# Chapter 9

# Turing Machines

In our discussion so far we have encountered some fundamental ideas, in particular the concepts of regular and context-free languages and their association with finite automata and pushdown accepters. Our study has revealed that the regular languages form a proper subset of the context-free languages and, therefore, that pushdown automata are more powerful than finite automata. We also saw that context-free languages, while fundamental to the study of programming languages, are limited in scope. This was made clear in the last chapter, where our results showed that some simple languages, such as $\{a^n b^n c^n\}$and $\{ww\}$, are not context-free. This prompts us to look beyond context-free languages and investigate howone might define newlanguage families that include these examples. To do so, we return to the general picture of an automaton. If we compare finite automata with pushdown automata, we see that the nature of the temporary storage creates the difference between them. If there is no storage, we have a finite automaton; if the storage is a stack, we have the more powerful pushdown automaton. Extrapolating from this observation, we can expect to discover even more powerful language families if we give the automaton more flexible storage. For example, what would happen if, in the general scheme of Figure 1.3, we used two stacks, three stacks, a queue, or some other storage device? Does each storage device define a newkind of automaton and through it a newlanguage family? This approach raises a large number of questions, most of which turn out to be uninteresting. It is more instructive to ask a more ambitious question and consider howfar the concept of an automaton can be pushed. What can we say about the most powerful of automata and the limits of computation? This leads to the fundamental concept of a **Turing machine** and, in turn, to a precise definition of the idea of a mechanical or algorithmic computation.

We begin our study with a formal definition of a Turing machine, then develop some feeling for what is involved by doing some simple programs. Next we argue that, while the mechanism of a Turing machine is quite rudimentary, the concept is broad enough to cover very complex processes. The discussion culminates in the **Turing thesis**, which maintains that any computational process, such as those carried out by present-day computers, can be done on a Turing machine.

## 9.1 The Standard Turing Machine

Although we can envision a variety of automata with complex and sophisticated storage devices, a Turing machine's storage is actually quite simple. It can be visualized as a single, one-dimensional array of cells, each of which can hold a single symbol. This array extends indefinitely in both directions and is therefore capable of holding an unlimited amount of information. The information can be read and changed in any order. We will call such a storage device a **tape** because it is

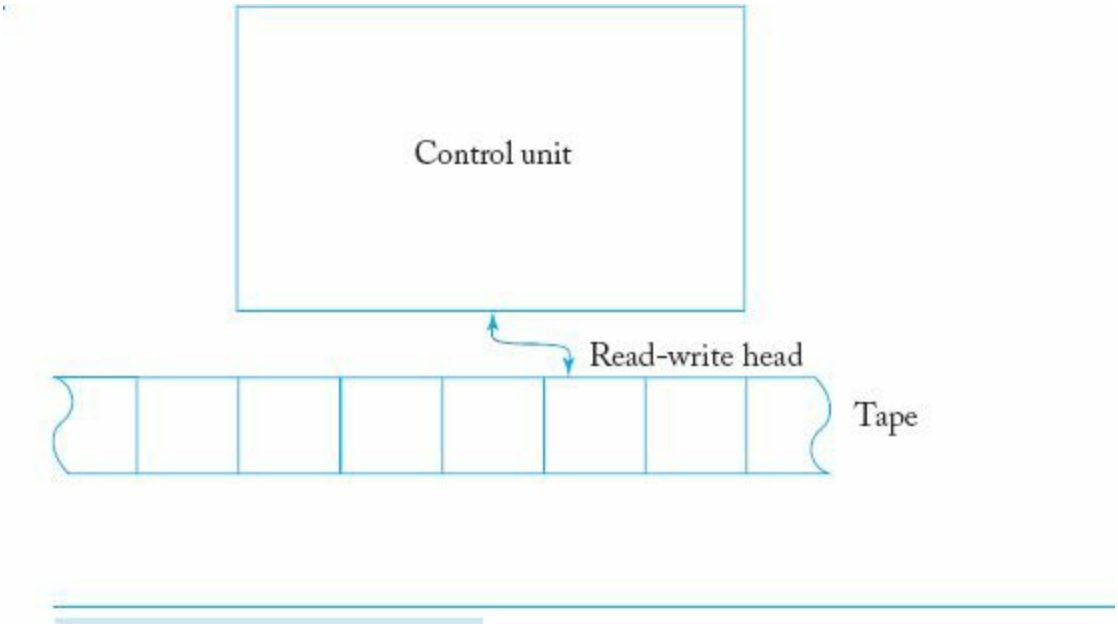
analogous to the magnetic tapes used in older computers.

## Definition of a Turing Machine

A Turing machine is an automaton whose temporary storage is a tape. This tape is divided into cells, each of which is capable of holding one symbol. Associated with the tape is a **read-write head** that can travel right or left on the tape and that can read and write a single symbol on each move. To deviate slightly from the general scheme of Chapter 1, the automaton that we use as a Turing machine will have neither an input file nor any special output mechanism. Whatever input and output is necessary will be done on the machine's tape. We will see later that this modification of our general model in Section 1.2 is of little consequence. We could retain the input file and a specific output mechanism without affecting any of the conclusions we are about to draw, but we leave them out because the resulting automaton is a little easier to describe.

A diagram giving an intuitive visualization of a Turing machine is shown in Figure 9.1. Definition 9.1 makes the notion precise.

**Figure 9.1**




**Definition 9.1**

A Turing machine $M$ is defined by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F),$$

where

$Q$ is the set of internal states,

$\Sigma$ is the input alphabet

| | |
|---|---|
| $\Gamma$ | is the finite set of symbols called the **tape alphabet**, |
| $\delta$ | is the transition function, |
| $\square \in \Gamma$ | is a special symbol called the **blank**, |
| $q_0 \in Q$ | is the initial state, |
| $F \subseteq Q$ | is the set of final states. |

In the definition of a Turing machine, we assume that $\Sigma \subseteq \Gamma - \{\square\}$, that is, that the input alphabet is a subset of the tape alphabet, not including the blank. Blanks are ruled out as input for reasons that will become apparent shortly. The transition function $\delta$ is defined as

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}.$$

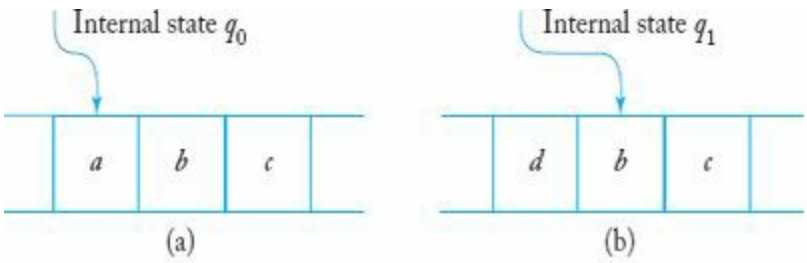
In general,$\delta$ is a partial function on $Q \times \Gamma$; its interpretation gives the principle by which a Turing machine operates. The arguments of $\delta$ are the current state of the control unit and the current tape symbol being read. The result is a new state of the control unit, a new tape symbol, which replaces the old one, and a move symbol, $L$ or $R$. The move symbol indicates whether the read-write head moves left or right one cell after the new symbol has been written on the tape.

