3

THE CHURCH–TURING THESIS

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 TURING MACHINES

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.

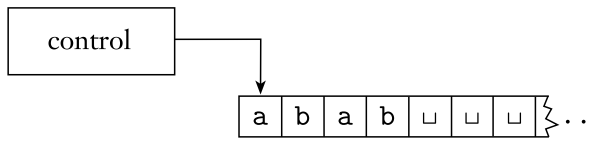


FIGURE 3.1
Schematic of a Turing machine

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 $B = \{w\#w|\ w \in \{0,1\}^*\}$. 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.

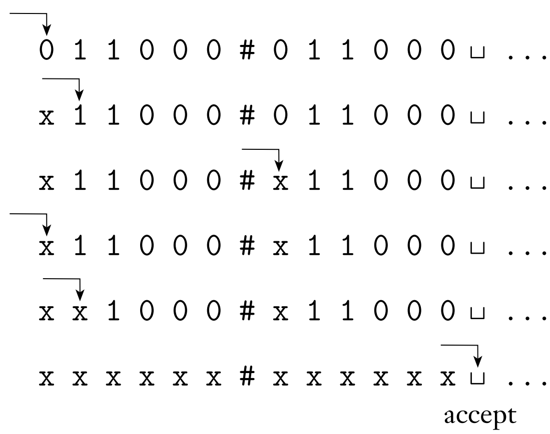


FIGURE 3.2
Snapshots of Turing machine M_1 computing on input 011000#011000

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 pushdown 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: $Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$. 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 $\delta(q, a) = (r, b, \mathrm{L})$, 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. $\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ 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 $w = w_1 w_2 \ldots w_n \in \Sigma^*$ 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 $u\,q\,v$ 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, 1011$q_7$01111 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.

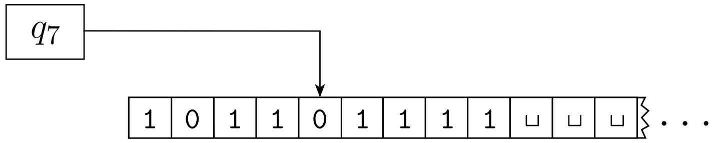


FIGURE 3.4
A Turing machine with configuration $\texttt{1011}q_7\texttt{01111}$

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, $ua\, q_i\, bv$ and $u\, q_j\, acv$ are two configurations. Say that

$$ua\, q_i\, bv \quad \text{yields} \quad u\, q_j\, acv$$

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

$$ua\, q_i\, bv \quad \text{yields} \quad uac\, q_j\, v$$

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

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 $c\, q_j v$ for the right-moving transition. For the right-hand end, the configuration $ua\, q_i$ is equivalent to $ua\, q_i\, \sqcup$ 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 $\delta\colon Q' \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$, where Q' is Q without q_{accept} and q_{reject}. A Turing machine M ***accepts*** input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ 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

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

[2]It 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 $A = \{\texttt{0}^{2^n} \mid n \geq 0\}$, 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 $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$:

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{\texttt{0}\}$, and
- $\Gamma = \{\texttt{0},\texttt{x},\sqcup\}$.
- 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.

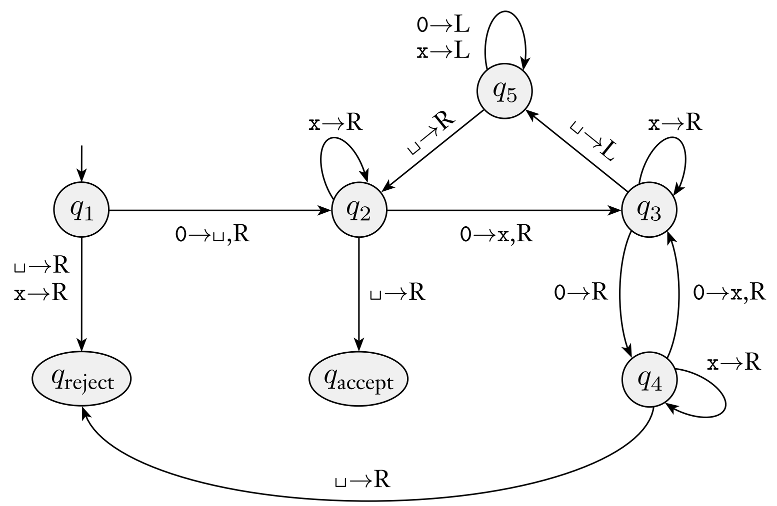


FIGURE 3.8
State diagram for Turing machine M_2

In this state diagram, the label 0→⊔,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 ⊔, 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→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.

