## Fine Hall Library

**From:** DocDel [docdel@Princeton.EDU]
**Sent:** Monday, February 02, 2004 9:14 AM
**To:** finelib@Princeton.EDU
**Subject:** Document Delivery Pull Slip

THIS ITEM WAS REQUESTED BY: Engineering on 2/2/2004 9:12:48 AM

Please scan and finish this article for Article Express located at SM. Thank you.

ILLiad Transaction Number: 115738

This request is for: Sanjeev Arora (arora)
Delivery Method: Hold for Pickup
Electronic Delivery: Yes

Call Number: QA7.M3447
Branch Location: SM

Journal Title: Mathematics Today—Twelve Informal Essays
Article Author: M. Davis
Article Title: What is a computation?

Journal Vol: Journal Issue:
Journal Month: Journal Year: 1978
Article Pages: 241-267

This request has been sent by bc - Borrowing.
ILLiad Transaction Number: 115738

Thank you,
Article Express Staff

Barb in ST

# What is a Computation?

*Martin Davis*

On numerous occasions during the Second World War, members of the German high command had reason to believe that the allies knew the contents of some of their most secret communications. Naturally, the Nazi leadership was most eager to locate and eliminate this dangerous leak. They were convinced that the problem was one of treachery. The one thing they did not suspect was the simple truth: the British were able to systematically decipher their secret codes. These codes were based on a special machine, the "Enigma," which the German experts were convinced produced coded messages that were entirely secure. In fact, a young English mathematician, Alan Turing, had designed a special machine for the purpose of decoding messages enciphered using the Enigma. This is not the appropriate place to speculate on the extent to which the course of history might have been different without Turing's ingenious device, but it can hardly be doubted that it played an extremely important role.

In this essay we will discuss some work which Alan Turing did a few years before the Second World War whose consequences are still being developed. What Turing did around 1936 was to give a cogent and complete logical analysis of the notion of "computation." Thus it was that although people have been computing for centuries, it has only been since 1936 that we have possessed a satisfactory answer to the question: "What is a computation?" Turing's analysis provided the framework for important mathematical investigations in a number of directions, and we shall survey a few of them.

Turing's analysis of the computation process led to the conclusion that it should be possible to construct "universal" computers which could be programmed to carry out any possible computation. The existence of a logical analysis of the computation process also made it possible to show that

## Alan M. Turing

Alan M. Turing was born in 1912, the second son in an upper class English family. After a precocious childhood, he had a distinguished career as a student at Cambridge University. It was shortly after graduation that Turing published his revolutionary work on computability. Turing's involvement in the deciphering of German secret codes during the Second World War has only recently become public knowledge. His work has included important contributions to mathematical logic and other branches of mathematics. He was one of the first to write about the possibility of computer intelligence and his writings on the subject are still regarded as fundamental. His death of cyanide poisoning in June 1954 was officially adjudged suicide.

certain mathematical problems are incapable of computational solution, that they are, as one says, *unsolvable*. Turing himself gave some simple examples of unsolvable problems. Later investigators found that many mathematical

problems for which computational solutions had been sought unsuccessfully for many years were, in fact, unsolvable. Turing's logical proof of the existence of "universal" computers was prophetic of the modern all-purpose digital computer and played a key role in the thinking of such pioneers in the development of modern computers as John von Neumann. (Likely these ideas also played a role in Turing's seeing how to translate his crytographic work on the German codes into a working machine.) Along with the development of modern computers has come a new branch of applied mathematics: *theory of computation*, the application of mathematics to the theoretical understanding of computation. Not surprisingly, Turing's analysis of computation has played a pivotal role in this development.

Although Turing's work on giving a precise explication of the notion of computation was fundamental because of the cogency and completeness of his analysis, it should be stated that various other mathematicians were independently working on this problem at about the same time, and that a number of their formulations have turned out to be logically equivalent to that of Turing. In fact the specific formulation we will use is closest to one originally due to the American mathematician Emil Post.

## The Turing–Post Language

Turing based his precise definition of computation on an analysis of what a human being actually does when he computes. Such a person is following a set of rules which must be carried out in a completely mechanical manner. Ingenuity may well be involved in setting up these rules so that a computation may be carried out efficiently, but once the rules are laid down, they must be carried out in a mercilessly exact way. If we watch a human being calculating something (whether he is carrying out a long division, performing an algebraic manipulation, or doing a calculus problem), we observe symbols being written, say on a piece of paper, and the behavior of the person doing the calculating changing as he notes various specific symbols appearing as results of computation steps.

The problem which Turing faced and solved was this: how can one extract from this process what is essential and eliminate what is irrelevant? Of course some things are clearly irrelevant; obviously it does not matter whether our calculator is or is not drinking coffee as he works, whether he is using pencil or pen, or whether his paper is lined, unlined, or quadruled. Turing's method was to introduce a series of restrictions on the calculator's behavior, each of which could clearly be seen to be inessential. However, when he was done all that was left were a few very simple basic steps performed over and over again many times.

We shall trace Turing's argument. In the first place, he argued that we can restrict the calculator to write on a linear medium, that is, on a tape, rather

## Emil L. Post

Emil L. Post was born in Poland in 1897, but arrived in New York City at the age of seven, and lived there for the remainder of his life. His life was plagued by tragic problems: he lost his left arm while still a child and was troubled as an adult by recurring episodes of a disabling mental illness. While still an undergraduate at City College he worked out a generalization of the differential calculus which later turned out to be of practical importance. His doctoral dissertation at Columbia University initiated the modern metamathematical method in logic. His researches while a postdoctoral fellow at Princeton in the early 1920's anticipated later work of Gödel and Turing, but remained unpublished until much later, partly because of the lack of a receptive atmosphere for such work at the time, and partly because Post never completed the definitive development he was seeking. His work in computability theory included the independent discovery of Turing's analysis of the computation process, various important unsolvability results, and the first investigations into degrees of unsolvability (which provide a classification of unsolvable problems). He died quite unexpectedly in 1954 while under medical care.

than on a two-dimensional sheet of paper. Instead of paper tape (such as is used in an adding machine) we can, if we prefer, think of magnetic tape as used in a tape recorder. (Of course, in this latter case, the symbols occur as magnetic signals rather than as marks on paper, but conceptually this makes no difference whatsoever.) It is easy to convince oneself that the use of a two-dimensional sheet of paper plays no essential role in the computational process and that we really are not giving up any computational power by restricting ourselves to a linear tape. Thus the "two-dimensional" multiplication:

$$\begin{array}{r} 26 \\ \times\ 32 \\ \hline 52 \\ 780 \\ \hline 832 \end{array}$$

can be written on a "tape" as follows:

$$26 \times 32 = 52 + 780 = 832.$$

We suppose that the linear tape is marked off into individual squares and that only one symbol can occupy a square. Again, this is a matter of convenience and involves no particular limitations. So, our multiplication example might look like this:

| 2 | 6 | × | 3 | 2 | = | 5 | 2 | + | 7 | 8 | 0 | = | 8 | 3 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

.