## Example 9.1

Figure 9.2 shows the situation before and after the move

$$\delta\ (q_0, a) = (q_1, d, R).$$

**Figure 9.2**




The situation (a) before the move and (b) after the move.

We can think of a Turing machine as a rather simple computer. It has a processing unit, which has a finite memory, and in its tape, it has a secondary storage of unlimited capacity. The instructions that such a computer can carry out are very limited: It can sense a symbol on its tape and use the result to decide what to do next. The only actions the machine can perform are to rewrite the current symbol,

to change the state of the control, and to move the read-write head. This small instruction set may seem inadequate for doing complicated things, but this is not so. Turing machines are quite powerful in principle. The transition function δ defines howthis computer acts, and we often call it the "program" of the machine.

As always, the automaton starts in the given initial state with some information on the tape. It then goes through a sequence of steps controlled by the transition function δ. During this process, the contents of any cell on the tape may be examined and changed many times. Eventually, the whole process may terminate, which we achieve in a Turing machine by putting it into a **halt state.** A Turing machine is said to halt whenever it reaches a configuration for which δ is not defined; this is possible because δ is a partial function. In fact, we will assume that no transitions are defined for any final state, so the Turing machine will halt whenever it enters a final state.

---

## Example 9.2

Consider the Turing machine defined by

$$Q = \{q_0, q_1\},$$

$$\Sigma = \{a, b\},$$

$$\Gamma = \{a, b, \square\},$$

$$F = \{q_1\},$$

and

$$\delta\,(q_0, a) = (q_0, b, R),$$

$$\delta\,(q_0, b) = (q_0, b, R),$$

$$\delta\,(q_0, \square) = (q_1, \square, L),$$

If this Turing machine is started in state $q_0$ with the symbol $a$ under the read-write head, the applicable transition rule is $\delta\,(q_0,a) = (q_0,b,R)$. Therefore, the read-write head will replace the $a$ with a b, then move right on the tape. The machine will remain in state $q_0$. Any subsequent $a$ will also be replaced with a b, but b's will not be modified. When the machine encounters the first blank, it will move left one cell, then halt in final state $q_1$.

Figure 9.3 shows several stages of the process for a simple initial configuration.

---

**Figure 9.3**


A sequence of moves.

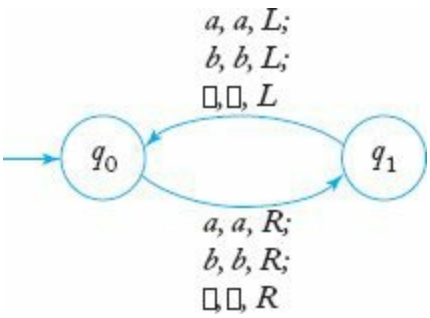
As before, we can use transition graphs to represent Turing machines. Now we label the edges of the graph with three items: the current tape symbol, the symbol that replaces it, and the direction in which the read-write head is to move. The Turing machine in Example 9.2 is represented by the transition graph in Figure 9.4.

**Figure 9.4**


## Example 9.3

Look at the Turing machine in Figure 9.5. To see what will happen, we can trace a typical case. Suppose that the tape initially contains $ab\ldots$, with the read-write head on the $a$. The machine then reads the $a$, but does not change it. Its next state is $q_1$ and the read-write head moves right, so that it is now over the $b$. This symbol is also read and left unchanged. The machine goes back into state $q_0$ and the read-write head moves left. We are now back exactly in the original state, and the sequence of moves starts again. It is clear from this that the machine, whatever the initial information on its tape, will run forever, with the read-write head moving alternately right then left, but making no modifications to the tape. This is an instance of a Turing machine that does not halt. In analogy with programming terminology, we say that the Turing machine is in an **infinite loop**.

**Figure 9.5**




Since one can make several different definitions of a Turing machine, it is worthwhile to summarize the main features of our model, which we will call a **standard Turing machine**:

1. The Turing machine has a tape that is unbounded in both directions, allowing any number of left and right moves.

2. The Turing machine is deterministic in the sense that δ defines at most one move for each configuration.

3. There is no special input file. We assume that at the initial time the tape has some specified content. Some of this may be considered input. Similarly, there is no special output device. Whenever the machine halts, some or all of the contents of the tape may be viewed as output.

These conventions were chosen primarily for the convenience of subsequent discussion. In Chapter 10, we will look at other versions of Turing machines and discuss their relation to our standard model.

Here, as in the case of pda's, the most convenient way to exhibit a sequence of configurations of a Turing machine uses the idea of an instantaneous description. Any configuration is completely determined by the current state of the control unit, the contents of the tape, and the position of the read-write head. We will use the notation in which

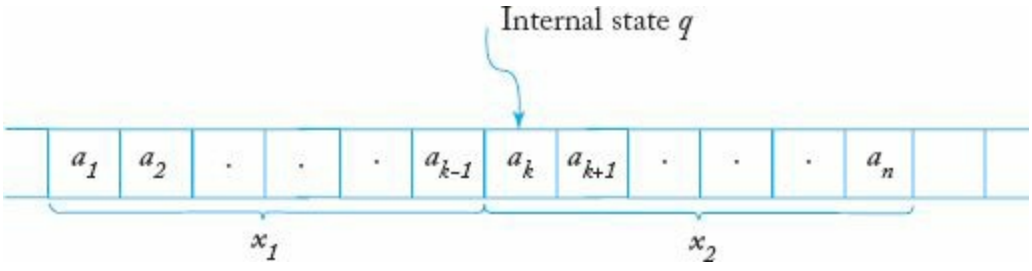
$$x_1 q x_2$$

or

$$a_1 a_2 \ldots a_{k-1} q a_k a_{k+1} \ldots a_n$$

is the instantaneous description of a machine in state $q$ with the tape depicted in Figure 9.6. The symbols $a_1, \ldots, a_n$ show the tape contents, while $q$ defines the state of the control unit. This convention is chosen so that the position of the read-write head is over the cell containing the symbol immediately following $q$.

**Figure 9.6**




The instantaneous description gives only a finite amount of information to the right and left of the read-write head. The unspecified part of the tape is assumed to contain all blanks; normally such blanks are irrelevant and are not shown explicitly in the instantaneous description. If the position of blanks is relevant to the discussion, however, the blank symbol may appear in the instantaneous description. For example, the instantaneous description $q\square\omega$ indicates that the read-write head is on the cell to the immediate left of the first symbol of $w$ and that this cell contains a blank.

**Example 9.4**

The pictures drawn in Figure 9.3 correspond to the sequence of instantaneous descriptions $q_0aa$, $bq_0a$, $bbq_0\square$, $bq_1b$.

A move from one configuration to another will be denoted by $\vdash$. Thus, if

$$\delta\,(q_1, c)=(q_2, e, R),$$

then the move

$$abq_1cd \vdash abeq_2d$$

is made whenever the internal state is $q_1$, the tape contains *abcd*, and the read-write head is on the *c*. The symbol $\overset{*}{\vdash}$ has the usual meaning of an arbitrary number of moves. Subscripts, such as $\vdash_M$, are used in arguments to distinguish between several machines.

