Solution: (a) To construct a regular expression for the set of bit strings with even length, we use the fact that such a string can be obtained by concatenating zero or more strings each consisting of two bits. The set of strings of two bits is specified by the regular expression **(00 ∪ 01 ∪ 10 ∪ 11)**. Consequently, the set of strings with even length is specified by **(00 ∪ 01 ∪ 10 ∪ 11)*** .

(b) A bit string ending with a 0 and not containing 11 must be the concatenation of one or more strings where each string is either a 0 or a 10. (To see this, note that such a bit string must consist of 0 bits or 1 bits each followed by a 0; the string cannot end with a single 1 because we know it ends with a 0.) It follows that the regular expression **(0 ∪ 10)* (0 ∪ 10)** specifies the set of bit strings that do not contain 11 and end with a 0. [Note that the set specified by **(0 ∪ 10)*** includes the empty string, which is not in this set, because the empty string does not end with a 0.]

(c) A bit string containing an odd number of 0s must contain at least one 0, which tells us that it starts with zero or more 1s, followed by a 0, followed by zero or more 1s. That is, each such bit string begins with a string of the form 1^j01^k for nonnegative integers j and k. Because the bit string contains an odd number of 0s, additional bits after this initial block can be split into blocks each starting with a 0 and containing one more 0. Each such block is of the form 01^p01^q, where p and q are nonnegative integers. Consequently, the regular expression **1*01*(01*01*)*** specifies the set of bit strings with an odd number of 0s. ◀

Kleene's Theorem

In 1956 Kleene proved that regular sets are the sets that are recognized by a finite-state automaton. Consequently, this important result is called Kleene's theorem.

THEOREM 1

KLEENE'S THEOREM A set is regular if and only if it is recognized by a finite-state automaton.

Links

Kleene's theorem is one of the central results in automata theory. We will prove the *only if* part of this theorem, namely, that every regular set is recognized by a finite-state automaton. The proof of the *if* part, that a set recognized by a finite-state automaton is regular, is left as an exercise for the reader.

Proof: Recall that a regular set is defined in terms of regular expressions, which are defined recursively. We can prove that every regular set is recognized by a finite-state automaton if we can do the following things.

1. Show that $\emptyset$ is recognized by a finite-state automaton.
2. Show that $\{\lambda\}$ is recognized by a finite-state automaton.
3. Show that $\{a\}$ is recognized by a finite-state automaton whenever a is a symbol in I.
4. Show that AB is recognized by a finite-state automaton whenever both A and B are.
5. Show that $A \cup B$ is recognized by a finite-state automaton whenever both A and B are.
6. Show that A^* is recognized by a finite-state automaton whenever A is.

We now consider each of these tasks. First, we show that $\emptyset$ is recognized by a nondeterministic finite-state automaton. To do this, all we need is an automaton with no final states. Such an automaton is shown in Figure 1(a).

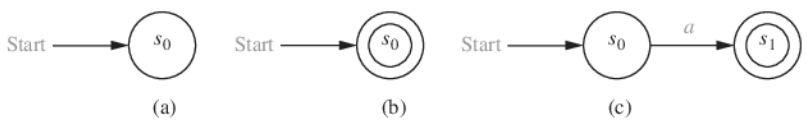


FIGURE 1 Nondeterministic Finite-State Automata That Recognize Some Basic Sets.

Second, we show that $\{\lambda\}$ is recognized by a finite-state automaton. To do this, all we need is an automaton that recognizes λ, the null string, but not any other string. This can be done by making the start state s_0 a final state and having no transitions, so that no other string takes s_0 to a final state. The nondeterministic automaton in Figure 1(b) shows such a machine.

Third, we show that $\{a\}$ is recognized by a nondeterministic finite-state automaton. To do this, we can use a machine with a starting state s_0 and a final state s_1. We have a transition from s_0 to s_1 when the input is a, and no other transitions. The only string recognized by this machine is a. This machine is shown in Figure 1(c).

Next, we show that AB and $A \cup B$ can be recognized by finite-state automata if A and B are languages recognized by finite-state automata. Suppose that A is recognized by $M_A = (S_A, I, f_A, s_A, F_A)$ and B is recognized by $M_B = (S_B, I, f_B, s_B, F_B)$.

We begin by constructing a finite-state machine $M_{AB} = (S_{AB}, I, f_{AB}, s_{AB}, F_{AB})$ that recognizes AB, the concatenation of A and B. We build such a machine by combining the machines for A and B in series, so a string in A takes the combined machine from s_A, the start state of M_A, to s_B, the start state of M_B. A string in B should take the combined machine from s_B to a final state of the combined machine. Consequently, we make the following construction. Let S_{AB} be $S_A \cup S_B$. [Note that we can assume that S_A and S_B are disjoint.] The starting state s_{AB} is the same as s_A. The set of final states, F_{AB}, is the set of final states of M_B with s_{AB} included if and only if $\lambda \in A \cap B$. The transitions in M_{AB} include all transitions in M_A and in M_B, as well as some new transitions. For every transition in M_A that leads to a final state, we form a transition in M_{AB} from the same state to s_B, on the same input. In this way, a string in A takes M_{AB} from s_{AB} to s_B, and then a string in B takes s_B to a final state of M_{AB}. Moreover, for every transition from s_B we form a transition in M_{AB} from s_{AB} to the same state. Figure 2(a) contains an illustration of this construction.

We now construct a machine $M_{A\cup B} = (S_{A\cup B}, I, f_{A\cup B}, s_{A\cup B}, F_{A\cup B})$ that recognizes $A \cup B$. This automaton can be constructed by combining M_A and M_B in parallel, using a new start state that has the transitions that both s_A and s_B have. Let $S_{A\cup B} = S_A \cup S_B \cup \{s_{A\cup B}\}$, where $s_{A\cup B}$ is a new state that is the start state of $M_{A\cup B}$. Let the set of final states $F_{A\cup B}$ be $F_A \cup F_B \cup \{s_{A\cup B}\}$ if $\lambda \in A \cup B$, and $F_A \cup F_B$ otherwise. The transitions in $M_{A\cup B}$ include all those in M_A and in M_B. Also, for each transition from s_A to a state s on input i we include a transition from $s_{A\cup B}$ to s on input i, and for each transition from s_B to a state s on input i we include a transition from $s_{A\cup B}$ to s on input i. In this way, a string in A leads from $s_{A\cup B}$ to a final state in the new machine, and a string in B leads from $s_{A\cup B}$ to a final state in the new machine. Figure 2(b) illustrates the construction of $M_{A\cup B}$.

Finally, we construct $M_{A^*} = (S_{A^*}, I, f_{A^*}, s_{A^*}, F_{A^*})$, a machine that recognizes A^*, the Kleene closure of A. Let S_{A^*} include all states in S_A and one additional state s_{A^*}, which is the starting state for the new machine. The set of final states F_{A^*} includes all states in F_A as well as the start state s_{A^*}, because λ must be recognized. To recognize concatenations of arbitrarily many strings from A, we include all the transitions in M_A, as well as transitions from s_{A^*} that match the transitions from s_A, and transitions from each final state that match the transitions from s_A. With this set of transitions, a string made up of concatenations of strings from A will take s_{A^*} to a final state when the first string in A has been read, returning to a final state when the second string in A has been read, and so on. Figure 2(c) illustrates the construction we used. ◁

A nondeterministic finite-state automaton can be constructed for any regular set using the procedure described in this proof. We illustrate how this is done with Example 3.

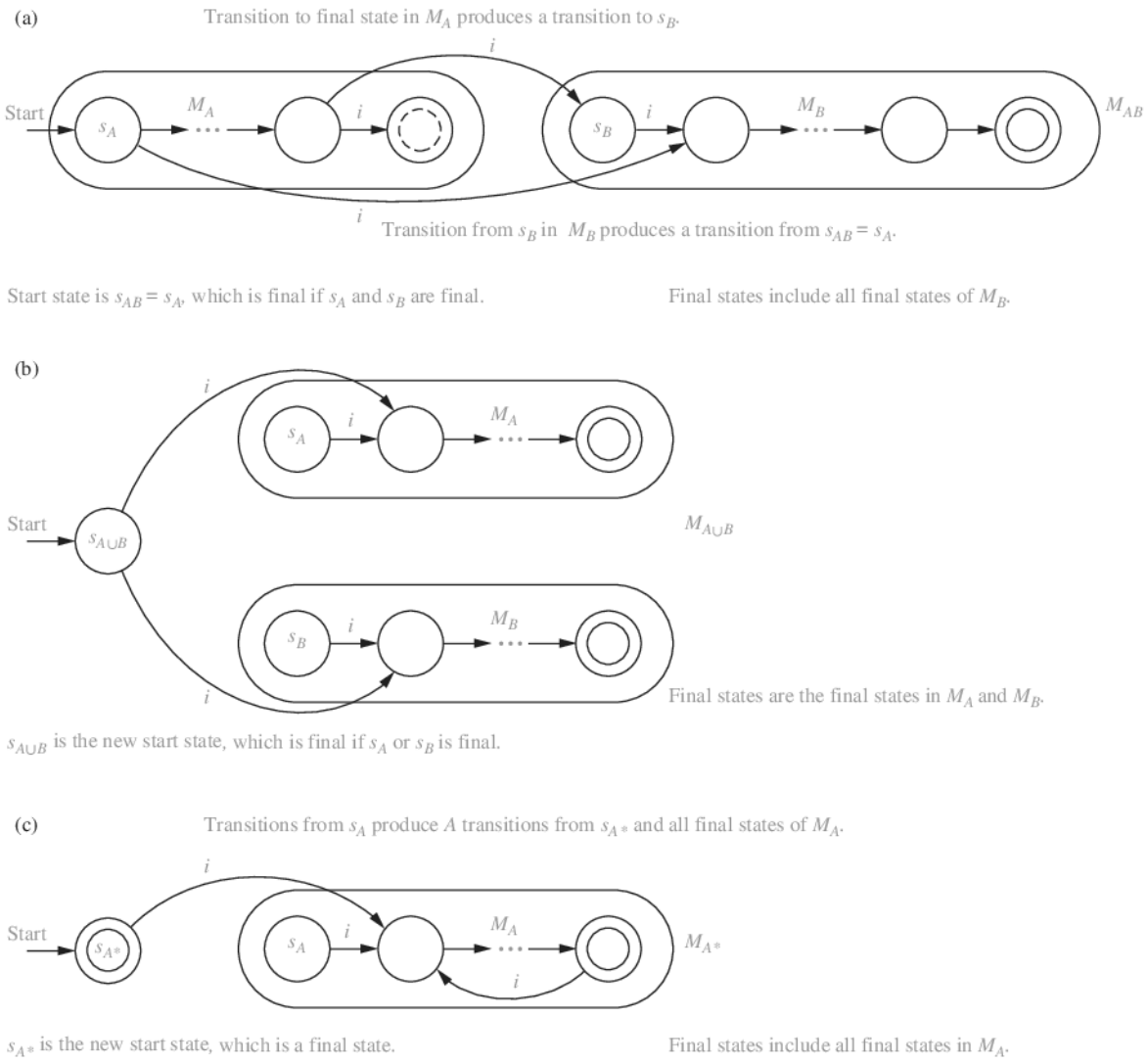


FIGURE 2 **Building Automata to Recognize Concatenations, Unions, and Kleene Closures.**

EXAMPLE 3 Construct a nondeterministic finite-state automaton that recognizes the regular set $\mathbf{1}^* \cup \mathbf{01}$.

Solution: We begin by building a machine that recognizes $\mathbf{1}^*$. This is done using the machine that recognizes **1** and then using the construction for M_{A^*} described in the proof. Next, we build a machine that recognizes **01**, using machines that recognize **0** and **1** and the construction in the proof for M_{AB}. Finally, using the construction in the proof for $M_{A \cup B}$, we construct the machine for $\mathbf{1}^* \cup \mathbf{01}$. The finite-state automata used in this construction are shown in Figure 3. The states in the successive machines have been labeled using different subscripts, even when a state is formed from one previously used in another machine. Note that the construction given here does not produce the simplest machine that recognizes $\mathbf{1}^* \cup \mathbf{01}$. A much simpler machine that recognizes this set is shown in Figure 3(b). ◀

Regular Sets and Regular Grammars

In Section 13.1 we introduced phrase-structure grammars and defined different types of grammars. In particular we defined regular, or type 3, grammars, which are grammars of the

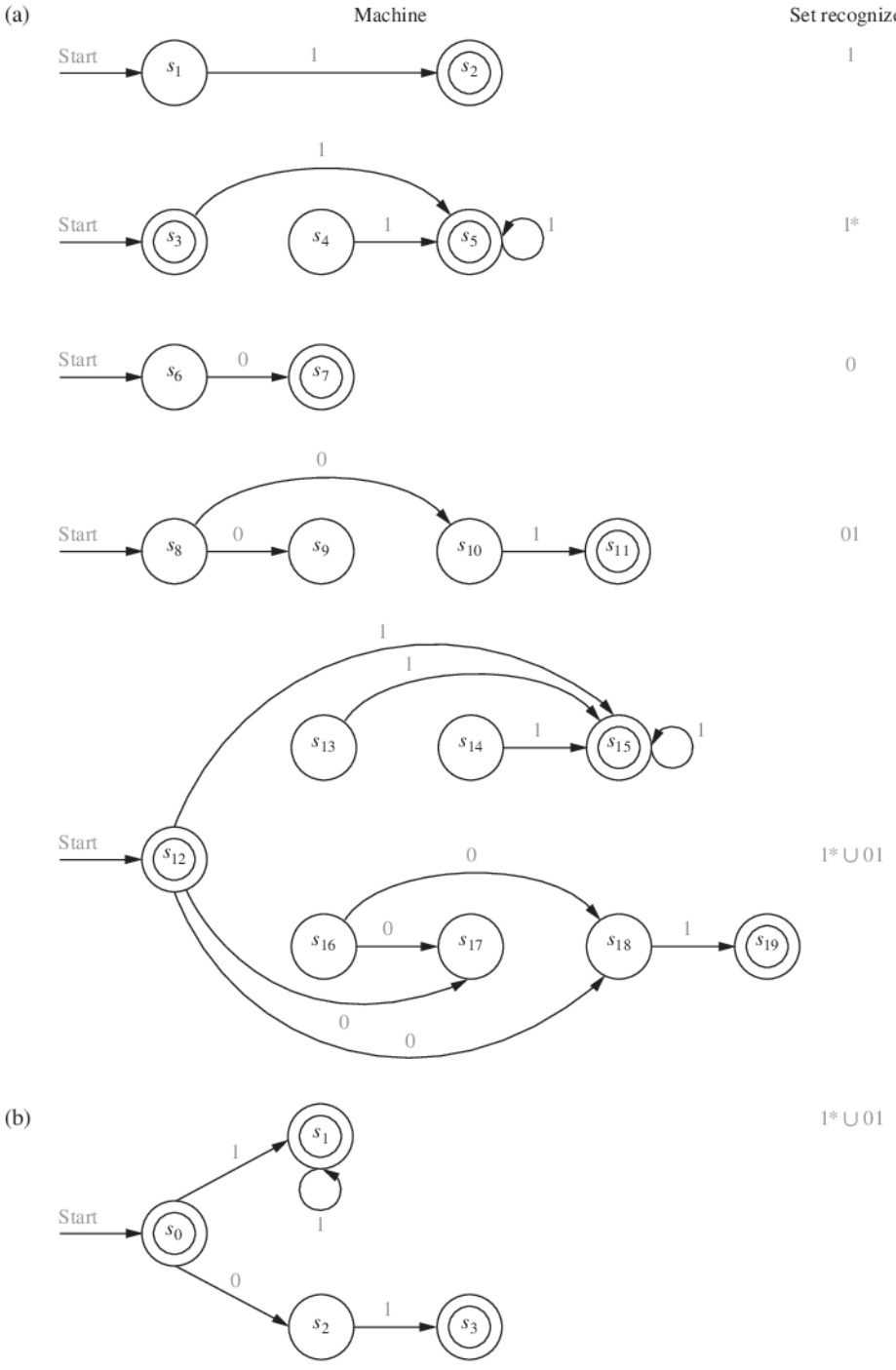


FIGURE 3 **Nondeterministic Finite-State Automata Recognizing 1* ∪ 01.**

form $G = (V, T, S, P)$, where each production is of the form $S \to \lambda$, $A \to a$, or $A \to aB$, where a is a terminal symbol, and A and B are nonterminal symbols. As the terminology suggests, there is a close connection between regular grammars and regular sets.

THEOREM 2 A set is generated by a regular grammar if and only if it is a regular set.

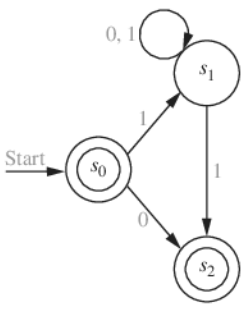


FIGURE 4 A Nondeterministic Finite-State Automaton Recognizing $L(G)$.

Proof: First we show that a set generated by a regular grammar is a regular set. Suppose that $G = (V, T, S, P)$ is a regular grammar generating the set $L(G)$. To show that $L(G)$ is regular we will build a nondeterministic finite-state machine $M = (S, I, f, s_0, F)$ that recognizes $L(G)$. Let S, the set of states, contain a state s_A for each nonterminal symbol A of G and an additional state s_F, which is a final state. The start state s_0 is the state formed from the start symbol S. The transitions of M are formed from the productions of G in the following way. A transition from s_A to s_F on input of a is included if $A \rightarrow a$ is a production, and a transition from s_A to s_B on input of a is included if $A \rightarrow aB$ is a production. The set of final states includes s_F and also includes s_0 if $S \rightarrow \lambda$ is a production in G. It is not hard to show that the language recognized by M equals the language generated by the grammar G, that is, $L(M) = L(G)$. This can be done by determining the words that lead to a final state. The details are left as an exercise for the reader. ◁

Before giving the proof of the converse, we illustrate how a nondeterministic machine is constructed that recognizes the same set as a regular grammar.

EXAMPLE 4 Construct a nondeterministic finite-state automaton that recognizes the language generated by the regular grammar $G = (V, T, S, P)$, where $V = \{0, 1, A, S\}$, $T = \{0, 1\}$, and the productions in P are $S \rightarrow 1A$, $S \rightarrow 0$, $S \rightarrow \lambda$, $A \rightarrow 0A$, $A \rightarrow 1A$, and $A \rightarrow 1$.

Solution: The state diagram for a nondeterministic finite-state automaton that recognizes $L(G)$ is shown in Figure 4. This automaton is constructed following the procedure described in the proof. In this automaton, s_0 is the state corresponding to S, s_1 is the state corresponding to A, and s_2 is the final state. ◀

We now complete the proof of Theorem 2.

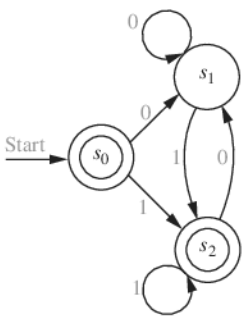


FIGURE 5 A Finite-State Automaton.

Proof: We now show that if a set is regular, then there is a regular grammar that generates it. Suppose that M is a finite-state machine that recognizes this set with the property that s_0, the starting state of M, is never the next state for a transition. (We can find such a machine by Exercise 20.) The grammar $G = (V, T, S, P)$ is defined as follows. The set V of symbols of G is formed by assigning a symbol to each state of S and each input symbol in I. The set T of terminal symbols of G is the set I. The start symbol S is the symbol formed from the start state s_0. The set P of productions in G is formed from the transitions in M. In particular, if the state s goes to a final state under input a, then the production $A_s \rightarrow a$ is included in P, where A_s is the nonterminal symbol formed from the state s. If the state s goes to the state t on input a, then the production $A_s \rightarrow aA_t$ is included in P. The production $S \rightarrow \lambda$ is included in P if and only if $\lambda \in L(M)$. Because the productions of G correspond to the transitions of M and the productions leading to terminals correspond to transitions to final states, it is not hard to show that $L(G) = L(M)$. We leave the details as an exercise for the reader. ◁

Example 5 illustrates the construction used to produce a grammar from an automaton that generates the language recognized by this automaton.

EXAMPLE 5 Find a regular grammar that generates the regular set recognized by the finite-state automaton shown in Figure 5.

Solution: The grammar $G = (V, T, S, P)$ generates the set recognized by this automaton where $V = \{S, A, B, 0, 1\}$, the symbols S, A, and B correspond to the states s_0, s_1, and s_2, respectively, $T = \{0, 1\}$, S is the start symbol; and the productions are $S \rightarrow 0A$, $S \rightarrow 1B$, $S \rightarrow 1$, $S \rightarrow \lambda$, $A \rightarrow 0A$, $A \rightarrow 1B$, $A \rightarrow 1$, $B \rightarrow 0A$, $B \rightarrow 1B$, and $B \rightarrow 1$. ◀

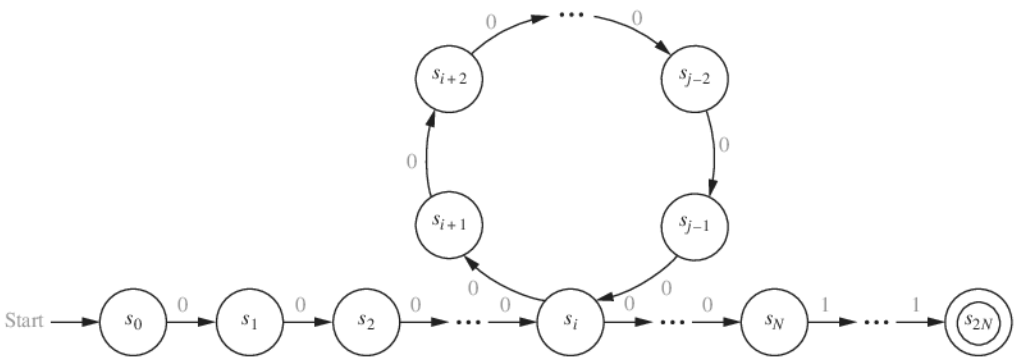


FIGURE 6 **The Path Produced by $0^N 1^N$.**

A Set Not Recognized by a Finite-State Automaton

We have seen that a set is recognized by a finite state automaton if and only if it is regular. We will now show that there are sets that are not regular by describing one such set. The technique used to show that this set is not regular illustrates an important method for showing that certain sets are not regular.