The next restriction we impose (here we are actually going a bit further than Turing did) is that the only symbols which may appear on our tape are 0 and 1. Here we are merely making use of the familiar fact that all information can be "coded" in terms of two symbols. It is this fact, for example, which furnishes the basis for Morse code in which the letters of the alphabet are represented as strings of "dots" and "dashes." Another example is binary arithmetric which forms the basis of modern digital computation.

Our next restriction has to do with the number of different symbols our calculator can take note of (or as we shall say, "scan") in a single observation. How many different symbols can a human calculator actually take in at one time? Certainly no one will be able to take in at a glance the distinction between two very long strings of zeros and ones which differ only at one place somewhere in the middle. One can take in at a glance, perhaps, five, six, seven, or eight symbols. Turing's restriction was more drastic. He assumed that in fact one can take in only a single symbol at a glance. To see that this places no essential restriction on what our calculator can accomplish, it suffices to realize that whatever he does as a result of scanning a group of, say, five symbols can always be broken up into separate operations performed viewing the symbols one at a time.

What kinds of things can the calculator actually do? He can replace a 0 by

a 1 or a 1 by a 0 on the square he is scanning at any particular moment, or he can decide to shift his attention to another square. Turing assumed that this shifting of attention is restricted to a square which is the immediate neighbor, either on the left or on the right, of the square previously scanned. Again, this is obviously no essential restriction: if one wants to shift one's attention to a square three to the right, one simply shifts one to the right three successive times. Also the calculator may observe the symbol in the square being scanned and make a decision accordingly. And presumably this decision should take the form: "Which instruction shall I carry out next?" Finally, the calculator may halt, signifying the end of the computation.

To summarize: any computation can be thought of as being carried out by a human calculator, working with strings of zeros and ones written on a linear tape, who executes instructions of the form:

Write the symbol 1
Write the symbol 0
Move one square to the right
Move one square to the left
Observe the symbol currently scanned and choose the next step accordingly
Stop

The procedure which our calculator is carrying out then takes the form of a list of instructions of these kinds. As in modern computing practice, it is convenient to think of these kinds of instructions as constituting a special *programming language*. A list of such instructions written in this language is then called a *program*.

We are now ready to introduce the Turing–Post Programming Language. In this language there are seven kinds of instructions:

PRINT 1
PRINT 0
GO RIGHT
GO LEFT
GO TO STEP i IF 1 IS SCANNED
GO TO STEP i IF 0 IS SCANNED
STOP

A Turing–Post program is then a list of instructions, each of which is of one of these seven kinds. Of course in an actual program the letter *i* in a step of either the fifth or sixth kind must be replaced by a definite (positive whole) number.

In order that a particular Turing–Post program begin to calculate, it must have some "input" data. That is, the program must begin scanning at a specific square of a tape already containing a sequence of zeros and ones. The

```
 1. PRINT 0
 2. GO TO STEP 2 IF 1 IS SCANNED
 4. PRINT 1
 5. GO RIGHT
 6. GO TO STEP 5 IF 1 IS SCANNED
 7. PRINT 1
 8. GO RIGHT
 9. GO TO STEP 1 IF 1 IS SCANNED
10. STOP
```

**Figure 1.** Doubling Program

symbol 0 functions as a "blank"; although the entire tape is infinite, there are never more than a finite number of ones that appear on it in the course of a computation. (A reader who is disturbed by the notion of an infinite tape can replace it for our purposes by a finite tape to which blank squares—that is, squares filled with zeros—are attached to the left or the right whenever necessary.)

Figure 1 exhibits a Turing–Post program consisting of ten instructions which we will use repeatedly for illustrative purposes. The presence of the "GO TO" instruction makes it possible for the same instruction to be executed over and over again in the course of a single computation. This can be seen in some detail in Figure 2 which shows the successive steps in one particular computation by the program of Figure 1. The computation is completely determined by the initial arrangement of symbols on the tape together with a specification of which square is initially scanned. In Figure 2 this latter information is given by an upward arrow ( ↑ ) below the scanned square. (Of course only a finite number of symbols from the tape can actually be explicitly exhibited; in Figure 3, we exhibit six adjacent symbols, and assume that all squares not explicitly shown are blank, that is contain the symbol 0.) Such combined information, consisting of the symbols on the tape (pictorially represented by showing a finite number of consecutive squares, the remainder of which are presumed to be blank) and the identity of the scanned square (designated by an arrow just below it) is called a *tape configuration.*

Figure 2 gives a list of such tape configurations, with the initial configuration at the top, each of which is transformed by an appropriate step of the program (from Figure 1) into the configuration shown below it. The program steps are listed alongside the tape configurations. The computation begins by executing the first step (which in our case results in replacing the 1 on the scanned square by 0) and continues through the successive steps of the program, except as "GO TO" instructions cause the computation to return to earlier instructions. Ultimately, Step 9 is executed with the tape configuration as shown at the bottom of Figure 2. Since 0 is being scanned, the computation continues to Step 10 and then halts.

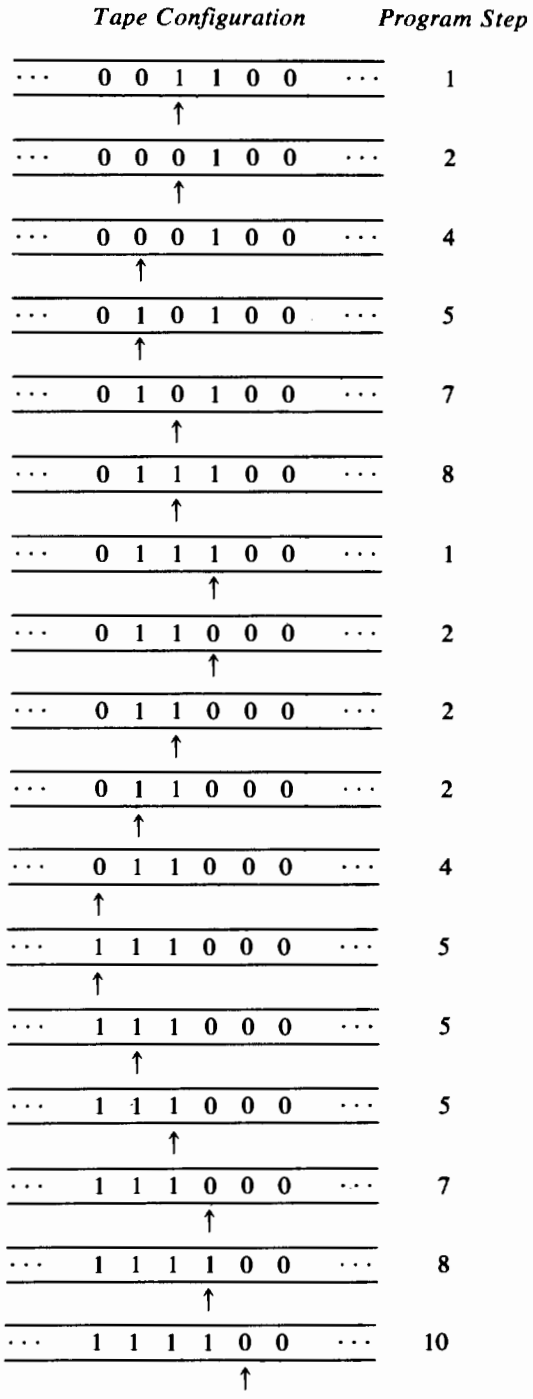



**Figure 2.** Steps in a Computation by Doubling Program

Given an alphabet of three symbols $a$, $b$, $c$, and three equations

$$ba = abc$$
$$bc = cba$$
$$ac = ca$$

we can obtain other equations by substitution:

$$\mathbf{ba}c = abcc$$
$$\mathbf{bac} = \mathbf{bca} = c\mathbf{baa} = cabca = c\mathbf{abca} = acbca = \ldots$$
$$or = cab\mathbf{ca} = cabac = \ldots$$
$$or = ca\mathbf{bca} = cacbaa = \ldots$$

