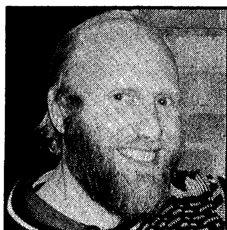
How Mathematicians Know What Computers Can't Do

Leon Harkleroad



Leon Harkleroad writes that his alleged specialty is computability (recursion theory). But he can also be found working on the history of mathematics (focusing on the life and works of Rózsa Péter) or, under the pen name Houston Euler, perpetrating mathematical humor. He is currently a visiting associate professor at Cornell University, where he recently developed a course on mathematics and music. Other activities include playing piano, playing first base, and folk dancing.

Explaining the Title

Before going any further, I should make a couple of points about the title of this paper. First of all, when I say "can't," I really mean "can't." I'm not going to engage in some sort of vague or speculative computer-bashing based on human chauvinism. Just because a computer has not yet composed symphonies of the caliber of Beethoven's doesn't mean that it can't. (After all, how many *people* have ever been able to reach such heights?) In saying that a computer can't perform a certain task, I will mean that the task is *provably impossible*.

The second point concerns how mathematicians come to be connected with the question of what computers can and can't do. We all know that mathematicians are often involved with computers, but being a mathematician is different from being a computer scientist. Why should mathematicians set limits for computers (or computer scientists)? Why is this article in a mathematics publication rather than *Hackers' Digest*? Actually, the history goes the other way around. I'll be using the words "computer" and "program," but much of what I'll be describing was done by mathematicians before electronic computers even existed. Several decades ago, mathematicians became interested in studying the concept of *algorithm*. Algorithms—cut-and-dried procedures for performing mathematical tasks—form a large part of the mathematical experience for all of us. Just think of the grade-school routine for adding numbers, the use of the quadratic formula to solve equations, the calculus rules for differentiating functions. I dare say that to many nonmathematicians, math consists primarily, if not totally, of a collection of algorithms.

Now you usually don't have to worry too much about precisely what an algorithm is, if you have one in hand—you tend to recognize a mechanical procedure when you see one, as with the examples above. But in the early twentieth century, people interested in the foundations of mathematics and mathematical logic started worrying about tasks for which there might not be an algorithm. As suggested earlier, saying that something can't be done is a strong claim. If you're going to *prove* that some task can never have an algorithm, you have to make mathematically precise what you mean by an algorithm in the first place. This was done in the 1930s and, in fact, many of the ideas involved were used in developing the first electronic computers. (Pioneers in the study of algorithms include Alan Turing,

Kurt Gödel, Rózsa Péter, Stephen Kleene, and Alonzo Church.) But in this paper I'll tend to be anachronistic (and sometimes mathematically informal) and describe things in terms of computers and programs.

Down to Brass Tacks

Even being informal, though, I need to be a little bit specific about what I mean by a computer. In general, I'll give the computer the benefit of the doubt and not place many restrictions on it nor assume it bears much resemblance to machines as we know them, from PCs to supercomputers. Essentially, all that is needed is to specify that the computer is (1) deterministic and (2) discrete and finite.

Deterministic. The basic picture is that we give the computer a program to run and then feed the computer some input. The computer, according to the instructions of the program and the input provided, then may give us some output. The requirement of determinism simply asks that we obtain the same results if we run the same program with the same input on different occasions. In other words, loosely speaking, for a given program, the output is a function of the input. Indeed, we may think of a program P as specifying a function f_P as follows: The domain of f_P is the set of inputs for which P yields an output, and if P on input i gives output o, then $f_P(i) = o$. There is still the question of what constitutes inputs, outputs, and programs, which leads to the other requirement for our computer.

Discrete and finite. Discreteness says, in effect, that we are dealing with a digital computer with digitized input and output. More specifically, we require that all communication with the computer be expressed in "words" composed from a finite alphabet of symbols. This could take the form of actual words typed on a keyboard, clicks that register the (discretized) positions of a mouse, a black/white pattern for pixels on a display screen, or some totally novel scheme. The point is that any input, output, or program can be represented by a set of choices, these choices being taken from some given finite set of alternatives. In addition, each "word" must be of finite length. A computer does us humans little good if we need to type an infinite number of letters or read an infinite display screen. (If we had the powers to do such things, we probably wouldn't need computers in the first place!)

