CHAPTER 4

HOW UNIVERSAL ARE TURING MACHINES?

What are the limits to what a computer can do? Must all computers be composed of Boolean logic and registers, or might there be other kinds, even more powerful? These questions take us to the most philosophically interesting topics in this book: Turing machines, computability, chaotic systems, Goedel's incompleteness theorem, and quantum computing—topics at the center of most discussions about what computers can and cannot do.

Because computers can do some things that seem very much like human thinking, people often worry that they threaten our unique position as rational beings, and there are some who seek reassurance in mathematical proofs of the limits of computers. There have been analogous controversies in human history. It was once considered important that the Earth be at the center of the universe, and our imagined position at the center was emblematic of our worth. The discovery that we occupied no central position—that our planet was just one of a number of planets in orbit around the Sun—was deeply disturbing to many people at the time, and the philosophical implications of astronomy became a topic of heated debate. A similar controversy arose over evolutionary theory, which also appeared as a threat to humankind's

uniqueness. At the root of these earlier philosophical crises was a misplaced judgment of the source of human worth. I am convinced that most of the current philosophical discussions about the limits of computers are based on a similar misjudgment.

TURING MACHINES

The central idea in the theory of computation is that of a *universal computer*—that is, a computer powerful enough to simulate any other computing device. The general-purpose computer described in the preceding chapters is an example of a universal computer; in fact most computers we encounter in everyday life are universal computers. With the right software and enough time and memory, any universal computer can simulate any other type of computer, or (as far as we know) any other device at all that processes information.

One consequence of this principle of universality is that the only important difference in power between two computers is their speed and the size of their memory. Computers may differ in the kinds of input and output devices connected to them, but these so-called peripherals are not essential characteristics of a computer, any more than its size or its cost or the color of its case. In terms of what they are able to do, all computers (and all other types of universal computing devices) are fundamentally identical.

The idea of a universal computer was recognized and described in 1937 by the British mathematician Alan Turing. Turing, like so many other computing pioneers, was interested in the problem of making a machine that could think, and he invented a scheme for a general-purpose computing machine. Turing referred to his imaginary construct as a "universal machine," since at that time the word "computer" still meant "a person who performs computations."

To picture a Turing machine, imagine a mathematician performing calculations on a scroll of paper. Imagine further that the scroll is infinitely long, so that we don't need to worry about running out of places to write things down. The mathematician will be able to solve any solvable computational problem no matter how many operations are involved, although it may take him an inordinate amount of time. Turing showed that any calculation that can be performed by a smart mathematician can also be performed by a stupid but meticulous clerk who follows a simple set of rules for reading and writing the information on the scroll. In fact, he showed that the human clerk can be replaced by a finite-state machine. The finite-state machine looks at only one symbol on the scroll at a time, so the scroll is best thought of as a narrow paper tape, with a single symbol on each line.

Today, we call the combination of a finite-state machine with an infinitely long tape a *Turing machine*. The tape of a Turing machine is analogous to, and serves much the same function as, the memory of a modern computer. All that the finite-state machine does is read or write a symbol on the tape and move back and forth according to a fixed and simple set of rules. Turing showed that any computable problem could be solved by writing symbols on the tape of a Turing machine—symbols that would specify not just the problem but also the method of solving it. The Turing machine computes the answer by moving back and forth across the tape, reading and writing symbols, until the solution is written on the tape.

I find Turing's particular construction difficult to think about. To me, the conventional computer, which has a memory instead of a tape, is a more comprehensible example of a universal machine. For instance, it is easier for me to see how a conventional computer can be programmed to simulate a Turing machine than vice versa. What is amazing to me is not so much Turing's imaginary construct but his hypothesis that there is only one type of universal computing machine. As far

as we know, no device built in the physical universe can have any more computational power than a Turing machine. To put it more precisely, any computation that can be performed by any physical computing device can be performed by any universal computer, as long as the latter has sufficient time and memory. This is a remarkable statement, suggesting as it does that a universal computer with the proper programming should be able to simulate the function of a human brain.

LEVELS OF POWER

How can Turing's hypothesis be true? Surely some other kind of computer could be more powerful than the ones we have described. For one thing, the computers we have discussed so far have been binary, that is, they represent everything in terms of 1 and 0. Wouldn't a computer be more powerful if it could represent things in terms of a three-state logic, like **Yes**, **No**, and **Maybe**? No, it would not. We know that a three-state computer would be able to do no more than a two-state computer, because you can simulate the one using the other. With a two-state computer, you can duplicate any operation that can be performed on a three-state computer, by encoding each of the three states as a pair of bits—00 for **Yes**, say, and 11 for **No**, and 01 for **Maybe**. For every possible function in three-state logic, there is a corresponding function in two-state logic which operates on this representation. This is not to say that three-state computers might not have some practical advantage over two-state computers: for instance, they might use fewer wires and therefore might be smaller, or cheaper to produce. But we can say for certain that they would not be able to do anything new. They would just be one more version of a universal machine.

