CHAPTER 1

The Role of Theory in Computer Science

Computer science is the study of computers and programs, the collections of instructions that direct the activity of computers. Although computers are made of simple elements, the tasks they perform are often very complex. The great disparity between the simplicity of computers and the complexity of computational tasks offers intellectual challenges of the highest order. It is the models and methods of analysis developed by computer science to meet these challenges that are the subject of theoretical computer science.

Computer scientists have developed models for machines, such as the random-access and Turing machines; for languages, such as regular and context-free languages; for programs, such as straight-line and branching programs; and for systems of programs, such as compilers and operating systems. Models have also been developed for data structures, such as heaps, and for databases, such as the relational and object-oriented databases.

Methods of analysis have been developed to study the efficiency of algorithms and their data structures, the expressibility of languages and the capacity of computer architectures to recognize them, the classification of problems by the time and space required to solve them, their inherent complexity, and limits that hold simultaneously on computational resources for particular problems. This book examines each of these topics in detail except for the first, analysis of algorithms and data structures, which it covers only briefly.

This chapter provides an overview of the book. Except for the mathematical preliminaries, the topics introduced here are revisited later.

1.1 A Brief History of Theoretical Computer Science

Theoretical computer science uses models and analysis to study computers and computation. It thus encompasses the many areas of computer science sufficiently well developed to have models and methods of analysis. This includes most areas of the field.

1.1.1 Early Years

TURING AND CHURCH: Theoretical computer science emerged primarily from the work of Alan Turing and Alonzo Church in 1936, although many others, such as Russell, Hilbert, and Boole, were important precursors. Turing and Church introduced formal computational models (the Turing machine and lambda calculus), showed that some well-stated computational problems have no solution, and demonstrated the existence of universal computing machines, machines capable of simulating every other machine of their type.

Turing and Church were logicians; their work reflected the concerns of mathematical logic. The origins of computers predate them by centuries, going back at least as far as the abacus, if we call any mechanical aid to computation a computer. A very important contribution to the study of computers was made by Charles Babbage, who in 1836 completed the design of his first programmable Analytical Engine, a mechanical computer capable of arithmetic operations under the control of a sequence of punched cards (an idea borrowed from the Jacquard loom). A notable development in the history of computers, but one of less significance, was the 1938 demonstration by Claude Shannon that Boolean algebra could be used to explain the operation of relay circuits, a form of electromechanical computer. He was later to develop his profound "mathematical theory of communication" in 1948 as well as to lay the foundations for the study of circuit complexity in 1949.

FIRST COMPUTERS: In 1941 Konrad Zuse built the Z3, the first general-purpose program-controlled computer, a machine constructed from electromagnetic relays. The Z3 read programs from a punched paper tape. In the mid-1940s the first programmable electronic computer (using vacuum tubes), the ENIAC, was developed by Eckert and Mauchly. Von Neumann, in a very influential paper, codified the model that now carries his name. With the invention of the transistor in 1947, electronic computers were to become much smaller and more powerful than the 30-ton ENIAC. The microminiaturization of transistors continues today to produce computers of ever-increasing computing power in ever-shrinking packages.

EARLY LANGUAGE DEVELOPMENT: The first computers were very difficult to program (cables were plugged and unplugged on the ENIAC). Later, programmers supplied commands by typing in sequences of 0's and 1's, the machine language of computers. A major contribution of the 1950s was the development of programming languages, such as FORTRAN, COBOL, and LISP. These languages allowed programmers to specify commands in mnemonic code and with high level constructs such as loops, arrays, and procedures.

As languages were developed, it became important to understand their expressiveness as well as the characteristics of the simplest computers that could translate them into machine language. As a consequence, formal languages and the automata that recognize them became an important topic of study in the 1950s. Nondeterministic models – models that may have more than one possible next state for the current state and input – were introduced during this time as a way to classify languages.

1.1.2 1950s

FINITE-STATE MACHINES: Occurring in parallel with the development of languages was the development of models for computers. The 1950s also saw the formalization of the finite-state machine (also called the sequential machine), the sequential circuit (the concrete realization of a sequential machine), and the pushdown automaton. Rabin and Scott pioneered the use of analytical tools to study the capabilities and limitations of these models.

FORMAL LANGUAGES: The late 1950s and 1960s saw an explosion of research on formal languages. By 1964 the Chomsky language hierarchy, consisting of the regular, context-free, context-sensitive, and recursively enumerable languages, was established, as was the correspondence between these languages and the memory organizations of machine types recognizing them, namely the finite-state machine, the pushdown automaton, the linear-bounded automaton, and the Turing machine. Many variants of these standard grammars, languages, and machines were also examined.

1.1.3 1960s

COMPUTATIONAL COMPLEXITY: The 1960s also saw the laying of the foundation for computational complexity with the classification of languages and functions by Hartmanis, Lewis, and Stearns and others of the time and space needed to compute them. Hierarchies of problems were identified and speed-up and gap theorems established. This area was to flower and lead to many important discoveries, including that by Cook (and independently Levin) of **NP**-complete languages, languages associated with many hard combinatorial and optimization problems, including the Traveling Salesperson problem, the problem of determining the shortest tour of cities for which all intercity distances are given. Karp was instrumental in demonstrating the importance of **NP**-complete languages. Because problems whose running time is exponential are considered intractable, it is very important to know whether a string in **NP**-complete languages can be recognized in a time polynomial in their length. This is called the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem, where **P** is the class of deterministic polynomial-time languages. The **P**-complete languages were also identified in the 1970s; these are the hardest languages in **P** to recognize on parallel machines.

1.1.4 1970s

COMPUTATION TIME AND CIRCUIT COMPLEXITY: In the early 1970s the connection between computation time on Turing machines and circuit complexity was established, thereby giving the study of circuits renewed importance and offering the hope that the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem could be resolved via circuit complexity.

PROGRAMMING LANGUAGE SEMANTICS: The 1970s were a very productive period for formal methods in the study of programs and languages. The area of programming language semantics was very active; models and denotations were developed to give meaning to the phrase "programming language," thereby putting language development on a solid footing. Formal methods for ensuring the correctness of programs were also developed and applied to program development. The 1970s also saw the emergence of the relational database model and the

development of the relational calculus as a means for the efficient reformulation of database queries.

SPACE-TIME TRADEOFFS: An important byproduct of the work on formal languages and semantics in the 1970s is the pebble game. In this game, played on a directed acyclic graph, pebbles are placed on vertices to indicate that the value associated with a vertex is located in the register of a central processing unit. The game allows the study of tradeoffs between the number of pebbles (or registers) and time (the number of pebble placements) and leads to space-time product inequalities for individual problems. These ideas were generalized in the 1980s to branching programs.

VLSI MODEL: When the very large-scale integration (VLSI) of electronic components onto semiconductor chips emerged in the 1970s, VLSI models for them were introduced and analyzed. Ideas from the study of pebble games were applied and led to tradeoff inequalities relating the complexity of a problem to products such as AT^2, where A is the area of a chip and T is the number of steps it takes to solve a problem. In the late 1970s and 1980s the layout of computers on VLSI chips also became an important research topic.

ALGORITHMS AND DATA STRUCTURES: While algorithms (models for programs) and data structures were introduced from the beginning of the field, they experienced a flowering in the 1970s and 1980s. Knuth was most influential in this development, as later were Aho, Hopcroft, and Ullman. New algorithms were invented for sorting, data storage and retrieval, problems on graphs, polynomial evaluation, solving linear systems of equations, computational geometry, and many other topics on both serial and parallel machines.

1.1.5 1980s and 1990s

PARALLEL COMPUTING AND I/O COMPLEXITY: The 1980s also saw the emergence of many other theoretical computer science research topics, including parallel and distributed computing, cryptography, and I/O complexity. A variety of concrete and abstract models of parallel computers were developed, ranging from VLSI-based models to the parallel random-access machine (PRAM), a collection of synchronous processors alternately reading from and writing to a common array of memory cells and computing locally. Parallel algorithms and data structures were developed, as were classifications of problems according to the extent to which they are parallelizable. I/O complexity, the study of data movement among memory units in a memory hierarchy, emerged around 1980. Memory hierarchies take advantage of the temporal and spatial locality of problems to simulate fast, expensive memories with slow and inexpensive ones.

DISTRIBUTED COMPUTING: The emergence of networks of computers brought to light some hard logical problems that led to a theory of distributed computing, that is, computing with multiple and potentially asynchronous processors that may be widely dispersed. The problems addressed in this area include reaching consensus in the presence of malicious adversaries, handling processor failures, and efficiently coordinating the activities of agents when interprocessor latencies are large. Although some of the problems addressed in distributed computing were first introduced in the 1950s, this topic is associated with the 1980s and 1990s.

CRYPTOGRAPHY: While cryptography has been important for ages, it became a serious concern of complexity theorists in the late 1970s and an active research area in the 1980s and 1990s. Some of the important cryptographic issues are a) how to exchange information secretly without having to exchange a private key with each communicating agent, b) how to identify with high assurance the sender of a message, and c) how to convince another agent that you have the solution to a problem without transferring the solution to him or her.

As this brief history illustrates, theoretical computer science speaks to many different computational issues. As the range of issues addressed by computer science grows in sophistication, we can expect a commensurate growth in the richness of theoretical computer science.

1.2 Mathematical Preliminaries

In this section we introduce basic concepts used throughout the book. Since it is presumed that the reader has already met most of this material, this presentation is abbreviated.

1.2.1 Sets

A **set** A is a non-repeating and unordered collection of elements. For example, $A_{50s} = \{\text{Cobol, Fortran, Lisp}\}$ is a set of elements that could be interpreted as the names of languages designed in the 1950s. Because the elements in a set are unordered, {Cobol, Fortran, Lisp} and {Lisp, Cobol, Fortran} denote the same set. It is very convenient to recognize the **empty set** $\emptyset$, a set that does not have any elements. The set $\mathcal{B} = \{0, 1\}$ containing 0 and 1 is used throughout this book.

The notation $a \in A$ means that element a is contained in set A. For example, Cobol $\in A_{50s}$ means that Cobol is a language invented in the 1950s. A set can be finite or infinite. The **cardinality** of a finite set A, denoted $|A|$, is the number of elements in A. We say that a set A is a **subset** of a set B, denoted $A \subseteq B$, if every element of A is an element of B. If $A \subseteq B$ but B contains elements not in A, we say that A is a **proper subset** and write $A \subset B$.

The **union** of two sets A and B, denoted $A \cup B$, is the set containing elements that are in A, B or both. For example, if $A_0 = \{1, 2, 3\}$ and $B_0 = \{4, 3, 5\}$, then $A_0 \cup B_0 = \{5, 4, 3, 1, 2\}$. The **intersection** of sets A and B, denoted $A \cap B$, is the set containing elements that are in both A and B. Hence, $A_0 \cap B_0 = \{3\}$. If A and B have no elements in common, denoted $A \cap B = \emptyset$, they are said to be **disjoint sets**. The **difference** between sets A and B, denoted $A - B$, is the set containing the elements that are in A but not in B. Thus, $A_0 - B_0 = \{1, 2\}$. (See Fig. 1.1.)