EXAMPLE 6 Show that the set $\{0^n 1^n \mid n = 0, 1, 2, \ldots\}$, made up of all strings consisting of a block of 0s followed by a block of an equal number of 1s, is not regular.

Solution: Suppose that this set were regular. Then there would be a nondeterministic finite-state automaton $M = (S, I, f, s_0, F)$ recognizing it. Let N be the number of states in this machine, that is, $N = |S|$. Because M recognizes all strings made up of a number of 0s followed by an equal number of 1s, M must recognize $0^N 1^N$. Let $s_0, s_1, s_2, \ldots, s_{2N}$ be the sequence of states that is obtained starting at s_0 and using the symbols of $0^N 1^N$ as input, so that $s_1 = f(s_0, 0)$, $s_2 = f(s_1, 0), \ldots, s_N = f(s_{N-1}, 0), s_{N+1} = f(s_N, 1), \ldots, s_{2N} = f(s_{2N-1}, 1)$. Note that s_{2N} is a final state.

Because there are only N states, the pigeonhole principle shows that at least two of the first $N + 1$ of the states, which are $s_0, \ldots, s_N$, must be the same. Say that s_i and s_j are two such identical states, with $0 \leq i < j \leq N$. This means that $f(s_i, 0^t) = s_j$, where $t = j - i$. It follows that there is a loop leading from s_i back to itself, obtained using the input 0 a total of t times, in the state diagram shown in Figure 6.

Now consider the input string $0^N 0^t 1^N = 0^{N+t} 1^N$. There are t more consecutive 0s at the start of this block than there are consecutive 1s that follow it. Because this string is not of the form $0^n 1^n$ (because it has more 0s than 1s), it is not recognized by M. Consequently, $f(s_0, 0^{N+t} 1^N)$ cannot be a final state. However, when we use the string $0^{N+t} 1^N$ as input, we end up in the same state as before, namely, s_{2N}. The reason for this is that the extra t 0s in this string take us around the loop from s_i back to itself an extra time, as shown in Figure 6. Then the rest of the string leads us to exactly the same state as before. This contradiction shows that $\{0^n 1^n \mid n = 0, 1, 2, \ldots\}$ is not regular. ◀

More Powerful Types of Machines

Links

Finite-state automata are unable to carry out many computations. The main limitation of these machines is their finite amount of memory. This prevents them from recognizing languages that are not regular, such as $\{0^n 1^n \mid n = 0, 1, 2, \ldots\}$. Because a set is regular if and only if it is the language generated by a regular grammar, Example 6 shows that there is no regular grammar

that generates the set $\{0^n 1^n \mid n = 0, 1, 2, \dots\}$. However, there is a context-free grammar that generates this set. Such a grammar was given in Example 5 in Section 13.1.

Because of the limitations of finite-state machines, it is necessary to use other, more powerful, models of computation. One such model is the **pushdown automaton**. A pushdown automaton includes everything in a finite-state automaton, as well as a stack, which provides unlimited memory. Symbols can be placed on the top or taken off the top of the stack. A set is recognized in one of two ways by a pushdown automaton. First, a set is recognized if the set consists of all the strings that produce an empty stack when they are used as input. Second, a set is recognized if it consists of all the strings that lead to a final state when used as input. It can be shown that a set is recognized by a pushdown automaton if and only if it is the language generated by a context-free grammar.

However, there are sets that cannot be expressed as the language generated by a context-free grammar. One such set is $\{0^n 1^n 2^n \mid n = 0, 1, 2, \dots\}$. We will indicate why this set cannot be recognized by a pushdown automaton, but we will not give a proof, because we have not developed the machinery needed. (However, one method of proof is given in Exercise 28 of the supplementary exercises at the end of this chapter.) The stack can be used to show that a string begins with a sequence of 0s followed by an equal number of 1s by placing a symbol on the stack for each 0 (as long as only 0s are read), and removing one of these symbols for each 1 (as long as only 1s following the 0s are read). But once this is done, the stack is empty, and there is no way to determine that there are the same number of 2s in the string as 0s.

There are other machines called **linear bounded automata**, more powerful than pushdown automata, that can recognize sets such as $\{0^n 1^n 2^n \mid n = 0, 1, 2, \dots\}$. In particular, linear bounded automata can recognize context-sensitive languages. However, these machines cannot recognize all the languages generated by phrase-structure grammars. To avoid the limitations of special types of machines, the model known as a **Turing machine**, named after the British mathematician Alan Turing, is used. A Turing machine is made up of everything included in a finite-state machine together with a tape, which is infinite in both directions. A Turing machine has read and write capabilities on the tape, and it can move back and forth along this tape. Turing machines can recognize all languages generated by phrase-structure grammars. In addition, Turing machines can model all the computations that can be performed on a computing machine. Because of their power, Turing machines are extensively studied in theoretical computer science. We will briefly study them in Section 13.5.

Alan Turing invented Turning machines before modern computers existed!

Links

ALAN MATHISON TURING (1912–1954) Alan Turing was born in London, although he was conceived in India, where his father was employed in the Indian Civil Service. As a boy, he was fascinated by chemistry, performing a wide variety of experiments, and by machinery. Turing attended Sherborne, an English boarding school. In 1931 he won a scholarship to King's College, Cambridge. After completing his dissertation, which included a rediscovery of the central limit theorem, a famous theorem in statistics, he was elected a fellow of his college. In 1935 Turing became fascinated with the decision problem, a problem posed by the great German mathematician Hilbert, which asked whether there is a general method that can be applied to any assertion to determine whether the assertion is true. Turing enjoyed running (later in life running as a serious amateur in competitions), and one day, while resting after a run, he discovered the key ideas needed to solve the decision problem. In his solution, he invented what is now called a **Turing machine** as the most general model of a computing machine. Using these machines, he found a problem, involving what he called computable numbers, that could not be decided using a general method.

From 1936 to 1938 Turing visited Princeton University to work with Alonzo Church, who had also solved Hilbert's decision problem. In 1939 Turing returned to King's College. However, at the outbreak of World War II, he joined the Foreign Office, performing cryptanalysis of German ciphers. His contribution to the breaking of the code of the Enigma, a mechanical German cipher machine, played an important role in winning the war.

After the war, Turing worked on the development of early computers. He was interested in the ability of machines to think, proposing that if a computer could not be distinguished from a person based on written replies to questions, it should be considered to be "thinking." He was also interested in biology, having written on morphogenesis, the development of form in organisms. In 1954 Turing committed suicide by taking cyanide, without leaving a clear explanation. Legal troubles related to a homosexual relationship and hormonal treatments mandated by the court to lessen his sex drive may have been factors in his decision to end his life.

Exercises

1. Describe in words the strings in each of these regular sets.
a) 1^*0 **b)** 1^*00^*
c) $111 \cup 001$ **d)** $(1 \cup 00)^*$
e) $(00^*1)^*$ **f)** $(0 \cup 1)(0 \cup 1)^*00$

2. Describe in words the strings in each of these regular sets.
a) 001^* **b)** $(01)^*$
c) $01 \cup 001^*$ **d)** $0(11 \cup 0)^*$
e) $(101^*)^*$ **f)** $(0^* \cup 1)11$

3. Determine whether 0101 belongs to each of these regular sets.
a) 01^*0^* **b)** $0(11)^*(01)^*$
c) $0(10)^*1^*$ **d)** $0^*10(0 \cup 1)$
e) $(01)^*(11)^*$ **f)** $0^*(10 \cup 11)^*$
g) $0^*(10)^*11$ **h)** $01(01 \cup 0)1^*$

4. Determine whether 1011 belongs to each of these regular sets.
a) 10^*1^* **b)** $0^*(10 \cup 11)^*$
c) $1(01)^*1^*$ **d)** $1^*01(0 \cup 1)$
e) $(10)^*(11)^*$ **f)** $1(00)^*(11)^*$
g) $(10)^*1011$ **h)** $(1 \cup 00)(01 \cup 0)1^*$

5. Express each of these sets using a regular expression.
a) the set consisting of the strings 0, 11, and 010
b) the set of strings of three 0s followed by two or more 0s
c) the set of strings of odd length
d) the set of strings that contain exactly one 1
e) the set of strings ending in 1 and not containing 000

6. Express each of these sets using a regular expression.
a) the set containing all strings with zero, one, or two bits
b) the set of strings of two 0s, followed by zero or more 1s, and ending with a 0
c) the set of strings with every 1 followed by two 0s
d) the set of strings ending in 00 and not containing 11
e) the set of strings containing an even number of 1s

7. Express each of these sets using a regular expression.
a) the set of strings of one or more 0s followed by a 1
b) the set of strings of two or more symbols followed by three or more 0s
c) the set of strings with either no 1 preceding a 0 or no 0 preceding a 1
d) the set of strings containing a string of 1s such that the number of 1s equals 2 modulo 3, followed by an even number of 0s

8. Construct deterministic finite-state automata that recognize each of these sets from I^*, where I is an alphabet.
a) $\emptyset$ **b)** $\{\lambda\}$ **c)** $\{a\}$, where $a \in I$

9. Construct nondeterministic finite-state automata that recognize each of the sets in Exercise 8.

10. Construct nondeterministic finite-state automata that recognize each of these sets.
a) $\{\lambda, 0\}$ **b)** $\{0, 11\}$ **c)** $\{0, 11, 000\}$

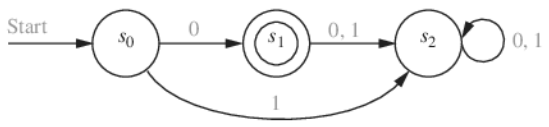
***11.** Show that if A is a regular set, then A^R, the set of all reversals of strings in A, is also regular.

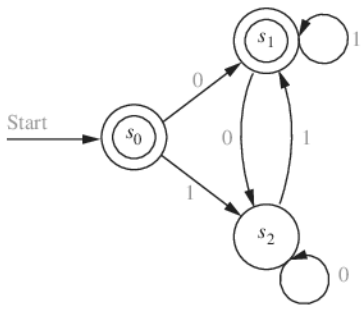
12. Using the constructions described in the proof of Kleene's theorem, find nondeterministic finite-state automata that recognize each of these sets.
a) 01^* **b)** $(0 \cup 1)1^*$ **c)** $00(1^* \cup 10)$

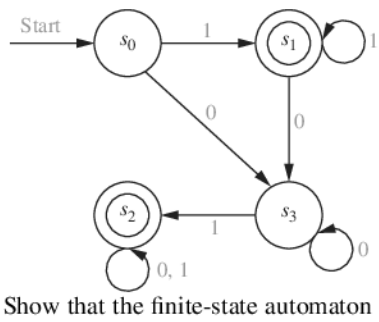
13. Using the constructions described in the proof of Kleene's theorem, find nondeterministic finite-state automata that recognize each of these sets.
a) 0^*1^* **b)** $(0 \cup 11)^*$ **c)** $01^* \cup 00^*1$

14. Construct a nondeterministic finite-state automaton that recognizes the language generated by the regular grammar $G = (V, T, S, P)$, where $V = \{0, 1, S, A, B\}$, $T = \{0, 1\}$, S is the start symbol, and the set of productions is
a) $S \to 0A$, $S \to 1B$, $A \to 0$, $B \to 0$.
b) $S \to 1A$, $S \to 0$, $S \to \lambda$, $A \to 0B$, $B \to 1B$, $B \to 1$.
c) $S \to 1B$, $S \to 0$, $A \to 1A$, $A \to 0B$, $A \to 1$, $A \to 0$, $B \to 1$.

In Exercises 15–17 construct a regular grammar $G = (V, T, S, P)$ that generates the language recognized by the given finite-state machine.

15.


16.


17.


18. Show that the finite-state automaton constructed from a regular grammar in the proof of Theorem 2 recognizes the set generated by this grammar.

19. Show that the regular grammar constructed from a finite-state automaton in the proof of Theorem 2 generates the set recognized by this automaton.

20. Show that every nondeterministic finite-state automaton is equivalent to another such automaton that has the property that its starting state is never revisited.

***21.** Let $M = (S, I, f, s_0, F)$ be a deterministic finite-state automaton. Show that the language recognized by M, $L(M)$, is infinite if and only if there is a word x recognized by M with $l(x) \geq |S|$.

***22.** One important technique used to prove that certain sets are not regular is the **pumping lemma**. The pumping lemma states that if $M = (S, I, f, s_0, F)$ is a deterministic finite-state automaton and if x is a string in $L(M)$, the language recognized by M, with $l(x) \geq |S|$, then there are strings u, v, and w in I^* such that $x = uvw$, $l(uv) \leq |S|$ and $l(v) \geq 1$, and $uv^i w \in L(M)$ for $i = 0, 1, 2, \ldots$. Prove the pumping lemma. [*Hint:* Use the same idea as was used in Example 5.]

***23.** Show that the set $\{0^{2n}1^n \mid n = 0, 1, 2, \ldots\}$ is not regular using the pumping lemma given in Exercise 22.

***24.** Show that the set $\{1^{n^2} \mid n = 0, 1, 2, \ldots\}$ is not regular using the pumping lemma from Exercise 22.

***25.** Show that the set of palindromes over $\{0, 1\}$ is not regular using the pumping lemma given in Exercise 22. [*Hint:* Consider strings of the form $0^N 10^N$.]

****26.** Show that a set recognized by a finite-state automaton is regular. (This is the *if* part of Kleene's theorem.)

Suppose that L is a subset of I^*, where I is a nonempty set of symbols. If $x \in I^*$, we let $L/x = \{z \in I^* \mid xz \in L\}$. We say that the strings $x \in I^*$ and $y \in I^*$ are **distinguishable with respect to** L if $L/x \neq L/y$. A string z for which $xz \in L$ but $yz \notin L$, or $xz \notin L$, but $yz \in L$ is said to **distinguish** x and y with respect to L. When $L/x = L/y$, we say that x and y are **indistinguishable** with respect to L.

27. Let L be the set of all bit strings that end with 01. Show that 11 and 10 are distinguishable with respect to L and that the strings 1 and 11 are indistinguishable with respect to L.

28. Suppose that $M = (S, I, f, s_0, F)$ is a deterministic finite-state machine. Show that if x and y are two strings in I^* that are distinguishable with respect to $L(M)$, then $f(s_0, x) \neq f(s_0, y)$.

***29.** Suppose that L is a subset of I^* and for some positive integer n there are n strings in I^* such that every two of these strings are distinguishable with respect to L. Prove that every deterministic finite-state automaton recognizing L has at least n states.

***30.** Let L_n be the set of strings with at least n bits in which the nth symbol from the end is a 0. Use Exercise 29 to show that a deterministic finite-state machine recognizing L_n must have at least 2^n states.

***31.** Use Exercise 29 to show that the language consisting of all bit strings that are palindromes (that is, strings that equal their own reversals) is not regular.

13.5 Turing Machines

Introduction

Links

"Machines take me by surprise with great frequency" – Alan Turing

The finite-state automata studied earlier in this chapter cannot be used as general models of computation. They are limited in what they can do. For example, finite-state automata are able to recognize regular sets, but are not able to recognize many easy-to-describe sets, including $\{0^n 1^n \mid n \geq 0\}$, which computers recognize using memory. We can use finite-state automata to compute relatively simple functions such as the sum of two numbers, but we cannot use them to compute functions that computers can, such as the product of two numbers. To overcome these deficiencies we can use a more powerful type of machine known as a Turing machine, after Alan Turing, the famous mathematician and computer scientist who invented them in the 1930s.

Basically, a Turing machine consists of a control unit, which at any step is in one of finitely many different states, together with a tape divided into cells, which is infinite in both directions. Turing machines have read and write capabilities on the tape as the control unit moves back and forth along this tape, changing states depending on the tape symbol read. Turing machines are more powerful than finite-state machines because they include memory capabilities that finite-state machines lack. We will show how to use Turing machines to recognize sets, including sets that cannot be recognized by finite-state machines. We will also show how to compute functions using Turing machines. Turing machines are the most general models of computation; essentially, they can do whatever a computer can do. Note that Turing machines are much more powerful than real computers, which have finite memory capabilities.

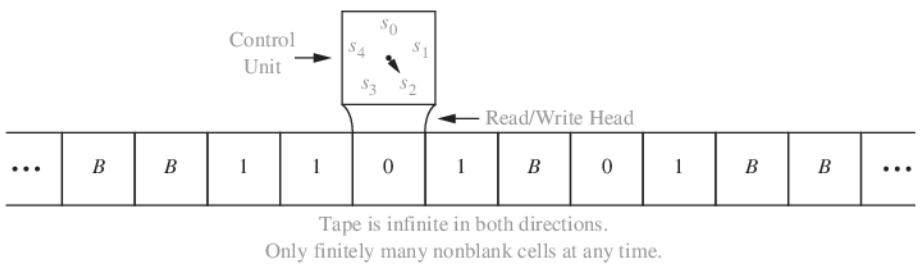


FIGURE 1 **A Representation of a Turing Machine.**

Definition of Turing Machines

We now give the formal definition of a Turing machine. Afterward we will explain how this formal definition can be interpreted in terms of a control head that can read and write symbols on a tape and move either right or left along the tape.

DEFINITION 1

A *Turing machine* $T = (S, I, f, s_0)$ consists of a finite set S of states, an alphabet I containing the blank symbol B, a partial function f from $S \times I$ to $S \times I \times \{R, L\}$, and a starting state s_0.

Recall from Section 2.3 that a partial function is defined only for those elements in its domain of definition. This means that for some (state, symbol) pairs the partial function f may be undefined, but for a pair for which it is defined, there is a unique (state, symbol, direction) triple associated to this pair. We call the five-tuples corresponding to the partial function in the definition of a Turing machine the **transition rules** of the machine.

To interpret this definition in terms of a machine, consider a control unit and a tape divided into cells, infinite in both directions, having only a finite number of nonblank symbols on it at any given time, as pictured in Figure 1. The action of the Turing machine at each step of its operation depends on the value of the partial function f for the current state and tape symbol.

At each step, the control unit reads the current tape symbol x. If the control unit is in state s and if the partial function f is defined for the pair (s, x) with $f(s, x) = (s', x', d)$, the control unit

1. enters the state s',
2. writes the symbol x' in the current cell, erasing x, and
3. moves right one cell if $d = R$ or moves left one cell if $d = L$.

We write this step as the five-tuple (s, x, s', x', d). If the partial function f is undefined for the pair (s, x), then the Turing machine T will *halt*.

A common way to define a Turing machine is to specify a set of five-tuples of the form (s, x, s', x', d). The set of states and input alphabet is implicitly defined when such a definition is used.

At the beginning of its operation a Turing machine is assumed to be in the initial state s_0 and to be positioned over the leftmost nonblank symbol on the tape. If the tape is all blank, the control head can be positioned over any cell. We will call the positioning of the control head over the leftmost nonblank tape symbol the *initial position* of the machine.

Example 1 illustrates how a Turing machine works.

EXAMPLE 1 Extra Examples

What is the final tape when the Turing machine T defined by the seven five-tuples $(s_0, 0, s_0, 0, R)$, $(s_0, 1, s_1, 1, R)$, (s_0, B, s_3, B, R), $(s_1, 0, s_0, 0, R)$, $(s_1, 1, s_2, 0, L)$ (s_1, B, s_3, B, R), and $(s_2, 1, s_3, 0, R)$ is run on the tape shown in Figure 2(a)?

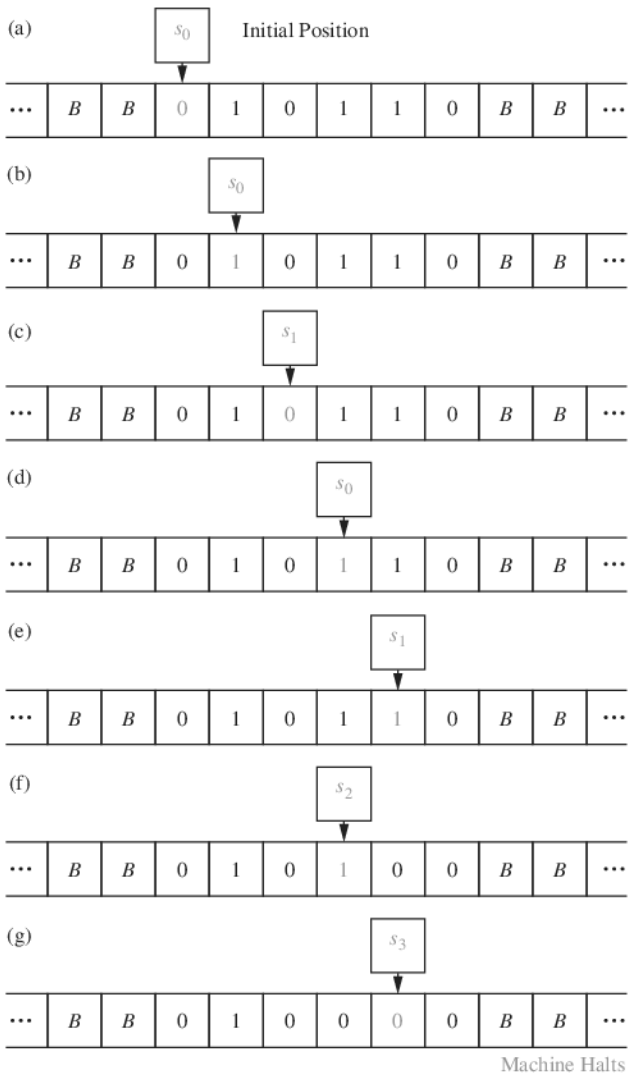


FIGURE 2 **The Steps Produced by Running T on the Tape in Figure 1.**

Solution: We start the operation with T in state s_0 and with T positioned over the leftmost nonblank symbol on the tape. The first step, using the five-tuple $(s_0, 0, s_0, 0, R)$, reads the 0 in the leftmost nonblank cell, stays in state s_0, writes a 0 in this cell, and moves one cell right. The second step, using the five-tuple $(s_0, 1, s_1, 1, R)$, reads the 1 in the current cell, enters state s_1, writes a 1 in this cell, and moves one cell right. The third step, using the five-tuple $(s_1, 0, s_0, 0, R)$, reads the 0 in the current cell, enters state s_0, writes a 0 in this cell, and moves one cell right. The fourth step, using the five-tuple $(s_0, 1, s_1, 1, R)$, reads the 1 in the current cell, enters state s_1, writes a 1 in this cell, and moves right one cell. The fifth step, using the five-tuple $(s_1, 1, s_2, 0, L)$, reads the 1 in the current cell, enters state s_2, writes a 0 in this cell, and moves left one cell. The sixth step, using the five-tuple $(s_2, 1, s_3, 0, R)$, reads the 1 in the current cell, enters the state s_3, writes a 0 in this cell, and moves right one cell. Finally, in the seventh step, the machine halts because there is no five-tuple beginning with the pair $(s_3, 0)$ in the description of the machine. The steps are shown in Figure 2.

