal on Computing,* and *Theoretical Computer Science.* Each issue of *ACM SIGACT News* and *Bulletin of the EATCS* contains a column on complexity theory.

Acknowledgments Eric Allender, Donna Brown, Bevan Das, Lane Hemaspaandra, Leonard Pitt, Kenneth Regan, and Martin Tompa kindly read earlier versions of this chapter and suggested numerous helpful improvements. Karen Walny drew the figures and checked the references.

16 References

Angluin, D. 1987. Learning regular sets from queries and counterexamples. *Inform. Comput.* 75(2):87–106.

Angluin, D. 1988. Queries and concept learning. *Machine Learning* 2(4):319–342.

Angluin, D. 1992. Computational learning theory: survey and selected bibliography. *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, Victoria, B.C., Canada, pp. 351–369.

Arora, S., Lund, C., Motwani, R., Sudan, M., and Szegedy, M. 1992. Proof verification and hardness of approximation problems. *Proceedings, 33rd Annual Symposium on Foundations of Computer Science,* Pittsburgh, PA, IEEE, pp. 14–23.

Babai, L., and Moran, S. 1988. Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity classes, *J. Comput. System Sci.* 36(2):254–276.

Baker, T., Gill, J., and Solovay, R. 1975. Relativizations of the P =? NP question. *SIAM J. Comput.* 4(4):431–442.

Balcázar, J. L., Díaz, J., and Gabarró, J. 1988. *Structural Complexity I*, Springer-Verlag, Berlin.

Balcázar, J. L., Díaz, J., and Gabarró, J. 1990. *Structural Complexity II*, Springer-Verlag, Berlin.

Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. 1987. Occam's razor. *Inform. Process. Lett.* 24(6):377–380.

Book, R. V. 1974. Comparing complexity classes. *J. Comput. System Sci.* 9(2):213–229.

Borodin, A. 1972. Computational complexity and the existence of complexity gaps. *J. ACM* 19(1):158–174.

Borodin, A. 1977. On relating time and space to size and depth. *SIAM J. Comput.* 4(4):733–744.

Bovet, D. P. and Crescenzi, P. 1994. *Introduction to the Theory of Complexity*, Prentice Hall International (UK) Limited, Hertfordshire, U.K.

Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. 1981. Alternation. *J. ACM* 28(1):114–133.

Cook, S. A. 1971. The complexity of theorem-proving procedures. *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, Shaker Heights, OH, pp. 151–158.

Cook, S. A., and Reckhow, R. A. 1973. Time bounded random access machines. *J. Comput. System Sci.* 7(4):354–375.

Fischer, M. J. 1980. On developing a theory of distributed computing: summary of current research. Tech. Rept. 80-09-03, Dept. of Computer Science, Univ. of Washington, Seattle, WA.

Fortnow, L., and Sipser, M. 1988. Are there interactive protocols for co-NP languages? *Inform. Process. Lett.* 28(5):249–251.

Fortnow, L., Rompel, J., and Sipser, M. 1988. On the power of multi-prover interactive protocols. *Proceedings, Structure in Complexity Theory: Third Annual Conference,* Washington, D.C., IEEE, pp. 156–161.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman and Company, San Francisco.

Gill, J. 1977. Computational complexity of probabilistic Turing machines. *SIAM J. Comput.* 6(4):675–695.

Goldwasser, S., Micali, S., and Rackoff, C. 1989. The knowledge complexity of interactive proof systems. *SIAM J. Comput.* 18(1):186–208.

Hartmanis, J., ed. 1989. *Computational Complexity Theory,* American Mathematical Society, Providence, R.I.

Hartmanis, J. 1994. On computational complexity and the nature of computer science. *Commun. ACM* 37(10):37–43.

Hartmanis, J., and Stearns, R. E. 1965. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.* 117:285–306.

Hopcroft, J., Paul, W., and Valiant, L. 1977. On time versus space. *J. ACM* 24(3):332–337.

Immerman, N. 1988. Nondeterministic space is closed under complementation. *SIAM J. Comput.* 17(5):935–938.

Jones, N. D. 1975. Space-bounded reducibility among combinatorial problems. *J. Comput. System Sci.* 11(1):68–85.

Karp, R. M. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations,* ed. R. E. Miller and J. W. Thatcher, pp. 85–103, Plenum Press, New York.

Kearns, M. J., and Vazirani, U. V. 1994. *Introduction to Computational Learning Theory,* M.I.T. Press, Cambridge, MA.

Kobayashi, K. 1985. On proving time constructibility of functions. *Theoret. Comput. Sci.* 35(2,3):215–225.

Ladner, R. E. 1975. On the structure of polynomial time reducibility. *J. ACM* 22(1):155–171.

Lautemann, C. 1983. BPP and the polynomial hierarchy. *Inform. Process. Lett.* 17(4):215–217.

Li, M., and Vitányi, P. M. B. 1993. *An Introduction to Kolmogorov Complexity and Its Applications*, Springer-Verlag, New York.

Maass, W., Schnitger, G., and Szemeredi, E. 1987. Two tapes are better than one for off-line Turing machines. *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, New York, NY, pp. 94–100.

Papadimitriou, C. H. 1994. *Computational Complexity*, Addison-Wesley Publishing Company, Reading, MA.

Pippenger, N., and Fischer, M. J. 1979. Relations among complexity measures. *J. ACM* 26(2):361–381.

Pitt, L., and Valiant, L. G. 1988. Computational limitations on learning from examples. *J. ACM* 35(4):965–984.

Ruzzo, W. L. 1981. On uniform circuit complexity. *J. Comput. System Sci.* 22(3):365–383.

Savitch, W. J. 1970. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. System Sci.* 4(2):177–192.

Seiferas, J. I., Fischer, M. J., and Meyer, A. R. 1978. Separating nondeterministic time complexity classes. *J. ACM* 25(1)146–167.

Shamir, A. 1992. IP = PSPACE. *J. ACM* 39(4):869–877.

Sipser, M. 1983. A complexity theoretic approach to randomness. *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, Boston, MA, pp. 330–335.

Sipser, M. 1992. The history and status of the P versus NP question. *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, Victoria, B.C., Canada, pp. 603–618.

Stearns, R. E. 1990. Juris Hartmanis: the beginnings of computational complexity. In *Complexity Theory Retrospective,* ed. A. L. Selman, pp. 5–18, Springer-Verlag, New York.

Stockmeyer, L. J. 1976. The polynomial-time hierarchy. *Theoret. Comput. Sci.* 3(1)1–22.

Stockmeyer, L. J., and Chandra, A. K. 1979. Intrinsically difficult problems. *Scientific American* 240(5):140–159.

Stockmeyer, L. J., and Meyer, A. R. 1973. Word problems requiring exponential time: preliminary report. *Proceedings of the 5th Annual ACM Symposium on Theory of Computing,* Austin, TX, pp. 1–9.

Szelepcsényi, R. 1988. The method of forced enumeration for nondeterministic automata. *Acta Inform.* 26(3):279–284.

Toda, S. 1991. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.* 20(5):865–877.

Valiant, L. G. 1984. A theory of the learnable. *Commun. ACM* 27(11):1134–1142.

van Leeuwen, J. 1990. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity,* Elsevier Science Publishers, Amsterdam, The Netherlands, and M.I.T. Press, Cambridge, MA.

Wagner, K., and Wechsung, G. 1986. *Computational Complexity,* D. Reidel Publishing, Dordrecht, The Netherlands.

Wrathall, C. 1976. Complete sets and the polynomial-time hierarchy. *Theoret. Comput. Sci.* 3(1):23–33.