# COMPUTABILITY

Computability (see Refs. 1 to 5) is the field of theoretical computer science that deals with properties of computational problems—that is, what problems can and cannot be solved by a computer, and for those that can be solved, whether they can be solved efficiently. Whereas many problems can be solved by a computer, there are some interesting and important problems that are simply not computable. That is, it is impossible to design algorithms and, hence, computer programs that will solve such problems. For example, it is impossible to write a computer program that will decide if another program will always halt. There are other problems that are indeed computable, but the known algorithms take so much time or space that they may as well not be computable. The classic *traveling salesman problem*—the problem of finding the shortest route through a set of cities, visiting each city only once, is an example of such a problem.

In this article we characterize, as precisely as possible, those problems that can and cannot be solved by a computer. We are not concerned here with the efficiency or practicality of the solutions; we are merely interested in any computer solution to the problem, even one that is unrealistically slow, that operates on an imaginary computer, or even one that exploits unbounded time and memory space. The equally important problem of characterizing the problems that can and cannot be computed efficiently is addressed elsewhere in this encyclopedia. (See COMPUTATIONAL COMPLEXITY THEORY.)

In order to characterize what problems can and cannot be solved by a computer, we must first define what we mean by a computer and what it means for a problem to be computable. To this end, we introduce an abstract model of computation called a Turing machine. We show that computer programs can be viewed as computing functions on these Turing machines and that the set of all computer programs that can be written can be characterized in terms of the set of computable functions. We further show that these computable functions can be precisely characterized in terms of the class of

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 1999 John Wiley & Sons, Inc.

so-called recursive functions, which are a well-defined set of functions on the natural numbers.

With the notion of computable functions in hand, we go on to examine the equally interesting problem of characterizing what problems cannot be solved by computers. To simplify this task, we show that determining what problems can be solved by a computer is equivalent to determining what so-called decision problems can be solved by a computer. Hence, the problems that cannot be solved by a computer are the undecidable problems. We investigate this class of problems by returning to the mathematical description of a Turing machine. We employ a method of argument called diagonalization to show that there are more different decision problems than there are different algorithms or programs and, hence, that there are indeed decision problems for which no algorithm exists. Finally, we give several examples illustrating the impact and importance of these problems.

## TURING MACHINES AND CHURCH'S THESIS

To characterize what problems can and cannot be solved by a computer, we must first explain what we mean by a computer and, in turn, what we mean by something being computable. These problems were indirectly posed to the mathematics community in 1931 by David Hilbert in the form of his famous Entscheidungsproblem, which can be paraphrased as follows:

> Is there some general mechanical procedure which could, in principle, solve all the problems of mathematics (belonging to some suitably well-defined class of problems) one after the other?

As researchers eventually discovered, the answer to this question is "no." There is no such mechanical procedure that can solve all the problems of mathematics. However, in proving this result, researchers made significant progress in formally characterizing the power of an automatic machine or computer, and the mechanical procedures or algorithms that could operate on it. One important contribution toward addressing the *Entscheidungsproblem* was that of the British mathematician Alan Turing.

Turing's effort to mathematically characterize the functioning of a machine in terms of sets of primitive operations led him to introduce the notion of a Turing machine. Intuitively, what Turing did was to define and characterize mathematically the operation of an idealized computer, a Turing machine. He then characterized the functions that can be computed by any possible instantiation of a Turing machine (i.e., *any* possible computer). The functions that can be computed are referred to as the computable functions.

There are several variations on the exact description of a Turing machine. Informally, a Turing machine is a tape player that can be in one of a finite number of states $Q = \{q_1, \ldots, q_u\}$. The machine operates on a tape of potentially infinite length that is divided into squares, each of which contains a symbol. The symbol is either blank, $\square$, or is drawn from an input language, $\Sigma$ (e.g., 0, 1). Collectively, these symbols compose the finite set of tape symbols, $\Gamma$. The tape player has a head that can both read and write symbols and it operates by reading the tape one square at a time. Starting in the internal start state $q_1$, the action of the tape player is determined by a transition function $\delta$ that maps its internal state, $q$, and the symbol currently being read, $s$, into a resulting action to be taken. The actions that the Turing machine can perform are to: write a symbol, $s'$, onto the tape, move the tape one square left or right and/or to change its internal state (to $q'$). The transition function therefore encodes the algorithm implemented by the Turing machine. The Turing machine will continue to make transitions in accordance with the transition function $\delta$ until it reaches the single halt and accept state $q_2$ or the single halt and reject state $q_3$. Figure 1 shows an example of a Turing machine with the tape head positioned over a 0 symbol.

