Department of Mathematics and Computer Science

Computation Theory problems

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.

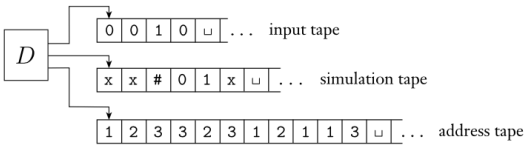


FIGURE 3.17
Deterministic TM D simulating nondeterministic TM N

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.

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.

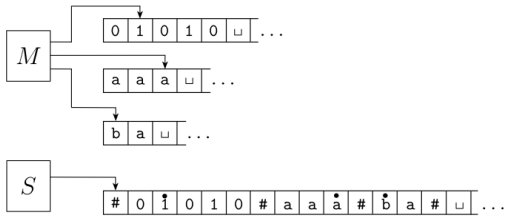


FIGURE 3.14
Representing three tapes with one

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."

1.

2. Yes, we can encode Turing machines.

Every Turing machine has some description in some finite alpabet, for example, in english. We can encode every english letter with an 8-bit binary string. Therefore every Turing machine has a binary encoding. From here it is easy to see that we also can create encodings to natural numbers.

Many approaches can be used to introduce a computable encoding for TMs. A simple one can be the compiler from any Turing-complete programming language (if we assume the Church-Turing thesis) for instance, C or Assembly.

It is possible to create T-computable encodings without Church-Turing assumption and purly

from the definition of the Turing machine. To do this, we have to construct a TM that assigns a natural number (or any other encoding) to the transition function of any given TM. However this can be extremly difficult, so we use the Church-Turing assumption.

An enumerator is a 7-tuple, $(Q,\ \Sigma,\ \Gamma,\ \delta,\ q_0,\ q_{print},\ q_{reject})$, where $Q,\ \Sigma,\ \Gamma$ are all finite sets and:

1. Q is the set of states,
2. Σ is the second tape alphabet (i.e. printer alphabet).
3. Γ is the first tape alphabet (i.e. working tape alphabet).
4. $\delta:\ Q \times \Gamma \to Q \times \Gamma \times \{L, R\} \times \Sigma_\epsilon$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{print} \in Q$ is the accept/print state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{print}$.

The above definition is different from the Turing machine definition in:

1. q_{accept} is now q_{print} since enumerators print every string they accept.
2. Σ used to be the input alphabet but since enumerators have no inputs we use it for the printer alphabet.
3. Σ_ϵ is added to the right side of the transition function since enumerators should print every string they accept. So it starts with an empty second tape and then each time it writes a symbol $\sigma \in \Sigma_\epsilon$ on the second tape (second tape is write only) and moves the head to the right. When the machine goes to the q_{print}, it halts and the string on the printer tape is the accepted string.

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.

3.

4. Firstly, the the sentence "Evaluate p over all possible settings of $x_1, x_2, ..., x_k$" is not trivialy translatable into formal Turing machine description (for example transition function). But it is possible to make this statement more accurate to be acceptable.
Secondly, it is instructed that M must reject in case p doesn't evaluate to 0 for any setting of $x_1, x_2, ..., x_k$. However, this is not an acceptable instruction for a Turing machine because this is not a finite step; M can never be sure that p will not evaluate to 0 for any setting of $x_1, x_2, ..., x_k$.
Note that being determinestic or non-determinestic has no effect on the validity of this Turing machine description. However, it can be easier to resolve the first problem in a non-determinestic environment.
We know that there is no decider for integer polynomial roots (Diophantine equations) [see MRDP theorem and Hilbert's tenth problem], but we can easily create a recognizer for this problem by removing the part about rejecting from M's description.

a. To show 2-PDAs are more powerful than 1-PDAs, we need to find a language L that is recognized by a 2-PDA and not by a 1-PDA.

Let $L = \left\{a^n b^n c^n \mid n \geqslant 0\right\}$.

Using pumping lemma (as stated in *Example 2.36*) we see that L is not a CFL and therefore no conventional PDA (i.e. 1-PDA) recognize this language.