Note that T changes the first pair of consecutive 1s on the tape to 0s and then halts. ◀

Using Turing Machines to Recognize Sets

Turing machines can be used to recognize sets. To do so requires that we define the concept of a final state as follows. A *final state* of a Turing machine T is a state that is not the first state in any five-tuple in the description of T using five-tuples (for example, state s_3 in Example 1).

We can now define what it means for a Turing machine to recognize a string. Given a string, we write consecutive symbols in this string in consecutive cells.

DEFINITION 2

Let V be a subset of an alphabet I. A Turing machine $T = (S, I, f, s_0)$ *recognizes* a string x in V^* if and only if T, starting in the initial position when x is written on the tape, halts in a final state. T is said to recognize a subset A of V^* if x is recognized by T if and only if x belongs to A.

Note that to recognize a subset A of V^* we can use symbols not in V. This means that the input alphabet I may include symbols not in V. These extra symbols are often used as markers (see Example 3).

When does a Turing machine T *not* recognize a string x in V^*? The answer is that x is not recognized if T does not halt or halts in a state that is not final when it operates on a tape containing the symbols of x in consecutive cells, starting in the initial position. (The reader should understand that this is one of many possible ways to define how to recognize sets using Turing machines.)

We illustrate this concept with Example 2.

EXAMPLE 2 Find a Turing machine that recognizes the set of bit strings that have a 1 as their second bit, that is, the regular set $\mathbf{(0 \cup 1)1(0 \cup 1)^*}$.

Solution: We want a Turing machine that, starting at the leftmost nonblank tape cell, moves right, and determines whether the second symbol is a 1. If the second symbol is 1, the machine should move into a final state. If the second symbol is not a 1, the machine should not halt or it should halt in a nonfinal state.

To construct such a machine, we include the five-tuples $(s_0, 0, s_1, 0, R)$ and $(s_0, 1, s_1, 1, R)$ to read in the first symbol and put the Turing machine in state s_1. Next, we include the five-tuples $(s_1, 0, s_2, 0, R)$ and $(s_1, 1, s_3, 1, R)$ to read in the second symbol and either move to state s_2 if this symbol is a 0, or to state s_3 if this symbol is a 1. We do not want to recognize strings that have a 0 as their second bit, so s_2 should not be a final state. We want s_3 to be a final state. So, we can include the five-tuple $(s_2, 0, s_2, 0, R)$. Because we do not want to recognize the empty string or a string with one bit, we also include the five-tuples $(s_0, B, s_2, 0, R)$ and $(s_1, B, s_2, 0, R)$.

The Turing machine T consisting of the seven five-tuples listed here will terminate in the final state s_3 if and only if the bit string has at least two bits and the second bit of the input string is a 1. If the bit string contains fewer than two bits or if the second bit is not a 1, the machine will terminate in the nonfinal state s_2. ◀

Given a regular set, a Turing machine that always moves to the right can be built to recognize this set (as in Example 2). To build the Turing machine, first find a finite-state automaton that recognizes the set and then construct a Turing machine using the transition function of the finite-state machine, always moving to the right.

We will now show how to build a Turing machine that recognizes a nonregular set.

EXAMPLE 3 Find a Turing machine that recognizes the set $\{0^n 1^n \mid n \geq 1\}$.

Solution: To build such a machine, we will use an auxiliary tape symbol M as a marker. We have $V = \{0, 1\}$ and $I = \{0, 1, M\}$. We wish to recognize only a subset of strings in V^*. We will have one final state, s_6. The Turing machine successively replaces a 0 at the leftmost position of

the string with an M and a 1 at the rightmost position of the string with an M, sweeping back and forth, terminating in a final state if and only if the string consists of a block of 0s followed by a block of the same number of 1s.

Although this is easy to describe and is easily carried out by a Turing machine, the machine we need to use is somewhat complicated. We use the marker M to keep track of the leftmost and rightmost symbols we have already examined. The five-tuples we use are $(s_0, 0, s_1, M, R)$, $(s_1, 0, s_1, 0, R)$, $(s_1, 1, s_1, 1, R)$, (s_1, M, s_2, M, L), (s_1, B, s_2, B, L), $(s_2, 1, s_3, M, L)$, $(s_3, 1, s_3, 1, L)$, $(s_3, 0, s_4, 0, L)$, (s_3, M, s_5, M, R), $(s_4, 0, s_4, 0, L)$, (s_4, M, s_0, M, R), and (s_5, M, s_6, M, R). For example, the string 000111 would successively become $M00111$, $M0011M$, $MM011M$, $MM01MM$, $MMM1MM$, $MMMMMM$ as the machine operates until it halts. Only the changes are shown, as most steps leave the string unaltered.

We leave it to the reader (Exercise 13) to explain the actions of this Turing machine and to explain why it recognizes the set $\{0^n 1^n \mid n \geq 1\}$. ◀

It can be shown that a set can be recognized by a Turing machine if and only if it can be generated by a type 0 grammar, or in other words, if the set is generated by a phrase-structure grammar. The proof will not be presented here.

Computing Functions with Turing Machines

A Turing machine can be thought of as a computer that finds the values of a partial function. To see this, suppose that the Turing machine T, when given the string x as input, halts with the string y on its tape. We can then define $T(x) = y$. The domain of T is the set of strings for which T halts; $T(x)$ is undefined if T does not halt when given x as input. Thinking of a Turing machine as a machine that computes the values of a function on strings is useful, but how can we use Turing machines to compute functions defined on integers, on pairs of integers, on triples of integers, and so on?

To consider a Turing machine as a computer of functions from the set of k-tuples of nonnegative integers to the set of nonnegative integers (such functions are called **number-theoretic functions**), we need a way to represent k-tuples of integers on a tape. To do so, we use **unary representations** of integers. We represent the nonnegative integer n by a string of $n + 1$ 1s so that, for instance, 0 is represented by the string 1 and 5 is represented by the string 111111. To represent the k-tuple $(n_1, n_2, \ldots, n_k)$, we use a string of $n_1 + 1$ 1s, followed by an asterisk, followed by a string of $n_2 + 1$ 1s, followed by an asterisk, and so on, ending with a string of $n_k + 1$ 1s. For example, to represent the four-tuple (2, 0, 1, 3) we use the string$111 * 1 * 11 * 1111$.

We can now consider a Turing machine T as computing a sequence of number-theoretic functions $T, T^2, \ldots, T^k, \ldots$. The function T^k is defined by the action of T on k-tuples of integers represented by unary representations of integers separated by asterisks.

EXAMPLE 4 Construct a Turing machine for adding two nonnegative integers.

Solution: We need to build a Turing machine T that computes the function $f(n_1, n_2) = n_1 + n_2$. The pair (n_1, n_2) is represented by a string of $n_1 + 1$ 1s followed by an asterisk followed by $n_2 + 1$ 1s. The machine T should take this as input and produce as output a tape with $n_1 + n_2 + 1$ 1s. One way to do this is as follows. The machine starts at the leftmost 1 of the input string, and carries out steps to erase this 1, halting if $n_1 = 0$ so that there are no more 1s before the asterisk, replaces the asterisk with the leftmost remaining 1, and then halts. We can use these five-tuples to do this: $(s_0, 1, s_1, B, R)$, $(s_1, *, s_3, B, R)$, $(s_1, 1, s_2, B, R)$, $(s_2, 1, s_2, 1, R)$, and $(s_2, *, s_3, 1, R)$. ◀

Unfortunately, constructing Turing machines to compute relatively simple functions can be extremely demanding. For example, one Turing machine for multiplying two nonnegative integers found in many books has 31 five-tuples and 11 states. If it is challenging to construct Turing machines to compute even relatively simple functions, what hope do we have of building

Turing machines for more complicated functions? One way to simplify this problem is to use a multitape Turing machine that uses more than one tape simultaneously and to build up multitape Turing machines for the composition of functions. It can be shown that for any multitape Turing machine there is a one-tape Turing machine that can do the same thing.

Different Types of Turing Machines

There are many variations on the definition of a Turing machine. We can expand the capabilities of a Turing machine in a wide variety of ways. For example, we can allow a Turing machine to move right, left, or not at all at each step. We can allow a Turing machine to operate on multiple tapes, using $(2 + 3n)$-tuples to describe the Turing machine when n tapes are used. We can allow the tape to be two-dimensional, where at each step we move up, down, right, or left, not just right or left as we do on a one-dimensional tape. We can allow multiple tape heads that read different cells simultaneously. Furthermore, we can allow a Turing machine to be **nondeterministic**, by allowing a (state, tape symbol) pair to possibly appear as the first elements in more than one five-tuple of the Turing machine. We can also reduce the capabilities of a Turing machine in different ways. For example, we can restrict the tape to be infinite in only one dimension or we can restrict the tape alphabet to have only two symbols. All these variations of Turing machines have been studied in detail.

The crucial point is that no matter which of these variations we use, or even which combination of variations we use, we never increase or decrease the power of the machine. Anything that one of these variations can do can be done by the Turing machine defined in this section, and vice versa. The reason that these variations are useful is that sometimes they make doing some particular job much easier than if the Turing machine defined in Definition 1 were used. They never extend the capability of the machine. Sometimes it is useful to have a wide variety of Turing machines with which to work. For example, one way to show that for every nondeterministic Turing machine, there is a deterministic Turing machine that recognizes the same language is to use a deterministic Turing machine with three taps. (For details on variations of Turing machines and demonstrations of their equivalence, see [HoMoUl01].)

Besides introducing the notion of a Turing machine, Turing also showed that it is possible to construct a single Turing machine that can simulate the computations of every Turing machine when given an encoding of this target Turing machine and its input. Such a machine is called a **universal Turing machine**. (See a book on the theory of computation, such as [Si06], for more about universal Turing machines.)

The Church–Turing Thesis

Links

Turing machines are relatively simple. They can have only finitely many states and they can read and write only one symbol at a time on a one-dimensional tape. But it turns out that Turing machines are extremely powerful. We have seen that Turing machines can be built to add numbers and to multiply numbers. Although it may be difficult to actually construct a Turing machine to compute a particular function that can be computed with an algorithm, such a Turing machine can always be found. This was the original goal of Turing when he invented his machines. Furthermore, there is a tremendous amount of evidence for the **Church–Turing thesis**, which states that given any problem that can be solved with an effective algorithm, there is a Turing machine that can solve this problem. The reason this is called a *thesis* rather than a theorem is that the concept of solvability by an effective algorithm is informal and imprecise, as opposed to the notion of solvability by a Turing machine, which is formal and precise. Certainly, though, any problem that can be solved using a computer with a program written in any language, perhaps using an unlimited amount of memory, should be considered effectively solvable. (Note that Turing machines have unlimited memory, unlike computers in the real world, which have only a finite amount of memory.)

Many different formal theories have been developed to capture the notion of effective computability. These include Turing's theory and Church's lambda-calculus, as well as theories proposed by Stephen Kleene and by E. L. Post. These theories seem quite different on the surface. The surprising thing is that they can be shown to be equivalent by demonstrating that they define exactly the same class of functions. With this evidence, it seems that Turing's original ideas, formulated before the invention of modern computers, describe the ultimate capabilities of these machines. The interested reader should consult books on the theory of computation, such as [HoMoUl01] and [Si96], for a discussion of these different theories and their equivalence.

For the remainder of this section we will briefly explore some of the consequences of the Church–Turing thesis and we will describe the importance of Turing machines in the study of the complexity of algorithms. Our goal will be to introduce some important ideas from theoretical computer science to entice the interested student to further study. We will cover a lot of ground quickly without providing explicit details. Our discussion will also tie together some of the concepts discussed in previous parts of the book with the theory of computation.

Computational Complexity, Computability, and Decidability

Throughout this book we have discussed the computational complexity of a wide variety of problems. We described the complexity of these problems in terms of the number of operations used by the most efficient algorithms that solve them. The basic operations used by algorithms differ considerably; we have measured the complexity of different algorithms in terms of bit operations, comparisons of integers, arithmetic operations, and so on. In Section 3.3, we defined various classes of problems in terms of their computational complexity. However, these definitions were not precise, because the types of operations used to measure their complexity vary so drastically. Turing machines provide a way to make the concept of computational complexity precise. If the Church–Turing thesis is true, it would then follow that if a problem can be solved using an effective algorithm, then there is a Turing machine that can solve this problem. When a Turing machine is used to solve a problem, the input to the problem is encoded as a string of symbols that is written on the tape of this Turing machine. How we encode input depends on the domain of this input. For example, as we have seen, we can encode a positive integer using a string of 1s. We can also devise ways to express pairs of integers, negative integers, and so on. Similarly, for graph algorithms, we need a way to encode graphs as strings of symbols. This can be done in many ways and can be based on adjacency lists or adjacency matrices. (We omit the details of how this is done.) However, the way input is encoded does not matter as long as it is relatively efficient, as a Turing machine can always change one encoding into another encoding. We will now use this model to make precise some of the notions concerning computational complexity that were informally introduced in Section 3.3.

The kind of problems that are most easily studied by using Turing machines are those problems that can be answered either by a "yes" or by a "no."

DEFINITION 3

A *decision problem* asks whether statements from a particular class of statements are true. Decision problems are also known as *yes-or-no problems*.

Given a decision problem, we would like to know whether there is an algorithm that can determine whether statements from the class of statements it addresses are true. For example, consider the class of statements each of which asks whether a particular integer n is prime. This is a decision problem because the answer to the question "Is n prime?" is either yes or no. Given this decision problem, we can ask whether there is an algorithm that can decide whether each of the statements in the decision problem is true, that is, given an integer n, deciding whether n is prime. The answer is that there is such an algorithm. In particular, in Section 3.5 we discussed the algorithm that determines whether a positive integer n is prime by checking whether it is divisible by primes not exceeding its square root. (There are many other algorithms for determining whether a positive integer is prime.) The set of inputs for which the answer to

the yes–no problem is "yes" is a subset of the set of possible inputs, that is, it is a subset of the set of strings of the input alphabet. In other words, solving a yes–no problem is the same as recognizing the language consisting of all bit strings that represent input values to the problem leading to the answer "yes." Consequently, solving a yes–no problem is the same as recognizing the language corresponding to the input values for which the answer to the problem is "yes."

DECIDABILITY When there is an effective algorithm that decides whether instances of a decision problem are true, we say that this problem is **solvable** or **decidable**. For instance, the problem of determining whether a positive integer is prime is a solvable problem. However, if no effective algorithm exists for solving a problem, then we say the problem is **unsolvable** or **undecidable**. To show that a decision problem is solvable we need only construct an algorithm that can determine whether statements of the particular class are true. On the other hand, to show that a decision problem is unsolvable we need to prove that no such algorithm exists. (The fact that we tried to find such an algorithm but failed, does not prove the problem is unsolvable.)

By studying only decision problems, it may seem that we are studying only a small set of problems of interest. However, most problems can be recast as decision problems. Recasting the types of problems we have studied in this book as decision problems can be quite complicated, so we will not go into the details of this process here. The interested reader can consult references on the theory of computation, such as [Wo87], which, for example, explains how to recast the traveling salesperson problem (described in Section 9.6) as a decision problem. (To recast the traveling salesman problem as a decision problem, we first consider the decision problem that asks whether there is a Hamilton circuit of weight not exceeding k, where k is a positive integer. With some additional effort it is possible to use answers to this question for different values of k to find the smallest possible weight of a Hamilton circuit.)

In Section 3.1 we introduced the halting problem and proved that it is an unsolvable problem. That discussion was somewhat informal because the notion of a procedure was not precisely defined. A precise definition of the halting problem can be made in terms of Turing machines.

DEFINITION 4

The *halting problem* is the decision problem that asks whether a Turing machine T eventually halts when given an input string x.

With this definition of the halting problem, we have Theorem 1.

THEOREM 1

The halting problem is an unsolvable decision problem. That is, no Turing machine exists that, when given an encoding of a Turing machine T and its input string x as input, can determine whether T eventually halts when started with x written on its tape.

The proof of Theorem 1 given in Section 3.1 for the informal definition of the halting problem still applies here.

Other examples of unsolvable problems include:

(i) the problem of determining whether two context-free grammars generate the same set of strings;

(ii) the problem of determining whether a given set of tiles can be used with repetition allowed to cover the entire plane without overlap; and

(iii) *Hilbert's Tenth Problem*, which asks whether there are integer solutions to a given polynomial equation with integer coefficients. (This question occurs tenth on the famous list of 23 problems Hilbert posed in 1900. Hilbert envisioned that the work done to solve these problems would help further the progress of mathematics in the twentieth century. The unsolvability of Hilbert's Tenth Problem was established in 1970 by Yuri Matiyasevich.)

Links

COMPUTABILITY A function that can be computed by a Turing machine is called **computable** and a function that cannot be computed by a Turing machine is called **uncomputable**. It is fairly straightforward, using a countability argument, to show that there are number-theoretic functions that are not computable (see Exercise 39 in Section 2.5). However, it is not so easy to actually produce such a function. The **busy beaver function** defined in the preamble to Exercise 31 is an example of an uncomputable function. One way to show that the busy beaver function is not computable is to show that it grows faster than any computable function. (See Exercise 32.)

Note that every decision problem can be reformulated as the problem of computing a function, namely, the function that has the value 1 when the answer to the problem is "yes" and that has the value 0 when the answer to the problem is "no." A decision problem is solvable if and only if the corresponding function constructed in this way is computable.

THE CLASSES P AND NP In Section 3.3 we informally defined the classes of problems called P and NP. We are now able to define these concepts precisely using the notions of deterministic and nondeterministic Turing machines.

We first elaborate on the difference between a deterministic Turing machine and a nondeterministic Turing machine. The Turing machines we have studied in this section have all been deterministic. In a deterministic Turing machine $T = (S, I, f, s_0)$, transition rules are defined by the partial function f from $S \times I$ to $S \times I \times \{R, L\}$. Consequently, when transition rules of the machine are represented as five-tuples of the form (s, x, s', x', d), where s is the current state, x is the current tape symbol, s' is the next state, x' is the symbol that replaces x on the tape, and d is the direction the machine moves on the tape, no two transition rules begin with the same pair (s, x).

In a nondeterministic Turing machine, allowed steps are defined using a relation consisting of five-tuples rather than using a partial function. The restriction that no two transition rules begin with the same pair (s, x) is eliminated; that is, there may be more than one transition rule beginning with each (state, tape symbol) pair. Consequently, in a nondeterministic Turing machine, there is a choice of transitions for some pairs of the current state and the tape symbol being read. At each step of the operation of a nondeterministic Turing machine, the machine picks one of the different choices of the transition rules that begin with the current state and tape symbol pair. This choice can be considered to be a "guess" of which step to use. Just as for deterministic Turing machines, a nondeterministic Turing machine halts when there is no transition rule in its definition that begins with the current state and tape symbol. Given a nondeterministic Turing machine T, we say that a string x is recognized by T if and only if there exists some sequence of transitions of T that ends in a final state when the machine starts in the initial position with x written on the tape. The nondeterministic Turing machine T recognizes the set A if x is recognized by T if and only if $x \in A$. The nondeterministic Turing machine T is said to solve a decision problem if it recognizes the set consisting of all input values for which the answer to the decision problem is yes.

DEFINITION 5

A decision problem is in P, the *class of polynomial-time problems*, if it can be solved by a deterministic Turing machine in polynomial time in terms of the size of its input. That is, a decision problem is in P if there is a deterministic Turing machine T that solves the decision problem and a polynomial $p(n)$ such that for all integers n, T halts in a final state after no more than $p(n)$ transitions whenever the input to T is a string of length n. A decision problem is in *NP*, the *class of nondeterministic polynomial-time problems*, if it can be solved by a nondeterministic Turing machine in polynomial time in terms of the size of its input. That is, a decision problem is in NP if there is a nondeterministic Turing machine T that solves the problem and a polynomial $p(n)$ such that for all integers n, T halts for every choice of transitions after no more than $p(n)$ transitions whenever the input to T is a string of length n.

Problems in P are called **tractable**, whereas problems not in P are called **intractable**. For a problem to be in P, a deterministic Turing machine must exist that can decide in polynomial time whether a particular statement of the class addressed by the decision problem is true. For example, determining whether an item is in a list of n elements is a tractable problem. (We will not provide details on how this fact can be shown; the basic ideas used in the analyses of algorithms earlier in the text can be adapted when Turing machines are employed.) For a problem to be in NP, it is necessary only that there be a nondeterministic Turing machine that, when given a true statement from the set of statements addressed by the problem, can verify its truth in polynomial time by making the correct guess at each step from the set of allowable steps corresponding to the current state and tape symbol. The problem of determining whether a given graph has a Hamilton circuit is an NP problem, because a nondeterministic Turing machine can easily verify that a simple circuit in a graph passes through each vertex exactly once. It can do this by making a series of correct guesses corresponding to successively adding edges to form the circuit. Because every deterministic Turing machine can also be considered to be a nondeterministic Turing machine where each (state, tape symbol) pair occurs in exactly one transition rule defining the machine, every problem in P is also in NP. In symbols, $P \subseteq NP$.

One of the most perplexing open questions in theoretical computer science is whether every problem in NP is also in P, that is, whether $P = NP$. As mentioned in Section 3.3, there is an important class of problems, the class of NP-complete problems, such that a problem is in this class if it is in the class NP and if it can be shown that if it is also in the class P, then *every* problem in the class NP must also be in the class P. That is, a problem is NP-complete if the existence of a polynomial-time algorithm for solving it implies the existence of a polynomial-time algorithm for every problem in NP. In this book we have discussed several different NP-complete problems, such as determining whether a simple graph has a Hamilton circuit and determining whether a proposition in n-variables is a tautology.