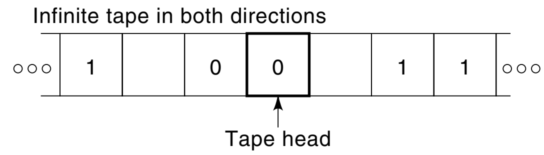


**Figure 1.** Turing machine.

To summarize, a Turing machine, $\mathscr{M}$, is specified by a 7-tuple: $\mathscr{M} = \{Q, \Sigma, \Gamma, \delta, q_1, q_2, q_3\}$, where

- $Q = \{q_1, \ldots, q_u\}$ is a finite set of states.
- $\Sigma$ = a finite input language, typically $\{0, 1\}$.
- $\Gamma = \{s_1, \ldots, s_v\}$ is a finite set of tape symbols such that $\Sigma \subset \Gamma$ (typically $\Gamma = \Sigma \cup \square$).
- $\delta$ = the transition function $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, where $L$ means the tape is moved one square to the left and $R$ means it is moved one square to the right.
- $q_1 \in Q$ is the single start state.
- $q_2 \in Q$ is the single halt and accept state.
- $q_3 \in Q$ is the single halt and reject state.

We restrict our attention to deterministic Turing machines, that is, we assume that for every pair of internal states and tape symbols $(q, s) \in (Q \times \Gamma)$, there is exactly one triplet consisting of an internal state, a tape symbol to be written, and a tape movement operator, $(q', s', \{L, R\})$ in $(Q \times \Gamma \times \{L, R\})$ such that $\delta(q, s) \rightarrow (q', s', \{L, R\})$. When the Turing machine $\mathscr{M}$ runs on the given tape input it goes through the transitions as specified by $\delta$ until it reaches state $q_2$ or $q_3$ in $Q$, whereupon $\mathscr{M}$ halts.

Natural numbers can be encoded on the tape as input. For example, the natural number $x$ can be encoded as a sequence of zeroes and ones, representing $x$ in binary notation. Similarly, there is a scheme to encode tuples of natural numbers $(x_1, \ldots, x_n)$. Suppose $\phi$ is an $n$-variable partial function from natural numbers to natural numbers. Then a particular Turing machine $\mathscr{M}$ *computes* the function $\phi$ if for any tuple of natural numbers $(x_1, \ldots, x_n)$ the following holds: (1) If $\phi(x_1, \ldots, x_n)$ is defined and $\phi(x_1, \ldots, x_n) = y$, then when $M$ is run with $x_1, \ldots, x_n$ initially encoded on the tape, and the tape is blank everywhere else, $\mathscr{M}$ eventually halts in $q_2$ with $x_1, \ldots, x_n, y$ encoded on the tape; and (2) If $\phi(x_1, \ldots, x_n)$ is undefined, then when $M$ is run with $x_1, \ldots, x_n$ initially on the tape and blank everywhere else, $\mathscr{M}$ either fails to halt or halts in state

$q_3$. We say that a partial function $\phi$ is (*Turing*) *computable* when there is a Turing machine $\mathscr{M}$ that computes it.

While Turing conceptualized the notion of computability with respect to his Turing machines, the mathematical notion of computability is more fundamental and is not limited to Turing's conceptualization of a computer. Indeed, a number of different mathematicians, including Markov, Post, Kleene, Church, and Gödel contributed in different ways to these ideas. Out of their work came several characterizations of the notion of an algorithm and the set of computable functions. Church, Turing, and Markov all claimed that the class of functions they had defined coincided with the class of computable functions. It turns out that all their characterizations were equivalent. This claim was captured in *Church's Thesis* (sometimes referred to as the *Church–Turing Thesis*), which can be stated as follows:

The class of problems that can be solved in any reasonable model of computation is exactly the same as the class of problems that can be solved by a Turing machine.

That is, any problem for which we can find an algorithm that can be programmed in any programming language and run on any computer, real or imaginary, even one exploiting unbounded time and memory space is computable by a Turing machine. Hence, we can use the concept of computable functions to characterize the problems that can be solved by a computer. Observe that Church's Thesis is indeed a thesis or a claim, rather than a theorem, since it is predicated on an intuitive and informally defined notion of computability and cannot be supported by mathematical proof. Nevertheless, Church's thesis is widely accepted as true.

## RECURSIVE FUNCTIONS

In the previous section we explained that every effective computation can be carried out by a Turing machine, and that the Turing computable functions (computable functions) characterized exactly what can be computed. In this section we provide a precise mathematical characterization of the class of computable functions by introducing the notion of recursive functions. The set of computable functions is equivalent to the set of partial recursive functions. Hence, a function on the natural numbers is computable if and only if it is a partial recursive function. In what follows we provide a precise definition of the class of recursive functions.