An Unprogrammable Function

With these assumptions in place, we are now in a position to investigate our abstract computer's capabilities. The first thing to notice is that even though I did not specify the exact form of the inputs and outputs, we don't lose anything by taking them to be positive integers. Consider, for example, the case when the alphabet consists of the symbols !, @, and *, so that a typical statement to the computer might be @**!*@. We can systematically list the words for this alphabet, starting with the one-character words, then the two-character words, and so on: !, @, *, !!, !@, !*, @!, @@, @*, *!, *@, **, !!!, ... By numbering these in order, we can identify ! with the code number 1, @ with 2, !* with 6, etc. By referring to the code numbers rather than the words, we may thus consider the inputs and outputs as positive integers. This example uses a three-symbol alphabet, but of course a similar argument works regardless of the (finite) size of our alphabet.

Likewise, we can number the programs. Not every word formed from the alphabet is necessarily a program, but we can assign the code number 1 to the first (in the list of all possible words) program, the code number 2 to the second program, etc. For convenience, let P_n be the program with code number n, and let f_n denote f_{P_n}, the function specified by P_n as in the discussion of determinism. By the previous paragraph, we may take the domain and range of f_n to be subsets of Z^+, the set of positive integers.

Now consider the function g with domain Z^+ defined by

$$g(n) = \begin{cases} 2 & \text{if } f_n(n) = 1 \\ 1 & \text{otherwise.} \end{cases}$$

Regardless of the number M, g cannot be the function f_M, because g and f_M behave differently when applied to M. Namely, either $g(M) = 2$, in which case $f_M(M) = 1 \neq 2$, or $g(M) = 1$, in which case $f_M(M) = 1$ does not hold. Thus no program for our computer specifies g.

What Compilers Can't Do

Readers familiar with Cantor's proof of the uncountability of the set of real numbers will recognize the above as basically the same diagonal argument. In fact, in cardinality terms, what is going on is that the set of programs $\{P_1, P_2, \ldots\}$ is countable while the set of functions from Z^+ to Z^+ is uncountable, so there are more functions than can be specified by programs. The explicit argument in the previous paragraph has some interesting consequences, though, so let's examine it more closely.

Although we know that we have no program for g, at first glance the definition of g seems to suggest otherwise. Why can't we formalize the following as a legitimate program?

> Given input n, determine the program P_n.
> Then feed P_n the input n.
> If P_n returns 1, then give 2 as output for g,
> else give 1 as output.

One part of this pseudo-program that might look suspicious is the first step, in which P_n is found by searching the list of words and discarding the non-programs until the nth program is reached. It is conceivable that you may not be able to program the computer to tell whether or not a word represents a program. But in fact that poses no problem. That is the kind of decision a *compiler* makes in actual computers. And when abstract computers are considered more formally than we are doing here, it is straightforward, if sometimes messy, to go through the details of writing a procedure to test words for program-hood.

Well, if the first step doesn't keep g from being computable, what does? It turns out that there is a subtle problem with the if-then-else in the pseudo-program. In case P_n returns some output, there is no trouble with telling whether or not that output is 1. But consider the case when P_n does not return any output: The catch is that you won't necessarily know that's the case! There you are, sitting in front of the computer waiting for output. The output hasn't come yet, but it might come in the next minute...or the next day...or the next year.... And as long as you're

left in limbo, you won't know whether the output for g should be 1 or 2—nor will the computer. *If* the computer had some way of knowing whether or not P_n was going to give an output, then it would be able to assign the value for g. (Namely, if P_n on input n is known not to yield an output, the computer can announce that $g(n)$ is 1. And if P_n is known to yield output, the computer can just execute P_n, find that output, and announce $g(n)$ accordingly.) Since g can't be programmed, that means that neither can you program the decision of whether or not P_n gives an output.

