

3

So far in our development of the theory of computation, we have presented several models of computing devices. Finite automata are good models for devices that have a small amount of memory. Pushdown automata are good models for devices that have an unlimited memory that is usable only in the last in, first out manner of a stack. We have shown that some very simple tasks are beyond the capabilities of these models. Hence they are too restricted to serve as models of general purpose computers.

3 1

We turn now to a much more powerful model, first proposed by Alan Turing in 1936, called the ***Turing machine***. Similar to a finite automaton but with an unlimited and unrestricted memory, a Turing machine is a much more accurate model of a general purpose computer. A Turing machine can do everything that a real computer can do. Nonetheless, even a Turing machine cannot solve certain problems. In a very real sense, these problems are beyond the theoretical limits of computation.

The Turing machine model uses an infinite tape as its unlimited memory. It has a tape head that can read and write symbols and move around on the tape.

Initially the tape contains only the input string and is blank everywhere else. If the machine needs to store information, it may write this information on the tape. To read the information that it has written, the machine can move its head back over it. The machine continues computing until it decides to produce an output. The outputs *accept* and *reject* are obtained by entering designated accepting and rejecting states. If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting.

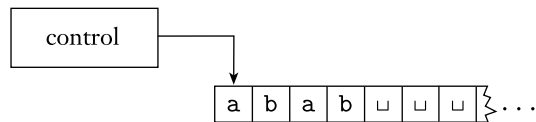


FIGURE 3.1
Schematic of a Turing machine

The following list summarizes the differences between finite automata and Turing machines.

1. A Turing machine can both write on the tape and read from it.
2. The read–write head can move both to the left and to the right.
3. The tape is infinite.
4. The special states for rejecting and accepting take effect immediately.

Let's introduce a Turing machine M_1 for testing membership in the language $B = \{w\#w \mid w \in \{0,1\}^*\}$. We want M_1 to accept if its input is a member of B and to reject otherwise. To understand M_1 better, put yourself in its place by imagining that you are standing on a mile-long input consisting of millions of characters. Your goal is to determine whether the input is a member of B —that is, whether the input comprises two identical strings separated by a $\#$ symbol. The input is too long for you to remember it all, but you are allowed to move back and forth over the input and make marks on it. The obvious strategy is to zig-zag to the corresponding places on the two sides of the $\#$ and determine whether they match. Place marks on the tape to keep track of which places correspond.

We design M_1 to work in that way. It makes multiple passes over the input string with the read–write head. On each pass it matches one of the characters on each side of the $\#$ symbol. To keep track of which symbols have been checked already, M_1 crosses off each symbol as it is examined. If it crosses off all the symbols, that means that everything matched successfully, and M_1 goes into an accept state. If it discovers a mismatch, it enters a reject state. In summary, M_1 's algorithm is as follows.

M_1 = “On input string w :

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, *reject*. Cross off symbols as they are checked to keep track of which symbols correspond.
2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, *reject*; otherwise, *accept*.”

The following figure contains several nonconsecutive snapshots of M_1 's tape after it is started on input 011000#011000.

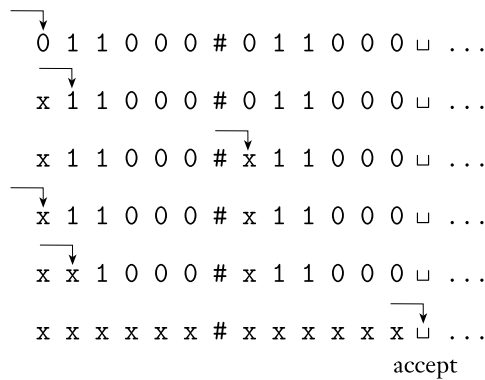


FIGURE 3.2

Snapshots of Turing machine M_1 computing on input 011000#011000

This description of Turing machine M_1 sketches the way it functions but does not give all its details. We can describe Turing machines in complete detail by giving formal descriptions analogous to those introduced for finite and push-down automata. The formal descriptions specify each of the parts of the formal definition of the Turing machine model to be presented shortly. In actuality, we almost never give formal descriptions of Turing machines because they tend to be very big.

FORMAL DEFINITION OF A TURING MACHINE

The heart of the definition of a Turing machine is the transition function δ because it tells us how the machine gets from one step to the next. For a Turing machine, δ takes the form: $Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$. That is, when the machine

is in a certain state q and the head is over a tape square containing a symbol a , and if $\delta(q, a) = (r, b, L)$, the machine writes the symbol b replacing the a , and goes to state r . The third component is either L or R and indicates whether the head moves to the left or right after writing. In this case, the L indicates a move to the left.

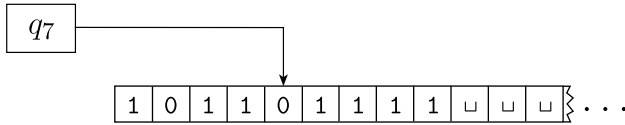
DEFINITION 3.3

A **Turing machine** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the **blank symbol** \sqcup ,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ computes as follows. Initially, M receives its input $w = w_1w_2 \dots w_n \in \Sigma^*$ on the leftmost n squares of the tape, and the rest of the tape is blank (i.e., filled with blank symbols). The head starts on the leftmost square of the tape. Note that Σ does not contain the blank symbol, so the first blank appearing on the tape marks the end of the input. Once M has started, the computation proceeds according to the rules described by the transition function. If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move, even though the transition function indicates L. The computation continues until it enters either the accept or reject states, at which point it halts. If neither occurs, M goes on forever.

As a Turing machine computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a **configuration** of the Turing machine. Configurations often are represented in a special way. For a state q and two strings u and v over the tape alphabet Γ , we write uqv for the configuration where the current state is q , the current tape contents is uv , and the current head location is the first symbol of v . The tape contains only blanks following the last symbol of v . For example, $1011q_701111$ represents the configuration when the tape is 101101111 , the current state is q_7 , and the head is currently on the second 0. Figure 3.4 depicts a Turing machine with that configuration.

**FIGURE 3.4**

A Turing machine with configuration 1011 q_7 01111

Here we formalize our intuitive understanding of the way that a Turing machine computes. Say that configuration C_1 **yields** configuration C_2 if the Turing machine can legally go from C_1 to C_2 in a single step. We define this notion formally as follows.

Suppose that we have a , b , and c in Γ , as well as u and v in Γ^* and states q_i and q_j . In that case, $uaq_i bv$ and $uq_j acv$ are two configurations. Say that

$$uaq_i bv \text{ yields } uq_j acv$$

if in the transition function $\delta(q_i, b) = (q_j, c, L)$. That handles the case where the Turing machine moves leftward. For a rightward move, say that

$$uaq_i bv \text{ yields } uacq_j v$$

if $\delta(q_i, b) = (q_j, c, R)$.

Special cases occur when the head is at one of the ends of the configuration. For the left-hand end, the configuration $q_i bv$ yields $q_j cv$ if the transition is left-moving (because we prevent the machine from going off the left-hand end of the tape), and it yields $cq_j v$ for the right-moving transition. For the right-hand end, the configuration uaq_i is equivalent to $uaq_i \sqcup$ because we assume that blanks follow the part of the tape represented in the configuration. Thus we can handle this case as before, with the head no longer at the right-hand end.

The **start configuration** of M on input w is the configuration $q_0 w$, which indicates that the machine is in the start state q_0 with its head at the leftmost position on the tape. In an **accepting configuration**, the state of the configuration is q_{accept} . In a **rejecting configuration**, the state of the configuration is q_{reject} . Accepting and rejecting configurations are **halting configurations** and do not yield further configurations. Because the machine is defined to halt when in the states q_{accept} and q_{reject} , we equivalently could have defined the transition function to have the more complicated form $\delta: Q' \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, where Q' is Q without q_{accept} and q_{reject} . A Turing machine M **accepts** input w if a sequence of configurations C_1, C_2, \dots, C_k exists, where

1. C_1 is the start configuration of M on input w ,
2. each C_i yields C_{i+1} , and
3. C_k is an accepting configuration.

The collection of strings that M accepts is *the language of M* , or *the language recognized by M* , denoted $L(M)$.

DEFINITION 3.5

Call a language *Turing-recognizable* if some Turing machine recognizes it.¹

When we start a Turing machine on an input, three outcomes are possible. The machine may *accept*, *reject*, or *loop*. By *loop* we mean that the machine simply does not halt. Looping may entail any simple or complex behavior that never leads to a halting state.

A Turing machine M can fail to accept an input by entering the q_{reject} state and rejecting, or by looping. Sometimes distinguishing a machine that is looping from one that is merely taking a long time is difficult. For this reason, we prefer Turing machines that halt on all inputs; such machines never loop. These machines are called *deciders* because they always make a decision to accept or reject. A decider that recognizes some language also is said to *decide* that language.

DEFINITION 3.6

Call a language *Turing-decidable* or simply *decidable* if some Turing machine decides it.²

Next, we give examples of decidable languages. Every decidable language is Turing-recognizable. We present examples of languages that are Turing-recognizable but not decidable after we develop a technique for proving undecidability in Chapter 4.

EXAMPLES OF TURING MACHINES

As we did for finite and pushdown automata, we can formally describe a particular Turing machine by specifying each of its seven parts. However, going to that level of detail can be cumbersome for all but the tiniest Turing machines. Accordingly, we won't spend much time giving such descriptions. Mostly we

¹It is called a *recursively enumerable language* in some other textbooks.

²It is called a *recursive language* in some other textbooks.

will give only higher level descriptions because they are precise enough for our purposes and are much easier to understand. Nevertheless, it is important to remember that every higher level description is actually just shorthand for its formal counterpart. With patience and care we could describe any of the Turing machines in this book in complete formal detail.

To help you make the connection between the formal descriptions and the higher level descriptions, we give state diagrams in the next two examples. You may skip over them if you already feel comfortable with this connection.

EXAMPLE 3.7

Here we describe a Turing machine (TM) M_2 that decides $A = \{0^{2^n} \mid n \geq 0\}$, the language consisting of all strings of 0s whose length is a power of 2.

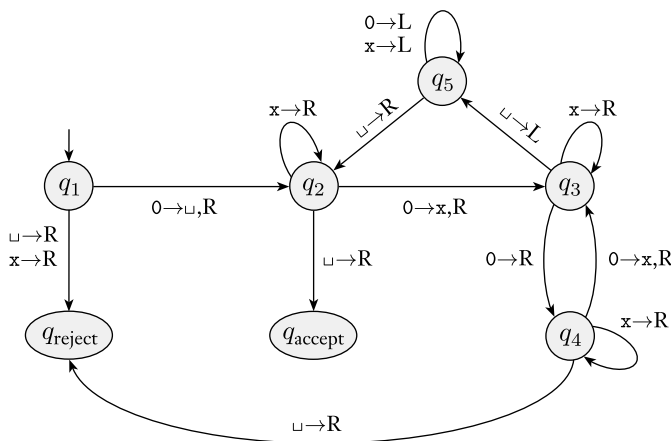
M_2 = “On input string w :

1. Sweep left to right across the tape, crossing off every other 0.
2. If in stage 1 the tape contained a single 0, *accept*.
3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, *reject*.
4. Return the head to the left-hand end of the tape.
5. Go to stage 1.”

Each iteration of stage 1 cuts the number of 0s in half. As the machine sweeps across the tape in stage 1, it keeps track of whether the number of 0s seen is even or odd. If that number is odd and greater than 1, the original number of 0s in the input could not have been a power of 2. Therefore, the machine rejects in this instance. However, if the number of 0s seen is 1, the original number must have been a power of 2. So in this case, the machine accepts.

Now we give the formal description of $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$:

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0\}$, and
- $\Gamma = \{0, x, \sqcup\}$.
- We describe δ with a state diagram (see Figure 3.8).
- The start, accept, and reject states are q_1 , q_{accept} , and q_{reject} , respectively.

**FIGURE 3.8**

State diagram for Turing machine M_2

In this state diagram, the label $0 \rightarrow \sqcup, R$ appears on the transition from q_1 to q_2 . This label signifies that when in state q_1 with the head reading 0, the machine goes to state q_2 , writes \sqcup , and moves the head to the right. In other words, $\delta(q_1, 0) = (q_2, \sqcup, R)$. For clarity we use the shorthand $0 \rightarrow R$ in the transition from q_3 to q_4 , to mean that the machine moves to the right when reading 0 in state q_3 but doesn't alter the tape, so $\delta(q_3, 0) = (q_4, 0, R)$.

This machine begins by writing a blank symbol over the leftmost 0 on the tape so that it can find the left-hand end of the tape in stage 4. Whereas we would normally use a more suggestive symbol such as # for the left-hand end delimiter, we use a blank here to keep the tape alphabet, and hence the state diagram, small. Example 3.11 gives another method of finding the left-hand end of the tape.

Next we give a sample run of this machine on input 0000. The starting configuration is $q_1 0000$. The sequence of configurations the machine enters appears as follows; read down the columns and left to right.

$q_1 0000$	$\sqcup q_5 x 0 x \sqcup$	$\sqcup x q_5 x x \sqcup$
$\sqcup q_2 000$	$q_5 \sqcup x 0 x \sqcup$	$\sqcup q_5 x x x \sqcup$
$\sqcup x q_3 00$	$\sqcup q_2 x 0 x \sqcup$	$q_5 \sqcup x x x \sqcup$
$\sqcup x 0 q_4 0$	$\sqcup x q_2 0 x \sqcup$	$\sqcup q_2 x x x \sqcup$
$\sqcup x 0 x q_3 \sqcup$	$\sqcup x x q_3 x \sqcup$	$\sqcup x q_2 x x \sqcup$
$\sqcup x 0 q_5 x \sqcup$	$\sqcup x x x q_3 \sqcup$	$\sqcup x x q_2 x \sqcup$
$\sqcup x q_5 0 x \sqcup$	$\sqcup x x q_5 x \sqcup$	$\sqcup x x x q_2 \sqcup$
		$\sqcup x x x \sqcup q_{\text{accept}}$

EXAMPLE 3.9

The following is a formal description of $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$, the Turing machine that we informally described (page 167) for deciding the language $B = \{w\#w \mid w \in \{0,1\}^*\}$.

- $Q = \{q_1, \dots, q_8, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0,1,\#\}$, and $\Gamma = \{0,1,\#,x,\sqcup\}$.
- We describe δ with a state diagram (see the following figure).
- The start, accept, and reject states are q_1 , q_{accept} , and q_{reject} , respectively.

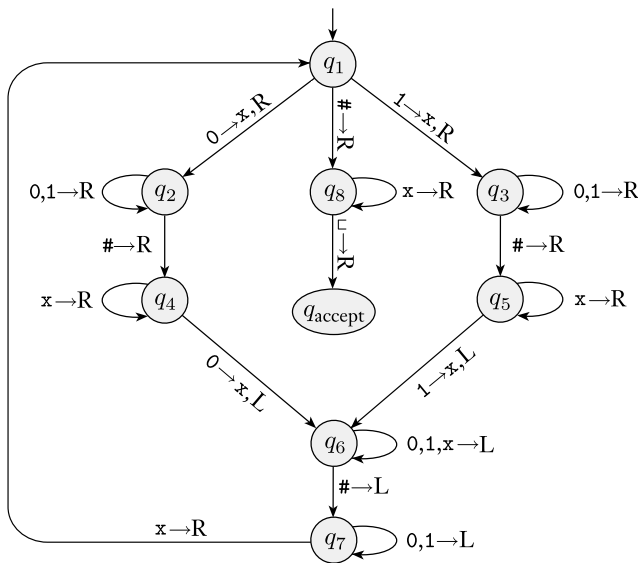


FIGURE 3.10
State diagram for Turing machine M_1

In Figure 3.10, which depicts the state diagram of TM M_1 , you will find the label $0,1 \rightarrow R$ on the transition going from q_3 to itself. That label means that the machine stays in q_3 and moves to the right when it reads a 0 or a 1 in state q_3 . It doesn't change the symbol on the tape.

Stage 1 is implemented by states q_1 through q_7 , and stage 2 by the remaining states. To simplify the figure, we don't show the reject state or the transitions going to the reject state. Those transitions occur implicitly whenever a state lacks an outgoing transition for a particular symbol. Thus because in state q_5 no outgoing arrow with a $\#$ is present, if a $\#$ occurs under the head when the machine is in state q_5 , it goes to state q_{reject} . For completeness, we say that the head moves right in each of these transitions to the reject state. ■

EXAMPLE 3.11

Here, a TM M_3 is doing some elementary arithmetic. It decides the language $C = \{a^i b^j c^k \mid i \times j = k \text{ and } i, j, k \geq 1\}$.

$M_3 =$ “On input string w :

1. Scan the input from left to right to determine whether it is a member of $a^+b^+c^+$ and *reject* if it isn't.
2. Return the head to the left-hand end of the tape.
3. Cross off an a and scan to the right until a b occurs. Shuttle between the b 's and the c 's, crossing off one of each until all b 's are gone. If all c 's have been crossed off and some b 's remain, *reject*.
4. Restore the crossed off b 's and repeat stage 3 if there is another a to cross off. If all a 's have been crossed off, determine whether all c 's also have been crossed off. If yes, *accept*; otherwise, *reject*.”

Let's examine the four stages of M_3 more closely. In stage 1, the machine operates like a finite automaton. No writing is necessary as the head moves from left to right, keeping track by using its states to determine whether the input is in the proper form.

Stage 2 looks equally simple but contains a subtlety. How can the TM find the left-hand end of the input tape? Finding the right-hand end of the input is easy because it is terminated with a blank symbol. But the left-hand end has no terminator initially. One technique that allows the machine to find the left-hand end of the tape is for it to mark the leftmost symbol in some way when the machine starts with its head on that symbol. Then the machine may scan left until it finds the mark when it wants to reset its head to the left-hand end. Example 3.7 illustrated this technique; a blank symbol marks the left-hand end.

A trickier method of finding the left-hand end of the tape takes advantage of the way that we defined the Turing machine model. Recall that if the machine tries to move its head beyond the left-hand end of the tape, it stays in the same place. We can use this feature to make a left-hand end detector. To detect whether the head is sitting on the left-hand end, the machine can write a special symbol over the current position while recording the symbol that it replaced in the control. Then it can attempt to move the head to the left. If it is still over the special symbol, the leftward move didn't succeed, and thus the head must have been at the left-hand end. If instead it is over a different symbol, some symbols remained to the left of that position on the tape. Before going farther, the machine must be sure to restore the changed symbol to the original.

Stages 3 and 4 have straightforward implementations and use several states each. ■

EXAMPLE 3.12

Here, a TM M_4 is solving what is called the *element distinctness problem*. It is given a list of strings over $\{0,1\}$ separated by #s and its job is to accept if all the strings are different. The language is

$$E = \{\#x_1\#x_2\#\cdots\#x_l \mid \text{each } x_i \in \{0,1\}^* \text{ and } x_i \neq x_j \text{ for each } i \neq j\}.$$

Machine M_4 works by comparing x_1 with x_2 through x_l , then by comparing x_2 with x_3 through x_l , and so on. An informal description of the TM M_4 deciding this language follows.