Now we give an informal description of a 2-PDA that recognizes L.

1. Start by reading a's and for each a push it into the *stack1* and *stack2*.

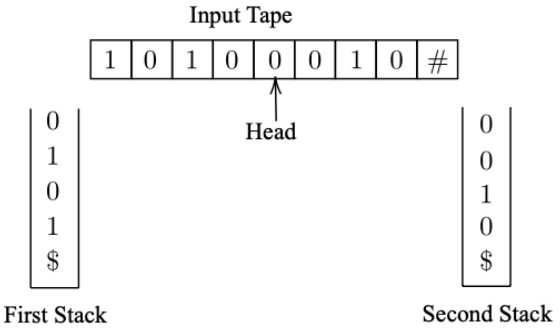
(We assume the end of stack marker $\$$ is already in both stacks.)

2. When the a's are over start reading b's and for each b pop *stack1*. If any a's appear or *stack1* becomes empty, *reject*.
3. When the b's are over start reading c's and for each c pop *stack2*. If any a's or b's appear or *stack2* becomes empty, *reject*.
4. Finally check if both *stack1* and *stack2* are empty (i.e. the top of the stack is the $\$$.) if so *accept*, otherwise *reject*.

Therefore 2-PDAs are more powerful than 1-PDAs. ✓

b. Since the most powerful FSM(finite state machine) is a Turing machine, we simulate a Turing machine tape with two stacks and show that a Turing machine is equivalent in power to a 2-PDA and thus 3-PDAs cannot be more powerful than 2-PDAs.

Proof idea: The idea is that the first stack can hold what is to the left of the head, while the second stack holds what is to the right of the head. When we want to get a symbol which is in the first stack, we will pop all the content in the first stack above that symbol and push these content in the second stack. In this way we can access any symbol stored in the stack without losing any content. This allows to simulate the tape of Turing machine M, where the M's head is corresponding to the heads of the two stacks of the 2-PDA.



Detailed proof: Let L be $L(M)$ for some one tape Turing machine M and 2-PDA S. Then S simulates M as the following:

Copying the input tape into our two stacks with respect to the above restriction. (i.e. S simulating the first state of M):

1. S begins with a bottom-of-stack marker on each stack, this marker is considered the start symbol for the stacks, and must not appear elsewhere on the stacks. The marker indicates that the stack is empty.

($\$$ is used in the above example)

2. Suppose that $w\#$ is on the input of M. S copies the input w onto its first stack, and stops when reading the endmarker on the input.
3. S pops each symbol from its first stack and pushes it onto its second stack.

The first stack of S is empty. The second stack holds w, with the left end of w is at the top.

4. The empty first stack indicates the fact that M has a blank to the left of cell scanned by the tape. S has a second stack holding w indicates the head points to the left most symbol in the string w.

Note that S knows the state q of M, because S simulates the state of M in its own finite control. Also S knows the symbol X scanned by the head of M, because it's at the top of the second stack. If the second stack contains only the bottom of stack marker, this means that M's head has just scanned a blank. Thus, S knows the next move of M. The next state of M is recorded in a component of S's finite control, instead of the previous state.

Simulating the Turing machine moves (moving left/right, reading, editing):

1) Moving left: Pop from the first stack and push it to the second one.

2) Moving right: Pop from the second stack and push it to the first one.

3) Reading: Return the top of the second stack

4) Editing: To replace X by Y, pop X from the top of the second stack then push Y into it.

S accepts if the new state of M is accepting. Otherwise, S simulates another move of M in the same manner.

And the proof is complete.

As a matter of fact in terms of power we have:

$$1\text{-PDA} \prec 2\text{-PDA} = k\text{-PDA} = \text{Turing machine}$$

5.

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

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

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

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

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

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

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

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

THEOREM 4.2

A_{NFA} is a decidable language.

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

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

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

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

THEOREM 4.5

EQ_{DFA} is a decidable language.