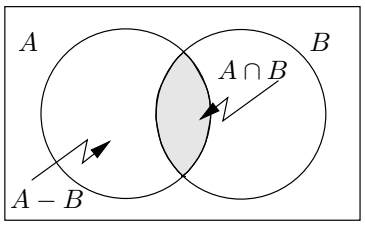

Figure 1.1 A Venn diagram showing the intersection and difference of sets A and B. Their union is the set of elements in both A and B.

The following simple properties hold for arbitrary sets A and B and the operations of set union, intersection, and difference:

$$A \cup B = B \cup A$$
$$A \cap B = B \cap A$$
$$A \cup \emptyset = A$$
$$A \cap \emptyset = \emptyset$$
$$A - \emptyset = A$$

The **power set** of a set A, denoted 2^A, is the set of all subsets of A including the empty set. For example, $2^{\{2,5,9\}} = \{\emptyset, \{2\}, \{5\}, \{9\}, \{2,5\}, \{2,9\}, \{5,9\}, \{2,5,9\}\}$. We use 2^A to denote the power set A as a reminder that it has $2^{|A|}$ elements. To see this, observe that for each subset B of the set A there is a binary n-tuple $(e_1, e_2, \ldots, e_{|A|})$ where e_i is 1 if the ith element of A is in B and 0 otherwise. Since there are $2^{|A|}$ ways to assign 0's and 1's to $(e_1, e_2, \ldots, e_{|A|})$, 2^A has $2^{|A|}$ elements.

The **Cartesian product** of two sets A and B, denoted $A \times B$, is another set, the set of pairs $\{(a, b) \mid a \in A, b \in B\}$. For example, when $A_0 = \{1, 2, 3\}$ and $B_0 = \{4, 3, 5\}$, $A_0 \times B_0 = \{(1,4), (1,3), (1,5), (2,4), (2,3), (2,5), (3,4), (3,3), (3,5)\}$. The Cartesian product of k sets $A_1, A_2, \ldots, A_k$, denoted $A_1 \times A_2 \times \cdots \times A_k$, is the set of k-tuples $\{(a_1, a_2, \ldots, a_k) \mid a_1 \in A_1, a_2 \in A_2, \ldots, a_k \in A_k\}$ whose components are drawn from the respective sets. If for each $1 \leq i \leq k$, $A_i = A$, the k-fold Cartesian product $A_1 \times A_2 \times \cdots \times A_k$ is denoted A^k. An element of A^k is a k-tuple $(a_1, a_2, \ldots, a_k)$ where $a_i \in A$. Thus, the binary n-tuple $(e_1, e_2, \ldots, e_{|A|})$ of the preceding paragraph is an element of $\{0, 1\}^n$.

1.2.2 Number Systems

Integers are widely used to describe problems. The infinite set $\mathbb{N}$ consisting of 0 and the positive integers $\{1, 2, 3, \ldots\}$ is called the set of **natural numbers**. The set of positive and negative integers and zero, $\mathbb{Z}$, consists of the integers $\{0, 1, -1, 2, -2, \ldots\}$.

In the **standard decimal representation** of the natural numbers, each integer n is represented as a sum of powers of 10. For example, $867 = 8 \times 10^2 + 6 \times 10^1 + 7 \times 10^0$. Since computers today are binary machines, it is convenient to represent integers over base 2 instead of 10. The **standard binary representation** for the natural numbers represents each integer as a sum of powers of 2. That is, for some $k \geq 0$ each integer n can be represented as a k-tuple $\boldsymbol{x} = (x_{k-1}, x_{k-2}, \ldots, x_1, x_0)$, where each of $x_{k-1}, x_{k-2}, \ldots, x_1, x_0$ has value 0 or 1 and n satisfies the following identity:

$$n = x_{k-1}2^{k-1} + x_{k-2}2^{k-2} + \cdots + x_1 2^1 + x_0 2^0$$

The largest integer that can be represented with k bits is $2^{k-1} + 2^{k-2} + \cdots + 2^1 + 2^0 = 2^k - 1$. (See Problem 1.1.) Also, the k-tuple representation for n is unique; that is, two different integers cannot have the same representation. When leading 0's are suppressed, the standard binary representation for 1, 15, 32, and 97 are (1), $(1,1,1,1)$, $(1,0,0,0,0,0)$, and $(1,1,0,0,0,0,1)$, respectively.

We denote with $x + y$, $x - y$, $x * y$, and x/y the results of addition, subtraction, multiplication, and division of integers.

1.2.3 Languages and Strings

An **alphabet** A is a finite set with at least two elements. A string $\boldsymbol{x}$ is an element $(a_1, a_2, \ldots, a_k)$ of the Cartesian product A^k in which we drop the commas and parentheses. Thus, we write $\boldsymbol{x} = a_1 a_2 \cdots a_k$, and say that $\boldsymbol{x}$ is a **string over the alphabet** A. A string $\boldsymbol{x}$ in A^k is said to have **length** k, denoted $|\boldsymbol{x}| = k$. Thus, 011 is a string of length three over $A = \{0, 1\}$.

Consider now the Cartesian product $A^k \times A^l = A^{k+l}$, which is the $(k+l)$-fold Cartesian product of A with itself. Let $\boldsymbol{x} = a_1 a_2 \cdots a_k \in A^k$ and $\boldsymbol{y} = b_1 b_2 \cdots b_l \in A^l$. Then a string $\boldsymbol{z} = c_1 c_2 \cdots c_{k+l} \in A^{k+l}$ can be written as the **concatenation** of strings $\boldsymbol{x}$ and $\boldsymbol{y}$ of length k and l, denoted, $\boldsymbol{z} = \boldsymbol{x} \cdot \boldsymbol{y}$, where

$$\boldsymbol{x} \cdot \boldsymbol{y} = a_1 a_2 \cdots a_k b_1 b_2 \cdots b_l$$

That is, $c_i = a_i$ for $1 \leq i \leq k$ and $c_i = b_{i-k}$ for $k + 1 \leq i \leq k + l$.

The **empty string**, denoted ϵ, is a special string with the property that when concatenated with any other string $\boldsymbol{x}$ it returns $\boldsymbol{x}$; that is, $\boldsymbol{x} \cdot \epsilon = \epsilon \cdot \boldsymbol{x} = \boldsymbol{x}$. The empty string is said to have zero length. As a special case of A^k, we let A^0 denote the **set containing the empty string**; that is, $A^0 = \{\epsilon\}$.

The **concatenation of sets of strings** A and B, denoted $A \cdot B$, is the set of strings formed by concatenating each string in A with each string in B. For example, $\{00, 1\} \cdot \{a, bb\} = \{00a, 00bb, 1a, 1bb\}$. The concatenation of a set A with the empty set $\emptyset$, denoted $A \cdot \emptyset$, is the empty set because it contains no elements; that is,

$$A \cdot \emptyset = \emptyset \cdot A = \emptyset$$

When no confusion arises, we write AB instead of $A \cdot B$.

A **language** L over an alphabet A is a collection of strings of potentially different lengths over A. For example, $\{00, 010, 1110, 1001\}$ is a **finite language** over the alphabet $\{0, 1\}$. (It is finite because it contains a bounded number of strings.) The set of all strings of all lengths over the alphabet A, including the empty string, is denoted A^* and called the **Kleene closure** of A. For example, $\{0\}^*$ contains ϵ, the empty string, as well as 0, 00, 000, 0000, ... Also, $\{00 \cup 1\}^* = \{\epsilon, 1, 00, 001, 100, 0000, \ldots\}$. It follows that a language L over the alphabet A is a subset of A^*, denoted $L \subseteq A^*$.

The **positive closure** of a set A, denoted A^+, is the set of all strings over A except for the empty string. For example, $0(0^*10^*)^+$ is the set of binary strings beginning with 0 and containing at least one 1.

1.2.4 Relations

A subset R of the Cartesian product of sets is called a **relation**. A **binary relation** R is a subset of the Cartesian product of two sets. Three examples of binary relations are $R_0 = \{(0, 0), (1, 1), (2, 4), (3, 9), (4, 16)\}$, $R_1 = \{$(red, 0), (green, 1), (blue, 2)$\}$, and $R_2 = \{$(small, short), (medium, middle), (medium, average), (large, tall)$\}$. The relation R_0 is a **function** because for each first component of a pair there is a unique second component. R_1 is also a function, but R_2 is not a function.

A binary relation R **over a set** A is a subset of $A \times A$; that is, both components of each pair are drawn from the same set. We use two notations to denote membership of a pair (a, b) in a binary relation R over A, namely $(a, b) \in R$ and the new notation aRb. Often it is more convenient to say aRb than to say $(a, b) \in R$.

A binary relation R is **reflexive** if for all $a \in A$, aRa. It is **symmetric** if for all $a, b \in A$, aRb if and only if bRa. It is **transitive** if for all $a, b, c \in A$, if aRb and bRc, then aRc.

A binary relation R is an **equivalence relation** if it is reflexive, symmetric, and transitive. For example, the pairs (a, b), $a, b \in \mathbb{N}$, for which both a and b have the same remainder on division by 3, is an equivalence relation. (See Problem 1.3.)

If R is an equivalence relation and aRb, then a and b are said to be **equivalent elements**. We let $E[a]$ be the set of elements in A that are equivalent to a under the relation R and call it the **equivalence class** of elements equivalent to a. It is not difficult to show that for all $a, b \in A$, $E[a]$ and $E[b]$ are either equal or disjoint. (See Problem 1.4.) Thus, the equivalence classes of an equivalence relation over a set A partition the elements of A into disjoint sets. For example, the partition $\{0^*, 0(0^*10^*)^+, 1(0+1)^*\}$ of the set $(0+1)^*$ of binary strings defines an equivalence relation R. The equivalence classes consist of strings containing zero or more 0's, strings starting with 0 and containing at least one 1, and strings beginning with 1. It follows that $00R000$ and $1001R11$ hold but not $10R01$.

1.2.5 Graphs

A **directed graph** $G = (V, E)$ consists of a finite set V of distinct **vertices** and a finite set of pairs of distinct vertices $E \subseteq V \times V$ called **edges**. **Edge** e **is incident on vertex** v if e contains v. A directed graph is **undirected** if for each edge (v_1, v_2) in E the edge (v_2, v_1) is also in E. Figure 1.2 shows two examples of directed graphs, some of whose vertices are labeled with symbols denoting gates, a topic discussed in Section 1.2.7. In a directed graph the edge (v_1, v_2) is directed from the vertex v_1 to the vertex v_2, shown with an arrow from v_1 to v_2. The **in-degree** of a vertex in a directed graph is the number of edges directed into it; its **out-degree** is the number of edges directed away from it; its **degree** is the sum of its in- and out-degree. In a directed graph an **input vertex** has in-degree zero, whereas an **output vertex** either has out-degree zero or is simply any vertex specially designated as an output vertex. A **walk** in a graph (directed or undirected) is a tuple of vertices $(v_1, v_2, \ldots, v_p)$ with the property that (v_i, v_{i+1}) is in E for $1 \leq i \leq p-1$. A walk $(v_1, v_2, \ldots, v_p)$ is **closed** if $v_1 = v_p$. A **path** is a walk with distinct vertices. A **cycle** is a closed walk with $p-1$ distinct vertices, $p \geq 3$. The **length of a path** is the number of edges on the path. Thus, the path $(v_1, v_2, \ldots, v_p)$ has length $p-1$. A **directed acyclic graph (DAG)** is a directed graph that has no cycles.

