Turing Machines

At its logical base every digital computer embodies one of these pencil-and-paper devices invented by the British mathematician A. M. Turing. The machines mark off the limits of computability

by John E. Hopcroft

In 1900 David Hilbert, the preeminent mathematician of his time, challenged the world of mathematics with a list of unsolved problems, presented in Paris before the International Congress of Mathematicians. The 23rd problem on the list was to discover a method for establishing the truth or falsity of any statement in a language of formal logic called the predicate calculus. Thirty-six years were to pass before the problem was settled, and its resolution marked an extraordinary and unexpected turn in mathematics. At the University of Cambridge a young fellow of mathematics in King's College named Alan Mathison Turing had become familiar with Hilbert's 23rd problem through a series of lectures given by M. H. A. Newman. Turing pondered the problem during long afternoon runs in the English countryside, and it was after one of these runs that the answer came to him. Hilbert's problem was impossible to solve.

The publication in which Turing announced his result has had a significance far beyond the immediate problem it addressed. In attacking Hilbert's problem, Turing was forced to ask how the concept of method might be given a precise definition. Beginning with the intuitive idea that a method is an algorithm—a procedure that can be mechanically carried out without creative intervention—he showed how the idea can be refined into a detailed model of the process of computation in which any algorithm is broken down into a sequence of simple, atomic steps. The resulting model of computation is the logical construct called a Turing machine.

The simplest way to describe the Turing machine is in terms of mechanical parts such as wheels, punched tape and a scanner that can move back and forth over the tape. The machinery is not essential—at the most fundamental level Turing's device is the embodiment of a method of mathematical reasoning—but it would be misleading to dispense entirely with the mechanical metaphor. That metaphor was suggestive to Turing himself, who was a pioneer in the development of the digital computer. Moreover, the claims of the computer scientist on the Turing machine as a conceptual tool are now at least as strong as the claims of the logician. Its significance for the theory of computing is fundamental: given a large but finite amount of time, the Turing machine is capable of any computation that can be done by any modern digital computer, no matter how powerful.

The universal capability of the Turing machine does not imply that it would be a practical computer. Any real computer can work many times faster than the Turing machine, because in the design of the real computer clarity of operation is willingly sacrificed for speed and efficiency. Nevertheless, for the theoretical study of the ultimate problem-solving capacity of the real computer the Turing machine has become indispensable. For example, it has enabled mathematicians and computer scientists to prove there are many problems in addition to Hilbert's problem that cannot be solved, no matter how fast or how powerful a computer is applied to their solution.

Turing Machine Operation

What is a Turing machine and how does it work? Andrew Hodges, in his recent biography of Turing, compares it to an ordinary typewriter. Like the typewriter, the Turing machine incorporates a movable printing head that prints discrete symbols, drawn from a finite alphabet, one at a time on a printing surface. To simplify the movements of the printing head, the printing surface is assumed to be a tape marked off into discrete frames or segments. The printing head of the Turing machine therefore needs to move in only one dimension, to the left or to the right, and the actions of the machine need not take account of such irrelevant complexities as the length of the printed line or the width of the space between two lines. Only one printed symbol is allowed in each frame of the tape, but there is no limit imposed on the length of the tape or, consequently, on the length of the string of symbols that can be printed on it.

The movable printing head of the Turing machine can carry out two other functions as well as printing. Like many typewriters manufactured in the past decade, it can remove or erase one symbol at a time from the printing surface. Unlike the typewriter, the printing head can also "read," or register the symbolic content of each tape frame one frame at a time. In this way the symbols on the tape can serve as input to the machine and play a role in determining its subsequent action.

A typewriter can assume one of several states, or modes of operation, in the course of its activity. In its "home" state it prints lowercase letters and numerals, whereas in its "shift" state it prints uppercase letters and special symbols. Similarly, a Turing machine can assume any one of a finite number of states. Each state presumably constitutes a different setup or configuration of the machine, but because the Turing machine is a relatively abstract device, usually no attempt is made to give a more concrete, mechanical description of the states. It will suffice to describe each state of the machine in terms of the effects that state has on the activity of the machine.

The activity of a Turing machine is made up entirely of discrete, instantaneous steps, and each step is determined by two initial conditions: the current state of the machine and the symbol that occupies the tape frame currently being scanned. Given some pair of initial conditions, the machine receives a three-part instruction for its next operating step. The first part of the instruction designates the symbol the machine is to leave in the tape frame being scanned. For example, if the instruction specifies that the symbol 1 is to be left in the frame, the machine prints the symbol if the frame is blank, leaves the symbol alone if a 1 already occupies the frame or erases the symbol if it is not a 1 and replaces it with a 1.

The second part of the instruction

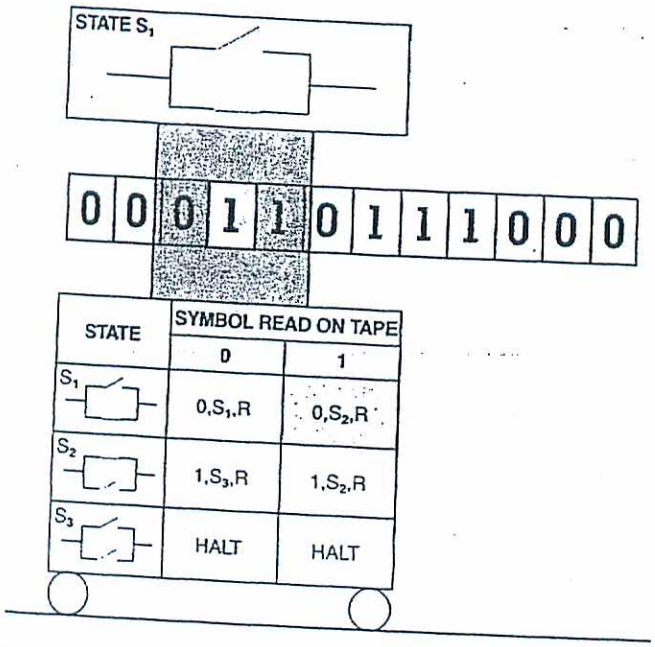
Scientific American, May 1994

specifies the next state of the machine. As with the specification of the symbol in the first part of the instruction, the designation of a state may require the machine to change its state or to remain in its current one. The third part of the instruction specifies whether the printing head is to scan one frame to the left or to the right along the tape.

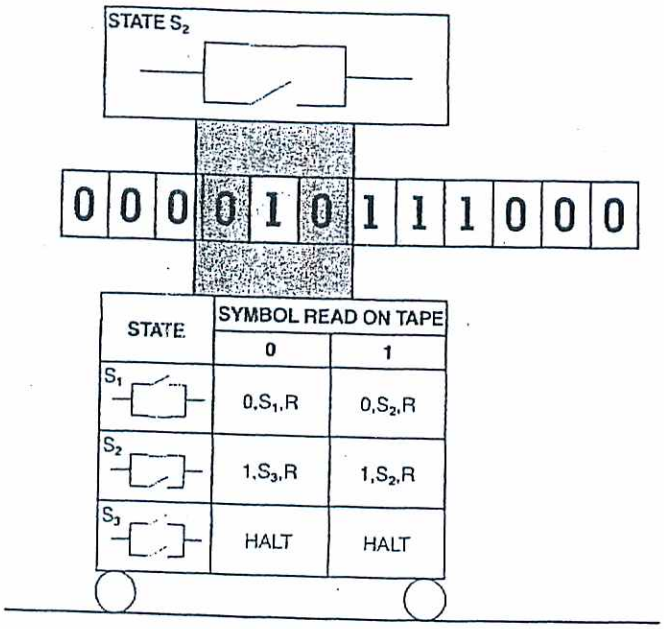
The entire instruction can be abbreviated by listing the three values of the variables—the tape symbol, the machine state and the motion of the printing head—in a fixed order. For example, if a given pair of initial conditions are to make the machine leave the symbol 1 on the scanned tape frame, cause the machine to assume state S_2 and move the printing head one frame to the left, the instruction is abbreviated $(1,S_2,L)$.