### Example 9.5

The action of the Turing machine in Figure 9.3 can be represented by

$$q_0aa \vdash bq_0a \vdash bbq_0\square \vdash bq_1b$$

or

$$q_0aa \overset{*}{\vdash} bq_1b.$$

For further discussion, it is convenient to summarize the various observations just made in a formal way.

### Definition 9.2

Let $M= (Q, \Sigma,\Gamma,\delta,q_0,, \square,F)$ be a Turing machine. Then any string $a_i\ldots a_{k-1}q_1a_ka_{k+1}\ldots a_n$, with $a_i \in \Gamma$ and $q_1 \in Q$, is an instantaneous description of *M*. A move

$$a_1 \cdots a_{k-1}q_1a_ka_{k+1} \cdots a_n \vdash a_1 \cdots a_{k-1}bq_2a_{k+1} \cdots a_n$$

is possible if and only if

$$\delta\,(q_1,a_k) = (q_2,b,R).$$

A move

$$a_1 \cdots a_{k-1} q_1 a_k a_{k+1} \cdots a_n \vdash a_1 \cdots q_2 a_{k-1} b a_{k+1} \cdots a_n$$

is possible if and only if

$$\delta\,(q_1,a_k) = (q_2,b,L).$$

$M$ is said to halt starting from some initial configuration $x_1 q_i x_2$ if

$$x_1 q_i x_2 \overset{*}{\vdash} y_1 q_j a y_2$$

for any $q_j$ and $a$, for which $\delta\,(q_j,a)$ is undefined. The sequence of configurations leading to a halt state will be called a **computation.**

---

Example 9.3 shows the possibility that a Turing machine will never halt, proceeding in an endless loop from which it cannot escape. This situation plays a fundamental role in the discussion of Turing machines, so we use a special notation for it. We will represent it by indicating that, starting from the initial configuration $x_1 q x_2$, the machine goes into a loop and never halts.

$$x_1 q x_2 \overset{*}{\vdash} \infty,$$

## Turing Machines as Language Accepters

Turing machines can be viewed as accepters in the following sense. A string $\omega$ is written on the tape, with blanks filling out the unused portions. The machine is started in the initial state $q_0$ with the read-write head positioned on the leftmost symbol of $\omega$. If, after a sequence of moves, the Turing machine enters a final state and halts, then $w$ is considered to be accepted.

**Definition 9.3**

Let $M = (Q,\Sigma,\Gamma,\delta;q_0,\square,F)$ be a Turing machine. Then the language accepted by $M$ is

$$L\,(M) = \left\{ w \in \Sigma^+ : q_0 w \overset{*}{\vdash} x_1 q_f x_2 \text{ for some } q_f \in F, x_1, x_2 \in \Gamma^* \right\}$$

---

This definition indicates that the input $w$ is written on the tape with blanks on either side. The reason for excluding blanks from the input now becomes clear: It assures us that all the input is restricted to a well-defined region of the tape, bracketed by blanks on the right and left. Without this convention, the machine could not limit the region in which it must look for the input; no matter how many blanks it saw, it could never be sure that there was not some nonblank input somewhere else on the tape.

Definition 9.3 tells us what must happen when $w \in \Gamma L(M)$. It says nothing about the outcome for any other input. When $w$ is not in $L(M)$, one of two things can happen: The machine can halt in a nonfinal state or it can enter an infinite loop and never halt. Any string for which $M$ does not halt is by definition not in $L(M)$.

## Example 9.6

For $\Sigma = \{0,1\}$, design a Turing machine that accepts the language denoted by the regular expression $00^*$.

This is an easy exercise in Turing machine programming. Starting at the left end of the input, we read each symbol and check that it is a 0. If it is, we continue by moving right. If we reach a blank without encountering anything but 0, we terminate and accept the string. If the input contains a 1 anywhere, the string is not in $L(00^*)$, and we halt in a nonfinal state. To keep track of the computation, two internal states $Q = \{q_0,q_1\}$ and one final state $F = \{q_1\}$ are sufficient. As transition function we can take As long as a 0 appears under the read-write head, the head will move to the right. If at any time a 1 is read, the machine will halt in the nonfinal state $q_0$, since $\delta(q_0,1)$ is undefined. Note that the Turing machine also halts in a final state if started in state $q_0$ on a blank. We could interpret this as acceptance of $\lambda$, but for technical reasons the empty string is not included in Definition 9.3.

$$\delta(q_0, 0) = (q_0, 0, R),$$
$$\delta(q_0, \square) = (q_1, \square, R).$$

The recognition of more complicated languages is more difficult. Since Turing machines have a primitive instruction set, the computations that we can program easily in a higher-level language are often cumbersome on a Turing machine. Still, it is possible, and the concept is easy to understand, as the next examples illustrate.

## Example 9.7

For $\Sigma = \{a,b\}$, design a Turing machine that accepts

$$L = \{a_n b_n : n \geq 1\}.$$

Intuitively, we solve the problem in the following fashion. Starting at the leftmost $a$, we check it off by replacing it with some symbol, say $x$. We then let the read-write head travel right to find the leftmost $b$, which in turn is checked off by replacing it with another symbol, say $y$. After that, we go left again to the leftmost $a$, replace it with an $x$, then move to the leftmost $b$and replace it with $y$, and so on. Traveling back and forth this way, we match each $a$ with a corresponding $b$. If after some time no a's or b's remain, then the string must be in $L$.

Working out the details, we arrive at a complete solution for which $Q = \{q_0,q_1,q_2,q_3,q_4\}$, $F = \{q_4\}$, $\Sigma = \{a,b\}$, $\Gamma = \{a,b,x,y,\square\}$. The transitions can be broken into several parts. The set

$$\delta\ (q_0, a)=(q_1, x,R),$$

$$\delta\ (q_1, a)=(q_1, a,R),$$

$$\delta\ (q_1, y)=(q_1, y,R),$$

$$\delta\ (q_1, b)=(q_2, y,R),$$

replaces the leftmost $a$ with an $x$, then causes the read-write head to travel right to the first $b$, replacing it with a $y$. When the $y$ is written, the machine enters a state $q_2$, indicating that an $a$ has been successfully paired with a $b$.

The next set of transitions reverses the direction until an $x$ is encountered, repositions the read-write head over the leftmost $a$, and returns control to the initial state.

$$\delta\ (q_2,y)=(q_2,y,L),$$

$$\delta\ (q_2,a)=(q_2,a,L),$$

$$\delta\ (q_2,x)=(q_0,x,R),$$

We are now back in the initial state $q_0$, ready to deal with the next $a$ and $b$.