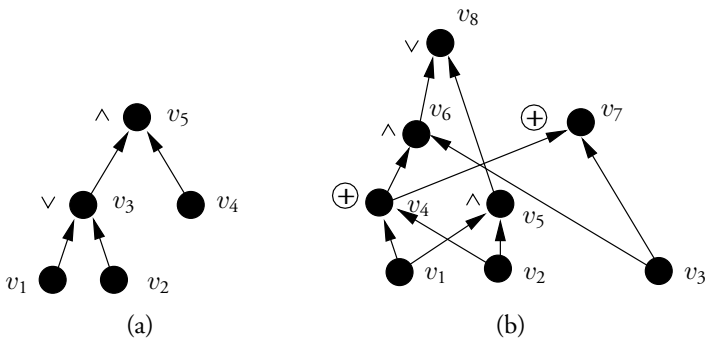

Figure 1.2 Two directed acyclic graphs representing logic circuits.

Logic circuits are DAGs in which all vertices except input vertices carry the labels of gates. Input vertices carry the labels of **Boolean variables**, variables assuming values over the set $\mathcal{B} = \{0, 1\}$. The graph of Fig. 1.2(a) is the logic circuit of Fig. 1.3(c), whereas the graph of Fig. 1.2(b) is the logic circuit of Fig. 1.4. (The figures are shown in Section 1.4.1, Logic Circuits.) The set of labels of logic gates used in a DAG is called the **basis** Ω for the DAG. The **size** of a circuit is the number of non-input vertices that it contains. Its **depth** is the length of the longest directed path from an input vertex to an output vertex.

1.2.6 Matrices

An $m \times n$ **matrix** is an array of elements containing m rows and n columns. (See Chapter 6.) The **adjacency matrix** of a graph G with n vertices is an $n \times n$ matrix whose entries are 0 or 1. The entry in the ith row and jth column is 1 if there is an edge from vertex i to vertex j and 0 otherwise. The adjacency matrix A for the graph in Fig. 1.2(a) is

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

1.2.7 Functions

The engineering component of computer science is concerned with the design, development, and testing of hardware and software. The theoretical component is concerned with questions of feasibility and optimality. For example, one might ask if there exists a program H that can determine whether an arbitrary program P on an arbitrary input I will halt or not. This is an example of an unsolvable computational problem. While it is a fascinating topic, practice often demands answers to less ethereal questions, such as "Can a particular problem be solved on a general-purpose computer with storage space S in T steps?"

To address feasibility and optimality it is important to have a precise definition of the tasks under examination. Functions serve this purpose. A **function** (or **mapping**) $f : \mathcal{D} \mapsto \mathcal{R}$ is a relation f on the Cartesian product $\mathcal{D} \times \mathcal{R}$ subject to the requirement that for each $d \in \mathcal{D}$ there is at most one pair (d, r) in f. If $(d, r) \in f$, we say that the value of f on d is r, denoted $f(d) = r$. The **domain** and **codomain** of f are $\mathcal{D}$ and $\mathcal{R}$, respectively. The sets $\mathcal{D}$ and $\mathcal{R}$ can be finite or infinite. For example, let $f_{\text{mult}} : \mathbb{N}^2 \mapsto \mathbb{N}$ of domain $\mathcal{D} = \mathbb{N}^2$ and codomain $\mathcal{R} = \mathbb{N}$ map a pair of natural numbers x and y ($\mathbb{N} = \{0, 1, 2, 3, \ldots\}$) into their product z; that is, $f(x, y) = z = x * y$. A **function** $f : \mathcal{D} \mapsto \mathcal{R}$ **is partial** if for some $d \in \mathcal{D}$ no value in $\mathcal{R}$ is assigned to $f(d)$. Otherwise, a **function is complete**.

If the domain of a function is the Cartesian product of n sets, the function is said to have n **input variables**. If the codomain of a function is the Cartesian product of m sets, the function is said to have m **output variables**. If the input variables of such a function are all drawn from the set A and the output variables are all drawn from the set B, this information is often captured by the notation $f^{(n,m)} : A^n \mapsto B^m$. However, we frequently do not use exponents or we use only one exponent to parametrize a class of problems.

A **finite function** is one whose domain and codomain are both finite sets. Finite functions can be completely defined by tables of pairs $\{(d, r)\}$, where d is an element of its domain and r is the corresponding element of its codomain.

Binary functions are complete finite functions whose domains and codomains are Cartesian products over the binary set $\mathcal{B} = \{0, 1\}$. **Boolean functions** are binary functions whose codomain is $\mathcal{B}$. The tables below define three Boolean functions on two input variables and one Boolean function on one input variable. They are called **truth tables** because the values 1 and 0 are often associated with the values **True** and **False**, respectively.

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

x	$\overline{x}$
0	1
1	0

The above tables define the AND function $x \wedge y$ (its value is **True** when x and y are **True**), the OR function $x \vee y$ (its value is **True** when either x or y or both are **True**), the EXCLUSIVE OR function $x \oplus y$ (its value is **True** only when either x or y is **True**, that is, when x is **True** and y is **False** and vice versa), and the NOT function $\overline{x}$ (its value is **True** when x is **False** and vice versa). The notation $f_{\wedge}^{(2,1)} : \mathcal{B}^2 \mapsto \mathcal{B}$, $f_{\vee}^{(2,1)} : \mathcal{B}^2 \mapsto \mathcal{B}$, $f_{\oplus}^{(2,1)} : \mathcal{B}^2 \mapsto \mathcal{B}$, $f_{\neg}^{(1,1)} : \mathcal{B} \mapsto \mathcal{B}$ for these functions makes explicit their number of input and output variables. We generally suppress the second superscript when functions are Boolean. The physical devices that compute the AND, OR, NOT, and EXCLUSIVE OR functions are called **gates**.

Many computational problems are described by functions $f : \mathcal{A}^* \mapsto \mathcal{C}^*$ from the (unbounded) set of strings over an alphabet $\mathcal{A}$ to the set of strings over a potentially different alphabet $\mathcal{C}$. Since the letters in every finite alphabet $\mathcal{A}$ can be encoded as fixed-length strings over the binary alphabet $\mathcal{B} = \{0, 1\}$, there is no loss of generality in assuming that functions are mappings $f : \mathcal{B}^* \mapsto \mathcal{B}^*$, that is, from strings over $\mathcal{B}$ to strings over $\mathcal{B}$.

Functions with unbounded domains can be used to identify languages. A language L over the alphabet $\mathcal{A}$ is uniquely determined by a **characteristic function** $f : \mathcal{A}^* \mapsto \mathcal{B}$ with the property that $L = \{\boldsymbol{x} \mid \boldsymbol{x} \in A^* \text{ such that } f(\boldsymbol{x}) = 1\}$. This statement means that L is the set of strings $\boldsymbol{x}$ in A^* for which f on them, namely $f(\boldsymbol{x})$, has value 1.

We often restrict a function $f : \mathcal{B}^* \mapsto \mathcal{B}^*$ to input strings of length n, n arbitrary. The domain of such a function is $\mathcal{B}^n$. Its codomain consists of those strings into which strings of length n map. This set may contain strings of many lengths. It is often convenient to map strings of length n to strings of a fixed length containing the same information. This can be done as follows. Let $h(n)$ be the length of a longest string that is the value of an input string of length n. Encode letters in $\mathcal{B}$ by repeating them (replace 0 by 00 and 1 by 11) and then add as a prefix as many instances of 01 as necessary to insure that each string in the codomain of f_n has $2h(n)$ characters. For example, if $h(4) = 3$ and $f(0110) = 10$, encode the value 10 as 011100. This encoding provides a function $f_n : \mathcal{B}^n \mapsto \mathcal{B}^{2h(n)}$ containing all the information that is in the original version of f_n.

It is often useful to work with functions $f : \mathbb{R} \mapsto \mathbb{R}$ whose domains and codomains are real numbers $\mathbb{R}$. Functions of this type include linear functions, polynomials, exponentials, and logarithms. A **polynomial** $p(x) : \mathbb{R} \mapsto \mathbb{R}$ of degree $k - 1$ in the variable x is specified by a set of k real coefficients, $c_{k-1}, \ldots, c_1, c_0$, where $p(x) = c_{k-1}x^{k-1} + \cdots + c_1x^1 + c_0$.

A **linear function** is a polynomial of degree 1. An **exponential function** is a function of the form $E(x) = a^x$ for some real a – for example, $2^{1.5} = 2.8284271\ldots$. The **logarithm** to the base a of b, denoted $\log_a b$, is the value of x such that $a^x = b$. For example, the logarithm to base 2 of $2.8284271\ldots$ is 1.5 and the logarithm to base 10 of 100 is 2. A function $f(x)$ is **polylogarithmic** if for some polynomial $p(x)$ we can write $f(x)$ as $p(\log_2 x)$; that is, it is a polynomial in the logarithm of x.

Two other functions used often in this book are the floor and ceiling functions. Their domains are the reals, but their codomains are the integers. The **ceiling function**, denoted $\lceil x \rceil : \mathbb{R} \mapsto \mathbb{Z}$, maps the real x to the smallest integer greater or equal to it. The **floor function**, denoted $\lfloor x \rfloor : \mathbb{R} \mapsto \mathbb{Z}$, maps the real x to the largest integer less than or equal to it. Thus, $\lceil 3.5 \rceil = 4$ and $\lceil 15.0001 \rceil = 16$. Similarly, $\lfloor 3.5 \rfloor = 3$ and $\lfloor 15.0001 \rfloor = 15$. The following bounds apply to the floor and ceiling functions.

$$f(x) - 1 \leq \lfloor f(x) \rfloor \leq f(x)$$
$$f(x) \leq \lceil f(x) \rceil \leq f(x) + 1$$

As an example of the application of the ceiling function we note that $\lceil \log_2 n \rceil$ is the number of bits necessary to represent the integer n.

1.2.8 Rate of Growth of Functions

Throughout this book we derive mathematical expressions for quantities such as space, time, and circuit size. Generally these expressions describe functions $f : \mathbb{N} \mapsto \mathbb{R}$ from the non-negative integers to the reals, such as the functions $f_1(n)$ and $f_2(n)$ defined as

$$f_1(n) = 4.5n^2 + 3n$$
$$f_2(n) = 3^n + 4.5n^2$$

When n is large we often wish to simplify expressions such as these to make explicit their dominant or most rapidly growing term. For example, for large values of n the dominant terms in $f_1(n)$ and $f_2(n)$ are $4.5n^2$ and 3^n respectively, as we show. A term dominates when n is large if the value of the function is approximately the value of this term, that is, if the function is within some multiplicative factor of the term.