The best way to understand how a Turing machine works is to try to build one. In this context building a Turing machine means to construct a table of instructions that specify the action of the Turing machine for every possible pair made up of one state and one tape

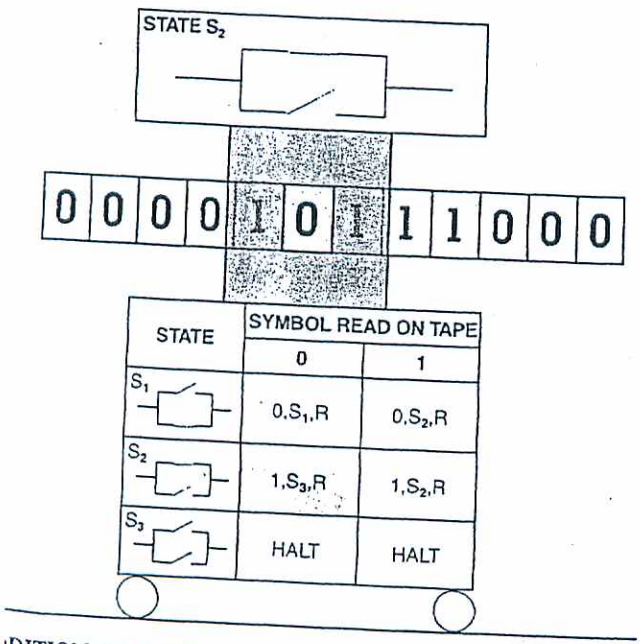
1



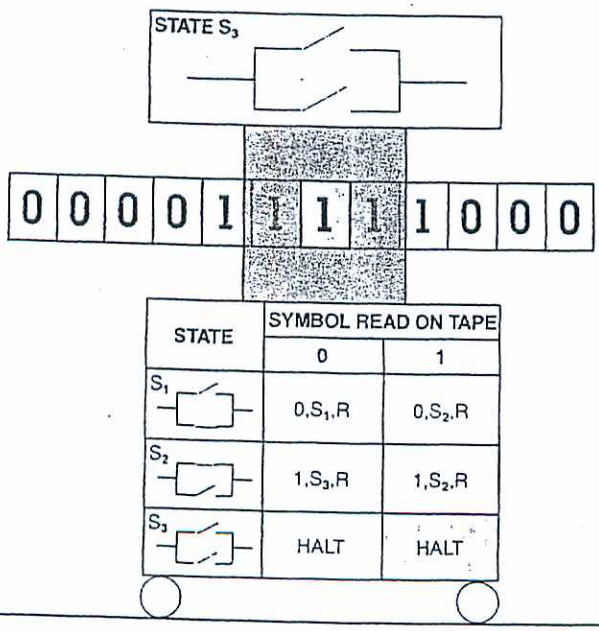
2



3



4



'DITION OF 2 AND 3 is accomplished by a Turing machine in r steps. Each number to be added is represented on a tape in unary ation: as a string of 1's bounded by 0's at both ends. The machine register the content of one tape frame at a time (*colored frames*) huttling to the left or to the right across the tape in a series of dis- e moves. The goal of the machine is to generate a string of five ecutive 1's and halt. The table of instructions, shown in the low- art of each machine in the diagram, is a fixed set of moves for all ible initial conditions and gives a procedure for adding any two bers. Following the instructions, the machine removes the 0 sep- ng two strings of 1's and shifts the left string one frame to the right to join the right string. The number of initial conditions available in the table of instructions must be large enough to meet all the contingencies that might arise on the tape; that number can be increased by increasing the number of internal states, or configurations, that are built into the Turing machine. For every possible combination of tape symbol and machine state the table of instructions must either halt the machine or specify three variables. First, it must give the symbol that is to be left in the frame of the tape currently being registered; second, it must specify the state the machine is to assume before it registers another tape frame, and third, it must indicate whether the machine is to move across the tape one frame to the left or to the right.

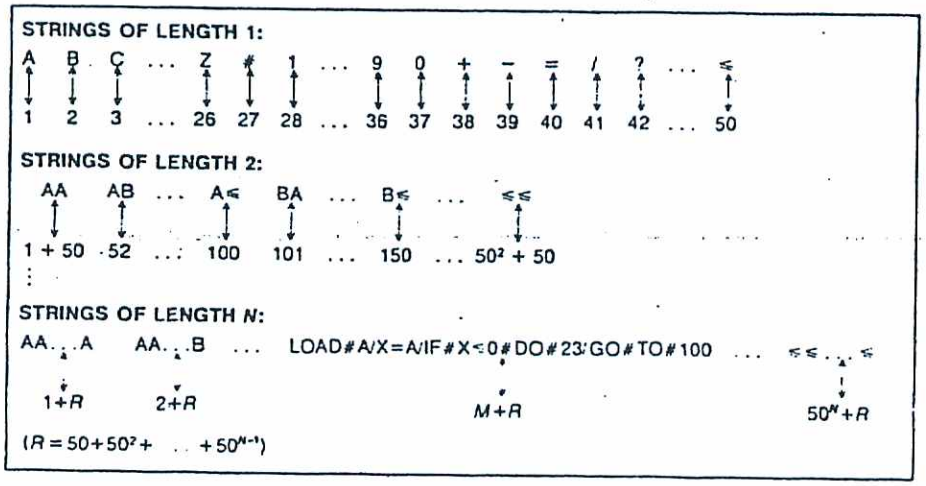
symbol. In practice the construction of the table also means granting enough possible states to the machine for it to do the task at hand.

The Adding Machine

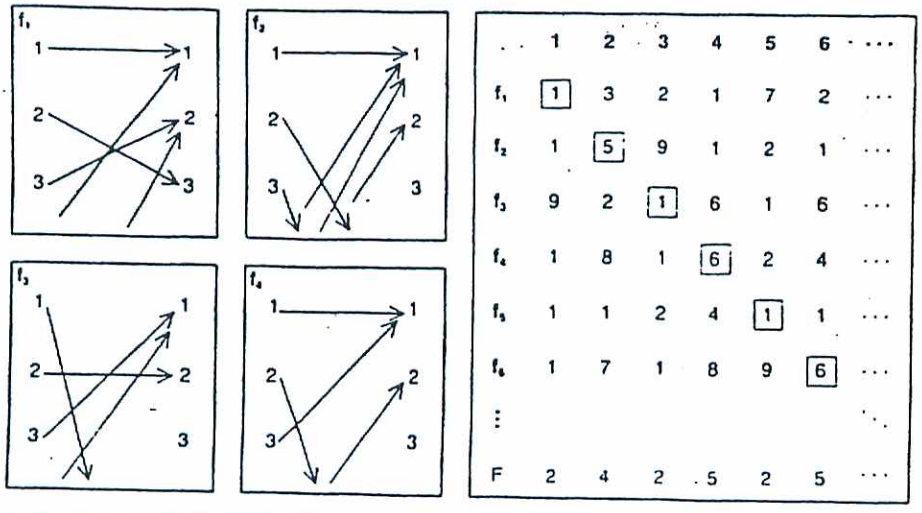
Consider how one might go about designing a Turing machine that adds two numbers and halts. It is customary, although by no means essential, to allow only two symbols to be printed on the tape, say 0 and 1. Any given number N can then be represented on the tape by a string of N 1's. If two numbers M and N are to be printed on the tape, they can be represented by a string of M 1's, followed on the right by a 0, followed in turn on the right by a string of N 1's. Assume the Turing machine is in its initial state S_1 and its printing head is scanning the leftmost 1 in the string of M 1's. To construct a Turing machine that adds, what one wants is to generate a table of instructions that will cause the Turing machine eventually to print a string of $M + N$ 1's and then halt.

