Chapter Three

The Computable

Not everything that can be defined can be computed.

> *Computer science is no more about computers*
> *than astronomy is about telescopes.*
> Edsger Dijkstra

3.1 The Turing Paradigm

In retrospect, humans have been remarkably uncurious for too long about information processing. Animals take complex inputs when seeing, smelling, touching, or hearing, and then produce behaviors that depend in complicated ways on these inputs. Human behavior can be even more perplexing and hard to understand. Phenomena like these we can observe every day. It would seem natural to wonder: Just how do living organisms process information and decide what to do? Curiously, until recent decades little intellectual effort has been put into understanding this question. To be fair to our predecessors, however, it is clear that, until recently, anyone attempting to study information processing would have been stymied by a fundamental impediment—no way was known of even formulating the question.

This only changed in the 1930s, when Alan Turing published a mathematical paper, "On Computable Numbers, with an Application to the *Entscheidungsproblem*," that inaugurated one of the most significant scientific revolutions in history.[1] The *Entscheidungsproblem* (or decision problem, in English) refers to a question raised by mathematician David Hilbert in 1928 concerned with deciding the validity of statements in mathematical logic. However, in his paper Turing went far beyond answering this one

question. He formulated a notion that has changed how we view the world. Through its technological impact, this notion has changed how we live. His discovery was that computation, or the execution of step-by-step procedures for processing information, could be defined and studied systematically. Since that time we have been on a recognizable track toward understanding what such procedures can and cannot do. That is to say, we have come to understand computation. We have also been exploiting that understanding to produce technology, but technology is not my concern here.

The technical concept of computability makes an important distinction: It is one thing to specify, even unambiguously, what result you expect from a computation for every input of data. It is quite another to specify a step-by-step computation that gets you there. The difference is not immediately apparent. Nevertheless, Alan Turing proved that there exist problems for which there is no ambiguity as to what result is desired, but for which there is no set of step-by-step instructions that will get you the right result for every input. This was a stunning finding. Research over the past several decades has developed a rich science for making even finer distinctions, particularly with regard to efficiency. It turns out that there are also problems that are not computable efficiently enough to be practical, even if in principle they can be computed. That fact poses its own problems: We want computations not only to exist in principle, but also to deliver answers within a reasonable period of time. To obtain the result we should not have to wait for months, or years, or until after our galaxy has ceased to exist.

These laws of computation apply to all algorithms. Because ecorithms are algorithms, though of a special kind, they too must follow the same basic laws as computation in general. This new science of the ultimate limitations on the possibility and the efficiency with which computations for learning and evolution can proceed offers a fundamental new approach to understanding these phenomena of learning and evolution, because, regardless of how they are implemented—in silicon, DNA, neurons, or something else entirely—there are some ultimate logical laws that limit what these mechanisms can do.

Turing's paper contained several ingredients that are now seen as fundamental to the study of computation. First, he described a model, now called the Turing machine, that captures the phenomenon he was attempting to describe, namely that of mechanistic step-by-step procedures. Sec-

ond, he proved a strong *possibility* result for what can be achieved on his model. In particular, he showed how to design a universal Turing machine that is capable of executing every possible mechanical procedure. This universality property is what enables computer technology to be so pervasively useful, and would be utterly astonishing were it not so commonplace now by virtue of its effectiveness. Third, Turing also proved a strong *impossibility* result, that not all well-defined mathematical problems can be solved mechanically.

Turing's impossibility result is as striking as universality is on the positive side. It is concerned with the problem of predicting, for an arbitrary computer program and an input for it, whether that program started on that input will ever halt its computation after a finite number of steps, as opposed to getting stuck in a loop in perpetuity. This so-called Halting Problem is well defined. Once we specify a language for expressing the programs there is no ambiguity at all about what would and what would not constitute a solution to it. It would be good to be able to tell ahead of time whether a computer program will get stuck in a perpetual loop. Yet, as Turing showed, it cannot be solved in all cases by any Turing machine.[2] We will never be able to solve this problem routinely.

Many of the foremost thinkers of the early part of the twentieth century had wondered, somewhat informally, whether mechanical procedures existed for resolving all mathematically well-posed questions. Some, such as the philosopher Bertrand Russell and the mathematician David Hilbert, were optimistic. Turing's discovery that one could define precisely what such an assertion meant, and then *prove* that such a statement was false, had revolutionary implications. The shock of this is still taking its time to permeate the community of the educated.

Important as the three particulars of Turing's paper are—namely Turing machines, universality, and noncomputability—they become even more significant when viewed as an instance of a general class of what I call a Turing triad: an unambiguous model of computation that captures some real-world phenomenon (mechanical calculation in Turing's specific case), and both possibility and impossibility results about that model. Learning, evolution, and intelligence are all manifestations of computational processes. As realized in nature, they may be subtle and operate near the limits of computational feasibility. We may need a correspondingly sophisticated understanding of computation before we can unravel their secrets. My