To highlight dominant terms we introduce the **big Oh**, **big Omega** and **big Theta** notation. They are defined for functions whose domains and codomains are the integers or the reals.

DEFINITION 1.2.1 *Let $f : \mathbb{R} \mapsto \mathbb{R}$ and $g : \mathbb{R} \mapsto \mathbb{R}$ be two functions whose domains and codomains are either the integers or the reals. If there are positive constants x_0 and $K > 0$ such that for all $|x| \geq x_0$,*

$$|f(x)| \leq K\,|g(x)|$$

we write

$$f(x) = O(g(x))$$

and say that "$f(x)$ is **big Oh of** *$g(x)$" or it grows no more rapidly in x than $g(x)$. Under the same conditions we also write*

$$g(x) = \Omega(f(x))$$

and say that "$g(x)$ is **big Omega of** *$f(x)$" or that it grows at least as rapidly in x as $f(x)$.*

If $f(x) = O(g(x))$ and $g(x) = O(f(x))$, we write

$$f(x) = \Theta(g(x)) \text{ or } g(x) = \Theta(f(x))$$

and say that "$f(x)$ is **big Theta of** *$g(x)$" and "$g(x)$ is* **big Theta of** *$f(x)$" or that the two functions have the same rate of growth in x.*

The **big Oh** notation is illustrated by the expressions for $f_1(n)$ and $f_2(n)$ above.

EXAMPLE 1.2.1 *We show that $f_1(n) = 4.5n^2 + 3n$ is $O(n^k)$ for any $k \geq 2$; that is, $f_1(n)$ grows no more rapidly than n^k for $k \geq 2$. We also show that $n^k = O(f_1(n))$ for $k \leq 2$; that is, that n^k grows no more rapidly than $f_1(n)$ for $k \leq 2$. From the above definitions it follows that $f_1(n) = \Theta(n^2)$; that is, $f_1(n)$ and n^2 have the same rate of growth. We say that $f_1(n)$ is a* **quadratic function** *in n.*

To prove the first statement, we need to exhibit a natural number n_0 and a constant $K_0 > 0$ such that for all $n \geq n_0$, $f_1(n) \leq K_0 n^k$. If we can show that $f_1(n) \leq K_0 n^2$, then we have shown $f_1(n) \leq K_0 n^k$ for all $k \geq 2$. To show the former, we must show the following for some $K_0 > 0$ and for all $n \geq n_0$:

$$4.5n^2 + 3n \leq K_0 n^2$$

We try $K_0 = 5.5$ and find that the above inequality is equivalent to $3n \leq n^2$ or $3 \leq n$. Thus, we can choose $n_0 = 3$ and we are done.

To prove the second statement, namely, that $n^k = O(f_1(n))$ for $k \leq 2$, we must exhibit a natural number n_1 and some $K_1 > 0$ such that for all $n \geq n_1$, $n^k \leq K_2 f_1(n)$. If we can show that $n^2 \leq K_1 f_1(n)$, then we have shown $n^k \leq K_2 f_1(n)$. To show the former, we must show the following for some $K_1 > 0$ and for all $n \geq n_1$:

$$n^2 \leq K_1(4.5n^2 + 3n)$$

Clearly, if $K_1 = 1/4.5$ the inequality holds for $n \geq 0$, since $3K_1 n$ is positive. Thus, we choose $n_1 = 0$ and we are done.

EXAMPLE 1.2.2 *We now show that the slightly more complex function $f_2(n) = 3^n + 4.5n^2$ grows as 3^n; that is, $f_2(n) = \Theta(3^n)$, an exponential function in n. Because $3^n \leq f_2(n)$ for all $n \geq 0$, it follows that $3^n = O(f_2(n))$. To show that $f_2(n) = O(3^n)$, we demonstrate that $f_2(n) \leq 2(3^n)$ holds for $n \geq 4$. This is equivalent to the following inequality:*

$$4.5n^2 \leq 3^n$$

To prove this holds, we show that $h(n) = 3^n/n^2$ is an increasing function of n for $n \geq 2$ and that $h(4) \geq 4.5$. To show that $h(n)$ is an increasing function of n, we compute the ratio $r(n) = h(n+1)/h(n)$ and show that $r(n) \geq 1$ for $n \geq 2$. But $r(n) = 3n^2/(n+1)^2$ and $r(n) \geq 1$ when $3n^2 \geq (n+1)^2$ or when $n(n-1) \geq 1/2$, which holds for $n \geq 2$. Since $h(3) = 3$ and $h(4) = 81/16 > 5$, the desired conclusion follows.

1.3 Methods of Proof

In this section we briefly introduce several methods of proof that are used in this book, namely, proof by induction, proof by contradiction, and the pigeonhole principle. In the previous

section we saw proof by reduction: in each step the condition to be established was translated into another condition until a condition was found that was shown to be true.

Proofs by induction use **predicates**, that is, functions of the kind $P : \mathbb{N} \mapsto \mathcal{B}$. The truth value of the predicate $P : \mathbb{N} \mapsto \mathcal{B}$ on the natural number n, denoted $P(n)$, is 1 or 0 depending on whether or not the predicate is **True** or **False**.

Proofs by induction are used to prove statements of the kind, "For all natural numbers n, predicate (or property) P is true." Consider the function $S_1 : \mathbb{N} \mapsto \mathbb{N}$ defined by the following sum:

$$S_1(n) = \sum_{j=1}^{n} j \tag{1.1}$$

We use induction to prove that $S_1(n) = n(n+1)/2$ is true for each $n \in \mathbb{N}$.

DEFINITION 1.3.1 *A* **proof by induction** *has a predicate P, a* **basis step**, *an* **induction hypothesis**, *and an* **inductive step**. *The basis establishes that $P(k)$ is true for integer k. The induction hypothesis assumes that for some fixed but arbitrary natural number $n \geq k$, the statements $P(k), P(k+1), \ldots, P(n)$ are true. The inductive step is a proof that $P(n+1)$ is true given the induction hypothesis.*

It follows from this definition that a proof by induction with the predicate P establishes that P is true for all natural numbers larger than or equal to k because the inductive step establishes the truth of $P(n+1)$ for arbitrary integer n greater than or equal to k. Also, induction may be used to show that a predicate holds for a subset of the natural numbers. For example, the hypothesis that every even natural number is divisible by 2 is one that would be defined only on the even numbers.

The following proof by induction shows that $S_1(n) = n(n+1)/2$ for $n \geq 0$.

LEMMA 1.3.1 *For all $n \geq 0$, $S_1(n) = n(n+1)/2$.*

Proof PREDICATE: The value of the predicate P on n, $P(n)$, is **True** if $S_1(n) = n(n+1)/2$ and **False** otherwise.

BASIS STEP: Clearly, $S_1(0) = 0$ from both the sum and the closed form given above.

INDUCTION HYPOTHESIS: $S_1(k) = k(k+1)/2$ for $k = 0, 1, 2, \ldots, n$.

INDUCTIVE STEP: By the definition of the sum for S_1 given in (1.1), $S_1(n+1) = S_1(n) + n + 1$. Thus, it follows that $S_1(n+1) = n(n+1)/2 + n + 1$. Factoring out $n+1$ and rewriting the expression, we have that $S_1(n+1) = (n+1)((n+1)+1)/2$, exactly the desired form. Thus, the statement of the theorem follows for all values of n. ■

We now define proof by contradiction.

DEFINITION 1.3.2 *A* **proof by contradiction** *has a predicate P. The complement $\neg P$ of P is shown to be* **False**, *which implies that P is* **True**.

The examples shown earlier of strings in the language $L = \{00 \cup 1\}^*$ suggest that L contains only strings other than ϵ with an odd number of 1's. Let P be the predicate "L contains strings other than ϵ with an even number of 1's." We show that it is true by assuming

it is false, namely, by assuming "L contains only strings with an odd number of 1's" and showing that this statement is false. In particular, we show that L contains the string 11. From the definition of the Kleene closure, L contains strings of all lengths in the "letters" 00 and 1. Thus, it contains a string containing two instances of 1 and the predicate P is true.

Induction and proof by contradiction can also be used to establish the pigeonhole principle. The **pigeonhole principle** states that if there are n pigeonholes, $n + 1$ or more pigeons, and every pigeon occupies a hole, then some hole must have at least two pigeons. We reformulate the principle as follows:

LEMMA 1.3.2 *Given two finite sets A and B with $|A| > |B|$, there does not exist a* **naming function** *$\nu : A \mapsto B$ that gives to each element a in A a* **name** *$\nu(a)$ in B such that every element in A has a unique name.*

Proof BASIS: $|B| = 1$. To show that the statement is **True**, assume it is **False** and show that a contradiction occurs. If it is **False**, every element in A can be given a unique name. However, since there is one name (the one element of B) and more than one element in A, we have a contradiction.

INDUCTION HYPOTHESIS: There is no naming function $\nu : A \mapsto B$ when $|B| \leq n$ and $|A| > |B|$.

INDUCTIVE STEP: When $|B| = n+1$ and $|A| > |B|$ we show there is no naming function $\nu : A \mapsto B$. Consider an element $b \in B$. If two elements of A have the name b, the desired conclusion holds. If not, remove b from B, giving the set B', and remove from A the element, if any, whose name is b, giving the set A'. Since $|A'| > |B'|$ and $|B'| \leq n$, by the induction hypothesis, there is no naming function obtained by restricting ν to A'. Thus, the desired conclusion holds. ■

1.4 Computational Models

A variety of computer models are examined in this book. In this section we give the reader a taste of five models, the logic circuit, the finite-state machine, the random-access machine, the pushdown automaton, and the Turing machine. We also briefly survey the problem of language recognition.

1.4.1 Logic Circuits

A **logic gate** is a physical device that realizes a Boolean function. A **logic circuit**, as defined in Section 1.2, is a directed acyclic graph in which all vertices except input vertices carry the labels of gates.