Recursive functions map natural numbers to natural numbers. The class of recursive and partial recursive functions is composed of a distinguished set of *initial functions* that are assumed to be computable, and the functions that can be obtained from the initial functions by repeated application of any combination of three specific *generating rules for functions.* There are three initial functions:

1. *Zero Function.* $Z:\mathscr{N}\rightarrow\mathscr{N}$ such that $Z(x) = 0$ for all $x$. That is, $Z(x)$ is the function on the natural numbers that for any given input $x$ returns 0.
2. *Successor Function.* $S:\mathscr{N}\rightarrow\mathscr{N}$ such that $S(x) = x + 1$ for all $x$. In other words, the successor function simply returns the successor of $x$ (i.e., $x + 1$).
3. *Projection Function.* $\pi_{ni}:\mathscr{N}^n\rightarrow\mathscr{N}$ for $1 \leq i \leq n$, such that $\pi_{ni}(x_1, x_2, \ldots, x_n) = x_i$. The projection function therefore, returns the value of the $i$th of $n$ natural numbers which are specified as inputs to the function.

Additionally there are three generating rules for functions:

1. *Composition.* Let $f$ and $h_1, h_2, \ldots, h_m$ be functions with $n$ parameters (i.e., $n$-place functions) and let $g$ be an $m$-place function. Then $f$ is obtained from $g$ and $h_1, h_2, \ldots, h_m$ by composition if

$$f(x_1, x_2, \ldots, x_n) = g[h_1(x_1, x_2, \ldots, x_n), \ldots, h_m(x_1, x_2, \ldots, x_n)]$$

2. *Primitive Recursion.* Let $f$ be an $m$-place function, $h$ an $(m + 1)$-place function, and $g$ an $(m - 1)$-place function. The following system of equations determines a unique $m$-place function $f$, by primitive recursion.

$$f(x_1, \ldots, x_{m-1}, 0) = g(x_1, \ldots, x_{m-1})$$
$$f(x_1, \ldots, x_{m-1}, y+1) = h[x_1, \ldots, x_{m-1}, y, f(x_1, \ldots, x_{m-1}, y)]$$

3. *Minimization.* Let $f$ be an $m$-place function and $g$ be an $(m + 1)$-place function. Then $f$ is obtained from $g$ with the aid of a minimization operator or a least number operator, if for any $x_1, \ldots, x_m$, and $y$, $f(x_1, \ldots, x_m) = y$ holds if and only if $g(x_1, \ldots, x_m, 0) \ldots g(x_1, \ldots, x_m, y - 1)$ are defined and are not equal to 0, while $g(x_1, \ldots, x_m, y) = 0$.

*Recursive functions* are total functions (i.e., functions that are defined for every argument) that are obtained from the initial functions by means of a finite number of applications of the generating rules for composition, primitive recursion, and minimization. If the class of functions includes partial functions (i.e., functions which may not be defined for some arguments), then they are called *partial recursive functions.* If minimization is not employed to obtain the total functions then the functions are said to be *primitive recursive.* Most of the computable functions of everyday interest can be characterized by the class of primitive recursive functions.

## DECISION PROBLEMS AND DECIDABILITY

We have seen that the class of partial recursive functions coincides with the functions that are computable. This gives a precise mathematical characterization of the functions and, hence, the problems, that are computable.

While it is extremely important to characterize the class of functions that are computable, many important problems in computer science and engineering are not computable. That is, when we think of these problems as some function to be solved, we find that they cannot be computed—there is no guaranteed mechanical procedure or algorithm to solve them. In this section, we examine the equally interesting class of functions that are not computable. We do so by returning to our notion of a Turing machine and introducing the concept of a decision problem and the related notion of decidability. We show below that to study the problems that can and cannot be solved by a computer, it is sufficient to study the special class of problems known as decision problems.

We make two assumptions in order to simplify the discussion. First, we assume that all functions of interest map strings (A string is a finite sequence of elements.) over a finite alphabet $\Sigma$ to strings over $\Sigma$. Typically, the finite alphabet will be {0, 1}. This assumption is not restrictive, because any enumerable set of objects (i.e., any set of objects that can be listed by a computer) can always be encoded by a set of strings over {0, 1}, just as we encoded the natural numbers as binary strings in the discussion above.

Our second assumption is that functions are limited to those whose range (possible outputs) consists of only two elements, 0 and 1. With these functions, if $f(x) = 1$, then we say that $x$ is accepted; if $f(x) = 0$ then we say that $x$ is rejected. For example, the function $ODD(n)$ that returns a 1 if $n$ is odd and a 0 if $n$ is not odd, is such a function. Similarly, the problem of determining whether or not there are $n$ consecutive 7's in the decimal expansion of $\pi$, can be expressed using such a function. Problems that can be expressed using this restricted set of functions are referred to as *decision problems;* if the decision problem is computable, then we say that it is *decidable.*

In characterizing the problems that can and cannot be solved by a computer, it is sufficient to consider only the decision problems. To provide the intuition behind this claim, we assume without loss of generality, that all functions have a range consisting of the natural number, $\mathcal{N}$. If $f(x) = i$ is a function over $\mathcal{N}$, then the decision problem $\mathcal{D}_f$, associated with $f$ takes as input $(x, y)$ and is defined in such a way that it accepts the input if and only if $y = f(x)$. Therefore, by using the decision problem, the function $f(x) = i$ can be computed by posing the following sequence of questions of $\mathcal{D}_f$ until a yes result is returned: Is $0 = f(x)$? Is $1 = f(x)$? Is $2 = f(x)$?, . . .? Is $i = f(x)$? As a result, one can show that for any function $f$, one can construct a corresponding decision problem $\mathcal{D}_f$, with the property that $\mathcal{D}_f$ is computable if and only if $f$ is computable.