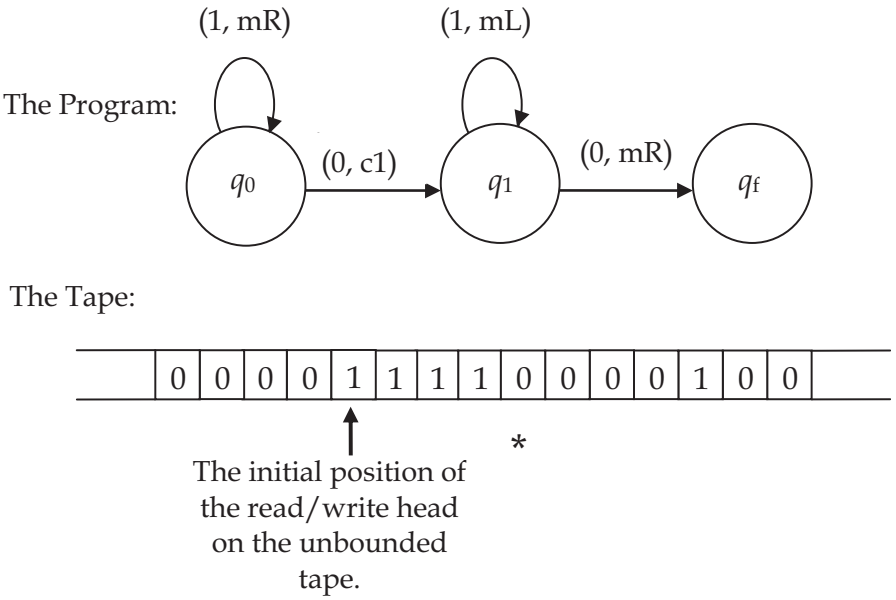

Figure 3.1 An example of a simple Turing machine. The diagram at the top describes the program that controls the machine. The input is the sequence of 0s and 1s on successive squares of the tape. The machine has three states q_0, q_1, and q_f. It starts in state q_0 and with the read/write head on the square pointed to by the thick arrow. If the machine is in state q_0 and the symbol under the head is 1 (as it is initially in this example), then the path indicated by the arrow out of the q_0 node with label starting with a 1 will be taken, in this case the arrow labeled (1, mR) with endpoint q_0. Executing this (1, mR) will result in the contents of the square being unchanged and the head moving one square to the right. The endpoint of the arrow indicates that the next state will be q_0 again. An arrow labeled (1, mL) would mean the same except that the head moves to the left. An arrow labeled (1, c0) would mean that the square is changed from 1 to 0 and the head does not move. The labels (0, mR), (0, mL), and (0, c1) have analogous meanings and apply when instead the symbol under the head is 0. The computation halts if and when a final state q_f is reached. The reader may verify by working through this example that, eventually, when the read/write head reaches the 0 at the * sign, the machine will change the 0 there to a 1 and change the state to q_1, then move the head back finally to the starting position, and then halt in state q_f. (Note that we can obtain an example of a machine that never halts on this input by changing the (0, mR) arrow from q_1 to go to q_0 rather than to q_f.)

strategy for shedding light on them will be to seek Turing triads for these phenomena also.

3.2 Robust Computational Models

The reader may have noticed that in the previous section there was an unexplained leap. The assertion that the Halting Problem was not computable by any Turing machine was identified with the claim that it was not computable by any conceivable mechanical procedure. To justify this leap, we will need a notion known as the robustness of models under variation, one of computer science's deepest and most fortunate mysteries.

We have seen that an essential ingredient of the Turing methodology is that of defining a model of computation that captures a real-world phenomenon, in this case that of mechanical processes, including those that no one had (or has yet) envisaged. That last part is crucial: With his machine, Turing aimed to capture all processes a human could exploit while performing a mental task that can be regarded as mechanical as opposed to requiring creativity or inspiration. The audaciousness of the attempt has attracted many who would prove Turing's machine insufficient to the power Turing claimed for it. However, when different individuals have tried to define their own notions of mechanical processes in hopes of creating models of greater power, all the models they have devised—no matter how different they may seem—could be proved to have no greater capabilities than those of Turing machines. For example, having two tapes, or five tapes, or a two-dimensional tape adds no new power. Similarly, allowing the program to make random decisions, or transitions that have the parallelism suggested by quantum mechanics, also adds no new capabilities. Extensive efforts at finding models that have greater power than Turing machines, but still correspond to what one would instinctively regard as mechanical processes, have all failed. Therefore there is now overwhelming historical evidence that Turing's notion of computability is highly robust to variation in definition. This has placed Turing computability among the most securely established theories known to science.

This robustness under variation of the model offers the fundamental key and launching pad for our study here. For learning and evolution, robust models are as indispensable as they are for general computation. Without this robustness the value of any model or theory is questionable. We are not interested in properties of arbitrary formalisms. We want some assurance

that we have captured the characteristics of some real-world phenomenon. Robustness of models is the only known source of such assurance.

The discovery of the notion of computability constituted a new approach to discovering truths about the world. The logician Kurt Gödel generously acknowledged that computability theory "has for the first time succeeded in giving an absolute definition of an interesting epistemological notion, i.e., one not depending on the formalism chosen."[3] What can be computed does not change as one varies the details of the model. In later chapters, I shall try to persuade the reader that, for the same reason, analogous absolute definitions should be sought also for other notions, and in particular learning and evolution.

There is, of course, no reason to believe that for every notion for which there is a word in a dictionary there exists an absolute definition, or a robust computational model that captures its essence. Indeed, computability, learnability, and evolvability may be among the few. For most other notions no such robust computational models are known, and although robust models may be discovered one day for some, for the rest no such models may exist at all. The question of whether notions such as free will or consciousness can be made theoryful by the algorithmic method pursued here hangs, I believe, on whether robust computational models can be found for them.

3.3 The Character of Computational Laws

Turing's contributions amounted to more than a series of specific discoveries; they provided a new way of pursuing science. In this, his importance demands comparison with that of Isaac Newton. Newton's influence on physics is without parallel, not because he described gravity or made any other particular discovery, but because it was through his work that it became accepted that the physical world obeys laws that can be described by mathematical equations, and that solving these equations could yield accurate predictions of what will happen in the future. Newton's theories not only had the immediate generality that they applied very broadly to mechanical systems. They had a higher level supergenerality in that they offered a blueprint for developing theories for fields that had yet to be conceived. Physicists have followed this lodestone of expressing physical laws by mathematical equations ever since. Electromagnetic theory, general relativity, and quantum mechanics are not implied by Newton's mechanics, but they follow the same intellectual pattern: physical laws expressed as mathematical equations. In

this sense, equations offered the wizardry that enabled successive generations of physicists to achieve an understanding of the physical world beyond that of which previous generations could have dreamed. Since the seventeenth century physics has been transformed several times as far as the range of phenomena that it could explain. Even as the particular discoveries of Newton have been superseded, physics is still being pursued with a methodology recognizably similar to that used by Newton.

No one knows why such supergenerality should exist in physics. It is sufficient for most purposes to recognize that it does. The physicist Eugene Wigner suggested that we simply enjoy its benefits: "The miracle of the appropriateness of the language of mathematics for the formulation of the laws of physics is a wonderful gift which we neither understand nor deserve. We should be grateful for it and hope that it will remain valid in future research and that it will extend, for better or for worse, to our pleasure, even though perhaps also to our bafflement, to wide branches of learning."[4]

Robust computational models, I expect, will turn out to provide supergenerality in computer science as mathematical equations have in physics. They will enable the extent and limits of computational phenomena, in all their variety, to be uncovered. Just as, in retrospect, the texture of all the discoveries in physics over the last three centuries can be recognized already in the work of Newton, the texture of much of the new science of the coming centuries will be traceable to Turing.

One can make some further observations regarding the two fields. Physics concentrates on understanding a minimal set of basic processes that are sufficient to explain the dynamics of the physical world, such as how particles move under natural forces. In contrast, computer science entertains much more diverse sets of processes—in fact, any process that can be formulated as step-by-step rules. As long as the trajectory of objects under the laws of physics can be simulated by step-by-step rules, as appears to be the case, computation will embrace all the processes studied in physics. However, computational processes, though more general than those of physics, are not totally arbitrary. They are governed by their own logical laws and limitations. The laws that govern them are our concern in the present chapter.

Prior to Turing, mathematics was dominated by the continuous mathematics used to describe physics, in which (classically, anyway) changes are thought of as taking place in arbitrarily small, infinitesimal increments. The Turing machine, however, is a discrete model. Before his time, discrete

mathematics had been little explored or developed; in fact, a seldom discussed influence of Turing's work is the rise of discrete mathematics subsequent to it. It is striking that for the phenomena that we shall study here, including learning and evolution, discrete models again provide the most immediate robust models and have been most useful in isolating the basic phenomena. Continuous models are ultimately of at least as great interest, but for the initial explorations necessary to identify the most fundamental concepts they are not the most fruitful.