Logic gates can be constructed in many different technologies. To make ideas concrete, Fig. 1.3(a) and (b) show electrical circuits for the AND and OR gates constructed with batteries, bulbs, and switches. Shown with each of these circuits is a logic symbol for the gate. These symbols are used to draw circuits, such as the circuit of Fig. 1.3(c) for the function $(x \vee y) \wedge z$. When electrical current flows out of the batteries through a switch or switches in these circuits, the bulbs are lit. In this case we say the value of the circuit is **True**; otherwise it is **False**. Shown below is the truth table for the function mapping the values of the three input variables of the circuit in Fig. 1.3(c) to the value of the one output variable. Here x, y, and z have value 1

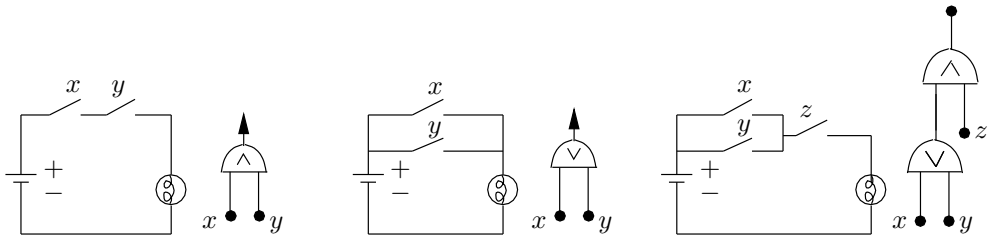

Figure 1.3 Three electrical circuits simulating logic circuits.

when the switch that carries its name is closed; otherwise they have value 0.

x	y	z	$(x \vee y) \wedge z$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Today's computers use transistor circuits instead of the electrical circuits of Fig. 1.3.

Logic circuits execute **straight-line programs**, programs containing only assignment statements. Thus, they have no loops or branches. (They may have loops if the number of times a loop is executed is fixed.) This point is illustrated by the "full-adder" circuit of Fig. 1.4, a circuit discussed at length in Section 2.7. Each external input and each gate is assigned a unique integer. Each is also assigned a variable whose value is the value of the external input or gate. The ith vertex is assigned the variable x_i. If x_i is associated with a gate that combines the results produced at the jth and kth gates with the operator $\odot$, we write an **assignment operation** of the form $x_i := x_j \odot x_k$. The sequence of assignment operations for a circuit is a straight-line program. Below is a straight-line program for the circuit of Fig. 1.4:

$$
\begin{aligned}
x_4 &:= x_1 \oplus x_2 \\
x_5 &:= x_4 \wedge x_3 \\
x_6 &:= x_1 \wedge x_2 \\
x_7 &:= x_4 \oplus x_3 \\
x_8 &:= x_5 \vee x_6
\end{aligned}
$$

The values computed for (x_8, x_7) are the standard binary representation for the number of 1's among x_1, x_2, and x_3. This can be seen by constructing a table of values for x_1, x_2, x_3, x_7,

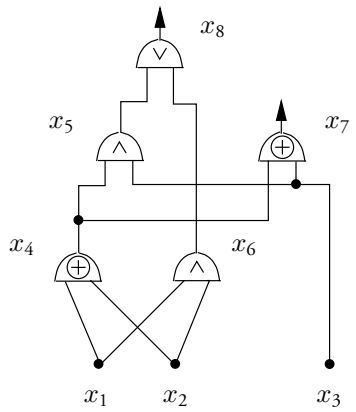

Figure 1.4 A full-adder circuit. Its output pair (x_8, x_7) is the standard binary representation for the number of 1's among its three inputs x_1, x_2, and x_3.

and x_8. Full-adder circuits can be combined to construct an adder for binary numbers. (In Section 2.2 we give another notation for straight-line programs.)

As shown in the truth table for Fig. 1.3(c), each logic circuit has associated with it a binary function that maps the values of its input variables to the values of its output variables. In the case of the full-adder, since x_8 and x_7 are its output variables, we associate with it the function $f_{\mathrm{FA}}^{(3,2)} : \mathcal{B}^3 \mapsto \mathcal{B}^2$, whose value is $f_{\mathrm{FA}}^{(3,2)}(x_1, x_2, x_3) = (x_8, x_7)$.

Algebraic circuits are similar to logic circuits except they may use operations over non-binary sets, such as addition and multiplication over a ring, a concept explained in Section 6.2.1. Algebraic circuits are the subject of Chapter 6. They are also described by DAGs and they execute straight-line programs where the operators are non-binary functions. Algebraic circuits also have associated with them functions that map the values of inputs to the values of outputs.

Logic circuits are the basic building blocks of all digital computers today. When such circuits are combined with binary memory cells, machines with memory can be constructed. The models for these machines are called finite-state machines.

1.4.2 Finite-State Machines

The **finite-state machine** (FSM) is a machine with memory. It executes a series of steps during each of which it takes its current state from the set Q of states and current external input from the set Σ of input letters and combines them in a logic circuit L to produce a successor state in Q and an output letter in Ψ, as suggested in Fig. 1.5. The logic circuit L can be viewed as having two parts, one that computes the **next-state function** $\delta : Q \times \Sigma \mapsto Q$, whose value is the next state of the FSM, and the other that computes the **output function** $\lambda : Q \mapsto \Psi$, whose value is the output of the FSM in the current state. A generic finite-state machine is shown in Fig. 1.5(a) along with a concrete FSM in Fig. 1.5(b) that provides as successor state and output the EXCLUSIVE OR of the current state and the external input. The **state diagram** of the FSM in Fig. 1.5(b) is shown in Fig. 1.8. Two (or more) finite-state machines that operate

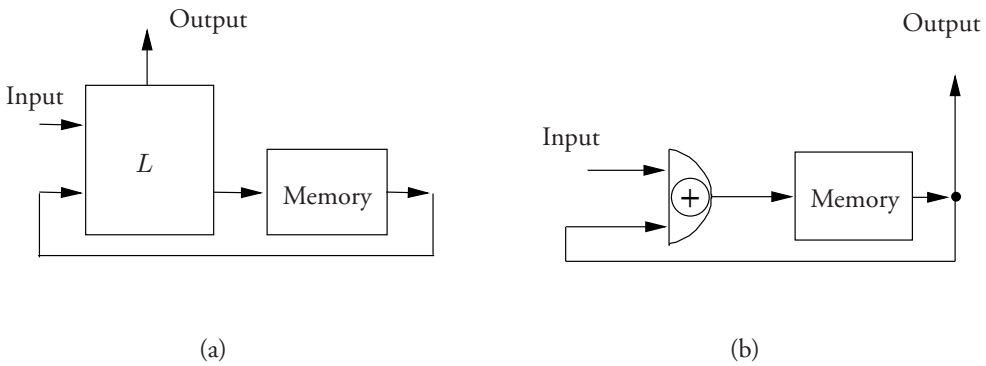

Figure 1.5 (a) The finite-state machine (FSM) model; at each unit of time its logic unit, L, operates on its current state (taken from its memory) and its current external input to compute an external output and a new state that it stores in its memory. (b) An FSM that holds in its memory a bit that is the EXCLUSIVE OR of the initial value stored in its memory and the external inputs received to the present time.

in lockstep can be interconnected to form a single FSM. In this case, some outputs of one FSM serve as inputs to the other.

Finite-state machines are ubiquitous today. They are found in microwave ovens, VCRs and automobiles. They can be simple or complex. One of the most useful FSMs is the general-purpose computer modeled by the random-access machine.

1.4.3 Random-Access Machine

The (bounded-memory) **random-access machine** (RAM) is modeled as a pair of interconnected finite-state machines, one a **central processing unit** (CPU) and the other a **random-access memory**, as suggested in Fig. 1.6. The random-access memory holds m b-bit words, each identified by an address. It also holds an output word (out_wrd) and a triple of inputs

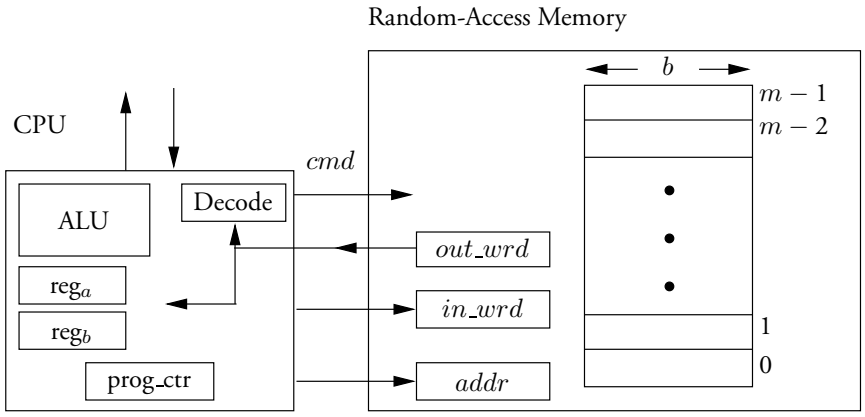

Figure 1.6 The bounded-memory random-access machine.

consisting of a command (cmd), an address ($addr$), and an input data word (in_wrd). cmd is either READ, WRITE, or NO-OP. A NO-OP command does nothing whereas a READ command changes the value of out_wrd to the value of the data word at address $addr$. A WRITE command replaces the data word at address $addr$ with the value of in_wrd.

The random-access memory holds data as well as **programs**, collections of instructions for the CPU. The CPU executes the **fetch-and-execute cycle** in which it repeatedly reads an instruction from the random-access memory and executes it. Its instructions typically include arithmetic, logic, comparison, and jump instructions. Comparisons are used to decide whether the CPU reads the next program instruction in sequence or jumps to an instruction out of sequence.

The general-purpose computer is much more complex than suggested by the above brief sketch of the RAM. It uses a rich variety of methods to achieve high speed at low cost with the available technology. For example, as the number of components that can fit on a semiconductor chip increases, designers have begun to use "super-scalar" CPUs, CPUs that issue multiple instructions in each time step. Also, memory hierarchies are becoming more prevalent as designers assemble collections of slower but larger memories with lower costs per bit to simulate expensive fast memories.

1.4.4 Other Models

There are many other models of computers with memory, some of which have an infinite supply of data words, such as the **Turing machine**, a machine consisting of a **control unit** (an FSM) and a **tape unit** that has a potentially infinite linear array of cells each containing letters from an alphabet that can be read and written by a **tape head** directed by the control unit. It is assumed that in each time step the head may move only from one cell to an adjacent one on the linear array. (See Fig. 1.7.) The Turing machine is a standard model of computation since no other machine model has been discovered that performs tasks it cannot perform.

The **pushdown automaton** is a restricted form of Turing machine in which the tape is used as a pushdown stack. Data is entered, deleted, and accessed only at the top of a stack. A

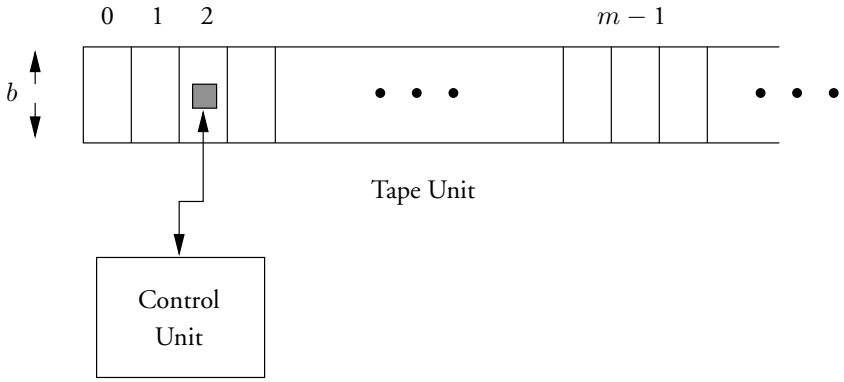

Figure 1.7 The Turing machine has a control unit that is a finite-state machine and a tape unit that controls reading and writing by a tape head and the movement of the tape head one cell at a time to the left or right of the current position.

pushdown stack can be simulated by a tape in which the cell to the right of the tape head is always blank. If the tape moves right from a cell, it writes a non-blank symbol in the cell. If it moves left, it writes a blank in that cell before leaving it.