Exercises

1. Let T be the Turing machine defined by the five-tuples: $(s_0, 0, s_1, 1, R)$, $(s_0, 1, s_1, 0, R)$, $(s_0, B, s_1, 0, R)$, $(s_1, 0, s_2, 1, L)$, $(s_1, 1, s_1, 0, R)$, and $(s_1, B, s_2, 0, L)$. For each of these initial tapes, determine the final tape when T halts, assuming that T begins in initial position.

a)

⋯	B	B	0	0	1	1	B	B	⋯

b)

⋯	B	B	1	0	1	B	B	B	⋯

c)

⋯	B	B	1	1	B	0	1	B	⋯

d)

⋯	B	B	B	B	B	B	B	B	⋯

2. Let T be the Turing machine defined by the five-tuples: $(s_0, 0, s_1, 0, R)$, $(s_0, 1, s_1, 0, L)$, $(s_0, B, s_1, 1, R)$, $(s_1, 0, s_2, 1, R)$, $(s_1, 1, s_1, 1, R)$, $(s_1, B, s_2, 0, R)$, and $(s_2, B, s_3, 0, R)$. For each of these initial tapes, determine the final tape when T halts, assuming that T begins in initial position.

a)

⋯	B	B	0	1	0	1	B	B	⋯

b)

⋯	B	B	1	1	1	B	B	B	⋯

c)

⋯	B	B	0	0	B	0	0	B	⋯

d)

⋯	B	B	B	B	B	B	B	B	⋯

Links

ALONZO CHURCH (1903–1995) Alonzo Church was born in Washington, D.C. He studied at Göttingen under Hilbert and later in Amsterdam. He was a member of the faculty at Princeton University from 1927 until 1967 when he moved to UCLA. Church was one of the founding members of the Association for Symbolic Logic. He made many substantial contributions to the theory of computability, including his solution to the decision problem, his invention of the lambda-calculus, and, of course, his statement of what is now known as the Church–Turing thesis. Among Church's students were Stephen Kleene and Alan Turing. He published articles past his 90th birthday.

3. What does the Turing machine described by the five-tuples $(s_0, 0, s_0, 0, R)$, $(s_0, 1, s_1, 0, R)$, (s_0, B, s_2, B, R), $(s_1, 0, s_1, 0, R)$, $(s_1, 1, s_0, 1, R)$, and (s_1, B, s_2, B, R) do when given

a) 11 as input?

b) an arbitrary bit string as input?

4. What does the Turing machine described by the five-tuples $(s_0, 0, s_0, 1, R)$, $(s_0, 1, s_0, 1, R)$, (s_0, B, s_1, B, L), $(s_1, 1, s_2, 1, R)$, do when given

a) 101 as input?

b) an arbitrary bit string as input?

5. What does the Turing machine described by the five-tuples $(s_0, 1, s_1, 0, R)$, $(s_1, 1, s_1, 1, R)$, $(s_1, 0, s_2, 0, R)$, $(s_2, 0, s_3, 1, L)$, $(s_2, 1, s_2, 1, R)$, $(s_3, 1, s_3, 1, L)$, $(s_3, 0, s_4, 0, L)$, $(s_4, 1, s_4, 1, L)$, and $(s_4, 0, s_0, 1, R)$ do when given

a) 11 as input?

b) a bit string consisting entirely of 1s as input?

6. Construct a Turing machine with tape symbols 0, 1, and B that, when given a bit string as input, adds a 1 to the end of the bit string and does not change any of the other symbols on the tape.

7. Construct a Turing machine with tape symbols 0, 1, and B that, when given a bit string as input, replaces the first 0 with a 1 and does not change any of the other symbols on the tape.

8. Construct a Turing machine with tape symbols 0, 1, and B that, given a bit string as input, replaces all 0s on the tape with 1s and does not change any of the 1s on the tape.

9. Construct a Turing machine with tape symbols 0, 1, and B that, given a bit string as input, replaces all but the leftmost 1 on the tape with 0s and does not change any of the other symbols on the tape.

10. Construct a Turing machine with tape symbols 0, 1, and B that, given a bit string as input, replaces the first two consecutive 1s on the tape with 0s and does not change any of the other symbols on the tape.

11. Construct a Turing machine that recognizes the set of all bit strings that end with a 0.

12. Construct a Turing machine that recognizes the set of all bit strings that contain at least two 1s.

13. Construct a Turing machine that recognizes the set of all bit strings that contain an even number of 1s.

14. Show at each step the contents of the tape of the Turing machine in Example 3 starting with each of these strings.

a) 0011 **b)** 00011 **c)** 101100 **d)** 000111

15. Explain why the Turing machine in Example 3 recognizes a bit string if and only if this string is of the form $0^n 1^n$ for some positive integer n.

***16.** Construct a Turing machine that recognizes the set $\{0^{2n} 1^n \mid n \geq 0\}$.

***17.** Construct a Turing machine that recognizes the set $\{0^n 1^n 2^n \mid n \geq 0\}$.

18. Construct a Turing machine that computes the function $f(n) = n + 2$ for all nonnegative integers n.

19. Construct a Turing machine that computes the function $f(n) = n - 3$ if $n \geq 3$ and $f(n) = 0$ for $n = 0, 1, 2$ for all nonnegative integers n.

20. Construct a Turing machine that computes the function $f(n) = n \bmod 3$ for every nonnegative integer n.

21. Construct a Turing machine that computes the function $f(n) = 3$ if $n \geq 5$ and $f(n) = 0$ if $n = 0, 1, 2, 3,$ or 4.

22. Construct a Turing machine that computes the function $f(n) = 2n$ for all nonnegative integers n.

23. Construct a Turing machine that computes the function $f(n) = 3n$ for all nonnegative integers n.

24. Construct a Turing machine that computes the function $f(n_1, n_2) = n_2 + 2$ for all pairs of nonnegative integers n_1 and n_2.

***25.** Construct a Turing machine that computes the function $f(n_1, n_2) = \min(n_1, n_2)$ for all nonnegative integers n_1 and n_2.

26. Construct a Turing machine that computes the function $f(n_1, n_2) = n_1 + n_2 + 1$ for all nonnegative integers n_1 and n_2.

Suppose that T_1 and T_2 are Turing machines with disjoint sets of states S_1 and S_2 and with transition functions f_1 and f_2, respectively. We can define the Turing machine T_1T_2, the **composite** of T_1 and T_2, as follows. The set of states of T_1T_2 is $S_1 \cup S_2$. T_1T_2 begins in the start state of T_1. It first executes the transitions of T_1 using f_1 up to, but not including, the step at which T_1 would halt. Then, for all moves for which T_1 halts, it executes the same transitions of T_1 except that it moves to the start state of T_2. From this point on, the moves of T_1T_2 are the same as the moves of T_2.

27. By finding the composite of the Turing machines you constructed in Exercises 18 and 22, construct a Turing machine that computes the function $f(n) = 2n + 2$.

28. By finding the composite of the Turing machines you constructed in Exercises 18 and 23, construct a Turing machine that computes the function $f(n) = 3(n + 2) = 3n + 6$.

29. Which of the following problems is a decision problem?

a) What is the smallest prime greater than n?

b) Is a graph G bipartite?

c) Given a set of strings, is there a finite-state automaton that recognizes this set of strings?

d) Given a checkerboard and a particular type of polyomino (see Section 1.8), can this checkerboard be tiled using polyominoes of this type?

30. Which of the following problems is a decision problem?

a) Is the sequence $a_1, a_2, \ldots, a_n$ of positive integers in increasing order?

b) Can the vertices of a simple graph G be colored using three colors so that no two adjacent vertices are the same color?

c) What is the vertex of highest degree in a graph G?

d) Given two finite-state machines, do these machines recognize the same language?

Let $B(n)$ be the maximum number of 1s that a Turing machine with n states with the alphabet $\{1, B\}$ may print on a tape that is initially blank. The problem of determining $B(n)$ for particular values of n is known as the **busy beaver problem**. This problem was first studied by Tibor Rado in 1962. Currently it is known that $B(2) = 4$, $B(3) = 6$, and $B(4) = 13$, but $B(n)$ is not known for $n \geq 5$. $B(n)$ grows rapidly; it is known that $B(5) \geq 4098$ and $B(6) \geq 3.5 \times 10^{18267}$.

***31.** Show that $B(2)$ is at least 4 by finding a Turing machine with two states and alphabet $\{1, B\}$ that halts with four consecutive 1s on the tape.

****32.** Show that the function $B(n)$ cannot be computed by any Turing machine. [*Hint:* Assume that there is a Turing machine that computes $B(n)$ in binary. Build a Turing machine T that, starting with a blank tape, writes n down in binary, computes $B(n)$ in binary, and converts $B(n)$ from binary to unary. Show that for sufficiently large n, the number of states of T is less than $B(n)$, leading to a contradiction.]

Key Terms and Results

TERMS

alphabet (or vocabulary): a set that contains elements used to form strings

language: a subset of the set of all strings over an alphabet

phrase-structure grammar (V, T, S, P): a description of a language containing an alphabet V, a set of terminal symbols T, a start symbol S, and a set of productions P

the production $w \rightarrow w_1$: w can be replaced by w_1 whenever it occurs in a string in the language

$w_1 \Rightarrow w_2$ (w_2 is directly derivable from w_1): w_2 can be obtained from w_1 using a production to replace a string in w_1 with another string

$w_1 \overset{*}{\Rightarrow} w_2$ (w_2 is derivable from w_1): w_2 can be obtained from w_1 using a sequence of productions to replace strings by other strings

type 0 grammar: any phrase-structure grammar

type 1 grammar: a phrase-structure grammar in which every production is of the form $w_1 \rightarrow w_2$, where $w_1 = lAr$ and $w_2 = lwr$, where $A \in N$, $l, r, w \in (N \cup T)*$ and $w \neq \lambda$, or $w_1 = S$ and $w_2 = \lambda$ as long as S is not on the right-hand side of another production

type 2, or context-free, grammar: a phrase-structure grammar in which every production is of the form $A \rightarrow w_1$, where A is a nonterminal symbol

type 3, or regular, grammar: a phrase-structure grammar where every production is of the form $A \rightarrow aB$, $A \rightarrow a$, or $S \rightarrow \lambda$, where A and B are nonterminal symbols, S is the start symbol, and a is a terminal symbol

derivation (or parse) tree: an ordered rooted tree where the root represents the starting symbol of a type 2 grammar, internal vertices represent nonterminals, leaves represent terminals, and the children of a vertex are the symbols on the right side of a production, in order from left to right, where the symbol represented by the parent is on the left-hand side

Backus–Naur form: a description of a context-free grammar in which all productions having the same nonterminal as their left-hand side are combined with the different right-hand sides of these productions, each separated by a bar, with nonterminal symbols enclosed in angular brackets and the symbol $\rightarrow$ replaced by ::=

finite-state machine (S, I, O, f, g, s_0) (or a Mealy machine): a six-tuple containing a set S of states, an input alphabet I, an output alphabet O, a transition function f that assigns a next state to every pair of a state and an input, an output function g that assigns an output to every pair of a state and an input, and a starting state s_0

AB (concatenation of A and B): the set of all strings formed by concatenating a string in A and a string in B in that order

A^* (Kleene closure of A): the set of all strings made up by concatenating arbitrarily many strings from A

deterministic finite-state automaton (S, I, f, s_0, F): a five-tuple containing a set S of states, an input alphabet I, a transition function f that assigns a next state to every pair of a state and an input, a starting state s_0, and a set of final states F

nondeterministic finite-state automaton (S, I, f, s_0, F): a five-tuple containing a set S of states, an input alphabet I, a transition function f that assigns a set of possible next states to every pair of a state and an input, a starting state s_0, and a set of final states F

language recognized by an automaton: the set of input strings that take the start state to a final state of the automaton

regular expression: an expression defined recursively by specifying that $\emptyset$, λ, and x, for all x in the input alphabet, are regular expressions, and that $(\mathbf{AB})$, $(\mathbf{A} \cup \mathbf{B})$, and $\mathbf{A}^*$ are regular expressions when $\mathbf{A}$ and $\mathbf{B}$ are regular expressions

regular set: a set defined by a regular expression (see page 820)

Turing machine $T = (S, I, f, s_0)$: a four-tuple consisting of a finite set S of states, an alphabet I containing the blank symbol B, a partial function f from $S \times I$ to $S \times I \times \{R, L\}$, and a starting state s_0

nondeterministic Turing machine: a Turing machine that may have more than one transition rule corresponding to each (state, tape symbol) pair

decision problem: a problem that asks whether statements from a particular class of statements are true

solvable problem: a problem with the property that there is an effective algorithm that can solve all instances of the problem

unsolvable problem: a problem with the property that no effective algorithm exists that can solve all instances of the problem

computable function: a function whose values can be computed using a Turing machine

uncomputable function: a function whose values cannot be computed using a Turing machine

P, the class of polynomial-time problems: the class of problems that can be solved by a deterministic Turing machine in polynomial time in terms of the size of the input

NP, the class of nondeterministic polynomial-time problems: the class of problems that can be solved by a nondeterministic Turing machine in polynomial time in terms of the size of the input

NP-complete: a subset of the class of NP problems with the property that if any one of them is in the class P, then all problems in NP are in the class P

RESULTS

For every nondeterministic finite-state automaton there is a deterministic finite-state automaton that recognizes the same set.

Kleene's theorem: A set is regular if and only if there is a finite-state automaton that recognizes it.

A set is regular if and only if it is generated by a regular grammar.

The halting problem is unsolvable.

Review Questions

1. a) Define a phrase-structure grammar.

b) What does it mean for a string to be derivable from a string w by a phrase-structure grammar G?

2. a) What is the language generated by a phrase-structure grammar G?

b) What is the language generated by the grammar G with vocabulary $\{S, 0, 1\}$, set of terminals $T = \{0, 1\}$, starting symbol S, and productions $S \to 000S$, $S \to 1$?

c) Give a phrase-structure grammar that generates the set $\{01^n \mid n = 0, 1, 2, \ldots\}$.

3. a) Define a type 1 grammar.

b) Give an example of a grammar that is not a type 1 grammar.

c) Define a type 2 grammar.

d) Give an example of a grammar that is not a type 2 grammar but is a type 1 grammar.

e) Define a type 3 grammar.

f) Give an example of a grammar that is not a type 3 grammar but is a type 2 grammar.

4. a) Define a regular grammar.

b) Define a regular language.

c) Show that the set $\{0^m 1^n \mid m, n = 0, 1, 2, \ldots\}$ is a regular language.

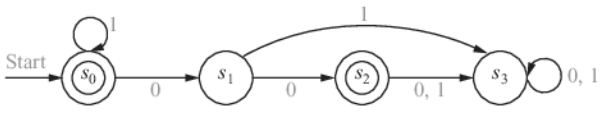
5. a) What is Backus–Naur form?

b) Give an example of the Backus–Naur form of the grammar for a subset of English of your choice.

6. a) What is a finite-state machine?

b) Show how a vending machine that accepts only quarters and dispenses a soft drink after 75 cents has been deposited can be modeled using a finite-state machine.

7. Find the set of strings recognized by the deterministic finite-state automaton shown here.



8. Construct a deterministic finite-state automaton that recognizes the set of bit strings that start with 1 and end with 1.

9. a) What is the Kleene closure of a set of strings?

b) Find the Kleene closure of the set $\{11, 0\}$.

10. a) Define a finite-state automaton.

b) What does it mean for a string to be recognized by a finite-state automaton?

11. a) Define a nondeterministic finite-state automaton.

b) Show that given a nondeterministic finite-state automaton, there is a deterministic finite-state automaton that recognizes the same language.

12. a) Define the set of regular expressions over a set I.

b) Explain how regular expressions are used to represent regular sets.

13. State Kleene's theorem.

14. Show that a set is generated by a regular grammar if and only if it is a regular set.

15. Give an example of a set not recognized by a finite-state automaton. Show that no finite-state automaton recognizes it.

16. Define a Turing machine.

17. Describe how Turing machines are used to recognize sets.

18. Describe how Turing machines are used to compute number-theoretic functions.

19. What is an unsolvable decision problem? Give an example of such a problem.

Supplementary Exercises

***1.** Find a phrase-structure grammar that generates each of these languages.
- **a)** the set of bit strings of the form $0^{2n}1^{3n}$, where n is a nonnegative integer
- **b)** the set of bit strings with twice as many 0s as 1s
- **c)** the set of bit strings of the form w^2, where w is a bit string

***2.** Find a phrase-structure grammar that generates the set $\{0^{2^n} \mid n \geq 0\}$.

For Exercises 3 and 4, let $G = (V, T, S, P)$ be the context-free grammar with $V = \{(,), S, A, B\}$, $T = \{(,)\}$, starting symbol S, and productions $S \to A, A \to AB, A \to B, B \to (A)$, and $B \to ()$, $S \to \lambda$.

3. Construct the derivation trees of these strings.
- **a)** (())
- **b)** ()(())
- **c)** ((()()))

***4.** Show that $L(G)$ is the set of all balanced strings of parentheses, defined in the preamble to Supplementary Exercise 55 in Chapter 4.

A context-free grammar is **ambiguous** if there is a word in $L(G)$ with two derivations that produce different derivation trees, considered as ordered, rooted trees.

5. Show that the grammar $G = (V, T, S, P)$ with $V = \{0, S\}$, $T = \{0\}$, starting state S, and productions $S \to 0S$, $S \to S0$, and $S \to 0$ is ambiguous by constructing two different derivation trees for 0^3.

6. Show that the grammar $G = (V, T, S, P)$ with $V = \{0, S\}$, $T = \{0\}$, starting state S, and productions $S \to 0S$ and $S \to 0$ is unambiguous.

7. Suppose that A and B are finite subsets of V^*, where V is an alphabet. Is it necessarily true that $|AB| = |BA|$?

8. Prove or disprove each of these statements for subsets A, B, and C of V^*, where V is an alphabet.
- **a)** $A(B \cup C) = AB \cup AC$
- **b)** $A(B \cap C) = AB \cap AC$
- **c)** $(AB)C = A(BC)$
- **d)** $(A \cup B)^* = A^* \cup B^*$

9. Suppose that A and B are subsets of V^*, where V is an alphabet. Does it follow that $A \subseteq B$ if $A^* \subseteq B^*$?

10. What set of strings with symbols in the set $\{0, 1, 2\}$ is represented by the regular expression $(\mathbf{2^*})(\mathbf{0} \cup (\mathbf{12^*}))^*$?

The **star height** $h(\mathbf{E})$ of a regular expression over the set I is defined recursively by

$h(\emptyset) = 0$;
$h(\mathbf{x}) = 0$ if $\mathbf{x} \in I$;
$h((\mathbf{E}_1 \cup \mathbf{E}_2)) = h((\mathbf{E}_1\mathbf{E}_2)) = \max(h(\mathbf{E}_1), h(\mathbf{E}_2))$ if $\mathbf{E}_1$ and $\mathbf{E}_2$ are regular expressions;
$h(\mathbf{E}^*) = h(\mathbf{E}) + 1$ if $\mathbf{E}$ is a regular expression.

11. Find the star height of each of these regular expressions.
- **a)** $\mathbf{0^*1}$
- **b)** $\mathbf{0^*1^*}$
- **c)** $\mathbf{(0^*01)^*}$
- **d)** $\mathbf{((0^*1)^*)^*}$
- **e)** $\mathbf{(010^*)(1^*01^*)^*((01)^*(10)^*)^*}$
- **f)** $\mathbf{(((((0^*)1)^*0)^*)1)^*}$

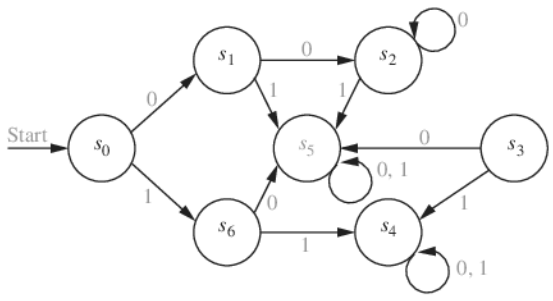
***12.** For each of these regular expressions find a regular expression that represents the same language with minimum star height.
- **a)** $\mathbf{(0^*1^*)^*}$
- **b)** $\mathbf{(0(01^*0)^*)^*}$
- **c)** $\mathbf{(0^* \cup (01)^* \cup 1^*)^*}$

13. Construct a finite-state machine with output that produces an output of 1 if the bit string read so far as input contains four or more 1s. Then construct a deterministic finite-state automaton that recognizes this set.

14. Construct a finite-state machine with output that produces an output of 1 if the bit string read so far as input contains four or more consecutive 1s. Then construct a deterministic finite-state automaton that recognizes this set.

15. Construct a finite-state machine with output that produces an output of 1 if the bit string read so far as input ends with four or more consecutive 1s. Then construct a deterministic finite-state automaton that recognizes this set.

16. A state s' in a finite-state machine is said to be **reachable** from state s if there is an input string x such that $f(s, x) = s'$. A state s is called **transient** if there is no nonempty input string x with $f(s, x) = s$. A state s is called a **sink** if $f(s, x) = s$ for all input strings x. Answer these questions about the finite-state machine with the state diagram illustrated here.



- **a)** Which states are reachable from s_0?
- **b)** Which states are reachable from s_2?
- **c)** Which states are transient?
- **d)** Which states are sinks?

***17.** Suppose that S, I, and O are finite sets such that $|S| = n$, $|I| = k$, and $|O| = m$.
- **a)** How many different finite-state machines (Mealy machines) $M = (S, I, O, f, g, s_0)$ can be constructed, where the starting state s_0 can be arbitrarily chosen?
- **b)** How many different Moore machines $M = (S, I, O, f, g, s_0)$ can be constructed, where the starting state s_0 can be arbitrarily chosen?

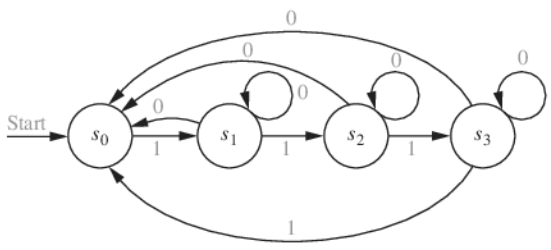
***18.** Suppose that S and I are finite sets such that $|S| = n$ and $|I| = k$. How many different finite-state automata $M = (S, I, f, s_0, F)$ are there where the starting state s_0

and the subset F of S consisting of final states can be chosen arbitrarily

a) if the automata are deterministic?

b) if the automata may be nondeterministic? (*Note:* This includes deterministic automata.)