A similar argument holds for four-state computers, or five-state computers, or computers with any finite number of

states. But what about computers that compute with analog signals—that is, signals with an infinite number of possible values? For example, imagine a computer that uses a continuous range of voltages to indicate numbers. Instead of just two or three or five possible messages, each signal could carry an infinite number of possible messages, corresponding to the continuous range of voltages. For instance, an analog computer might represent a number between 0 and 1 by a voltage between zero and one volt. The fraction could be represented to any level of precision, no matter the number of decimal places, by using the exact corresponding voltage.

Computers that represent quantities by such analog signals do exist, and in fact the earliest computers worked this way. They are called *analog computers*, to distinguish them from the digital computers we have been discussing, which have a discrete number of possible messages in each signal. One might suppose that analog computers would be more powerful, since they can represent a continuum of values, whereas digital computers can represent data only as discrete numbers. However, this apparent advantage disappears if we take a closer look. A true continuum is unrealizable in the physical world.

The problem with analog computers is that their signals can achieve only a limited degree of accuracy. Any type of analog signal—electrical, mechanical, chemical—will contain a certain amount of noise; that is, at a certain level of resolution, the signal will be essentially random. Any analog signal is bound to be affected by numerous irrelevant and unknown sources of noise: for example, an electrical signal can be disturbed by the random motion of molecules inside a wire, or by the magnetic field created when a light is turned on in the next room. In a very good electrical circuit, this noise can be made very small—say, a millionth the size of the signal itself—but it will always exist. While there are an infinite number of possible signal levels, only a finite number of levels represent meaningful distinctions—that is, rep-

resent information. If one part in a million in a signal is noise, then there are only about a million meaningful distinctions in the signal; therefore, information in the signal can be represented by a digital signal that uses twenty bits (2^{20} = 1,048,578). Doubling the number of meaningful distinctions in an analog computer would require making everything twice as accurate, whereas in a digital computer you could double the number of meaningful distinctions by adding a single bit. The very best analog computers have fewer than thirty bits of accuracy. Since digital computers often represent numbers using thirty-two or sixty-four bits, they can in practice generate a much larger number of meaningful distinctions than analog computers can.

Some people might argue that while the noise of an analog computer may not be meaningful, it is not necessarily useless. One can certainly imagine computations that are helped by the presence of noise. Later, for example, we will describe computations requiring random numbers. But a digital computer, too, can generate random noise if randomness is called for in a computation.

RANDOM NUMBERS

How can a digital computer generate randomness? Can a deterministic system like a computer produce a truly random sequence of numbers? In a formal sense, the answer is No, since everything a digital computer does is determined by its design and its inputs. But the same could be said of a roulette wheel—after all, the ball's final landing place is determined by the physics of the ball (its mass, its velocity) and the spinning wheel. If we knew the exact design of the apparatus and the exact "inputs" governing the spin of the wheel and the throw of the ball, we could predict the number on which the ball would land. The outcome appears random because it exhibits no obvious pattern and is difficult, in practice, to predict.

Like the roulette wheel, a computer can produce a sequence of numbers that is random in the same sense. In fact, using a mathematical model, the computer could simulate the physics of the roulette wheel and throw a simulated ball at a slightly different angle each time in order to produce each number in the sequence. Even if the angles at which the computer throws the simulated ball follow a consistent pattern, the simulated dynamics of the wheel would transform these tiny differences into what amounts to an unpredictable sequence of numbers. Such a sequence of numbers is called a *pseudorandom* sequence, because it only appears random to an observer who does not know how it was computed. The sequence produced by a pseudorandom number generator can pass all normal statistical tests of randomness.

A roulette wheel is an example of what physicists call a *chaotic system*—a system in which a small change in the initial conditions (the throw, the mass of the ball, the diameter of the wheel, and so forth) can produce a large change in the state to which the system evolves (the resulting number). This notion of a chaotic system helps explain how a deterministic set of interactions can produce unpredictable results. In a computer, there are simpler ways to produce a pseudorandom sequence than simulating a roulette wheel, but they are all conceptually similar to this model.