One simple way to accomplish the task is to shift the leftmost string of M 1's one frame to the right. If the shift is done properly, the 0 no longer separates the two strings, and the single string that results has length $M + N$. It is not possible for the Turing machine to shift the string of 1's all at once; for example, the printing head can advance only one frame at a time. One way to get the result is to create a series of instructions for a Turing machine with three states. In the first state the printing head scans the tape from left to right, one frame at a time, until it reaches the leftmost 1. It changes the 1 to a 0, enters the second state and continues moving to the right. In the second state, when the head finds a 1 in the current frame, it takes no action—it does not change either the tape symbol or the machine state—except to move one more frame to the right. In this way the head scans past the $M - 1$ 1's remaining in the first string. When a 0 is finally found, the instructions cause the head to change the 0 to a 1 and halt.

The reader is now invited to construct a Turing machine that will find the product of any two given numbers. The conventions for the input are the same as they were for the machine that adds: the two integers to be multiplied are represented by two consecutive strings of 1's, separated by a 0. The output, a string of 1's whose length must be equal to the product of the first two strings, can be set off as a third string to the right of the first two. One working design is given in the illustration on page 91. It is fair to say at the outset that the construction is tricky and requires some careful bookkeeping. It also seems fair to give a hint, which the puzzle fancier can avoid by not reading the next paragraph.



CHARACTER STRINGS of any finite length can be paired one for one with the positive integers, provided each character in the string is selected from some finite set. According to the definition formulated by the German mathematician Georg Cantor, such a pairing implies the two sets are equivalent in size, even though both are infinite sets. Any infinite set whose elements can be paired one for one with the positive integers is called a countable set. Because every computer program and every possible table of instructions for a Turing machine can be encoded as a finite string of symbols, the pairing shows that the number of possible computer programs and the number of possible Turing machines are both countably infinite numbers.

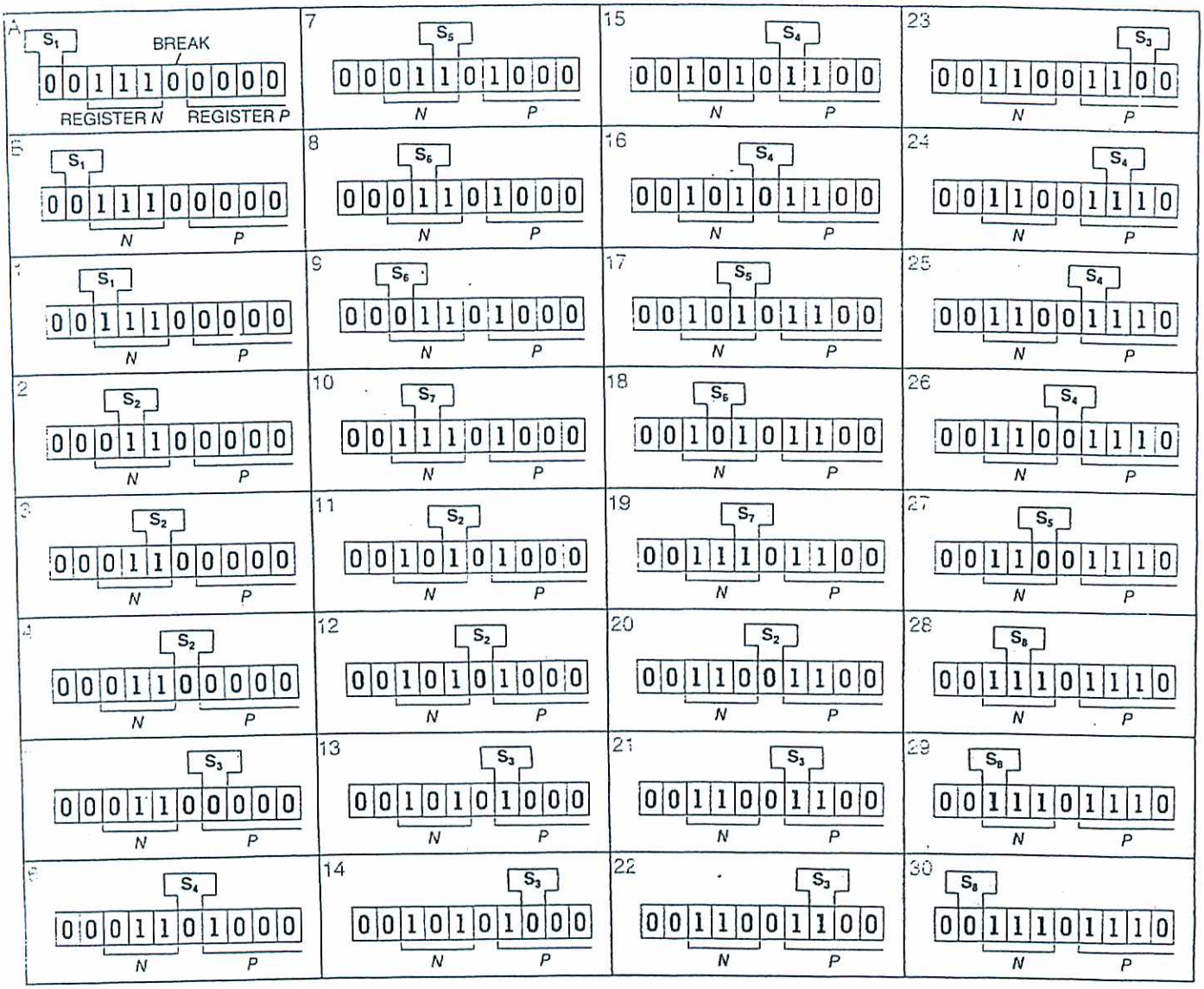


SET OF MATHEMATICAL FUNCTIONS of the positive whole numbers that take integer values is a noncountable set; that is, there are too many such functions for them to be paired one for one with the infinite set of whole numbers. Suppose, on the contrary, all the functions could be named according to their subscripts and then listed in a fixed order. The value of the function corresponding to each integer could then be given as an infinite row of digits. Since a function is determined by its infinite row of digits, consider a function constructed by changing the first digit in the first row, the second digit in the second row and so on. The row of digits that define the constructed function (*color*) must differ from any row of digits in the original list by at least one digit. Hence the supposition that all the functions could be listed leads to the contradiction that a new function not in the original list can always be constructed.

Building Complex Programs

Multiplication is much easier to do if one first develops a routine for copying a string of 1's immediately to the right of some given point. It is always possible to copy N 1's with an N-state machine; the states can effectively count the 1's in the string. Since the number of states must not grow indefinitely large, however, it is desirable to find a way to copy the string without counting its elements. One way is to send a marker across the string from left to right; the marker is a 0 that successively takes the place of each 1 in the string. For each advance of the marker from one frame to the next the printing head is instructed to scan past the right end of the string, skip over a 0 that indicates spacing and then change the next 0 it finds to a 1. If the printing head shuttles back and forth once for each advance of the marker, a new string of N 1's is written immediately to the right of the first string [*see illustration on opposite page*].

With a little practice one quickly learns how to build Turing machines that can run simple computational routines and how to combine those ma-