19. Construct a deterministic finite-state automaton that is equivalent to the nondeterministic automaton with the state diagram shown here.



20. What is the language recognized by the automaton in Exercise 19?

21. Construct finite-state automata that recognize these sets.

a) $\mathbf{0^*(10)^*}$

b) $\mathbf{(01 \cup 111)^*10^*(0 \cup 1)}$

c) $\mathbf{(001 \cup (11)^*)^*}$

***22.** Find regular expressions that represent the set of all strings of 0s and 1s

a) made up of blocks of even numbers of 1s interspersed with odd numbers of 0s.

b) with at least two consecutive 0s or three consecutive 1s.

c) with no three consecutive 0s or two consecutive 1s.

***23.** Show that if A is a regular set, then so is $\overline{A}$.

***24.** Show that if A and B are regular sets, then so is $A \cap B$.

***25.** Find finite-state automata that recognize these sets of strings of 0s and 1s.

a) the set of all strings that start with no more than three consecutive 0s and contain at least two consecutive 1s

b) the set of all strings with an even number of symbols that do not contain the pattern 101

c) the set of all strings with at least three blocks of two or more 1s and at least two 0s

***26.** Show that $\{0^{2^n} \mid n \in \mathbf{N}\}$ is not regular. You may use the pumping lemma given in Exercise 22 of Section 13.4.

***27.** Show that $\{1^p \mid p \text{ is prime}\}$ is not regular. You may use the pumping lemma given in Exercise 22 of Section 13.4.

***28.** There is a result for context-free languages analogous to the pumping lemma for regular sets. Suppose that $L(G)$ is the language recognized by a context-free language G. This result states that there is a constant N such that if z is a word in $L(G)$ with $l(z) \geq N$, then z can be written as $uvwxy$, where $l(vwx) \leq N$, $l(vx) \geq 1$, and uv^iwx^iy belongs to $L(G)$ for $i = 0, 1, 2, 3, \ldots$. Use this result to show that there is no context-free grammar G with $L(G) = \{0^n1^n2^n \mid n = 0, 1, 2, \ldots\}$.

***29.** Construct a Turing machine that computes the function $f(n_1, n_2) = \max(n_1, n_2)$.

***30.** Construct a Turing machine that computes the function $f(n_1, n_2) = n_2 - n_1$ if $n_2 \geq n_1$ and $f(n_1, n_2) = 0$ if $n_2 < n_1$.

Computer Projects

Write programs with these input and output.

1. Given the productions in a phrase-structure grammar, determine which type of grammar this is in the Chomsky classification scheme.

2. Given the productions of a phrase-structure grammar, find all strings that are generated using twenty or fewer applications of its production rules.

3. Given the Backus–Naur form of a type 2 grammar, find all strings that are generated using twenty or fewer applications of the rules defining it.

***4.** Given the productions of a context-free grammar and a string, produce a derivation tree for this string if it is in the language generated by this grammar.

5. Given the state table of a Moore machine and an input string, produce the output string generated by the machine.

6. Given the state table of a Mealy machine and an input string, produce the output string generated by the machine.

7. Given the state table of a deterministic finite-state automaton and a string, decide whether this string is recognized by the automaton.

8. Given the state table of a nondeterministic finite-state automaton and a string, decide whether this string is recognized by the automaton.

***9.** Given the state table of a nondeterministic finite-state automaton, construct the state table of a deterministic finite-state automaton that recognizes the same language.

****10.** Given a regular expression, construct a nondeterministic finite-state automaton that recognizes the set that this expression represents.

11. Given a regular grammar, construct a finite-state automaton that recognizes the language generated by this grammar.

12. Given a finite-state automaton, construct a regular grammar that generates the language recognized by this automaton.

***13.** Given a Turing machine, find the output string produced by a given input string.

Computations and Explorations

Use a computational program or programs you have written to do these exercises.

1. Solve the busy beaver problem for two states by testing all possible Turing machines with two states and alphabet $\{1, B\}$.

***2.** Solve the busy beaver problem for three states by testing all possible Turing machines with three states and alphabet $\{1, B\}$.

****3.** Find a busy beaver machine with four states by testing all possible Turing machines with four states and alphabet $\{1, B\}$.

****4.** Make as much progress as you can toward finding a busy beaver machine with five states.

****5.** Make as much progress as you can toward finding a busy beaver machine with six states.

Writing Projects

Respond to these with essays using outside sources.

1. Describe how the growth of certain types of plants can be modeled using a Lidenmeyer system. Such a system uses a grammar with productions modeling the different ways plants can grow.

2. Describe the Backus–Naur form (and extended Backus–Naur form) rules used to specify the syntax of a programming language, such as Java, LISP, or Ada, or the database language SQL.

3. Explain how finite-state machines are used by spell-checkers.

4. Explain how finite-state machines are used in the study of network protocols.

5. Explain how finite-state machines are used in speech recognition programs.

6. Compare the use of Moore machines versus Mealy machines in the design of hardware systems and computer software.

7. Explain the concept of minimizing finite-state automata. Give an algorithm that carries out this minimization.

8. Give the definition of cellular automata. Explain their applications. Use the Game of Life as an example.

9. Define a pushdown automaton. Explain how pushdown automata are used to recognize sets. Which sets are recognized by pushdown automata? Provide an outline of a proof justifying your answer.

10. Define a linear-bounded automaton. Explain how linear-bounded automata are used to recognize sets. Which sets are recognized by linear-bounded automata? Provide an outline of a proof justifying your answer.

11. Look up Turing's original definition of what we now call a Turing machine. What was his motivation for defining these machines?

12. Describe the concept of the universal Turing machine. Explain how such a machine can be built.

13. Explain the kinds of applications in which nondeterministic Turing machines are used instead of deterministic Turing machines.

14. Show that a Turing machine can simulate any action of a nondeterministic Turing machine.

15. Show that a set is recognized by a Turing machine if and only if it is generated by a phrase-structure grammar.

16. Describe the basic concepts of the lambda-calculus and explain how it is used to study computability of functions.

17. Show that a Turing machine as defined in this chapter can do anything a Turing machine with n tapes can do.

18. Show that a Turing machine with a tape infinite in one direction can do anything a Turing machine with a tape infinite in both directions can do.

APPENDIX

1 Axioms for the Real Numbers and the Positive Integers

In this book we have assumed an explicit set of axioms for the set of real numbers and for the set of positive integers. In this appendix we will list these axioms and we will illustrate how basic facts, also used without proof in the text, can be derived using them.

Axioms for Real Numbers

The standard axioms for real numbers include both the **field (or algebraic) axioms**, used to specify rules for basic arithmetic operations, and the **order axioms**, used to specify properties of the ordering of real numbers.

THE FIELD AXIOMS We begin with the field axioms. As usual, we denote the sum and product of two real numbers x and y by $x + y$ and $x \cdot y$, respectively. (Note that the product of x and y is often denoted by xy without the use of the dot to indicate multiplication. We will not use this abridged notation in this appendix, but will within the text.) Also, by convention, we perform multiplications before additions unless parentheses are used. Although these statements are axioms, they are commonly called *laws* or *rules*. The first two of these axioms tell us that when we add or multiply two real numbers, the result is again a real number; these are the *closure laws*.

- **Closure law for addition** For all real numbers x and y, $x + y$ is a real number.
- **Closure law for multiplication** For all real numbers x and y, $x \cdot y$ is a real number.

The next two axioms tell us that when we add or multiply three real numbers, we get the same result regardless of the order of operations; these are the *associative laws*.

- **Associative law for addition** For all real numbers x, y, and z, $(x + y) + z = x + (y + z)$.
- **Associative law for multiplication** For all real numbers x, y, and z, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.

Two additional algebraic axioms tell us that the order in which we add or multiply two numbers does not matter; these are the *commutative laws*.

- **Commutative law for addition** For all real numbers x and y, $x + y = y + x$.
- **Commutative law for multiplication** For all real numbers x and y, $x \cdot y = y \cdot x$.

The next two axioms tell us that 0 and 1 are additive and multiplicative identities for the set of real numbers. That is, when we add 0 to a real number or multiply a real number by 1 we do not change this real number. These laws are called *identity laws*.

- **Additive identity law** For every real number x, $x + 0 = 0 + x = x$.
- **Multiplicative identity law** For every real number x, $x \cdot 1 = 1 \cdot x = x$.

Although it seems obvious, we also need the following axiom.

- **Identity elements axiom** The additive identity 0 and the multiplicative identity 1 are distinct, that is $0 \neq 1$.

Two additional axioms tell us that for every real number, there is a real number that can be added to this number to produce 0, and for every nonzero real number, there is a real number by which it can be multiplied to produce 1. These are the *inverse laws*.

- **Inverse law for addition** For every real number x, there exists a real number $-x$ (called the *additive inverse* of x) such that $x + (-x) = (-x) + x = 0$.
- **Inverse law for multiplication** For every nonzero real number x, there exists a real number $1/x$ (called the *multiplicative inverse* of x) such that $x \cdot (1/x) = (1/x) \cdot x = 1$.

The final algebraic axioms for real numbers are the *distributive laws*, which tell us that multiplication distributes over addition; that is, that we obtain the same result when we first add a pair of real numbers and then multiply by a third real number or when we multiply each of these two real numbers by the third real number and then add the two products.

- **Distributive laws** For all real numbers x, y, and z, $x \cdot (y + z) = x \cdot y + x \cdot z$ and $(x + y) \cdot z = x \cdot z + y \cdot z$.

ORDER AXIOMS Next, we will state the *order axioms* for the real numbers, which specify properties of the "greater than" relation, denoted by $>$, on the set of real numbers. We write $x > y$ (and $y < x$) when x is greater than y, and we write $x \geq y$ (and $y \leq x$) when $x > y$ or $x = y$. The first of these axioms tells us that given two real numbers, exactly one of three possibilities occurs: the two numbers are equal, the first is greater than the second, or the second is greater than the first. This rule is called the *trichotomy law*.

- **Trichotomy law** For all real numbers x and y, exactly one of $x = y$, $x > y$, or $y > x$ is true.

Next, we have an axiom, called *the transitivity law*, that tells us that if one number is greater than a second number and this second number is greater than a third, then the first number is greater than the third.

- **Transitivity law** For all real numbers x, y, and z, if $x > y$ and $y > z$, then $x > z$.

We also have two *compatibility laws*, which tell us that when we add a number to both sides in a greater than relationship, the greater than relationship is preserved and when we multiply both sides of a greater than relationship by a *positive real number* (that is, a real number x with $x > 0$), the greater than relationship is preserved.

- **Additive compatibility law** For all real numbers x, y, and z, if $x > y$, then $x + z > y + z$.
- **Multiplicative compatibility law** For all real numbers x, y, and z, if $x > y$ and $z > 0$, then $x \cdot z > y \cdot z$.

We leave it to the reader (see Exercise 15) to prove that for all real numbers x, y, and z, if $x > y$ and $z < 0$, then $x \cdot z < y \cdot z$. That is, multiplication of an inequality by a negative real number reverses the direction of the inequality.

The final axiom for the set of real numbers is the *completeness property*. Before we state this axiom, we need some definitions. First, given a nonempty set A of real numbers, we say that the real number b is an **upper bound** of A if for every real number a in A, $b \geq a$. A real number s is a **least upper bound** of A if s is an upper bound of A and whenever t is an upper bound of A, then we have $s \leq t$.

- **Completeness property** Every nonempty set of real numbers that is bounded above has a least upper bound.

Using Axioms to Prove Basic Facts

The axioms we have listed can be used to prove many properties that are often used without explicit mention. We give several examples of results we can prove using axioms and leave the proof of a variety of other properties as exercises. Although the results we will prove seem quite obvious, proving them using only the axioms we have stated can be challenging.

THEOREM 1 The additive identity element 0 of the real numbers is unique.

Proof: To show that the additive identity element 0 of the real numbers is unique, suppose that $0'$ is also an additive identity for the real numbers. This means that $0' + x = x + 0' = x$ whenever x is a real number. By the additive identity law, it follows that $0 + 0' = 0'$. Because $0'$ is an additive identity, we know that $0 + 0' = 0$. It follows that $0 = 0'$, because both equal $0 + 0'$. This shows that 0 is the unique additive identity for the real numbers. ◁

THEOREM 2 The additive inverse of a real number x is unique.

Proof: Let x be a real number. Suppose that y and z are both additive inverses of x. Then,

$$\begin{aligned}
y &= 0 + y && \text{by the additive identity law} \\
&= (z + x) + y && \text{because } z \text{ is an additive inverse of } x \\
&= z + (x + y) && \text{by the associative law for addition} \\
&= z + 0 && \text{because } y \text{ is an additive inverse of } x \\
&= z && \text{by the additive identity law.}
\end{aligned}$$

It follows that $y = z$. ◁

Theorems 1 and 2 tell us that the additive identity and additive inverses are unique. Theorems 3 and 4 tell us that the multiplicative identity and multiplicative inverses of nonzero real numbers are also unique. We leave their proofs as exercises.

THEOREM 3 The multiplicative identity element 1 of the real numbers is unique.

THEOREM 4 The multiplicative inverse of a nonzero real number x is unique.

THEOREM 5 For every real number x, $x \cdot 0 = 0$.

Proof: Suppose that x is a real number. By the additive inverse law, there is a real number y that is the additive inverse of $x \cdot 0$, so we have $x \cdot 0 + y = 0$. By the additive identity law, $0 + 0 = 0$. Using the distributive law, we see that $x \cdot 0 = x \cdot (0 + 0) = x \cdot 0 + x \cdot 0$. It follows that

$$0 = x \cdot 0 + y = (x \cdot 0 + x \cdot 0) + y.$$

Next, note that by the associative law for addition and because $x \cdot 0 + y = 0$, it follows that

$$(x \cdot 0 + x \cdot 0) + y = x \cdot 0 + (x \cdot 0 + y) = x \cdot 0 + 0.$$

Finally, by the additive identity law, we know that $x \cdot 0 + 0 = x \cdot 0$. Consequently, $x \cdot 0 = 0$. ◁

THEOREM 6 For all real numbers x and y, if $x \cdot y = 0$, then $x = 0$ or $y = 0$.

Proof: Suppose that x and y are real numbers and $x \cdot y = 0$. If $x \neq 0$, then, by the multiplicative inverse law, x has a multiplicative inverse $1/x$, such that $x \cdot (1/x) = (1/x) \cdot x = 1$. Because $x \cdot y = 0$, we have $(1/x) \cdot (x \cdot y) = (1/x) \cdot 0 = 0$ by Theorem 5. Using the associate law for multiplication, we have $((1/x) \cdot x) \cdot y = 0$. This means that $1 \cdot y = 0$. By the multiplicative identity rule, we see that $1 \cdot y = y$, so $y = 0$. Consequently, either $x = 0$ or $y = 0$. ◁

THEOREM 7 The multiplicative identity element 1 in the set of real numbers is greater than the additive identity element 0.

Proof: By the trichotomy law, either $0 = 1$, $0 > 1$, or $1 > 0$. We know by the identity elements axiom that $0 \neq 1$.

So, assume that $0 > 1$. We will show that this assumption leads to a contradiction. By the additive inverse law, 1 has an additive inverse -1 with $1 + (-1) = 0$. The additive compatibility law tells us that $0 + (-1) > 1 + (-1) = 0$; the additive identity law tells us that $0 + (-1) = -1$. Consequently, $-1 > 0$, and by the multiplicative compatibility law, $(-1) \cdot (-1) > (-1) \cdot 0$. By Theorem 5 the right-hand side of last inequality is 0. By the distributive law, $(-1) \cdot (-1) + (-1) \cdot 1 = (-1) \cdot (-1 + 1) = (-1) \cdot 0 = 0$. Hence, the left-hand side of this last inequality, $(-1) \cdot (-1)$, is the unique additive inverse of -1, so this side of the inequality equals 1. Consequently this last inequality becomes $1 > 0$, contradicting the trichotomy law because we had assumed that $0 > 1$.

Because we know that $0 \neq 1$ and that it is impossible for $0 > 1$, by the trichotomy law, we conclude that $1 > 0$. ◁

Links

ARCHIMEDES (287 B.C.E.–212 B.C.E.) Archimedes was one of the greatest scientists and mathematicians of ancient times. He was born in Syracuse, a Greek city-state in Sicily. His father, Phidias, was an astronomer. Archimedes was educated in Alexandria, Egypt. After completing his studies, he returned to Syracuse, where he spent the rest of his life. Little is known about his personal life; we do not know whether he was ever married or had children. Archimedes was killed in 212 B.C.E. by a Roman soldier when the Romans overran Syracuse.

Archimedes made many important discoveries in geometry. His method for computing the area under a curve was described two thousand years before his ideas were re-invented as part of integral calculus. Archimedes also developed a method for expressing large integers inexpressible by the usual Greek method. He discovered a method for computing the volume of a sphere, as well as of other solids, and he calculated an approximation of π. Archimedes was also an accomplished engineer and inventor; his machine for pumping water, now called *Archimedes' screw*, is still in use today. Perhaps his best known discovery is the *principle of buoyancy*, which tells us that an object submerged in liquid becomes lighter by an amount equal to the weight it displaces. Some histories tell us that Archimedes was an early streaker, running naked through the streets of Syracuse shouting "Eureka" (which means "I have found it") when he made this discovery. He is also known for his clever use of machines that held off Roman forces sieging Syracuse for several years during the Second Punic War.

The next theorem tells us that for every real number there is an integer (where by an *integer*, we mean 0, the sum of any number of 1s, and the additive inverses of these sums) greater than this real number. This result is attributed to the Greek mathematician Archimedes. The result can be found in Book V of Euclid's *Elements*.

THEOREM 8 **ARCHIMEDEAN PROPERTY** For every real number x there exists an integer n such that $n > x$.

Proof: Suppose that x is a real number such that $n \leq x$ for every integer n. Then x is an upper bound of the set of integers. By the completeness property it follows that the set of integers has a least upper bound M. Because $M - 1 < M$ and M is a least upper bound of the set of integers, $M - 1$ is not an upper bound of the set of integers. This means that there is an integer n with $n > M - 1$. This implies that $n + 1 > M$, contradicting the fact that M is an upper bound of the set of integers. ◁

Axioms for the Set of Positive Integers

The axioms we now list specify the set of positive integers as the subset of the set of integers satisfying four key properties. We assume the truth of these axioms in this textbook.

- **Axiom 1** The number 1 is a positive integer.
- **Axiom 2** If n is a positive integer, then $n + 1$, the *successor* of n, is also a positive integer.
- **Axiom 3** Every positive integer other than 1 is the successor of a positive integer.
- **Axiom 4** **The Well-Ordering Property** Every nonempty subset of the set of positive integers has a least element.

In Sections 5.1 and 5.2 it is shown that the well-ordering principle is equivalent to the principle of mathematical induction.

- **Mathematical induction axiom** If S is a set of positive integers such that $1 \in S$ and for all positive integers n if $n \in S$, then $n + 1 \in S$, then S is the set of positive integers.

Most mathematicians take the real number system as already existing, with the real numbers satisfying the axioms we have listed in this appendix. However, mathematicians in the nineteenth century developed techniques to construct the set of real numbers, starting with more basic sets of numbers. (The process of constructing the real numbers is sometimes studied in advanced undergraduate mathematics classes. A treatment of this can be found in [Mo91], for instance.) The first step in the process is the construction of the set of positive integers using axioms 1–3 and either the well-ordering property or the mathematical induction axiom. Then, the operations of addition and multiplication of positive integers are defined. Once this has been done, the set of integers can be constructed using equivalence classes of pairs of positive integers where $(a, b) \sim (c, d)$ if and only if $a + d = b + c$; addition and multiplication of integers can be defined using these pairs (see Exercise 21). (Equivalence relations and equivalence classes are discussed in Chapter 9.) Next, the set of rational numbers can be constructed using the equivalence classes of pairs of integers where the second integer in the pair is not zero, where $(a, b) \approx (c, d)$ if and only if $a \cdot d = b \cdot c$; addition and multiplication of rational numbers can be defined in terms of these pairs (see Exercise 22). Using infinite sequences, the set of real numbers can then be constructed from the set of rational numbers. The interested reader will find it worthwhile to read through the many details of the steps of this construction.

Exercises

Use only the axioms and theorems in this appendix in the proofs in your answers to these exercises.

1. Prove Theorem 3, which states that the multiplicative identity element of the real numbers is unique.

2. Prove Theorem 4, which states that for every nonzero real number x, the multiplicative inverse of x is unique.

3. Prove that for all real numbers x and y, $(-x) \cdot y = x \cdot (-y) = -(x \cdot y)$.

4. Prove that for all real numbers x and y, $-(x + y) = (-x) + (-y)$.

5. Prove that for all real numbers x and y, $(-x) \cdot (-y) = x \cdot y$.

6. Prove that for all real numbers x, y, and z, if $x + z = y + z$, then $x = y$.

7. Prove that for every real number x, $-(-x) = x$.

Define the **difference** $x - y$ of real numbers x and y by $x - y = x + (-y)$, where $-y$ is the additive inverse of y, and the **quotient** x/y, where $y \neq 0$, by $x/y = x \cdot (1/y)$, where $1/y$ is the multiplicative inverse of y.

8. Prove that for all real numbers x and y, $x = y$ if and only if $x - y = 0$.

9. Prove that for all real numbers x and y, $-x - y = -(x + y)$.

10. Prove that for all nonzero real numbers x and y, $1/(x/y) = y/x$, where $1/(x/y)$ is the multiplicative inverse of x/y.

11. Prove that for all real numbers w, x, y, and z, if $x \neq 0$ and $z \neq 0$, then $(w/x) + (y/z) = (w \cdot z + x \cdot y)/(x \cdot z)$.

12. Prove that for every positive real number x, $1/x$ is also a positive real number.

13. Prove that for all positive real numbers x and y, $x \cdot y$ is also a positive real number.

14. Prove that for all real numbers x and y, if $x > 0$ and $y < 0$, then $x \cdot y < 0$.

15. Prove that for all real numbers x, y, and z, if $x > y$ and $z < 0$, then $x \cdot z < y \cdot z$.

16. Prove that for every real number x, $x \neq 0$ if and only if $x^2 > 0$.

17. Prove that for all real numbers w, x, y, and z, if $w < x$ and $y < z$, then $w + y < x + z$.

18. Prove that for all positive real numbers x and y, if $x < y$, then $1/x > 1/y$.

19. Prove that for every positive real number x, there exists a positive integer n such that $n \cdot x > 1$.

***20.** Prove that between every two distinct real numbers there is a rational number (that is, a number of the form x/y, where x and y are integers with $y \neq 0$).