After one pass through this part of the computation, the machine will have carried out the partial computation

$$q_0aa\cdots abb\cdots b \overset{*}{\vdash} xq_0a\cdots ayb\cdots b,$$

so that a single $a$ has been matched with a single $b$. After two passes, we will have completed the partial computation and so on, indicating that the matching process is being carried out properly.

$$q_0aa\cdots abb\cdots b \overset{*}{\vdash} xxq_0\cdots ayy\cdots b,$$

When the input is a string $a^nb^n$, the rewriting continues this way, stopping only when there are no more $a$'s to be erased. When looking for the leftmost $a$, the read-write head travels left with the machine in state $q_2$. When an $x$ is encountered, the direction is reversed to get the $a$. But now, instead of finding an $a$ it will find a $y$. To terminate, a final check is made to see if all $a$'s and $b$'s have been replaced (to detect input where an $a$ follows a $b$). This can be done by

$$\delta\ (q_0,y)=(q_3,y,R),$$

$$\delta\ (q_3,y)=(q_3,y,R),$$

$$\delta\ (q_3,\square)=(q_4,\square,R),$$

If we input a string not in the language, the computation will halt in a nonfinal state. For example,

if we give the machine a string $a^n b^m$, with $n > m$, the machine will eventually encounter a blank in state $q_1$. It will halt because no transition is specified for this case. Other input not in the language will also lead to a nonfinal halting state (see Exercise 3 at the end of this section).

The particular input *aabb* gives the following successive instantaneous descriptions:

$$\begin{aligned} q_0aabb &\vdash xq_1abb \vdash xaq_1bb \vdash xq_2ayb \\ &\vdash q_2xayb \vdash xq_0ayb \vdash xxq_1yb \\ &\vdash xxyq_1b \vdash xxq_2yy \vdash xq_2xyy \\ &\vdash xxq_0yy \vdash xxyq_3y \vdash xxyyq_3\square \\ &\vdash xxyy\square q_4\square. \end{aligned}$$

At this point the Turing machine halts in a final state, so the string *aabb* is accepted.

You are urged to trace this program with several more strings in $L$, as well as with some not in $L$.

---

**Example 9.8**

Design a Turing machine that accepts

$$L=\{a^n b^n c^n : n \geq 1\}.$$

The ideas used in Example 9.7 are easily carried over to this case. We match each $a$,$b$, and $c$ by replacing them in order by $x$,$y$, and $z$, respectively. At the end, we check that all original symbols have been rewritten. Although conceptually a simple extension of the previous example, writing the actual program is tedious. We leave it as a somewhat lengthy, but straightforward exercise. Notice that even though $\{a^n b^n\}$ is a context-free language and $\{a^n\ b^n\ c^n\}$ is not, they can be accepted by Turing machines with very similar structures.

---

One conclusion we can draw from this example is that a Turing machine can recognize some languages that are not context-free, a first indication that Turing machines are more powerful than pushdown automata.

## Turing Machines as Transducers

We have had little reason so far to study transducers; in language theory, accepters are quite adequate. But as we will shortly see, Turing machines are not only interesting as language accepters, they also provide us with a simple abstract model for digital computers in general. Since the primary purpose of a computer is to transform input into output, it acts as a transducer. If we want to model computers using Turing machines, we have to look at this aspect more closely.

The input for a computation will be all the nonblank symbols on the tape at the initial time. At the conclusion of the computation, the output will be whatever is then on the tape. Thus, we can view a

Turing machine transducer $M$ as an implementation of a function $f$ defined by

$$\widehat{w} = f(w),$$

provided that

$$q_0 w \overset{*}{\vdash}_M q_f \widehat{w},$$

for some final state $q_f$.

**Definition 9.4**

A function $f$ with domain $D$ is said to be **Turing-computable** or just **computable** if there exists some Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ such that

$$q_0 w \overset{*}{\vdash}_M q_f f(w), \qquad q_f \in F,$$

for all $w \in D$.

As we will shortly claim, all the common mathematical functions, no matter howcomplicated, are Turing-computable. We start by looking at some simple operations, such as addition and arithmetic comparison.

**Example 9.9**

Given two positive integers $x$ and $y$, design a Turing machine that computes $x + y$.

We first have to choose some convention for representing positive integers. For simplicity, we will use unary notation in which any positive integer $x$ is represented by $w(x) \in \{1\}^+$, such that

$$|w(x)| = x.$$

We must also decide how $x$ and $y$ are placed on the tape initially and howtheir sum is to appear at the end of the computation. We will assume that $w(x)$ and $w(y)$ are on the tape in unary notation, separated by a single 0, with the read-write head on the leftmost symbol of $w(x)$. After the computation, $w(x+y)$ will be on the tape followed by a single 0, and the read-write head will be positioned at the left end of the result. We therefore want to design a Turing machine for performing the computation

$$q_0 w(x) 0 w(y) \overset{*}{\vdash} q_f w(x+y) 0,$$

where $q_f$ is a final state. Constructing a program for this is relatively simple. All we need to do is to move the separating 0 to the right end of $w(y)$, so that the addition amounts to nothing more than the

coalescing of the two strings. To achieve this, we construct $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, with $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $F = \{q_4\}$, and

$$\delta(q_0, 1) = (q_0, 1, R),$$

$$\delta(q_0, 0) = (q_0, 1, R),$$

$$\delta(q_1, 1) = (q_1, 1, R),$$

$$\delta(q_1, \square) = (q_2, \square, L),$$

$$\delta(q_2, 1) = (q_3, 0, L),$$

$$\delta(q_3, 1) = (q_3, 1, L),$$

$$\delta(q_3, \square) = (q_4, \square, R),$$

Note that in moving the 0 right we temporarily create an extra 1, a fact that is remembered by putting the machine into state $q_1$. The transition $\delta(q_2, 1) = (q_3, 0, R)$ is needed to remove this at the end of the computation. This can be seen from the sequence of instantaneous descriptions for adding 111 to 11:

$$\begin{aligned} q_0 111011 &\vdash 1 q_0 11011 \vdash 11 q_0 1011 \vdash 111 q_0 011 \\ &\vdash 1111 q_1 11 \vdash 11111 q_1 1 \vdash 111111 q_1 \square \\ &\vdash 11111 q_2 1 \vdash 1111 q_3 10 \\ &\overset{*}{\vdash} q_3 \square 111110 \vdash q_4 111110. \end{aligned}$$

Unary notation, although cumbersome for practical computations, is very convenient for programming Turing machines. The resulting programs are much shorter and simpler than if we had used another representation, such as binary or decimal.

---

Adding numbers is one of the fundamental operations of any computer, one that plays a part in the synthesis of more complicated instructions. Other basic operations are copying strings and simple comparisons. These can also be done easily on a Turing machine.

### Example 9.10

Design a Turing machine that copies strings of 1's. More precisely, find a machine that performs the computation

$$q_0 w \overset{*}{\vdash} q_f w w,$$

for any $w \in \{1\}^+$.