Digital computers are predictable and unpredictable in exactly the same senses as the rest of the physical world. They follow deterministic laws, but these laws have complicated consequences that are extremely difficult to predict. It is often impractical to guess what computers are going to do before they do it. As is true of physical systems, it does not take much to make a computation complex. In computers, chaotic systems—systems whose outcomes depend sensitively on the initial conditions—are the norm.

COMPUTABILITY

While a universal computer can compute anything that can be computed by any other computing device, there are some things that are just impossible to compute. Of course, it is not possible to compute answers to vaguely defined questions, like "What is the meaning of life?" or questions for which we lack data, like "What is the winning number in tomorrow's lottery?" But there are also flawlessly defined computational problems that are impossible to solve. Such problems are called *noncomputable*.

I should warn you that noncomputable problems hardly ever come up in practice. In fact, it is difficult to find examples of a well-defined noncomputable problem that anybody wants to compute. A rare example of a well-defined, useful, but noncomputable problem is the *halting problem*. Imagine that I want to write a computer program that will examine another computer program and determine whether or not that program will eventually stop. If the program being examined has no loops or recursive subroutine calls, it is bound to finish eventually, but if it does have such constructs the program may well go on forever. It turns out that there is no algorithm for examining a program and determining whether or not it is fatally infected with an endless loop. Moreover, it's not that no one has yet discovered such an algorithm; rather, no such algorithm is possible. The halting problem is noncomputable.

To understand why, imagine for a moment that I do have such a program, called Test-for-Halt, and that it takes the program to be tested as an input. (Treating a program as data may seem strange, but it's perfectly possible, because a program, just like anything else, can be represented as bits.) I could insert the Test-for-Halt program as a subroutine in another program, called Paradox, which will perform Test-for-Halt on Paradox itself. Imagine that I have written the Paradox program so that whatever Test-for-Halt determines, Paradox will do the opposite. If Test-for-Halt determines that Paradox is eventually going to halt, then Paradox is programmed to go into an infinite loop. If Test-for-Halt determines that Paradox is going to go on forever, then Paradox is programmed to halt. Since Paradox contradicts Test-for-Halt, Test-for-Halt doesn't work on Paradox; therefore, it doesn't work on all programs. And therefore a program that computes the halting function cannot exist.

The halting problem, which was dreamed up by Alan Turing, is chiefly important as an example of a noncomputable problem, and most noncomputable problems that do come up in practice are similar to or equivalent to it. But a computer's inability to solve the halting problem is not a weakness of the computer, because the halting problem is inherently unsolvable. There is no machine that can be constructed that can solve the halting problem. And as far as we know, there is nothing that can perform any other computation that cannot be performed by a universal machine. The class of problems that are computable by a digital computer apparently includes every problem that is computable by any kind of device. (This last statement is sometimes called the Church thesis, after one of Turing's contemporaries, Alonzo Church. Mathematicians had been thinking about computation and logic for centuries but—in one of the more dazzling examples of synchrony in science—Turing, Church, and another British mathematician named Emil Post all independently invented the idea of universal computation at roughly the same time. They had very different ways of describing it, but they all published their results in 1937, setting the stage for the computer revolution soon to follow.)

Another noncomputable function, closely related to the halting problem, is the problem of deciding whether any given mathematical statement is true or false. There is no algorithm that can solve this problem, either—a conclusion of Goedel's incompleteness theorem, which was proved by

Kurt Goedel in 1931, just before Turing described the halting problem. Goedel's theorem came as a shock to many mathematicians, who until then had generally assumed that *any* mathematical statement could be proved true or false. Goedel's theorem states that within any self-consistent mathematical system powerful enough to express arithmetic, there exist statements that can neither be proved true nor false. Mathematicians saw their job as proving or disproving statements, and Goedel's theorem proved that their "job" was in certain instances impossible.

Some mathematicians and philosophers have ascribed almost mystical properties to Goedel's incompleteness theorem. A few believe that the theorem proves that human intuition somehow surpasses the power of a computer—that human beings may be able to "intuit" truths that are impossible for machines to prove or disprove. This is an emotionally appealing argument, and it is sometimes seized upon by philosophers who don't like being compared to computers. But the argument is fallacious. Whether or not people can successfully make intuitive leaps that cannot be made by computers, Goedel's incompleteness theorem provides no reason to believe that there are mathematical statements that can be proved by a mathematician but can't be proved by a computer. As far as we know, any theorem that can be proved by a human being can also be proved by a computer. Humans cannot compute noncomputable problems any more than computers can.