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

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

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

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

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

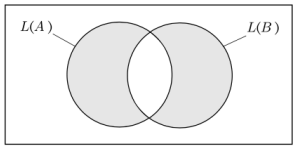


FIGURE 4.6
The symmetric difference of $L(A)$ and $L(B)$

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

THEOREM 4.3

A_{REX} is a decidable language.

PROOF The following TM P decides A_{REX}.

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

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

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

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

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

THEOREM 4.4

E_{DFA} is a decidable language.

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

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

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

THEOREM 4.7

A_{CFG} is a decidable language.

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

To make this Turing machine into a decider, we need to ensure that the algorithm tries only finitely many derivations. In Problem 2.38 (page 158) we showed that if G were in Chomsky normal form, any derivation of w has $2n - 1$ steps, where n is the length of w. In that case, checking only derivations with $2n - 1$ steps to determine whether G generates w would be sufficient. Only finitely many such derivations exist. We can convert G to Chomsky normal form by using the procedure given in Section 2.1.

PROOF The TM S for A_{CFG} follows.

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

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

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

6.

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

AN UNDECIDABLE LANGUAGE

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\cdots$
M_1	*accept*		*accept*		
M_2	*accept*	*accept*	*accept*	*accept*	
M_3					$\cdots$
M_4	*accept*	*accept*			
$\vdots$			$\vdots$		

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

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

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*	
M_4	*accept*	*accept*	*reject*	*reject*	
$\vdots$			$\vdots$		

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

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

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\cdots$	$\langle D\rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	*reject*		*accept*	
$\vdots$			$\vdots$		$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		?	
$\vdots$			$\vdots$				$\ddots$

FIGURE 4.21
If D is in the figure, a contradiction occurs at "?"

7.

A TURING-UNRECOGNIZABLE LANGUAGE

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

THEOREM 4.22

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

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

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

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

M = "On input w:

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

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

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

COROLLARY 4.23

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

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

8.

Some languages are not Turing-recognizable.

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

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

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

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

$$\begin{array}{rccccccccccc} \Sigma^* = \{ & \varepsilon, & 0, & 1, & 00, & 01, & 10, & 11, & 000, & 001, & \cdots & \} ; \\ A = \{ & & 0, & & 00, & 01, & & & 000, & 001, & \cdots & \} ; \\ \chi_A = & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \cdots & . \end{array}$$

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

9. In the previous problem we proved that $A_T^- M$ is not Turing-recognizable.

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

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

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

10.

11.

12.

THEOREM 5.1

$HALT_{\mathsf{TM}}$ is undecidable.

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

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

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

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

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

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

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

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

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

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

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

THEOREM 5.2

E_{TM} is undecidable.

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

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

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

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

M_1 = "On input x:

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

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

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

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

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

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

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

13.

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

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

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

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

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

THEOREM 5.10

E_{LBA} is undecidable.

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

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

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

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

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

14.

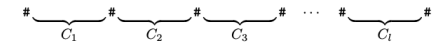


FIGURE 5.11
A possible input to B

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

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

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

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

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

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

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

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

5.2 A SIMPLE UNDECIDABLE PROBLEM

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

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

$$\left[\frac{\texttt{a}}{\texttt{ab}}\right]$$

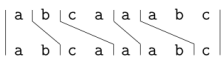
and a collection of dominos looks like

$$\left\{\left[\frac{\texttt{b}}{\texttt{ca}}\right], \left[\frac{\texttt{a}}{\texttt{ab}}\right], \left[\frac{\texttt{ca}}{\texttt{a}}\right], \left[\frac{\texttt{abc}}{\texttt{c}}\right]\right\}.$$

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

$$\left[\frac{\texttt{a}}{\texttt{ab}}\right]\left[\frac{\texttt{b}}{\texttt{ca}}\right]\left[\frac{\texttt{ca}}{\texttt{a}}\right]\left[\frac{\texttt{a}}{\texttt{ab}}\right]\left[\frac{\texttt{abc}}{\texttt{c}}\right]$$

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



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

$$\left\{\left[\frac{\texttt{abc}}{\texttt{ab}}\right], \left[\frac{\texttt{ca}}{\texttt{a}}\right], \left[\frac{\texttt{acc}}{\texttt{ba}}\right]\right\}$$

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

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

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

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

Part 1. The construction begins in the following manner.