To solve the problem, we implement the following intuitive process:

**1.** Replace every 1 by an $x$.

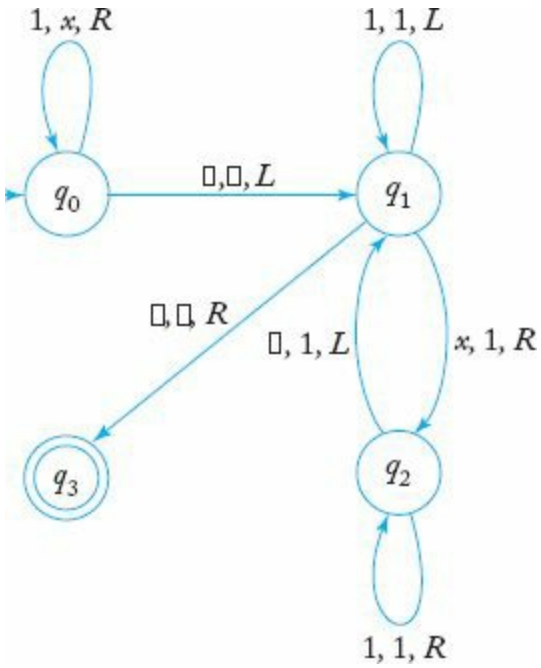
**2.** Find the rightmost $x$ and replace it with 1.

**3.** Travel to the right end of the current nonblank region and create a 1 there.

**4.** Repeat Steps 2 and 3 until there are no more x's.

The solution is shown in the transition graph in Figure 9.7. It may be a little hard to see at first that the solution is correct, so let us trace the program with the simple string 11. The computation performed in this case is

**Figure 9.7**




$$
\begin{aligned}
q_0 11 &\vdash x q_0 1 \vdash x x q_0 \square \vdash x q_1 x \\
&\vdash x 1 q_2 \square \vdash x q_1 11 \vdash q_1 x 11 \\
&\vdash 1 q_2 11 \vdash 11 q_2 1 \vdash 111 q_2 \square \\
&\vdash 11 q_1 11 \vdash 1 q_1 111 \\
&\vdash q_1 1111 \vdash q_1 \square 1111 \vdash q_3 1111.
\end{aligned}
$$

## Example 9.11

Let $x$ and $y$ be two positive integers represented in unary notation. Construct a Turing machine that will halt in a final state $q_y$ if $x \geq y$, and that will halt in a nonfinal state $q_n$ if $x < y$. More specifically, the machine is to perform the computation

$$q_0 w(x)\,0w(y) \overset{*}{\vdash} q_y w(x)\,0w(y) \qquad \text{if } x \geq y,$$
$$q_0 w(x)\,0w(y) \overset{*}{\vdash} q_n w(x)\,0w(y) \qquad \text{if } x < y.$$

To solve this problem, we can use the idea in Example 9.7 with some minor modifications. Instead of matching a's and b's, we match each 1 on the left of the dividing 0 with the 1 on the right. At the end of the matching, we will have on the tape either

$$xx\ldots110xx\ldots x\,\square$$

or

$$xx\ldots xx0xx\ldots x11\,\square$$

depending on whether $x > y$ or $y > x$. In the first case, when we attempt to match another 1, we encounter the blank at the right of the working space. This can be used as a signal to enter the state $q_y$. In the second case, we still find a 1 on the right when all 1's on the left have been replaced. We use this to get into the other state $q_n$. The complete program for this is straightforward and is left as an exercise.

This example makes the important point that a Turing machine can be programmed to make decisions based on arithmetic comparisons. This kind of simple decision is common in the machine language of computers, where alternate instruction streams are entered, depending on the outcome of an arithmetic operation.

---

## EXERCISES

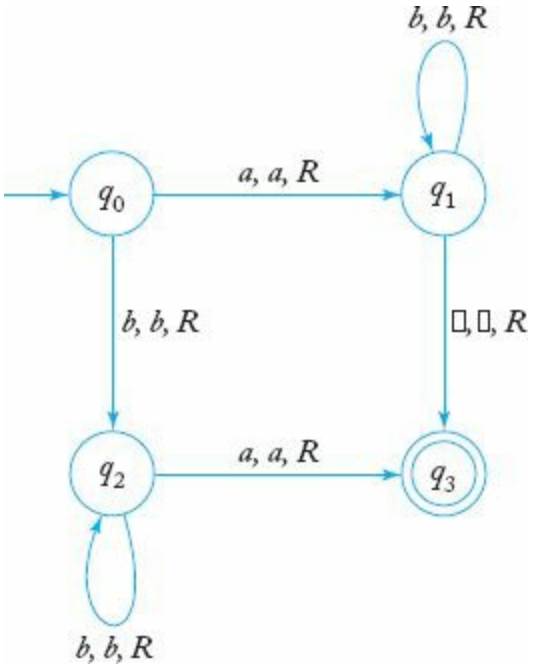
** 1. Write a Turing machine simulator in some higher-level programming language. Such a simulator should accept as input the description of any Turing machine, together with an initial configuration, and should produce as output the result of the computation.

2. Design a Turing machine with no more than three states that accepts the language $L(a\ (a + b)^*)$. Assume that $\Sigma = \{a,b\}$. Is it possible to do this with a two-state machine?

3. Determine what the Turing machine in Example 9.7 does when presented with the inputs *aba* and *aaabbbb*.

4. Is there any input for which the Turing machine in Example 9.7 goes into an infinite loop?

5. What language is accepted by the Turing machine whose transition graph is in the figure below?




**6.** What happens in Example 9.10 if the string $w$ contains any symbol other than 1?

**7.** Construct Turing machines that will accept the following languages on $\{a, b\}$.

(a) $L = L(aba^*b)$.

(b) $L = \{w : |w| \text{ is even}\}$.

(c) $L = \{w : |w| \text{ is a multiple of } 3\}$.

(d) $L = \{a^n b^m : n \geq 1, n \neq m\}$.

(e) $L = \{w : n_a(w) = n_b(w)\}$.

(f) $L = \{a^n b^m a^{n+m} : n \geq 0, m \geq 1\}$.

(g) $L = \{a^n b^n a^n b^n : n \geq 0\}$.

(h) $L = \{a^n b^{2n} : n \geq 1\}$.

For each problem, write out δ in complete detail, then check your answers by tracing several test examples.

**8.** Design a Turing machine that accepts the language

$$L = \{ww : w \in \{a,b\}^+\}.$$

**9.** Construct a Turing machine to compute the function

$$f(w) = w^R,$$

where $w \in \{0,1\}^+$.

**10.** Design a Turing machine that finds the middle of a string of even length. Specifically, if $w = a_1 a_2 \ldots a_n a_{n+1} \ldots a_{2n}$, with $a_i \in \Sigma$, the Turing machine should produce $\widehat{w} = a_1 a_2 \ldots a_n c a_{n+1} \ldots a_{2n}$, where

$c \in \Gamma - \Sigma$

**11.** Design Turing machines to compute the following functions for $x$ and $y$ positive integers represented in unary.

(a) $f(x) = 3x$.

(b) $f(x, y) = x - y, \quad x > y,$
$= 0, \quad x \leq y.$

(c) $f(x, y) = 2x + 3y$.

(d) $f(x) = \frac{x}{2}, \quad$ if $x$ is even,
$= \frac{x+1}{2}, \quad$ if $x$ is odd.

(e) $f(x) = x \bmod 5$.

(f) $f(x) = \lfloor \frac{x}{2} \rfloor$, where $\lfloor \frac{x}{2} \rfloor$ denotes the largest integer less than or equal to $\frac{x}{2}$.