M_4 = “On input w :

1. Place a mark on top of the leftmost tape symbol. If that symbol was a blank, *accept*. If that symbol was a #, continue with the next stage. Otherwise, *reject*.
2. Scan right to the next # and place a second mark on top of it. If no # is encountered before a blank symbol, only x_1 was present, so *accept*.
3. By zig-zagging, compare the two strings to the right of the marked #s. If they are equal, *reject*.
4. Move the rightmost of the two marks to the next # symbol to the right. If no # symbol is encountered before a blank symbol, move the leftmost mark to the next # to its right and the rightmost mark to the # after that. This time, if no # is available for the rightmost mark, all the strings have been compared, so *accept*.
5. Go to stage 3.”

This machine illustrates the technique of marking tape symbols. In stage 2, the machine places a mark above a symbol, # in this case. In the actual implementation, the machine has two different symbols, # and $\overset{\cdot}{\#}$, in its tape alphabet. Saying that the machine places a mark above a # means that the machine writes the symbol $\overset{\cdot}{\#}$ at that location. Removing the mark means that the machine writes the symbol without the dot. In general, we may want to place marks over various symbols on the tape. To do so, we merely include versions of all these tape symbols with dots in the tape alphabet. ■

We conclude from the preceding examples that the described languages A , B , C , and E are decidable. All decidable languages are Turing-recognizable, so these languages are also Turing-recognizable. Demonstrating a language that is Turing-recognizable but undecidable is more difficult. We do so in Chapter 4.

3.2

VARIANTS OF TURING MACHINES

Alternative definitions of Turing machines abound, including versions with multiple tapes or with nondeterminism. They are called *variants* of the Turing machine model. The original model and its reasonable variants all have the same power—they recognize the same class of languages. In this section, we describe some of these variants and the proofs of equivalence in power. We call this invariance to certain changes in the definition *robustness*. Both finite automata and pushdown automata are somewhat robust models, but Turing machines have an astonishing degree of robustness.

To illustrate the robustness of the Turing machine model, let's vary the type of transition function permitted. In our definition, the transition function forces the head to move to the left or right after each step; the head may not simply stay put. Suppose that we had allowed the Turing machine the ability to stay put. The transition function would then have the form $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$. Might this feature allow Turing machines to recognize additional languages, thus adding to the power of the model? Of course not, because we can convert any TM with the “stay put” feature to one that does not have it. We do so by replacing each stay put transition with two transitions: one that moves to the right and the second back to the left.

This small example contains the key to showing the equivalence of TM variants. To show that two models are equivalent, we simply need to show that one can simulate the other.

MULTITAPE TURING MACHINES

A *multitape Turing machine* is like an ordinary Turing machine with several tapes. Each tape has its own head for reading and writing. Initially the input appears on tape 1, and the others start out blank. The transition function is changed to allow for reading, writing, and moving the heads on some or all of the tapes simultaneously. Formally, it is

$$\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, S\}^k,$$

where k is the number of tapes. The expression

$$\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, L, R, \dots, L)$$

means that if the machine is in state q_i and heads 1 through k are reading symbols a_1 through a_k , the machine goes to state q_j , writes symbols b_1 through b_k , and directs each head to move left or right, or to stay put, as specified.

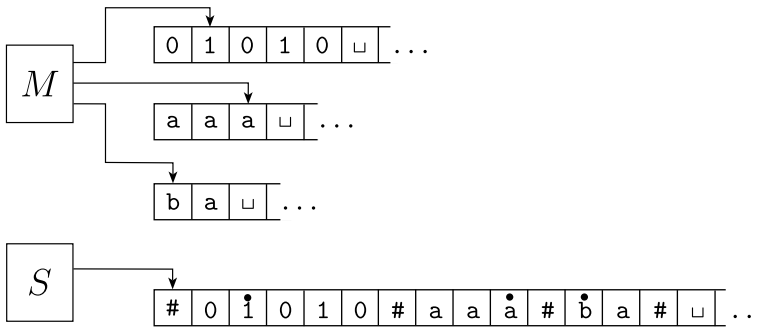
Multitape Turing machines appear to be more powerful than ordinary Turing machines, but we can show that they are equivalent in power. Recall that two machines are equivalent if they recognize the same language.

THEOREM 3.13

Every multitape Turing machine has an equivalent single-tape Turing machine.

PROOF We show how to convert a multitape TM M to an equivalent single-tape TM S . The key idea is to show how to simulate M with S .

Say that M has k tapes. Then S simulates the effect of k tapes by storing their information on its single tape. It uses the new symbol $\#$ as a delimiter to separate the contents of the different tapes. In addition to the contents of these tapes, S must keep track of the locations of the heads. It does so by writing a tape symbol with a dot above it to mark the place where the head on that tape would be. Think of these as “virtual” tapes and heads. As before, the “dotted” tape symbols are simply new symbols that have been added to the tape alphabet. The following figure illustrates how one tape can be used to represent three tapes.

**FIGURE 3.14**

Representing three tapes with one

$S =$ “On input $w = w_1 \cdots w_n$:

1. First S puts its tape into the format that represents all k tapes of M . The formatted tape contains

$$\# \overset{\cdot}{w}_1 \overset{\cdot}{w}_2 \cdots w_n \# \sqcup \# \overset{\cdot}{\sqcup} \# \cdots \#.$$

2. To simulate a single move, S scans its tape from the first $\#$, which marks the left-hand end, to the $(k + 1)$ st $\#$, which marks the right-hand end, in order to determine the symbols under the virtual heads. Then S makes a second pass to update the tapes according to the way that M ’s transition function dictates.
3. If at any point S moves one of the virtual heads to the right onto a $\#$, this action signifies that M has moved the corresponding head onto the previously unread blank portion of that tape. So S writes a blank symbol on this tape cell and shifts the tape contents, from this cell until the rightmost $\#$, one unit to the right. Then it continues the simulation as before.”

COROLLARY 3.15

A language is Turing-recognizable if and only if some multitape Turing machine recognizes it.

PROOF A Turing-recognizable language is recognized by an ordinary (single-tape) Turing machine, which is a special case of a multitape Turing machine. That proves one direction of this corollary. The other direction follows from Theorem 3.13.

NONDETERMINISTIC TURING MACHINES

A nondeterministic Turing machine is defined in the expected way. At any point in a computation, the machine may proceed according to several possibilities. The transition function for a nondeterministic Turing machine has the form

$$\delta: Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

The computation of a nondeterministic Turing machine is a tree whose branches correspond to different possibilities for the machine. If some branch of the computation leads to the accept state, the machine accepts its input. If you feel the need to review nondeterminism, turn to Section 1.2 (page 47). Now we show that nondeterminism does not affect the power of the Turing machine model.

THEOREM 3.16

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

PROOF IDEA We can simulate any nondeterministic TM N with a deterministic TM D . The idea behind the simulation is to have D try all possible branches of N 's nondeterministic computation. If D ever finds the accept state on one of these branches, D accepts. Otherwise, D 's simulation will not terminate.

We view N 's computation on an input w as a tree. Each branch of the tree represents one of the branches of the nondeterminism. Each node of the tree is a configuration of N . The root of the tree is the start configuration. The TM D searches this tree for an accepting configuration. Conducting this search carefully is crucial lest D fail to visit the entire tree. A tempting, though bad, idea is to have D explore the tree by using depth-first search. The depth-first search strategy goes all the way down one branch before backing up to explore other branches. If D were to explore the tree in this manner, D could go forever down one infinite branch and miss an accepting configuration on some other branch. Hence we design D to explore the tree by using breadth-first search instead. This strategy explores all branches to the same depth before going on to explore any branch to the next depth. This method guarantees that D will visit every node in the tree until it encounters an accepting configuration.

PROOF The simulating deterministic TM D has three tapes. By Theorem 3.13, this arrangement is equivalent to having a single tape. The machine D uses its three tapes in a particular way, as illustrated in the following figure. Tape 1 always contains the input string and is never altered. Tape 2 maintains a copy of N 's tape on some branch of its nondeterministic computation. Tape 3 keeps track of D 's location in N 's nondeterministic computation tree.

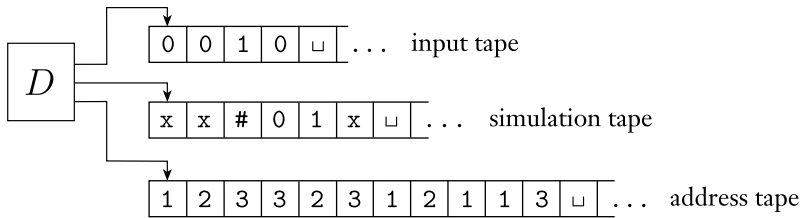


FIGURE 3.17
Deterministic TM D simulating nondeterministic TM N

Let's first consider the data representation on tape 3. Every node in the tree can have at most b children, where b is the size of the largest set of possible choices given by N 's transition function. To every node in the tree we assign an address that is a string over the alphabet $\Gamma_b = \{1, 2, \dots, b\}$. We assign the address 231 to the node we arrive at by starting at the root, going to its 2nd child, going to that node's 3rd child, and finally going to that node's 1st child. Each symbol in the string tells us which choice to make next when simulating a step in one branch in N 's nondeterministic computation. Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration. In that case, the address is invalid and doesn't correspond to any node. Tape 3 contains a string over Γ_b . It represents the branch of N 's computation from the root to the node addressed by that string unless the address is invalid. The empty string is the address of the root of the tree. Now we are ready to describe D .

1. Initially, tape 1 contains the input w , and tapes 2 and 3 are empty.
2. Copy tape 1 to tape 2 and initialize the string on tape 3 to be ϵ .
3. Use tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before each step of N , consult the next symbol on tape 3 to determine which choice to make among those allowed by N 's transition function. If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, *accept* the input.
4. Replace the string on tape 3 with the next string in the string ordering. Simulate the next branch of N 's computation by going to stage 2.

COROLLARY 3.18

A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.

PROOF Any deterministic TM is automatically a nondeterministic TM, and so one direction of this corollary follows immediately. The other direction follows from Theorem 3.16.

We can modify the proof of Theorem 3.16 so that if N always halts on all branches of its computation, D will always halt. We call a nondeterministic Turing machine a **decider** if all branches halt on all inputs. Exercise 3.3 asks you to modify the proof in this way to obtain the following corollary to Theorem 3.16.

COROLLARY 3.19

A language is decidable if and only if some nondeterministic Turing machine decides it.

ENUMERATORS

As we mentioned earlier, some people use the term *recursively enumerable language* for Turing-recognizable language. That term originates from a type of Turing machine variant called an **enumerator**. Loosely defined, an enumerator is a Turing machine with an attached printer. The Turing machine can use that printer as an output device to print strings. Every time the Turing machine wants to add a string to the list, it sends the string to the printer. Exercise 3.4 asks you to give a formal definition of an enumerator. The following figure depicts a schematic of this model.

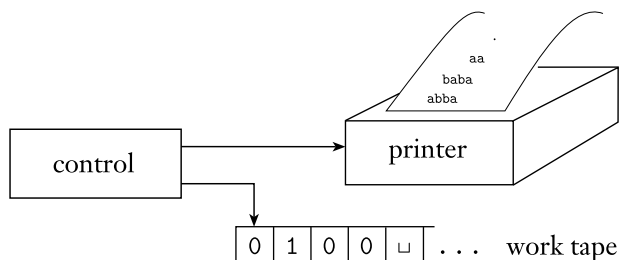


FIGURE 3.20
Schematic of an enumerator

An enumerator E starts with a blank input on its work tape. If the enumerator doesn't halt, it may print an infinite list of strings. The language enumerated by E is the collection of all the strings that it eventually prints out. Moreover, E may generate the strings of the language in any order, possibly with repetitions. Now we are ready to develop the connection between enumerators and Turing-recognizable languages.

THEOREM 3.21

A language is Turing-recognizable if and only if some enumerator enumerates it.

PROOF First we show that if we have an enumerator E that enumerates a language A , a TM M recognizes A . The TM M works in the following way.

$M =$ "On input w :

1. Run E . Every time that E outputs a string, compare it with w .
2. If w ever appears in the output of E , *accept*."

Clearly, M accepts those strings that appear on E 's list.

Now we do the other direction. If TM M recognizes a language A , we can construct the following enumerator E for A . Say that s_1, s_2, s_3, \dots is a list of all possible strings in Σ^* .

$E =$ "Ignore the input.

1. Repeat the following for $i = 1, 2, 3, \dots$
2. Run M for i steps on each input, s_1, s_2, \dots, s_i .
3. If any computations accept, print out the corresponding s_j ."

If M accepts a particular string s , eventually it will appear on the list generated by E . In fact, it will appear on the list infinitely many times because M runs from the beginning on each string for each repetition of step 1. This procedure gives the effect of running M in parallel on all possible input strings.

EQUIVALENCE WITH OTHER MODELS

So far we have presented several variants of the Turing machine model and have shown them to be equivalent in power. Many other models of general purpose computation have been proposed. Some of these models are very much like Turing machines, but others are quite different. All share the essential feature of Turing machines—namely, unrestricted access to unlimited memory—distinguishing them from weaker models such as finite automata and pushdown automata. Remarkably, *all* models with that feature turn out to be equivalent in power, so long as they satisfy reasonable requirements.³

³For example, one requirement is the ability to perform only a finite amount of work in a single step.

To understand this phenomenon, consider the analogous situation for programming languages. Many, such as Pascal and LISP, look quite different from one another in style and structure. Can some algorithm be programmed in one of them and not the others? Of course not—we can compile LISP into Pascal and Pascal into LISP, which means that the two languages describe *exactly* the same class of algorithms. So do all other reasonable programming languages. The widespread equivalence of computational models holds for precisely the same reason. Any two computational models that satisfy certain reasonable requirements can simulate one another and hence are equivalent in power.

This equivalence phenomenon has an important philosophical corollary. Even though we can imagine many different computational models, the class of algorithms that they describe remains the same. Whereas each individual computational model has a certain arbitrariness to its definition, the underlying class of algorithms that it describes is natural because the other models arrive at the same, unique class. This phenomenon has had profound implications for mathematics, as we show in the next section.

3.3

THE DEFINITION OF ALGORITHM

Informally speaking, an *algorithm* is a collection of simple instructions for carrying out some task. Commonplace in everyday life, algorithms sometimes are called *procedures* or *recipes*. Algorithms also play an important role in mathematics. Ancient mathematical literature contains descriptions of algorithms for a variety of tasks, such as finding prime numbers and greatest common divisors. In contemporary mathematics, algorithms abound.

Even though algorithms have had a long history in mathematics, the notion of algorithm itself was not defined precisely until the twentieth century. Before that, mathematicians had an intuitive notion of what algorithms were, and relied upon that notion when using and describing them. But that intuitive notion was insufficient for gaining a deeper understanding of algorithms. The following story relates how the precise definition of algorithm was crucial to one important mathematical problem.

HILBERT'S PROBLEMS

In 1900, mathematician David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris. In his lecture, he identified 23 mathematical problems and posed them as a challenge for the coming century. The tenth problem on his list concerned algorithms.

Before describing that problem, let's briefly discuss polynomials. A ***polynomial*** is a sum of terms, where each ***term*** is a product of certain variables and a

constant, called a *coefficient*. For example,

$$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z = 6x^3yz^2$$

is a term with coefficient 6, and

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

is a polynomial with four terms, over the variables x , y , and z . For this discussion, we consider only coefficients that are integers. A *root* of a polynomial is an assignment of values to its variables so that the value of the polynomial is 0. This polynomial has a root at $x = 5$, $y = 3$, and $z = 0$. This root is an *integral root* because all the variables are assigned integer values. Some polynomials have an integral root and some do not.

Hilbert's tenth problem was to devise an algorithm that tests whether a polynomial has an integral root. He did not use the term *algorithm* but rather "a process according to which it can be determined by a finite number of operations."⁴ Interestingly, in the way he phrased this problem, Hilbert explicitly asked that an algorithm be "devised." Thus he apparently assumed that such an algorithm must exist—someone need only find it.

As we now know, no algorithm exists for this task; it is algorithmically unsolvable. For mathematicians of that period to come to this conclusion with their intuitive concept of algorithm would have been virtually impossible. The intuitive concept may have been adequate for giving algorithms for certain tasks, but it was useless for showing that no algorithm exists for a particular task. Proving that an algorithm does not exist requires having a clear definition of algorithm. Progress on the tenth problem had to wait for that definition.

The definition came in the 1936 papers of Alonzo Church and Alan Turing. Church used a notational system called the λ -calculus to define algorithms. Turing did it with his "machines." These two definitions were shown to be equivalent. This connection between the informal notion of algorithm and the precise definition has come to be called the *Church–Turing thesis*.

The Church–Turing thesis provides the definition of algorithm necessary to resolve Hilbert's tenth problem. In 1970, Yuri Matijasevič, building on the work of Martin Davis, Hilary Putnam, and Julia Robinson, showed that no algorithm exists for testing whether a polynomial has integral roots. In Chapter 4 we develop the techniques that form the basis for proving that this and other problems are algorithmically unsolvable.

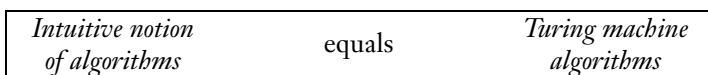


FIGURE 3.22
The Church–Turing thesis

⁴Translated from the original German.

Let's phrase Hilbert's tenth problem in our terminology. Doing so helps to introduce some themes that we explore in Chapters 4 and 5. Let

$$D = \{p \mid p \text{ is a polynomial with an integral root}\}.$$

Hilbert's tenth problem asks in essence whether the set D is decidable. The answer is negative. In contrast, we can show that D is Turing-recognizable. Before doing so, let's consider a simpler problem. It is an analog of Hilbert's tenth problem for polynomials that have only a single variable, such as $4x^3 - 2x^2 + x - 7$. Let

$$D_1 = \{p \mid p \text{ is a polynomial over } x \text{ with an integral root}\}.$$

Here is a TM M_1 that recognizes D_1 :

$M_1 =$ "On input $\langle p \rangle$: where p is a polynomial over the variable x .

1. Evaluate p with x set successively to the values $0, 1, -1, 2, -2, 3, -3, \dots$. If at any point the polynomial evaluates to 0, *accept*."

If p has an integral root, M_1 eventually will find it and accept. If p does not have an integral root, M_1 will run forever. For the multivariable case, we can present a similar TM M that recognizes D . Here, M goes through all possible settings of its variables to integral values.

Both M_1 and M are recognizers but not deciders. We can convert M_1 to be a decider for D_1 because we can calculate bounds within which the roots of a single variable polynomial must lie and restrict the search to these bounds. In Problem 3.21 you are asked to show that the roots of such a polynomial must lie between the values

$$\pm k \frac{c_{\max}}{c_1},$$

where k is the number of terms in the polynomial, c_{\max} is the coefficient with the largest absolute value, and c_1 is the coefficient of the highest order term. If a root is not found within these bounds, the machine *rejects*. Matijasevič's theorem shows that calculating such bounds for multivariable polynomials is impossible.

TERMINOLOGY FOR DESCRIBING TURING MACHINES

We have come to a turning point in the study of the theory of computation. We continue to speak of Turing machines, but our real focus from now on is on algorithms. That is, the Turing machine merely serves as a precise model for the definition of algorithm. We skip over the extensive theory of Turing machines themselves and do not spend much time on the low-level programming of Turing machines. We need only to be comfortable enough with Turing machines to believe that they capture all algorithms.