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

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

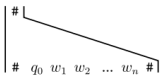


FIGURE 5.16
Beginning of the MPCP match

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

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

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

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

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

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

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

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

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

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

THEOREM 5.15

PCP is undecidable.

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

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

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

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

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

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

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

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

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

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

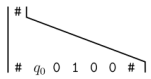
$$\text{put } \left[\frac{a}{a}\right] \text{ into } P'.$$

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

Part 1 places the domino

$$\left[\frac{\#}{\#q_0\texttt{0100}\#}\right] = \left[\frac{t_1}{b_1}\right]$$

in P', and the match begins



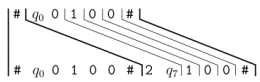
In addition, part 2 places the domino

$$\left[\frac{q_0\texttt{0}}{\texttt{2}q_7}\right]$$

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

$$\left[\frac{\texttt{0}}{\texttt{0}}\right], \left[\frac{\texttt{1}}{\texttt{1}}\right], \left[\frac{\texttt{2}}{\texttt{2}}\right], \text{ and } \left[\frac{\sqcup}{\sqcup}\right]$$

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



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

Part 5.

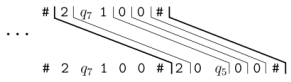
$$\text{Put } \left[\frac{\#}{\#}\right] \text{ and } \left[\frac{\#}{\sqcup\#}\right] \text{ into } P'.$$

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

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

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

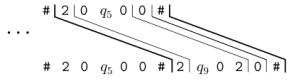
So the latest partial match extends to



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

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

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

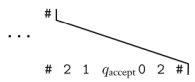


Note that as we construct a match, we are forced to simulate M on input w. This process continues until M reaches a halting state. If the accept state occurs, we want to let the top of the partial match "catch up" with the bottom so that the match is complete. We can arrange for that to happen by adding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

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

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

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



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

Part 7. Finally, we add the domino

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

and complete the match:

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

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

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

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

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

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

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

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

we let P be the collection

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

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

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

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

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

is there to allow the top to add the extra $*$ at the end of the match.

15.

THEOREM 5.30

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

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

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

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

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

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

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

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

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

16.

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

So J is basically saying that for every $w \in J$, if w starts with a 0 then there exists some TM M that accepts it and if it starts with a 1 then there is not any TM M that accepts it.

Note that from the proofs provided in the textbook, we know that $\overline{A_{TM}}$ is not Turing-recognizable.

I) J is not Turing-recognizable.

First we show that J is not Turing-recognizable. We do so by showing that $\overline{A_{TM}}$ is reducible to J. $\overline{A_{TM}} \leq_m J$. Since $\overline{A_{TM}}$ is not Turing-recognizable using theorem 5.28 we conclude that J is not Turing-recognizable.

Let $f(y) = 1y$ be the reduction.

The following TM F computes the reducing function f.

F = "On input y:

1. Output $1y$."

By definition a function $f : \Sigma^* \to \Sigma^*$ is a computable function if some Turing machine M, on every input y, halts with just $f(y)$ on its tape. Thus f is a computable function.

Now we show that for every y, $y \in \overline{A_{TM}} \iff f(y) \in J$.

Let $y \in \overline{A_{TM}}$ then $f(y) = 1y$. Since $y \in \overline{A_{TM}}$ then there is not any TM M that accepts

the string y and by definition of J, we get that $1y \in J \to f(y) \in J$.

Let $f(y) \in J \to 1y \in J$, by definition of J, we get that there is not any TM M that accepts the string y so $y \notin A_{TM} \to y \in \overline{A_{TM}}$.

So we conclude that f is a reduction from $\overline{A_{TM}}$ to J.