Besides the discrete versus continuous dichotomy, there is a more fundamental difference between physics and computer science. In physics we think of the equation as the immutable fundamental law, expressing such facts as that the gravitational force between two objects is proportional to the square of the inverse distance, and to no other function of the distance. In computation we have much broader latitude in constructing programs than this. We allow arbitrary programs composed of steps from some repertoire of basic steps. The immutable laws of computation are not constraints on how programs can be composed. Rather, like the noncomputability of the Halting Problem, they state what can or cannot be achieved by *any* program of a specified kind.

In computation the laws are statements, subject to mathematical proof or refutation, but their relevance relies on the robustness of the model in question. One may consider the laws of physics to be analogous to the laws of computation. However, as far as not being subject to mathematical verification,

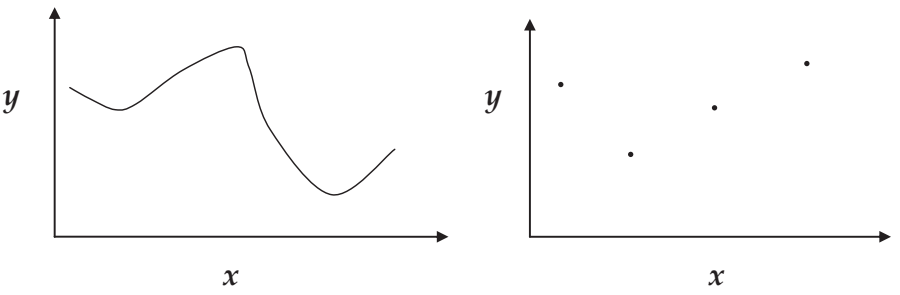

Figure 3.2 In a continuous model there are infinitely many possible states that are related to each other smoothly. The left-hand diagram shows an example where each state is represented by a point on the curve. In a discrete model the states need have no such relation. The right-hand diagram shows a model with four states, as indicated by the dots.

the laws of physics correspond in computation rather to assertions about the robustness of the models. The commonality between the laws of physics and robustness questions in computational models can be also stated positively—in both cases one needs to go to realities beyond mathematical formalisms for supporting evidence or falsification.

3.4 Polynomial Time Computation

Once computers had become more widely available and broader efforts were made to program them, the importance of understanding computational limitations in finer detail than computability theory provides came to the fore. The study of these limitations came to be known as computational complexity.[5] In that field one does not distinguish merely whether an algorithm for a specified task does or does not exist. One also quantifies how many steps any algorithm, if one exists, must take.

Using this idea, one can try to classify both familiar and unfamiliar tasks according to the number of basic operations that are required to perform them. The process of long multiplication for obtaining the product of two numbers is a familiar enough algorithm, taught in elementary schools worldwide. To find the product of two numbers, each of n digits in standard decimal notation, it takes about n^2 basic operations on pairs of single digits, as illustrated in Figure 3.3.

For long multiplication the actual number of operations on individual pairs of digits may be $4n^2$ or $5n^2$ or cn^2 for some fixed number c, depending on what exactly you consider an operation. It will not, however, grow faster than n^2, such as n^3 or n^4; nor will it grow more slowly, such as $n^{1.9}$. We can describe the order of growth using what is known as O notation, while omitting the less important detail of the value of c. We simply say that the long multiplication algorithm is an $O(n^2)$ algorithm.[6]

In general, we distinguish between an algorithm taking polynomial time versus one that takes exponential time. It is polynomial time if it takes $O(n^k)$ basic steps for some constant k, where n is the number of digits or bits needed to write down the input. Of course, it is best if k is a small number such as 1 or 2. An exponential time algorithm takes the form k^n (such as 2^n or 10^n). Exponential time algorithms become impractical even for moderate input sizes. For example, for a task taking 10^n steps, if n is just 30, then 1,000,000,000,000,000,000,000,000,000,000 steps are needed. A computer doing a trillion steps per second would take more than 30 billion years,

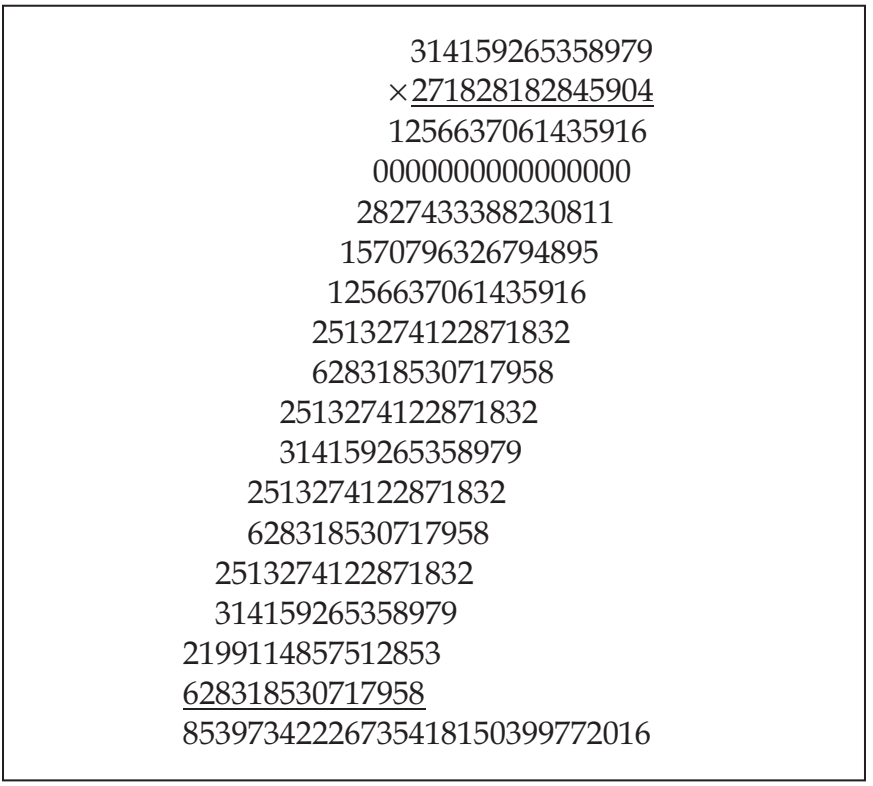

Figure 3.3 When performing long multiplication on two numbers each of n decimal digits, here $n = 15$, we multiply the first n-digit number by each of the n digits of the second number in turn, and then add the results. This can all be done by performing proportional to n^2 basic operations, additions and multiplications on pairs of single digits. However, these n^2 operations look repetitious. This raises the question of whether the same result can be achieved with far fewer operations. It is hard to explain why this very natural question had to wait till the 1960s to be asked and answered.

more than twice the currently estimated age of this universe, to accomplish this. Running many computers in parallel does not change the picture too much. If you had one computer for every particle in this universe, of which there are currently believed to be fewer than 10^{90}, then within this 30 billion years one could do $10^{90} \times 10^{30}$, or 10^{120}, operations. For a task taking 10^n steps, one could then solve instances of size $n = 120$. If we increased the speed of each computer by a factor of 1,000 we would increase the allowed input size only by 3, to get a limit on n of 123.