Exercises 21 and 22 involve the notion of an equivalence relation, discussed in Chapter 9 of the text.

***21.** Define a relation $\sim$ on the set of ordered pairs of positive integers by $(w, x) \sim (y, z)$ if and only if $w + z = x + y$. Show that the operations $[(w, x)]_\sim + [(y, z)]_\sim = [(w + y, x + z)]_\sim$ and $[(w, x)]_\sim \cdot [(y, z)]_\sim = [(w \cdot y + x \cdot z, x \cdot y + w \cdot z)]_\sim$ are well-defined, that is, they do not depend on the representative of the equivalence classes chosen for the computation.

***22.** Define a relation $\approx$ on ordered pairs of integers with second entry nonzero by $(w, x) \approx (y, z)$ if and only if $w \cdot z = x \cdot y$. Show that the operations $[(w, x)]_\approx + [(y, z)]_\approx = [(w \cdot z + x \cdot y, x \cdot z)]_\approx$ and $[(w, x)]_\approx \cdot [(y, z)]_\approx = [(w \cdot y, x \cdot z)]_\approx$ are well-defined, that is, they do not depend on the representative of the equivalence classes chosen for the computation.

APPENDIX

2 Exponential and Logarithmic Functions

In this appendix we review some of the basic properties of exponential functions and logarithms. These properties are used throughout the text. Students requiring further review of this material should consult precalculus or calculus books, such as those mentioned in the Suggested Readings.

Exponential Functions

Let n be a positive integer, and let b be a fixed positive real number. The function $f_b(n) = b^n$ is defined by

$$f_b(n) = b^n = b \cdot b \cdot b \cdot \cdots \cdot b,$$

where there are n factors of b multiplied together on the right-hand side of the equation.

We can define the function $f_b(x) = b^x$ for all real numbers x using techniques from calculus. The function $f_b(x) = b^x$ is called the **exponential function to the base *b***. We will not discuss how to find the values of exponential functions to the base b when x is not an integer.

Two of the important properties satisfied by exponential functions are given in Theorem 1. Proofs of these and other related properties can be found in calculus texts.

THEOREM 1 Let b be a positive real number and x and y real numbers. Then

1. $b^{x+y} = b^x b^y$, and
2. $(b^x)^y = b^{xy}$.

We display the graphs of some exponential functions in Figure 1.

Logarithmic Functions

Suppose that b is a real number with $b > 1$. Then the exponential function b^x is strictly increasing (a fact shown in calculus). It is a one-to-one correspondence from the set of real numbers to the set of nonnegative real numbers. Hence, this function has an inverse $\log_b x$, called the **logarithmic function to the base *b***. In other words, if b is a real number greater than 1 and x is a positive real number, then

$$b^{\log_b x} = x.$$

The value of this function at x is called the **logarithm of *x* to the base *b***.

From the definition, it follows that

$$\log_b b^x = x.$$

We give several important properties of logarithms in Theorem 2.

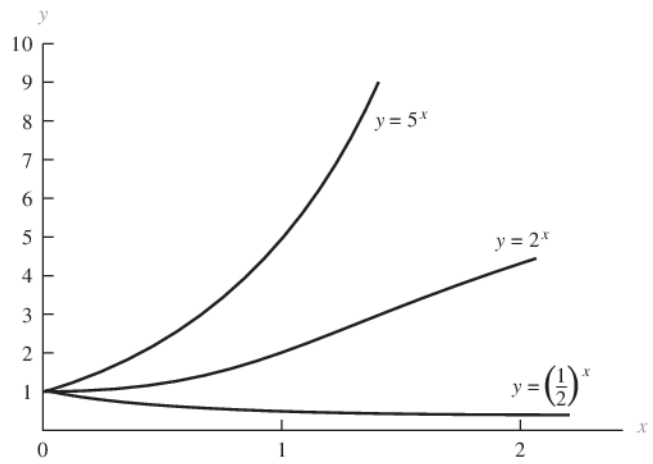


FIGURE 1 **Graphs of the Exponential Functions to the Bases $\frac{1}{2}$, 2, and 5.**

THEOREM 2

Let b be a real number greater than 1. Then

1. $\log_b(xy) = \log_b x + \log_b y$ whenever x and y are positive real numbers, and
2. $\log_b(x^y) = y \log_b x$ whenever x is a positive real number and y is a real number.

Proof: Because $\log_b(xy)$ is the unique real number with $b^{\log_b(xy)} = xy$, to prove part 1 it suffices to show that $b^{\log_b x + \log_b y} = xy$. By part 1 of Theorem 1, we have

$$\begin{aligned} b^{\log_b x + \log_b y} &= b^{\log_b x} b^{\log_b y} \\ &= xy. \end{aligned}$$

To prove part 2, it suffices to show that $b^{y \log_b x} = x^y$. By part 2 of Theorem 1, we have

$$\begin{aligned} b^{y \log_b x} &= (b^{\log_b x})^y \\ &= x^y. \end{aligned}$$

◁

The following theorem relates logarithms to two different bases.

THEOREM 3

Let a and b be real numbers greater than 1, and let x be a positive real number. Then

$$\log_a x = \log_b x / \log_b a.$$

Proof: To prove this result, it suffices to show that

$$b^{\log_a x \cdot \log_b a} = x.$$

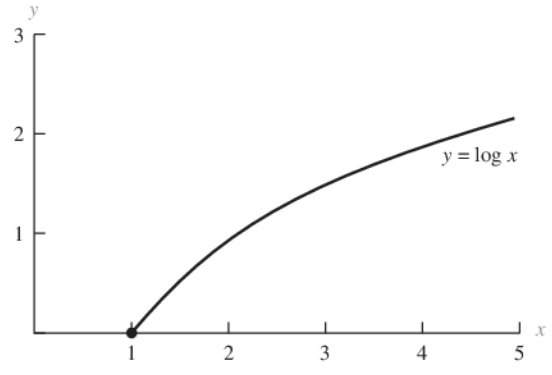


FIGURE 2 **The Graph of $f(x) = \log x$.**

By part 2 of Theorem 1, we have

$$\begin{aligned} b^{\log_a x \cdot \log_b a} &= (b^{\log_b a})^{\log_a x} \\ &= a^{\log_a x} \\ &= x. \end{aligned}$$

This completes the proof. ◁

Because the base used most often for logarithms in this text is $b = 2$, the notation $\log x$ is used throughout the test to denote $\log_2 x$.

The graph of the function $f(x) = \log x$ is displayed in Figure 2. From Theorem 3, when a base b other than 2 is used, a function that is a constant multiple of the function $\log x$, namely, $(1/\log b)\log x$, is obtained.

Exercises

1. Express each of the following quantities as powers of 2.
a) $2 \cdot 2^2$ **b)** $(2^2)^3$ **c)** $2^{(2^2)}$

2. Find each of the following quantities.
a) $\log_2 1024$ **b)** $\log_2 1/4$ **c)** $\log_4 8$

3. Suppose that $\log_4 x = y$ where x is a positive real number. Find each of the following quantities.
a) $\log_2 x$ **b)** $\log_8 x$ **c)** $\log_{16} x$

☞ **4.** Let a, b, and c be positive real numbers. Show that $a^{\log_b c} = c^{\log_b a}$.

5. Draw the graph of $f(x) = b^x$ for all real numbers x if b is
a) 3. **b)** 1/3. **c)** 1.

6. Draw the graph of $f(x) = \log_b x$ for positive real numbers x if b is
a) 4. **b)** 100. **c)** 1000.

APPENDIX

3 Pseudocode

The algorithms in this text are described both in English and in **pseudocode**. Pseudocode is an intermediate step between an English language description of the steps of a procedure and a specification of this procedure using an actual programming language. The advantages of using pseudocode include the simplicity with which it can be written and understood and the ease of producing actual computer code (in a variety of programming languages) from the pseudocode. We will describe the particular types of **statements**, or high-level instructions, of the pseudocode that we will use. Each of these statements in pseudocode can be translated into one or more statements in a particular programming language, which in turn can be translated into one or more (possibly many) low-level instructions for a computer.

This appendix describes the format and syntax of the pseudocode used in the text. This pseudocode is designed so that its basic structure resembles that of commonly used programming languages, such as C++ and Java, which are currently the most commonly taught programming languages. However, the pseudocode we use will be a lot looser than a formal programming language because a lot of English language descriptions of steps will be allowed.

This appendix is not meant for formal study. Rather, it should serve as a reference guide for students when they study the descriptions of algorithms given in the text and when they write pseudocode solutions to exercises.

Procedure Statements

The pseudocode for an algorithm begins with a **procedure** statement that gives the name of an algorithm, lists the input variables, and describes what kind of variable each input is. For instance, the statement

procedure *maximum*(L: list of integers)

is the first statement in the pseudocode description of the algorithm, which we have named *maximum*, that finds the maximum of a list L of integers.

Assignments and Other Types of Statements

An assignment statement is used to assign values to variables. In an assignment statement the left-hand side is the name of the variable and the right-hand side is an expression that involves constants, variables that have been assigned values, or functions defined by procedures. The right-hand side may contain any of the usual arithmetic operations. However, in the pseudocode in this book it may include any well-defined operation, even if this operation can be carried out only by using a large number of statements in an actual programming language.

The symbol $:=$ is used for assignments. Thus, an assignment statement has the form

variable := *expression*

For example, the statement

```
max := a
```

assigns the value of a to the variable max. A statement such as

```
x := largest integer in the list L
```

can also be used. This sets x equal to the largest integer in the list L. To translate this statement into an actual programming language would require more than one statement. Also, the instruction

```
interchange a and b
```

can be used to interchange a and b. We could also express this one statement with several assignment statements (see Exercise 2), but for simplicity, we will often prefer this abbreviated form of pseudocode.

Comments

In the pseudocode in this book, statements enclosed in curly braces are not executed. Such statements serve as comments or reminders that help explain how the procedure works. For instance, the statement

```
{x is the largest element in L}
```

can be used to remind the reader that at that point in the procedure the variable x equals the largest element in the list L.

Conditional Constructions

The simplest form of the conditional construction that we will use is

```
if condition then statement
```

or

```
if condition then
    block of statements
```

Here, the condition is checked, and if it is true, then the statement or block of statements given is carried out. In particular, the pseudocode

```
if condition then
    statement 1
    statement 2
    statement 3
    .
    .
    .
    statement n
```

tells us that the statements in the block are executed sequentially if the condition is true.

For example, in Algorithm 1 in Section 3.1, which finds the maximum of a set of integers, we use a conditional statement to check whether $max < a_i$ for each variable; if it is, we assign the value of a_i to max.

Often, we require the use of a more general type of construction. This is used when we wish to do one thing when the indicated condition is true, but another when it is false. We use the construction

```
if condition then statement 1
else statement 2
```

Note that either one or both of statement 1 and statement 2 can be replaced with a block of statements.

Sometimes, we require the use of an even more general form of a conditional. The general form of the conditional construction that we will use is

```
if condition 1 then statement 1
else if condition 2 then statement 2
else if condition 3 then statement 3
    .
    .
    .
else if condition n then statement n
else statement n + 1
```

When this construction is used, if condition 1 is true, then statement 1 is carried out, and the program exits this construction. In addition, if condition 1 is false, the program checks whether condition 2 is true; if it is, statement 2 is carried out, and so on. Thus, if none of the first $n - 1$ conditions hold, but condition n does, statement n is carried out. Finally, if none of condition 1, condition 2, condition 3, . . . , condition n is true, then statement $n + 1$ is executed. Note that any of the $n + 1$ statements can be replaced by a block of statements.

Loop Constructions

There are two types of loop construction in the pseudocode in this book. The first is the "for" construction, which has the form

for *variable* := *initial value* **to** *final value*
    statement

or

for *variable* := *initial value* **to** *final value*
    block of statements

where *initial value* and *final value* are integers. Here, at the start of the loop, *variable* is assigned *initial value* if *initial value* is less than or equal to *final value*, and the statements at the end of this construction are carried out with this value of *variable*. Then *variable* is increased by one, and the statement, or the statements in the block, are carried out with this new value of *variable*. This is repeated until *variable* reaches *final value*. After the instructions are carried out with *variable* equal to *final value*, the algorithm proceeds to the next statement. When *initial value* exceeds *final value*, none of the statements in the loop is executed.

We can use the "for" loop construction to find the sum of the positive integers from 1 to n with the following pseudocode.

$sum := 0$
for $i := 1$ **to** n
    $sum := sum + i$

Also, the more general "for" statement, of the form

for all elements with a certain property

is used in this text. This means that the statement or block of statements that follow are carried out successively for the elements with the given property.

The second type of loop construction that we will use is the "while" construction. This has the form

while condition
    statement

or

while condition
    block of statements

When this construction is used, the condition given is checked, and if it is true, the statements that follow are carried out, which may change the values of the variables that are part of the condition.

If the condition is still true after these instructions have been carried out, the instructions are carried out again. This is repeated until the condition becomes false. As an example, we can find the sum of the integers from 1 to n using the following block of pseudocode including a "while" construction.

$sum := 0$
while $n > 0$
    $sum := sum + n$
    $n := n - 1$

Note that any "for" construction can be turned into a "while" construction (see Exercise 3). However, it is often easier to understand the "for" construction. So, when it makes sense, we will use the "for" construction in preference to the corresponding "while" construction.

Loops within Loops

Loops or conditional statements are often used within other loops or conditional statements. In the pseudocode used in this book, we use successive levels of indentation to indicate nested loops, which are loops within loops, and which blocks of commands correspond to which loops.

Using Procedures in Other Procedures

We can use a procedure from within another procedure (or within itself in a recursive program) simply by writing the name of this procedure followed by the inputs to this procedure. For instance,

$max(L)$

will carry out the procedure *max* with the input list L. After all the steps of this procedure have been carried out, execution carries on with the next statement in the procedure.

Return Statements

We use a **return** statement to show where a procedure produces output. A return statement of the form

return x

produces the current value of x as output. The output x can involve the value of one or more functions, including the same function under evaluation, but at a smaller value. For instance, the statement

return $f(n-1)$

is used to call the algorithm with input of $n - 1$. This means that the algorithm is run again with input equal to $n - 1$.

Exercises

1. What is the difference between the following blocks of two assignment statements?

$a := b$
$b := c$

and

$b := c$
$a := b$

2. Give a procedure using assignment statements to interchange the values of the variables x and y. What is the minimum number of assignment statements needed to do this?

3. Show how a loop of the form

for $i :=$ *initial value* **to** *final value*
    statement

can be written using the "while" construction.

Suggested Reading

Among the resources available for learning more about the topics covered in this book are printed materials and relevant websites. Printed resources are described in this section of suggested readings. Readings are listed by chapter and keyed to particular topics of interest. Some general references also deserve special mention. A book you may find particularly useful is the *Handbook of Discrete and Combinatorial Mathematics* by Rosen [Ro00], a comprehensive reference book. Additional applications of discrete mathematics can be found in Michaels and Rosen [MiRo91], which is also available online on the companion website for this book. Deeper coverage of many topics in computer science, including those discussed in this book, can be found in Gruska [Gr97]. Biographical information about many of the mathematicians and computer scientists mentioned in this book can be found in Gillispie [Gi70] and on the MacTutor website at *http://www-history.mcs.st-and.ac.uk/*.

To find pertinent websites, consult the links found in the Web Resources Guide on the companion website for this book. Its address is *www.mhhe.com/rosen*.

CHAPTER 1

An entertaining way to study logic is to read Lewis Carroll's book [Ca78]. General references for logic include M. Huth and M. Ryan [HuRy04], Mendelson [Me09], Stoll [St74], and Suppes [Su87]. A comprehensive treatment of logic in discrete mathematics can be found in Gries and Schneider [GrSc93]. System specifications are discussed in Ince [In93]. Smullyan's knights and knaves puzzles were introduced in [Sm78]. He has written many fascinating books on logic puzzles including [Sm92] and [Sm98]. Prolog is discussed in depth in Nilsson and Maluszynski [NiMa95] and in Clocksin and Mellish [ClMe94]. The basics of proofs are covered in Cupillari [Cu05], Morash [Mo91], Solow [So09], Velleman [Ve06], and Wolf [Wo98]. The science and art of constructing proofs is discussed in a delightful way in three books by Pólya: [Po62], [Po71], and [Po90]. Problems involving tiling checkerboards using dominoes and polyominoes are discussed in Golomb [Go94] and Martin [Ma91].

CHAPTER 2

Lin and Lin [LiLi81] is an easily read text on sets and their applications. Axiomatic developments of set theory can be found in Halmos [Ha60], Monk [Mo69], and Stoll [St74]. Brualdi [Br09], and Reingold, Nievergelt, and Deo [ReNiDe77] contain introductions to multisets. Fuzzy sets and their application to expert systems and artificial intelligence are treated in Negoita [Ne85] and Zimmerman [Zi91]. Calculus books, such as Apostol [Ap67], Spivak [Sp94], and Thomas and Finney [ThFi96], contain discussions of functions. The best printed source of information about integer sequences is Sloan and Plouffe [SlPl95]. Books on proofs, such as [Ve06], often cover countability in some depth. Stanat and McAllister [StMc77] has a thorough section on countability. Chapter 17 of Aigner, Ziegler, and Hoffman [AiZiHo09] provides an excellent discussion of cardinality and the continuum hypothesis. Discussions of the mathematical foundations needed for computer science can be found in Arbib, Kfoury, and Moll [ArKfMo80], Bobrow and Arbib [BoAr74], Beckman [Be80], and Tremblay and Manohar [TrMa75]. Matrices and their operations are covered in all linear algebra books, such as Curtis [Cu84] and Strang [St09].

CHAPTER 3

The articles by Knuth [Kn77] and Wirth [Wi84] are accessible introductions to the subject of algorithms. Among the best introductions to algorithms are Cormen, Leierson, Rivest, and Stein [CoLeRiSt09] and Kleinberg and Tardos [KlTa05]. Extensive material on big-O estimates of functions can be found in Knuth [Kn97a]. General references for algorithms and their complexity include Aho, Hopcroft, and Ullman [AhHoUl74]; Baase and Van Gelder [BaGe99]; Cormen, Leierson, Rivest, and Stein [CoLeRiSt09]; Gonnet [Go84]; Goodman and Hedetniemi [GoHe77]; Harel [Ha87]; Horowitz and Sahni [HoSa82]; Kreher and Stinson [KrSt98]; the famous series of books by Knuth on the art of computer programming [Kn97a], [Kn97b], and [Kn98]; Kronsjö [Kr87]; Levitin [Le06]; Manber [Ma89]; Pohl and Shaw [PoSh81]; Purdom and Brown [PuBr85]; Rawlins [Ra92]; Sedgewick [Se03]; Wilf [Wi02]; and Wirth [Wi76]. Sorting and searching algorithms and their complexity are studied in detail in Knuth [Kn98].

CHAPTER 4

References for number theory include Hardy and Wright [HaWrWiHe08]; LeVeque [Le77]; Rosen [Ro10]; and Stark [St78]. More about the history of number theory can be found in Ore [Or88]. Algorithms for computer arithmetic are discussed in Knuth [Kn97b] and Pohl and Shaw [PoSh81]. More information about algorithms for finding primes and for factorization can be found in Crandall

and Pomerance [CrPo10]. Applications of number theory to cryptography are covered in Denning [De82]; Menezes, van Oorschot, and Vanstone [MeOoVa97]; Rosen [Ro10]; Seberry and Pieprzyk [SePi89]; Sinkov [Si66]; and Stinson [St05]. The RSA public-key system was described by Rivest, Shamir, and Adleman in [RiShAd78]; its discovery by Cocks is described in [Si99], which also provides an appealing account of the history of cryptography.

CHAPTER 5

An accessible introduction to mathematical induction can be found [Gu10] and in Sominskii [So61]. Books that contain thorough treatments of mathematical induction and recursive definitions include Liu [Li85]; Sahni [Sa85]; Stanat and McAllister [StMc77]; and Tremblay and Manohar [TrMa75]. Computational geometry is covered in [DeOr11] and [Or00]. The Ackermann function, introduced in 1928 by W. Ackermann, arises in the theory of recursive function (see Beckman [Be80] and McNaughton [Mc82], for instance) and in the analysis of the complexity of certain set theoretic algorithms (see Tarjan [Ta83]). Recursion is studied in Roberts [Ro86]; Rohl [Ro84]; and Wand [Wa80]. Discussions of program correctness and the logical machinery used to prove that programs are correct can be found in Alagic and Arbib [AlAr78]; Anderson [An79]; Backhouse [Ba86]; Sahni [Sa85]; and Stanat and McAllister [StMc77].

CHAPTER 6

General references for counting techniques and their applications include Allenby and Slomson [AlSl10]; Anderson [An89]; Berman and Fryer [BeFr72]; Bogart [Bo00]; Bona [Bo07]; Bose and Manvel [BoMa86]; Brualdi [Br09]; Cohen [Co78]; Grimaldi [Gr03]; Gross [Gr07]; Liu [Li68]; Pólya, Tarjan, and Woods [PoTaWo83]; Riordan [Ri58]; Roberts and Tesman [RoTe03]; Tucker [Tu06]; and Williamson [Wi85]. Vilenkin [Vi71] contains a selection of combinatorial problems and their solutions. A selection of more difficult combinatorial problems can be found in Lovász [Lo79]. Information about Internet protocol addresses and datagrams can be found in Comer [Co05]. Applications of the pigeonhole principle can be found in Brualdi [Br09]; Liu [Li85]; and Roberts and Tesman [RoTe03]. A wide selection of combinatorial identities can be found in Riordan [Ri68] and in Benjamin and Quinn [BeQu03]. Combinatorial algorithms, including algorithms for generating permutations and combinations, are described by Even [Ev73]; Lehmer [Le64]; and Reingold, Nievergelt, and Deo [ReNiDe77].