We provide a formal definition of a decision problem and related concepts, in terms of a Turing machine. Recall that in the previous section, we characterized Turing machines in terms of the arbitrary functions that they compute. Since we have elected to simplify our discussion by focusing on the restricted set of functions having a range of only two elements it will also be helpful to modify our definition of a Turing machine to be an acceptor or rejecter of a string, rather than computing a function on a string. Such Turing machines are called *deciders.* That is, the Turing machine takes as input a string $x$ over $\Sigma$ and must either accept or reject the string $x$. As before, a Turing machine $\mathcal{M}$ will be specified by the same 7-tuple, and the computation steps of the Turing machine on a particular input $x$ will again be prescribed by the transition function. However, now we will say that $\mathcal{M}$ halts and accepts $x$ if $\mathcal{M}$ halts in state $q_2$; $\mathcal{M}$ halts and rejects $x$ if $\mathcal{M}$ halts in state $q_3$; and $\mathcal{M}$ does not halt and rejects $x$ if it never reaches $q_2$ or $q_3$. The language accepted by $\mathcal{M}$ will be the set of all strings that are accepted by $\mathcal{M}$. (The language corresponds to all of the domain elements of the function that map to 1.) Note that there is a unique language accepted by any particular Turing machine.

With this modified definition of a Turing machine we define the following:

Denote $\Sigma^*$ to be the set of all strings over $\Sigma$. A *language L* (over $\Sigma$) is simply a subset of $\Sigma^*$. The *decision problem* associated with $L$ is a function that takes as input a string $x$ over $\Sigma$ and outputs 1 (*accepts*) on $x$ if $x \in L$ and outputs 0 (*rejects*) on $x$ if $x \notin L$.

A language $L$ is *decidable* or *computable* or *recursive* if there exists a Turing machine (over $\Sigma$) that halts on all inputs, and that accepts exactly those strings in $L$.

A language $L$ is *semi-decidable* or *semi-computable* or *recursively enumerable* if there exists a Turing machine (over $\Sigma$) that accepts exactly those strings in $L$.

Note the distinction between decidable and semidecidable. If $L$ is decidable, then there is an algorithm that always halts and can always distinguish members of $L$ from nonmembers. For example, the (encodings of the) set of all prime numbers is decidable. On the other hand, consider the example above of determining for any natural number $n$ whether there are $n$ consecutive 7's in the decimal expansion of $\pi$. One can imagine a procedure that would generate consecutive digits in the expansion of $\pi$ one at a time, watching for consecutive 7's. If $n$ 7's appear consecutively, the procedure will halt and output a 1 for yes. The problem with this procedure, of course, is that it is not guaranteed to halt. Since the expansion of $\pi$ is infinite, there is no point in the execution of the procedure at which the procedure knows it can halt and answer no, if it has not seen $n$ consecutive 7's. Thus, the procedure is not guaranteed to halt.

A related concept is that of *effective enumerability*. A set (or language) is said to be effectively enumerable if there exists a Turing machine that can output the complete list (possibly with repetition) of the elements in the set (or language). It can be shown that a set (or language) is effectively enumerable if and only if it is recursively enumerable (i.e., semidecidable).

While decidable problems (sets) and semidecidable problems (sets) are not equivalent, there are some interesting relationships between them. In particular, every decidable problem is semidecidable. Also, a problem is decidable if and only if both it and its complement are semidecidable.

### Turing Machine Enumeration

As mentioned earlier, some important decision problems arising in science are not decidable or even semidecidable. In what follows, we will set up the framework to prove some of these impossibility results. The intuition behind the simplest impossibility result, that some languages are not semidecidable, is the fact that the set of all semidecidable languages is not very large (technically, it is countable), whereas the set of all languages is very large (it is not countable). In this section we will first describe how to represent a Turing machine by a unique natural number. This unique representation is then used in a subsequent section to show that there are functions that are not semidecidable.

We will only consider Turing machines whose input alphabet, $\Sigma$ is {0, 1}. This restriction is not necessary, but it will slightly simplify our discussion. We will also assume that our Turing machine uses the following conventions: (1) The states are ordered $q_1, . . ., q_u$, and the tape symbols are also ordered $s_1, . . ., s_v$; (2) $q_1$ is the start state; (3) $q_2$ is the single halt and accept state; (4) $q_3$ is the single halt and reject state; (4) $s_1$ denotes 0, and $s_2$ denotes 1, and the remaining tape symbols are $s_3, . . ., s_v$. Note that the number of states, $u$, and the number of tape symbols, $v$, is always finite but can be arbitrarily large. Any Turing machine accepting some language over {0, 1} can be reconfigured to satisfy the above conventions and still accept the same language.