II) $\overline{J}$ is not Turing-recognizable.

To show that $\overline{J}$ is not Turing-recognizable, we give a reduction from A_{TM} to the complement of $\overline{J}$ —namely, J. Hence we show that $A_{TM} \leq_m J$. This implies that $\overline{A_{TM}} \leq_m \overline{J}$. Since $\overline{A_{TM}}$ is not Turing-recognizable using theorem 5.28 we conclude that $\overline{J}$ is not Turing-recognizable.

Let $g(x) = 0x$ be the reduction.

The following TM G computes the reducing function g.

G = "On input x:

1. Output $0x$."

By definition a function $g : \Sigma^* \to \Sigma^*$ is a computable function if some Turing machine M, on every input x, halts with just $g(x)$ on its tape. Thus g is a computable function.

Now we show that for every x, $x \in A_{TM} \iff g(x) \in J$.

Let $x \in A_{TM}$ then $g(x) = 0x$. Since $x \in A_{TM}$ then there exists a TM M that accepts the string x and by definition of J, we get that $0x \in J \to g(x) \in J$.

Let $g(x) \in J \to 0x \in J$, by definition of J, we get that there exists a TM M that accepts the string x so $x \in A_{TM}$.

So we conclude that g is a reduction from A_{TM} to J.

17.

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

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

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

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

18.

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

19.

20. Let A be a decidable Turing machine that for any input $3 \geq n \in 2\mathbb{N}$, checks whether n can be expressed as the sum of two prime numbers. We construct $G^{A_{TM}}$ as follows:
$G^{A_{TM}}$ = "On any input:

1. Construct the TM N as follows:
   N = "On any input:

   (a) Run A on all natural numbers.

   (b) If A rejects any number, *accept*.

2. Query $< N, \epsilon >$ from the oracle for A_{TM}.

3. If $< N, \epsilon > \in A_{TM}$, *reject*; *accept* other wise.

If $G^{A_{TM}}$ accepts, then Goldbach's conjecture is true, and it is false other wise.

21. M_0 = "On any input:

1. *reject*

M_1 = "On any input:

1. Write "1" on the tape, where the head is and don't move the head.

2. Write "blank_symbol" on the tape, where the head is and don't move the head.

3. *reject*

Let P be the set of TM descriptions with the given property.
$P = \{< M > \mid M$ is a TM and there exist some $x \in \Sigma^*$ such that, when M starts with x on its tape, in some stage of computation it writes the blank_symbol over some non-blank_symbol character on its tape. $\}$.
It is clear that $< M_0 > \in P$ and $< M_1> \notin P$. So from Rice's theorem, we can conclude that P is not decidable.

Now we give one final definition in preparation for the next section. If $\mathcal{M}$ is a model, we let the ***theory of*** $\mathcal{M}$, written $\mathrm{Th}(\mathcal{M})$, be the collection of true sentences in the language of that model.

A DECIDABLE THEORY

Number theory is one of the oldest branches of mathematics and also one of its most difficult. Many innocent-looking statements about the natural numbers with the plus and times operations have confounded mathematicians for centuries, such as the twin prime conjecture mentioned earlier.

In one of the celebrated developments in mathematical logic, Alonzo Church, building on the work of Kurt Gödel, showed that no algorithm can decide in general whether statements in number theory are true or false. Formally, we write $(\mathcal{N}, +, \times)$ to be the model whose universe is the natural numbers[4] with the usual $+$ and $\times$ relations. Church showed that $\mathrm{Th}(\mathcal{N}, +, \times)$, the theory of this model, is undecidable.

Before looking at this undecidable theory, let's examine one that is decidable. Let $(\mathcal{N}, +)$ be the same model, without the $\times$ relation. Its theory is $\mathrm{Th}(\mathcal{N}, +)$. For example, the formula $\forall x \exists y \, [x + x = y]$ is true and is therefore a member of $\mathrm{Th}(\mathcal{N}, +)$, but the formula $\exists y \forall x \, [x + x = y]$ is false and is therefore not a member.