CHAPTER 7

Useful references for discrete probability theory include Feller [Fe68], Nabin [Na00], and Ross [Ro09a]. Ross [Ro02], which focuses on the application of probability theory to computer science, provides examples of average case complexity analysis and covers the probabilistic method. Aho and Ullman [AhUl95] includes a discussion of various aspects of probability theory important in computer science, including programming applications of probability. The probabilistic method is discussed in a chapter in Aigner, Ziegler, and Hoffman [AiZiHo09], a monograph devoted to clever, insightful, and brilliant proofs, that is, proofs that Paul Erdős described as coming from "*The Book*." Extensive coverage of the probabilistic method can be found in Alon and Spencer [AlSp00]. Bayes' theorem is covered in [PaPi01]. Additional material on spam filters can be found in [Zd05].

CHAPTER 8

Many different models using recurrence relations can be found in Roberts and Tesman [RoTe03] and Tucker [Tu06]. Exhaustive treatments of linear homogeneous recurrence relations with constant coefficients, and related inhomogeneous recurrence relations, can be found in Brualdi [Br09], Liu [Li68], and Mattson [Ma93]. Divide-and-conquer algorithms and their complexity are covered in Roberts and Tesman [RoTe03] and Stanat and McAllister [StMc77]. Descriptions of fast multiplication of integers and matrices can be found in Aho, Hopcroft, and Ullman [AhHoUl74] and Knuth [Kn97b]. An excellent introduction to generating functions can be found in Pólya, Tarjan, and Woods [PoTaWo83]. Generating functions are studied in detail in Brualdi [Br09]; Cohen [Co78]; Graham, Knuth, and Patashnik [GrKnPa94]; Grimaldi [Gr03]; and Roberts and Tesman [RoTe03]. Additional applications of the principle of inclusion–exclusion can be found in Liu [Li85] and [Li68]; Roberts and Tesman [RoTe03]; and Ryser [Ry63].

CHAPTER 9

General references for relations, including treatments of equivalence relations and partial orders, include Bobrow and Arbib [BoAr74]; Grimaldi [Gr03]; Sanhi [Sa85]; and Tremblay and Manohar [TrMa75]. Discussions of relational models for databases are given in Date [Da82] and Aho and Ullman [AhUl95]. The original papers by Roy and Warshall for finding transitive closures can be found in [Ro59] and [Wa62], respectively. Directed graphs are studied in Chartrand, Lesniak, and Zhang [ChLeZh05]; Gross and Yellen [GrYe05]; Robinson and Foulds [RoFo80]; Roberts and Tesman [RoTe03]; and Tucker [Tu06]. The application of lattices to information flow is treated in Denning [De82].

CHAPTER 10

General references for graph theory include Agnarsson and Greenlaw [AgGr06]; Aldous, Wilson, and Best [AlWiBe00]; Behzad and Chartrand [BeCh71]; Chartrand, Lesniak, and Zhang [ChLeZh05]; Chartrand and Zhang [ChZh04]; Bondy and Murty [BoMu10]; Chartrand and Oellermann [ChOe93]; Graver and Watkins [GrWa77]; Roberts and Tesman [RoTe03]; Tucker [Tu06]; West [We00]; Wilson [Wi85]; and Wilson and Watkins [WiWa90]. A wide variety of applications of graph theory can be found in Chartrand [Ch77]; Deo [De74]; Foulds [Fo92]; Roberts

and Tesman [RoTe03]; Roberts [Ro76]; Wilson and Beineke [WiBe79]; and McHugh [Mc90]. In depth treatments of the use of graph theory to study social networks, and other types of networks, appears in Easley and Kleinberg [EaKl10] and Newman [Ne10]. Applications involving large graphs, including the Web graph, are discussed in Hayes [Ha00a] and [Ha00b].

A comprehensive description of algorithms in graph theory can be found in Gibbons [Gi85] and in Kocay and Kreher [KoKr04]. Other references for algorithms in graph theory include Buckley and Harary [BuHa90]; Chartrand and Oellermann [ChOe93]; Chachra, Ghare, and Moore [ChGhMo79]; Even [Ev73] and [Ev79]; Hu [Hu82]; and Reingold, Nievergelt, and Deo [ReNiDe77]. A translation of Euler's original paper on the Königsberg bridge problem can be found in Euler [Eu53]. Dijkstra's algorithm is studied in Gibbons [Gi85]; Liu [Li85]; and Reingold, Nievergelt, and Deo [ReNiDe77]. Dijkstra's original paper can be found in [Di59]. A proof of Kuratowski's theorem can be found in Harary [Ha69] and Liu [Li68]. Crossing numbers and thicknesses of graphs are studied in Chartrand, Lesniak, and Zhang [ChLeZh05]. References for graph coloring and the four-color theorem are included in Barnette [Ba83] and Saaty and Kainen [SaKa86]. The original conquest of the four-color theorem is reported in Appel and Haken [ApHa76]. Applications of graph coloring are described by Roberts and Tesman [RoTe03]. The history of graph theory is covered in Biggs, Lloyd, and Wilson [BiLlWi86]. Interconnection networks for parallel processing are discussed in Akl [Ak89] and Siegel and Hsu [SiHs88].

CHAPTER 11

Trees are studied in Deo [De74], Grimaldi [Gr03], Knuth [Kn97a], Roberts and Tesman [RoTe03], and Tucker [Tu06]. The use of trees in computer science is described by Gotlieb and Gotlieb [GoGo78], Horowitz and Sahni [HoSa82], and Knuth [Kn97a, 98]. Roberts and Tesman [RoTe03] covers applications of trees to many different areas. Prefix codes and Huffman coding are covered in Hamming [Ha80]. Backtracking is an old technique; its use to solve maze puzzles can be found in the 1891 book by Lucas [Lu91]. An extensive discussion of how to solve problems using backtracking can be found in Reingold, Nievergelt, and Deo [ReNiDe77]. Gibbons [Gi85] and Reingold, Nievergelt, and Deo [ReNiDe77] contain discussions of algorithms for constructing spanning trees and minimal spanning trees. The background and history of algorithms for finding minimal spanning trees is covered in Graham and Hell [GrHe85]. Prim and Kruskal described their algorithms for finding minimal spanning trees in [Pr57] and [Kr56], respectively. Sollin's algorithm is an example of an algorithm well suited for parallel processing; although Sollin never published a description of it, his algorithm has been described by Even [Ev73] and Goodman and Hedetniemi [GoHe77].

CHAPTER 12

Boolean algebra is studied in Hohn [Ho66], Kohavi [Ko86], and Tremblay and Manohar [TrMa75]. Applications of Boolean algebra to logic circuits and switching circuits are described by Hayes [Ha93], Hohn [Ho66], Katz and Borriello [KaBo04], and Kohavi [Ko86]. The original papers dealing with the minimization of sum-of-products expansions using maps are Karnaugh [Ka53] and Veitch [Ve52]. The Quine-McCluskey method was introduced in McCluskey [Mc56] and Quine [Qu52] and [Qu55]. Threshold functions are covered in Kohavi [Ko86].

CHAPTER 13

General references for formal grammars, automata theory, and the theory of computation include Davis, Sigal, and Weyuker [DaSiWe94]; Denning, Dennis, and Qualitz [DeDeQu81]; Hopcroft, Motwani, and Ullman [HoMoUl06]; Hopkin and Moss [HoMo76]; Lewis and Papadimitriou [LePa97]; McNaughton [Mc82]; and Sipser [Si06]. Mealy machines and Moore machines were originally introduced in Mealy [Me55] and Moore [Mo56]. The original proof of Kleene's theorem can be found in [Kl56]. Powerful models of computation, including pushdown automata and Turing machines, are discussed in Brookshear [Br89], Hennie [He77], Hopcroft and Ullman [HoUl79], Hopkin and Moss [HoMo76], Martin [Ma03], Sipser [Si06], and Wood [Wo87]. Barwise and Etchemendy [BaEt93] is an excellent introduction to Turing machines. Interesting articles about the history and application of Turing machines and related machines can be found in Herken [He88]. Busy beaver machines were first introduced by Rado in [Ra62], and information about them can be found in Dewdney [De84] and [De93], the article by Brady in Herken [He88], and in Wood [Wo87].

APPENDIXES

A discussion of axioms for the real number and for the integers can be found in Morash [Mo91]. Detailed treatments of exponential and logarithmic functions can be found in calculus books such as Apostol [Ap67], Spivak [Sp94], and Thomas and Finney [ThFi96]. Pohl and Shaw [PoSh81] use a form of pseudocode that has the same features as those described in Appendix 3. Most textbooks on algorithms, such as Cormen, Leierson, Rivest, and Stein [CoLeRiSt09] and Kleinberg and Tardos [KlTa05], use versions of pseudocode similar to the pseudocode in this text.

REFERENCES

[**AgGr06**] G. Agnarsson and R. Greenlaw, *Graph Theory: Modeling, Applications, and Algorithms*, Prentice Hall, Englewood Cliffs, NJ, 2006.

[**AhHoUl74**] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison- Wesley, Reading, MA, 1974.

[**AhUl95**] Alfred V. Aho and Jeffrey D. Ullman, *Foundations of Computer Science, C Edition*, Computer Science Press, New York, 1995.

[**AiZiHo09**] Martin Aigner, Günter M. Ziegler, and Karl H. Hofmann, *Proofs from THE BOOK*, 4th ed., Springer, Berlin, 2009.

[**Ak89**] S. G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Englewood Cliffs, NJ, 1989.

[**AlAr78**] S. Alagic and M. A. Arbib, *The Design of Well-Structured and Correct Programs*, Springer-Verlag, New York, 1978.

[**AlWiBe00**] J. M. Aldous, R. J. Wilson, and S. Best, *Graphs and Applications: An Introductory Approach*, Springer, New York, 2000.

[**AlSl10**] R.B.J.T. Allenby and A. Slomson, *How to Count: An Introduction to Combinatorics*, 2d. ed., Chapman and Hall/CRC, Boca Raton, Florida, 2010.

[**AlSp00**] Noga Alon and Joel H. Spencer, *The Probabilistic Method*, 2d ed., Wiley, New York, 2000.

[**An89**] I. Anderson, *A First Course in Combinatorial Mathematics*, 2d. ed., Oxford University Press, New York, 1989.

[**An79**] R. B. Anderson, *Proving Programs Correct*, Wiley, New York, 1979.

[**Ap67**] T. M. Apostol, *Calculus*, Vol. I, 2d ed., Wiley, New York, 1967.

[**ApHa76**] K. Appel and W. Haken, "Every Planar Map Is 4-colorable," *Bulletin of the AMS*, 82 (1976), 711–712.

[**ArKfMo80**] M. A. Arbib, A. J. Kfoury, and R. N. Moll, *A Basis for Theoretical Computer Science*, Springer-Verlag, New York, 1980.

[**AvCh90**] B. Averbach and O. Chein, *Problem Solving Through Recreational Mathematics*, W.H. Freeman, San Francisco, 1980.

[**BaGe99**] S. Baase and A. Van Gelder, *Computer Algorithms: Introduction to Design and Analysis*, 3d ed., Addison-Wesley, Reading, MA, 1999.

[**Ba86**] R. C. Backhouse, *Program Construction and Verification*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[**Ba83**] D. Barnette, *Map Coloring, Polyhedra, and the Four-Color Problem*, Mathematical Association of America, Washington, DC, 1983.

[**BaEt93**] Jon Barwise and John Etchemendy, *Turing's World 3.0 for the Macintosh*, CSLI Publications, Stanford, CA, 1993.

[**Be80**] F. S. Beckman, *Mathematical Foundations of Programming*, Addison-Wesley, Reading, MA, 1980.

[**BeCh71**] M. Behzad and G. Chartrand, *Introduction to the Theory of Graphs*, Allyn & Bacon, Boston, 1971.

[**BeQu03**] A. Benjamin and J. J. Quine, *Proofs that Really Count*, Mathematical Association of America, Washington, DC, 2003.

[**Be86**] J. Bentley, *Programming Pearls*, Addison-Wesley, Reading, MA, 1986.

[**BeFr72**] G. Berman and K. D. Fryer, *Introduction to Combinatorics*, Academic Press, New York, 1972.

[**BiLlWi99**] N. L. Biggs, E. K. Lloyd, and R. J. Wilson, *Graph Theory 1736–1936*, Oxford University Press, Oxford, England, 1999.

[**BoAr74**] L. S. Bobrow and M. A. Arbib, *Discrete Mathematics*, Saunders, Philadelphia, 1974.

[**Bo00**] K. P. Bogart, *Introductory Combinatorics*, 3d ed. Academic Press, San Diego, 2000.

[**Bo07**] M. Bona, *Enumerative Combinatorics*, McGraw-Hill, New York, 2007.

[**BoMu10**] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, Springer, New York, 2010.

[**Bo04**] P. Bork, L. J. Jensen, C. von Mering, A. K. Ramani, I. Lee, and E. M. Marcotte, "Protein interaction networks from yeast to human," *Current Opinion in Structural Biology*, 14 (2004), 292–299.

[**BoMa86**] R. C. Bose and B. Manvel, *Introduction to Combinatorial Theory*, Wiley, New York, 1986.

[**Bo00**] A. Brodera, R. Kumar, F. Maghoula, P. Raghavan, S. Rajagopalan, R. Statac, A. Tomkins, and Janet Wiener, "Graph structure in the Web," *Computer Networks*, 33 (2000), 309–320.

[**Br89**] J. G. Brookshear, *Theory of Computation*, Benjamin Cummings, Redwood City, CA, 1989.

[**Br09**] R. A. Brualdi, *Introductory Combinatorics*, 5th ed., Prentice-Hall, Englewood Cliffs, NJ, 2009.

[**BuHa90**] F. Buckley and F. Harary, *Distance in Graphs*, Addison-Wesley, Redwood City, CA, 1990.

[**Ca79**] L. Carmony, "Odd Pie Fights," *Mathematics Teacher* 72 (January, 1979), 61–64.

[**Ca78**] L. Carroll, *Symbolic Logic*, Crown, New York, 1978.

[**ChGhMo79**] V. Chachra, P. M. Ghare, and J. M. Moore, *Applications of Graph Theory Algorithms*, North-Holland, New York, 1979.

[**Ch77**] G. Chartrand, *Graphs as Mathematical Models*, Prindle, Weber & Schmidt, Boston, 1977.

[**ChLeZh10**] G. Chartrand, L. Lesniak, and P. Zhang, *Graphs and Digraphs*, 5th ed., Chapman and Hall/CRC, Boca Raton, 2010.

[**ChOe93**] G. Chartrand and O. R. Oellermann, *Applied Algorithmic Graph Theory*, McGraw-Hill, New York, 1993.

[**ChZh04**] G. Chartrand and P. Zhang, *Introduction to Graph Theory*, McGraw-Hill, New York, 2004.

[**ClMe94**] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, 4th ed., Springer-Verlag, New York, 1994.

[**Co78**] D. I. A. Cohen, *Basic Techniques of Combinatorial Theory*, Wiley, New York, 1978.

[**Co05**] D. Comer, *Internetworking with TCP/IP, Principles, Protocols, and Architecture*, Vol. 1, 5th ed., Prentice-Hall, Englewood Cliffs, NJ, 2005.

[**CoLeRiSt09**] T. H. Cormen, C. E. Leierson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, Cambridge, MA, 2009.

[**CrPo10**] Richard Crandall and Carl Pomerance, 2d ed., *Prime Numbers: A Computational Perspective*, Springer-Verlag, New York, 2010.

[**Cu05**] Antonella Cupillari, *The Nuts and Bolts of Proofs*, 3d ed., Academic Press, San Diego, 2005.

[**Cu84**] C. W. Curtis, *Linear Algebra*, Springer-Verlag, New York, 1984.

[**Da82**] C. J. Date, *An Introduction to Database Systems*, 3d ed., Addison-Wesley, Reading, MA, 1982.

[**DaSiWe94**] M. Davis, R. Sigal, and E. J. Weyuker, *Computability, Complexity, and Languages*, 2d ed., Academic Press, San Diego, 1994.

[**Da10**] T. Davis, "The Mathematics of Sudoku," November, 2010, available at *http://geometer.org/mathcircles/sudoku.pdf*

[**De82**] D. E. R. Denning, *Cryptography and Data Security*, Addison-Wesley, Reading, MA, 1982.

[**DeDeQu81**] P. J. Denning, J. B. Dennis, and J. E. Qualitz, *Machines, Languages, and Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[**De74**] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[**DeOr11**] S. L. Devadoss and J. O'Rourke, *Discrete and Computational Geometry*, Princeton University Press, Princeton, NJ, 2011.

[**De02**] K. Devlin, *The Millennium Problems: The Seven Greatest Unsolved Mathematical Puzzles of Our Time*, Basic Book, New York, 2002.

[**De84**] A. K. Dewdney, "Computer Recreations," *Scientific American*, 251, no. 2 (August 1984), 19–23; 252, no. 3 (March 1985), 14–23; 251, no. 4 (April 1985), 20–30.

[**De93**] A. K. Dewdney, *The New Turing Omnibus: Sixty-Six Excursions in Computer Science*, W. H. Freeman, New York, 1993.

[**Di59**] E. Dijkstra, "Two Problems in Connexion with Graphs," *Numerische Mathematik*, 1 (1959), 269–271.

[**EaKl10**] D. Easley and J. Kleinberg, *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*, Cambridge University Press, New York, 2010.

[**Eu53**] L. Euler, "The Koenigsberg Bridges," *Scientific American*, 189, no. 1 (July 1953), 66–70.

[**Ev73**] S. Even, *Algorithmic Combinatorics*, Macmillan, New York, 1973.

[**Ev79**] S. Even, *Graph Algorithms*, Computer Science Press, Rockville, MD, 1979.

[**Fe68**] W. Feller, *An Introduction to Probability Theory and Its Applications*, Vol. 1, 3d ed., Wiley, New York, 1968.

[**Fo92**] L. R. Foulds, *Graph Theory Applications*, Springer-Verlag, New York, 1992.

[**GaJo79**] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to NP-Completeness*, Freeman, New York, 1979.

[**Gi85**] A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, Cambridge, England, 1985.

[**Gi70**] C. C. Gillispie, ed., *Dictionary of Scientific Biography*, Scribner's, New York, 1970.

[**Go94**] S. W. Golomb, *Polyominoes*, Princeton University Press, Princeton, NJ, 1994.

[**Go84**] G. H. Gonnet, *Handbook of Algorithms and Data Structures*, Addison-Wesley, London, 1984.

[**GoHe77**] S. E. Goodman and S. T. Hedetniemi, *Introduction to the Design and Analysis of Algorithms*, McGraw-Hill, New York, 1977.

[**GoGo78**] C. C. Gotlieb and L. R. Gotlieb, *Data Types and Structures*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[**GrHe85**] R. L. Graham and P. Hell, "On the History of the Minimum Spanning Tree Problem," *Annals of the History of Computing*, 7 (1985), 43–57.

[**GrKnPa94**] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*, 2d ed., Addison-Wesley, Reading, MA, 1994.

[**GrRoSp90**] Ronald L. Graham, Bruce L. Rothschild, and Joel H. Spencer, *Ramsey Theory*, 2d ed., Wiley, New York, 1990.

[**GrWa77**] J. E. Graver and M. E. Watkins, *Combinatorics with Emphasis on the Theory of Graphs*, Springer-Verlag, New York, 1977.

[**GrSc93**] D. Gries and F. B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, New York, 1993.

[**Gr03**] R. P. Grimaldi, *Discrete and Combinatorial Mathematics*, 5th ed., Addison-Wesley, Reading, MA, 2003.

[**Gr07**] J. L. Gross, *Combinatorial Methods with Computer Applications*, Chapman and Hall/CRC, Boca Raton, FL, 2007.

[**GrYe05**] J. L. Gross and J. Yellen, *Graph Theory and Its Applications*, 2d ed., CRC Press, Boca Raton, FL, 2005.

[**GrYe03**] J. L. Gross and J. Yellen, *Handbook of Graph Theory*, CRC Press, Boca Raton, FL, 2003.

[**Gr90**] Jerrold W. Grossman, *Discrete Mathematics: An Introduction to Concepts, Methods, and Applications*, Macmillan, New York, 1990.

[**Gr97**] J. Gruska, *Foundations of Computing*, International Thomsen Computer Press, London, 1997.

[**Gu10**] D. A. Gunderson, *Handbook of Mathematical Induction*, Chapman and Hall/CRC, Boca Raton, Florida, 2010.

[**Ha60**] P. R. Halmos, *Naive Set Theory*, D. Van Nostrand, New York, 1960.

[**Ha80**] R. W. Hamming, *Coding and Information Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[**Ha69**] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[**HaWrWiHe08**] G. H. Hardy, E. M. Wright, A. Wiles, and R. Heath-Brown, *An Introduction to the Theory of Numbers*, 6th ed., Oxford University Press, USA, New York, 2008.

[**Ha87**] D. Harel, *Algorithmics, The Spirit of Computing*, Addison-Wesley, Reading, MA, 1987.

[**Ha00a**] Brian Hayes, "Graph-Theory in Practice, Part I," *American Scientist*, 88, no. 1 (2000), 9–13.

[**Ha00b**] Brian Hayes, "Graph-Theory in Practice, Part II," *American Scientist*, 88, no. 2 (2000), 104–109.

[**Ha93**] John P. Hayes, *Introduction to Digital Logic Design*, Addison-Wesley, Reading, MA, 1993.

[**He77**] F. Hennie, *Introduction to Computability*, Addison-Wesley, Reading, MA, 1977.

[**He88**] R. Herken, *The Universal Turing Machine, A Half-Century Survey*, Oxford University Press, New York, 1988.

[**Ho76**] C. Ho, "Decomposition of a Polygon into Triangles," *Mathematical Gazette*, 60 (1976), 132–134.

[**Ho99**] D. Hofstadter, Gödel, Escher, *Bach: An Internal Golden Braid*, Basic Books, New York, 1999.

[**Ho66**] F. E. Hohn, *Applied Boolean Algebra*, 2d ed., Macmillan, New York, 1966.

[**HoMoUl06**] J. E. Hopcroft, R Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3d ed., Addison-Wesley, Boston, MA, 2006.

[**HoMo76**] D. Hopkin and B. Moss, *Automata*, Elsevier, North-Holland, New York, 1976.

[**HoSa82**] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD, 1982.

[**Hu82**] T.C. Hu, *Combinatorial Algorithms*, Addison-Wesley, Reading, MA, 1982.