Some computers are serial: they execute one operation on a fixed amount of data per time step. Others are parallel; that is, they have multiple (usually communicating) subcomputers that operate simultaneously. They may operate synchronously or asynchronously and they may be connected via a simple or a complex network. An example of a simple network is a wire between two computers. An example of a complex network is a crossbar switch consisting of 25 switches at the intersection of five columns and five rows of wires; closing the switch at the intersection of a row and a column connects the two wires and the two computers to which they are attached.

We close this section by emphasizing the importance of models of computers. Good models provide a level of abstraction at which important facts and insights can be developed without losing so much detail that the results are irrelevant to practice.

1.4.5 Formal Languages

In Chapters 4 and 5 the finite-state machine, pushdown automaton, and Turing machine are characterized by their language recognition capability. Formal methods for specifying languages have led to efficient ways to parse and recognize programming languages. This is illustrated by the finite-state machine of Fig. 1.8. Its initial state is q_0, its final state is q_1 and its inputs can assume values 0 or 1. An output of 0 is produced when the machine is in state q_0 and an output of 1 is produced when it is in state q_1. The output before the first input is received is 0.

After the first input the output of the FSM of Fig. 1.8 is equal to the input. After multiple inputs the output is the EXCLUSIVE OR of the 1's and 0's among the inputs, as we show by induction. The inductive hypothesis is clearly true after one input. Suppose it is true after k inputs; we show that it remains true after $k+1$ inputs, and therefore for all inputs. The output uniquely determines the state. There are two cases to consider: after k inputs either the FSM is in state q_0 or it is in state q_1. For each state, there are two cases to consider based on the value of the $k+1$st input. In all four cases it is easy to see that after the $k+1$st input the output is the EXCLUSIVE OR of the first $k+1$ inputs.

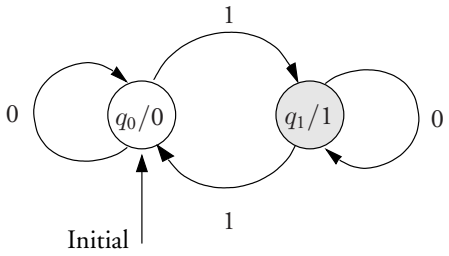

Figure 1.8 A state diagram for a finite-state machine whose circuit model is given in Fig. 1.5(b). q_0 is the initial state of the machine and q_1 is its final state. If the machine is in q_0, it has received an even number of 1 inputs, whereas if it is in q_1, it has received an odd number of 1's.

The **language recognized** by an FSM is defined in two ways. It is the set of input strings that cause the FSM to produce a particular letter as its last output or to enter one of the set of final states on its last input. Thus, the FSM of Fig. 1.8 recognizes the set of binary strings containing an odd number of 1's. It also recognizes the set of binary strings containing an even number of 1's because they result in a last output of 0.

An FSM can also **compute a function**. The most general function that it computes in T steps is the function $f_M^{(T)} : Q \times \Sigma^T \mapsto Q \times \Psi^T$ that maps the initial state s and the T inputs $w_1, w_2, \ldots, w_T$ to the T outputs $y_1, y_2, \ldots, y_T$. It can also compute any other function obtained by ignoring some outputs or fixing either the initial state or some inputs or both.

The class of languages recognized by finite-state machines (the **regular languages**) is not rich enough to describe easily the important programming languages that are in use today. As a consequence, other languages, such as the context-free languages, are employed. **Context-free languages** (which include the regular languages) require computers with potentially unbounded storage for their recognition. The class of computers that recognizes exactly the context-free languages are the nondeterministic pushdown automata, pushdown automata in which the control unit is nondeterministic; that is, some of its states can have multiple potential successor states.

The strings in regular and context-free languages (and other languages as well) can be generated by grammars. A context-free grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$ consists of sets of **terminal** and **non-terminal symbols**, $\mathcal{T}$ and $\mathcal{N}$ respectively, and rules $\mathcal{R}$ by which each non-terminal is replaced with one or more strings of terminals and non-terminals. All string generations start with the special **start** non-terminal S. The language generated by G, $L(G)$, contains the strings of terminal characters produced by rewriting strings in this fashion. This is illustrated by the context-free grammar G with two rules shown below.

EXAMPLE 1.4.1 *$G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$, where $\mathcal{N} = \{\text{S}\}$, $\mathcal{T} = \{a, b\}$, and $\mathcal{R}$ consists of the two rules*

$$(a) \quad \text{S} \rightarrow a\text{S}b \qquad (b) \quad \text{S} \rightarrow ab$$

Each application of a rule derives another string, as shown below. This grammar has only two derivations, namely S → aSb and S → ab. The second derivation is always the last to be used. (Recall that the language $L(G)$ contains only terminal strings.)

$$\begin{aligned} \text{S} &\rightarrow a\text{S}b \\ &\rightarrow aa\text{S}bb \\ &\rightarrow aaa\text{S}bbb \\ &\rightarrow aaaabbbb \end{aligned}$$

As can be seen by inspection, the only strings in $L(G)$ are of the form $a^k b^k$, where a^k denotes the letter a repeated k times. Thus, $L(G) = \{a^k b^k \mid k \geq 1\}$.

Once a grammar for a regular or context-free language is known, it is possible to **parse** a string in the language. In the above example this amounts to determining the number of times that the first rule is applied.

To develop some intuition for the use of the pushdown automaton as a recognizer for context-free languages, observe that we can determine the number of applications of the first rule in this language by pushing each instance of a onto a stack and then popping a's as b's are

encountered. The number of a's can then be matched with the number of b's and if they are not equal, the string is declared not in the language. If equal, the number of instances of the first rule is determined.

Programming languages contain strings of characters and digits representing names and the values of variables. Such strings can typically be scanned with finite-state machines. Once scanned, these strings can be assigned tokens that are then used in a later parsing phase, which today is typically based on a generalization of parsing for context-free languages.

1.5 Computational Complexity

Computational complexity is examined in concrete and abstract terms. The concrete analysis of computational limits is done using models that capture the exchange of space for time. It also is done via the study of circuit complexity, the minimal size and depth of circuits for functions. Computational complexity is studied abstractly via complexity classes, the classification of languages by the time and/or space they need.

1.5.1 A Computational Inequality

Computational inequalities play an important role in this book. We now sketch the derivation of a computational inequality for the finite-state machine and specialize it to the RAM. The idea is very simple: we simulate with a circuit the computation of a function f by an FSM and then compare the size of the circuit produced with the size of the smallest circuit for f. **Simulation**, which we use to derive this result, is a central idea in theoretical computer science. For example, it is used to show that a problem is **NP**-complete. We use it here to relate the resources available to compute a function f with an FSM to the inherent complexity of f.

Shown in Fig. 1.5(a) is the standard model for an FSM. As suggested, a circuit L combines the current state held in the memory M together with an external input to form an external output and a successor state which is held in M. If the input, output, and state are represented as binary tuples, the circuit L can be realized by a logic circuit with Boolean gates. Let the FSM compute the function $f : \mathcal{B}^n \mapsto \mathcal{B}^m$ in T steps; that is, its state and/or T external inputs contain the n Boolean inputs to f and its T outputs contain the m Boolean outputs of f. (The inputs and outputs must appear in the same positions on each computation to prevent the application of hidden computational resources.)

The function f can also be computed by the circuit shown in Fig. 1.9, which is obtained by unwinding the loop of Fig. 1.5(a) using T copies of the logic circuit L for the FSM. This

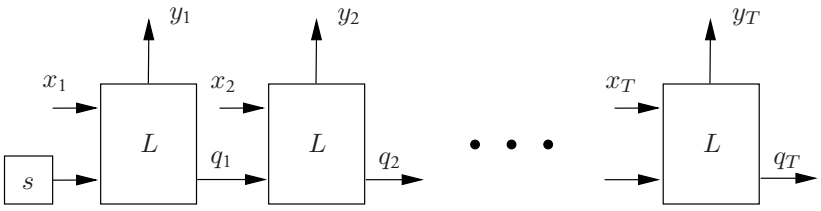

Figure 1.9 A circuit that computes the same function as an FSM (see Fig. 1.5(a)) in T steps. It has the same initial state s, receives the same inputs and produces the same outputs.

follows because the inputs $x_1, x_2, \ldots, x_T$ that would be given to the FSM over time can be given simultaneously to this circuit and it will produce the T outputs that would be produced by the FSM. This circuit has $T \cdot C(L)$ gates, where $C(L)$ is the actual or equivalent number of gates used to realize L. (The circuit L may be realized with a technology that does not formally use gates.) Since this circuit is not necessarily the smallest circuit for the function f, we have the following inequality, where $C(f)$ is the size of the smallest circuit for f:

$$C(f) \leq T \cdot C(L)$$

This result is important because it imposes a constraint on every computation done by a sequential machine. This inequality has two interpretations. First, if the product $T \cdot C(L)$ (the **equivalent number of logic operations employed**) of the number of time steps T and the equivalent number of logic operations $C(L)$ per step is too small, namely, less than $C(f)$, the FSM cannot compute function f because the above inequality would be violated. This is a form of **impossibility theorem for bounded computations.** Second, a complex function, one for which $C(f)$ is large, requires a large value for the product $T \cdot C(L)$. In light of the first interpretation of $T \cdot C(L)$ as the equivalent number of logic operations employed, it makes sense to call $W = T \cdot C(L)$ the **computational work** done by the FSM to compute f.

The above computational inequality can be specialized to the bounded-memory RAM with S bits of memory. When S is large, as it usually is, $C(L)$ for the RAM is proportional to S. As a consequence, for the RAM we have the following computational inequality for some positive constant κ:

$$C(f) \leq \kappa ST$$

This inequality shows the central role of circuit complexity in theoretical computer science. It also demonstrates that the space-time product, ST, is an important measure of the complexity of a problem. Functions with large circuit size can be computed by a RAM only if it either has a large storage capacity or executes many time steps or both. Similar results exist for the Turing machine.

1.5.2 Tradeoffs in Space, Time, and I/O Operations

Computational inequalities of the kind sketched above are important but often difficult to apply because it is hard to show that functions have a large circuit size. For this reason space-time tradeoffs have been studied under the assumption that the type of algorithm or program allowed is restricted. For example, if only straight-line programs are considered, then the *pebble game* sketched below and discussed in detail in Chapter 10 can be used to derive tradeoff inequalities.