We are now ready to describe our encoding. Consider a Turing machine satisfying the above conventions: $\mathscr{M} = (Q = \{q_1, \ldots, q_u\}, \Sigma = \{0, 1\}, \Gamma = \{s_1, \ldots, s_v\}, \delta, q_1, q_2, q_3)$. We will now designate "move left" by $D_1$, and "move right" by $D_2$. We can represent a transition of this Turing machine, $\delta(q_i, s_j) \rightarrow (q_k, s_l, D_m)$ by a 5-tuple $(i, j, k, l, m)$, which we will encode uniquely by the 0-1 sequence $0^i10^j10^k10^l10^m$ (i.e., $i$ 0's followed by a 1 followed by $j$ 0's followed by a 1, etc.). In this way, the binary code for $\mathscr{M}$ is: $111\text{code}_1 11\text{code}_2 11 \ldots 11\text{code}r111$, where $\text{code}_i$ is the code for one of the possible transitions, two consecutive 1's are used to separate each $\text{code}_i$ and the entire sequence begins and ends with three 1's.

For example, consider an unrealistically simple deterministic Turing machine with states $q_1$, $q_2$, $q_3$, input symbols {0, 1} and tape symbols {0, 1, □}. $\mathscr{M} = (Q = \{q_1, q_2, q_3\}, \Sigma = \{0, 1\}, \Gamma = \{0, 1, \square\}, \delta, q_1, q_2, q_3)$. Recalling our conventions, the elements 0, 1, □ of Γ will be renamed $s_1$, $s_2$, $s_3$, respectively. The code for $\mathscr{M}$ is $111\text{code}_1 11\text{code}_2 11\text{code}_3 11 \ldots 11\text{code}_9 111$, where $\text{code}_1, \ldots, \text{code}_9$ are the codes for the 9 transitions. (Since there is one transition for every possible pair consisting of a state and a tape symbol, there are a total of 9 transitions specified by $\delta$.) Below, we specify 4 of the 9 transitions of $\delta$, along with the code associated with these transitions.

(a) $\delta(q_1, 1) = \{(q_3, 0, D_2)\}$; $\text{code}_1 = 0100100010100$
(b) $\delta(q_3, 0) = \{q_1, 1, D_2)\}$; $\text{code}_2 = 0001010100100$
(c) $\delta(q_3, 1) = \{(q_2, 0, D_2)\}$; $\text{code}_3 = 00010010010100$
(d) $\delta(q_3, \square) = \{(q_3, 1, D_1)\}$; $\text{code}_4 = 0001000100010010$

Given an encoding as specified above, it is possible to completely recover the specification of the Turing machine, up to renaming of the states and tape alphabet. It is also possible, given an arbitrary natural number in binary notation, to decide whether or not it is a valid encoding of a Turing machine. Thus the set of all Turing machines is *countable* since it can be mapped in a one-to-one fashion to the set of natural numbers. If $\mathscr{M}$ is a Turing machine satisfying the above conventions, $\langle \mathscr{M} \rangle$ will denote the binary number that encodes $\mathscr{M}$. Also $\langle \mathscr{M}, x \rangle$ will denote the concatenation of the encodings of the pair of numbers where the first number in the pair is $\langle \mathscr{M} \rangle$, and the second number in the pair is $x$, (where $x$ is the input to the Turing machine).

### Universal Turing Machines

The universal language, $L_u$ over $\Sigma = \{0, 1\}$ is defined to be the set of pairs $\langle \mathscr{M}, x \rangle$ such that $\langle \mathscr{M} \rangle$ encodes a Turing machine over $\Sigma$ and $x$ is an input string over $\Sigma$ and such that $\mathscr{M}$ accepts $x$. A universal Turing machine, $U$ over $\Sigma$ is simply a Turing machine that accepts $L_U$. It is an important fact that $L_U$ is semidecidable. That is, there exists a universal Turing machine $U$ that takes as input $\langle \mathscr{M}, x \rangle$ such that: (a) if $\mathscr{M}$ accepts $x$, then $U$ halts on $\langle \mathscr{M}, x \rangle$ and accepts; (b) if $\mathscr{M}$ does not accept $x$, then $U$ may or may not halt on $\langle \mathscr{M}, x \rangle$ and does not accept. The idea behind the construction of $U$ is quite simple: $U$ first decodes $\langle \mathscr{M}, x \rangle$ into the pair of numbers $\langle \mathscr{M} \rangle$, which represents the "program" and $x$ which is the input. Then $U$ simulates the execution of $\mathscr{M}$ on the input $x$. If the simulation terminates, then $U$ accepts if and only if the simulation accepts. Otherwise, if the simulation does not terminate, then $U$ will also fail to terminate on $\langle \mathscr{M}, x \rangle$.