The point is that numbers such as 123 are very modest as input sizes. An input of 123 digits requires less than two lines on this page to write down. Most applications of computers need much larger inputs. For example, if one is scheduling a fleet of aircraft, then the input size will be in the thousands. If one is performing computations on data read in from the World Wide Web, perhaps analyzing articles written about some topic, then input sizes may be in the millions. In all such cases algorithms taking 10^n steps would be totally impractical. Even having all the resources of our universe at our disposal would be far from enough.

The class of tasks or problems that can be computed in polynomial time is represented by the capital letter P. The task of multiplying two integers is therefore a member of this class P, because the standard algorithm for it, as we have noted, takes $O(n^2)$ steps, which is polynomial. In general, P characterizes what can be computed in practice.[7]

Fundamental to computational complexity is the distinction between the outcome one wants to achieve, say, finding the product of two numbers, and the many possible ways of achieving it. Also fundamental is the notion that there exist easily specified problems that are computable in principle in the sense of Turing but for which all algorithms are impractically inefficient. The idea that we should classify tasks according to their computational difficulty appears to be very natural, and so this idea plays a central role in computer science. Yet there is an implication that goes a little against the grain of traditional science education. In conventional mathematics and science courses the computational tasks presented are invariably limited to those that are easy to compute, such as arithmetic and linear algebra. This tradition has the obvious justification that it presents only methods that are practical. However, a traditional education along these lines does leave the mistaken impression that *every* easily specified problem can be solved efficiently. It ill-prepares the student to face entirely novel challenges and find approaches to them that are computationally feasible. Turing's wartime work in breaking codes was centered on the problem of deducing from an encrypted message the original message, without performing an exhaustive search of the exponentially many possible keys that might have been used for encryption. Similarly, many natural tasks people would like to solve by computer, such as scheduling, involve finding the best solution from potentially exponentially many solutions. For many of these tasks exponential time algorithms are known, but none faster. Our

scientific culture is still in the process of absorbing the significance of this phenomenon.

The impracticality of exponential time computations is self-evident. While the boundary between the practically computable and the infeasible is not sharp, the polynomial time criterion is the most convenient place that has been found to put that boundary. Clearly, polynomial time with high exponent, such as n^{100}, is as infeasible in practice as exponential time, even for modest values of n. However, the polynomial versus exponential distinction has proved very useful, simply because the majority of algorithms that are known for important problems conveniently dichotomize between feasibly low degree polynomials, such as quadratic, $O(n^2)$, and proper exponentials, such as 2^n. Hence, for reasons that are not understood, this polynomial versus exponential criterion is more useful in practice than its bare definition justifies. Experience shows that if someone claims to be able to compute a function routinely for arbitrary inputs of significant size, but claims that the problem is not in P, then there is a good chance that more can be and needs to be said. Perhaps the inputs are not really arbitrary but restricted to a special subclass or a probability distribution for which the problem is indeed solvable in polynomial time. More often than not, on further examination, one can explain such unexpectedly good performance. Indeed, much current research in computer science centers on the question of identifying the circumstances, sometimes one application at a time, in which polynomial time computation can be achieved in some useful sense, even if not in complete generality.

When defining computation, there is a further important distinction. A computation is deterministic if each step is uniquely determined from what has gone before. In the definition of P this is assumed. However, for all practical purposes we can relax this constraint of determinism to permit computations that make random choices as if they were tossing coins. These algorithms may still arrive at the correct answer with high probability, even if not with certainty. So-called randomized algorithms, ones that do arrive at correct answers with high probability for every input, are as effective as deterministic ones in practice, the probabilities involved arising only from the coin tosses the algorithm makes. Such algorithms give a wrong answer only for combinations of coin tosses that occur very rarely, such as getting heads only three times in a thousand tosses. Further, for every input, the probability that an error occurs can be driven down to be exponentially

small by simply repeating the algorithm enough times, the probability of error being independent for each repetition. One can extend the definition of standard deterministic Turing machines to allow them to make decisions according to the toss of a coin in this way. These are called randomized Turing machines. The corresponding polynomial class is called BPP, for *bounded probabilistic polynomial* time. It is possible that P and BPP are mathematically identical, in which case every computation that uses randomization could be simulated in polynomial time by one without it. This question of whether P and BPP are equal is currently unresolved.

A class broader still than BPP is called BQP, for *bounded quantum polynomial* time. This class is inspired by quantum physics, which posits that a physical system can be in multiple states at the same time, in a certain specific sense. It is natural to ask whether such quantum phenomena can be exploited to speed up computation. Oversimplifying a little, quantum phenomena may permit a million computations to be pursued simultaneously in a single quantum computer, while conventional computers would need to

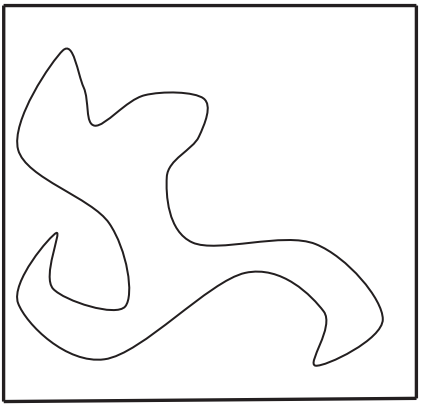

Figure 3.4 An example of a randomized algorithm. One can estimate the area of any shape by drawing it in a square of known area, throwing darts randomly at the square, and counting what fraction fall within the shape. This will work for any shape. All that is required is that the darts have uniform probability of hitting any part of the square, and that the successive throws be independent of each other. The only risk of getting a bad approximation of the area is that of being unlucky and getting throws that are not representative of uniformity. The probability of such an outcome goes down as one throws more and more darts. Randomized algorithms and the class BPP have essentially this guarantee of success.

do these one after the other in a million phases on a single machine, or in parallel on a million machines. Much effort has been expended to understand the power of polynomial time quantum computation. On the one hand, one would like to understand better, in mathematical terms, the power of the restricted parallelism that quantum theory seems to offer. On the other, one would like to know whether such machines can be constructed at all in this physical world, since quantum computers require certain capabilities that have not been shown to be realizable.

One can define the class PhysP to be the maximal class of problems that the physical universe we live in permits to be computed in polynomial time. Identifying the limits of the class PhysP would appear to be one of the great scientific questions of our time. BQP is a natural candidate. If it turns out not to be realizable, then BPP is the most natural alternative known.

While identifying PhysP is fundamentally a question about physics, mathematics may have a role in resolving it. It is possible that one can prove by a mathematical demonstration that P = BPP or BPP = BQP or even P = BPP = BQP. This last eventuality, for example, would show once and for all that polynomial time quantum machines have no more power than polynomial time versions of the standard deterministic machines defined by Turing that was illustrated in Figure 3.1.