**12.** Design a Turing machine with $\Gamma = \{0,1,\square\}$ that, when started on any cell containing a blank or a 1, will halt if and only if its tape has a 0 somewhere on it.

**13.** Write out a complete solution for Example 9.8.

**14.** Give the sequence of instantaneous descriptions that the Turing machine in Example 9.10 goes through when presented with the input 111. What happens when this machine is started with 110 on its tape?

**15.** Give convincing arguments that the Turing machine in Example 9.10 does in fact carry out the indicated computation.

**16.** Complete all the details in Example 9.11.

**17.** Suppose that in Example 9.9 we had decided to represent $x$ and $y$ in binary. Write a Turing machine program for doing the indicated computation in this representation.

**18.** Sketch how Example 9.9 could be solved if $x$ and $y$ were represented in decimal.

**19.** You may have noticed that all the examples in this section had only one final state. Is it generally true that for any Turing machine, there exists another one with only one final state that accepts the same language?

**20.** Definition 9.2 excludes the empty string from any language accepted by a Turing machine. Modify the definition so that languages that contain $\lambda$ may be accepted.

## 9.2 Combining Turing Machines for Complicated Tasks

We have shown explicitly how some important operations found in all computers can be done on a Turing machine. Since, in digital computers, such primitive operations are the building blocks for more complex instructions, let us see howthese basic operations can also be put together on a Turing machine. To demonstrate howTuring machines can be combined, we follow a practice common in

programming. We start with a high-level description, then refine it successively until the program is in the actual language with which we are working. We can describe Turing machines several ways at a high level; block diagrams or pseudocode are the two approaches we will use most frequently in subsequent discussions. In a block diagram, we encapsule computations in boxes whose function is described, but whose interior details are not shown. By using such boxes, we implicitly claim that they can actually be constructed. As a first example, we combine the machines in Examples 9.9 and 9.11.

## Example 9.12

Design a Turing machine that computes the function

$$f(x,y) = x + y \qquad \text{if } x \geq y,$$

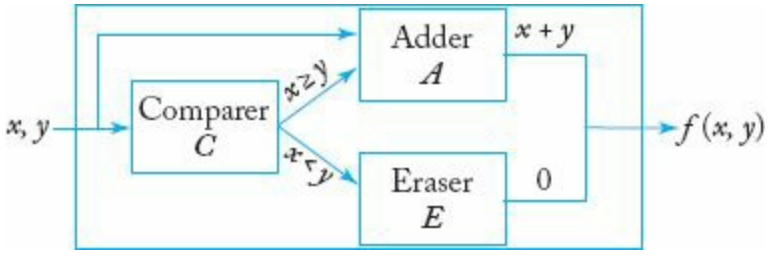
$$= 0 \text{ if} \qquad x < y.$$

For the sake of discussion, assume that $x$ and $y$ are positive integers in unary representation. The value zero will be represented by 0, with the rest of the tape blank.

The computation of $f(x, y)$ can be visualized at a high level by means of the diagram in Figure 9.8. The diagram shows that we first use a comparing machine, like that in Example 9.11, to determine whether or not $x \geq y$. If so, the comparer sends a start signal to the adder, which then computes $x + y$. If not, an erasing program is started that changes every 1 to a blank.

In subsequent discussions, we will often use such high-level, black-diagram representations of Turing machines. It is certainly quicker and clearer than the corresponding extensive set of δ's. Before we accept this high-level view, we must justify it. What, for example, is meant by saying that the comparer sends a start signal to the adder? There is nothing in Definition 9.1 that offers that possibility. Nevertheless, it can be done in a straightforward way.

**Figure 9.8**




The program for the comparer $C$ is written as suggested in Example 9.11, using a Turing machine having states indexed with $C$. For the adder, we use the idea in Example 9.9, with states indexed with A. For the eraser $E$, we construct a Turing machine having states indexed with $E$. The computations to be done by $C$ are

$$q_{C,0}w(x)\,0w(y) \overset{*}{\vdash} q_{A,0}w(x)\,0w(y) \qquad \text{if } x \geq y,$$

and

$$q_{C,0}w(x)\,0w(y) \overset{*}{\vdash} q_{E,0}w(x)\,0w(y) \qquad \text{if } x < y.$$

If we take $q_{A,0}$ and $q_{E,0}$ as the initial states of $A$ and E, respectively, we see that $C$ starts either $A$ or $E$.

The computations performed by the adder will be

$$q_{A,0}w(x)\,0w(y) \overset{*}{\vdash} q_{A,f}w(x+y)\,0,$$

and that of the eraser $E$ will be

$$q_{E,0}w(x)\,0w(y) \vdash q_{E,f}0.$$

The result is a single Turing machine that combines the action of $C$,$A$, and $E$ as indicated in Figure 9.8.

---

Another useful, high-level viewof Turing machines involves pseudocode. In computer programming, pseudocode is a way of outlining a computation using descriptive phrases whose meaning we claim to understand. While this description is not usable on the computer, we assume that we can translate it into the appropriate language when needed. One simple kind of pseudocode is exemplified by the idea of a macroinstruction, which is a single-statement shorthand for a sequence of lower-level statements. We first define the macroinstruction in terms of the lower-level language. We then use the macroinstruction in a program with the assumption that the relevant low-level code is substituted for each occurrence of the macroinstruction. This idea is very useful in Turing machine programming.

## Example 9.13

Consider the macroinstruction

$$\textit{if } a \text{ then } q_j \text{ else } q_k,$$

with the following interpretation. If the Turing machine reads an a, then regardless of its current state, it is to go into state $q_j$ without changing the tape content or moving the read-write head. If the symbol read is not an a, the machine is to go into state $q_k$ without changing anything.

To implement this macroinstruction requires several relatively obvious steps of a Turing machine.

$$\delta(q_i, a) = (q_{j0}, a, R) \quad \text{for all } q_i \in Q,$$

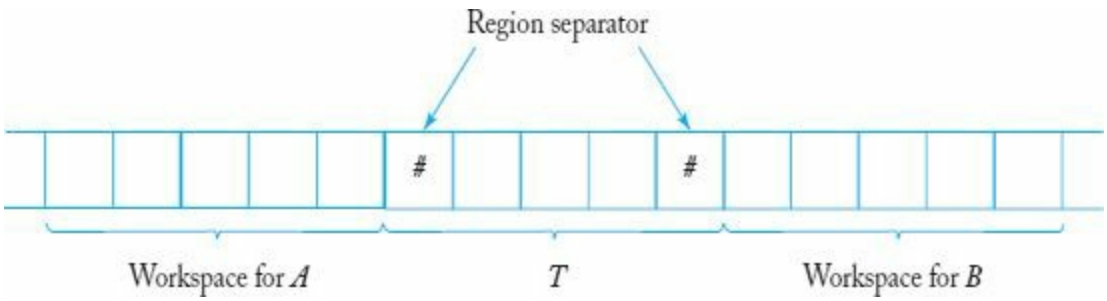
$$\begin{aligned}
\delta(q_i, b) &= (q_{k0}, b, R) && \text{for all } q_i \in Q \text{ and all } b \in \Gamma - \{a\}, \\
\delta(q_{j0}, c) &= (q_j, c, L) && \text{for all } c \in \Gamma, \\
\delta(q_{k0}, c) &= (q_k, c, L) && \text{for all } c \in \Gamma.
\end{aligned}$$

The states $q_{j0}$ and $q_{k0}$ are newstates, introduced to take care of complications arising from the fact that in a standard Turing machine the read-write head changes position in each move. In the

macroinstruction, we want to change the state, but leave the read-write head where it is. We let the head move right, but put the machine into a state $q_j$ or $q_{k0}$. This indicates that a left move must be made before entering the desired state $q_j$ or $q_k$.