THEOREM 6.12

$\mathrm{Th}(\mathcal{N}, +)$ is decidable.

PROOF IDEA This proof is an interesting and nontrivial application of the theory of finite automata that we presented in Chapter 1. One fact about finite automata that we use appears in Problem 1.37, (page 89) where you were asked to show that they are capable of doing addition if the input is presented in a special form. The input describes three numbers in parallel by representing one bit of each number in a single symbol from an eight-symbol alphabet. Here we use a generalization of this method to present i-tuples of numbers in parallel using an alphabet with 2^i symbols.

We give an algorithm that can determine whether its input, a sentence ϕ in the language of $(\mathcal{N}, +)$, is true in that model. Let

$$\phi = \mathsf{Q}_1 x_1 \, \mathsf{Q}_2 x_2 \cdots \mathsf{Q}_l x_l \, [\psi],$$

where $\mathsf{Q}_1, \ldots, \mathsf{Q}_l$ each represents either $\exists$ or $\forall$ and ψ is a formula without quantifiers that has variables $x_1, \ldots, x_l$. For each i from 0 to l, define formula ϕ_i as

$$\phi_i = \mathsf{Q}_{i+1} x_{i+1} \, \mathsf{Q}_{i+2} x_{i+2} \cdots \mathsf{Q}_l x_l \, [\psi].$$

Thus $\phi_0 = \phi$ and $\phi_l = \psi$.

[4]For convenience in this chapter, we change our usual definition of $\mathcal{N}$ to be $\{0, 1, 2, \ldots\}$.

Formula ϕ_i has i free variables. For $a_1, \ldots, a_i \in \mathcal{N}$, write $\phi_i(a_1, \ldots, a_i)$ to be the sentence obtained by substituting the constants $a_1, \ldots, a_i$ for the variables $x_1, \ldots, x_i$ in ϕ_i.

For each i from 0 to l, the algorithm constructs a finite automaton A_i that recognizes the collection of strings representing i-tuples of numbers that make ϕ_i true. The algorithm begins by constructing A_l directly, using a generalization of the method in the solution to Problem 1.37. Then, for each i from l down to 1, it uses A_i to construct A_{i-1}. Finally, once the algorithm has A_0, it tests whether A_0 accepts the empty string. If it does, ϕ is true and the algorithm accepts.

PROOF For $i > 0$, define the alphabet

$$\Sigma_i = \left\{ \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 1 \end{bmatrix}, \ldots, \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 1 \end{bmatrix} \right\}.$$

Hence Σ_i contains all size i columns of 0s and 1s. A string over Σ_i represents i binary integers (reading across the rows). We also define $\Sigma_0 = \{[]\}$, where $[]$ is a symbol.

We now present an algorithm that decides $\mathrm{Th}(\mathcal{N}, +)$. On input ϕ, where ϕ is a sentence, the algorithm operates as follows. Write ϕ and define ϕ_i for each i from 0 to l, as in the proof idea. For each such i, construct a finite automaton A_i from ϕ_i that accepts strings over Σ_i corresponding to i-tuples $a_1, \ldots, a_i$ whenever $\phi_i(a_1, \ldots, a_i)$ is true, as follows.

To construct the first machine A_l, observe that $\phi_l = \psi$ is a Boolean combination of atomic formulas. An atomic formula in the language of $\mathrm{Th}(\mathcal{N}, +)$ is a single addition. Finite automata can be constructed to compute any of these individual relations corresponding to a single addition and then combined to give the automaton A_l. Doing so involves the use of the regular language closure constructions for union, intersection, and complementation to compute Boolean combinations of the atomic formulas.

Next, we show how to construct A_i from A_{i+1}. If $\phi_i = \exists x_{i+1} \, \phi_{i+1}$, we construct A_i to operate as A_{i+1} operates, except that it nondeterministically guesses the value of a_{i+1} instead of receiving it as part of the input.