One thing we do know is that each of these three classes—P, BPP, and BQP—itself has substantial robustness under variation. Attempts to characterize deterministic, randomized, or quantum computations have yielded just one good candidate computational model for each class. This robustness for the randomized and quantum classes is known only for problems with yes/no answers. This currently leaves two main candidates, BPP and BQP, for the practically computable yes/no problems in this physical world. Having two candidates is only a small embarrassment, further alleviated by the fact that the range of natural problems that have been identified to be in BQP but are not known to be in BPP is somewhat limited.

If we leave aside the constraint of polynomial time, of course, any of the algorithm types—deterministic, randomized, or quantum—is provably as good as any other. The characteristic robustness of Turing machines remains. In the opposite direction, if we instead impose more and more constraints, beyond the polynomial constraint, to reach the learning and evolvability classes that we shall meet later, the robustness criteria become increasingly challenging to satisfy.

3.5 Possible Ultimate Limitations

The Turing methodology that we described earlier when applied to a specific task such as integer multiplication would consist of the following three components. Define an appropriate model that captures the realistic cost of computing the task. Prove possibility results, in this case efficient algorithms for the task that take few steps. Prove some impossibility result that shows, for example, that for the model defined no algorithm exists that takes fewer than so many steps.

There are many problems that we would wish to solve efficiently but do not know how. The most efficient algorithms known for a large class of these problems take an exponential, rather than a polynomial, number of steps. Certainly, there is no necessary reason why the best currently known algorithm should be the best possible algorithm. Let's return to the problem of multiplication. The basic question is this: What is the most efficient method for multiplying two n-digit numbers? This question has inspired a research program that has been pursued for half a century now. In 1960 an initial algorithm that took only $O(n^{1.6})$ steps was discovered by Anatoly Karatsuba, working in the Soviet Union.[8] For large values of n, this already improved substantially on the classical $O(n^2)$ method. It is more than a little surprising that this discovery, that integers could be multiplied much faster than by the standard method taught to children worldwide, came so recently.

After this initial discovery there rapidly followed a sequence of improvements. These culminated in the algorithm published by Arnold Schönhage and Volker Strassen in 1971.[9] This had runtime close to but not quite linear—that is, $O(n)$—but better than $O(n^{1+x})$ for any $x>0$. As a result of these developments we now know that integer multiplication is easier to compute than anyone would have had reason to suspect in earlier centuries, when only $O(n^2)$ methods were known and nothing better suspected.

To our embarrassment, we do not yet know whether multiplication is substantially more difficult than addition, which can be done by the standard method in linear time. Ironically, there has been little (in fact close to no) progress on establishing such an impossibility result. It is clear that any algorithm would need to look at all the $2n$ digits of the two input numbers, and hence that this computation cannot be done in fewer than $2n$ steps. However, the possibility remains that there exist linear time algorithms for multiplication, say $10n$ operations on pairs of one-digit numbers, as there are for addition. Resolving whether such a linear time algorithm exists for

integer multiplication with inputs and outputs represented in the standard decimal or binary notation remains a major challenge for theoretical computer science. Is multiplication an inherently harder task than addition, or does it just appear to be so?

Multiplication is a comparatively simple problem, and clearly already computable in practice by means of the ancient algorithm. The question of whether polynomial time algorithms will be found someday for any of the many problems for which we currently have only exponential time algorithms is addressed in the field of complexity theory. We shall now review some of these results in the remainder of this section. The reader may find this interesting background in computer science, but it is not indispensable for what comes later.

A celebrated class of problems is the so-called NP, or nondeterministic polynomial time, class. These are characterized as the problems for which solutions may or may not be hard to find but for which a candidate solution is easily verified. For example, suppose we want to know whether a target number x, say 923, can be factored as the product of two smaller numbers p and q. Then for any given candidate pair p, q we can easily verify whether or not they are the factors of x simply by multiplying them together and checking whether the answer equals x. (For example, given the candidate numbers 71 and 13, it is easy to determine whether or not 71×13 is equal to the target 923.) But given just the number 923, there is no similarly easy route known to discovering the 71 or the 13. One naïve method for discovering such factors would be to enumerate all numbers less than the target x and test each one for whether it divides x exactly. This would, however, take about 10^n steps for n digit numbers. (The best method currently known for finding the factors of n-digit numbers is exponential in the cube root of n, which is a considerable improvement, but still not polynomial.)

The primality problem is the problem of determining for an arbitrary number x whether it has factors other than itself and 1. It is an NP problem since this verification of a particular candidate solution can be done in polynomial time (i.e., as we have observed, given an n-digit number x and two further numbers p and q, we can verify whether $pq = x$ in $O(n^2)$ steps). It turns out that, in fact, there do exist some very clever algorithms that can determine in polynomial time whether a number is prime. They reveal whether factors *exist*, but, curiously, not what these factors are.[10] Hence this particular NP problem of determining primality is in fact also in P.

Currently, there is no way known for finding the factors in polynomial time, even with randomization (BPP). The apparent exponential gap for classical computation between the difficulty of *testing* whether factors exist and *finding* them if they do is the basis of widely used cryptographic schemes, notably the RSA cryptosystem.[11] In the RSA system you choose two large prime numbers p and q, and multiply them together to get their product x. You then make public only the result x, and keep p and q secret. Anyone in the world who sees x can encipher messages intended for you, but only you, who know p and q, will be able to decrypt any such message. The point is that generating arbitrary primes p and q requires only the generation of some random numbers and testing whether they are prime. The eavesdropper needs to do the apparently much harder task of actually finding the factors of a particular x. (This factoring problem is known to be in BQP, or computable in polynomial time on a quantum Turing machine. This fact lends some intrigue, at least, to the question of whether it is feasible to construct quantum computers.)

The importance of NP is that it captures the very general process of mental search.[12] We call these problems mental search because they can be solved by searching objects one generates internally in one's head or computer. They do not require searching in the outside world, as one would when searching for a phrase in the World Wide Web or for oil in the ground. Given a particular problem, one can characterize a set of potential solutions large enough that any true solution must be in that set of candidates. For the problem of determining whether some n-digit number x can be factored, one may specify the potential solutions as the integers $\{2, 3, \ldots, x-1\}$. Finding the solution is simply a matter of testing each number, one by one, to see whether it divides x. Such exhaustive searches are not feasible for large values of n, for this or any other problem. For any NP problem the crucial question therefore is whether a more efficient process for detecting the existence of solutions than such an exhaustive search is possible.