With that in mind, let's standardize the way we describe Turing machine algorithms. Initially, we ask: What is the right level of detail to give when describing

such algorithms? Students commonly ask this question, especially when preparing solutions to exercises and problems. Let's entertain three possibilities. The first is the *formal description* that spells out in full the Turing machine's states, transition function, and so on. It is the lowest, most detailed level of description. The second is a higher level of description, called the *implementation description*, in which we use English prose to describe the way that the Turing machine moves its head and the way that it stores data on its tape. At this level we do not give details of states or transition function. The third is the *high-level description*, wherein we use English prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head.

In this chapter, we have given formal and implementation-level descriptions of various examples of Turing machines. Practicing with lower level Turing machine descriptions helps you understand Turing machines and gain confidence in using them. Once you feel confident, high-level descriptions are sufficient.

We now set up a format and notation for describing Turing machines. The input to a Turing machine is always a string. If we want to provide an object other than a string as input, we must first represent that object as a string. Strings can easily represent polynomials, graphs, grammars, automata, and any combination of those objects. A Turing machine may be programmed to decode the representation so that it can be interpreted in the way we intend. Our notation for the encoding of an object O into its representation as a string is $\langle O \rangle$. If we have several objects O_1, O_2, \dots, O_k , we denote their encoding into a single string $\langle O_1, O_2, \dots, O_k \rangle$. The encoding itself can be done in many reasonable ways. It doesn't matter which one we pick because a Turing machine can always translate one such encoding into another.

In our format, we describe Turing machine algorithms with an indented segment of text within quotes. We break the algorithm into stages, each usually involving many individual steps of the Turing machine's computation. We indicate the block structure of the algorithm with further indentation. The first line of the algorithm describes the input to the machine. If the input description is simply w , the input is taken to be a string. If the input description is the encoding of an object as in $\langle A \rangle$, the Turing machine first implicitly tests whether the input properly encodes an object of the desired form and rejects it if it doesn't.

EXAMPLE 3.23

Let A be the language consisting of all strings representing undirected graphs that are connected. Recall that a graph is *connected* if every node can be reached from every other node by traveling along the edges of the graph. We write

$$A = \{ \langle G \rangle \mid G \text{ is a connected undirected graph} \}.$$

The following is a high-level description of a TM M that decides A .

M = “On input $\langle G \rangle$, the encoding of a graph G :

1. Select the first node of G and mark it.
2. Repeat the following stage until no new nodes are marked:
3. For each node in G , mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they all are marked. If they are, *accept*; otherwise, *reject*.”

For additional practice, let’s examine some implementation-level details of Turing machine M . Usually we won’t give this level of detail in the future and you won’t need to either, unless specifically requested to do so in an exercise. First, we must understand how $\langle G \rangle$ encodes the graph G as a string. Consider an encoding that is a list of the nodes of G followed by a list of the edges of G . Each node is a decimal number, and each edge is the pair of decimal numbers that represent the nodes at the two endpoints of the edge. The following figure depicts such a graph and its encoding.

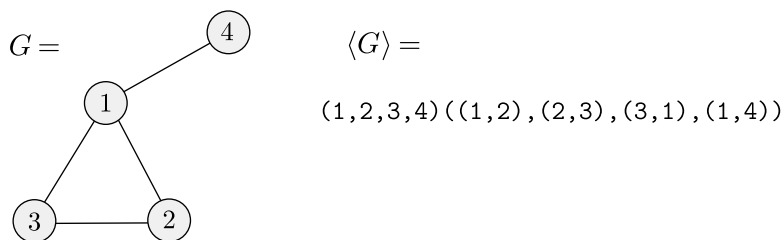


FIGURE 3.24

A graph G and its encoding $\langle G \rangle$

When M receives the input $\langle G \rangle$, it first checks to determine whether the input is the proper encoding of some graph. To do so, M scans the tape to be sure that there are two lists and that they are in the proper form. The first list should be a list of distinct decimal numbers, and the second should be a list of pairs of decimal numbers. Then M checks several things. First, the node list should contain no repetitions; and second, every node appearing on the edge list should also appear on the node list. For the first, we can use the procedure given in Example 3.12 for TM M_4 that checks element distinctness. A similar method works for the second check. If the input passes these checks, it is the encoding of some graph G . This verification completes the input check, and M goes on to stage 1.

For stage 1, M marks the first node with a dot on the leftmost digit.

For stage 2, M scans the list of nodes to find an undotted node n_1 and flags it by marking it differently—say, by underlining the first symbol. Then M scans the list again to find a dotted node n_2 and underlines it, too.

- ^A3.5 Examine the formal definition of a Turing machine to answer the following questions, and explain your reasoning.
- Can a Turing machine ever write the blank symbol \sqcup on its tape?
 - Can the tape alphabet Γ be the same as the input alphabet Σ ?
 - Can a Turing machine's head *ever* be in the same location in two successive steps?
 - Can a Turing machine contain just a single state?
- 3.6 In Theorem 3.21, we showed that a language is Turing-recognizable iff some enumerator enumerates it. Why didn't we use the following simpler algorithm for the forward direction of the proof? As before, s_1, s_2, \dots is a list of all strings in Σ^* .
- $E =$ "Ignore the input.
- Repeat the following for $i = 1, 2, 3, \dots$
 - Run M on s_i .
 - If it accepts, print out s_i ."
- 3.7 Explain why the following is not a description of a legitimate Turing machine.
- $M_{\text{bad}} =$ "On input $\langle p \rangle$, a polynomial over variables x_1, \dots, x_k :
- Try all possible settings of x_1, \dots, x_k to integer values.
 - Evaluate p on all of these settings.
 - If any of these settings evaluates to 0, *accept*; otherwise, *reject*."
- 3.8 Give implementation-level descriptions of Turing machines that decide the following languages over the alphabet $\{0, 1\}$.
- $\{w \mid w \text{ contains an equal number of 0s and 1s}\}$
 - $\{w \mid w \text{ contains twice as many 0s as 1s}\}$
 - $\{w \mid w \text{ does not contain twice as many 0s as 1s}\}$

PROBLEMS

- 3.9 Let a k -PDA be a pushdown automaton that has k stacks. Thus a 0-PDA is an NFA and a 1-PDA is a conventional PDA. You already know that 1-PDAs are more powerful (recognize a larger class of languages) than 0-PDAs.
- Show that 2-PDAs are more powerful than 1-PDAs.
 - Show that 3-PDAs are not more powerful than 2-PDAs.
(Hint: Simulate a Turing machine tape with two stacks.)
- ^A3.10 Say that a *write-once Turing machine* is a single-tape TM that can alter each tape square at most once (including the input portion of the tape). Show that this variant Turing machine model is equivalent to the ordinary Turing machine model. (Hint: As a first step, consider the case whereby the Turing machine may alter each tape square at most twice. Use lots of tape.)

- 3.11 A *Turing machine with doubly infinite tape* is similar to an ordinary Turing machine, but its tape is infinite to the left as well as to the right. The tape is initially filled with blanks except for the portion that contains the input. Computation is defined as usual except that the head never encounters an end to the tape as it moves leftward. Show that this type of Turing machine recognizes the class of Turing-recognizable languages.

- 3.12 A *Turing machine with left reset* is similar to an ordinary Turing machine, but the transition function has the form

$$\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{R, \text{RESET}\}.$$

If $\delta(q, a) = (r, b, \text{RESET})$, when the machine is in state q reading an a , the machine's head jumps to the left-hand end of the tape after it writes b on the tape and enters state r . Note that these machines do not have the usual ability to move the head one symbol left. Show that Turing machines with left reset recognize the class of Turing-recognizable languages.

- 3.13 A *Turing machine with stay put instead of left* is similar to an ordinary Turing machine, but the transition function has the form

$$\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{R, S\}.$$

At each point, the machine can move its head right or let it stay in the same position. Show that this Turing machine variant is *not* equivalent to the usual version. What class of languages do these machines recognize?

- 3.14 A *queue automaton* is like a push-down automaton except that the stack is replaced by a queue. A *queue* is a tape allowing symbols to be written only on the left-hand end and read only at the right-hand end. Each write operation (we'll call it a *push*) adds a symbol to the left-hand end of the queue and each read operation (we'll call it a *pull*) reads and removes a symbol at the right-hand end. As with a PDA, the input is placed on a separate read-only input tape, and the head on the input tape can move only from left to right. The input tape contains a cell with a blank symbol following the input, so that the end of the input can be detected. A queue automaton accepts its input by entering a special accept state at any time. Show that a language can be recognized by a deterministic queue automaton iff the language is Turing-recognizable.

- 3.15 Show that the collection of decidable languages is closed under the operation of

- | | |
|-------------------|---------------------|
| a. union. | d. complementation. |
| b. concatenation. | e. intersection. |
| c. star. | |

- 3.16 Show that the collection of Turing-recognizable languages is closed under the operation of

- | | |
|-------------------|------------------|
| a. union. | d. intersection. |
| b. concatenation. | e. homomorphism. |
| c. star. | |

- *3.17 Let $B = \{\langle M_1 \rangle, \langle M_2 \rangle, \dots\}$ be a Turing-recognizable language consisting of TM descriptions. Show that there is a decidable language C consisting of TM descriptions such that every machine described in B has an equivalent machine in C and vice versa.

- *3.18 Show that a language is decidable iff some enumerator enumerates the language in the standard string order.
- *3.19 Show that every infinite Turing-recognizable language has an infinite decidable subset.
- *3.20 Show that single-tape TMs that cannot write on the portion of the tape containing the input string recognize only regular languages.
- 3.21 Let $c_1x^n + c_2x^{n-1} + \cdots + c_nx + c_{n+1}$ be a polynomial with a root at $x = x_0$. Let c_{\max} be the largest absolute value of a c_i . Show that

$$|x_0| < (n+1) \frac{c_{\max}}{|c_1|}.$$

- 3.22** Let A be the language containing only the single string s , where

$$s = \begin{cases} 0 & \text{if life never will be found on Mars.} \\ 1 & \text{if life will be found on Mars someday.} \end{cases}$$

Is A decidable? Why or why not? For the purposes of this problem, assume that the question of whether life will be found on Mars has an unambiguous YES or NO answer.



SELECTED SOLUTIONS

- 3.1** (b) $q_1 00, \sqcup q_2 0, \sqcup x q_3 0, \sqcup q_5 x \sqcup, q_5 \sqcup x \sqcup, \sqcup q_2 x \sqcup, \sqcup x q_2 \sqcup, \sqcup x \sqcup q_{\text{accept}}$.
- 3.2** (a) $q_1 11, x q_3 1, x 1 q_3 \sqcup, x 1 \sqcup q_{\text{reject}}$.
- 3.3** We prove both directions of the iff. First, if a language L is decidable, it can be decided by a deterministic Turing machine, and that is automatically a nondeterministic Turing machine.
- Second, if a language L is decided by a nondeterministic TM N , we modify the deterministic TM D that was given in the proof of Theorem 3.16 as follows.
- Move stage 4 to be stage 5.
- Add new stage 4: *Reject* if all branches of N 's nondeterminism have rejected.
- We argue that this new TM D' is a decider for L . If N accepts its input, D' will eventually find an accepting branch and accept, too. If N rejects its input, all of its branches halt and reject because it is a decider. Hence each of the branches has finitely many nodes, where each node represents one step of N 's computation along that branch. Therefore, N 's entire computation tree on this input is finite, by virtue of the theorem about trees given in the statement of the exercise. Consequently, D' will halt and reject when this entire tree has been explored.
- 3.5** (a) Yes. The tape alphabet Γ contains \sqcup . A Turing machine can write any characters in Γ on its tape.
- (b) No. Σ never contains \sqcup , but Γ always contains \sqcup . So they cannot be equal.
- (c) Yes. If the Turing machine attempts to move its head off the left-hand end of the tape, it remains on the same tape cell.
- (d) No. Any Turing machine must contain two distinct states: q_{accept} and q_{reject} . So, a Turing machine contains at least two states.

3.8 (a) “On input string w :

1. Scan the tape and mark the first 0 that has not been marked. If no unmarked 0 is found, go to stage 4. Otherwise, move the head back to the front of the tape.
2. Scan the tape and mark the first 1 that has not been marked. If no unmarked 1 is found, *reject*.
3. Move the head back to the front of the tape and go to stage 1.
4. Move the head back to the front of the tape. Scan the tape to see if any unmarked 1s remain. If none are found, *accept*; otherwise, *reject*.”

3.10 We first simulate an ordinary Turing machine by a write-twice Turing machine. The write-twice machine simulates a single step of the original machine by copying the entire tape over to a fresh portion of the tape to the right-hand side of the currently used portion. The copying procedure operates character by character, marking a character as it is copied. This procedure alters each tape square twice: once to write the character for the first time, and again to mark that it has been copied. The position of the original Turing machine’s tape head is marked on the tape. When copying the cells at or adjacent to the marked position, the tape content is updated according to the rules of the original Turing machine.

To carry out the simulation with a write-once machine, operate as before, except that each cell of the previous tape is now represented by two cells. The first of these contains the original machine’s tape symbol and the second is for the mark used in the copying procedure. The input is not presented to the machine in the format with two cells per symbol, so the very first time the tape is copied, the copying marks are put directly over the input symbols.

3.15 (a) For any two decidable languages L_1 and L_2 , let M_1 and M_2 be the TMs that decide them. We construct a TM M' that decides the union of L_1 and L_2 :

“On input w :

1. Run M_1 on w . If it accepts, *accept*.
2. Run M_2 on w . If it accepts, *accept*. Otherwise, *reject*.”

M' accepts w if either M_1 or M_2 accepts it. If both reject, M' rejects.

3.16 (a) For any two Turing-recognizable languages L_1 and L_2 , let M_1 and M_2 be the TMs that recognize them. We construct a TM M' that recognizes the union of L_1 and L_2 :

“On input w :

1. Run M_1 and M_2 alternately on w step by step. If either accepts, *accept*. If both halt and reject, *reject*.”

If either M_1 or M_2 accepts w , M' accepts w because the accepting TM arrives to its accepting state after a finite number of steps. Note that if both M_1 and M_2 reject and either of them does so by looping, then M' will loop.

3.22 The language A is one of the two languages $\{0\}$ or $\{1\}$. In either case, the language is finite and hence decidable. If you aren’t able to determine which of these two languages is A , you won’t be able to describe the decider for A . However, you can give two Turing machines, one of which is A ’s decider.

4

DECIDABILITY

In Chapter 3 we introduced the Turing machine as a model of a general purpose computer and defined the notion of algorithm in terms of Turing machines by means of the Church–Turing thesis.

In this chapter we begin to investigate the power of algorithms to solve problems. We demonstrate certain problems that can be solved algorithmically and others that cannot. Our objective is to explore the limits of algorithmic solvability. You are probably familiar with solvability by algorithms because much of computer science is devoted to solving problems. The unsolvability of certain problems may come as a surprise.

Why should you study unsolvability? After all, showing that a problem is unsolvable doesn't appear to be of any use if you have to solve it. You need to study this phenomenon for two reasons. First, knowing when a problem is algorithmically unsolvable *is* useful because then you realize that the problem must be simplified or altered before you can find an algorithmic solution. Like any tool, computers have capabilities and limitations that must be appreciated if they are to be used well. The second reason is cultural. Even if you deal with problems that clearly are solvable, a glimpse of the unsolvable can stimulate your imagination and help you gain an important perspective on computation.

PROOF IDEA We simply need to present a TM M that decides A_{DFA} .

M = “On input $\langle B, w \rangle$, where B is a DFA and w is a string:

1. Simulate B on input w .
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*.”

PROOF We mention just a few implementation details of this proof. For those of you familiar with writing programs in any standard programming language, imagine how you would write a program to carry out the simulation.

First, let's examine the input $\langle B, w \rangle$. It is a representation of a DFA B together with a string w . One reasonable representation of B is simply a list of its five components: Q, Σ, δ, q_0 , and F . When M receives its input, M first determines whether it properly represents a DFA B and a string w . If not, M rejects.

Then M carries out the simulation directly. It keeps track of B 's current state and B 's current position in the input w by writing this information down on its tape. Initially, B 's current state is q_0 and B 's current input position is the leftmost symbol of w . The states and position are updated according to the specified transition function δ . When M finishes processing the last symbol of w , M accepts the input if B is in an accepting state; M rejects the input if B is in a nonaccepting state.

We can prove a similar theorem for nondeterministic finite automata. Let

$$A_{\text{NFA}} = \{ \langle B, w \rangle \mid B \text{ is an NFA that accepts input string } w \}.$$

THEOREM 4.2

A_{NFA} is a decidable language.

PROOF We present a TM N that decides A_{NFA} . We could design N to operate like M , simulating an NFA instead of a DFA. Instead, we'll do it differently to illustrate a new idea: Have N use M as a subroutine. Because M is designed to work with DFAs, N first converts the NFA it receives as input to a DFA before passing it to M .

N = “On input $\langle B, w \rangle$, where B is an NFA and w is a string:

1. Convert NFA B to an equivalent DFA C , using the procedure for this conversion given in Theorem 1.39.
2. Run TM M from Theorem 4.1 on input $\langle C, w \rangle$.
3. If M accepts, *accept*; otherwise, *reject*.”

Running TM M in stage 2 means incorporating M into the design of N as a subprocedure.

Similarly, we can determine whether a regular expression generates a given string. Let $A_{\text{REX}} = \{\langle R, w \rangle \mid R \text{ is a regular expression that generates string } w\}$.

THEOREM 4.3

A_{REX} is a decidable language.

PROOF The following TM P decides A_{REX} .

$P =$ “On input $\langle R, w \rangle$, where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent NFA A by using the procedure for this conversion given in Theorem 1.54.
2. Run TM N on input $\langle A, w \rangle$.
3. If N accepts, *accept*; if N rejects, *reject*.”

Theorems 4.1, 4.2, and 4.3 illustrate that, for decidability purposes, it is equivalent to present the Turing machine with a DFA, an NFA, or a regular expression because the machine can convert one form of encoding to another.

Now we turn to a different kind of problem concerning finite automata: *emptiness testing* for the language of a finite automaton. In the preceding three theorems we had to determine whether a finite automaton accepts a particular string. In the next proof we must determine whether or not a finite automaton accepts any strings at all. Let

$$E_{\text{DFA}} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}.$$

THEOREM 4.4

E_{DFA} is a decidable language.

PROOF A DFA accepts some string iff reaching an accept state from the start state by traveling along the arrows of the DFA is possible. To test this condition, we can design a TM T that uses a marking algorithm similar to that used in Example 3.23.

$T =$ “On input $\langle A \rangle$, where A is a DFA:

1. Mark the start state of A .
2. Repeat until no new states get marked:
3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, *accept*; otherwise, *reject*.”

The next theorem states that determining whether two DFAs recognize the same language is decidable. Let

$$EQ_{DFA} = \{\langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}.$$

THEOREM 4.5

EQ_{DFA} is a decidable language.