More precisely, A_i contains a state for each A_{i+1} state and a new start state. Every time A_i reads a symbol

$$\begin{bmatrix} b_1 \\ \vdots \\ b_{i-1} \\ b_i \end{bmatrix},$$

where every $b_j \in \{0,1\}$ is a bit of the number a_j, it nondeterministically guesses $z \in \{0,1\}$ and simulates A_{i+1} on the input symbol

$$\begin{bmatrix} b_1 \\ \vdots \\ b_{i-1} \\ b_i \\ z \end{bmatrix}.$$

Initially, A_i nondeterministically guesses the leading bits of a_{i+1} corresponding to suppressed leading 0s in a_1 through a_i by nondeterministically branching using ε-transitions from its new start state to all states that A_{i+1} could reach from its start state with input strings of the symbols

$$\left\{ \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \right\}$$

in Σ_{i+1}. Clearly, A_i accepts its input $(a_1, \ldots, a_i)$ if some a_{i+1} exists where A_{i+1} accepts $(a_1, \ldots, a_{i+1})$.

If $\phi_i = \forall x_{i+1} \, \phi_{i+1}$, it is equivalent to $\neg \exists x_{i+1} \neg \, \phi_{i+1}$. Thus, we can construct the finite automaton that recognizes the complement of the language of A_{i+1}, then apply the preceding construction for the $\exists$ quantifier, and finally apply complementation once again to obtain A_i.

Finite automaton A_0 accepts any input iff ϕ_0 is true. So the final step of the algorithm tests whether A_0 accepts ε. If it does, ϕ is true and the algorithm accepts; otherwise, it rejects.

6.28 Reduce $\mathrm{Th}(\mathcal{N},<)$ to $\mathrm{Th}(\mathcal{N},+)$, which we've already shown to be decidable. Show how to convert a sentence ϕ_1 over the language of $(\mathcal{N},<)$ to a sentence ϕ_2 over the language of $(\mathcal{N},+)$ while preserving truth or falsity in the respective models. Replace every occurrence of $i < j$ in ϕ_1 with the formula $\exists k\big[\,(i{+}k{=}j){\wedge}(k{+}k{\neq}k)\,\big]$ in ϕ_2, where k is a different new variable each time.

Sentence ϕ_2 is equivalent to ϕ_1 because "i is less than j" means that we can add a nonzero value to i and obtain j. Putting ϕ_2 into prenex-normal form, as required by the algorithm for deciding $\mathrm{Th}(\mathcal{N},+)$, requires a bit of additional work. The new existential quantifiers are brought to the front of the sentence. To do so, these quantifiers must pass through Boolean operations that appear in the sentence. Quantifiers can be brought through the operations of $\wedge$ and $\vee$ without change. Passing through $\neg$ changes $\exists$ to $\forall$ and vice versa. Thus, $\neg\exists k\,\psi$ becomes the equivalent expression $\forall k\,\neg\psi$, and $\neg\forall k\,\psi$ becomes $\exists k\,\neg\psi$.

22.

THEOREM 6.5

A_{TM} is undecidable.

PROOF We assume that Turing machine H decides A_{TM}, for the purpose of obtaining a contradiction. We construct the following machine B.

B = "On input w:

1. Obtain, via the recursion theorem, own description $\langle B\rangle$.
2. Run H on input $\langle B, w\rangle$.
3. Do the opposite of what H says. That is, *accept* if H rejects and *reject* if H accepts."

Running B on input w does the opposite of what H declares it does. Therefore, H cannot be deciding A_{TM}. Done!

23.

6.25 Assume for the sake of contradiction that some TM X decides a property P, and P satisfies the conditions of Rice's theorem. One of these conditions says that TMs A and B exist where $\langle A \rangle \in P$ and $\langle B \rangle \notin P$. Use A and B to construct TM R:

$R =$ "On input w:

1. Obtain own description $\langle R \rangle$ using the recursion theorem.
2. Run X on $\langle R \rangle$.
3. If X accepts $\langle R \rangle$, simulate B on w.
   If X rejects $\langle R \rangle$, simulate A on w."