(The expressions in boldface type are the symbols about to be replaced.) In this context can be raised questions such as: "Can we deduce from the three equations listed above that $bacabca = acbca$?" The word problem defined by the three equations is the general question: to determine of an arbitrary given equation between two words, whether or not it can be deduced from the three given equations.

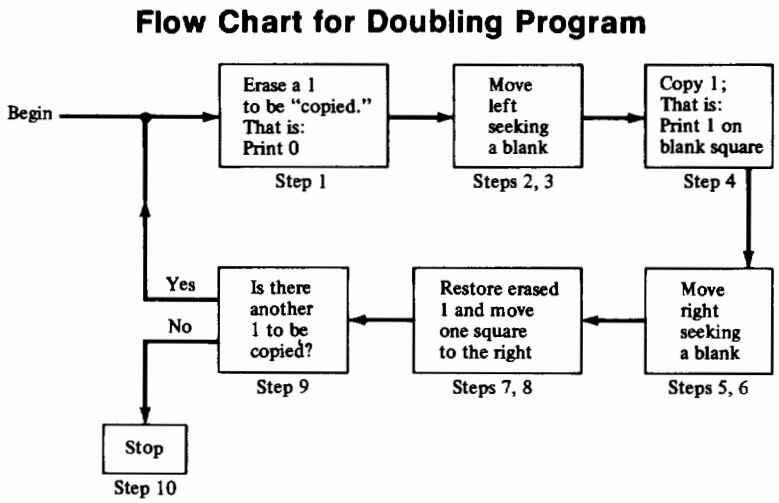
**Figure 3.** A Word Problem

The computation shown in Figure 2 begins with two ones on the tape and ends with four. It is because this happens generally that we call the program in Figure 1 a "doubling program." To put it precisely: beginning with a tape configuration the nonblank portion of which consists of a row of ones with the scanned square containing the leftmost of the ones, the doubling program will eventually halt with a block of twice as many ones on the tape as were there to begin with. It is by no means obvious at a glance (even to an experienced computer programmer) that our doubling program really behaves in the manner just stated. Readers who want to understand how the doubling program works may examine the "flow chart" in the box on p. 250.

The fact that this doubling program is so short and accomplishes such a simple task should not be permitted to obscure the point of Turing's analysis of the computation process: we have reason to be confident that any computation whatsoever can be carried out by a suitable Turing–Post program.

As we have seen, once a STOP instruction is executed, computation comes to a halt. If, however, no STOP instruction is ever encountered in the course of a computation, the computation will (in principle, of course) continue forever. The question "When can we say that a computation will eventually halt?" will play a crucial role later in our discussion. To see how this can be answered in a simple example, consider the following three-step Turing–Post program:

1. GO RIGHT
2. GO TO STEP 1 IF 0 IS SCANNED
3. STOP




The diagram above shows schematically how the doubling program listed in Figure 1 actually works. The underlying idea is to double the number of ones by simply copying them one at a time. Each 1 to be copied is (temporarily) replaced by a 0 which acts as a place marker (Step 1). Next the computation moves left over all the ones (which as the computation progresses will include newly printed ones) seeking the first unused (i.e., blank) square (Steps 2, 3—which will be repeated over and over again until a blank square is encountered). The 1 is now copied (Step 4). Next the computation returns rightward until it encounters the 0 which takes the place of the 1 which has just been copied (Steps 5,6—which again are repeated). The copied 1 is restored (Step 7). The computation moves one square to the right seeking another 1 to copy (Step 8). If there is another 1 to be copied the computation goes back to Step 1; otherwise it advances to Step 10 and halts (Steps 9, 10).

This program will halt as soon as motion to the right reaches a square containing the symbol 1. For once that happens the program will move on to Step 3 and halt. That being the case, suppose we begin with a tape on which there are no ones to the right of the initially scanned square. (For example, the entire tape could be blank or there could be some ones but all to the left of the initially scanned square.) In this case, the first two steps will be carried out over and over again forever, since a 1 will never be encountered. After step 2 is performed, step 1 will be performed again. This makes it clear that a computation from a Turing–Post program *need not actually ever halt.* In the case of this simple three-step program it is very easy to tell from the initial tape configuration whether the computation will

eventually halt or continue forever: to repeat, if there is a 1 to the right of the initially scanned square the computation will eventually halt; whereas if there are only blanks to the right of the initially scanned square the computation will continue forever. We shall see later that the question of predicting whether a particular Turing–Post program will eventually halt contains surprising subtleites.

## Codes for Turing–Post Programs

All of the dramatic consequences of Turing's analysis of the computation process proceed from Turing's realization that it is possible to encode a Turing–Post program by a string of zeros and ones. Since such a string can itself be placed on the tape being used by another (or even the same) Turing–Post program, this leads to the possibility of thinking of Turing–Post programs as being capable of performing computations on other Turing–Post programs.

There are many ways by which Turing–Post programs can be encoded by strings of zeros and ones. We shall describe one such way. We first represent each Turing–Post instruction by an appropriate sequence of zeros and ones according to the following code:

| Code | Instruction |
|---|---|
| 000 | PRINT 0 |
| 001 | PRINT 1 |
| 010 | GO LEFT |
| 011 | GO RIGHT |
| $1010\underbrace{\ldots}_{i}01$ | GO TO STEP i IF 0 IS SCANNED |
| $1101\underbrace{\ldots}_{i}10$ | GO TO STEP i IF 1 IS SCANNED |
| 100 | STOP |

This table gives the representation of each Turing–Post instruction by a string of zeros and ones. For example the code for the instruction

GO TO STEP 3 IF 0 IS SCANNED

is: 1010001. To represent an entire program, we simply write down in order the representation of each individual instruction and then place an additional 1 at the very beginning and 111 at the very end as punctuation marks.