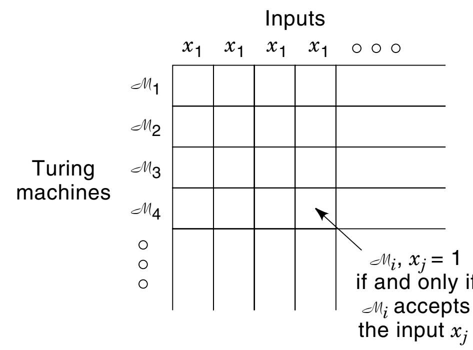


**Figure 2.** Diagonal language.

### Diagonalization

In this section we will show that there exists a language that is not semidecidable. We have already seen that each Turing machine can be represented by a unique binary number. Therefore, it is possible to order all Turing machines over $\Sigma = \{0, 1\}$: $\mathscr{M}_1, \mathscr{M}_2, \mathscr{M}_3, \ldots$ according to their encodings, where $\mathscr{M}_i < \mathscr{M}_j$ if the encoding $\langle \mathscr{M}_i \rangle$ for $\mathscr{M}_i$ is less than the encoding $\langle \mathscr{M}_j \rangle$ for $\mathscr{M}_j$. We can also order all binary inputs $x_1, x_2, x_3, \ldots$ in the obvious way.

Now we define the *diagonal* language $L_D$ as the set of all binary strings $x_i$ such that $x_i$ is the $i$th binary input, and $\mathscr{M}_i$, the $i$th Turing machine, does *not* accept $x_i$. We now show that $L_D$ is not semidecidable. Construct a table as shown in Fig. 2, where the horizontal axis is labeled with all possible binary input strings, $x_1, x_2, x_3, \ldots$ and the vertical axis is labeled with all possible Turing machines, $\mathscr{M}_1, \mathscr{M}_2, \mathscr{M}_3, \ldots$. Entry $(i, j)$ in the table has value 1 if $\mathscr{M}_i$ accepts $x_j$ and has value 0 if $\mathscr{M}_i$ does not accept $x_j$. While the constructed table is infinite in both directions (and therefore cannot actually be written down by any human), for any particular $i$ and $j$, the entry $(i, j)$ is well defined.

We claim that $L_D$ cannot be accepted by any of the Turing machines that have been listed in the table. This is because for every $i$, the machine $\mathscr{M}_i$ gives the wrong answer on the input $x_i$; in other words, $\mathscr{M}_i$ accepts $x_i$ if and only if $x_i$ is not in $L_D$. Put another way, a language over {0, 1} can be described by an infinite 0-1 sequence, where the $j$th element in the sequence is 1 if and only if the $j$th input, $x_j$ is in the language. In the table shown in Fig. 2, the set of *all* languages that are accepted by Turing machines are listed by the rows of our table: row $i$ describes the language accepted by $\mathscr{M}_i$. $L_D$, again viewed as an infinite 0-1 sequence, is carefully chosen to be the complement of the sequence along the diagonal. $L_D$ is thus different from every row since $L_D$ differs from any row $k$ on the $k$th input.

The above argument is called a diagonalization argument and it was originally devised by Cantor in order to show that the real numbers are uncountable; that is, there is no one-to-one function from the real numbers to the natural numbers.

### Reductions

In the previous section we have argued that there exists a language (albeit an unnatural one) that is not semidecidable.

Now we can use this language to show that many other more natural languages are also not semidecidable, and even to show that some languages which are semidecidable are not decidable. The technique that we use here is called the method of *reductions*. It is also a variation of the primary method used in complexity theory to give evidence that a function is difficult to compute. (See COMPUTATIONAL COMPLEXITY THEORY.)

Let $A$ and $B$ be two decision problems, or even more generally let $A$ and $B$ be two languages or functions. Then $A$ *reduces* to $B$ if we can use a program solving $B$ to solve $A$. Intuitively, if $A$ reduces to $B$, then $A$ is not harder than $B$. This idea can be used in the contrapositive form to get negative results. In other words, if $A$ reduces to $B$, and $A$ is not decidable, then it follows that $B$ is also not decidable.

Here is a simple example illustrating the idea. Consider the complement of the diagonal language, $\overline{LD}$, which is defined to be the set of strings $x_i$ such that $x_i$ is the $i$th binary string and $\mathcal{M}_i$, the $i$th Turing machine, accepts $x_i$. (The usual notion of a complement of a language $A$ would be defined to be the set of all strings over $\{0, 1\}$ that are not in $A$. The languages $L_D$ and $\overline{LD}$ are not complements in this usual sense since some strings are not valid encodings of Turing machines and hence are not in either $L_D$ or $\overline{LD}$.) It is not hard to see that $L_D$ reduces to $\overline{LD}$ and also that $\overline{LD}$ reduces to $L_D$ in the following sense: there exists a Turing machine that always halts and accepts $L_D$, if and only if there is a Turing machine that always halts and accepts $\overline{LD}$. Thus it follows that $\overline{LD}$ cannot be decidable since if it were decidable, then $L_D$ would also be decidable, and we already know that $L_D$ is not even semidecidable.