The primality problem does have such a fast alternative algorithm, but it is by no means typical of NP problems. For a very large class, and one could say for the majority of natural NP problems, no algorithm is known that puts them in P or BPP or even, like the factoring problem, BQP. Remarkably, it has been shown that all the members of a very large class of NP problems are in fact provably equivalent to each other, in the sense that a polynomial time algorithm for one would give a polynomial time algorithm for any

other. This class has the further remarkable property that each member is provably the hardest member of NP. In other words, for these so-called NP-complete problems, no one currently knows a polynomial time algorithm for any of them, but if someone did find such an algorithm for *any* one, then polynomial time algorithms would follow for *all* problems in NP.[13]

An example of such an NP-complete problem is the Traveling Salesman Problem. Here one is given a map containing some cities, the distances between the pairs of cities that have direct roads between them, and a number x. The problem is to determine whether there is a tour that traverses every city exactly once and has total distance no more than x. This problem is in NP because given a candidate tour it is easy enough to verify that each city is traversed exactly once and that the total length of roads used in the tour is less than x. Scheduling problems in all their variety offer a host of other mental search tasks for which we wish we had efficient algorithms, but we now know that most are NP-complete. NP-complete problems can be found in every area of mathematics. For example, for algebraic equations we have the primality question as to whether, given an n digit integer c, the equation $xy = c$ has a solution in integers x and y. Of course some equations are easier to solve than this. Given integers a, b, and c, whether the equation $ax^2 + bx + c = 0$ has an integer solution can be solved by the standard formula for solving quadratic equations. On the other hand, the superficially similar question of whether the equation $ax^2 + by + c = 0$ has any integer solutions x, y, is NP-complete![14] Yes, we are told only about the easy things in high school.

Because intensive efforts to find polynomial time algorithms for NP-complete problems have to date failed, many currently conjecture that $P \neq NP$, or equivalently, that no polynomial time algorithm exists for NP-complete problems. (NP-complete problems are similarly conjectured not to be in BPP or QBP either.) Whether this conjecture is in fact a computational law, like Turing's proven assertion that the Halting Problem is not computable, is potentially resolvable, and I expect that it will be proved or disproved one day. The postulate $P \neq NP$, while it remains unresolved in either direction, might be compared to laws in physics, which likewise have not been mathematically proven. Of course, a physical law cannot be proven by mathematics. Such computation postulates can play analogous roles to physical laws in the sense that we can make good use of them as working hypotheses, at least until someone disproves them. In this instance the working hypoth-

esis is that polynomial time algorithms do *not* exist for NP-complete problems. The eventuality that this is disproved could, of course, be a very happy one if it is accompanied by the discovery of an efficient algorithm for all NP-complete problems, which would have revolutionary consequences if it was efficient enough.

In later chapters as I move on to consider learning, reasoning, and evolution, I shall seek to follow the Turing triad: establishing a robust computational model, proving some strong possibility results, and proving some impossibility results that explain the ultimate limitations. As in complexity theory generally, proving impossibility results is particularly challenging. We may need to be ready to postulate certain algorithmic laws, in analogy with NP-completeness, without being able to prove them. Such postulates can then be treated as working hypotheses, at least until someone disproves them and finds, unexpectedly and pleasantly, that a whole range of computational phenomena, currently believed to be infeasible, is indeed feasible.

A potentially wider class of computations still than NP is #P (pronounced "sharp P"). This is the class of problems that enumerate the number of solutions of NP problems. They give a number as the output. This class also has a class of its hardest members, called the #P-complete problems, analogous to the NP-complete problems. It is clear that for an NP-complete problem counting the number of solutions is at least as hard as detecting whether there are any, since the answer will be a number, and if it is greater than zero, then we will know that there exist solutions. More interestingly, there are many natural problems where testing whether there exists a solution is in P but counting their number is #P-complete. This means that while the existence of solutions can be detected fast, counting the number of solutions is as hard as for NP-complete problems.[15] Examples of such problems abound in the context of reliability—for example, where one wants to determine the probability that a complex network or system will fail from the failure probabilities of the components. Since the probability of something happening is closely related to the number of ways it can happen, these problems can be viewed as counting problems. It turns out that this class #P is at least as powerful as not only NP but also the quantum class BQP.[16] It remains a possibility therefore that a yet undiscovered polynomial time algorithm exists that computes all problems in #P, and hence also all problems in BPP, BQP, and NP.

The importance of these complexity classes derives from the additional fact that they are useful for classifying naturally occurring problems. Many problems that arise are mental search problems or their corresponding counting problems. It just so happens that when we come across a new task that we would like to have solved, if we cannot find a polynomial time algorithm for it, then more often than not, we can prove that it is complete in (i.e., is a hardest member of) its class NP or #P. Logically they could fall in

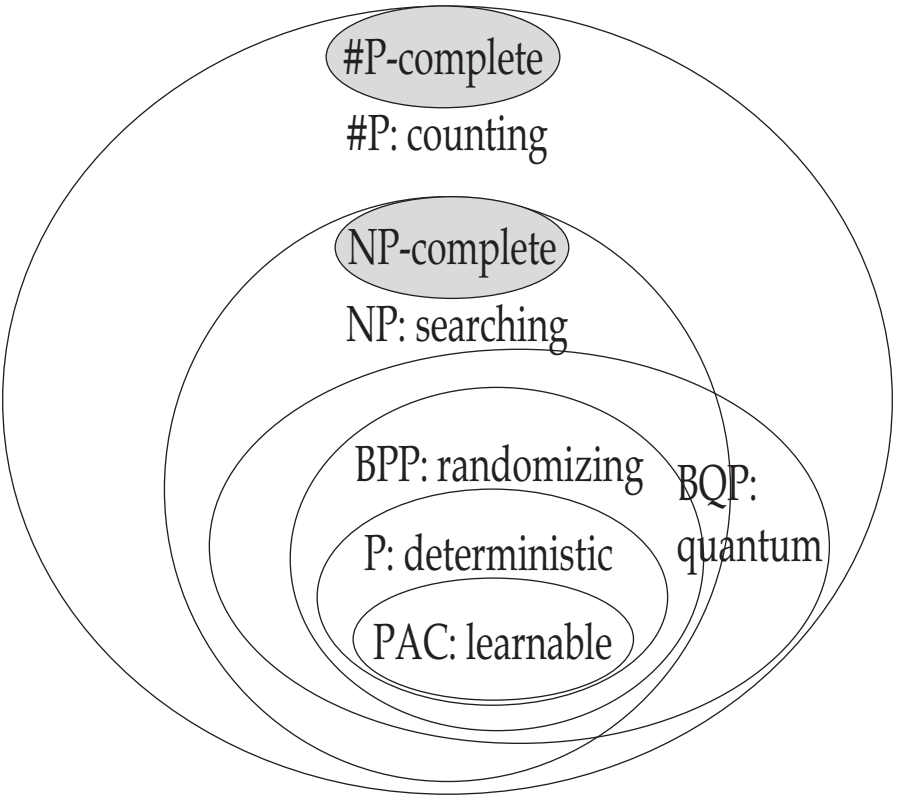

Figure 3.5 An illustration of the relative computational power of some complexity classes, as understood in 2013. Each ellipse represents a class of problems or tasks. Each point in each ellipse represents a problem, such as testing numbers for being prime, or the Traveling Salesman Problem.[17] The first glimpse that natural problems were related in this elegant way was given in a historic paper published by Stephen Cook in 1971 that defined the NP-complete class. The diagram illustrates the previously unsuspected rich structure that is now known to abound among different problems. The PAC class represents the feasibly learnable and is the subject of Chapter 5.

between, but for reasons we do not understand, they rarely do. For this reason this theory gives useful guidance as to the practical solvability of new problems as they arise. Why natural problems should dichotomize in this way is not predicted or explained by any known theory. It is one of those Wignerian mysteries that we neither understand nor deserve, but should be grateful for and simply enjoy.