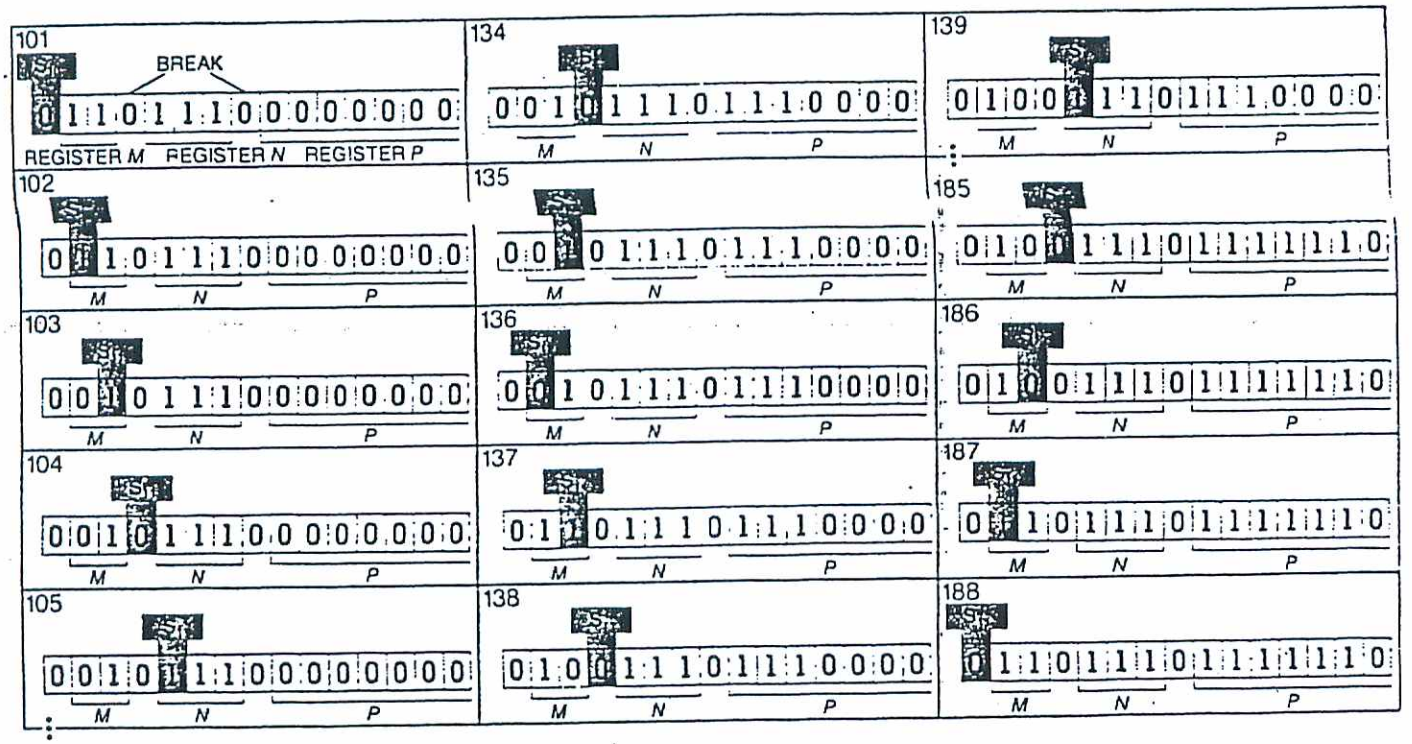
STATE	TAPE SYMBOL 0	COMMENT	SAMPLE MOVES	TAPE SYMBOL 1	COMMENT	SAMPLE MOVES
S_1	0, S_1, R	Begin with right shift to left end of register *N*.	A, B	0, S_2, R	Mark left end of register *N* with 0.	1
S_2	0, S_3, R	Right shift across break.	4, 12, 20	1, S_2, R	Right shift across register *N* from marker to break.	2, 3, 11
S_3	1, S_4, L	Copy 1 at right end of register *P*.	5, 14, 23	1, S_3, R	Right shift across 1's in register *P*.	13, 21, 22
S_4	0, S_5, L	Left shift across break.	6, 16, 26	1, S_4, L	Left shift across register *P*.	15, 24, 25
S_5	1, S_8, L	0 marker detected to the immediate left of break, indicating that a complete copy of register *N* has been added to register *P*.	27	1, S_6, L	Left shift from break one unit into register *N*. State cannot remain S_5; if it did, the eventual encounter with the 0 marker would cause the Turing machine to halt prematurely.	7, 17
S_6	1, S_7, R	Begin move of 0 marker one unit to right.	9, 18	1, S_6, L	Continue left shift across register *N* to 0 marker.	8
S_7		Condition is not encountered, hence any dummy instruction or none at all may be inserted.		0, S_2, R	Complete move of 0 marker one unit to right; repeat cycle.	10, 19
S_8	0, —, —	Copy routine completed; stop or wait for new instruction.	30	1, S_8, L	Left shift to left end of register *N*.	28, 29

COPYING TURING MACHINE is a component of more elaborate devices. Given any string of 1's marked on a tape, the machine writes a second string with the same number of 1's to the right of the 0 that marks the end of the first string. Here the machine copies three 1's.

thines in order to carry out more complex calculations. For example, a polynomial expression can be evaluated by combining the routines for adding, copying and multiplying. Even more versatile are short, elementary routines for symbol manipulation, such as "Move the printing head to the right until it encounters a 0" and "Move the marker in the leftmost string of 1's one frame to the right." Variations of these short routines are exploited both in the Turing machine that copies a string of 1's and in the machine that multiplies.

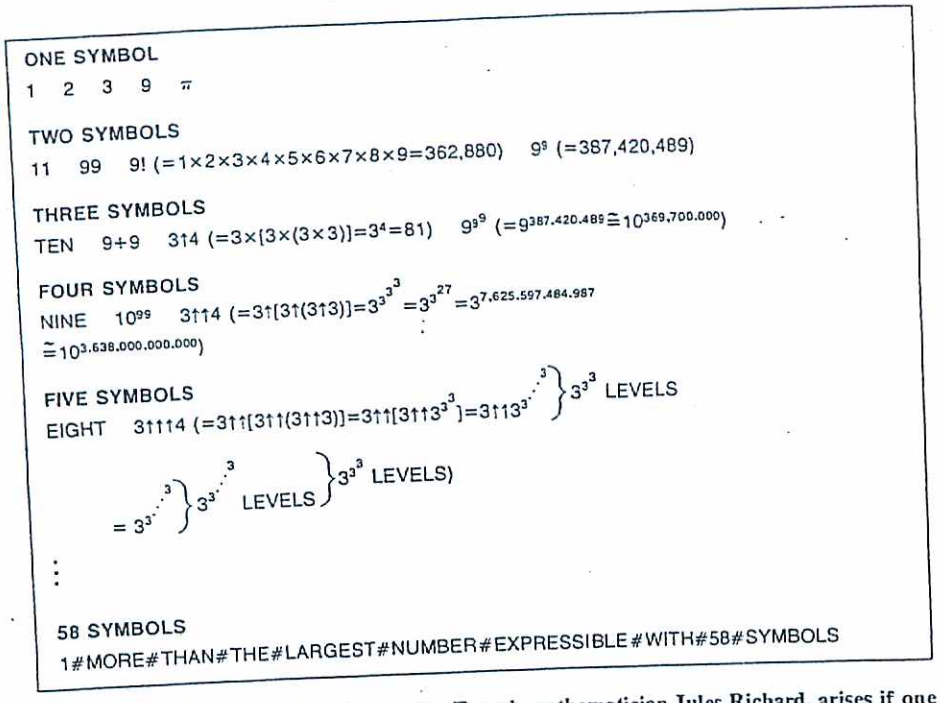
The Universal Turing Machine

If one is inclined to try building, say, the Turing machine that multiplies, one soon begins to appreciate the difficulties that must be faced in the design of a useful computer program. Most small Turing machines, namely the ones with only a few possible states, do not carry out any useful or even sensible task. Many of them get caught in infinite loops and shuttle back and forth on a tape indefinitely without halting. From among the machines that do perform reasonable tasks, one must choose a combination of machines that work together efficiently. The initial impression can be that the simplest tasks are fiendishly difficult and that realistic computation is hopeless. Such difficulties can be frustrating, but they are merely techni-



STATE	TAPE SYMBOL 0	COMMENT	SAMPLE MOVES	TAPE SYMBOL 1	COMMENT	SAMPLE MOVES
S_{10}	0, S_{10}, R	Begin with right shift to left end of register *M*.	101	0, S_{11}, R	Mark left end of register *M* with 0.	102
S_{11}	0, S_1, R	Right shift across break between register *M* and register *N*; begin copy routine *(see illustration on page 89)*.	104, 138	1, S_{11}, R	Right shift across register *M* from marker to break.	103
S_8	0, S_{12}, L	Left shift across break between register *N* and register *M*; end of copy routine.	134, 185	1, S_8, L	Left shift to left end of register *N*.	Not shown
S_{12}	1, S_{15}, L	0 marker detected to the immediate left of break between register *N* and register *M*, indicating that the product of the numbers stored in registers *M* and *N* has been stored in register *P*.	186	1, S_{13}, L	Left shift from break between register *N* and register *M* one unit into register *M*. State cannot remain S_{12}; if it did, the eventual encounter with the 0 marker would cause the Turing machine to halt prematurely.	135
S_{13}	1, S_{14}, R	Begin move of 0 marker in register *M* one unit to right.	136	1, S_{13}, R	Continue left shift across register *M* to 0 marker.	Not shown
S_{14}		Condition is not encountered; hence any dummy instruction or none at all may be inserted.		0, S_{11}, R	Complete move of 0 marker in register *M* one unit to right; repeat cycle.	137
S_{15}	0, —, —	Multiplication routine completed; stop or wait for new instruction.	188	1, S_{15}, L	Left shift to left end of register *M*.	187