PROOF To prove this theorem, we use Theorem 4.4. We construct a new DFA C from A and B , where C accepts only those strings that are accepted by either A or B but not by both. Thus, if A and B recognize the same language, C will accept nothing. The language of C is

$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B)).$$

This expression is sometimes called the **symmetric difference** of $L(A)$ and $L(B)$ and is illustrated in the following figure. Here, $\overline{L(A)}$ is the complement of $L(A)$. The symmetric difference is useful here because $L(C) = \emptyset$ iff $L(A) = L(B)$. We can construct C from A and B with the constructions for proving the class of regular languages closed under complementation, union, and intersection. These constructions are algorithms that can be carried out by Turing machines. Once we have constructed C , we can use Theorem 4.4 to test whether $L(C)$ is empty. If it is empty, $L(A)$ and $L(B)$ must be equal.

$F =$ “On input $\langle A, B \rangle$, where A and B are DFAs:

1. Construct DFA C as described.
2. Run TM T from Theorem 4.4 on input $\langle C \rangle$.
3. If T accepts, *accept*. If T rejects, *reject*.”

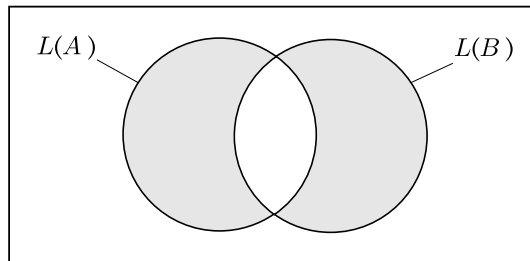


FIGURE 4.6

The symmetric difference of $L(A)$ and $L(B)$

DECIDABLE PROBLEMS CONCERNING CONTEXT-FREE LANGUAGES

Here, we describe algorithms to determine whether a CFG generates a particular string and to determine whether the language of a CFG is empty. Let

$$A_{\text{CFG}} = \{\langle G, w \rangle \mid G \text{ is a CFG that generates string } w\}.$$

THEOREM 4.7

A_{CFG} is a decidable language.

PROOF IDEA For CFG G and string w , we want to determine whether G generates w . One idea is to use G to go through all derivations to determine whether any is a derivation of w . This idea doesn't work, as infinitely many derivations may have to be tried. If G does not generate w , this algorithm would never halt. This idea gives a Turing machine that is a recognizer, but not a decider, for A_{CFG} .

To make this Turing machine into a decider, we need to ensure that the algorithm tries only finitely many derivations. In Problem 2.26 (page 157) we showed that if G were in Chomsky normal form, any derivation of w has $2n - 1$ steps, where n is the length of w . In that case, checking only derivations with $2n - 1$ steps to determine whether G generates w would be sufficient. Only finitely many such derivations exist. We can convert G to Chomsky normal form by using the procedure given in Section 2.1.

PROOF The TM S for A_{CFG} follows.

$S =$ "On input $\langle G, w \rangle$, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n - 1$ steps, where n is the length of w ; except if $n = 0$, then instead list all derivations with one step.
3. If any of these derivations generate w , *accept*; if not, *reject*."

The problem of determining whether a CFG generates a particular string is related to the problem of compiling programming languages. The algorithm in TM S is very inefficient and would never be used in practice, but it is easy to describe and we aren't concerned with efficiency here. In Part Three of this book, we address issues concerning the running time and memory use of algorithms. In the proof of Theorem 7.16, we describe a more efficient algorithm for recognizing general context-free languages. Even greater efficiency is possible for recognizing deterministic context-free languages.

Recall that we have given procedures for converting back and forth between CFGs and PDAs in Theorem 2.20. Hence everything we say about the decidability of problems concerning CFGs applies equally well to PDAs.

Let's turn now to the emptiness testing problem for the language of a CFG. As we did for DFAs, we can show that the problem of determining whether a CFG generates any strings at all is decidable. Let

$$E_{CFG} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}.$$

THEOREM 4.8

E_{CFG} is a decidable language.

PROOF IDEA To find an algorithm for this problem, we might attempt to use TM S from Theorem 4.7. It states that we can test whether a CFG generates some particular string w . To determine whether $L(G) = \emptyset$, the algorithm might try going through all possible w 's, one by one. But there are infinitely many w 's to try, so this method could end up running forever. We need to take a different approach.

In order to determine whether the language of a grammar is empty, we need to test whether the start variable can generate a string of terminals. The algorithm does so by solving a more general problem. It determines *for each variable* whether that variable is capable of generating a string of terminals. When the algorithm has determined that a variable can generate some string of terminals, the algorithm keeps track of this information by placing a mark on that variable.

First, the algorithm marks all the terminal symbols in the grammar. Then, it scans all the rules of the grammar. If it ever finds a rule that permits some variable to be replaced by some string of symbols, all of which are already marked, the algorithm knows that this variable can be marked, too. The algorithm continues in this way until it cannot mark any additional variables. The TM R implements this algorithm.

PROOF

$R =$ "On input $\langle G \rangle$, where G is a CFG:

1. Mark all terminal symbols in G .
2. Repeat until no new variables get marked:
3. Mark any variable A where G has a rule $A \rightarrow U_1 U_2 \cdots U_k$ and each symbol U_1, \dots, U_k has already been marked.
4. If the start variable is not marked, *accept*; otherwise, *reject*."

Next, we consider the problem of determining whether two context-free grammars generate the same language. Let

$$EQ_{CFG} = \{\langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}.$$

Theorem 4.5 gave an algorithm that decides the analogous language EQ_{DFA} for finite automata. We used the decision procedure for E_{DFA} to prove that EQ_{DFA} is decidable. Because E_{CFG} also is decidable, you might think that we can use a similar strategy to prove that EQ_{CFG} is decidable. But something is wrong with this idea! The class of context-free languages is *not* closed under complementation or intersection, as you proved in Exercise 2.2. In fact, EQ_{CFG} is not decidable. The technique for proving so is presented in Chapter 5.

Now we show that context-free languages are decidable by Turing machines.

THEOREM 4.9

Every context-free language is decidable.

PROOF IDEA Let A be a CFL. Our objective is to show that A is decidable. One (bad) idea is to convert a PDA for A directly into a TM. That isn't hard to do because simulating a stack with the TM's more versatile tape is easy. The PDA for A may be nondeterministic, but that seems okay because we can convert it into a nondeterministic TM and we know that any nondeterministic TM can be converted into an equivalent deterministic TM. Yet, there is a difficulty. Some branches of the PDA's computation may go on forever, reading and writing the stack without ever halting. The simulating TM then would also have some non-halting branches in its computation, and so the TM would not be a decider. A different idea is necessary. Instead, we prove this theorem with the TM S that we designed in Theorem 4.7 to decide A_{CFG} .

PROOF Let G be a CFG for A and design a TM M_G that decides A . We build a copy of G into M_G . It works as follows.

M_G = "On input w :

1. Run TM S on input $\langle G, w \rangle$.
2. If this machine accepts, *accept*; if it rejects, *reject*."

Theorem 4.9 provides the final link in the relationship among the four main classes of languages that we have described so far: regular, context-free, decidable, and Turing-recognizable. Figure 4.10 depicts this relationship.

given input string. We call it A_{TM} by analogy with A_{DFA} and A_{CFG} . But, whereas A_{DFA} and A_{CFG} were decidable, A_{TM} is not. Let

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}.$$

THEOREM 4.11

A_{TM} is undecidable.

Before we get to the proof, let's first observe that A_{TM} is Turing-recognizable. Thus, this theorem shows that recognizers *are* more powerful than deciders. Requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize. The following Turing machine U recognizes A_{TM} .

U = "On input $\langle M, w \rangle$, where M is a TM and w is a string:

1. Simulate M on input w .
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

Note that this machine loops on input $\langle M, w \rangle$ if M loops on w , which is why this machine does not decide A_{TM} . If the algorithm had some way to determine that M was not halting on w , it could *reject* in this case. However, an algorithm has no way to make this determination, as we shall see.

The Turing machine U is interesting in its own right. It is an example of the *universal Turing machine* first proposed by Alan Turing in 1936. This machine is called universal because it is capable of simulating any other Turing machine from the description of that machine. The universal Turing machine played an important early role in the development of stored-program computers.

THE DIAGONALIZATION METHOD

The proof of the undecidability of A_{TM} uses a technique called *diagonalization*, discovered by mathematician Georg Cantor in 1873. Cantor was concerned with the problem of measuring the sizes of infinite sets. If we have two infinite sets, how can we tell whether one is larger than the other or whether they are of the same size? For finite sets, of course, answering these questions is easy. We simply count the elements in a finite set, and the resulting number is its size. But if we try to count the elements of an infinite set, we will never finish! So we can't use the counting method to determine the relative sizes of infinite sets.

For example, take the set of even integers and the set of all strings over $\{0,1\}$. Both sets are infinite and thus larger than any finite set, but is one of the two larger than the other? How can we compare their relative size?

Cantor proposed a rather nice solution to this problem. He observed that two finite sets have the same size if the elements of one set can be paired with the elements of the other set. This method compares the sizes without resorting to counting. We can extend this idea to infinite sets. Here it is more precisely.

DEFINITION 4.12

Assume that we have sets A and B and a function f from A to B . Say that f is **one-to-one** if it never maps two different elements to the same place—that is, if $f(a) \neq f(b)$ whenever $a \neq b$. Say that f is **onto** if it hits every element of B —that is, if for every $b \in B$ there is an $a \in A$ such that $f(a) = b$. Say that A and B are the **same size** if there is a one-to-one, onto function $f: A \rightarrow B$. A function that is both one-to-one and onto is called a **correspondence**. In a correspondence, every element of A maps to a unique element of B and each element of B has a unique element of A mapping to it. A correspondence is simply a way of pairing the elements of A with the elements of B .

Alternative common terminology for these types of functions is **injective** for one-to-one, **surjective** for onto, and **bijective** for one-to-one and onto.

EXAMPLE 4.13

Let \mathcal{N} be the set of natural numbers $\{1, 2, 3, \dots\}$ and let \mathcal{E} be the set of even natural numbers $\{2, 4, 6, \dots\}$. Using Cantor's definition of size, we can see that \mathcal{N} and \mathcal{E} have the same size. The correspondence f mapping \mathcal{N} to \mathcal{E} is simply $f(n) = 2n$. We can visualize f more easily with the help of a table.

n	$f(n)$
1	2
2	4
3	6
\vdots	\vdots

Of course, this example seems bizarre. Intuitively, \mathcal{E} seems smaller than \mathcal{N} because \mathcal{E} is a proper subset of \mathcal{N} . But pairing each member of \mathcal{N} with its own member of \mathcal{E} is possible, so we declare these two sets to be the same size. ■

DEFINITION 4.14

A set A is **countable** if either it is finite or it has the same size as \mathcal{N} .

EXAMPLE 4.15

Now we turn to an even stranger example. If we let $\mathcal{Q} = \{\frac{m}{n} \mid m, n \in \mathcal{N}\}$ be the set of positive rational numbers, \mathcal{Q} seems to be much larger than \mathcal{N} . Yet these two sets are the same size according to our definition. We give a correspondence with \mathcal{N} to show that \mathcal{Q} is countable. One easy way to do so is to list all the

elements of \mathcal{Q} . Then we pair the first element on the list with the number 1 from \mathcal{N} , the second element on the list with the number 2 from \mathcal{N} , and so on. We must ensure that every member of \mathcal{Q} appears only once on the list.

To get this list, we make an infinite matrix containing all the positive rational numbers, as shown in Figure 4.16. The i th row contains all numbers with numerator i and the j th column has all numbers with denominator j . So the number $\frac{i}{j}$ occurs in the i th row and j th column.

Now we turn this matrix into a list. One (bad) way to attempt it would be to begin the list with all the elements in the first row. That isn't a good approach because the first row is infinite, so the list would never get to the second row. Instead we list the elements on the diagonals, which are superimposed on the diagram, starting from the corner. The first diagonal contains the single element $\frac{1}{1}$, and the second diagonal contains the two elements $\frac{2}{1}$ and $\frac{1}{2}$. So the first three elements on the list are $\frac{1}{1}$, $\frac{2}{1}$, and $\frac{1}{2}$. In the third diagonal, a complication arises. It contains $\frac{3}{1}$, $\frac{2}{2}$, and $\frac{1}{3}$. If we simply added these to the list, we would repeat $\frac{1}{1} = \frac{2}{2}$. We avoid doing so by skipping an element when it would cause a repetition. So we add only the two new elements $\frac{3}{1}$ and $\frac{1}{3}$. Continuing in this way, we obtain a list of all the elements of \mathcal{Q} .

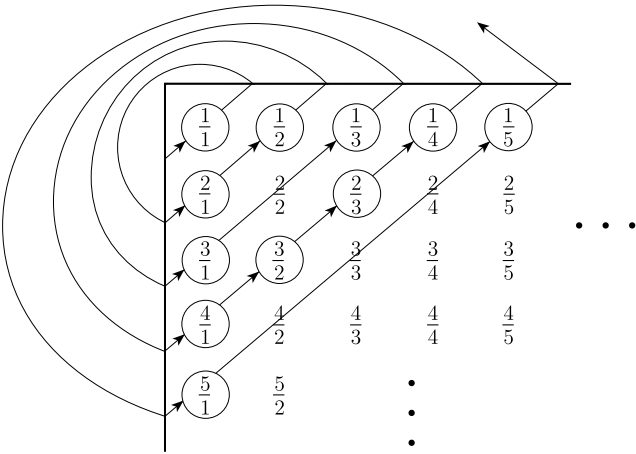


FIGURE 4.16
A correspondence of \mathcal{N} and \mathcal{Q}

After seeing the correspondence of \mathcal{N} and \mathcal{Q} , you might think that any two infinite sets can be shown to have the same size. After all, you need only demonstrate a correspondence, and this example shows that surprising correspondences do exist. However, for some infinite sets, no correspondence with \mathcal{N} exists. These sets are simply too big. Such sets are called *uncountable*.

The set of real numbers is an example of an uncountable set. A *real number* is one that has a decimal representation. The numbers $\pi = 3.1415926\dots$ and

$\sqrt{2} = 1.4142135\dots$ are examples of real numbers. Let \mathcal{R} be the set of real numbers. Cantor proved that \mathcal{R} is uncountable. In doing so, he introduced the diagonalization method.

THEOREM 4.17

\mathcal{R} is uncountable.

PROOF In order to show that \mathcal{R} is uncountable, we show that no correspondence exists between \mathcal{N} and \mathcal{R} . The proof is by contradiction. Suppose that a correspondence f existed between \mathcal{N} and \mathcal{R} . Our job is to show that f fails to work as it should. For it to be a correspondence, f must pair all the members of \mathcal{N} with all the members of \mathcal{R} . But we will find an x in \mathcal{R} that is not paired with anything in \mathcal{N} , which will be our contradiction.

The way we find this x is by actually constructing it. We choose each digit of x to make x different from one of the real numbers that is paired with an element of \mathcal{N} . In the end, we are sure that x is different from any real number that is paired.

We can illustrate this idea by giving an example. Suppose that the correspondence f exists. Let $f(1) = 3.14159\dots$, $f(2) = 55.55555\dots$, $f(3) = \dots$, and so on, just to make up some values for f . Then f pairs the number 1 with $3.14159\dots$, the number 2 with $55.55555\dots$, and so on. The following table shows a few values of a hypothetical correspondence f between \mathcal{N} and \mathcal{R} .

n	$f(n)$
1	3.14159...
2	55.55555...
3	0.12345...
4	0.50000...
\vdots	\vdots

We construct the desired x by giving its decimal representation. It is a number between 0 and 1, so all its significant digits are fractional digits following the decimal point. Our objective is to ensure that $x \neq f(n)$ for any n . To ensure that $x \neq f(1)$, we let the first digit of x be anything different from the first fractional digit 1 of $f(1) = 3.\underline{1}4159\dots$. Arbitrarily, we let it be 4. To ensure that $x \neq f(2)$, we let the second digit of x be anything different from the second fractional digit 5 of $f(2) = 55.\underline{5}5555\dots$. Arbitrarily, we let it be 6. The third fractional digit of $f(3) = 0.12\underline{3}45\dots$ is 3, so we let x be anything different—say, 4. Continuing in this way down the diagonal of the table for f , we obtain all the digits of x , as shown in the following table. We know that x is not $f(n)$ for any n because it differs from $f(n)$ in the n th fractional digit. (A slight problem arises because certain numbers, such as $0.1999\dots$ and $0.2000\dots$, are equal even though their decimal representations are different. We avoid this problem by never selecting the digits 0 or 9 when we construct x .)

n	$f(n)$	
1	3. <u>1</u> 4159...	
2	55.5 <u>5</u> 555...	
3	0.123 <u>4</u> 5...	$x = 0.4641 \dots$
4	0.500 <u>0</u> 0...	
\vdots	\vdots	

The preceding theorem has an important application to the theory of computation. It shows that some languages are not decidable or even Turing-recognizable, for the reason that there are uncountably many languages yet only countably many Turing machines. Because each Turing machine can recognize a single language and there are more languages than Turing machines, some languages are not recognized by any Turing machine. Such languages are not Turing-recognizable, as we state in the following corollary.

COROLLARY 4.18

Some languages are not Turing-recognizable.

PROOF To show that the set of all Turing machines is countable, we first observe that the set of all strings Σ^* is countable for any alphabet Σ . With only finitely many strings of each length, we may form a list of Σ^* by writing down all strings of length 0, length 1, length 2, and so on.

The set of all Turing machines is countable because each Turing machine M has an encoding into a string $\langle M \rangle$. If we simply omit those strings that are not legal encodings of Turing machines, we can obtain a list of all Turing machines.

To show that the set of all languages is uncountable, we first observe that the set of all infinite binary sequences is uncountable. An *infinite binary sequence* is an unending sequence of 0s and 1s. Let \mathcal{B} be the set of all infinite binary sequences. We can show that \mathcal{B} is uncountable by using a proof by diagonalization similar to the one we used in Theorem 4.17 to show that \mathcal{R} is uncountable.

Let \mathcal{L} be the set of all languages over alphabet Σ . We show that \mathcal{L} is uncountable by giving a correspondence with \mathcal{B} , thus showing that the two sets are the same size. Let $\Sigma^* = \{s_1, s_2, s_3, \dots\}$. Each language $A \in \mathcal{L}$ has a unique sequence in \mathcal{B} . The i th bit of that sequence is a 1 if $s_i \in A$ and is a 0 if $s_i \notin A$, which is called the *characteristic sequence* of A . For example, if A were the language of all strings starting with a 0 over the alphabet $\{0,1\}$, its characteristic sequence χ_A would be

$$\begin{array}{lcl} \Sigma^* = \{ & \epsilon, & 0, \quad 1, \quad 00, \quad 01, \quad 10, \quad 11, \quad 000, \quad 001, \quad \dots \} ; \\ A = \{ & & 0, \quad \quad 00, \quad 01, \quad \quad \quad 000, \quad 001, \quad \dots \} ; \\ \chi_A = & 0 & 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad \dots \end{array}$$

The function $f: \mathcal{L} \rightarrow \mathcal{B}$, where $f(A)$ equals the characteristic sequence of A , is one-to-one and onto, and hence is a correspondence. Therefore, as \mathcal{B} is uncountable, \mathcal{L} is uncountable as well.