For example here is the code for the doubling program shown in Figure 1:

1000010110110001011110111110001011110101001111

To make this clear, here is the breakdown of this code:

| Begin | Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 7 | Step 8 | Step 9 | Step 10 | End |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 000 | 010 | 110110 | 001 | 011 | 110111110 | 001 | 011 | 11010 | 100 | 111 |

It is important to notice that the code of a Turing–Post program can be deciphered in a unique, direct, and straightforward way, yielding the program of which it is the code. First remove the initial 1 and the final 111 which are just punctuation marks. Then, proceeding from left to right, mark off the first group of 3 digits. If this group of 3 digits is 000, 001, 010, 011, or 100 the corresponding instructoin is: PRINT 0, PRINT 1, GO LEFT, GO RIGHT, or STOP, respectively. Otherwise the group of 3 digits is 101 or 110, and the first instruction is a "GO TO." The code will then have one of the forms:

$$10\underbrace{100\ldots01}_{i} \qquad 11\underbrace{011\ldots10}_{i}$$

corresponding to

GO TO STEP i IF 0 IS SCANNED

and

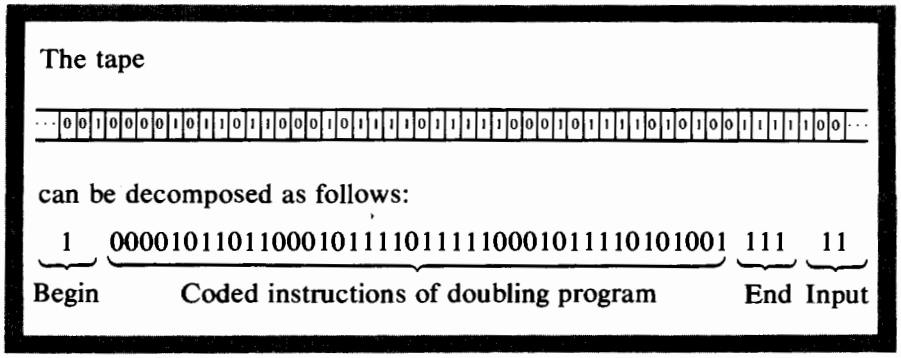
GO TO STEP i IF 1 IS SCANNED

respectively. Having obtained the first instruction, cross out its code and continue the process, still proceeding from left to right. Readers who wish to test their understanding of this process may try to decode the string:

10100011010011000001010101011 1

## The Universal Program

We are now ready to see how Turing's analysis of the computation process together with the method for coding Turing–Post programs we have just introduced leads to a conclusion that at first sight seems quite astonishing. Namely, there exists a single (appropriately constructed) Turing–Post program which can compute anything whatever that is computable. Such a program $U$ (for "universal") can be induced to simulate the behavior of any given Turing–Post program $P$ by simply placing *code* $(P)$, the string of zeros and ones which represents $P$, on a tape and permitting $U$ to operate on it. More precisely, the non-blank portion of the tape is to consist of *code* $(P)$ followed by an input string $v$ on which $P$ can work. (For clarity, we employ capital letters to stand for particular Turing–Post programs and lowercase




letters to stand for strings of zeros and ones.) For example, the string shown in the box above signifies that $U$ should simulate the behavior of the doubling program when 11 is the input. Thus, at the end of the computation by $U$, the tape should look just like the final tape in Figure 2.

Now, a universal Turing–Post program $U$ is supposed to perform in this way not only for our doubling program, but for *every* Turing–Post program. Let us be precise: $U$ is to begin its computation presented with a tape whose nonblank portion consists of *code* ($P$) for some Turing–Post program $P$ (initially scanning the first symbol, necessarily 1, of this code) followed by a string $v$. $U$ is then supposed to compute exactly the same result as the program $P$ would get when starting with the string $v$ as the nonblank part of the tape (scanning the initial symbol of $v$). Such a program $U$ can then be used to simulate any desired Turing–Post program $P$ by simply placing the string *code* ($P$) on the tape.

What reason do we have for believing that there is such a program $U$? To help convince ourselves, let us begin by thinking how a human calculator could do what $U$ is supposed to do. Faced with the tape contents on which $U$ is supposed to work, such a person could begin by scanning this string of zeros and ones, from left to right, searching for the first place that 3 consecutive ones appear. This triple 111 marks the end of *code* ($P$) and the beginning of the input string. Our human calculator can then write *code* ($P$) on one sheet of paper and the input string on another. As already explained, he can decode the string *code* ($P$) and obtain the actual Turing–Post program $P$. Finally, he can "play machine," carrying out the instruction of $P$, applied to the given input string in a robotlike fashion. If and when the computation comes to a halt, our calculator can report the final tape contents as output. This shows that a human calculator can do what we would like $U$ to do. But now, invoking Turing's analysis of the computation process, we are led to believe that there must be a Turing–Post program which can carry out the process we have just described, a universal Turing–Post program.

The evidence we have given for the existence of such a program is rather unsatisfactory because it depends on Turing's analysis of the computation

process. It certainly is not a mathematical proof. But in fact, if one is willing to do some tedious but not very difficult work, one can circumvent the need to refer to Turing's analysis at all and can, in fact, write out in detail an explicit universal Turing–Post program. This was done in fact by Turing himself (in a slightly different, but entirely equivalent context) in his fundamental 1936 paper. And subsequently, it has been redone many times. The success of the construction of the universal program is in itself evidence for the correctness of Turing's analysis. It is not appropriate here to carry out the construction of a universal program in detail; we hope, merely, that the reader is convinced that such a program exists. (Experienced computer programmers will have no difficulty in writing their own universal program if they wish to do so.)

We have conceived of Turing–Post programs as consisting of lists of written instructions. But clearly, given any particular Turing–Post program $P$, it would be possible to build a machine that would actually carry out the instructions of $P$ in sequence. In particular, this can be done for our universal program $U$. The machine we get in this way would be an example of an all-purpose or universal computing machine. The code for a particular program $P$ placed on its tape could then be thought of as a "program" for doing the computation which $P$ does. Thus, Turing's analysis leads us, in a very straightforward manner, to the concept of an all-purpose computer which can be programmed to carry out any computation whatever.

## The Halting Problem

We are now in a position to demonstrate a truly astonishing result: we are able to state a simple problem, the so-called *halting problem*, for which we can prove that no computational solution exists.

The halting problem for a particular Turing–Post program is the problem of distinguishing between initial tape configurations which lead to the program's eventually halting and initial tape configurations which lead the program to compute forever. We saw above that certain input strings may cause a particular program to run forever, due to an infinite loop caused by the "GO TO" instruction. It would surely be desirable to have a method for determining in advance which input data leads the program to halt and which does not. This is the halting problem: given a particular Turing–Post program, can we computationally test a given tape configuration to see whether or not the program will eventually halt when begun with that tape configuration.

The answer is no. *There is no computaton procedure for testing a given tape expression to determine whether or not the universal program U will eventually halt when begun with that tape configuration.* The fact that there is no such procedure for the universal program shows of course that there can't be such procedures for Turing–Post programs in general, since the uni-

versal program is itself a Turing–Post program. Before we see how this unsolvability theorem can be proved, it is worthwhile to reflect on how exciting and remarkable it is that it should be possible to prove such a result. Here is a problem which is easy to state and easy to understand which we *know* cannot be solved. Note that we are not saying simply that we don't know how to solve the problem, or that the solution is difficult. We are saying: *there is no solution.*

Readers may be reminded of the fact that the classical problems of angle trisection and circle squaring also turned out to have no solution. This is a good analogy, but with a very significant difference: the impossibility proofs for angle trisection and circle squaring are for constructions using specific instruments (straightedge and compass); using more powerful instruments, there is no difficulty with either of these geometric construction problems. Matters are quite different with the halting problem; here what we will show is that there is no solution using any methods available to human beings.