The standard **pebble game** is played on a directed acyclic graph (DAG), the graph of a straight-line program. The input vertices of a DAG have no edges directed into them. Output vertices have no edges directed away from them. Internal vertices are non-input vertices. A predecessor of a vertex v is a vertex u that has an edge directed to v. The pebble game is played with pebbles that are placed on vertices according to the following rules:

- Initially no vertices carry pebbles.
- A pebble can be placed on an input vertex at any time.

- A pebble can be placed on an internal vertex only if all of its predecessor vertices carry pebbles.
- The pebble moved to a vertex can be a pebble residing on one of its immediate predecessors.
- A pebble can be removed from a vertex at any time.
- Every output vertex must carry a pebble at some time.

Space S in this game is the maximum number of pebbles used to play the game on a DAG. Time T is the number of times that pebbles are placed on vertices. If enough pebbles are available to play the game, each vertex is pebbled once and T is the number of vertices in the graph. If, however, there are not enough pebbles, some vertices will have to be pebbled more than once. In this case a tradeoff between space and time will be exhibited.

For a particular DAG G we may seek to determine the minimum number of pebbles, $S_{\min}$, needed to place pebbles on all output vertices at some time and for a given number of pebbles S to determine the minimum time T needed when S pebbles are used. Methods for computing $S_{\min}$ and bounding S and T simultaneously have been developed. For example, the four-point (four-input) fast Fourier transform (FFT) graph shown in Fig. 1.10 has $S_{\min} = 3$ and can be pebbled in the minimum number of steps with five pebbles.

Let the FFT graph of Fig. 1.10 be pebbled with the minimum number S of pebbles. Initially no pebbles reside on the graph. Thus, there is a first point in time at which S pebbles reside on the graph. The dark gray vertices identify one possible placement of pebbles at such a point in time. The light gray vertices will have had pebbles placed on them prior to this time and will have to be repebbled again later to pebble output vertices that cannot be reached from the placement of the dark gray vertices. This demonstrates that for this graph if the minimum number of pebbles is used, some vertices will have to be repebbled. Although the n-point FFT graph, n a power of two, has only $n \log n + n$ vertices, we show in Section 10.5.5 that its vertices must be repebbled enough times that S and T satisfy $(S+1)T \geq n^2/16$. Thus, either S is much larger than the minimum space or T is much larger than the number of vertices or both.

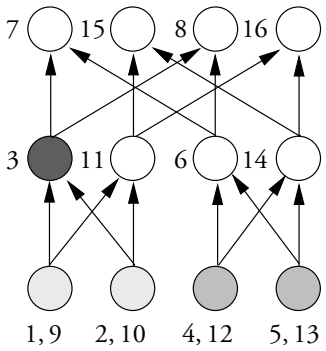

Figure 1.10 A pebbling of a four-input FFT graph at the point at which the maximum number of pebbles (three) is used. Numbers specify the order in which vertices can be pebbled. A maximum of three pebbles is used. Some vertices are pebbled twice.

Space-time tradeoffs can also be studied with the branching program, a type of program that permits data-dependent computations. (See Section 10.9.) While branching programs provide more flexibility than does the pebble game, they are worth considering only for problems in which the algorithms used involve branching and have access to an external random-access memory to permit data-dependent reading of inputs, a strong assumption. For many problems only straight-line programs are used, in which case the pebble game is the model of choice.

A serious problem arises when the storage capacity of a primary memory is too small for a problem, so that a slow secondary memory, such as a disk, must be used for intermediate storage. This results in time-consuming **input/output operations (I/O)** between primary and secondary memory. If too many I/O operations are done, the overall performance of the system can deteriorate markedly. This problem has been exacerbated by the growing disparity between the speed of CPUs and that of memories; the speed of CPUs is increasing over time at a greater rate than that of memories. In fact, the **latency** of a disk, the time between the issuance of a request for data and the time it is answered, can be 100,000 to 1,000,000 times the length of a CPU cycle. As a consequence, the amount of time spent swapping data between primary and secondary memory may dominate the time to perform computations. A second pebble game, the **red-blue pebble game**, has been introduced to study this problem. (See Chapter 11.)

The red-blue pebble game is played with both red and blue pebbles. The (hot) red pebbles correspond to primary memory locations and the (cool) blue pebbles correspond to secondary memory locations. Red pebbles are played according to the rules of the above pebble game. The additional rules that apply to the red and blue pebbles allow a red pebble to be swapped for a blue one and vice versa. In addition, blue pebbles reside only on inputs initially and must reside on outputs finally. The number of red pebbles is limited, but the number of blue pebbles is not.

The goal of the red-blue pebble game is to minimize the number of times that red and blue pebbles are swapped, since each swap corresponds to an expensive input/output (I/O) operation. Let T be the number of I/O operations and S be the number of red pebbles. Upper and lower bounds on the exchange of S for T have been derived for a large number of problems. For example, for the problem of multiplying two $n \times n$ matrices in about $2n^3$ steps with the classical algorithm, it has been shown that a red-blue pebble-game strategy leads to a product ST^2 proportional to n^6 and that this cannot be beaten except by a small multiplicative factor.

1.5.3 Complexity Classes

Complexity classes provide a way to group languages of similar computational complexity. For example, the **nondeterministic polynomial-time languages** (**NP**) are languages that can be solved in time that is polynomial in the size of their input when the machine in question is a nondeterministic Turing machine (TM). Nondeterministic Turing machines can have more than one state that is a successor to the current state for the current input. Thus, they can make choices between successor states. A language L is in **NP** if there is a nondeterministic TM such that, given an arbitrary string in L, there is some choice of successor states for the TM control unit that causes the TM to enter an accepting state in a number of steps that is polynomial in the length of the input.

An **NP**-complete language L_0 must satisfy two conditions. First, L_0 must be in **NP** and second, it must be true that for each language L in **NP** a string $\boldsymbol{x}$ in L can be translated

into a string $\boldsymbol{y}$ of L_0 using an algorithm whose running time is a polynomial in the length of $\boldsymbol{x}$ such that $\boldsymbol{y}$ is in L_0 if and only if $\boldsymbol{x}$ is in L. As a consequence of this definition, if any **NP**-complete language can be solved in deterministic polynomial time, then every language in **NP** can, including all the other **NP**-complete languages. However, the best algorithms known today for **NP**-complete languages all have exponential running time. Thus, for long strings these algorithms are impractical. If solutions to large **NP**-complete languages are needed, we are limited to approximate solutions.

1.5.4 Circuit Complexity

Circuit complexity is a notoriously difficult subject. Despite decades of research, we have failed to find methods to show that individual functions have super-polynomial circuit size or more than poly-logarithmic depth. Nonetheless, the circuit is such a simple and appealing model that it continues to attract a considerable amount of attention. Some very interesting exponential lower bounds on circuit size have been derived when the circuits are monotone, that is, realized by AND and OR gates but no NOTs.

1.6 Parallel Computation

The VLSI machine and the PRAM are examples of parallel machines. The VLSI machine reflects constraints that exist when finite-state machines are realized through the very large-scale integration of components on semiconductor chips. In the VLSI model the area of a chip is important because large chips have a much higher probability of containing a disabling defect than smaller ones. Consequently, the absolute size of chips is limited. However, the width of lines that can be drawn on chips has been shrinking over time, thereby increasing the number of wires, gates, and binary memory cells that can be placed on them. This has the effect of increasing the effective **chip area**, the real chip area normalized by the cross section of wires.

Figure 1.11(a) is a VLSI diagram representing the types of material that can be deposited on the surface of a pure crystalline semiconductor substrate to form different types of conducting regions. Some of the rectangular regions serve as wires whereas overlaps of other regions create transistors. In turn, collections of transistors form gates. This VLSI diagram describes a NAND gate, a gate whose Boolean function is the NOT of the AND of its two inputs. Shown in Fig. 1.11(b) is the logic symbol for the NAND gate. The small circle at the output of the AND gate denotes the NOT of the gate value.

Given the premium attached to chip real estate, a large number of economical and very regular finite-state machine designs have been made for VLSI chips. One of the most important of these is the **systolic array**, a one- or two-dimensional array of processors (FSMs) that are identical, except possibly for those along the periphery of the array. These processors operate in synchrony; that is, they perform the same operation at the same time. They also communicate only with their nearest neighbors. (The word "systolic" is derived from "systole," a "rhythmically recurrent contraction" such as that of the heart.)

Systolic arrays are typically used to compute specific functions such as the **convolution** $\boldsymbol{c} = \boldsymbol{a} \otimes \boldsymbol{b}$ of the n-tuple $\boldsymbol{a} = (a_0, a_1, \ldots, a_{n-1})$ with the m-tuple $\boldsymbol{b} = (b_0, b_1, \ldots, b_{m-1})$. The jth component, c_j, of the convolution $\boldsymbol{c} = \boldsymbol{a} \otimes \boldsymbol{b}$, $0 \leq j \leq (n + m - 2)$, is defined as

$$c_j = \sum_{r+s=j} a_r * b_s$$

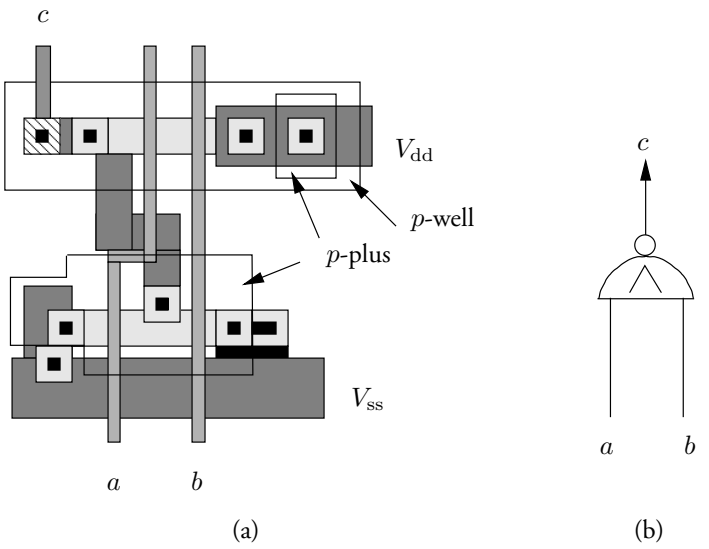

Figure 1.11 (a) A layout diagram for a VLSI chip and (b) its logic symbol.

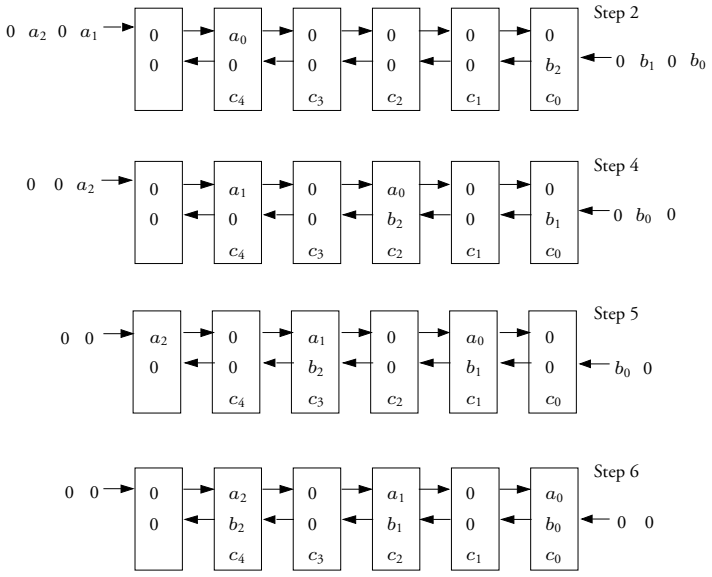

Figure 1.12 A systolic array for the convolution of two binary sequences.