If $\langle R \rangle \in P$, then X accepts $\langle R \rangle$ and $L(R) = L(B)$. But $\langle B \rangle \notin P$, contradicting $\langle R \rangle \in P$, because P agrees on TMs that have the same language. We arrive at a similar contradiction if $\langle R \rangle \notin P$. Therefore, our original assumption is false. Every property satisfying the conditions of Rice's theorem is undecidable.

24.

THEOREM 6.7

MIN_{TM} is not Turing-recognizable.

PROOF Assume that some TM E enumerates MIN_{TM} and obtain a contradiction. We construct the following TM C.

$C =$ "On input w:

1. Obtain, via the recursion theorem, own description $\langle C \rangle$.
2. Run the enumerator E until a machine D appears with a longer description than that of C.
3. Simulate D on input w."

Because MIN_{TM} is infinite, E's list must contain a TM with a longer description than C's description. Therefore, step 2 of C eventually terminates with some TM D that is longer than C. Then C simulates D and so is equivalent to it. Because C is shorter than D and is equivalent to it, D cannot be minimal. But D appears on the list that E produces. Thus, we have a contradiction.

25.

THEOREM 6.8

Let $t\colon \Sigma^* \longrightarrow \Sigma^*$ be a computable function. Then there is a Turing machine F for which $t(\langle F \rangle)$ describes a Turing machine equivalent to F. Here we'll assume that if a string isn't a proper Turing machine encoding, it describes a Turing machine that always rejects immediately.

In this theorem, t plays the role of the transformation, and F is the fixed point.

PROOF Let F be the following Turing machine.

F = "On input w:

1. Obtain, via the recursion theorem, own description $\langle F \rangle$.
2. Compute $t(\langle F \rangle)$ to obtain the description of a TM G.
3. Simulate G on w."

Clearly, $\langle F \rangle$ and $t(\langle F \rangle) = \langle G \rangle$ describe equivalent Turing machines because F simulates G.

26.

Lemma. Let M be a Turing machine with language alphabet Σ. There exists a Turing machine M' such that it runs M on all possible inputs (Σ^*) in parallel.

Proof. Let M' be a non-determinestic Turing machine that when it starts on an empty tape, does one of the following:

1. randomly chose a character x from Σ, write x on the tape and move the head to the right.
2. Simulate M on the string currently on the tape.

---

27. We construct $T^{A_{TM}}$ as follows:

$T^{A_{TM}}$ = "On input $< M >$, where M is a TM:

1. Construct the TM N as follows:
   N = "On any input:
   (a) Run M on all strings in Σ^*.
   (b) If M accepts any string, *accept*.
2. Query $< N, \epsilon >$ to the oracle for A_{TM}.
3. If $< N, \epsilon > \in A_{TM}$, *reject*; *accept* other wise.

It is clear that $T^{A_{TM}}$ decides E_{TM}.

28. Let C be a decidable TM that checks whether a finite sequence of dominos is a match. We construct $P^{A_{TM}}$ as follows:
$P^{A_{TM}}$ = "On input $(d_1, d_2, ..., d_n)$, where d_i is a valid domino:

1. Construct the TM N as follows:
   N = "On any input:

   (a) Run C on all non-empty finite sequences of dominos.

   (b) If C accepts any sequence, *accept*.

2. Query $< N, \epsilon >$ to the oracle for A_{TM}.

3. If $< N, \epsilon > \in A_{TM}$, *accept*; *reject* other wise.

It is clear that $P^{A_{TM}}$ decides PCP.

29. We already know that there are countably many TMs, and also there are uncountably many languages on any non-trivial finite alphabet. Note that the ability to send queries to an oracle can be described in the description of a Turing machine with finitly many characters (for example "$< M, w > \in A_{TM}$"). Therefore the set of descriptions for oracle TMs is also countable.
Furthermore, we can easily show that the set of descriptions for TMs with access to any finite number of oracles is still countable. Thus, there are languages unrecognizable to oracle TMs.