Going a step further, we can replace macroinstructions with subprograms. Normally, a macroinstruction is replaced by actual code at each occurrence, whereas a subprogram is a single piece of code that is invoked repeatedly whenever needed. Subprograms are fundamental to high-level programming languages, but they can also be used with Turing machines. To make this plausible, let us outline briefly howa Turing machine can be used as a subprogram that can be invoked repeatedly by another Turing machine. This requires a newfeature: the ability to store information on the calling program's configuration so the configuration can be recreated on return from the subprogram. For example, say machine $A$ in state $q_i$ invokes machine B. When $B$ is finished, we would like to resume program $A$ in state $q_i$, with the read-write head (which may have moved during B's operation) in its original place. At other times, $A$ may call $B$ from state $q_j$, in which case control should return to this state. To solve the control transfer problem, we must be able to pass information from $A$ to $B$ and vice versa, be able to recreate $A$'s configuration when it recovers control from $B$, and assure that the temporarily suspended computations of $A$ are not affected by the execution of B. To solve this, we can divide the tape into several regions as shown in Figure 9.9.

**Figure 9.9**




Before $A$ calls $B$, it writes the information needed by $B$ (e.g., $A$'s current state, the arguments for $B$) on the tape in some region $T$. $A$ then passes control to $B$ by making a transition to the start state of B. After transfer, $B$ will use $T$ to find its input. The workspace for $B$ is separate from $T$ and from the workspace for A, so no interference can occur. When $B$ is finished, it will return relevant results to region $T$, where $A$ will expect to find it. In this way, the two programs can interact in the required fashion. Note that this is very similar to what actually happens in a real computer when a subprogram is called.

We can nowprogram Turing machines in pseudocode, provided that we know(in theory at least) howto translate this pseudocode into an actual Turing machine program.

**Example 9.14**

Design a Turing machine that multiplies two positive integers in unary notation.

A multiplication machine can be constructed by combining the ideas we encountered in adding and copying. Let us assume that the initial and final tape contents are to be as indicated in Figure 9.10.

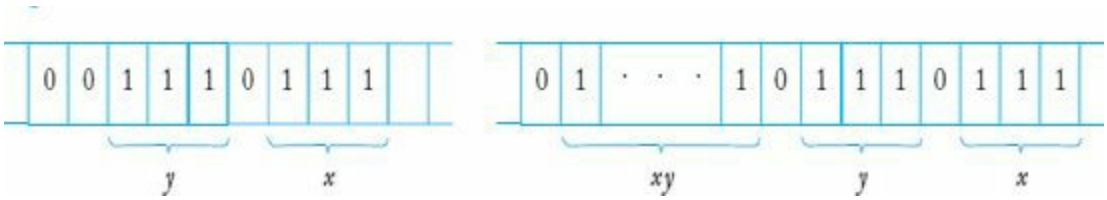
The process of multiplication can then be visualized as a repeated copying of the multiplicand $y$ for each 1 in the multiplier $x$, whereby the string $y$ is added the appropriate number of times to the partially computed product. The following pseudocode shows the main steps of the process.

**1.** Repeat the following steps until $x$ contains no more 1's. Find a 1 in $x$ and replace it with another symbol $a$. Replace the leftmost 0 by $0y$.

**2.** Replace all $a$'s with 1's.

Although this pseudocode is sketchy, the idea is simple enough that there should be no doubt that it can be done.

**Figure 9.10**




In spite of the descriptive nature of these examples, it is not too farfetched to conjecture that Turing machines, while rather primitive in principle, can be combined in many ways to make them quite powerful. Our examples were not general and detailed enough for us to claim that we have proved anything, but it should be plausible at this point that Turing machines can do some quite complicated things.

## EXERCISES

1. Write out the complete solution to Example 9.14.
2. Establish a convention for representing positive and negative integers in unary notation. With your convention, sketch the construction of a subtracter for computing $x - y$.
3. Using adders, subtracters, comparers, copiers, or multipliers, draw block diagrams for Turing machines that compute the functions

   (a) $f(n) = n(n+1)$,

   (b) $f(n) = n^5$,

   (c) $f(n) = 2^n$

   (d) $f(n) = n!$,

   (e) $f(n) = n^{n!}$,

   for all positive integers $n$.
4. Use a block diagram to sketch the implementation of a function $f$ defined for all $w_1, w_2, w_3 \in \{1\}^+$ by

$$f(w^1,w^2,w^3)= i,$$

where $i$ is such that $|w_i|= \max(|w_1|,|w_2|,|w_3|)$ if no two $w$'s have the same length, and $i = 0$ otherwise.

**5.** Provide a 'high-level' description for Turing machines that accept the following languages on $\{a, b\}$. For each problem, define a set of appropriate macroinstructions that you feel are reasonably easy to implement. Then use them for the solution.

(a) $L = \{ww_R\}$.

(b) $L = \{w_1w_2 : w_1 \neq w_2 : |w_1| = |w_2|\}$.

(c) The complement of the language in part (a).

(d) $L = \{a^n\ b^m : m = n^2, \geq 1\}$.

(e) $L = \{a^n : n \text{ is a prime number}\}$.

**6.** Suggest a method for representing rational numbers on a Turing machine, then sketch a method for adding and subtracting such numbers.

**7.** Sketch the construction of a Turing machine that can perform the addition and multiplication of positive integers $x$ and $y$ given in the usual decimal notation.

**8.** Give an implementation of the macroinstruction

$$\text{searchright}\ (a, q_i, q_j),$$

which indicates that the machine is to search its tape to the right of the current position for the first occurrence of the symbol $a$. If an $a$ is encountered before a blank, the machine is to go into state $q_i$, otherwise it is to go into state $q_j$.

**9.** Use the macroinstruction in the previous exercise to design a Turing machine on $\Sigma = \{a,b\}$ that accepts the language $L\ (ab^*ab^*a)$.

**10.** Use the macroinstruction searchright in Exercise 8 to create a Turing machine program that replaces the symbol immediately to the left of the leftmost $a$ by a blank. If the input contains no $a$, replace the rightmost nonblank symbol by $a\ b$.

## 9.3 Turing's Thesis

The preceding discussion not only shows how a Turing machine can be constructed from simpler parts, but also illustrates a negative aspect of working with such low-level automata. While it takes very little imagination or ingenuity to translate a block diagram or pseudocode into the corresponding Turing machine program, actually doing it is time-consuming, error-prone, and adds little to our understanding. The instruction set of a Turing machine is so restricted that any argument, solution, or proof for a nontrivial problem is quite tedious.