The proof of the unsolvability of the halting problem is remarkably simple. It uses the method known as indirect proof or *reductio ad absurdum.* That is, we suppose that what is stated in italics above is false, that in fact, we possess a computing procedure which, given an initial tape configuration will enable us to determine whether or not the universal program will eventually halt when started in that configuration. Then we show that this supposition is impossible; this is done in the box on p. 256.

## Other Unsolvable Problems

In the 1920's the great German mathematician David Hilbert pointed to a certain problem as the fundamental problem of the newly developing field of mathematical logic. This problem, which we may call the decision problem for elementary logic, can be explained as follows: a finite list of statements called *premises* is given together with an additional statement called the *conclusion.* The logical structure of the statements is to be explicitly exhibited in terms of "not," "and," "or," "implies," "for all," and "there exists." Hilbert wanted a computing procedure for testing whether or not the conclusion can be deduced using the rules of logic from the premises. Hilbert regarded this problem as especially important because he expected that its solution would lead to a purely mechanical technique for settling the truth or falsity of the most diverse mathematical statements. (Such statements could be taken as the conclusion, and an appropriate list of axioms as the premises to which the supposed computing procedure could be applied.) Thus the very existence of an unsolvable mathematical problem (in particular, the halting problem) immediately suggested that Hilbert's decision problem for elementary logic was itself unsolvable. This conclusion turned out to be correct, as was carefully shown by Turing and, quite independently, by the American logician Alonzo Church. Turing represented the

## Unsolvability of Halting Problem

Suppose we possess a computing procedure which solves the halting problem for the universal program $U$. Then we can imagine more complicated procedures of which this supposed procedure is a part. Specifically, we consider the following procedure which begins with a string $v$ of zeros and ones:

1. Try to decode $v$ as the code for a Post–Turing program, i.e., try to find $P$ with $code\ (P) = v$. If there is no such $P$, go back to the beginning of Step 1; otherwise go on to Step 2.
2. Make a copy of $v$ and place it to the right of $v$ getting a longer string which we can write as $vv$ (or equivalently as $code\ (P)v$ since $code\ (P) = v$).
3. Use our (pretended) halting problem procedure to find out whether or not the universal program $U$ will eventually halt if it begins with this string $vv$ as the nonblank portion of the tape, scanning the leftmost symbol. If $U$ will eventually halt, go back to the beginning of Step 3; otherwise stop.

This proposed procedure would eventually stop if, first, $v = code(P)$ for some Turing–Post program $P$ (so we will leave Step 1 and go on to Step 2), and, second, if also $U$ will never halt if it begins to scan the leftmost symbol of $vv$. Since $U$ beginning with $code\ (P)v$ simulates the behavior of $P$ beginning with $v$, we conclude that our supposed procedure applied to the string $v$ will eventually stop if and only if $v = code(P)$ where $P$ is a computing procedure that will never stop beginning with $v$ on its tape.

By Turing's analysis, there should be a Turing–Post program $P_0$ which carries out this very procedure. That is, $P_0$ will eventually halt beginning with the input $v$ if and only if $U$ will never halt beginning with the input $vv$. Now let $v_0 = code(P_0)$. Does $U$ eventually halt beginning with the input $v_0v_0$? By what we have just said, *$P_0$ eventually halts beginning with the input $v_0$ if and only if $U$ will never halt beginning with the input $v_0v_0$*. But, as we will show, this contradicts our explanation of how $U$ works as a universal program. Since $v_0 = code(P_0)$, $U$ will act, given the input $v_0v_0$, to simulate the behavior of $P_0$ when begun on input $v_0$. So *$U$ will eventually halt beginning with the input $v_0v_0$ if and only if $P_0$ will eventually halt beginning with the input $v_0$*. But this contradicts the previous italicized statement. The only way out of this contradiction is to conclude that what we were pretending is untenable. In other words, the halting problem for $U$ is not solvable.

theory of Turing–Post programs in logical terms and showed that a solution to the decision problem for elementary logic would lead to a solution of the halting problem. (This connection between logic and programs was rediscovered many years later and now forms the basis for certain investigations into the problem of proving the correctness of computer programs.)

The unsolvability of the decision problem for elementary logic was important, not only because of the particular importance of this problem, but also because (unlike the halting problem) it was an unsolvable problem that people had actually tried to solve. A decade went by before another such example turned up. Early in the century the Norwegian Axel Thue had emphasized the importance of what are now called "word problems." In 1947, Emil Post showed how the unsolvability of the halting problem leads to the existence of an unsolvable word problem. Post's proof is discussed in the box on p. 258. Here we merely explain what a word problem is.

In formulating a word problem one begins with a (finite) collection, called an *alphabet*, of symbols, called *letters*. Any string of letters is called a *word* on the alphabet. A word problem is specified by simply writing down a (finite) list of equations between words. Figure 3 exhibits a word problem specified by a list of 3 equations on the alphabet $a$, $b$, $c$. From the given equations many other equations may be derived by making substitutions in any word of equivalent expressions found in the list of equations. In the example of Figure 3, we derive the equation $bac = abcc$ by replacing the part $ba$ by $abc$ as permitted by the first given equation.

We have explained how to *specify* the data for a word problem, but we have not yet stated what the problem is. It is simply the problem of determining for two arbitrary given words on the given alphabet, whether one can be transformed into the other by a sequence of substitutions that are legitimate using the given equations. We show in the box on p. 258 that we can specify a particular word problem that is unsolvable. In other words, no computational process exists for determining whether or not two words can be transformed into one another using the given equations. Work on unsolvable word problems has turned out to be extremely important, leading to unsolvability results in different parts of mathematics (for example, in group theory and in topology).

Another important problem that eventually turned out to be unsolvable first appeared as the tenth in a famous list of problems given by David Hilbert in 1900. This problem involves so-called "Diophantine" equations. An equation is called Diophantine when we are only interested in solutions in integers (i.e., whole numbers). It is easy to see that the equation

$$4x - 2y = 3$$

has no solutions in integers (because the left side would have to be even while the right side is odd). On the other hand the equation

$$4x - y = 3$$

## An Unsolvable Word Problem

One way to find a word problem that is unsolvable is to invent one whose solution would lead to a solution for the halting problem, which we know to be unsolvable. Specifically, we will show how to use a Turing–Post program $P$ (which we assume consists of $n$ instructions) to construct a word problem in such a way that a solution to the word problem we construct could be used to solve the halting problem for $P$. Therefore, if we begin with a problem $P$ whose halting problem is unsolvable, we will obtain an unsolvable word problem.