We will now show that the universal language, $L_U$ is not decidable by reducing $\overline{LD}$ to $L_U$. (Recall that we have already seen that $L_U$ is semidecidable.) Suppose that $L_U$ is decidable and $\mathcal{M}_U$ accepts exactly $L_U$ and always halts. We will now describe an algorithm for $\overline{LD}$, which uses $\mathcal{M}_U$ as a subroutine: (1) Given $y$, determine $i$ such that $y = x_i$. That is, $y$ is the $i$th string in the infinite ordering of all possible input strings, and determine $\mathcal{M}_i$, the $i$th Turing machine. (2) Run $\mathcal{M}_U$ on $\langle \mathcal{M}_i, x_i \rangle$ and accept $y$ if and only if $\mathcal{M}_U$ accepts.

There are hundreds of problems that are known to be undecidable, and the proof is generally a reduction. As a last example, we will show that the famous halting problem is not decidable. The halting language, $L_{\text{Halt}}$ are those numbers $\langle \mathcal{M}, x \rangle$, such that $\mathcal{M}$ eventually halts on the input $x$. The halting problem is semidecidable: simply simulate $\mathcal{M}$ on $x$; if $\mathcal{M}$ halts and accepts, then the simulation will halt and in this case $\langle \mathcal{M}, x \rangle$ will be accepted; if $\mathcal{M}$ halts and rejects, then the simulation will halt and $\langle \mathcal{M}, x \rangle$ will be accepted; otherwise if $\mathcal{M}$ does not halt on $x$, then the simulation will not halt on $\langle \mathcal{M}, x \rangle$, and thus $\langle \mathcal{M}, x \rangle$ will not be accepted.

We will now show that the halting problem is not decidable by reducing $L_U$ to $L_{\text{Halt}}$. Assume for the sake of contradiction that $\mathcal{M}_{\text{Halt}}$ is a Turing machine that always halts, and accepts exactly$L_{\text{Halt}}$. Now we want to use $\mathcal{M}_{\text{Halt}}$ to construct another Turing machine, $\mathcal{M}_U$, that always halts and accepts exactly $L_U$. Given input $\langle \mathcal{M}, x \rangle$, $\mathcal{M}_U$ simulates $\mathcal{M}_{\text{Halt}}$ on $\langle \mathcal{M}, x \rangle$. If $\mathcal{M}_{\text{Halt}}$ accepts, then we simulate $\mathcal{M}$ on $x$ and accept $\langle \mathcal{M}, x \rangle$ if and only if $\mathcal{M}$ accepts $x$. Otherwise ($\mathcal{M}_{\text{Halt}}$ rejects $\langle \mathcal{M}, x \rangle$), $\mathcal{M}_U$ rejects $\langle \mathcal{M}, x \rangle$. To prove the correctness of $\mathcal{M}_U$, notice that this algorithm first checks whether or not $\mathcal{M}$ halts on $x$, using $\mathcal{M}_{\text{Halt}}$. As long as it does halt, then we are guaranteed that the simulation of $\mathcal{M}$ on $x$ will not enter an infinite loop, so we can safely simulate $\mathcal{M}$ on $x$. Otherwise, if $\mathcal{M}$ does not halt on $x$, then we want to reject $\langle \mathcal{M}, x \rangle$. Now we can complete the argument showing that the halting problem is not decidable. Suppose for sake of contradiction that it were decidable. Then $\mathcal{M}_{\text{Halt}}$ described above exists, and therefore, we can obtain $\mathcal{M}_U$, a Turing machine that always halts and accepts exactly $L_U$. But $L_U$ is not decidable, and thus we have reached a contradiction and can therefore conclude that $L_{\text{Halt}}$ is not decidable.

### Other Undecidable Problems

Many problems of fundamental importance in mathematics, science, economics, and engineering are also known to be undecidable (see Ref. 6). Here we give some examples of such problems. In all cases, proving that a problem is undecidable requires a reduction from another problem already known to be undecidable. However, in many of the cases below, the reductions are highly sophisticated.

**Post's Correspondence Problem.** The input to this problem is a collection of dominos. Each domino contains one string (over $\Sigma = \{a, b, \ldots, z\}$) on each side. For example, $\{(b, ca)(a, ab), (ca, a), (abc, c)\}$ is a collection of four dominos. The task is to decide if there exists a listing of the dominos, possibly with repetitions, such that the top string equals the bottom string. For example, a listing that satisfies this property for the above set of four dominos is:

$$(a, ab), (b, ca), (ca, a), (a, ab), (abc, c)$$

since the concatenation of the elements in the first half of the tuple is *abcaaabc*, which equals the concatenation of the elements in the second half of the tuple. An example where it is not possible to obtain such a listing is: $\{(abc, ab), (ca, a), (acc, ba)\}$ because the top string will always have greater length than the bottom string. However, sometimes it is not possible for more subtle reasons and, in fact, Post's correspondence problem is not decidable. The reduction in this case is a more complicated one than the simple reductions illustrated above and uses the idea of computation histories.