These basic questions, about the relative extents of these complexity classes, are related intimately to the question of the real extent of PhysP, the class that this universe *physically* permits to be computed efficiently. If the quantum class BQP turns out to equal PhysP, we still would like to know whether NP or #P is within that class, and hence also permitted by physics. Questions about the relative power of these complexity classes can be viewed therefore as questions of physics also.

3.6 Simple Algorithms with Complicated Behavior

Ultimately, as we have seen, there are some limitations on what algorithms can do. Another way of saying this is that our powers to specify what we wish to compute are greater than the expressive power of computation itself. Nevertheless, the language of algorithms, despite these limitations, can be itself very expressive. Turing's result that there exist universal Turing machines that can simulate any computation is a clear statement of the breathtaking power of algorithms, the algorithm in question there being the one that controls the universal machine.

That, of course, we have already seen. A different facet of the richness of algorithms is that even some very simple specific cases can have behaviors that are mystifying to mere mortals. A well-known example of a simple procedure that has so far defied analysis is the following:

1) Start with any positive integer n.
2) Repeat until $n = 1$:
   (a) If n is even, replace n with $n/2$.
   (b) If n is odd, replace n with $3n + 1$.

For example, starting from $n = 44$, we get the following sequence: 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. For a fixed starting point, such as $n = 44$, computing the successive members of the resulting sequence is easy enough. What is not known is whether the sequence generated for *every*

starting point n eventually reaches the value $n = 1$ and terminates. Many starting points have been tried since the mathematician Lothar Collatz posed the problem in 1937. They all resulted in computations that did terminate at $n = 1$. But—somewhat shockingly, given how simple the problem is to describe—no one has been able to offer a proof that this process would terminate for every possible starting point, or that it would not.

Collatz's problem is an example of apparent inherent complexity in simple procedures, even those isolated from any complex environment. In this case the notion of input can be removed altogether by considering a compound procedure that feeds the starting numbers $n = 2$, 3, 4 in succession to the basic procedure, going on to the next starting number when the sequence generated by the previous one has terminated at $n = 1$. Asking whether this compound procedure will ever get to every starting number n, rather than get stuck in perpetuity after a specific n, is equivalent to the original problem. In this light we should not be so shocked by the noncomputability of the Halting Problem, which would need to be able to make some kind of prediction about the ultimate fate not just of one, but of any computation.

3.7 The Perceptron Algorithm

Our journey through the major themes of computational complexity now brings us finally to the vicinity of our destination, the study of ecorithms. Our final point of departure is a simple but important algorithm that, like Collatz's problem, can also have complicated behavior, but these complications can be attributed to the outside environment in which it operates. This example is the perceptron algorithm, proposed by Frank Rosenblatt in the 1950s.[18]

The perceptron algorithm operates in the following context. Assume that there is a set of potential examples, each one specified by some description, and further that there is a criterion for which some of the examples are true examples and the others false. For instance, an example may be an individual flower, and the criterion may be whether that flower belongs to species A or species B.[19] The perceptron algorithm requires that the examples be described somehow. For this case, let us say that the description consists of two numbers x and y that specify the length and width of one of its petals.

The perceptron algorithm is a member of the class of supervised learning algorithms, which means it can be trained to do the work of classifying ex-

amples according to our criterion and descriptions. First, the algorithm is given descriptions of a set of training examples, as well as the correct label of each. For instance, one flower may have a petal 3 units long and 1 unit wide, and be labeled as a member of species *A*. In a subsequent phase the algorithm is fed with a set of test examples, which consist of descriptions of examples but no labels. The goal of the algorithm is to predict reliably for each test example whether it is true or false, or as in the flower case, an instance of species *A* or species *B*.

The perceptron algorithm works when there is a certain mathematical criterion, known as a linear separator, dividing the two possible classes. This criterion, in the case of our flowers, is a rule of the form

$$px + qy > r$$

where p, q, and r are numbers, such that every flower that satisfies it is of species *A*, and every one that does not is of species *B*. For example, suppose that $p = 2$, $q = -3$ and $r = 2$, so that the rule is

$$2x - 3y > 2.$$

Then a flower with petal length 5 and width 2 would be classified as type *A* since $(2 \times 5) - (3 \times 2) = 4$, and $4 > 2$. On the other hand a flower with petal length 3 and width 2 would be classified as type *B* since $(2 \times 3) - (3 \times 2) = 0$ and $0 < 2$. In graphical terms this means that if all the examples are plotted in two dimensions, representing the length by x and the width by y, then there is a straight line corresponding to equation $2x - 3y = 2$, so that all the species *A* flowers lie on one side of this line, and the species *B* flowers on the other (or on it). This is illustrated in Figure 3.6.

Of course, the perceptron algorithm does not know the true equation for the separator in advance. Instead, it must find it. The algorithm works by scanning through the training data, possibly many times. At each instant it maintains a hypothesis, of the form of $ax + by > c$, about the linear separator. We shall for simplicity work with the case $c = 0$.[20] The algorithm then starts with the hypothesis $0x + 0y > 0$. It goes through each training example one by one, and if the example label is correctly predicted by the current hypothesis, then the hypothesis is not changed. If the example label is not predicted correctly, then the hypothesis is updated so as to be "more likely,"

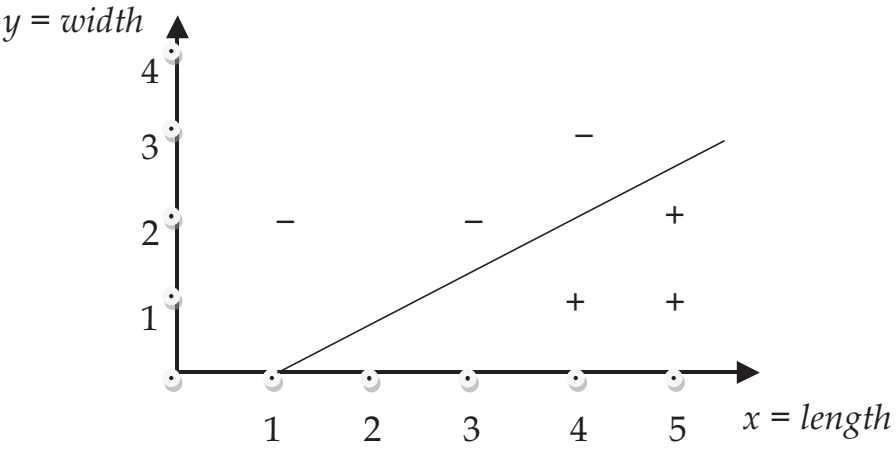

Figure 3.6 The sloping line contains the points satisfying $2x - 3y = 2$. The points $(x = 4, y = 1)$, $(x = 5, y = 1)$, and $(x = 5, y = 2)$ all satisfy $2x - 3y > 2$ and are marked as "+," while the points $(x = 1, y = 2)$, $(x = 3, y = 2)$, and $(x = 4, y = 3)$ do not and are marked as "–." In other words, the flowers of species A will lie below the line, and those of species B above or on it.

in a certain sense, to be correct on that same example if presented again later.