We will use an alphabet consisting of the $n + 4$ symbols:

$$1 \; 0 \; h \; q_1 \; q_2 \; \ldots \; q_n \; q_{n+1}.$$

The fact that the $i$th step of $P$ is about to be carried out and that there is some given tape configuration is coded by a certain word (sometimes called a Post word) in this alphabet. This Post word is constructed by writing down the string of zeros and ones constituting the current nonblank part of the tape, placing an $h$ to its left and right (as punctuation marks) and inserting the symbol $q_i$ (remember that it is the $i$th instruction which is about to be executed) immediately to the left of the symbol being scanned. For example, with a tape configuration

$$\begin{array}{c} 11011 \\ \uparrow \end{array}$$

and instruction number 4 about to be executed, the corresponding Post word would be

$$h110q_411h\,.$$

This correspondence between tape configurations and words makes it possible to translate the steps of a program into equations between words. For example, suppose that the fifth instruction of a certain program is

PRINT 0.

We translate this instruction into the equations

$$q_40 = q_50, \qquad q_41 = q_50\,,$$

which in turn yield the equation between Post words

$$h110q_411h = h110q_501h$$

corresponding to the next step in the computation. Suppose next that the fifth instruction is

GO RIGHT.

It requires 6 equations to fully translate this instruction, of which two

typical ones are

$$q_5 0\ 1 = 0 q_6 1\ , \qquad q_5 1\ h = 1 q_6 0 h\ .$$

In a similar manner each of the instructions of a program can be translated into a list of equations. In particular when the $i$th instruction is STOP, the corresponding equation will be:

$$q_i = q_{n+1}\ .$$

So the presence of the symbol $q_{n+1}$ in a Post word serves as a signal that the computation has halted. Finally, the four equations

$$q_{n+1}0 = q_{n+1}\ , \qquad q_{n+1}1 = q_{n+1}$$
$$0q_{n+1} = q_{n+1}\ , \qquad 1q_{n+1} = q_{n+1}$$

serve to transform any Post word containing $q_{n+1}$ into the word $hq_{n+1}h$. Putting all of the pieces together we see how to obtain a word problem which "translates" any given Turing–Post program.

Now let a Turing–Post program $P$ begin scanning the leftmost symbol of the string $v$; the corresponding Post word is $hq_1vh$. Then if $P$ will eventually halt, the equation

$$hq_1vh = hq_{n+1}h$$

will be derivable from the corresponding equations as we could show by following the computation step by step. If on the other hand $P$ will never halt, it is possible to prove that this same equation will not be derivable. (The idea of the proof is that every time we use one of the equations which translates an instruction, we are either carrying the computation forward, or—in case we substitute from right to left —undoing a step already taken. So, if $P$ never halts, we can never get $hq_1vh$ equal to any word containing $q_{n+1}$.) Finally, if we could solve this word problem we could use the solution to test the equation

$$hq_1vh = hq_{n+1}h$$

and therefore to solve the halting problem for $P$. If, therefore, we start with a Turing–Post program $P$ which we know has an unsolvable halting problem, we will obtain an unsolvable word problem.

has many (even infinitely many) solutions in integers (e.g., $x = 1$, $y = 1$; $x = 2$, $y = 5$). The Pythagorean equation

$$x^2 + y^2 = z^2$$

also has infinitely many integer solutions (of which $x = 3$, $y = 4$, $z = 5$ was already known to the ancient Egyptians). Hilbert's tenth problem was to find a computing procedure for testing a Diophantine equation (in any number of unknowns and of any degree) to determine whether or not it has an integer solution.

Since I have been directly involved with this problem and related matters over the past thirty years, my discussion of Hilbert's tenth problem will necessarily have a rather personal character. I first became interested in the problem while I was an undergraduate at City College of New York on reading my teacher Emil Post's remark in one of his papers that the problem "begs for an unsolvability proof." In my doctoral dissertation at Princeton, I proved the unsolvability of a more difficult (and hence easier to prove unsolvable) related problem. At the International Congress of Mathematicians in 1950, I was delighted to learn that Julia Robinson, a young mathematician from California, had been working on the same problem from a different direction: she had been developing ingenious techniques for expressing various complicated mathematical relationships using Diophantine equations. A decade later Hilary Putnam (a philosopher with training in mathematical logic) and I, working together, saw how we could make further progress by combining Julia Robinson's methods with mine. Julia Robinson improved our results still further, and we three then published a joint paper in which we proved that if there were even one Diophantine equation whose solutions satisfy a special condition (involving the relative size of the numbers constituting such a solution), then Hilbert's tenth problem would be unsolvable.

In subsequent years, much of my effort was devoted to seeking such a Diophantine equation (working alone and also with Hilary Putnam), but with no success. Finally such an equation was found in 1970 by the then 22-year old Russian mathematician Yuri Matiyasevich. Matiyasevich's brilliant proof that his equation satisfied the required condition involved surprisingly elementary mathematics. His work not only showed that Hilbert's tenth problem is unsolvable, but has also led to much new and interesting work.

## Undecidable Statements

The work of Bertrand Russell and Alfred North Whitehead in their three-volume magnum opus *Principia Mathematica*, completed by 1920, made it clear that all *existing* mathematical proofs could be translated into the specific logical system they had provided. It was assumed without question by most mathematicians that this system would suffice to prove or disprove any statement of ordinary mathematics. Therefore mathematicians were shocked by the discovery in 1931 by Kurt Gödel (then a young Viennese mathematician) that there are statements about the whole numbers which can neither be proved nor disproved in the logical system of *Principia Mathematica* (or similar systems); such statements are called *undecidable*. Turing's work (which was in part inspired by Gödel's) made it possible to understand Gödel's discovery from a different, and indeed a more general, perspective.

**Julia B. Robinson**

Julia B. Robinson was born in 1919 in St. Louis, Missouri, but has lived most of her life in California. Her education was at the University of California, Berkeley, where she obtained her doctorate in 1948. She has always been especially fascinated by mathematical problems which involve both mathematical logic and the theory of numbers. Her contributions played a key role in the unsolvability proof for Hilbert's tenth problem. In 1975 she was elected to the National Academy of Sciences, the first woman mathematician to be so honored.

Let us write $N(P, v)$ to mean that the Turing–Post program $P$ will *never* halt when begun with $v$ on its tape (as usual, scanning its leftmost symbol). So, for any particular Turing–Post program $P$ and string $v$, $N(P, v)$ is a perfectly definite statement which is either true (in case $P$ will never halt in the described situation) or false (in case $P$ will eventually halt). When $N(P, v)$ is false, this fact can always be demonstrated by exhibiting the complete sequence of tape configurations produced by $P$ leading to termination. However, when $N(P, v)$ is true no finite sequence of tape configurations will suffice to demonstrate the fact. Of course we may still be able to prove that a particular $N(P, v)$ is true by a logical analysis of $P$'s behavior.

Let us try to be very rigorous about this notion of *proof*. Suppose that cer-

tain strings of symbols (possibly paragraphs of English) have been singled out as proofs of particular statements of the form $N(P, v)$. Suppose furthermore that we possess a computing procedure that can test an alleged proof $\Pi$ that $N(P, v)$ is true and determine whether $\Pi$ is or is not actually such a proof. Whatever our rules of proof may be, this requirement is surely needed for communication purposes. It must be possible in principle to perform such a test in order that $\Pi$ should serve its purpose of eliminating doubts concerning the truth of $N(P, v)$. (In practice, published mathematical proofs are in highly condensed form and do not meet this strict requirement. Disputes are resolved by putting in more detail as needed. But it is essential that *in principle* it is always possible to include sufficient detail so that proofs are susceptible to mechanical verification.)