**Hilbert's Tenth Problem.** This problem was first posed by Hilbert as the tenth in his famous list of problems. This problem, also known as the diophantine equation problem, is as follows: Given a multivariate polynomial equation with integer coefficients, does it have an integer solution? For example, $4x_2 - x_2 = 1$ has no integer solution, whereas by Fermat's last theorem, $x_1^a + x_2^a = x_3^a$ has no integer solution for all $a \geq 3$. This problem, posed by Hilbert in 1900, remained open until it was proven to be undecidable by Matiyasevich in 1973.

**Rice's Theorem.** We've seen that a fundamental question about programs (whether a program halts or not) cannot be answered by a program that always halts. There are similar results for many other questions about programs, such as whether a program halts on a particular input, whether a program accepts a particular input, or whether a program ever halts. Rice's theorem says that *any* nontrivial question about the behavior of programs cannot be answered by a program that always halts. More precisely, suppose that $\mathcal{C}$ is a proper, nonempty subset of the set of all semidecidable languages.

Then the following problem is not decidable: Given $\langle \mathscr{M} \rangle$, is the language accepted by $\mathscr{M}$ in $\mathscr{C}$?

**Word Problems for Groups.** An important class of problems from algebra are the combinatorial word problems for presentations of various algebraic structures. In 1947, Post showed that the word problem for semigroups was undecidable. The analogous result for groups was open for many years, and was finally established in 1982 by Novikov.

**Fractal Geometry.** Complex systems abound in many fields, including biology, physics, ecology, and meteorology. Geometrically one can display the working of such complex systems as fractal images, where fractals are defined by rather simple iterative methods. However, it turns out that simple questions about fractals cannot be answered by programs that always halt. Penrose originally conjectured that the Mandelbrot set is undecidable in a nonstandard theory of computation over the real numbers. More recently, some fractal properties have been shown to be undecidable in the usual sense.

**Decidability of Logical Theories.** The problem of determining if a mathematical logic statement is true or false is not decidable. In order to state this problem rigorously, we need to define what a mathematical statement is, and what it means for it to be true or false. The underlying alphabet consists of the symbols: $\wedge$, $\vee$, $\neg$, (,), $\forall$, $\exists$, $x$, $R_1, \ldots, R_k$. Variables $x_1, x_2, \ldots$ are denoted by $x$, $xx$, and so forth. The $R_i$'s are relation symbols, each of a fixed arity. A well-formed formula is defined inductively as follows: (1) $R_i(x_1, \ldots, x_j)$ is an atomic formula, as long as $R_i$ has arity $j$; (2) If $A$ and $B$ are formulas, then so are $A \wedge B$, $A \vee B$, $\neg A$, $\forall x_i A$, $\exists x_i A$. A *model* is a collection of underlying elements (a universe) together with an assignment of relations to the relation symbols. For example, $(N, +, \times)$ is the model whose universe is the natural numbers, and with two relations, addition and multiplication. Now the decision problem, $Th(N, +, \times)$ associated with the model $(N, +, \times)$ is the set of encodings of well-formed formulas that are true over $(N, +, \times)$. It is a fundamental theorem in mathematical logic that $Th(N, +, \times)$ is not decidable. That is, given an arbitrary formula $\Phi$ as specified above, where the only two relations are addition and multiplication, there is no procedure that always halts and decides whether or not $\Phi$ is true.

## ACKNOWLEDGMENTS

Author Pitassi's research was supported by NSF Grant CCR-9457782, US-Israel BSF Grant 95-00238, and Grant INT-9600919/ME-103 from NSF and MŠMT (Czech Republic). Authors McIlraith's and Brecht's research was supported by the Natural Sciences and Engineering Research Council (NSERC).

## BIBLIOGRAPHY

1. H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability,* New York: McGraw-Hill, 1967.
2. N. Pippenger, *Theories of Computability,* Cambridge: Cambridge University Press, 1997.
3. C. Papadimitriou, *Computational Complexity,* Reading, MA: Addison-Wesley, 1994.
4. M. Sipser, *Introduction to the Theory of Computation,* Boston: PWS Publishing, 1997.
5. J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages and Computation,* Reading, MA: Addison-Wesley, 1979.
6. M. Davis, Unsolvable Problems, in J. Barwise (ed.), *Handbook of Mathematical Logic,* New York: Elsevier Science, 1977.

TONIANN PITASSI
University of Arizona
SHEILA MCILRAITH
Stanford University
TIMOTHY BRECHT
University of Waterloo

**COMPUTATIONAL BIOLOGY.** See BIOLOGY COMPUTING.

**COMPUTATIONAL COMPLEXITY.** See GRAPH THEORY.