To be more precise, if the hypothesis misclassifies a true positive example (u, v) as negative (i.e., because $au + bv \leq 0$), then a is updated by having u added to it, and b by having v added to it. The left-hand side of the updated hypothesis will then be $(a + u)x + (b + v)y$, and it will have value $(a + u)u + (b + v)v$ if presented with the same example (u, v) on a subsequent run through the data. The value of the sum will be larger than before by a positive quantity $u^2 + v^2$, and hence will be "more likely" to exceed 0 in value and correctly identify the positive example as true. For the opposite case, when a negative example is misclassified to be positive, a is updated by having u subtracted from it, and b by having v subtracted from it. This will have the effect of reducing the value of the left-hand side by $u^2 + v^2$ if the same input (u, v) is presented again later, and hence will make it "more likely" to be less than 0 and hence result in a correct negative classification in that eventuality.

Depending on the order in which training data is fed to the perceptron, it could generate exceedingly many distinct histories of hypotheses. The interesting fact about the perceptron algorithm is that, in spite of our lack of control over its exact fate as we let it loose on arbitrary data, it nonetheless

manages to achieve something quite remarkable. The most basic statement of the power of this algorithm, proved by Albert Novikoff soon after the algorithm was first proposed, is that if there is a true linear separator, then the algorithm is sure to find it, or another hypothesis that also correctly classifies all the examples, after having made misclassifications only a finite number of times. Furthermore, an upper bound on the number of such misclassifications can be computed given the data. This upper bound is equal to M/m^2, where M is the square of the distance of the furthest data point in the training set from the point (0, 0) and m is the margin. The margin has a trickier definition: It is the minimum distance of any data point from the separating line for the line for which this distance is the largest. The consequence of m being in the denominator is that the closer the data points are to the separator, the more mistakes this procedure can potentially make.

The importance of the algorithm derives from several additional facts. First, it works within bounds of the form M/m^2 not just for problems with two variables but for problems with any number of variables. Second, in practice, it often works well even for data that is corrupted by noise. Third, there are general methods for dealing with data that is separable not by linear relations but by more complex curves. For example, suppose that the two categories are not separated by a straight line as they are in Figure 3.6, but we suspect that some more complex curve would separate them. In our two-dimensional case we could try to learn the separator $ax + by + cxy + dx^2 + ey^2 > f$ where x, y are variables and a, b, c, d, and e are the constants to be learned. This inequality is not linear in x, y, since it contains higher order terms such as x^2. However, it can be viewed as linear if we regard the set of variables not as $\{x, y\}$ but as $\{x, y, xy, x^2, y^2\}$. We can translate any example given as a pair $\{x, y\}$ of numbers to the corresponding five numbers $\{x, y, xy, x^2, y^2\}$ by multiplication. In this way the perceptron algorithm can be applied directly to nonlinearly separable data also.

This linearization is an important idea that greatly extends the range of applicability of the perceptron algorithm, but it is not the complete panacea that it may seem. If there are few nonlinear terms, and we know which they are, then there are no problems. But if there are numerous terms potentially to look for, then this will introduce higher, possibly exponential, costs.

The criterion that only a finite number of mistakes are made over any, even infinite, number of examples does not appear to be a natural fit for human learning. It raises the question of what outcome we should really require

True Value	Example	Classification by Previous Hypothesis	Updated Hypothesis
			$0x+0y+0z>0$
+	(4, 1, 1)	−	$4x+1y+1z>0$
−	(1, 2, 1)	+	$3x-1y+0z>0$
+	(5, 1, 1)	+	$3x-1y+0z>0$
−	(3, 2, 1)	+	$0x-3y-1z>0$
+	(5, 2, 1)	−	$5x-1y+0z>0$
−	(4, 3, 1)	+	$1x-4y-1z>0$
+	(4, 1, 1)	−	$5x-3y+0z>0$
−	(1, 2, 1)	−	$5x-3y+0z>0$
+	(5, 1, 1)	+	$5x-3y+0z>0$
−	(3, 2, 1)	+	$2x-5y-1z>0$
+	(5, 2, 1)	−	$7x-3y+0z>0$
−	(4, 3, 1)	+	$3x-6y-1z>0$
+	(4, 1, 1)	+	$3x-6y-1z>0$
−	(1, 2, 1)	−	$3x-6y-1z>0$
+	(5, 1, 1)	+	$3x-6y-1z>0$
−	(3, 2, 1)	−	$3x-6y-1z>0$
+	(5, 2, 1)	+	$3x-6y-1z>0$
−	(4, 3, 1)	−	$3x-6y-1z>0$

Figure 3.7 Example of a run of the perceptron algorithm in three dimensions on the set of six examples +(4, 1, 1), −(1, 2, 1), +(5, 1, 1), −(3, 2, 1), +(5, 2, 1), −(4, 3, 1) repeated in that order three times. The signs indicate the labels of the examples. The initial hypothesis is $0x+0y+0z>0$. The first example (4, 1, 1) when substituted in the left-hand side of the initial hypothesis gives 0, and hence does not satisfy it, as indicated by the negative sign in the third column. The first column indicates that the true label of this first example (4, 1, 1) is positive. The algorithm therefore adds the coordinates (4, 1, 1) of the example to the coefficients (0, 0, 0) of the hypothesis, to give $4x+1y+1z>0$ as the updated hypothesis. After the six examples are cycled through twice, the hypothesis $3x-6y-1z>0$ is obtained. In the third cycle it is confirmed that this hypothesis satisfies all six examples.

of a learning algorithm before we declare it successful. This is the main question that will be addressed in Chapter 5. Before we get there, however, we need to take a more general look at what a computationally sound, mechanistic explanation of a natural phenomenon—whether of evolution, or cognition, or some other process of interest—might look like.

But there is one intuition suggested by the perceptron algorithm that will be important for what comes later. Learning is achieved in many steps that are plausible but innocuous when viewed one by one in isolation. These steps work because there is an overall algorithmic plan. In combination the steps achieve something, in particular, some kind of convergence. We shall claim that evolution is similar. The many small steps taken do not make too much sense one by one. But there is an algorithmic plan, so that taken in unison the many steps do achieve something remarkable.