MULTIPLICATION can be done with a Turing machine by embedding in it another Turing machine that can copy a string of 1's. In the example the machine requires 88 cycles to find the product of 2 and 3; the breaks in the numbering are machine cycles that jump to states defined for the machine that copies a string of 1's *(see illustration on page 89)*. In the last cycle the product of 2 and 3 is displayed as a string of six 1's in the section of the tape called the *P* register, immediately after the 0 that separates it from the two multipliers.



RICHARD PARADOX, named after the French mathematician Jules Richard, arises if one supposes the positive integers can be ordered, or listed, according to the number of symbols needed to specify them. Isolated integers of remarkable size can be designated with the help of special symbols such as the arrow notation introduced by Donald E. Knuth of Stanford University. Nevertheless, according to its own description, the number allegedly specified above with 58 symbols is larger than any number that can be specified with 58 symbols, which is a paradox. A similar paradox arises if one tries to find a given Turing machine in a list of Turing machines arranged according to the length of the string of symbols needed to encode each machine.

cal; with a few well-chosen routines the power of the Turing machine for solving problems increases explosively, and one is struck not by the weakness of the machine but by its potential. As Turing was able to show, it is possible to combine simple Turing machines into a machine that can carry out any task that can be explicitly described.

The electronic computer, which in part owes its existence to Turing's conceptual machines, is now probably the most convincing demonstration of those machines' computational power. In the course of his work Turing pointed out that any Turing machine M can be encoded on a tape as a sequence of 0's and 1's. The fundamental reason the encoding can be done is that every Turing machine is uniquely defined by its table of instructions; that table must be finite in length because both the machine states and the alphabet of tape symbols are finite in number.

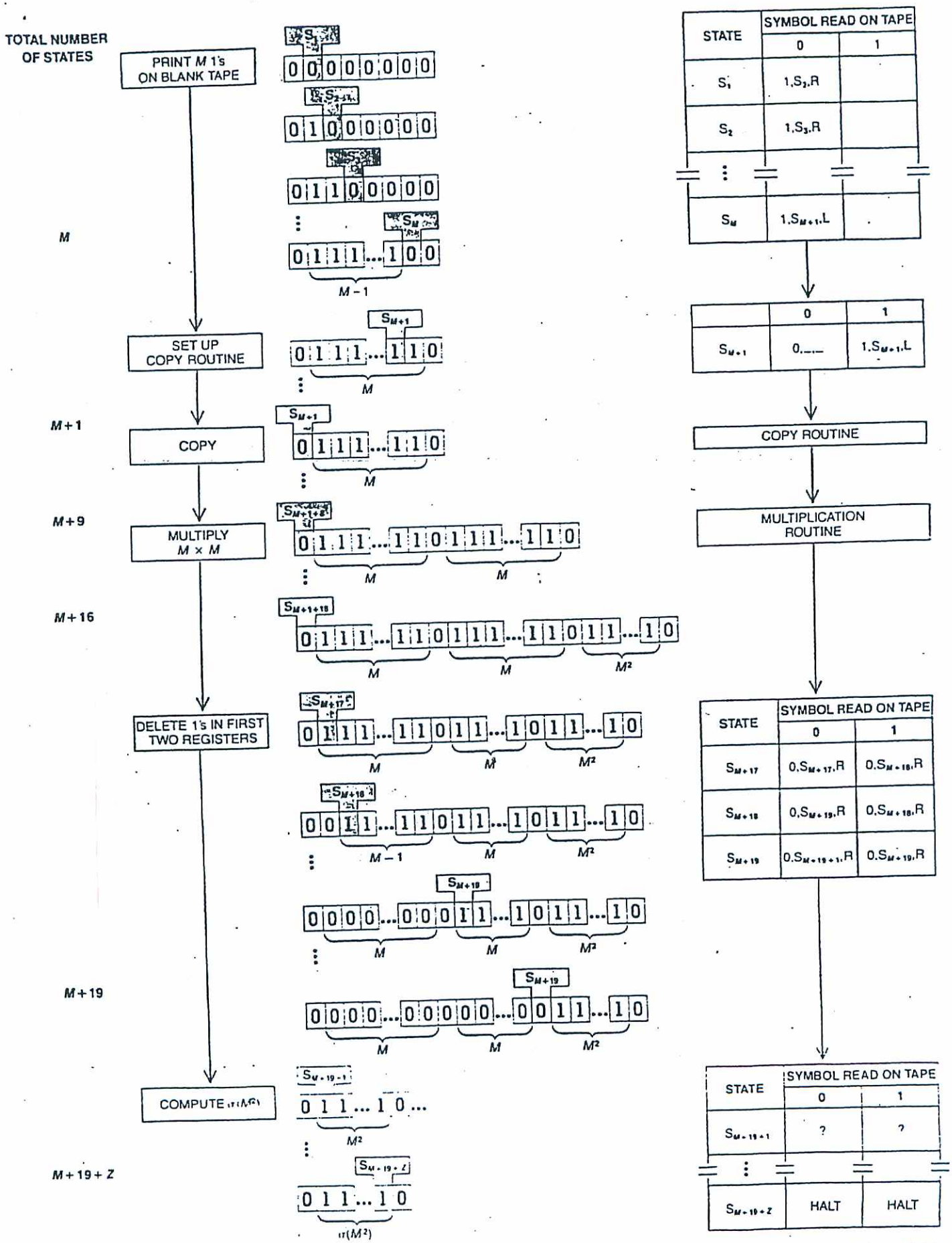
NUMBER OF STATES	MAXIMUM NUMBER OF PRINTED 1's	LOWER LIMIT FOR VALUE OF σ
3	$\sigma(3)$	6
4	$\sigma(4)$	12
5	$\sigma(5)$	17
6	$\sigma(6)$	35
7	$\sigma(7)$	22,961
8	$\sigma(8)$	$3^{3^2} \cong 7.9 \times 10^{43}$
9	$\sigma(9)$	$3^{3^2}+1$
10	$\sigma(10)$	$a^{a^{\cdot^{\cdot^{\cdot^a}}}}\} a^{a^{\cdot^{\cdot^{\cdot^a}}}}\} \cdots \} a^a$

"BUSY BEAVER" PROBLEM is to find the maximum number of 1's that can be printed by an N-state Turing machine that begins its operation on a tape initially filled with 0's and eventually comes to a halt. The number, which depends on N, is the value of a function designated $\sigma(N)$. In 1962 Tibor Rado of Ohio State University proved the function grows too fast to be computable. Lower bounds on the function estimated for small values of N are shown. The lower bound for $\sigma(10)$ can be expressed by a number a whose value is approximately $\sqrt{8}$; $\sigma(10)$ is an exponentiated stack of a's, where the number of a's in the stack is expressed by another stack of a's. The process of defining the height of one stack by another is carried out 10 times.

Turing showed that the operation of Turing machine M on any sequence of tape symbols X can be simulated by another Turing machine called the universal machine. The symbols on the tape registered by the universal Turing machine are grouped into two major sections: at the left is the encoded description of Turing machine M and at the right is the sequence of symbols X that would be encountered by M as it scanned its own tape. The universal machine is then constructed so that its printing head shuttles back and forth between the left and right sections of the tape. Through an elaborate system of markers the universal machine keeps track of the encoded state of M that is being consulted. Turing proved that the effect of the universal machine on the sequence of symbols X is exactly what the effect of M would be on the same sequence of tape symbols.