Although one is hard pressed to come up with specific examples of noncomputable problems, one can easily prove that most of the possible mathematical functions are noncomputable. This is because any program can be specified in a finite number of bits, whereas specifying a function usually requires an infinite number of bits, so there are a lot more functions than programs. Consider the kind of mathematical function that converts one number into another—the cosine, say, or the logarithm. Mathematicians can define all kinds of bizarre functions of this type: for example the function that converts every decimal number into the sum of its digits. As far as I know, this function is a useless one, but a mathematician would regard it as a legitimate function simply because it converts every number into exactly one other number. It can be proved mathematically that there are infinitely more functions than programs. Therefore, for most functions there is no corresponding program that can compute them. The actual counting involves all kinds of difficulties (including counting infinite things and distinguishing between various degrees of infinity!), but the conclusion is correct: statistically speaking, most mathematical functions are noncomputable. Fortunately, almost all these noncomputable functions are useless, and virtually all the functions we might want to compute are computable.

QUANTUM COMPUTING

As noted earlier, the pseudorandom number sequences produced by computers look random, but there is an underlying algorithm that generates them. If you know how a sequence is generated, it is necessarily predictable and not random. If ever we needed an inherently unpredictable random-number sequence, we would have to augment our universal machine with a nondeterministic device for generating randomness.

One might imagine such a randomness-generating device as being a kind of electronic roulette wheel, but, as we have seen, such a device is not truly random because of the laws of physics. The only way we know how to achieve genuinely unpredictable effects is to rely on quantum mechanics. Unlike the classical physics of the roulette wheel, in which effects are determined by causes, quantum mechanics produces effects that are purely probabilistic. There is no way of predicting, for example, when a given uranium atom will decay into lead. Therefore one could use a Geiger counter to

generate truly random data sequences—something impossible in principle for a universal computer to do.

The laws of quantum mechanics raise a number of questions about universal computers that no one has yet answered. At first glance, it would seem that quantum mechanics fits nicely with digital computers, since the word "quantum" conveys essentially the same notion as the word "digital." Like digital phenomena, quantum phenomena exist only in discrete states. From the quantum point of view, the (apparently) continuous, analog nature of the physical world—the flow of electricity, for example—is an illusion caused by our seeing things on a large scale rather than an atomic scale. The good news of quantum mechanics is that at the atomic scale everything is discrete, everything is digital. An electric charge contains a certain number of electrons, and there is no such thing as half an electron. The bad news is that the rules governing how objects interact at this scale are counterintuitive.

For instance, our commonsense notions tell us that one thing cannot be in two places at the same time. In the quantum mechanical world this is not exactly true, because in quantum mechanics nothing can be exactly in any place at all. A single subatomic particle exists everywhere at once, and we are just more likely to observe such a particle at one place than at another. For most purposes, we can think of a particle as being where we observe it to be, but to explain all observed effects we have to acknowledge that the particle is in more than one place. Almost everyone, including many physicists, find this concept difficult to comprehend.

Might we take advantage of quantum effects to build a more powerful type of computer? As of now, this question remains unanswered, but there are suggestions that such a thing is possible. Atoms seem able to compute certain problems easily, such as how they stick together—problems that are very difficult to compute on a conventional computer. For instance, when two hydrogen atoms bind to an oxygen

atom to form a water molecule, these atoms somehow "compute" that the angle between the two bonds should be 107 degrees. It is possible to approximately calculate this angle from quantum mechanical principles using a digital computer, but it takes a long time, and the more accurate the calculation the longer it takes. Yet every molecule in a glass of water is able to perform this calculation almost instantly. How can a single molecule be so much faster than a digital computer?

The reason it takes the computer so long to calculate this quantum mechanical problem is that the computer would have to take into account an infinite number of possible configurations of the water molecule to produce an exact answer. The calculation must allow for the fact that the atoms comprising the molecule can be in all configurations at once. This is why the computer can only approximate the answer in a finite amount of time. One way of explaining how the water molecule can make the same calculation is to imagine it trying out every possible configuration simultaneously—in other words, using parallel processing. Could we harness this simultaneous computing capability of quantum mechanical objects to produce a more powerful computer? Nobody knows for sure.

Recently there have been some intriguing hints that we may be able to build a quantum computer that takes advantage of a phenomenon known as *entanglement*. In a quantum mechanical system, when two particles interact, their fates can become linked in a way utterly unlike anything we see in the classical physical world: when we measure some characteristic of one of them, it affects what we measure in the other, even if the particles are physically separated. Einstein called this effect, which involves no time delay, "spooky action at a distance," and he was famously unhappy with the notion that the world could work that way.