Thus we have shown that the set of all languages cannot be put into a correspondence with the set of all Turing machines. We conclude that some languages are not recognized by any Turing machine.

AN UNDECIDABLE LANGUAGE

Now we are ready to prove Theorem 4.11, the undecidability of the language

$$A_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w\}.$$

PROOF We assume that A_{TM} is decidable and obtain a contradiction. Suppose that H is a decider for A_{TM} . On input $\langle M, w \rangle$, where M is a TM and w is a string, H halts and accepts if M accepts w . Furthermore, H halts and rejects if M fails to accept w . In other words, we assume that H is a TM, where

$$H(\langle M, w \rangle) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{if } M \text{ does not accept } w. \end{cases}$$

Now we construct a new Turing machine D with H as a subroutine. This new TM calls H to determine what M does when the input to M is its own description $\langle M \rangle$. Once D has determined this information, it does the opposite. That is, it rejects if M accepts and accepts if M does not accept. The following is a description of D .

$D =$ “On input $\langle M \rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M \rangle \rangle$.
2. Output the opposite of what H outputs. That is, if H accepts, *reject*; and if H rejects, *accept*.”

Don’t be confused by the notion of running a machine on its own description! That is similar to running a program with itself as input, something that does occasionally occur in practice. For example, a compiler is a program that translates other programs. A compiler for the language Python may itself be written in Python, so running that program on itself would make sense. In summary,

$$D(\langle M \rangle) = \begin{cases} \text{accept} & \text{if } M \text{ does not accept } \langle M \rangle \\ \text{reject} & \text{if } M \text{ accepts } \langle M \rangle. \end{cases}$$

What happens when we run D with its own description $\langle D \rangle$ as input? In that case, we get

$$D(\langle D \rangle) = \begin{cases} \text{accept} & \text{if } D \text{ does not accept } \langle D \rangle \\ \text{reject} & \text{if } D \text{ accepts } \langle D \rangle. \end{cases}$$

No matter what D does, it is forced to do the opposite, which is obviously a contradiction. Thus, neither TM D nor TM H can exist.

Let's review the steps of this proof. Assume that a TM H decides A_{TM} . Use H to build a TM D that takes an input $\langle M \rangle$, where D accepts its input $\langle M \rangle$ exactly when M does not accept its input $\langle M \rangle$. Finally, run D on itself. Thus, the machines take the following actions, with the last line being the contradiction.

- H accepts $\langle M, w \rangle$ exactly when M accepts w .
- D rejects $\langle M \rangle$ exactly when M accepts $\langle M \rangle$.
- D rejects $\langle D \rangle$ exactly when D accepts $\langle D \rangle$.

Where is the diagonalization in the proof of Theorem 4.11? It becomes apparent when you examine tables of behavior for TMs H and D . In these tables we list all TMs down the rows, M_1, M_2, \dots , and all their descriptions across the columns, $\langle M_1 \rangle, \langle M_2 \rangle, \dots$. The entries tell whether the machine in a given row accepts the input in a given column. The entry is *accept* if the machine accepts the input but is blank if it rejects or loops on that input. We made up the entries in the following figure to illustrate the idea.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	\dots
M_1	accept		accept		
M_2	accept	accept	accept	accept	
M_3					\dots
M_4	accept	accept			
\vdots			\vdots		

FIGURE 4.19
Entry i, j is *accept* if M_i accepts $\langle M_j \rangle$

In the following figure, the entries are the results of running H on inputs corresponding to Figure 4.19. So if M_3 does not accept input $\langle M_2 \rangle$, the entry for row M_3 and column $\langle M_2 \rangle$ is *reject* because H rejects input $\langle M_3, \langle M_2 \rangle \rangle$.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	\dots
M_1	accept	reject	accept	reject	
M_2	accept	accept	accept	accept	
M_3	reject	reject	reject	reject	\dots
M_4	accept	accept	reject	reject	
\vdots			\vdots		

FIGURE 4.20
Entry i, j is the value of H on input $\langle M_i, \langle M_j \rangle \rangle$

In the following figure, we added D to Figure 4.20. By our assumption, H is a TM and so is D . Therefore, it must occur on the list M_1, M_2, \dots of all TMs. Note that D computes the opposite of the diagonal entries. The contradiction occurs at the point of the question mark where the entry must be the opposite of itself.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	\dots	$\langle D \rangle$	\dots
M_1	<u>accept</u>	reject	accept	reject		accept	
M_2	accept	<u>accept</u>	accept	accept		accept	
M_3	reject	reject	<u>reject</u>	reject	\dots	reject	\dots
M_4	accept	accept	reject	<u>reject</u>		accept	
\vdots			\vdots		\ddots		
D	reject	reject	accept	accept		<u>?</u>	
\vdots			\vdots				\ddots

FIGURE 4.21

If D is in the figure, a contradiction occurs at “?”

A TURING-UNRECOGNIZABLE LANGUAGE

In the preceding section, we exhibited a language—namely, A_{TM} —that is undecidable. Now we exhibit a language that isn’t even Turing-recognizable. Note that A_{TM} will not suffice for this purpose because we showed that A_{TM} is Turing-recognizable (page 202). The following theorem shows that if both a language and its complement are Turing-recognizable, the language is decidable. Hence for any undecidable language, either it or its complement is not Turing-recognizable. Recall that the complement of a language is the language consisting of all strings that are not in the language. We say that a language is *co-Turing-recognizable* if it is the complement of a Turing-recognizable language.

THEOREM 4.22

A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.

In other words, a language is decidable exactly when both it and its complement are Turing-recognizable.

PROOF We have two directions to prove. First, if A is decidable, we can easily see that both A and its complement \bar{A} are Turing-recognizable. Any decidable language is Turing-recognizable, and the complement of a decidable language also is decidable.

For the other direction, if both A and \overline{A} are Turing-recognizable, we let M_1 be the recognizer for A and M_2 be the recognizer for \overline{A} . The following Turing machine M is a decider for A .

$M =$ “On input w :

1. Run both M_1 and M_2 on input w in parallel.
2. If M_1 accepts, *accept*; if M_2 accepts, *reject*.”

Running the two machines in parallel means that M has two tapes, one for simulating M_1 and the other for simulating M_2 . In this case, M takes turns simulating one step of each machine, which continues until one of them accepts.

Now we show that M decides A . Every string w is either in A or \overline{A} . Therefore, either M_1 or M_2 must accept w . Because M halts whenever M_1 or M_2 accepts, M always halts and so it is a decider. Furthermore, it accepts all strings in A and rejects all strings not in A . So M is a decider for A , and thus A is decidable.

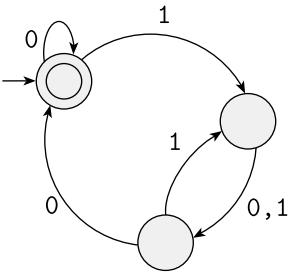
COROLLARY 4.23

$\overline{A_{TM}}$ is not Turing-recognizable.

PROOF We know that A_{TM} is Turing-recognizable. If $\overline{A_{TM}}$ also were Turing-recognizable, A_{TM} would be decidable. Theorem 4.11 tells us that A_{TM} is not decidable, so $\overline{A_{TM}}$ must not be Turing-recognizable.

EXERCISES

^A4.1 Answer all parts for the following DFA M and give reasons for your answers.



- | | |
|---|---|
| a. Is $\langle M, 0100 \rangle \in A_{DFA}$? | d. Is $\langle M, 0100 \rangle \in A_{REX}$? |
| b. Is $\langle M, 011 \rangle \in A_{DFA}$? | e. Is $\langle M \rangle \in E_{DFA}$? |
| c. Is $\langle M \rangle \in A_{DFA}$? | f. Is $\langle M, M \rangle \in EQ_{DFA}$? |

- 4.2 Consider the problem of determining whether a DFA and a regular expression are equivalent. Express this problem as a language and show that it is decidable.
- 4.3 Let $ALL_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \Sigma^*\}$. Show that ALL_{DFA} is decidable.
- 4.4 Let $A\epsilon_{CFG} = \{\langle G \rangle \mid G \text{ is a CFG that generates } \epsilon\}$. Show that $A\epsilon_{CFG}$ is decidable.
- ^A4.5 Let $E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$. Show that $\overline{E_{TM}}$, the complement of E_{TM} , is Turing-recognizable.
- 4.6 Let X be the set $\{1, 2, 3, 4, 5\}$ and Y be the set $\{6, 7, 8, 9, 10\}$. We describe the functions $f: X \rightarrow Y$ and $g: X \rightarrow Y$ in the following tables. Answer each part and give a reason for each negative answer.

n	$f(n)$	n	$g(n)$
1	6	1	10
2	7	2	9
3	6	3	8
4	7	4	7
5	6	5	6

- ^Aa. Is f one-to-one?
- b. Is f onto?
- c. Is f a correspondence?
- ^Ad. Is g one-to-one?
- e. Is g onto?
- f. Is g a correspondence?
- 4.7 Let \mathcal{B} be the set of all infinite sequences over $\{0,1\}$. Show that \mathcal{B} is uncountable using a proof by diagonalization.
- 4.8 Let $T = \{(i, j, k) \mid i, j, k \in \mathcal{N}\}$. Show that T is countable.
- 4.9 Review the way that we define sets to be the same size in Definition 4.12 (page 203). Show that “is the same size” is an equivalence relation.



PROBLEMS

- ^A4.10 Let $INFINITE_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) \text{ is an infinite language}\}$. Show that $INFINITE_{DFA}$ is decidable.
- 4.11 Let $INFINITE_{PDA} = \{\langle M \rangle \mid M \text{ is a PDA and } L(M) \text{ is an infinite language}\}$. Show that $INFINITE_{PDA}$ is decidable.
- ^A4.12 Let $A = \{\langle M \rangle \mid M \text{ is a DFA that doesn't accept any string containing an odd number of 1s}\}$. Show that A is decidable.
- 4.13 Let $A = \{\langle R, S \rangle \mid R \text{ and } S \text{ are regular expressions and } L(R) \subseteq L(S)\}$. Show that A is decidable.
- ^A4.14 Let $\Sigma = \{0,1\}$. Show that the problem of determining whether a CFG generates some string in 1^* is decidable. In other words, show that

$$\{\langle G \rangle \mid G \text{ is a CFG over } \{0,1\} \text{ and } 1^* \cap L(G) \neq \emptyset\}$$

is a decidable language.

- *4.15 Show that the problem of determining whether a CFG generates all strings in 1^* is decidable. In other words, show that $\{\langle G \rangle \mid G \text{ is a CFG over } \{0,1\} \text{ and } 1^* \subseteq L(G)\}$ is a decidable language.
- 4.16 Let $A = \{\langle R \rangle \mid R \text{ is a regular expression describing a language containing at least one string } w \text{ that has } 111 \text{ as a substring (i.e., } w = x111y \text{ for some } x \text{ and } y)\}$. Show that A is decidable.
- 4.17 Prove that EQ_{DFA} is decidable by testing the two DFAs on all strings up to a certain size. Calculate a size that works.
- *4.18 Let C be a language. Prove that C is Turing-recognizable iff a decidable language D exists such that $C = \{x \mid \exists y (\langle x, y \rangle \in D)\}$.
- *4.19 Prove that the class of decidable languages is not closed under homomorphism.
- 4.20 Let A and B be two disjoint languages. Say that language C *separates* A and B if $A \subseteq C$ and $B \subseteq \overline{C}$. Show that any two disjoint co-Turing-recognizable languages are separable by some decidable language.
- 4.21 Let $S = \{\langle M \rangle \mid M \text{ is a DFA that accepts } w^R \text{ whenever it accepts } w\}$. Show that S is decidable.
- 4.22 Let $PREFIX-FREE_{\text{REX}} = \{\langle R \rangle \mid R \text{ is a regular expression and } L(R) \text{ is prefix-free}\}$. Show that $PREFIX-FREE_{\text{REX}}$ is decidable. Why does a similar approach fail to show that $PREFIX-FREE_{\text{CFG}}$ is decidable?
- A*4.23 Say that an NFA is *ambiguous* if it accepts some string along two different computation branches. Let $AMBIG_{\text{NFA}} = \{\langle N \rangle \mid N \text{ is an ambiguous NFA}\}$. Show that $AMBIG_{\text{NFA}}$ is decidable. (Suggestion: One elegant way to solve this problem is to construct a suitable DFA and then run E_{DFA} on it.)
- 4.24 A *useless state* in a pushdown automaton is never entered on any input string. Consider the problem of determining whether a pushdown automaton has any useless states. Formulate this problem as a language and show that it is decidable.
- A*4.25 Let $BAL_{\text{DFA}} = \{\langle M \rangle \mid M \text{ is a DFA that accepts some string containing an equal number of 0s and 1s}\}$. Show that BAL_{DFA} is decidable. (Hint: Theorems about CFLs are helpful here.)
- *4.26 Let $PAL_{\text{DFA}} = \{\langle M \rangle \mid M \text{ is a DFA that accepts some palindrome}\}$. Show that PAL_{DFA} is decidable. (Hint: Theorems about CFLs are helpful here.)
- *4.27 Let $E = \{\langle M \rangle \mid M \text{ is a DFA that accepts some string with more 1s than 0s}\}$. Show that E is decidable. (Hint: Theorems about CFLs are helpful here.)
- 4.28 Let $C = \{\langle G, x \rangle \mid G \text{ is a CFG } x \text{ is a substring of some } y \in L(G)\}$. Show that C is decidable. (Hint: An elegant solution to this problem uses the decider for E_{CFG} .)
- 4.29 Let $C_{\text{CFG}} = \{\langle G, k \rangle \mid G \text{ is a CFG and } L(G) \text{ contains exactly } k \text{ strings where } k \geq 0 \text{ or } k = \infty\}$. Show that C_{CFG} is decidable.
- 4.30 Let A be a Turing-recognizable language consisting of descriptions of Turing machines, $\{\langle M_1 \rangle, \langle M_2 \rangle, \dots\}$, where every M_i is a decider. Prove that some decidable language D is not decided by any decider M_i whose description appears in A . (Hint: You may find it helpful to consider an enumerator for A .)
- 4.31 Say that a variable A in CFL G is *usable* if it appears in some derivation of some string $w \in G$. Given a CFG G and a variable A , consider the problem of testing whether A is usable. Formulate this problem as a language and show that it is decidable.

- [illegible]

4.1 (a) Yes. The DFA M accepts 0100.
 (b) No. M doesn't accept 011.
 (c) No. This input has only a single component and thus is not of the correct form.
 (d) No. The first component is not a regular expression and so the input is not of the correct form.
 (e) No. M 's language isn't empty.
 (f) Yes. M accepts the same language as itself.

- “On input $\langle M \rangle$, where M is a TM:
1. Repeat the following for $i = 1, 2, 3, \dots$.
 2. Run M for i steps on each input, s_1, s_2, \dots, s_i .
 3. If M has accepted any of these, *accept*. Otherwise, continue.”

- 4.10** The following TM I decides $INFINITE_{DFA}$.

1. Let k be the number of states of A .
2. Construct a DFA D that accepts all strings of length k or more.
3. Construct a DFA M such that $L(M) = L(A) \cap L(D)$.
4. Test $L(M) = \emptyset$ using the E_{DFA} decider T from Theorem 4.4.
5. If T accepts, *reject*; if T rejects, *accept*."

Copyright 2012 Cengage Learning. All Rights Reserved. May not be copied, scanned, or duplicated, in whole or in part. Due to electronic rights, some third party content may be suppressed from the eBook and/or eChapter(s). Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. Cengage Learning reserves the right to remove additional content at any time if subsequent rights restrictions require it.

4.12 The following TM decides A .

“On input $\langle M \rangle$:

1. Construct a DFA O that accepts every string containing an odd number of 1s.
2. Construct a DFA B such that $L(B) = L(M) \cap L(O)$.
3. Test whether $L(B) = \emptyset$ using the E_{DFA} decider T from Theorem 4.4.
4. If T accepts, *accept*; if T rejects, *reject*.”

4.14 You showed in Problem 2.18 that if C is a context-free language and R is a regular language, then $C \cap R$ is context free. Therefore, $1^* \cap L(G)$ is context free. The following TM decides the language of this problem.

“On input $\langle G \rangle$:

1. Construct CFG H such that $L(H) = 1^* \cap L(G)$.
2. Test whether $L(H) = \emptyset$ using the E_{CFG} decider R from Theorem 4.8.
3. If R accepts, *reject*; if R rejects, *accept*.”

4.23 The following procedure decides $\text{AMBIG}_{\text{NFA}}$. Given an NFA N , we design a DFA D that simulates N and accepts a string iff it is accepted by N along two different computational branches. Then we use a decider for E_{DFA} to determine whether D accepts any strings.

Our strategy for constructing D is similar to the NFA-to-DFA conversion in the proof of Theorem 1.39. We simulate N by keeping a pebble on each active state. We begin by putting a red pebble on the start state and on each state reachable from the start state along ϵ transitions. We move, add, and remove pebbles in accordance with N 's transitions, preserving the color of the pebbles. Whenever two or more pebbles are moved to the same state, we replace its pebbles with a blue pebble. After reading the input, we accept if a blue pebble is on an accept state of N or if two different accept states of N have red pebbles on them.

The DFA D has a state corresponding to each possible position of pebbles. For each state of N , three possibilities occur: It can contain a red pebble, a blue pebble, or no pebble. Thus, if N has n states, D will have 3^n states. Its start state, accept states, and transition function are defined to carry out the simulation.

4.25 The language of all strings with an equal number of 0s and 1s is a context-free language, generated by the grammar $S \rightarrow 1S0S \mid 0S1S \mid \epsilon$. Let P be the PDA that recognizes this language. Build a TM M for BAL_{DFA} , which operates as follows. On input $\langle B \rangle$, where B is a DFA, use B and P to construct a new PDA R that recognizes the intersection of the languages of B and P . Then test whether R 's language is empty. If its language is empty, *reject*; otherwise, *accept*.

5

REDUCIBILITY

In Chapter 4 we established the Turing machine as our model of a general purpose computer. We presented several examples of problems that are solvable on a Turing machine and gave one example of a problem, A_{TM} , that is computationally unsolvable. In this chapter we examine several additional unsolvable problems. In doing so, we introduce the primary method for proving that problems are computationally unsolvable. It is called *reducibility*.

A *reduction* is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem. Such reducibilities come up often in everyday life, even if we don't usually refer to them in this way.

For example, suppose that you want to find your way around a new city. You know that doing so would be easy if you had a map. Thus, you can reduce the problem of finding your way around the city to the problem of obtaining a map of the city.

Reducibility always involves two problems, which we call A and B . If A reduces to B , we can use a solution to B to solve A . So in our example, A is the problem of finding your way around the city and B is the problem of obtaining a map. Note that reducibility says nothing about solving A or B alone, but only about the solvability of A in the presence of a solution to B .

The following are further examples of reducibilities. The problem of traveling from Boston to Paris reduces to the problem of buying a plane ticket between the two cities. That problem in turn reduces to the problem of earning the money for the ticket. And that problem reduces to the problem of finding a job.