The successful simulation of Turing machine M by the universal Turing machine depends only on the fact that M is a machine that can be described exhaustively by a finite number of symbols. In principle, however, every digital computer can be described in the same way. The computer has a large but finite number of internal states, and its response to input data is entirely determined by the finite set of statements that make up its programming. Hence a complete description of any digital computer can be encoded on a tape as a sequence of 0's and 1's, and any input data can be encoded on the tape to the right of the description of the computer. By alternately consulting the description of the computer and the string of input data on the tape, the universal Turing machine can simulate the action of the computer on the input data step by step.

Given enough memory to serve as a tape for symbol manipulation, any real computer can play the role of the universal Turing machine. For example, if a home microcomputer were programmed to function as a universal Turing machine, and if a description of a large, "mainframe" computer were encoded on its input data, the microcomputer would simulate the action of the large computer on any string of data symbols. In this sense every digital computer can compute exactly the same class of mathematical functions, namely all the functions that are computable by some Turing machine. The existence of only one such class of functions strongly supports Turing's formal definition of computability: A mathematical function is computable if it can be computed by some Turing machine. Turing argued persuasively that his definition is equivalent to any reasonable interpretation of



PROOF THAT $\sigma(N)$ IS NONCOMPUTABLE begins with an estimate of the number of states needed to generate a string of M^2 1's on a tape initially filled with 0's. The Turing machines for copying and multiplying, shown in the illustrations on pages 89 and 91, are combined to form a machine that has $M + 16$ states. Three additional states delete the two leftmost strings of 1's, leaving a string of M^2 1's. It is assumed there is a Z-state machine that, given any string of N 1's, will generate a string of $\sigma(N)$ 1's (*type and tables in color*).

the intuitive concept of computability. It is probably worth mentioning, however, that it is pointless to ask for a rigorous mathematical proof that a formal definition such as Turing's fully captures some originally intuitive notion.

The Hilbert Program

In order to understand why Turing was so intent on defining computability it is necessary to have some sense of the history of mathematical logic before 1936. Rigorous mathematics as it is known today is a relatively recent development. The first serious attempt to reduce mathematical statements to statements in formal logic was begun by Gottlob Frege in 1879, with the publication of his *Begriffsschrift* ("The Notation of Concepts"). The problem posed by Hilbert in 1900 was therefore of direct significance to mathematics: if Frege's scheme could be carried to completion and if a method could be found for determining the truth or falsity of any statement in formal logic, then that method could also determine the truth or falsity of any mathematical statement, no matter how complex. If such a method could be found, mathematical conjectures such as Pierre de Fermat's "last theorem," which had resisted proof or disproof for centuries, would immediately be disposed of. An affirmative answer to Hilbert's bold challenge would reduce all mathematics to mechanical computation.

Two major developments in logic in the early decades of this century threw much of Hilbert's program into disarray. In 1901 Bertrand Russell discovered an irrefutable paradox in the elementary theory of sets, a theory that was essential to Frege's program of reducing mathematics to logic. Russell communicated his discovery to Frege just as the second volume of Frege's last major work, *Grundgesetze der Arithmetic* ("The Fundamental Laws of Arithmetic"), was about to be published. Frege ended the volume with a dispirited note: "A scientist can hardly meet with anything more undesirable than to have the foundation give way just as the work is finished. I was put in this position by a letter from Mr. Bertrand Russell when the work was nearly through the press." In spite of the flaws Frege was unable to remove from his work, Russell and Alfred North Whitehead were later able to salvage Frege's program and circumvent the paradox in set theory.

A second major discovery in logic was made by Kurt Gödel of the Institute for Advanced Study in Princeton, N.J. Implicit in Hilbert's program was the assumption that there must exist some method for distinguishing true statements from false ones in formal logic; the problem was to find the method. Gödel showed that the assumption is unjustified. In 1931 he proved that any consistent system of formal logic powerful enough to formulate statements in the theory of numbers must include true statements that cannot be proved. Because consistent axiomatic systems such as the one devised by Russell and Whitehead cannot encompass all the true statements in the subject matter they seek to formalize, such systems are said to be incomplete.

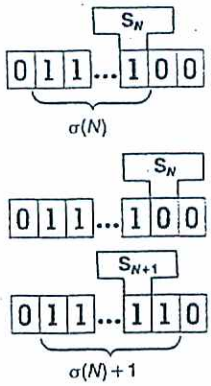
The Logic of Computability

Gödel's work effectively put an end to Hilbert's program. There can be no method for deciding whether some arbitrary statement in mathematics is true or false. If there were, the method would constitute a proof of all the true statements, and Gödel had demonstrated that within a consistent axiomatic system such a proof is impossible. The attention of logicians shifted from the concept of truth to the concept of provability. In this context there remained a simple analogue to Hilbert's question that had not been settled: Does a single method exist whereby all the provable statements in mathematics can be proved from a set of logical axioms?

The preeminent investigator of the logic of provability in the years immediately following Gödel's proof was Alonzo Church of Princeton University. Church and two of his students, Stephen C. Kleene and J. Barkley Rosser, developed a consistent formal language called the lambda calculus; it is useful for reasoning about mathematical functions, such as the square root, the logarithm and any more complicated functions that might be defined. (Lambda, the Greek letter corresponding to the Roman letter L, was chosen by Church to suggest that his formal system is a language.) The modern language for computer programming called Lisp (for list processing) is modeled on the lambda calculus. Kleene showed that large classes of mathematical functions, including all the functions employed by Gödel in his proof, could be expressed in the lambda calculus.

The next major step in this line of thought was taken by Church. He argued that if a mathematical function can be computed at all—meaning that it can be evaluated for every number in its domain of definition—then the function can be defined in the lambda calculus. Church's work showed that if there were such a thing as a mathematical function expressible in the lambda calculus that is not computable, there would be no method for determining whether or not a given mathematical statement is provable, let alone true. The final surviving

First notice that σ is uniformly increasing; that is, if $X>Y$, $\sigma(X)>\sigma(Y)$. No matter how many 1's are printed by an N-state Turing machine that halts, an $(N+1)$-state Turing machine can always be constructed that will add one 1 to the string of 1's and then halt.



STATE	SYMBOL READ ON TAPE	
	0	1
S_N	$1,S_{N+1},L$	$1,S_N,R$
S_{N+1}	HALT	HALT

Suppose there is a Z-state Turing machine that computes $\sigma(M^2)$.
By the definition of σ and the calculation shown in the illustration on the opposite page.

$$\sigma(M+19+Z) \geq \sigma(M^2). \qquad (1).$$

If M is large enough, however, statement (1) contradicts the fact that σ is uniformly increasing.
To demonstrate this, let $M = Z+20$, or $Z = M-20$.
Then $M+19+Z = M+19+M-20 = 2M-1$.
By elementary algebra $2M-1 = M^2-(M-1)^2$, and so $M+19+Z = M^2-(M-1)^2$.
Since $(M-1)^2 > 1$, $M^2-1 > M^2-(M-1)^2$.
Hence, because σ is uniformly increasing,

$$\sigma(M^2-1) > \sigma(M^2-[M-1]^2) = \sigma(M+19+Z) \geq \sigma(M^2).$$

Since $\sigma(M^2-1) > \sigma(M^2)$, however, σ cannot be uniformly increasing, and so the supposition that there is a Z-state Turing machine that computes $\sigma(M^2)$ leads to a contradiction.

FINAL STEPS IN PROOF that $\sigma(N)$ is noncomputable derive a contradiction from the assumption there is some Z-state Turing machine that, given a string of M^2 1's, computes $\sigma(M^2)$.

hypothesis in Hilbert's program would be disproved. In April, 1936, Church published a logical formula that is not computable in his system.