[**Hu07**] W. Huber, V. J. Carey, L. Long, S. Falcon, and R. Gentleman, "Graphs in molecular biology," *BMC Bioinformatics*, 8 (Supplement 6) (2007).

[**HuRy04**] M. Huth and M. Ryan, *Logic in Computer Science*, 2d ed., Cambridge University Press, Cambridge, England, 2004.

[**In93**] D. C. Ince, *An Introduction to Discrete Mathematics, Formal System Specification, and Z*, 2d ed., Oxford, New York, 1993.

[**KaMo64**] D. Kalish and R. Montague, *Logic: Techniques of Formal Reasoning*, Harcourt, Brace, Jovanovich, New York, 1964.

[**Ka53**] M. Karnaugh, "The Map Method for Synthesis of Combinatorial Logic Circuits," *Transactions of the AIEE*, part I, 72 (1953), 593–599.

[**KaBo04**] R. H. Katz and G. Borriello, *Contemporary Logic Design*, Prentice-Hall, Englewood Cliffs, NJ, 2004.

[**Kl56**] S. C. Kleene, "Representation of Events by Nerve Nets," *in Automata Studies*, 3–42, Princeton University Press, Princeton, NJ, 1956.

[**KlTa05**] J. Kleinberg and E. Tardos, *Algorithm Design*, Addison-Wesley, Boston, 2005

[**Kn77**] D. E. Knuth, "Algorithms," *Scientific American*, 236, no. 4 (April 1977), 63–80.

[**Kn97a**] D. E. Knuth, *The Art of Computer Programming, Vol. I: Fundamental Algorithms*, 3d ed., Addison-Wesley, Reading, MA, 1997.

[**Kn97b**] D. E. Knuth, *The Art of Computer Programming, Vol. II: Seminumerical Algorithms*, 3d ed., Addison-Wesley, Reading, MA, 1997.

[**Kn98**] D. E. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching*, 2d ed., Addison-Wesley, Reading, MA, 1998.

[**KoKr04**] W. Kocay and D. L. Kreher, *Graph Algorithms and Optimization*, Chapman and Hall/CRC, Boca Raton, Florida, 2004.

[**Ko86**] Z. Kohavi, *Switching and Finite Automata Theory*, 2d ed., McGraw-Hill, New York, 1986.

[**KrSt98**] Donald H. Kreher and Douglas R. Stinson, *Combinatorial Algorithms: Generation, Enumeration, and Search*, CRC Press, Boca Raton, FL, 1998.

[**Kr87**] L. Kronsjö, *Algorithms: Their Complexity and Efficiency*, 2d ed., Wiley, New York, 1987.

[**Kr56**] J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proceedings of the AMS*, 1 (1956), 48–50.

[**La10**] J. C. Lagarias (ed.), *The Ultimate Challenge: The* $3x + 1$ *Problem*, The American Mathematical Society, Providence, 2010.

[**Le06**] A. V. Levitin, *Introduction to the Design and Analysis of Algorithms*, 2d, ed., Addison-Wesley, Boston, MA, 2006.

[**Le64**] D. H. Lehmer, "The Machine Tools of Combinatorics," in E. F. Beckenbach (ed.), *Applied Combinatorial Mathematics*, Wiley, New York, 1964.

[**Le77**] W. J. LeVeque, *Fundamentals of Number Theory*, Addison-Wesley, Reading, MA, 1977.

[**LePa97**] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*, 2d ed., Prentice-Hall, Englewood Cliffs, NJ, 1997.

[**LiLi81**] Y. Lin and S. Y. T. Lin, *Set Theory with Applications*, 2d ed., Mariner, Tampa, FL, 1981.

[**Li68**] C. L. Liu, *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York, 1968.

[**Li85**] C. L. Liu, *Elements of Discrete Mathematics*, 2d ed., McGraw-Hill, New York, 1985.

[**Lo79**] L. Lovász, *Combinatorial Problems and Exercises*, North-Holland, Amsterdam, 1979.

[**Lu91**] E. Lucas, *Récréations Mathématiques*, Gauthier-Villars, Paris, 1891.

[**Ma89**] U. Manber, *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, Reading, MA, 1989.

[**Ma91**] G. E. Martin, *Polyominoes: A Guide to Puzzles and Problems in Tiling*, Mathematical Association of America, Washington, DC, 1991.

[**Ma03**] J. C. Martin, *Introduction to Languages and the Theory of Computation*, 3d ed., McGraw-Hill, New York, 2003.

[**Ma93**] H. F. Mattson, Jr., *Discrete Mathematics with Applications*, Wiley, New York, 1993.

[**Mc56**] E. J. McCluskey, Jr., "Minimization of Boolean Functions," *Bell System Technical Journal*, 35 (1956), 1417–1444.

[**Mc90**] J. A. McHugh, *Algorithmic Graph Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[**Mc82**] R. McNaughton, *Elementary Computability, Formal Languages, and Automata*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[**Me55**] G. H. Mealy, "A Method for Synthesizing Sequential Circuits," *Bell System Technical Journal*, 34 (1955), 1045–1079.

[**Me09**] E. Mendelson, *Introduction to Mathematical Logic*, 5th ed., Chapman and Hall/CRC Press, Boca Raton, 2009.

[**MeOoVa97**] A. J. Menezes, P. C. van Oorschoot, S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, FL, 1997.

[**MiRo91**] J. G. Michaels and K. H. Rosen, *Applications of Discrete Mathematics*, McGraw-Hill, New York, 1991.

[**Mo69**] J. R. Monk, *Introduction to Set Theory*, McGraw-Hill, New York, 1969.

[**Mo91**] R. P. Morash, *Bridge to Abstract Mathematics*, McGraw-Hill, New York, 1991.

[**Mo56**] E. F. Moore, "Gedanken-Experiments on Sequential Machines," *in Automata Studies*, 129–153, Princeton University Press, Princeton, NJ, 1956.

[**Na00**] Paul J. Nabin, *Duelling Idiots and Other Probability Puzzlers*, Princeton University Press, Princeton, NJ, 2000.

[**Ne85**] C. V. Negoita, *Expert Systems and Fuzzy Systems*, Benjamin Cummings, Menlo Park, CA, 1985.

[**Ne10**] M. Newman, *Networks: An Introduction*, Oxford University Press, New York, 2010.

[**NiMa95**] Ulf Nilsson and Jan Maluszynski, *Logic, Programming, and Prolog*, 2d ed., Wiley, Chichester, England, 1995.

[**Or00**] J. O'Rourke, *Computational Geometry in C*, Cambridge University Press, New York, 2000.

[**Or63**] O. Ore, *Graphs and Their Uses*, Mathematical Association of America, Washington, DC, 1963.

[**0r88**] O. Ore, *Number Theory and its History*, Dover, New York, 1988.

[**PaPi01**] A. Papoulis and S. U. Pillai, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, New York, 2001.

[**Pe87**] A. Pelc, "Solution of Ulam's Problem on Searching with a Lie," *Journal of Combinatorial Theory*, Series A, 44 (1987), 129–140.

[**Pe09**] M. S. Petrović, *Famous Puzzles of Great Mathematicians*, American Mathematical Society, Providence, 2009.

[**PoSh81**] I. Pohl and A. Shaw, *The Nature of Computation: An Introduction to Computer Science*, Computer Science Press, Rockville, MD, 1981.

[**Po62**] George Pólya, *Mathematical Discovery*, Vols. 1 and 2, Wiley, New York, 1962.

[**Po71**] George Pólya, *How to Solve It*, Princeton University Press, Princeton, NJ, 1971.

[**Po90**] George Pólya, *Mathematics and Plausible Reasoning*, Princeton University Press, Princeton, NJ, 1990.

[**PoTaWo83**] G. Pólya, R. E. Tarjan, and D. R. Woods, *Notes on Introductory Combinatorics*, Birkhäuser, Boston, 1983.

[**Pr57**] R. C. Prim, "Shortest Connection Networks and Some Generalizations," *Bell System Technical Journal*, 36 (1957), 1389–1401.

[**PuBr85**] P. W. Purdom, Jr. and C. A. Brown, *The Analysis of Algorithms*, Holt, Rinehart & Winston, New York, 1985.

[**Qu52**] W. V. Quine, "The Problem of Simplifying Truth Functions," *American Mathematical Monthly*, 59 (1952), 521–531.

[**Qu55**] W. V. Quine, "A Way to Simplify Truth Functions," *American Mathematical Monthly*, 62 (1955), 627–631.

[**Ra62**] T. Rado, "On Non-Computable Functions," *Bell System Technical Journal* (May 1962), 877–884.

[**Ra92**] Gregory J. E. Rawlins, *Compared to What? An Introduction to the Analysis of Algorithms*, Computer Science Press, New York, 1992.

[**ReNiDe77**] E. M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[**Ri58**] J. Riordan, *An Introduction to Combinatorial Analysis*, Wiley, New York, 1958.

[**Ri68**] J. Riordan, *Combinatorial Identities*, Wiley, New York, 1968.

[**RiShAd78**] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the Association for Computing Machinery*, 31, no. 2 (1978), 120–128.

[**Ro86**] E. S. Roberts, *Thinking Recursively*, Wiley, New York, 1986.

[**Ro76**] F. S. Roberts, *Discrete Mathematics Models*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[**RoTe03**] F. S. Roberts and B. Tesman, *Applied Combinatorics*, 2d ed., Prentice-Hall, Englewood Cliffs, NJ, 2003.

[**RoFo80**] D. F. Robinson and L. R. Foulds, *Digraphs: Theory and Techniques*, Gordon and Breach, New York, 1980.

[**Ro84**] J. S. Rohl, *Recursion via Pascal*, Cambridge University Press, Cambridge, England, 1984.

[**Ro10**] K. H. Rosen, *Elementary Number Theory and Its Applications*, 6th ed., Pearson, Boston, 2010.

[**Ro00**] K. H. Rosen, *Handbook of Discrete and Combinatorial Mathematics*, CRC Press, Boca Raton, FL, 2000.

[**Ro09**] Jason Rosenhouse, *The Monty Hall Problem*, Oxford University Press, New York, 2009.

[**Ro09a**] Sheldon M. Ross, *A First Course in Probability Theory*, 7th ed., Prentice-Hall, Englewood Cliffs, NJ, 2009.

[**Ro02**] Sheldon M. Ross, *Probability Models for Computer Science*, Harcourt/Academic Press, San Diego, 2002.

[**Ro59**] B. Roy, "Transitivité et Connexité," *C.R. Acad. Sci. Paris*, 249 (1959), 216.

[**Ry63**] H. Ryser, *Combinatorial Mathematics*, Mathematical Association of America, Washington, DC, 1963.

[**SaKa86**] T. L. Saaty and P. C. Kainen, *The Four-Color Problem: Assaults and Conquest*, Dover, New York, 1986.

[**Sa85**] S. Sahni, *Concepts in Discrete Mathematics*, Camelot, Minneapolis, 1985.

[**Sa00**] K. Sayood, *Introduction to Data Compression*, 2d ed., Academic Press, San Diego, 2000.

[**SePi89**] J. Seberry and J. Pieprzyk, *Cryptography: An Introduction to Computer Security*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[**Se03**] R. Sedgewick, *Algorithms in Java*, 3d ed., Addison-Wesley, Reading, MA, 2003.

[**SiHs88**] H. J. Siegel and W. T. Hsu, "Interconnection Networks," *in Computer Architectures*, V. M. Milutinovic (ed.), North-Holland, New York, 1988, pp. 225–264.

[**Si99**] S. Singh, *The Code Book*, Doubleday, New York, 1999.

[**Si66**] A. Sinkov, *Elementary Cryptanalysis*, Mathematical Association of America, Washington, DC, 1966.

[**Si06**] M. Sipser, *Introduction to the Theory of Computation*, Course Technology, Boston, 2006

[**SlPl95**] N. J. A. Sloane and S. Plouffe, *The Encyclopedia of Integer Sequences*, Academic Press, New York, 1995.

[**Sm78**] Raymond Smullyan, *What Is the Name of This Book?: The Riddle of Dracula and Other Logical Puzzles*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[**Sm92**] Raymond Smullyan, *Lady or the Tiger? And Other Logic Puzzles Including a Mathematical Novel That Features Godel's Great Discovery*, Times Book, New York, 1992.

[**Sm98**] Raymond Smullyan, *The Riddle of Scheherazade: And Other Amazing Puzzles, Ancient & Modern*, Harvest Book, Fort Washington, PA, 1998.

[**So09**] Daniel Solow, *How to Read and Do Proofs: An Introduction to Mathematical Thought Processes*, 5th ed., Wiley, New York, 2009.

[**So61**] I. S. Sominskii, *Method of Mathematical Induction*, Blaisdell, New York, 1961.

[**Sp94**] M. Spivak, *Calculus*, 3d ed., Publish or Perish, Wilmington, DE, 1994.

[**StMc77**] D. Stanat and D. F. McAllister, *Discrete Mathematics in Computer Science*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[**St78**] H. M. Stark, *An Introduction to Number Theory*, MIT Press, Cambridge, MA, 1978.

[**St05**] Douglas R. Stinson, *Cryptography, Theory and Practice*, 3d ed., Chapman and Hall/CRC, Boca Raton, FL, 2005.

[**St94**] P. K. Stockmeyer, "Variations on the Four-Post Tower of Hanoi Puzzle," *Congressus Numerantrum* 102 (1994), 3–12.

[**St74**] R. R. Stoll, *Sets, Logic, and Axiomatic Theories*, 2d ed., W. H. Freeman, San Francisco, 1974.

[**St09**] G. W. Strang, *Linear Algebra and Its Applications*, 4th ed., Wellesley Cambridge Press, Wellesley, MA, 2009

[**Su87**] P. Suppes, *Introduction to Logic*, D. Van Nostrand, Princeton, NJ, 1987.

[**Ta83**] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[**ThFi96**] G. B. Thomas and R. L. Finney, *Calculus and Analytic Geometry*, 9th ed., Addison-Wesley, Reading, MA, 1996.

[**TrMa75**] J. P. Tremblay and R. P. Manohar, *Discrete Mathematical Structures with Applications to Computer Science*, McGraw-Hill, New York, 1975.

[**Tu06**] Alan Tucker, *Applied Combinatorics*, 5th ed., Wiley, New York, 2006.

[**Ve52**] E. W. Veitch, "A Chart Method for Simplifying Truth Functions," *Proceedings of the ACM* (1952), 127–133.

[**Ve06**] David Velleman, *How to Prove It: A Structured Approach*, Cambridge University Press, New York, 2006.

[**Vi71**] N. Y. Vilenkin, *Combinatorics*, Academic Press, New York, 1971.

[**Wa80**] M. Wand, *Induction, Recursion, and Programming*, North-Holland, New York, 1980.

[**Wa62**] S. Warshall, "A Theorem on Boolean Matrices," *Journal of the ACM*, 9 (1962), 11–12.

[**We00**] D. B. West, *Introduction to Graph Theory*, 2d ed., Prentice Hall, Englewood Cliffs, NJ, 2000.

[**Wi02**] Herbert S. Wilf, *Algorithms and Complexity*, 2d ed., A. K. Peters, Natick, MA, 2002.

[**Wi85**] S. G. Williamson, *Combinatorics for Computer Science*, Computer Science Press, Rockville, MD, 1985.

[**Wi85a**] R. J. Wilson, *Introduction to Graph Theory*, 3d ed., Longman, Essex, England, 1985.

[**WiBe79**] R. J. Wilson and L. W. Beineke, *Applications of Graph Theory*, Academic Press, London, 1979.

[**WiWa90**] R. J. Wilson and J. J. Watkins, *Graphs, An Introductory Approach*, Wiley, New York, 1990.

[**Wi76**] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[**Wi84**] N. Wirth, "Data Structures and Algorithms," *Scientific American*, 251 (September 1984), 60–69.

[**Wo98**] Robert S. Wolf, *Proof, Logic, and Conjecture: The Mathematician's Toolbox*, W. H. Freeman, New York, 1998.

[**Wo87**] D. Wood, *Theory of Computation*, Harper & Row, New York, 1987.

[**Zd05**] J. Zdziarski, *Ending Spam: Bayesian Content Filtering and the Art of Statistical Language Classification*, No Starch Press, San Francisco, 2005.

[**Zi91**] H. J. Zimmermann, *Fuzzy Set Theory and Its Applications*, 2d ed., Kluwer, Boston, 1991.

Answers to Odd-Numbered Exercises

CHAPTER 1

Section 1.1

1. **a)** Yes, T **b)** Yes, F **c)** Yes, T **d)** Yes, F **e)** No **f)** No **3.** **a)** Mei does not have an MP3 player. **b)** There is pollution in New Jersey. **c)** $2 + 1 \neq 3$. **d)** The summer in Maine is not hot or it is not sunny. **5.** **a)** Steve does not have more than 100 GB free disk space on his laptop **b)** Zach does not block e-mails from Jennifer, or he does not block texts from Jennifer **c)** $7 \cdot 11 \cdot 13 \neq 999$ **d)** Diane did not ride her bike 100 miles on Sunday **7.** **a)** F **b)** T **c)** T **d)** T **e)** T **9.** **a)** Sharks have not been spotted near the shore. **b)** Swimming at the New Jersey shore is allowed, and sharks have been spotted near the shore. **c)** Swimming at the New Jersey shore is not allowed, or sharks have been spotted near the shore. **d)** If swimming at the New Jersey shore is allowed, then sharks have not been spotted near the shore. **e)** If sharks have not been spotted near the shore, then swimming at the New Jersey shore is allowed. **f)** If swimming at the New Jersey shore is not allowed, then sharks have not been spotted near the shore. **g)** Swimming at the New Jersey shore is allowed if and only if sharks have not been spotted near the shore. **h)** Swimming at the New Jersey shore is not allowed, and either swimming at the New Jersey shore is allowed or sharks have not been spotted near the shore. (Note that we were able to incorporate the parentheses by using the word "either" in the second half of the sentence.) **11.** **a)** $p \wedge q$ **b)** $p \wedge \neg q$ **c)** $\neg p \wedge \neg q$ **d)** $p \vee q$ **e)** $p \rightarrow q$ **f)** $(p \vee q) \wedge (p \rightarrow \neg q)$ **g)** $q \leftrightarrow p$ **13.** **a)** $\neg p$ **b)** $p \wedge \neg q$ **c)** $p \rightarrow q$ **d)** $\neg p \rightarrow \neg q$ **e)** $p \rightarrow q$ **f)** $q \wedge \neg p$ **g)** $q \rightarrow p$ **15.** **a)** $r \wedge \neg p$ **b)** $\neg p \wedge q \wedge r$ **c)** $r \rightarrow (q \leftrightarrow \neg p)$ **d)** $\neg q \wedge \neg p \wedge r$ **e)** $(q \rightarrow (\neg r \wedge \neg p)) \wedge \neg((\neg r \wedge \neg p) \rightarrow q)$ **f)** $(p \wedge r) \rightarrow \neg q$ **17.** **a)** False **b)** True **c)** True **d)** True **19.** **a)** Exclusive or: You get only one beverage. **b)** Inclusive or: Long passwords can have any combination of symbols. **c)** Inclusive or: A student with both courses is even more qualified. **d)** Either interpretation possible; a traveler might wish to pay with a mixture of the two currencies, or the store may not allow that. **21.** **a)** Inclusive or: It is allowable to take discrete mathematics if you have had calculus or computer science, or both. Exclusive or: It is allowable to take discrete mathematics if you have had calculus or computer science, but not if you have had both. Most likely the inclusive or is intended. **b)** Inclusive or: You can take the rebate, or you can get a low-interest loan, or you can get both the rebate and a low-interest loan. Exclusive or: You can take the rebate, or you can get a low-interest loan, but you cannot get both the rebate and a low-interest loan. Most likely the exclusive or is intended. **c)** Inclusive or: You can order two items from column A and none from column B, or three items from column B and none from column A, or five items including two from column A and three from column B. Exclusive or: You can order two items from column A or three items from column B, but not both. Almost certainly the exclusive or is intended. **d)** Inclusive or: More than 2 feet of snow or windchill below -100, or both, will close school. Exclusive or: More than 2 feet of snow or windchill below -100, but not both, will close school. Certainly the inclusive or is intended. **23.** **a)** If the wind blows from the northeast, then it snows. **b)** If it stays warm for a week, then the apple trees will bloom. **c)** If the Pistons win the championship, then they beat the Lakers. **d)** If you get to the top of Long's Peak, then you must have walked 8 miles. **e)** If you are world-famous, then you will get tenure as a professor. **f)** If you drive more than 400 miles, then you will need to buy gasoline. **g)** If your guarantee is good, then you must have bought your CD player less than 90 days ago. **h)** If the water is not too cold, then Jan will go swimming. **25.** **a)** You buy an ice cream cone if and only if it is hot outside. **b)** You win the contest if and only if you hold the only winning ticket. **c)** You get promoted if and only if you have connections. **d)** Your mind will decay if and only if you watch television. **e)** The train runs late if and only if it is a day I take the train. **27.** **a)** Converse: "I will ski tomorrow only if it snows today." Contrapositive: "If I do not ski tomorrow, then it will not have snowed today." Inverse: "If it does not snow today, then I will not ski tomorrow." **b)** Converse: "If I come to class, then there will be a quiz." Contrapositive: "If I do not come to class, then there will not be a quiz." Inverse: "If there is not going to be a quiz, then I don't come to class." **c)** Converse: "A positive integer is a prime if it has no divisors other than 1 and itself." Contrapositive: "If a positive integer has a divisor other than 1 and itself, then it is not prime." Inverse: "If a positive integer is not prime, then it has a divisor other than 1 and itself." **29.** **a)** 2 **b)** 16 **c)** 64 **d)** 16

31. **a)**

p	$\neg p$	$p \wedge \neg p$
T	F	F
F	T	F

b)

p	$\neg p$	$p \vee \neg p$
T	F	T
F	T	T

c)

p	q	$\neg q$	$p \vee \neg q$	$(p \vee \neg q) \rightarrow q$
T	T	F	T	T
T	F	T	T	F
F	T	F	F	T
F	F	T	T	F

d)

p	q	$p \vee q$	$p \wedge q$	$(p \vee q) \rightarrow (p \wedge q)$
T	T	T	T	T
T	F	T	F	F
F	T	T	F	F
F	F	F	F	T