So the bottom line is that you cannot automate the decision of whether or not inputs to programs leave you in limbo. In practical terms, this means that you cannot ask your compiler to warn you every time you have written a program that hangs up in an infinite loop. As mentioned before, a compiler *can* check for proper syntax, making sure that your program is a typographically legitimate string of symbols. But the compiler won't be able to test the semantics—the meaning—of programs sufficiently to guarantee you won't get caught in an infinite loop.

Solutions of Equations

By now we have seen a couple of tasks that a computer can't do: (1) compute a certain function and (2) test programs for infinite loops. Now, (1) deals with a function that may look somewhat cooked-up. On the other hand, (2) deals with a very reasonable, even important, task, but that task is specifically computer-related. Is there some natural and purely mathematical job we would like to see a computer do, but which it can't?

Let's back up for a minute and look at some things a computer *can* do. Certainly, a reasonable desire is to have the computer solve some equations—or at least tell whether or not the equations have any solutions. I'll stick to integers here, so the objective will be: Given a polynomial equation with integer coefficients, tell whether or not the equation has any integer solutions. (In the jargon of number theory, we are testing Diophantine equations for solutions.)

For example, is there an integer x satisfying

$$3x^8 + 7x^5 - 18x^2 - 427x + 10 = 0?$$

Well, if that equation holds for x, then

$$10 = x(-3x^7 - 7x^4 + 18x + 427),$$

and so 10 is a multiple of x. Thus x is one of the eight numbers $\pm 1, \pm 2, \pm 5, \pm 10$. To see whether or not the equation has a solution, just plug in each of those eight numbers for x. If any of the eight works, you have a solution, while if none of them work, there is no solution.

Generalizing, we have a nice algorithm for testing the polynomial equation $p(x) = 0$ for solutions: *If c, the constant term of p, equals 0, then $x = 0$ is a solution. If $c \neq 0$, find all of its divisors. Then plug each divisor of c into p. The equation $p(x) = 0$ has a solution iff one of these plug-ins yields 0 as an answer.* This algorithm can easily be written up as a BASIC or Pascal program—the divisors of c can be found either by brute force, checking all integers from $-|c|$ through $|c|$, or (preferably) by a more efficient search.

Now consider the equation $15x + 33y = 28$. Does it have any integer solutions? No, for a quite simple reason: Regardless of x and y, the left-hand side must be a

multiple of 3, so it cannot equal 28. In general, the linear equation $ax + by = c$ cannot be satisfied unless c is a multiple of the greatest common divisor of a and b [denoted $\gcd(a, b)$]. Conversely, if c is a multiple of $\gcd(a, b)$, then some standard elementary number theory implies that there exist x and y such that $ax + by = c$.

So again, a simple procedure based on divisibility lets us test $ax + by = c$ for solutions: *There is a solution iff* $\gcd(a, b)$ *is a divisor of* c. Of course, that still leaves the question of how to crank out the computation of $\gcd(a, b)$. But that can be done easily. You could by brute force check numbers from 2 through $|a|$ to find which of them divide both a and b. Or, more efficiently, you could use the Euclidean algorithm, a 2500-year-old "program" devised for this purpose.

A Homework Problem

Some basic number theory, then, lets us test a polynomial for zeros in case the polynomial is either a function of one variable or a linear function of two variables. At this stage you might expect me to pose the following exercise:

> Generalize the above and find an algorithm that, when given a polynomial equation with integer coefficients, determines whether or not the equation has an integer solution.

However, I will not assign that exercise for two good reasons. The first reason is that somebody already beat me to it. In 1900 the Second International Congress of Mathematicians met in Paris. David Hilbert gave an address there in which he posed several problems as challenges to the mathematical community, some of which have remained unsolved to this day. The "exercise" above was number 10 of the 23 problems that appeared in the published version of the address.