Turing, working independently of Church, had also grasped the technical connection between Hilbert's problem and the idea of a computable function. In attacking the problem, however, he proceeded in a far more direct and concrete manner than Church. What was needed was a simple but precise model of the process of computation, and Turing's machines were designed to meet that need. Once their properties were specified, however, Turing made a brilliant connection between the idea of a computable function and the results of mathematical work done some 50 years earlier by the German mathematician Georg Cantor. Cantor had argued that although there is no largest whole number, any infinite set of objects that can be counted, or paired with the positive whole numbers, is a set of the same size as the set of whole numbers. Since any Turing machine can be expressed as a character string of finite length, all possible Turing machines and with them all computable functions can be listed in numerical or alphabetical order; hence they can be paired one for one with the whole numbers [*see upper illustration on page 88*]. There is, of course, no fixed upper limit to the size of a Turing machine, and so there is no limit to the number of possible Turing machines. Nevertheless, Cantor's analysis shows that the set of all possible computable functions is the same size as the set of all whole numbers; both sets are called countable sets.

Cantor had also shown there are infinite sets that are not countable; they are larger than the set of whole numbers in the sense that they cannot be paired one for one with the whole numbers. One example of such a noncountable set is the set of all the functions of the positive whole numbers that take on integer values. A careful analysis shows there must be more such functions than there are whole numbers. The implication is that not all functions are computable: there are not enough computer programs to compute every possible function.

The Halting Problem

Which functions are noncomputable? Unfortunately the proofs by Church and by Turing do not readily yield examples of noncomputable functions. In the past 20 years, however, computer scientists have exploited Turing machines to construct several such functions. One of the early examples of a noncomputable function was devised by Tibor Rado of Ohio State University in 1962. Consider all the Turing machines that have some fixed number of states N. Suppose all the machines begin their operations on a blank tape, or in other words a tape on which a 0 is marked in every frame. Imagine for the moment that all the Turing machines that never halt are excluded from this set. Among the remainder of the Turing machines, pick the machine or the group of machines that print the largest number of 1's in succession on the blank tape before they halt. That number of 1's, for each value of N, is the value of Rado's function; it is usually designated $\sigma(N)$. The detailed proof that $\sigma(N)$ is not computable proceeds by assuming it can be computed and then deriving a contradiction. The argument is straightforward, but the technical details are rather intricate; they can be found in the illustrations on the preceding two pages.

One might think the construction defective because it assumes the N-state Turing machines that do not halt can be sorted out in advance. The objection is a serious one. Consider, therefore, how one might attempt to compute $\sigma(N)$ by brute force. List all the N-state Turing machines in some numerical order, simulate each one on a universal Turing machine and select the machine or the group of machines that print the most 1's. Although this method of computation seems to avoid the objection, the difficulty with Turing machines that do not halt reappears in an intractable form. Some of the N-state machines that do not halt can be eliminated by simple algorithms, but there are other machines for which no such decision can be made. If one cannot determine that a particular machine does not halt, the machine cannot be eliminated from the list of N-state machines and the simulation must continue. Since the machine may actually never halt, there is no guarantee the computation of $\sigma(N)$ can be carried through to completion.

Although the early impact of the Turing machine was in logic, it has also played a dominant role in computer science since the early 1960's. In 1965 Juris Hartmanis and Richard E. Stearns, then at the General Electric Research Laboratories in Schenectady, N.Y., showed that the Turing machine can help to establish tight bounds on the complexity of computations. Subsequent investigators began to classify problems according to the way in which the running time, or equivalently the number of computational steps, varies with the size of the problem. For example, suppose some number of points N are interconnected by lines to form a graph of vertexes and lines. The problem is to color the vertexes in such a way that no two vertexes connected by a line have the same color. Suppose furthermore the fastest method known for solving the problem requires a time that varies as some power of N, say N^2. The problem is then said to be in the class of problems that can be solved in polynomial time, designated P. The class P has increased in importance as many computer scientists have come to regard all problems that are not in P as intractable.

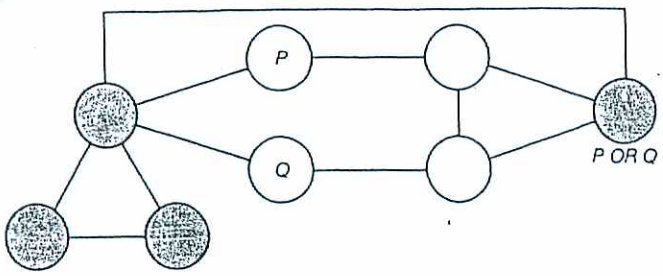
Modern Complexity Theory

Note that a problem is assigned to class P only if no instance of the problem requires more than polynomial time to solve. In other words, the method of solution for the problem is deterministic, in the sense that it guarantees a solution in a time less than some fixed power of the size of the problem, N. A nondeterministic Turing machine can also be defined: it is allowed to solve a problem by guessing the answer and then verifying the guess. For example, to determine whether an integer is composite, the nondeterministic machine guesses a divisor, divides and, if the division is exact, verifies that the number is composite. The deterministic machine, on

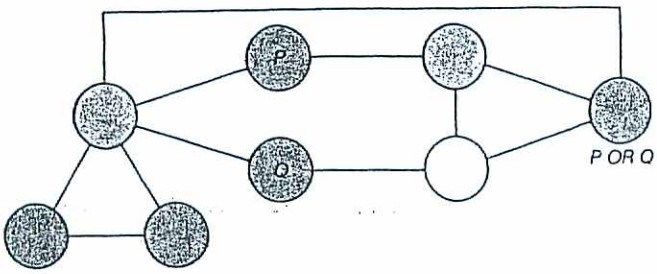
TWO PROBLEMS ARE EQUIVALENT if the solution to one problem immediately gives the solution to the other. Here the problem of determining the conditions under which a complex Boolean formula, or logical proposition, is true has been mapped, or transformed, into another problem, that of coloring the vertexes of a graph with three colors in such a way that no two vertexes connected by a line are the same color. The truth or falsity of any Boolean formula depends on the truth or falsity of the atomic, or simple, propositions that make it up and on the ways the truth values are combined by the connectives "or," "and" and "not." For every possible Boolean formula a simple graph of lines and vertexes can be constructed, which can be colored with three colors if and only if truth values can be given to the atomic propositions in such a way that the complex formula is true. For example, consider the complex formula *P or Q*, which is made up of the atomic proposition *P* and the atomic proposition *Q*. The formula *P or Q* is true only if proposition *P* is true, proposition *Q* is true or both propositions are true. These conditions are reflected in the coloring of the graph at the upper left (*a*), in which green represents true and blue represents false. The coloring can be completed in such a way that the rightmost vertex is green (true) only if either one or both of the labeled vertexes are colored green (*b–e*). Similarly, the complex formula *P and Q* is true only if the atomic propositions *P* and *Q* are both true, and that state of affairs can be reflected in a graph in which the vertex representing *P* and the vertex representing *Q* are both colored green (*f*). Finally, the formula *not-P* is true only if the vertex labeled *P* is colored blue (*g*), and the formula *P* is true only if the vertex labeled *not-P* is colored blue (*h*). At the lower right of the illustration is the graph that corresponds to the more complex Boolean formula (*P or Q*) *and* (*not-R*); the upper coloring (*i*) represents the constraints imposed by the connectives "or," "and" and "not." At the bottom (*j*) one of the four ways to color the graph is shown; it corresponds to one way of satisfying the Boolean formula.