Reducibility also occurs in mathematical problems. For example, the problem of measuring the area of a rectangle reduces to the problem of measuring its length and width. The problem of solving a system of linear equations reduces to the problem of inverting a matrix.

Reducibility plays an important role in classifying problems by decidability, and later in complexity theory as well. When A is reducible to B , solving A cannot be harder than solving B because a solution to B gives a solution to A . In terms of computability theory, if A is reducible to B and B is decidable, A also is decidable. Equivalently, if A is undecidable and reducible to B , B is undecidable. This last version is key to proving that various problems are undecidable.

In short, our method for proving that a problem is undecidable will be to show that some other problem already known to be undecidable reduces to it.

5.1

UNDECIDABLE PROBLEMS FROM LANGUAGE THEORY

We have already established the undecidability of A_{TM} , the problem of determining whether a Turing machine accepts a given input. Let's consider a related problem, HALT_{TM} , the problem of determining whether a Turing machine halts (by accepting or rejecting) on a given input. This problem is widely known as the **halting problem**. We use the undecidability of A_{TM} to prove the undecidability of the halting problem by reducing A_{TM} to HALT_{TM} . Let

$$HALT_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w \}.$$

THEOREM 5.1

$HALT_{TM}$ is undecidable.

PROOF IDEA This proof is by contradiction. We assume that $HALT_{TM}$ is decidable and use that assumption to show that A_{TM} is decidable, contradicting Theorem 4.11. The key idea is to show that A_{TM} is reducible to $HALT_{TM}$.

Let's assume that we have a TM R that decides $HALT_{TM}$. Then we use R to construct S , a TM that decides A_{TM} . To get a feel for the way to construct S , pretend that you are S . Your task is to decide A_{TM} . You are given an input of the form $\langle M, w \rangle$. You must output *accept* if M accepts w , and you must output *reject* if M loops or rejects on w . Try simulating M on w . If it accepts or rejects, do the same. But you may not be able to determine whether M is looping, and in that case your simulation will not terminate. That's bad because you are a decider and thus never permitted to loop. So this idea by itself does not work.

Instead, use the assumption that you have TM R that decides $HALT_{TM}$. With R , you can test whether M halts on w . If R indicates that M doesn't halt on w , reject because $\langle M, w \rangle$ isn't in A_{TM} . However, if R indicates that M does halt on w , you can do the simulation without any danger of looping.

Thus, if TM R exists, we can decide A_{TM} , but we know that A_{TM} is undecidable. By virtue of this contradiction, we can conclude that R does not exist. Therefore, $HALT_{TM}$ is undecidable.

PROOF Let's assume for the purpose of obtaining a contradiction that TM R decides $HALT_{TM}$. We construct TM S to decide A_{TM} , with S operating as follows.

$S =$ "On input $\langle M, w \rangle$, an encoding of a TM M and a string w :

1. Run TM R on input $\langle M, w \rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*."

Clearly, if R decides $HALT_{TM}$, then S decides A_{TM} . Because A_{TM} is undecidable, $HALT_{TM}$ also must be undecidable.

Theorem 5.1 illustrates our strategy for proving that a problem is undecidable. This strategy is common to most proofs of undecidability, except for the undecidability of A_{TM} itself, which is proved directly via the diagonalization method.

We now present several other theorems and their proofs as further examples of the reducibility method for proving undecidability. Let

$$E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}.$$

THEOREM 5.2

E_{TM} is undecidable.

PROOF IDEA We follow the pattern adopted in Theorem 5.1. We assume that E_{TM} is decidable and then show that A_{TM} is decidable—a contradiction. Let R be a TM that decides E_{TM} . We use R to construct TM S that decides A_{TM} . How will S work when it receives input $\langle M, w \rangle$?

One idea is for S to run R on input $\langle M \rangle$ and see whether it accepts. If it does, we know that $L(M)$ is empty and therefore that M does not accept w . But if R rejects $\langle M \rangle$, all we know is that $L(M)$ is not empty and therefore that M accepts some string—but we still do not know whether M accepts the particular string w . So we need to use a different idea.

Instead of running R on $\langle M \rangle$, we run R on a modification of $\langle M \rangle$. We modify $\langle M \rangle$ to guarantee that M rejects all strings except w , but on input w it works as usual. Then we use R to determine whether the modified machine recognizes the empty language. The only string the machine can now accept is w , so its language will be nonempty iff it accepts w . If R accepts when it is fed a description of the modified machine, we know that the modified machine doesn't accept anything and that M doesn't accept w .

PROOF Let's write the modified machine described in the proof idea using our standard notation. We call it M_1 .

$M_1 =$ "On input x :

1. If $x \neq w$, *reject*.
2. If $x = w$, run M on input w and *accept* if M does."

This machine has the string w as part of its description. It conducts the test of whether $x = w$ in the obvious way, by scanning the input and comparing it character by character with w to determine whether they are the same.

Putting all this together, we assume that TM R decides E_{TM} and construct TM S that decides A_{TM} as follows.

$S =$ "On input $\langle M, w \rangle$, an encoding of a TM M and a string w :

1. Use the description of M and w to construct the TM M_1 just described.
2. Run R on input $\langle M_1 \rangle$.
3. If R accepts, *reject*; if R rejects, *accept*."

Note that S must actually be able to compute a description of M_1 from a description of M and w . It is able to do so because it only needs to add extra states to M that perform the $x = w$ test.

If R were a decider for E_{TM} , S would be a decider for A_{TM} . A decider for A_{TM} cannot exist, so we know that E_{TM} must be undecidable.

Another interesting computational problem regarding Turing machines concerns determining whether a given Turing machine recognizes a language that also can be recognized by a simpler computational model. For example, we let $\text{REGULAR}_{\text{TM}}$ be the problem of determining whether a given Turing machine has an equivalent finite automaton. This problem is the same as determining whether the Turing machine recognizes a regular language. Let

$$\text{REGULAR}_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}.$$

THEOREM 5.3

$REGULAR_{TM}$ is undecidable.

PROOF IDEA As usual for undecidability theorems, this proof is by reduction from A_{TM} . We assume that $REGULAR_{TM}$ is decidable by a TM R and use this assumption to construct a TM S that decides A_{TM} . Less obvious now is how to use R 's ability to assist S in its task. Nonetheless, we can do so.

The idea is for S to take its input $\langle M, w \rangle$ and modify M so that the resulting TM recognizes a regular language if and only if M accepts w . We call the modified machine M_2 . We design M_2 to recognize the nonregular language $\{0^n 1^n \mid n \geq 0\}$ if M does not accept w , and to recognize the regular language Σ^* if M accepts w . We must specify how S can construct such an M_2 from M and w . Here, M_2 works by automatically accepting all strings in $\{0^n 1^n \mid n \geq 0\}$. In addition, if M accepts w , M_2 accepts all other strings.

Note that the TM M_2 is *not* constructed for the purposes of actually running it on some input—a common confusion. We construct M_2 only for the purpose of feeding its description into the decider for $REGULAR_{TM}$ that we have assumed to exist. Once this decider returns its answer, we can use it to obtain the answer to whether M accepts w . Thus, we can decide A_{TM} , a contradiction.

PROOF We let R be a TM that decides $REGULAR_{TM}$ and construct TM S to decide A_{TM} . Then S works in the following manner.

$S =$ “On input $\langle M, w \rangle$, where M is a TM and w is a string:

1. Construct the following TM M_2 .
 $M_2 =$ “On input x :
 1. If x has the form $0^n 1^n$, *accept*.
 2. If x does not have this form, run M on input w and *accept* if M accepts w .”
2. Run R on input $\langle M_2 \rangle$.
3. If R accepts, *accept*; if R rejects, *reject*.”

Similarly, the problems of testing whether the language of a Turing machine is a context-free language, a decidable language, or even a finite language can be shown to be undecidable with similar proofs. In fact, a general result, called Rice's theorem, states that determining *any property* of the languages recognized by Turing machines is undecidable. We give Rice's theorem in Problem 5.28.

So far, our strategy for proving languages undecidable involves a reduction from A_{TM} . Sometimes reducing from some other undecidable language, such as E_{TM} , is more convenient when we are showing that certain languages are undecidable. Theorem 5.4 shows that testing the equivalence of two Turing

machines is an undecidable problem. We could prove it by a reduction from A_{TM} , but we use this opportunity to give an example of an undecidability proof by reduction from E_{TM} . Let

$$EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}.$$

THEOREM 5.4

EQ_{TM} is undecidable.

PROOF IDEA Show that if EQ_{TM} were decidable, E_{TM} also would be decidable by giving a reduction from E_{TM} to EQ_{TM} . The idea is simple. E_{TM} is the problem of determining whether the language of a TM is empty. EQ_{TM} is the problem of determining whether the languages of two TMs are the same. If one of these languages happens to be \emptyset , we end up with the problem of determining whether the language of the other machine is empty—that is, the E_{TM} problem. So in a sense, the E_{TM} problem is a special case of the EQ_{TM} problem wherein one of the machines is fixed to recognize the empty language. This idea makes giving the reduction easy.

PROOF We let TM R decide EQ_{TM} and construct TM S to decide E_{TM} as follows.

$S =$ “On input $\langle M \rangle$, where M is a TM:

1. Run R on input $\langle M, M_1 \rangle$, where M_1 is a TM that rejects all inputs.
2. If R accepts, *accept*; if R rejects, *reject*.”

If R decides EQ_{TM} , S decides E_{TM} . But E_{TM} is undecidable by Theorem 5.2, so EQ_{TM} also must be undecidable.

REDUCTIONS VIA COMPUTATION HISTORIES

The computation history method is an important technique for proving that A_{TM} is reducible to certain languages. This method is often useful when the problem to be shown undecidable involves testing for the existence of something. For example, this method is used to show the undecidability of Hilbert’s tenth problem, testing for the existence of integral roots in a polynomial.

The computation history for a Turing machine on an input is simply the sequence of configurations that the machine goes through as it processes the input. It is a complete record of the computation of this machine.

DEFINITION 5.5

Let M be a Turing machine and w an input string. An **accepting computation history** for M on w is a sequence of configurations, C_1, C_2, \dots, C_l , where C_1 is the start configuration of M on w , C_l is an accepting configuration of M , and each C_i legally follows from C_{i-1} according to the rules of M . A **rejecting computation history** for M on w is defined similarly, except that C_l is a rejecting configuration.

Computation histories are finite sequences. If M doesn't halt on w , no accepting or rejecting computation history exists for M on w . Deterministic machines have at most one computation history on any given input. Nondeterministic machines may have many computation histories on a single input, corresponding to the various computation branches. For now, we continue to focus on deterministic machines. Our first undecidability proof using the computation history method concerns a type of machine called a linear bounded automaton.

DEFINITION 5.6

A **linear bounded automaton** is a restricted type of Turing machine wherein the tape head isn't permitted to move off the portion of the tape containing the input. If the machine tries to move its head off either end of the input, the head stays where it is—in the same way that the head will not move off the left-hand end of an ordinary Turing machine's tape.

A linear bounded automaton is a Turing machine with a limited amount of memory, as shown schematically in the following figure. It can only solve problems requiring memory that can fit within the tape used for the input. Using a tape alphabet larger than the input alphabet allows the available memory to be increased up to a constant factor. Hence we say that for an input of length n , the amount of memory available is linear in n —thus the name of this model.

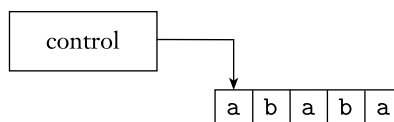


FIGURE 5.7
Schematic of a linear bounded automaton

Despite their memory constraint, linear bounded automata (LBAs) are quite powerful. For example, the deciders for A_{DFA} , A_{CFG} , E_{DFA} , and E_{CFG} all are LBAs. Every CFL can be decided by an LBA. In fact, coming up with a decidable language that can't be decided by an LBA takes some work. We develop the techniques to do so in Chapter 9.

Here, A_{LBA} is the problem of determining whether an LBA accepts its input. Even though A_{LBA} is the same as the undecidable problem A_{TM} where the Turing machine is restricted to be an LBA, we can show that A_{LBA} is decidable. Let

$$A_{LBA} = \{ \langle M, w \rangle \mid M \text{ is an LBA that accepts string } w \}.$$

Before proving the decidability of A_{LBA} , we find the following lemma useful. It says that an LBA can have only a limited number of configurations when a string of length n is the input.

LEMMA 5.8

Let M be an LBA with q states and g symbols in the tape alphabet. There are exactly qng^n distinct configurations of M for a tape of length n .

PROOF Recall that a configuration of M is like a snapshot in the middle of its computation. A configuration consists of the state of the control, position of the head, and contents of the tape. Here, M has q states. The length of its tape is n , so the head can be in one of n positions, and g^n possible strings of tape symbols appear on the tape. The product of these three quantities is the total number of different configurations of M with a tape of length n .

THEOREM 5.9

A_{LBA} is decidable.

PROOF IDEA In order to decide whether LBA M accepts input w , we simulate M on w . During the course of the simulation, if M halts and accepts or rejects, we accept or reject accordingly. The difficulty occurs if M loops on w . We need to be able to detect looping so that we can halt and reject.

The idea for detecting when M is looping is that as M computes on w , it goes from configuration to configuration. If M ever repeats a configuration, it would go on to repeat this configuration over and over again and thus be in a loop. Because M is an LBA, the amount of tape available to it is limited. By Lemma 5.8, M can be in only a limited number of configurations on this amount of tape. Therefore, only a limited amount of time is available to M before it will enter some configuration that it has previously entered. Detecting that M is looping is possible by simulating M for the number of steps given by Lemma 5.8. If M has not halted by then, it must be looping.

PROOF The algorithm that decides A_{LBA} is as follows.

$L =$ “On input $\langle M, w \rangle$, where M is an LBA and w is a string:

1. Simulate M on w for qng^n steps or until it halts.
2. If M has halted, *accept* if it has accepted and *reject* if it has rejected. If it has not halted, *reject*.”

If M on w has not halted within qng^n steps, it must be repeating a configuration according to Lemma 5.8 and therefore looping. That is why our algorithm rejects in this instance.

Theorem 5.9 shows that LBAs and TMs differ in one essential way: For LBAs the acceptance problem is decidable, but for TMs it isn't. However, certain other problems involving LBAs remain undecidable. One is the emptiness problem $E_{\text{LBA}} = \{\langle M \rangle \mid M \text{ is an LBA where } L(M) = \emptyset\}$. To prove that E_{LBA} is undecidable, we give a reduction that uses the computation history method.

THEOREM 5.10

E_{LBA} is undecidable.

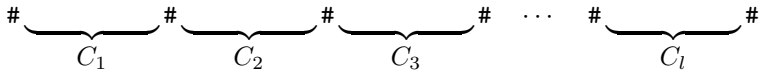
PROOF IDEA This proof is by reduction from A_{TM} . We show that if E_{LBA} were decidable, A_{TM} would also be. Suppose that E_{LBA} is decidable. How can we use this supposition to decide A_{TM} ?

For a TM M and an input w , we can determine whether M accepts w by constructing a certain LBA B and then testing whether $L(B)$ is empty. The language that B recognizes comprises all accepting computation histories for M on w . If M accepts w , this language contains one string and so is nonempty. If M does not accept w , this language is empty. If we can determine whether B 's language is empty, clearly we can determine whether M accepts w .

Now we describe how to construct B from M and w . Note that we need to show more than the mere existence of B . We have to show how a Turing machine can obtain a description of B , given descriptions of M and w .

As in the previous reductions we've given for proving undecidability, we construct B only to feed its description into the presumed E_{LBA} decider, but not to run B on some input.

We construct B to accept its input x if x is an accepting computation history for M on w . Recall that an accepting computation history is the sequence of configurations, C_1, C_2, \dots, C_l that M goes through as it accepts some string w . For the purposes of this proof, we assume that the accepting computation history is presented as a single string with the configurations separated from each other by the # symbol, as shown in Figure 5.11.

**FIGURE 5.11**A possible input to B

The LBA B works as follows. When it receives an input x , B is supposed to accept if x is an accepting computation history for M on w . First, B breaks up x according to the delimiters into strings C_1, C_2, \dots, C_l . Then B determines whether the C_i 's satisfy the three conditions of an accepting computation history.

1. C_1 is the start configuration for M on w .
2. Each C_{i+1} legally follows from C_i .
3. C_l is an accepting configuration for M .

The start configuration C_1 for M on w is the string $q_0 w_1 w_2 \cdots w_n$, where q_0 is the start state for M on w . Here, B has this string directly built in, so it is able to check the first condition. An accepting configuration is one that contains the q_{accept} state, so B can check the third condition by scanning C_l for q_{accept} . The second condition is the hardest to check. For each pair of adjacent configurations, B checks on whether C_{i+1} legally follows from C_i . This step involves verifying that C_i and C_{i+1} are identical except for the positions under and adjacent to the head in C_i . These positions must be updated according to the transition function of M . Then B verifies that the updating was done properly by zig-zagging between corresponding positions of C_i and C_{i+1} . To keep track of the current positions while zig-zagging, B marks the current position with dots on the tape. Finally, if conditions 1, 2, and 3 are satisfied, B accepts its input.

By inverting the decider's answer, we obtain the answer to whether M accepts w . Thus we can decide A_{TM} , a contradiction.

PROOF Now we are ready to state the reduction of A_{TM} to E_{LBA} . Suppose that TM R decides E_{LBA} . Construct TM S to decide A_{TM} as follows.

S = "On input $\langle M, w \rangle$, where M is a TM and w is a string:

1. Construct LBA B from M and w as described in the proof idea.
2. Run R on input $\langle B \rangle$.
3. If R rejects, *accept*; if R accepts, *reject*."

If R accepts $\langle B \rangle$, then $L(B) = \emptyset$. Thus, M has no accepting computation history on w and M doesn't accept w . Consequently, S rejects $\langle M, w \rangle$. Similarly, if R rejects $\langle B \rangle$, the language of B is nonempty. The only string that B can accept is an accepting computation history for M on w . Thus, M must accept w . Consequently, S accepts $\langle M, w \rangle$. Figure 5.12 illustrates LBA B .

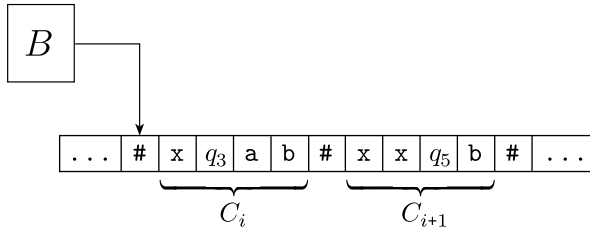


FIGURE 5.12
LBA B checking a TM computation history

We can also use the technique of reduction via computation histories to establish the undecidability of certain problems related to context-free grammars and pushdown automata. Recall that in Theorem 4.8 we presented an algorithm to decide whether a context-free grammar generates any strings—that is, whether $L(G) = \emptyset$. Now we show that a related problem is undecidable. It is the problem of determining whether a context-free grammar generates all possible strings. Proving that this problem is undecidable is the main step in showing that the equivalence problem for context-free grammars is undecidable. Let

$$ALL_{CFG} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \Sigma^* \}.$$

THEOREM 5.13

ALL_{CFG} is undecidable.