There are two basic requirements which it is natural to demand of our supposed rules of proof:

> *Soundness:* If there is a proof $\Pi$ that $N(P,v)$ is true, then $P$ will in fact never halt when begun with $v$ on its tape.
> *Completeness:* If $P$ will never halt when begun with $v$ on its tape, then there is a proof $\Pi$ that $N(P,v)$ is true.

Gödel's theorem asserts that no rules of proof can be both sound and complete! In other words, if a given set of rules of proof is sound, then there will be some true statement $N(P, v)$ which has no proof $\Pi$ according to the given rules of proof. (Such a true unprovable statement may be called *undecidable* since it will surely not be disprovable.)

To convince ourselves of the truth of Gödel's theorem, suppose we had found rules of proof which were both sound and complete. Suppose "proofs" according to these rules were particular strings of symbols on some specific finite alphabet. We begin by specifying a particular infinite sequence $\Pi_1, \Pi_2, \Pi_3, \ldots$ which includes all finite strings on this alphabet. Namely, let all strings of a given length be put in "alphabetical" order, and let shorter strings always precede longer ones. The sequence $\Pi_1, \Pi_2, \Pi_3, \ldots$ includes all possible proofs, as well as a lot of other things; in particular, it contains a high percentage of total nonsense—strings of symbols combined in completely meaningless ways. But, hidden among the nonsense, are all possible proofs.

Now we show how we can use our supposed rules of proof to solve the halting problem for some given Turing–Post program $P$. We wish to find out whether or not $P$ will eventually halt when begun on $v$. We have some friend begin to carry out the instructions of $P$ on input $v$ with the understanding that we will be informed at once if the process halts. Meanwhile we occupy ourselves by generating the sequence $\Pi_1, \Pi_2, \Pi_3, \ldots$ of possible proofs. As each $\Pi_i$ is generated we use our computing procedure to determine whether or not $\Pi_i$ is a proof of $N(P, v)$. Now, if $P$ will eventually halt, our friend will discover the fact and will so inform us. And, if $P$ will never halt, since our rules of proof are assumed to be complete, there will be a proof $\Pi_i$ of

$N(P, v)$ which we will discover. Having obtained this $\Pi_i$ we will be sure (because the rules are sound) that $P$ will indeed never halt. Thus, we have described a computing procedure (carried out with a little help from a friend) which would solve the halting problem for $P$. Since, as we well know, $P$ can have an unsolvable halting problem (e.g., $P$ could be the universal program $U$), we have arrived at a contradiction; this completes the proof of Gödel's theorem.

Of course, Gödel's theorem does not tell us that there is any particular pair $P, v$ for which we will never be able to convince ourselves that $N(P, v)$ is true. It is simply that, for any given sound rules of proof, there will be a pair $P, v$ for which $N(P, v)$ is true, but not provable *using the given rules*. There may well be other sound rules which decide this "undecidable" statement. But these other rules will in turn have their own undecidabilities.

## Complexity and Randomness

A computation is generally carried out in order to obtain a desired answer. In our discussion so far, we have pretty much ignored the "answer," contenting ourselves with discussing only the gross distinction between a computation which does at least halt eventually and one which goes on forever. Now we consider the question: how complex need a Turing–Post program be to produce some given output? This straightforward question will lead us to a mathematical theory of randomness and then to a dramatic extension of Gödel's work on undecidability.

We will only consider the case where there are at least 2 ones on the tape when the computation halts. The output is then to be read as consisting of the string of zeros and ones between the leftmost and rightmost ones on the tape, and not counting these extreme ones. Some such convention is necessary because of the infinite string of zeros and ones on the tape. In effect the first and last one serve merely as punctuation marks.

To make matters definite suppose that we wish to obtain as output a string consisting of 1022 ones. When we include the additional ones needed for punctuation, we see that what is required is a computation which on termination leaves a tape consisting of a block of 1024 ones and otherwise blank. One way to do this is simply to write the 1024 ones on the tape initially and do no computing at all. But surely we can do better. We can get a slight improvement by using our faithful doubling program (Figure 1). We need only write 512 ones on the tape and set the doubling program to work. We have already written out the code for the doubling program; it took 39 bits. (A *bit* is simply a zero or a one; the word abbreviates *bi*nary digi*t*.) So we have a description of a string of 1022 ones which uses $39 + 512 = 551$ bits. But surely we can do better. $1024 = 2^{10}$, so we should be able to get 1024 ones by starting with 1 and applying the doubling program 10 times. In Figure 4

```
 1  PRINT 0
 .
 .
 .
 9. GO TO STEP 1 IF 1 IS SCANNED
10. GO RIGHT
11. GO TO STEP 22 IF 0 IS SCANNED
12. GO RIGHT
13. GO TO STEP 12 IF 1 IS SCANNED
14. GO LEFT
15. PRINT 0
16. GO LEFT
17. GO TO STEP 16 IF 1 IS SCANNED
18. GO LEFT
19. GO TO STEP 18 IF 1 IS SCANNED
20. GO RIGHT
21. GO TO STEP 1 IF 1 IS SCANNED
22. STOP
```

**Figure 4.** A Program for Calculating Powers of 2

we give a 22-step program, the first nine steps of which are identical to the first nine steps of the doubling program, which accomplishes this. Beginning with a tape configuration

$$\overset{\uparrow}{1}0\underbrace{11\ldots1}_{n}$$

this program will halt with a block of $2^{n+1}$ ones on the tape.

It is not really important that the reader understand how this program works, but here is a rough account: the program works with two blocks of ones separated by a zero. The effect of Steps 1 through 9 (which is just the doubling program) is to double the number of ones to the left of the 0. Steps 10 through 21 then erase 1 of the ones to the right of the zero and return to Step 1. When all of the ones to the right of the zero have been erased, this will result in a zero being scanned at Step 11 resulting in a transfer to Step 22 and a halt. Thus the number of ones originally to the left of the zero is doubled as many times as there are ones originally to the right of the zero.

The full code for the program of Figure 4 contains 155 bits. To obtain the desired block of 1024 ones we need the input 10111111111. We are thus down to $155 + 11 = 166$ bits, a substantial improvement over 551 bits.

We are now ready for a definition. Let $w$ be any string of bits. Then we say that $w$ has *complexity* $n$ (or equivalently, *information content* $n$) and write $I(w) = n$ if:

1. There is a program $P$ and string $v$ such that the length of $code(P)$ plus the length of $v$ is $n$, and $P$ when begun with $v$ will eventually halt with output $w$ (that is with $1w1$) occupying the nonblank part of the tape, and
2. There is no number smaller than $n$ for which this is the case.

If $w$ is the string of 1022 ones, then we have shown that $I(w) \leq 166$. In general, if $w$ is a string of bits of length $n$, then we can easily show that $I(w) \leq n + 9$. Specifically, let the program $P$ consist of the single instruction: STOP. Since this program does not do anything, if it begins with input $1w1$, it will terminate immediately with $1w1$ still on the tape. Since $Code(P) = 1100111$, it must be the case that $I(w)$ is less than or equal to the length of the string $11001111w1$, that is, less than or equal to $n + 9$. (Naturally, the number 9 is just a technical artifact of our particular formulation and is of no theoretical importance.)