$q_1\texttt{0000}$	$\sqcup q_5\texttt{x0x}\sqcup$	$\sqcup\texttt{x}q_5\texttt{xx}\sqcup$
$\sqcup q_2\texttt{000}$	$q_5\sqcup\texttt{x0x}\sqcup$	$\sqcup q_5\texttt{xxx}\sqcup$
$\sqcup\texttt{x}q_3\texttt{00}$	$\sqcup q_2\texttt{x0x}\sqcup$	$q_5\sqcup\texttt{xxx}\sqcup$
$\sqcup\texttt{x0}q_4\texttt{0}$	$\sqcup\texttt{x}q_2\texttt{0x}\sqcup$	$\sqcup q_2\texttt{xxx}\sqcup$
$\sqcup\texttt{x0x}q_3\sqcup$	$\sqcup\texttt{xx}q_3\texttt{x}\sqcup$	$\sqcup\texttt{x}q_2\texttt{xx}\sqcup$
$\sqcup\texttt{x0}q_5\texttt{x}\sqcup$	$\sqcup\texttt{xxx}q_3\sqcup$	$\sqcup\texttt{xx}q_2\texttt{x}\sqcup$
$\sqcup\texttt{x}q_5\texttt{0x}\sqcup$	$\sqcup\texttt{xx}q_5\texttt{x}\sqcup$	$\sqcup\texttt{xxx}q_2\sqcup$
		$\sqcup\texttt{xxx}\sqcup q_{\text{accept}}$

■

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 $B = \{w\#w|\ w \in \{0,1\}^*\}$.

- $Q = \{q_1, \ldots, q_8, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0,1,\#\}$, 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.

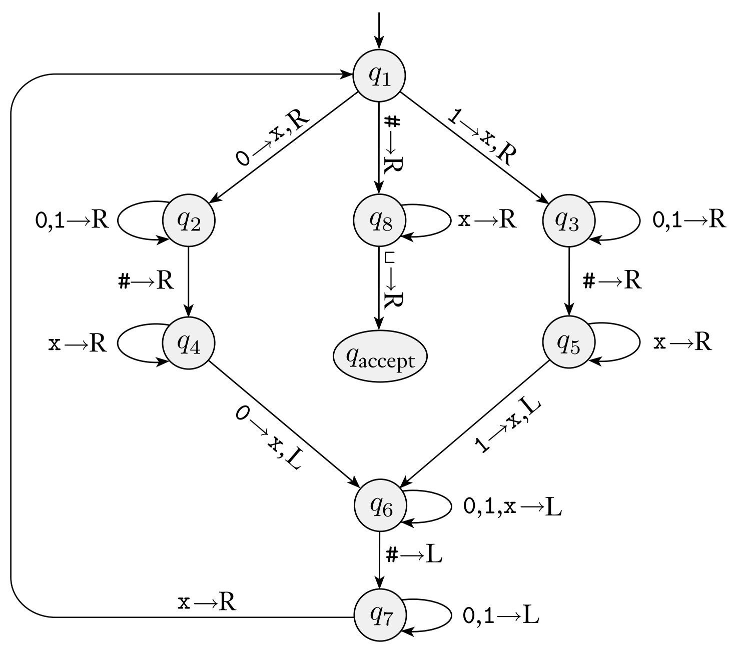


FIGURE 3.10
State diagram for Turing machine M_1

In Figure 3.10, which depicts the state diagram of TM M_1, you will find the label 0,1→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 $C = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k |\ i \times j = k \text{ and } i, j, k \geq 1\}$.

M_3 = "On input string w:

1. Scan the input from left to right to determine whether it is a member of $\mathtt{a}^+\mathtt{b}^+\mathtt{c}^+$ 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 $\dot{\#}$, in its tape alphabet. Saying that the machine places a mark above a # means that the machine writes the symbol $\dot{\#}$ 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 $\delta\colon Q\times\Gamma\longrightarrow Q\times\Gamma\times\{\mathrm{L},\mathrm{R},\mathrm{S}\}$. 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

$$\delta\colon Q\times\Gamma^k\longrightarrow Q\times\Gamma^k\times\{\mathrm{L},\mathrm{R},\mathrm{S}\}^k,$$

where k is the number of tapes. The expression

$$\delta(q_i,a_1,\ldots,a_k)=(q_j,b_1,\ldots,b_k,\mathrm{L},\mathrm{R},\ldots,\mathrm{L})$$

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.