We nowface a dilemma: We want to claim that Turing machines can perform not only the simple operations for which we have provided explicit programs, but also more complex processes as well,

describable by block diagrams or pseudocode. To defend such claims against challenge, we should showthe relevant programs explicitly. But doing so is unpleasant and distracting, and ought to be avoided if possible. Somehow, we would like to find a way of carrying out a reasonably rigorous discussion of Turing machines without having to write lengthy, low-level code. There is unfortunately no completely satisfactory way of getting out of the predicament; the best we can do is to reach a reasonable compromise. To see how we might achieve such a compromise, we turn to a somewhat philosophical issue.

We can drawsome simple conclusions from the examples in the previous section. The first is that Turing machines appear to be more powerful than pushdown automata (for a comment on this, see Exercise 2 at the end of this section). In Example 9.8, we sketched the construction of a Turing machine for a language that is not context-free and for which, consequently, no pushdown automaton exists. Examples 9.9, 9.10, and 9.11 show that Turing machines can do some simple arithmetic operations, perform string manipulations, and make some simple comparisons. The discussion also illustrates how primitive operations can be combined to solve more complex problems, how several Turing machines can be composed, and how one program can act as a subprogram for another. Since very complex operations can be built this way, we might suspect that a Turing machine begins to approach a typical computer in power.

Suppose we were to make the conjecture that, in some sense, Turing machines are equal in power to a typical digital computer? How could we defend or refute such a hypothesis? To defend it, we could take a sequence of increasingly more difficult problems and show how they are solved by some Turing machine. We might also take the machine language instruction set of a specific computer and design a Turing machine that can perform all the instructions in the set. This would undoubtedly tax our patience, but it ought to be possible in principle if our hypothesis is correct. Still, while every success in this direction would strengthen our conviction of the truth of the hypothesis, it would not lead to a proof. The difficulty lies in the fact that we don't know exactly what is meant by “a typical digital computer” and that we have no means for making a precise definition.

We can also approach the problem from the other side. We might try to find some procedure for which we can write a computer program, but for which we can show that no Turing machine can exist. If this were possible, we would have a basis for rejecting the hypothesis. But no one has yet been able to produce a counterexample; the fact that all such tries have been unsuccessful must be taken as circumstantial evidence that it cannot be done. Every indication is that Turing machines are in principle as powerful as any computer.

Arguments of this type led A. M. Turing and others in the mid-1930s to the celebrated conjecture called the **Turing thesis**. This hypothesis states that any computation that can be carried out by mechanical means can be performed by some Turing machine.

This is a sweeping statement, so it is important to keep in mind what Turing's thesis is. It is not something that can be proved. To do so, we would have to define precisely the term “mechanical means.” This would require some other abstract model and leave us no further ahead than before. The Turing thesis is more properly viewed as a definition of what constitutes a mechanical computation: A computation is mechanical if and only if it can be performed by some Turing machine.

If we take this attitude and regard the Turing thesis simply as a definition, we raise the question as to whether this definition is sufficiently broad. Is it far-reaching enough to cover everything we now do (and conceivably might do in the future) with computers? An unequivocal “yes” is not possible, but the evidence in its favor is very strong. Some arguments for accepting the Turing thesis as the

definition of a mechanical computation are

1. Anything that can be done on any existing digital computer can also be done by a Turing machine.
2. No one has yet been able to suggest a problem, solvable by what we intuitively consider an algorithm, for which a Turing machine program cannot be written.
3. Alternative models have been proposed for mechanical computation, but none of them is more powerful than the Turing machine model.

These arguments are circumstantial, and Turing's thesis cannot be proved by them. In spite of its plausibility, Turing's thesis is still an assumption. But viewing Turing's thesis simply as an arbitrary definition misses an important point. In some sense, Turing's thesis plays the same role in computer science as do the basis laws of physics and chemistry. Classical physics, for example, is based largely on Newton's laws of motion. Although we call them laws, they do not have logical necessity; rather, they are plausible models that explain much of the physical world. We accept them because the conclusions we draw from them agree with our experience and our observations. Such laws cannot be proved to be true, although they can possibly be invalidated. If an experimental result contradicts a conclusion based on the laws, we might begin to question their validity. On the other hand, repeated failure to invalidate a lawstrengthens our confidence in it. This is the situation for Turing's thesis, so we have some reason for considering it a basic lawof computer science. The conclusions we draw from it agree with what we know about real computers, and so far, all attempts to invalidate it have failed. There is always the possibility that someone will come up with another definition that will account for some subtle situations not covered by Turing machines but which still fall within the range of our intuitive notion of mechanical computation. In such an eventuality, some of our subsequent discussions would have to be modified significantly. However, the likelihood of this happening seems to be very small.

Having accepted Turing's thesis, we are in a position to give a precise definition of an algorithm.

**Definition 9.5**

An **algorithm** for a function $f : D \rightarrow R$ is a Turing machine $M$, which given as input any $d \in D$ on its tape, eventually halts with the correct answer $f(d) \in R$ on its tape. Specifically, we can require that

$$q_0 d \overset{*}{\vdash}_M q_f f(d), q_f \in F,$$

for all $d \in D$.

Identifying an algorithm with a Turing machine program allows us to prove rigorously such claims as “there exists an algorithm…” or “there is no algorithm….” However, to construct explicitly an algorithm for even relatively simple problems is a very lengthy undertaking. To avoid such unpleasant prospects, we can appeal to Turing's thesis and claim that anything we can do on any computer can also be done on a Turing machine. Consequently, we could substitute “C program” for “Turing machine” in Definition 9.5. This would ease the burden of exhibiting algorithms

considerably. Actually, as we have already done, we will go one step further and accept verbal descriptions or block diagrams as algorithms on the assumption that we could write a Turing machine program for them if we were challenged to do so. This greatly simplifies the discussion, but it obviously leaves us open to criticism. While "C program" is well defined, "clear verbal description" is not, and we are in danger of claiming the existence of nonexistent algorithms. But this danger is more than offset by the facts that we can keep the discussion simple and intuitively clear and that we can give concise descriptions for some rather complex processes. The reader who has any doubts about the validity of these claims can dispel them by writing a suitable program in some programming language.

## EXERCISES

** **1.** Consider the set of machine language instructions for a computer of your choice. Sketch how the various instructions in this set could be carried out by a Turing machine.

**2.** In the above discussion, we stated at one point that Turing machines appear to be more powerful than pushdown automata. Since the tape of a Turing machine can always be made to behave like a stack, it would seem that we can actually claim that a Turing machine is more powerful. What important factor is not taken into account in this argument?

** **3.** There are a number of enjoyable articles on Turing machines in the popular literature. A good one is a paper in *Scientific American*, May 1984, by J. E. Hopcroft, titled "Turing Machines". This paper talks about the ideas we have introduced here and also gives some of the historical context in which the work of Turing and others was done. Get a copy of this article and read it, then write a brief review of it.