PROOF This proof is by contradiction. To get the contradiction, we assume that ALL_{CFG} is decidable and use this assumption to show that A_{TM} is decidable. This proof is similar to that of Theorem 5.10 but with a small extra twist: It is a reduction from A_{TM} via computation histories, but we modify the representation of the computation histories slightly for a technical reason that we will explain later.

We now describe how to use a decision procedure for ALL_{CFG} to decide A_{TM} . For a TM M and an input w , we construct a CFG G that generates all strings if and only if M does not accept w . So if M does accept w , G does *not* generate some particular string. This string is—guess what—the accepting computation history for M on w . That is, G is designed to generate all strings that are *not* accepting computation histories for M on w .

To make the CFG G generate all strings that fail to be an accepting computation history for M on w , we utilize the following strategy. A string may fail to be an accepting computation history for several reasons. An accepting computation history for M on w appears as $\#C_1\#C_2\#\cdots\#C_i\#$, where C_i is the configuration of M on the i th step of the computation on w . Then, G generates all strings

1. that *do not* start with C_1 ,
2. that *do not* end with an accepting configuration, or
3. in which some C_i *does not* properly yield C_{i+1} under the rules of M .

If M does not accept w , no accepting computation history exists, so *all* strings fail in one way or another. Therefore, G would generate all strings, as desired.

Now we get down to the actual construction of G . Instead of constructing G , we construct a PDA D . We know that we can use the construction given in Theorem 2.20 (page 117) to convert D to a CFG. We do so because, for our purposes, designing a PDA is easier than designing a CFG. In this instance, D will start by nondeterministically branching to guess which of the preceding three conditions to check. One branch checks on whether the beginning of the input string is C_1 and accepts if it isn't. Another branch checks on whether the input string ends with a configuration containing the accept state, q_{accept} , and accepts if it isn't.

The third branch is supposed to accept if some C_i does not properly yield C_{i+1} . It works by scanning the input until it nondeterministically decides that it has come to C_i . Next, it pushes C_i onto the stack until it comes to the end as marked by the $\#$ symbol. Then D pops the stack to compare with C_{i+1} . They are supposed to match except around the head position, where the difference is dictated by the transition function of M . Finally, D accepts if it discovers a mismatch or an improper update.

The problem with this idea is that when D pops C_i off the stack, it is in reverse order and not suitable for comparison with C_{i+1} . At this point, the twist in the proof appears: We write the accepting computation history differently. Every other configuration appears in reverse order. The odd positions remain written in the forward order, but the even positions are written backward. Thus, an accepting computation history would appear as shown in the following figure.

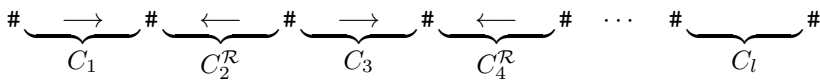


FIGURE 5.14

Every other configuration written in reverse order

In this modified form, the PDA is able to push a configuration so that when it is popped, the order is suitable for comparison with the next one. We design D to accept any string that is not an accepting computation history in the modified form.

In Exercise 5.1 you can use Theorem 5.13 to show that EQ_{CFG} is undecidable.

5.2

A SIMPLE UNDECIDABLE PROBLEM

In this section we show that the phenomenon of undecidability is not confined to problems concerning automata. We give an example of an undecidable problem concerning simple manipulations of strings. It is called the *Post Correspondence Problem*, or *PCP*.

We can describe this problem easily as a type of puzzle. We begin with a collection of dominos, each containing two strings, one on each side. An individual domino looks like

$$\begin{bmatrix} a \\ ab \end{bmatrix}$$

and a collection of dominos looks like

$$\left\{ \begin{bmatrix} b \\ ca \end{bmatrix}, \begin{bmatrix} a \\ ab \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} abc \\ c \end{bmatrix} \right\}.$$

The task is to make a list of these dominos (repetitions permitted) so that the string we get by reading off the symbols on the top is the same as the string of symbols on the bottom. This list is called a *match*. For example, the following list is a match for this puzzle.

$$\begin{bmatrix} a \\ ab \end{bmatrix} \begin{bmatrix} b \\ ca \end{bmatrix} \begin{bmatrix} ca \\ a \end{bmatrix} \begin{bmatrix} a \\ ab \end{bmatrix} \begin{bmatrix} abc \\ c \end{bmatrix}$$

Reading off the top string we get $abcaaaabc$, which is the same as reading off the bottom. We can also depict this match by deforming the dominos so that the corresponding symbols from top and bottom line up.

$$\begin{array}{ccccccccc} | & a & | & b & | & c & | & a & | & a & | & a & | & b & | & c & | \\ & & \searrow & & \searrow & & \searrow & & \searrow & & \searrow & & \searrow & & \searrow & \\ & a & | & b & | & c & | & a & | & a & | & a & | & b & | & c & | \end{array}$$

For some collections of dominos, finding a match may not be possible. For example, the collection

$$\left\{ \begin{bmatrix} abc \\ ab \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} acc \\ ba \end{bmatrix} \right\}$$

cannot contain a match because every top string is longer than the corresponding bottom string.

The Post Correspondence Problem is to determine whether a collection of dominos has a match. This problem is unsolvable by algorithms.

Before getting to the formal statement of this theorem and its proof, let's state the problem precisely and then express it as a language. An instance of the PCP is a collection P of dominos

$$P = \left\{ \left[\frac{t_1}{b_1} \right], \left[\frac{t_2}{b_2} \right], \dots, \left[\frac{t_k}{b_k} \right] \right\},$$

and a match is a sequence i_1, i_2, \dots, i_l , where $t_{i_1} t_{i_2} \dots t_{i_l} = b_{i_1} b_{i_2} \dots b_{i_l}$. The problem is to determine whether P has a match. Let

$$PCP = \{ \langle P \rangle \mid P \text{ is an instance of the Post Correspondence Problem with a match} \}.$$

THEOREM 5.15

PCP is undecidable.

PROOF IDEA Conceptually this proof is simple, though it involves many details. The main technique is reduction from A_{TM} via accepting computation histories. We show that from any TM M and input w , we can construct an instance P where a match is an accepting computation history for M on w . If we could determine whether the instance has a match, we would be able to determine whether M accepts w .

How can we construct P so that a match is an accepting computation history for M on w ? We choose the dominos in P so that making a match forces a simulation of M to occur. In the match, each domino links a position or positions in one configuration with the corresponding one(s) in the next configuration.

Before getting to the construction, we handle three small technical points. (Don't worry about them too much on your initial reading through this construction.) First, for convenience in constructing P , we assume that M on w never attempts to move its head off the left-hand end of the tape. That requires first altering M to prevent this behavior. Second, if $w = \varepsilon$, we use the string \sqcup in place of w in the construction. Third, we modify the PCP to require that a match starts with the first domino,

$$\left[\frac{t_1}{b_1} \right].$$

Later we show how to eliminate this requirement. We call this problem the Modified Post Correspondence Problem (MPCP). Let

$$MPCP = \{ \langle P \rangle \mid P \text{ is an instance of the Post Correspondence Problem with a match that starts with the first domino} \}.$$

Now let's move into the details of the proof and design P to simulate M on w .

PROOF We let TM R decide the PCP and construct S deciding A_{TM} . Let

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

where Q , Σ , Γ , and δ are the state set, input alphabet, tape alphabet, and transition function of M , respectively.

In this case, S constructs an instance of the PCP P that has a match iff M accepts w . To do that, S first constructs an instance P' of the MPCP. We describe the construction in seven parts, each of which accomplishes a particular aspect of simulating M on w . To explain what we are doing, we interleave the construction with an example of the construction in action.

Part 1. The construction begins in the following manner.

Put $\left[\frac{\#}{\#q_0w_1w_2 \cdots w_n\#} \right]$ into P' as the first domino $\left[\frac{t_1}{b_1} \right]$.

Because P' is an instance of the MPCP, the match must begin with this domino. Thus, the bottom string begins correctly with $C_1 = q_0w_1w_2 \cdots w_n$, the first configuration in the accepting computation history for M on w , as shown in the following figure.

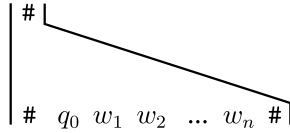


FIGURE 5.16

Beginning of the MPCP match

In this depiction of the partial match achieved so far, the bottom string consists of $\#q_0w_1w_2 \cdots w_n\#$ and the top string consists only of $\#$. To get a match, we need to extend the top string to match the bottom string. We provide additional dominos to allow this extension. The additional dominos cause M 's next configuration to appear at the extension of the bottom string by forcing a single-step simulation of M .

In parts 2, 3, and 4, we add to P' dominos that perform the main part of the simulation. Part 2 handles head motions to the right, part 3 handles head motions to the left, and part 4 handles the tape cells not adjacent to the head.

Part 2. For every $a, b \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{\text{reject}}$,

if $\delta(q, a) = (r, b, R)$, put $\left[\frac{qa}{br} \right]$ into P' .

Part 3. For every $a, b, c \in \Gamma$ and every $q, r \in Q$ where $q \neq q_{\text{reject}}$,

if $\delta(q, a) = (r, b, L)$, put $\left[\frac{cqa}{rcb} \right]$ into P' .

Part 4. For every $a \in \Gamma$,

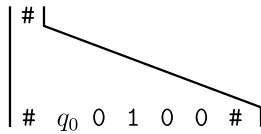
put $\begin{bmatrix} a \\ a \end{bmatrix}$ into P' .

Now we make up a hypothetical example to illustrate what we have built so far. Let $\Gamma = \{0, 1, 2, \sqcup\}$. Say that w is the string 0100 and that the start state of M is q_0 . In state q_0 , upon reading a 0, let's say that the transition function dictates that M enters state q_7 , writes a 2 on the tape, and moves its head to the right. In other words, $\delta(q_0, 0) = (q_7, 2, R)$.

Part 1 places the domino

$$\begin{bmatrix} \# \\ \#q_00100\# \end{bmatrix} = \begin{bmatrix} t_1 \\ b_1 \end{bmatrix}$$

in P' , and the match begins



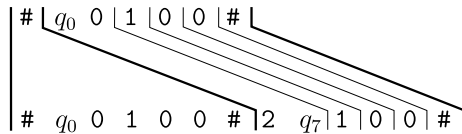
In addition, part 2 places the domino

$$\begin{bmatrix} q_00 \\ 2q_7 \end{bmatrix}$$

as $\delta(q_0, 0) = (q_7, 2, R)$ and part 4 places the dominos

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \text{ and } \begin{bmatrix} \sqcup \\ \sqcup \end{bmatrix}$$

in P' , as 0, 1, 2, and \sqcup are the members of Γ . Together with part 5, that allows us to extend the match to



Thus, the dominos of parts 2, 3, and 4 let us extend the match by adding the second configuration after the first one. We want this process to continue, adding the third configuration, then the fourth, and so on. For it to happen, we need to add one more domino for copying the # symbol.

Part 5.

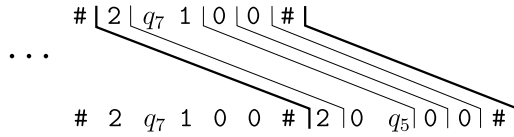
Put $\left[\begin{smallmatrix} \# \\ \# \end{smallmatrix} \right]$ and $\left[\begin{smallmatrix} \# \\ \sqcup \# \end{smallmatrix} \right]$ into P' .

The first of these dominos allows us to copy the $\#$ symbol that marks the separation of the configurations. In addition to that, the second domino allows us to add a blank symbol \sqcup at the end of the configuration to simulate the infinitely many blanks to the right that are suppressed when we write the configuration.

Continuing with the example, let's say that in state q_7 , upon reading a 1, M goes to state q_5 , writes a 0, and moves the head to the right. That is, $\delta(q_7, 1) = (q_5, 0, R)$. Then we have the domino

$$\left[\begin{smallmatrix} q_7 1 \\ 0 q_5 \end{smallmatrix} \right] \text{ in } P'.$$

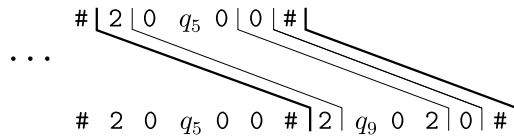
So the latest partial match extends to



Then, suppose that in state q_5 , upon reading a 0, M goes to state q_9 , writes a 2, and moves its head to the left. So $\delta(q_5, 0) = (q_9, 2, L)$. Then we have the dominos

$$\left[\begin{smallmatrix} 0 q_5 0 \\ q_9 0 2 \end{smallmatrix} \right], \left[\begin{smallmatrix} 1 q_5 0 \\ q_9 1 2 \end{smallmatrix} \right], \left[\begin{smallmatrix} 2 q_5 0 \\ q_9 2 2 \end{smallmatrix} \right], \text{ and } \left[\begin{smallmatrix} \sqcup q_5 0 \\ q_9 \sqcup 2 \end{smallmatrix} \right].$$

The first one is relevant because the symbol to the left of the head is a 0. The preceding partial match extends to

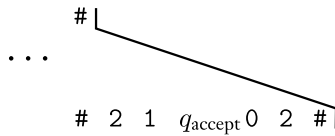


Note that as we construct a match, we are forced to simulate M on input w . This process continues until M reaches a halting state. If the accept state occurs, we want to let the top of the partial match “catch up” with the bottom so that the match is complete. We can arrange for that to happen by adding additional dominos.

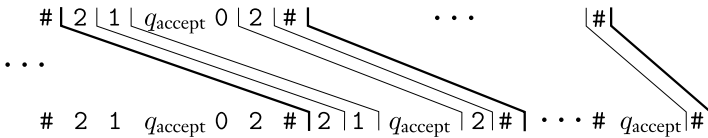
Part 6. For every $a \in \Gamma$,

put $\left[\frac{a q_{\text{accept}}}{q_{\text{accept}}} \right]$ and $\left[\frac{q_{\text{accept}} a}{q_{\text{accept}}} \right]$ into P' .

This step has the effect of adding “pseudo-steps” of the Turing machine after it has halted, where the head “eats” adjacent symbols until none are left. Continuing with the example, if the partial match up to the point when the machine halts in the accept state is



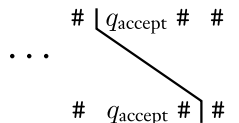
The dominoes we have just added allow the match to continue:



Part 7. Finally, we add the domino

$$\left[\frac{q_{\text{accept}} \# \#}{\#} \right]$$

and complete the match:



That concludes the construction of P' . Recall that P' is an instance of the MPCP whereby the match simulates the computation of M on w . To finish the proof, we recall that the MPCP differs from the PCP in that the match is required to start with the first domino in the list. If we view P' as an instance of

the PCP instead of the MPCP, it obviously has a match, regardless of whether M accepts w . Can you find it? (Hint: It is very short.)

We now show how to convert P' to P , an instance of the PCP that still simulates M on w . We do so with a somewhat technical trick. The idea is to take the requirement that the match starts with the first domino and build it directly into the problem instance itself so that it becomes enforced automatically. After that, the requirement isn't needed. We introduce some notation to implement this idea.

Let $u = u_1 u_2 \cdots u_n$ be any string of length n . Define $\star u$, $u\star$, and $\star u\star$ to be the three strings

$$\begin{aligned}\star u &= \star u_1 \star u_2 \star u_3 \star \cdots \star u_n \\ u\star &= u_1 \star u_2 \star u_3 \star \cdots \star u_n \star \\ \star u\star &= \star u_1 \star u_2 \star u_3 \star \cdots \star u_n \star.\end{aligned}$$

Here, $\star u$ adds the symbol \star before every character in u , $u\star$ adds one after each character in u , and $\star u\star$ adds one both before and after each character in u .

To convert P' to P , an instance of the PCP, we do the following. If P' were the collection

$$\left\{ \left[\frac{t_1}{b_1} \right], \left[\frac{t_2}{b_2} \right], \left[\frac{t_3}{b_3} \right], \dots, \left[\frac{t_k}{b_k} \right] \right\},$$

we let P be the collection

$$\left\{ \left[\frac{\star t_1}{\star b_1 \star} \right], \left[\frac{\star t_1}{b_1 \star} \right], \left[\frac{\star t_2}{b_2 \star} \right], \left[\frac{\star t_3}{b_3 \star} \right], \dots, \left[\frac{\star t_k}{b_k \star} \right], \left[\frac{\star \diamond}{\diamond} \right] \right\}.$$

Considering P as an instance of the PCP, we see that the only domino that could possibly start a match is the first one,

$$\left[\frac{\star t_1}{\star b_1 \star} \right],$$

because it is the only one where both the top and the bottom start with the same symbol—namely, \star . Besides forcing the match to start with the first domino, the presence of the \star s doesn't affect possible matches because they simply interleave with the original symbols. The original symbols now occur in the even positions of the match. The domino

$$\left[\frac{\star \diamond}{\diamond} \right]$$

is there to allow the top to add the extra \star at the end of the match.

5.3

MAPPING REDUCIBILITY

We have shown how to use the reducibility technique to prove that various problems are undecidable. In this section we formalize the notion of reducibility. Doing so allows us to use reducibility in more refined ways, such as for proving that certain languages are not Turing-recognizable and for applications in complexity theory.

The notion of reducing one problem to another may be defined formally in one of several ways. The choice of which one to use depends on the application. Our choice is a simple type of reducibility called *mapping reducibility*.¹

Roughly speaking, being able to reduce problem A to problem B by using a mapping reducibility means that a computable function exists that converts instances of problem A to instances of problem B . If we have such a conversion function, called a *reduction*, we can solve A with a solver for B . The reason is that any instance of A can be solved by first using the reduction to convert it to an instance of B and then applying the solver for B . A precise definition of mapping reducibility follows shortly.

COMPUTABLE FUNCTIONS

A Turing machine computes a function by starting with the input to the function on the tape and halting with the output of the function on the tape.

DEFINITION 5.17

A function $f: \Sigma^* \rightarrow \Sigma^*$ is a *computable function* if some Turing machine M , on every input w , halts with just $f(w)$ on its tape.

EXAMPLE 5.18

All usual, arithmetic operations on integers are computable functions. For example, we can make a machine that takes input $\langle m, n \rangle$ and returns $m + n$, the sum of m and n . We don't give any details here, leaving them as exercises. ■

EXAMPLE 5.19

Computable functions may be transformations of machine descriptions. For example, one computable function f takes input w and returns the description of a Turing machine $\langle M' \rangle$ if $w = \langle M \rangle$ is an encoding of a Turing machine M .

¹It is called *many-one reducibility* in some other textbooks.

The machine M' is a machine that recognizes the same language as M , but never attempts to move its head off the left-hand end of its tape. The function f accomplishes this task by adding several states to the description of M . The function returns ε if w is not a legal encoding of a Turing machine. ■

FORMAL DEFINITION OF MAPPING REDUCIBILITY

Now we define mapping reducibility. As usual, we represent computational problems by languages.

DEFINITION 5.20

Language A is **mapping reducible** to language B , written $A \leq_m B$, if there is a computable function $f: \Sigma^* \rightarrow \Sigma^*$, where for every w ,

$$w \in A \iff f(w) \in B.$$

The function f is called the **reduction** from A to B .