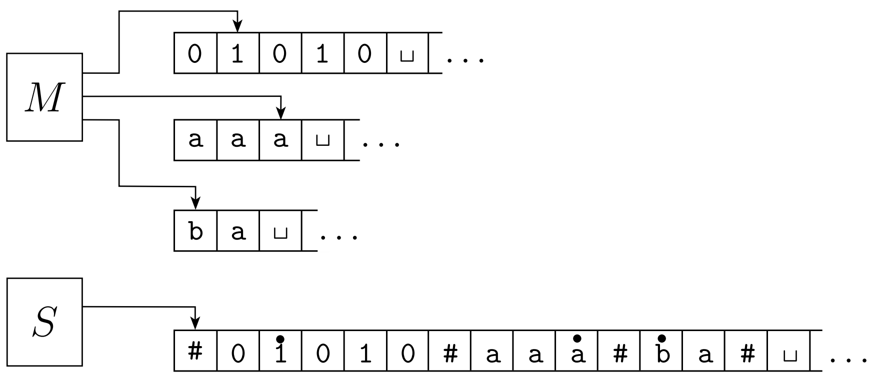


FIGURE 3.14
Representing three tapes with one

$S =$ "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

$$\#\dot{w}_1 w_2 \cdots w_n \#\dot{\sqcup}\#\dot{\sqcup}\# \cdots \#.$$

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

$$\delta\colon Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}).$$

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.

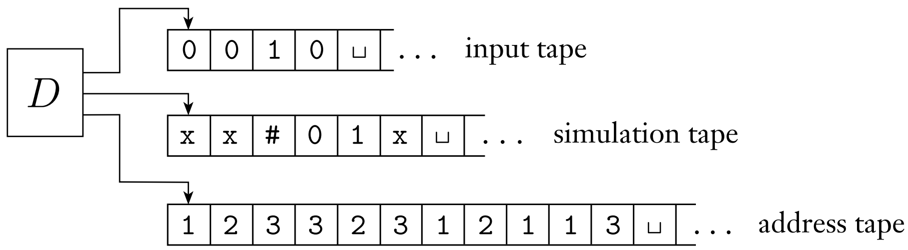


FIGURE 3.17
Deterministic TM D simulating nondeterministic TM N

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 $\Gamma_b = \{1, 2, \ldots, b\}$. 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.

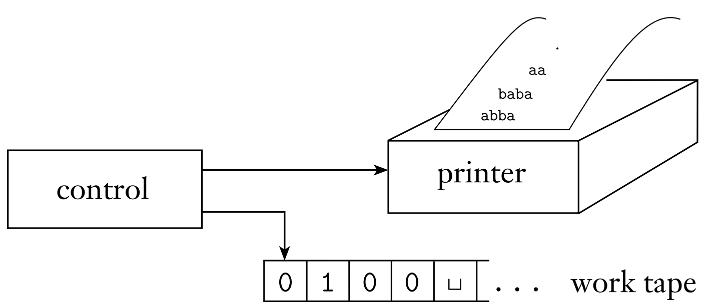


FIGURE 3.20
Schematic of an enumerator

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 $s_1, s_2, s_3, \ldots$ is a list of all possible strings in Σ^*.

$E =$ "Ignore the input.
1. Repeat the following for $i = 1, 2, 3, \ldots$.
2. Run M for i steps on each input, $s_1, s_2, \ldots, s_i$.
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]

[3]For 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.

Intuitive notion of algorithms	equals	*Turing machine algorithms*

FIGURE 3.22
The Church–Turing thesis

[4]Translated 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 $0, 1, -1, 2, -2, 3, -3, \ldots$. 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 $O_1, O_2, \ldots, O_k$, we denote their encoding into a single string $\langle O_1, O_2, \ldots, O_k \rangle$. 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

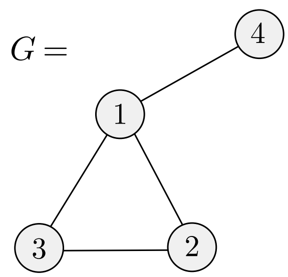
Let A be the language consisting of all strings representing undirected graphs that are connected. Recall that a graph is ***connected*** if every node can be reached from every other node by traveling along the edges of the graph. We write

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

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.