Hilbert did not ask for an algorithm to find a solution in case one exists, because that presents no problem. Brute force suffices again—just systematically try out the possible combinations of values for the variables until you find one that works. What brute force will not do is to inform you when the equation has no solutions. In this latter case, as brute force testing progresses, you will suspect more and more strongly that there is no solution, but without ever knowing for sure. Hilbert asked for some other algorithm that would definitively settle each case.

But the limbo-like state I just described, coming on the heels of our previous discussion, should keep you from being too surprised at the second reason I did not assign Hilbert's Tenth Problem: The "exercise" is impossible to do! No algorithm can exist for testing Diophantine equations for solutions. Notice, however, that Hilbert did not ask his audience to determine whether such an algorithm exists. He took for granted the existence of such a procedure; to him, the only question was to discover it. Remember, Hilbert's speech came some thirty years before the rigorous study of the concept of algorithm. (Ironically, some of the impetus for that study arose from another challenge of his, known as Hilbert's Program, but that's another story.)

Even after people started suspecting that Hilbert's Tenth Problem might be resolved by showing that the desired algorithm was nonexistent, the resolution took a while to come. Significant work was done in the 1950s and 1960s by three American mathematicians: Martin Davis, Hilary Putnam, and Julia Robinson. Then several more years elapsed before the breakthrough that settled the question. In 1970 the 22-year-old Russian mathematician Yuri Matiyasevich announced his solution of Hilbert's Tenth.

Impossibility by Comparison

How does one go about proving that a task like testing Diophantine equations for solutions cannot be algorithmized? The basic strategy is similar to the use of the comparison test for series, which tells you that if $0 \le a_n \le b_n$ for all n and Σa_n diverges, then Σb_n diverges. Of course, you use this when you are given some messy series Σb_n whose convergence status is unknown to you. You find some other series Σa_n as above that is less messy—clean enough that you can easily show it to be divergent. Then, since Σa_n blows up and the b_n's are at least as big as the a_n's, the sum Σb_n blows up too.

The solution of Hilbert's Tenth works the same way. Just as you can "reduce" the divergence of Σb_n to that of Σa_n, you can reduce the non-algorithmizability of the Diophantine question to that of a less messy problem. In essence, Matiyasevich showed that doing what Hilbert wanted is at least as hard as checking for infinite loops. Since we already know that this easier loop-checking task is impossible, so is Hilbert's task.

Matiyasevich accomplished the comparison by, in effect, encoding some program runs as equations so that runs with (resp., without) infinite loops are represented by equations without (resp., with) solutions. More precisely, the encoding is accomplished by a certain polynomial $h(z, \mathbf{x})$. (Here, for convenience, I'm lumping all the variables other than z into a vector.) This polynomial is "universal" in the sense that for every fixed n, program P_n on input n yields an output iff the equation $h(n, \mathbf{x}) = 0$ has a solution $\mathbf{x}$. Thus the amazing thing about h is that this single polynomial reflects—at least in some respects—the behavior of *all* programs, even programs that compute functions much more complex than polynomials.

The earlier work of Davis, Putnam, and Robinson had shown that the key to Hilbert's Tenth was exponentiation: If 2^x or some other function that grows exponentially fast could somehow have its behavior suitably reflected by a polynomial, then there would exist a universal polynomial like h that tells us about all programs. Matiyasevich's key accomplishment was to get a polynomial handle on the Fibonacci numbers. Since the nth Fibonacci number is approximately $[(1+\sqrt{5})/2]^n$, an exponential function of n, this guaranteed the existence of a universal polynomial and thereby settled Hilbert's Tenth. For more details, see Martin Davis's highly readable "Hilbert's Tenth Problem Is Unsolvable" [*American Mathematical Monthly* 80:3 (March 1973) 233–269].

Many other mathematical tasks can likewise be proved to be unalgorithmizable by encoding programs as appropriate mathematical objects and using the comparison strategy. Fortunately for mathematicians, computer scientists, and everybody else, there are still many things we *can* let the computer do for us. Now about those symphonies....

Acknowledgments. Many thanks to Cynthia Pitts Harkleroad, Harry Rosenzweig, and the referees for several helpful suggestions.