The following figure illustrates mapping reducibility.

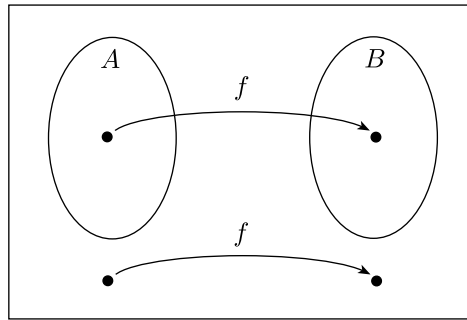


FIGURE 5.21
Function f reducing A to B

A mapping reduction of A to B provides a way to convert questions about membership testing in A to membership testing in B . To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$. The term *mapping reduction* comes from the function or mapping that provides the means of doing the reduction.

If one problem is mapping reducible to a second, previously solved problem, we can thereby obtain a solution to the original problem. We capture this idea in Theorem 5.22.

THEOREM 5.22

If $A \leq_m B$ and B is decidable, then A is decidable.

PROOF We let M be the decider for B and f be the reduction from A to B . We describe a decider N for A as follows.

$N =$ “On input w :

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs.”

Clearly, if $w \in A$, then $f(w) \in B$ because f is a reduction from A to B . Thus, M accepts $f(w)$ whenever $w \in A$. Therefore, N works as desired.

.....

The following corollary of Theorem 5.22 has been our main tool for proving undecidability.

COROLLARY 5.23

If $A \leq_m B$ and A is undecidable, then B is undecidable.

Now we revisit some of our earlier proofs that used the reducibility method to get examples of mapping reducibilities.

EXAMPLE 5.24

In Theorem 5.1 we used a reduction from A_{TM} to prove that $HALT_{TM}$ is undecidable. This reduction showed how a decider for $HALT_{TM}$ could be used to give a decider for A_{TM} . We can demonstrate a mapping reducibility from A_{TM} to $HALT_{TM}$ as follows. To do so, we must present a computable function f that takes input of the form $\langle M, w \rangle$ and returns output of the form $\langle M', w' \rangle$, where

$$\langle M, w \rangle \in A_{TM} \text{ if and only if } \langle M', w' \rangle \in HALT_{TM}.$$

The following machine F computes a reduction f .

$F =$ “On input $\langle M, w \rangle$:

1. Construct the following machine M' .
 $M' =$ “On input x :
 1. Run M on x .
 2. If M accepts, *accept*.
 3. If M rejects, enter a loop.”
2. Output $\langle M', w \rangle$.”

A minor issue arises here concerning improperly formed input strings. If TM F determines that its input is not of the correct form as specified in the input line “On input $\langle M, w \rangle$,” and hence that the input is not in A_{TM} , the TM outputs a

string not in $HALT_{TM}$. Any string not in $HALT_{TM}$ will do. In general, when we describe a Turing machine that computes a reduction from A to B , improperly formed inputs are assumed to map to strings outside of B . ■

EXAMPLE 5.25

The proof of the undecidability of the Post Correspondence Problem in Theorem 5.15 contains two mapping reductions. First, it shows that $A_{TM} \leq_m MPCP$ and then it shows that $MPCP \leq_m PCP$. In both cases, we can easily obtain the actual reduction function and show that it is a mapping reduction. As Exercise 5.6 shows, mapping reducibility is transitive, so these two reductions together imply that $A_{TM} \leq_m PCP$. ■

EXAMPLE 5.26

A mapping reduction from E_{TM} to EQ_{TM} lies in the proof of Theorem 5.4. In this case, the reduction f maps the input $\langle M \rangle$ to the output $\langle M, M_1 \rangle$, where M_1 is the machine that rejects all inputs. ■

EXAMPLE 5.27

The proof of Theorem 5.2 showing that E_{TM} is undecidable illustrates the difference between the formal notion of mapping reducibility that we have defined in this section and the informal notion of reducibility that we used earlier in this chapter. The proof shows that E_{TM} is undecidable by reducing A_{TM} to it. Let's see whether we can convert this reduction to a mapping reduction.

From the original reduction, we may easily construct a function f that takes input $\langle M, w \rangle$ and produces output $\langle M_1 \rangle$, where M_1 is the Turing machine described in that proof. But M accepts w iff $L(M_1)$ is *not* empty so f is a mapping reduction from A_{TM} to $\overline{E_{TM}}$. It still shows that E_{TM} is undecidable because decidability is not affected by complementation, but it doesn't give a mapping reduction from A_{TM} to E_{TM} . In fact, no such reduction exists, as you are asked to show in Exercise 5.5. ■

The sensitivity of mapping reducibility to complementation is important in the use of reducibility to prove nonrecognizability of certain languages. We can also use mapping reducibility to show that problems are not Turing-recognizable. The following theorem is analogous to Theorem 5.22.

THEOREM 5.28

If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable.

The proof is the same as that of Theorem 5.22, except that M and N are recognizers instead of deciders.

COROLLARY 5.29

If $A \leq_m B$ and A is not Turing-recognizable, then B is not Turing-recognizable.

In a typical application of this corollary, we let A be $\overline{A_{\text{TM}}}$, the complement of A_{TM} . We know that $\overline{A_{\text{TM}}}$ is not Turing-recognizable from Corollary 4.23. The definition of mapping reducibility implies that $A \leq_m B$ means the same as $\overline{A} \leq_m \overline{B}$. To prove that B isn't recognizable, we may show that $A_{\text{TM}} \leq_m \overline{B}$. We can also use mapping reducibility to show that certain problems are neither Turing-recognizable nor co-Turing-recognizable, as in the following theorem.

THEOREM 5.30

EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.

PROOF First we show that EQ_{TM} is not Turing-recognizable. We do so by showing that A_{TM} is reducible to $\overline{EQ_{\text{TM}}}$. The reducing function f works as follows.

$F =$ “On input $\langle M, w \rangle$, where M is a TM and w a string:

1. Construct the following two machines, M_1 and M_2 .
 $M_1 =$ “On any input:
 1. *Reject.*” $M_2 =$ “On any input:
 1. Run M on w . If it accepts, *accept.*”
2. Output $\langle M_1, M_2 \rangle$.”

Here, M_1 accepts nothing. If M accepts w , M_2 accepts everything, and so the two machines are not equivalent. Conversely, if M doesn't accept w , M_2 accepts nothing, and they are equivalent. Thus f reduces A_{TM} to $\overline{EQ_{\text{TM}}}$, as desired.

To show that $\overline{EQ_{\text{TM}}}$ is not Turing-recognizable, we give a reduction from A_{TM} to the complement of $\overline{EQ_{\text{TM}}}$ —namely, EQ_{TM} . Hence we show that $A_{\text{TM}} \leq_m EQ_{\text{TM}}$. The following TM G computes the reducing function g .

$G =$ “On input $\langle M, w \rangle$, where M is a TM and w a string:

1. Construct the following two machines, M_1 and M_2 .
 $M_1 =$ “On any input:
 1. *Accept.*” $M_2 =$ “On any input:
 1. Run M on w .
 2. If it accepts, *accept.*”
2. Output $\langle M_1, M_2 \rangle$.”

The only difference between f and g is in machine M_1 . In f , machine M_1 always rejects, whereas in g it always accepts. In both f and g , M accepts w iff M_2 always accepts. In g , M accepts w iff M_1 and M_2 are equivalent. That is why g is a reduction from A_{TM} to EQ_{TM} .

EXERCISES

- 5.1 Show that EQ_{CFG} is undecidable.
- 5.2 Show that EQ_{CFG} is co-Turing-recognizable.
- 5.3 Find a match in the following instance of the Post Correspondence Problem.

$$\left\{ \left[\frac{ab}{abab} \right], \left[\frac{b}{a} \right], \left[\frac{aba}{b} \right], \left[\frac{aa}{a} \right] \right\}$$

- 5.4 If $A \leq_m B$ and B is a regular language, does that imply that A is a regular language? Why or why not?
- ^A5.5 Show that A_{TM} is not mapping reducible to E_{TM} . In other words, show that no computable function reduces A_{TM} to E_{TM} . (Hint: Use a proof by contradiction, and facts you already know about A_{TM} and E_{TM} .)
- ^A5.6 Show that \leq_m is a transitive relation.
- ^A5.7 Show that if A is Turing-recognizable and $A \leq_m \bar{A}$, then A is decidable.
- ^A5.8 In the proof of Theorem 5.15, we modified the Turing machine M so that it never tries to move its head off the left-hand end of the tape. Suppose that we did not make this modification to M . Modify the PCP construction to handle this case.

PROBLEMS

- 5.9 Let $T = \{\langle M \rangle \mid M \text{ is a TM that accepts } w^R \text{ whenever it accepts } w\}$. Show that T is undecidable.
- ^A5.10 Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape when it is run on input w . Formulate this problem as a language and show that it is undecidable.
- ^A5.11 Consider the problem of determining whether a two-tape Turing machine ever writes a nonblank symbol on its second tape during the course of its computation on any input string. Formulate this problem as a language and show that it is undecidable.
- 5.12 Consider the problem of determining whether a single-tape Turing machine ever writes a blank symbol over a nonblank symbol during the course of its computation on any input string. Formulate this problem as a language and show that it is undecidable.
- 5.13 A *useless state* in a Turing machine is one that is never entered on any input string. Consider the problem of determining whether a Turing machine has any useless states. Formulate this problem as a language and show that it is undecidable.

- 5.14 Consider the problem of determining whether a Turing machine M on an input w ever attempts to move its head left when its head is on the left-most tape cell. Formulate this problem as a language and show that it is undecidable.
- 5.15 Consider the problem of determining whether a Turing machine M on an input w ever attempts to move its head left at any point during its computation on w . Formulate this problem as a language and show that it is decidable.
- 5.16 Let $\Gamma = \{0, 1, \sqcup\}$ be the tape alphabet for all TMs in this problem. Define the **busy beaver function** $BB: \mathcal{N} \rightarrow \mathcal{N}$ as follows. For each value of k , consider all k -state TMs that halt when started with a blank tape. Let $BB(k)$ be the maximum number of 1s that remain on the tape among all of these machines. Show that BB is not a computable function.
- 5.17 Show that the Post Correspondence Problem is decidable over the unary alphabet $\Sigma = \{1\}$.
- 5.18 Show that the Post Correspondence Problem is undecidable over the binary alphabet $\Sigma = \{0, 1\}$.
- 5.19 In the *silly Post Correspondence Problem*, *SPCP*, the top string in each pair has the same length as the bottom string. Show that the *SPCP* is decidable.
- 5.20 Prove that there exists an undecidable subset of $\{1\}^*$.
- 5.21 Let $AMBIG_{CFG} = \{\langle G \rangle \mid G \text{ is an ambiguous CFG}\}$. Show that $AMBIG_{CFG}$ is undecidable. (Hint: Use a reduction from *PCP*. Given an instance

$$P = \left\{ \left[\frac{t_1}{b_1} \right], \left[\frac{t_2}{b_2} \right], \dots, \left[\frac{t_k}{b_k} \right] \right\}$$

of the Post Correspondence Problem, construct a CFG G with the rules

$$\begin{aligned} S &\rightarrow T \mid B \\ T &\rightarrow t_1 T a_1 \mid \dots \mid t_k T a_k \mid t_1 a_1 \mid \dots \mid t_k a_k \\ B &\rightarrow b_1 B a_1 \mid \dots \mid b_k B a_k \mid b_1 a_1 \mid \dots \mid b_k a_k, \end{aligned}$$

where a_1, \dots, a_k are new terminal symbols. Prove that this reduction works.)

- 5.22 Show that A is Turing-recognizable iff $A \leq_m A_{TM}$.
- 5.23 Show that A is decidable iff $A \leq_m 0^* 1^*$.
- 5.24 Let $J = \{w \mid \text{either } w = 0x \text{ for some } x \in A_{TM}, \text{ or } w = 1y \text{ for some } y \in \overline{A_{TM}}\}$. Show that neither J nor \overline{J} is Turing-recognizable.
- 5.25 Give an example of an undecidable language B , where $B \leq_m \overline{B}$.
- 5.26 Define a **two-headed finite automaton** (2DFA) to be a deterministic finite automaton that has two read-only, bidirectional heads that start at the left-hand end of the input tape and can be independently controlled to move in either direction. The tape of a 2DFA is finite and is just large enough to contain the input plus two additional blank tape cells, one on the left-hand end and one on the right-hand end, that serve as delimiters. A 2DFA accepts its input by entering a special accept state. For example, a 2DFA can recognize the language $\{a^n b^n c^n \mid n \geq 0\}$.
- Let $A_{2DFA} = \{\langle M, x \rangle \mid M \text{ is a 2DFA and } M \text{ accepts } x\}$. Show that A_{2DFA} is decidable.
 - Let $E_{2DFA} = \{\langle M \rangle \mid M \text{ is a 2DFA and } L(M) = \emptyset\}$. Show that E_{2DFA} is not decidable.

5.27 A *two-dimensional finite automaton* (2DIM-DFA) is defined as follows. The input is an $m \times n$ rectangle, for any $m, n \geq 2$. The squares along the boundary of the rectangle contain the symbol # and the internal squares contain symbols over the input alphabet Σ . The transition function $\delta: Q \times (\Sigma \cup \{\#\}) \rightarrow Q \times \{L, R, U, D\}$ indicates the next state and the new head position (Left, Right, Up, Down). The machine accepts when it enters one of the designated accept states. It rejects if it tries to move off the input rectangle or if it never halts. Two such machines are equivalent if they accept the same rectangles. Consider the problem of determining whether two of these machines are equivalent. Formulate this problem as a language and show that it is undecidable.

A* 5.28 Rice's theorem. Let P be any nontrivial property of the language of a Turing machine. Prove that the problem of determining whether a given Turing machine's language has property P is undecidable.

In more formal terms, let P be a language consisting of Turing machine descriptions where P fulfills two conditions. First, P is nontrivial—it contains some, but not all, TM descriptions. Second, P is a property of the TM's language—whenever $L(M_1) = L(M_2)$, we have $\langle M_1 \rangle \in P$ iff $\langle M_2 \rangle \in P$. Here, M_1 and M_2 are any TMs. Prove that P is an undecidable language.

5.29 Show that both conditions in Problem 5.28 are necessary for proving that P is undecidable.

5.30 Use Rice's theorem, which appears in Problem 5.28, to prove the undecidability of each of the following languages.

- A a.** $INFINITE_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is an infinite language}\}.$
- b.** $\{\langle M \rangle \mid M \text{ is a TM and } 1011 \in L(M)\}.$
- c.** $ALL_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Sigma^*\}.$

5.31 Let

$$f(x) = \begin{cases} 3x + 1 & \text{for odd } x \\ x/2 & \text{for even } x \end{cases}$$

for any natural number x . If you start with an integer x and iterate f , you obtain a sequence, $x, f(x), f(f(x)), \dots$. Stop if you ever hit 1. For example, if $x = 17$, you get the sequence 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Extensive computer tests have shown that every starting point between 1 and a large positive integer gives a sequence that ends in 1. But the question of whether all positive starting points end up at 1 is unsolved; it is called the $3x + 1$ problem.

Suppose that A_{TM} were decidable by a TM H . Use H to describe a TM that is guaranteed to state the answer to the $3x + 1$ problem.

5.32 Prove that the following two languages are undecidable.

- a.** $OVERLAP_{CFG} = \{\langle G, H \rangle \mid G \text{ and } H \text{ are CFGs where } L(G) \cap L(H) \neq \emptyset\}.$ (Hint: Adapt the hint in Problem 5.21.)
- b.** $PREFIX-FREE_{CFG} = \{\langle G \rangle \mid G \text{ is a CFG where } L(G) \text{ is prefix-free}\}.$

5.33 Consider the problem of determining whether a PDA accepts some string of the form $\{wv \mid w \in \{0,1\}^*\}$. Use the computation history method to show that this problem is undecidable.

5.34 Let $X = \{\langle M, w \rangle \mid M \text{ is a single-tape TM that never modifies the portion of the tape that contains the input } w\}$. Is X decidable? Prove your answer.

5.10 Let $B = \{\langle M, w \rangle \mid M \text{ is a two-tape TM that writes a nonblank symbol on its second tape when it is run on } w\}$. Show that A_{TM} reduces to B . Assume for the sake of contradiction that TM R decides B . Then construct a TM S that uses R to decide A_{TM} .

$S =$ “On input $\langle M, w \rangle$:

1. Use M to construct the following two-tape TM T .
 $T =$ “On input x :
 1. Simulate M on x using the first tape.
 2. If the simulation shows that M accepts, write a nonblank symbol on the second tape.”
2. Run R on $\langle T, w \rangle$ to determine whether T on input w writes a nonblank symbol on its second tape.
3. If R accepts, M accepts w , so *accept*. Otherwise, *reject*.”

5.11 Let $C = \{\langle M \rangle \mid M \text{ is a two-tape TM that writes a nonblank symbol on its second tape when it is run on some input}\}$. Show that A_{TM} reduces to C . Assume for the sake of contradiction that TM R decides C . Construct a TM S that uses R to decide A_{TM} .

$S =$ “On input $\langle M, w \rangle$:

1. Use M and w to construct the following two-tape TM T_w .
 $T_w =$ “On any input:
 1. Simulate M on w using the first tape.
 2. If the simulation shows that M accepts, write a nonblank symbol on the second tape.”
2. Run R on $\langle T_w \rangle$ to determine whether T_w ever writes a nonblank symbol on its second tape.
3. If R accepts, M accepts w , so *accept*. Otherwise, *reject*.”

5.28 Assume for the sake of contradiction that P is a decidable language satisfying the properties and let R_P be a TM that decides P . We show how to decide A_{TM} using R_P by constructing TM S . First, let T_\emptyset be a TM that always rejects, so $L(T_\emptyset) = \emptyset$. You may assume that $\langle T_\emptyset \rangle \notin P$ without loss of generality because you could proceed with \overline{P} instead of P if $\langle T_\emptyset \rangle \in P$. Because P is not trivial, there exists a TM T with $\langle T \rangle \in P$. Design S to decide A_{TM} using R_P 's ability to distinguish between T_\emptyset and T .

$S =$ “On input $\langle M, w \rangle$:

1. Use M and w to construct the following TM M_w .
 $M_w =$ “On input x :
 1. Simulate M on w . If it halts and rejects, *reject*.
 If it accepts, proceed to stage 2.
 2. Simulate T on x . If it accepts, *accept*.”
2. Use TM R_P to determine whether $\langle M_w \rangle \in P$. If YES, *accept*.
 If NO, *reject*.”

TM M_w simulates T if M accepts w . Hence $L(M_w)$ equals $L(T)$ if M accepts w and \emptyset otherwise. Therefore, $\langle M_w \rangle \in P$ iff M accepts w .

- 5.30 (a) $INFINITE_{TM}$ is a language of TM descriptions. It satisfies the two conditions of Rice's theorem. First, it is nontrivial because some TMs have infinite languages and others do not. Second, it depends only on the language. If two TMs recognize the same language, either both have descriptions in $INFINITE_{TM}$ or neither do. Consequently, Rice's theorem implies that $INFINITE_{TM}$ is undecidable.