It is assumed that the components of $\boldsymbol{a}$ and $\boldsymbol{b}$ are drawn from a set over which the operations of $*$ (multiplication) and $\sum$ (addition) are defined, such as the integers.

Shown schematically in Fig. 1.12 on page 28 is the one-dimensional systolic array for the convolution $\boldsymbol{c} = \boldsymbol{a} \otimes \boldsymbol{b}$ at the second, fourth, fifth, and sixth steps of execution on input vectors $\boldsymbol{a} = (a_0, a_1, a_2)$ and $\boldsymbol{b} = (b_0, b_1, b_2)$. The components of these vectors are fed from the left and right, respectively, spaced by zero elements. The first component of $\boldsymbol{a}$ enters the array one step ahead of the first component of $\boldsymbol{b}$. The result of the convolution is the vector $\boldsymbol{c} = (c_0, c_1, c_2, c_3, c_4)$. There is one more cell in the array than there are components in the result. At each step the components of $\boldsymbol{a}$ and $\boldsymbol{b}$ in each cell are multiplied and added to the previous value of the component of $\boldsymbol{c}$ in that cell. After all components of the two input vectors pass through the cell, the convolution is computed.

The processors of a parallel computer generally do not communicate only with nearest neighbors, as in the systolic array. Instead, processors often can communicate with remote neighbors via a network. The type of networks chosen for a parallel computer can have a large impact on their effectiveness.

The processors of the PRAM mentioned in Section 1.1 operate synchronously, alternating between accessing a global memory and computing locally. Since the processors communicate by writing and reading values to and from the global memory, all processors are at the same distance from one another. Although the PRAM model makes two unrealistic assumptions, namely that processors a) can act in synchrony and b) they can communicate directly via global memory, it remains a good model in which to explore problems that are hard to parallelize, even with the flexibility offered by this model.

Problems

MATHEMATICAL PRELIMINARIES

1.1 Show that the sum $S(k)$ below has value $S(k) = 2^k - 1$:

$$S(k) = 2^{k-1} + 2^{k-2} + \cdots + 2^1 + 2^0$$

SETS, LANGUAGES, INTEGERS, AND GRAPHS

1.2 Let $A = \{\text{red, green, blue}\}$, $B = \{\text{green, violet}\}$, and $C = \{\text{red, yellow, blue, green}\}$. Determine the elements in $(A \cap C) \times (B - C)$.

1.3 Let the relation $R \subseteq \mathbb{N} \times \mathbb{N}$ be defined by pairs (a, b) such that a and b have the same remainder on division by 3. Show that R is an equivalence relation.

1.4 Let $R \subset A \times A$ be an equivalence relation. Let the set $E[a]$ be the elements in A equivalent under the relation R to the element a. Show that for all $a, b \in A$ the equivalence classes $E[a]$ and $E[b]$ are either equal or disjoint. Also show that A is the union of all equivalence classes.

1.5 In terms of the Kleene closure and the concatenation of sets, describe the languages containing the following:

a) Strings over $\{0, 1\}$ beginning with 01.

b) Strings beginning with 0 that alternate between 0 and 1.

1.6 Describe an algorithm to convert numbers from decimal to binary notation.

1.7 A graph $G = (V, E)$ can be described by adjacency lists, one list for each vertex in the graph. The **adjacency list** for vertex $v \in V$ is a list of vertices to which there is an edge from v. Generate adjacency lists for the two graphs of Fig. 1.2.

TASKS AS FUNCTIONS

1.8 Let $\mathbb{Z}_5$ be the set $\{0, 1, 2, 3, 4\}$. Let the addition operator $\oplus$ over this set be **modulo** 5; that is, if x and y are two such integers, $x \oplus y$ is obtained by adding x and y as integers and taking the remainder after division by 5. For example, $2 \oplus 2 = 4 \bmod 5$ whereas $3 \oplus 4 = 7 = 2 \bmod 5$. Provide a table describing the function $f_{\oplus} : \mathbb{Z}_5 \times \mathbb{Z}_5 \mapsto \mathbb{Z}_5$.

1.9 Give a truth table for the Boolean function whose value is **True** exactly when either x or y or both is **True** and z is **False**.

RATE OF GROWTH OF FUNCTIONS

1.10 For each of the fifteen unordered pairs of functions f and g below, determine whether $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, or $f(n) = \Theta(g(n))$.

a) n^3; c) n^6; e) $n^3 \log_2 n$;

b) $2^{n \log_2 n}$; d) $n2^n$; f) 2^{2^n}.

1.11 Show that $2.7n^2 + 6\sqrt{n}\lceil \log_2 n \rceil \leq 8.7n^2$ for $n \geq 3$.

METHODS OF PROOF

1.12 Let $S_r(n) = \sum_{j=1}^{n} j^r$ denote a sum of powers of integers. Use proof by induction to show that the following identities on arithmetic series hold:

a) $S_2(n) = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$

b) $S_3(n) = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4}$

COMPUTATIONAL MODELS

1.13 Produce a circuit and straight-line program for the Boolean function described in Problem 1.9.

1.14 A **state diagram** for a finite-state machine is a graph containing one vertex (or **state**) for each pattern of data that can be held in its memory and an edge from state p to state q if there is a value for the input data that causes the memory to change from p to q. Such an edge is labeled with the value of the input data that causes the transition. Outputs are generated by a finite-state machine when it is in a state. The vertices of its state diagram are labeled by these outputs.

Provide a state diagram for the finite-state machine described in Fig. 1.5(b).

1.15 Using the straight-line program given for the full-adder circuit in Section 1.4.1, describe how such a program would be placed in the random-access memory of the RAM and how the RAM would run the fetch-and-execute cycle to compute the values produced by the full-adder circuit. This is an example of circuit simulation by a program.

1.16 Describe the actions that could be taken by a Turing machine to simulate a circuit from a straight-line program for it. Illustrate your approach by applying it to the simulation of the full-adder circuit described in Section 1.4.1.

1.17 Suppose you are told that a function is computed in four time steps by a very simple finite-state machine, one whose logic circuit (but not its memory) can be realized with four logic gates. Suppose you are also told that the same function cannot be computed by a logic circuit with fewer than 20 logic gates. What can be said about these two statements? Explain your answer.

1.18 Describe a finite-state machine that recognizes the language consisting of those strings over $\{0, 1\}$ that end in 1.

1.19 Determine the language generated by the context-free grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$ where $\mathcal{N} = \{\text{S}, \text{M}, \text{N}\}$, $\mathcal{T} = \{a, b, c, d\}$ and $\mathcal{R}$ consists of the rules given below.

a) $\text{S} \rightarrow \text{MN}$ d) $\text{N} \rightarrow c\text{N}d$

b) $\text{M} \rightarrow a\text{M}b$ e) $\text{N} \rightarrow cd$

c) $\text{M} \rightarrow ab$

COMPUTATIONAL COMPLEXITY

1.20 Using the rules for the red pebble game, show how to pebble the FFT graph of Fig. 1.10 with five red pebbles by labeling the vertices with the time step on which it is pebbled. If a vertex has to be repebbled, it will be pebbled on two time steps.

1.21 Suppose that you are told that the n-point FFT graph can be pebbled with $\sqrt{n}$ pebbles in $n/4$ time steps for $n \geq 37$. What can you say about this statement?

1.22 You have been told that the FFT graph of Fig. 1.10 cannot be pebbled with fewer than three red pebbles. Show that it can be pebbled with two red pebbles in the red-blue pebble game by sketching how you would use blue pebbles to achieve this objective.

PARALLEL COMPUTATION

1.23 Using Fig. 1.12 as a guide, design a systolic array to convolve two sequences of length two. Sketch out each step of the convolution process.

1.24 Consider a version of the PRAM consisting of a collection of RAMs (see Fig. 1.13) with small local random-access memories that repeat the following three-step cycle until they halt: a) they simultaneously read one word from a common global memory, b) they execute one local instruction using local memory, and c) they write one word to the common memory. When reading and writing, the individual processors are allowed to read and write from the same location. If two RAMs write to the same location, they must be programmed so that they write a common value. (This is known as the concurrent-read, concurrent-write (CRCW) PRAM.) Each RAM has a unique integer associated with it and can use this number to decide where to read or write in the common memory.

Show that the CRCW PRAM can compute the AND of n Boolean variables in two cycles.

Hint: Reserve one word in common memory and initialize it with 0 and assign RAMs to the appropriate memory cells.

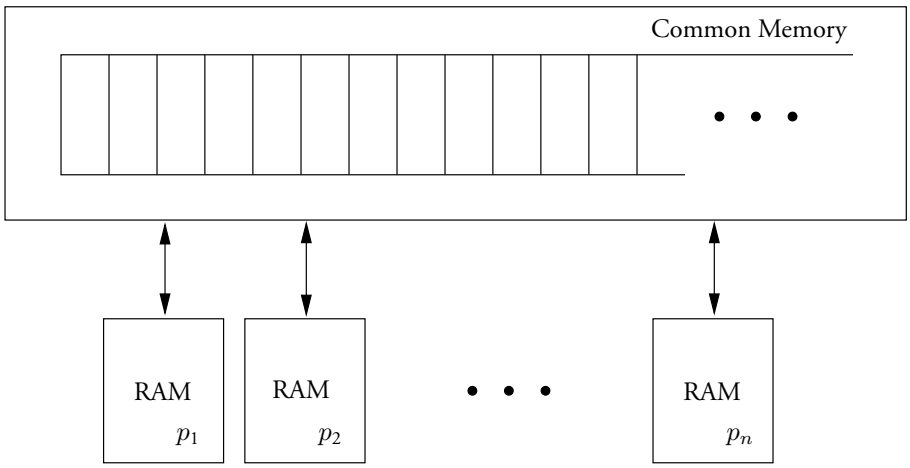

Figure 1.13 The PRAM model is a collection of synchronous RAMs accessing a common memory.

Chapter Notes

Since this chapter introduces concepts used elsewhere in the book, we postpone the bibliographic citations to later chapters. We remark here, however, that the notation for the rate of growth of functions in Section 1.2.8 is due to Knuth [170]. The reader interested in more information on the development of the digital computer, ranging from Babbage's seminal work in the 1830s to the pioneering work of the 1940s, should consult the collection of papers selected and edited by Brian Randell [267].