OR

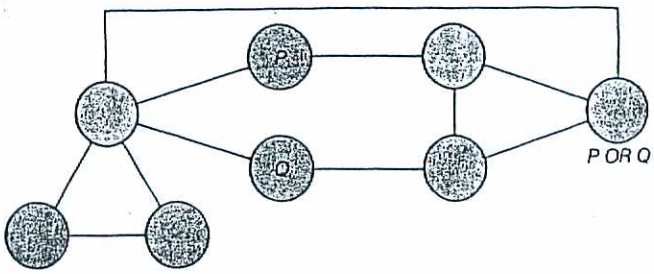
a



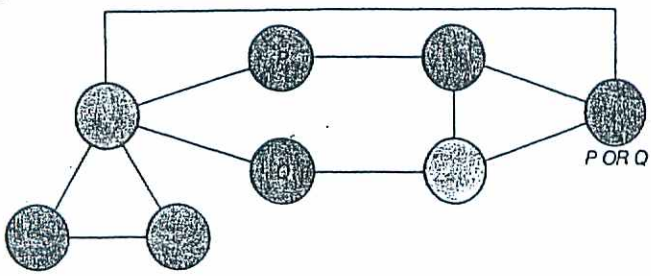
b



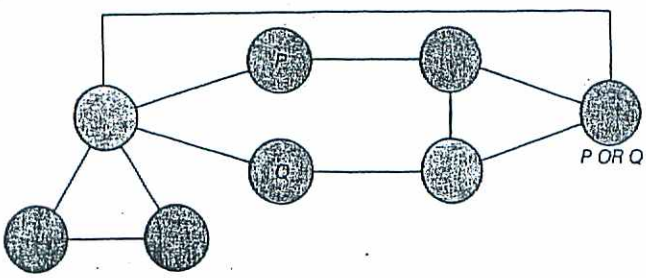
c



d

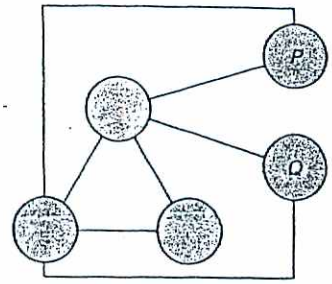


e



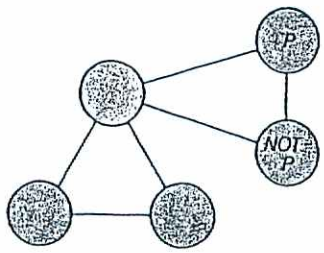
AND

f

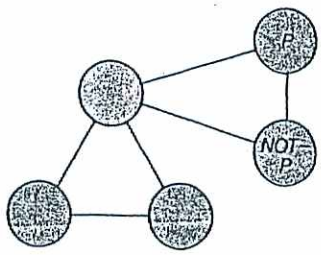


NOT

g

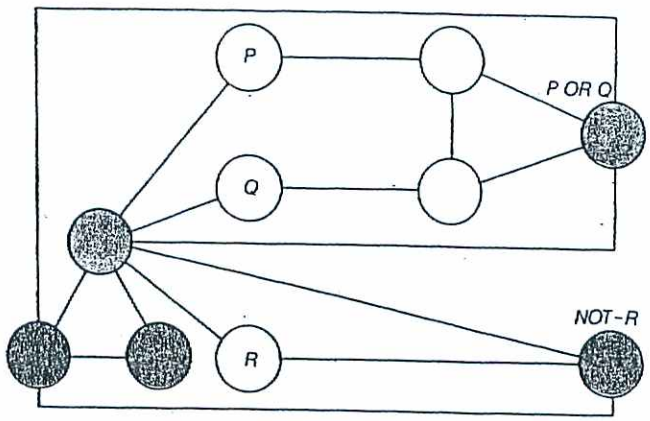


h

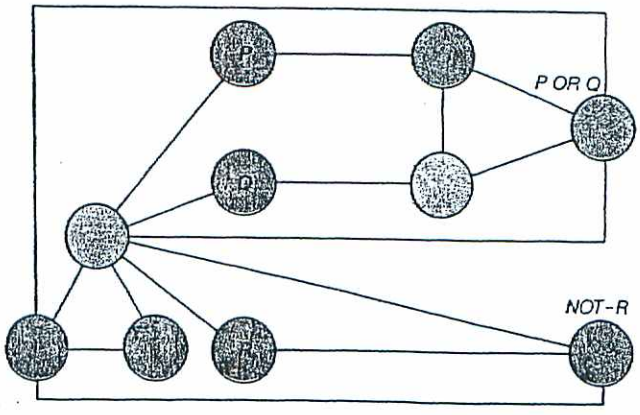


(P OR Q) AND (NOT-R)

i



j



the other hand, must search systematically for a divisor.

The time needed for solving a problem with a nondeterministic machine is measured by the length of the shortest computation, and so the nondeterministic machine would seem to have an enormous advantage over the deterministic one. Ordinary experience suggests it is easier to verify a solution than it is to find it in the first place. Nevertheless, no one has been able to prove that the problems solvable in polynomial time with a nondeterministic machine—the class of problems designated *NP*—are intrinsically any more difficult than the problems in the class *P*. Whether or not the class *P* is distinct from the class *NP*, which is called the *P-NP* problem, has become one of the major open questions in mathematics.

Important progress on the *P-NP* problem was made in 1970 by Stephen A. Cook of the University of Toronto. Cook was investigating the problem of determining the conditions under which a complex logical proposition is true. For example, the complex proposition formed when two simple propositions are linked by the word "or" is true if either one of the simple propositions is true or if both are true. In general it is quite difficult to determine the range of truth conditions for simple propositions that satisfy a complex proposition, or in other words make the complex proposition true. Cook was able to show that the problem, which is called the satisfiability problem, is as difficult as any other problem in class *NP*. There is an efficient algorithm for solving the satisfiability problem only if there is an efficient algorithm for solving every other problem in the class *NP*. Any problem having this property with respect to an entire class of problems is said to be complete for that class.

A year went by before most investigators grasped the significance of Cook's result. In 1971 Richard M. Karp of the University of California at Berkeley began to ask what other natural problems might play the same role as the satisfiability problem with respect to the class *NP*. Karp discovered that many important problems in operations research, including the problem of coloring a graph with three colors, are also as difficult as any problem that can be assigned to the class *NP*; that is, they are *NP*-complete. It can be shown directly, by mapping one problem into the domain of the other, that the graph-coloring problem and the satisfiability problem are equivalent [*see illustration on preceding page*].

It has now been proved in a similar way that several hundred problems, previously thought to be distinct, are actually notational variants of one another. All these problems are equivalent to the satisfiability problem and so all are *NP*-complete. Several other collections of such complete problems have been discovered, both for the class *P* and with respect to the classes of intractable problems for which the number of steps required for a solution on a Turing machine grows exponentially with the size of the problem. A direct assault on the *P-NP* problem, however, still seems premature. Some of the difficulty can be appreciated from recent developments in the theory of computation.

Relative Computability

The idea that a function is computable by a Turing machine can be extended by making computability depend on strings of symbols the machine may encounter on its tape. If a string found on the tape belongs to a previously specified set of strings *A*, the Turing machine can be instructed to advance to a special state that continues to compute the function in question. If the string does not belong to the set *A*, the machine delivers the decision that the function is not computable. The function is said to be computable relative to the set *A*. If the set *A* were made up of all the strings encoding Turing machines that eventually halt when they are given a blank tape as input, Rado's function $\sigma(N)$ would be computable relative to *A*.

In 1974 Theodore P. Baker of Cornell University, John Gill of Stanford University and Robert M. Solovay of Berkeley asked whether the nonequivalence of class *P* and class *NP* could be proved for relative computations. In the course of this work they made a startling discovery. They could specify two sets *A* and *B*, for which the relations between class *P* and class *NP* are contradictory. In other words, for computations relative to set *A*, both *P* and *NP* are equivalent, whereas for computations relative to set *B*, *P* and *NP* are not equivalent classes. Moreover, it was discovered that for any formal system there are relative computations for which one can assume either that *P* and *NP* are equivalent or that they are not, without detriment to the consistency of the system. Other investigators have since found that many other problems can be relativized in such a way that either of two possible outcomes is true.

This perplexing state of affairs is obviously unsatisfactory as it stands. No problem that has been relativized in two conflicting ways has yet been solved, and this fact is generally taken as evidence that solutions to such problems are beyond the current state of mathematics. Nevertheless, one must remember that the mere formulation of these seemingly intractable problems is made possible by the simple solution to an impenetrable problem of an earlier generation. The next major advance may seem as simple in retrospect as A. M. Turing's imaginative machines.