How many strings are there of length $n$ such that, say, $I(w) \leq n - 10$? (We assume $n > 10$; in the interesting cases $n$ is much larger than 10.) Each such $w$ would be associated with a program $P$ and string $v$ such that $Code(P)v$ is a string of bits of length less than or equal to $n - 10$. Since the total number of strings of bits of length $i$ is $2^i$, there are only:

$$2 + 4 + \ldots + 2^{n-10}$$

strings of bits of length $\leq n - 10$. This is the sum of a geometric series easily calculated to be $2^{n-9} - 2$. So we conclude: there are fewer than $2^{n-9}$ strings of bits w of length $n$ such that $I(w) \leq n - 10$.

Since there are $2^n$ strings of bits of length $n$, we see that the ratio of the number of strings of length $n$ with complexity $\leq n - 10$ to the total number of strings of length $n$ is no greater than

$$\frac{2^{n-9}}{2^n} = \frac{1}{2^9} = \frac{1}{512} < \frac{1}{500}.$$

This is less than 0.2%. In other words, more than 99.8% of all strings of length $n$ have complexity $> n - 10$. Now the complexity of the string of 1022 ones is, as we know, less than or equal to 166, thus much less than $1022 - 10 = 1012$. Of course, what makes this string so special is that the digit pattern is so regular that a comparatively short computational description is possible. Most strings are irregular or as we may say, *random*.

Thus we are led to an entirely different application of Turing's analysis of computation: a mathematical theory of random strings. This theory was developed around 1965 by Gregory Chaitin, who was at the time an undergraduate at City College of New York (and independently by the world famous A.N. Kolmogorov, a member of the Academy of Sciences of the U.S.S.R.). Chaitin later showed how his ideas could be used to obtain a dramatic extension of Gödel's incompleteness theorem, and it is with this reasoning of Chaitin's that we will conclude this essay.

Let us suppose that we have rules of proof for proving statements of the form $I(w) > n$ where $w$ is a string of bits and $n$ is a positive integer. As before, we assume that we have a computing procedure for testing an alleged proof $\Pi$ to see whether it really is one. We assume that the rules of proof are sound, so that if $\Pi$ is a proof of the statement $I(w) > n$, then the complexity of the string $w$ really is greater than $n$. Furthermore, let us make the very

reasonable assumption that we have another computing procedure which, given a proof $\Pi$ of a statement $I(w) > n$, will furnish us with the specific $w$ and $n$ for which $I(w) > n$ has been proved.

We now describe a new computing procedure we designate as $\Delta$. We begin generating the sequence $\Pi_1, \Pi_2, \Pi_3, \ldots$ of possible proofs as above. For each $\Pi_i$ we perform our test to determine whether or not $\Pi_i$ is a proof of a statement of the form $I(w) > n$. If the answer is affirmative we use our second procedure to find the specific $w$ and $n$. Finally we check to see whether $n > k_0$ where $k_0$ is some fixed large number. If so, we report $w$ as our answer; otherwise we go on to the next $\Pi_i$. By Turing's analysis this entire procedure $\Delta$ can be replaced by a Turing–Post program, where the fixed number $k_0$ is to be chosen at least as large as the length of this program. (The fact that $k_0$ can be chosen as large as this is not quite obvious; the basic reason is that far fewer than $k_0$ bits suffice to describe the number $k_0$.)

Now, a little thought will convince us that this Turing–Post program can never halt: if it did halt we would have a string $w$ for which we had a proof $\Pi_i$ that $I(w) > n$ where $n > k_0$. On the other hand this very program has length less than or equal to $k_0$ (and hence less than $n$) and has computed $w$, so that $I(w) < n$, in contradiction to the soundness of our proof rules. Conclusion: our rules of proof can yield a proof of no statement of the form $I(w) > n$ for which $n > k_0$. This is Chaitin's form of Gödel's theorem: given a sound set of rules of proof for statements of the form $I(w) > n$, there is a number $k_0$ such that no such statement is provable using the given rules for any $n > k_0$.

To fully understand the devastating import of this result it is important to realize that there exist rules of proof (presumably sound) for proving statements of the form $I(w) > n$ which include all methods of proof available in ordinary mathematics. (An example is the system obtained by using the ordinary rules of elementary logic applied to a powerful system of axioms, of which the most popular is the so-called Zermelo–Fraenkel axioms for set theory.) We are forced to conclude that there is some definite number $k_0$, such that it is in principle impossible, by ordinary mathematical methods, to prove that any string of bits has complexity greater than $k_0$. This is a remarkable limitation on the power of mathematics as we know it.

Although we have discussed a considerable variety of topics, we have touched on only a tiny part of the vast amount of work which Turing's analysis of the computation process has made possible. It has become possible to distinguish not only between solvable and unsolvable problems, but to study an entire spectrum of "degrees of unsolvability." The very notion of computation has been generalized to various infinite contexts. In the theory of formal languages, developed as part of computer science, various limitations on Turing–Post programs turn out to correspond in a natural way to different kinds of "grammars" for these languages. There has been much work on what happens to the number of steps and amount of tape needed when the programs are allowed to operate on several tapes simultaneously instead of on just one. "Nondeterministic" programs in which a given step may be

followed by several alternative steps have been studied, and a great deal of work has been done attempting to show that such programs are intrinsically capable of performing much faster than ordinary Turing–Post programs. These problems have turned out to be unexpectedly difficult, and much remains to be done.

## Suggestions for Further Reading


General

Chaitin, Gregory. Randomness and mathematical proof. *Scientific American* **232** (May 1975) 47–52.

Davis, Martin and Hersh, Reuben. Hilbert's 10th problem. *Scientific American* **229** (November 1973) 84–91.

Knuth, Donald E. Algorithms. *Scientific American* **236** (April 1977) 63–80, 148.

Knuth, Donald E. Mathematics and computer science: coping with finiteness. *Science* **194** (December 17, 1976) 1235–1242.

Turing, Sara. *Alan M. Turing*. W. Heffer, Cambridge, 1959.

Wang, Hao. Games, logic and computers. *Scientific American* **213** (November 1965) 98–106.

Technical

Davis, Martin. *Computability and Unsolvability*. McGraw-Hill, Manchester, 1958.

Davis, Martin. Hilbert's tenth problem is unsolvable. *American Mathematical Monthly* **80** (March 1973) 233–269.

Davis, Martin. *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*. Raven Pr, New York, 1965.

Davis, Martin. Unsolvable problems. In *Handbook of Mathematical Logic*, by Jon Barwise (Ed.). North-Holland, Leyden, 1977.

Minsky, Marvin. *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, 1967.

Rabin, Michael O. Complexity of computations. *Comm. Assoc. Comp. Mach.* **20** (1977) 625–633.

Trakhtenbrot, B.A. *Algorithms and Automatic Computing Machines*. D.C. Heath, Lexington, 1963.