A quantum computer would take advantage of entanglement: a one-bit quantum mechanical memory register would

store not just a 1 or a 0; it would store a superposition of many 1's and many 0's. This is analagous to an atom being in many places at once: a bit that it is in many states (1 or 0) at once. This is different from being in an intermediate state between a 1 and a 0, because each of the superposed 1's and 0's can be entangled with other bits within the quantum computer. When two such quantum bits are combined in a quantum logic block, each of their superposed states can interact in different ways, producing an even richer set of entanglements. The amount of computation that can be accomplished by a single quantum logic block is very large, perhaps even infinite.

The theory behind quantum computing is well established, but there are still problems in putting it to use. For one thing, how can we use all this computation to compute anything useful? The physicist Peter Shor recently discovered a way to use these quantum effects—at least, in principle—to do certain important and difficult calculations like factoring large numbers, and his work has renewed interest in quantum computers. But many difficulties are still there. One problem is that the bits in a quantum computer must remain entangled in order for the computation to work, but the smallest of disturbances—a passing cosmic ray, say, or possibly even the inherent noisiness of the vacuum itself—can destroy the entanglement. (Yes, in quantum mechanics even a vacuum does strange things.) This loss of entanglement, called *decoherence*, could turn out to be the Achilles heel of quantum mechanical computers. Moreover, Shor's methods seem to work only on a specific class of computations which can take advantage of a fast operation called a generalized Fourier transform. The problems that fit into this category may well turn out to be easy to compute on a classical Turing machine; if so, Shor's quantum ideas would be equivalent to some program on a conventional computer.

If it does become possible for quantum computers to search an infinite number of possibilities at once, then they would be qualitatively, fundamentally more powerful than conventional computing machines. Most scientists would be surprised if quantum mechanics succeeds in providing a kind of computer more powerful than a Turing machine, but science makes progress through a series of surprises. If you're hoping to be surprised by a new sort of computer, quantum mechanics is a good area to keep an eye on.

This leads us back to the philosophical issues touched on at the beginning of the chapter—that is, the relationship between the computer and the human brain. It is certainly conceivable, as at least one well-known physicist has speculated (to hoots from most of his colleagues), that the human brain takes advantage of quantum mechanical effects. Yet there is no evidence whatsoever that this is the case. Certainly, the physics of a neuron depends on quantum mechanics, just as the physics of a transistor does, but there is no evidence that neural processing takes place at the quantum mechanical level as opposed to the classical level; that is, there is no evidence that quantum mechanics is necessary to explain human thought. As far as we know, all the relevant computational properties of a neuron can be simulated on a conventional computer. If this is indeed the case, then it is also possible to simulate a network of tens of billions of such neurons, which means, in turn, that the brain can be simulated on a universal machine. Even if it turns out that the brain takes advantage of quantum computation, we will probably learn how to build devices that take advantage of the same effects—in which case it will still be possible to simulate the human brain with a machine.

The theoretical limitations of computers provide no useful dividing line between human beings and machines. As far as we know, the brain is a kind of computer, and thought is just a complex computation. Perhaps this conclusion sounds harsh to you, but in my view it takes nothing away from the wonder or value of human thought. The statement that thought is a complex computation is like the statement sometimes made by biologists that life is a complex chemical reaction: both statements are true, and yet they still may be

seen as incomplete. They identify the correct components, but they ignore the mystery. To me, life and thought are both made all the more wonderful by the realization that they emerge from simple, understandable parts. I do not feel diminished by my kinship to Turing's machine.

CHAPTER 5

ALGORITHMS AND HEURISTICS

When I was an undergraduate at MIT, one of my roommates had several dozen pairs of socks, each pair with a slightly different color or design. He frequently postponed doing his laundry until he was completely out of clean socks, so whenever he washed them he had the not inconsiderable task of matching them all up again in pairs. Here is the way he would do it: First, he would pull a random sock out of the pile of clean laundry, then he would extract another sock at random and compare it to the first to see if it matched. If it didn't, he would throw the second sock back and pull out another one. He would keep doing this until he found a match, and then he would go through the same sequence all over again with a new sock. Since he had to look though a lot of laundry, the process went very slowly—especially at the beginning, because there were a lot more socks to be examined before a match turned up.

He was studying for a degree in mathematics, and was apparently taking some kind of course in computers. One day when he had hauled his laundry basket back to our rooms, he announced, "I have decided to use a better algorithm for matching my socks." What he meant was that he was now going to use a procedure of a fundamentally different nature. He pulled out the first sock and set it on the table,