$\langle G \rangle =$

`(1,2,3,4)((1,2),(2,3),(3,1),(1,4))`

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.

Now M scans the list of edges. For each edge, M tests whether the two underlined nodes n_1 and n_2 are the ones appearing in that edge. If they are, M dots n_1, removes the underlines, and goes on from the beginning of stage 2. If they aren't, M checks the next edge on the list. If there are no more edges, $\{n_1, n_2\}$ is not an edge of G. Then M moves the underline on n_2 to the next dotted node and now calls this node n_2. It repeats the steps in this paragraph to check, as before, whether the new pair $\{n_1, n_2\}$ is an edge. If there are no more dotted nodes, n_1 is not attached to any dotted nodes. Then M sets the underlines so that n_1 is the next undotted node and n_2 is the first dotted node and repeats the steps in this paragraph. If there are no more undotted nodes, M has not been able to find any new nodes to dot, so it moves on to stage 4.

For stage 4, M scans the list of nodes to determine whether all are dotted. If they are, it enters the accept state; otherwise, it enters the reject state. This completes the description of TM M. ▪

EXERCISES

3.1 This exercise concerns TM M_2, whose description and state diagram appear in Example 3.7. In each of the parts, give the sequence of configurations that M_2 enters when started on the indicated input string.

a. `0`.
[A]**b.** `00`.
c. `000`.
d. `000000`.

3.2 This exercise concerns TM M_1, whose description and state diagram appear in Example 3.9. In each of the parts, give the sequence of configurations that M_1 enters when started on the indicated input string.

[A]**a.** `11`.
b. `1#1`.
c. `1##1`.
d. `10#11`.
e. `10#10`.

[A]**3.3** Modify the proof of Theorem 3.16 to obtain Corollary 3.19, showing that a language is decidable iff some nondeterministic Turing machine decides it. (You may assume the following theorem about trees. If every node in a tree has finitely many children and every branch of the tree has finitely many nodes, the tree itself has finitely many nodes.)

3.4 Give a formal definition of an enumerator. Consider it to be a type of two-tape Turing machine that uses its second tape as the printer. Include a definition of the enumerated language.

[A]**3.5** Examine the formal definition of a Turing machine to answer the following questions, and explain your reasoning.

- **a.** Can a Turing machine ever write the blank symbol $\sqcup$ on its tape?
- **b.** Can the tape alphabet Γ be the same as the input alphabet Σ?
- **c.** Can a Turing machine's head *ever* be in the same location in two successive steps?
- **d.** 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, $s_1, s_2, \ldots$ is a list of all strings in Σ^*.

$E =$ "Ignore the input.
1. Repeat the following for $i = 1, 2, 3, \ldots$.
2. Run M on s_i.
3. 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 $x_1, \ldots, x_k$:
1. Try all possible settings of $x_1, \ldots, x_k$ to integer values.
2. Evaluate p on all of these settings.
3. 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}.

- [A]**a.** $\{w \mid w$ contains an equal number of 0s and 1s$\}$
- **b.** $\{w \mid w$ contains twice as many 0s as 1s$\}$
- **c.** $\{w \mid w$ 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.

- **a.** Show that 2-PDAs are more powerful than 1-PDAs.
- **b.** Show that 3-PDAs are not more powerful than 2-PDAs. (Hint: Simulate a Turing machine tape with two stacks.)

[A]**3.10** 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

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

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

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

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

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

- [A]**a.** union.
- **b.** concatenation.
- **c.** star.
- **d.** intersection.
- **e.** homomorphism.

***3.17** Let $B = \{\langle M_1\rangle, \langle M_2\rangle, \ldots\}$ 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

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

A3.22 Let A be the language containing only the single string s, where

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

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)** $q_1\texttt{00}, \sqcup q_2\texttt{0}, \sqcup\texttt{x}q_3\sqcup, \sqcup q_5\texttt{x}\sqcup, q_5\sqcup\texttt{x}\sqcup, \sqcup q_2\texttt{x}\sqcup, \sqcup\texttt{x}q_2\sqcup, \sqcup\texttt{x}\sqcup q_{\text{accept}}$.

3.2 **(a)** $q_1\texttt{11}, \texttt{x}q_3\texttt{1}, \texttt{x1}q_3\sqcup, \texttt{x1}\sqcup q_{\text{reject}}$.

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 `1`s 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.