Computability and Complexity

Jon Kleinberg* Christos Papadimitriou†

(In *Computer Science: Reflections on the Field, Reflections from the Field*, Natl. Academies Press, 2004.)

1 The Quest for the Quintic Formula

One of the great obsessions of Renaissance sages was the solution of polynomial equations: find an x that causes a certain polynomial to evaluate to 0. Today we all learn in school how to solve quadratic equations (polynomial equations of degree two, such as $ax^2 + bx + c = 0$), even though many of us have to look up the formula every time (it's $x = \frac{1}{2a}(-b \pm \sqrt{b^2 - 4ac})$). Versions of this formula were known to the Babylonians as early as 2000 BC, and were rediscovered in many ancient cultures. The discovery of similar but much more complicated formulas for solving equations of degree three and four — the cubic and quartic formulae — had to wait until the 16th century AD. During the next three centuries, the greatest minds in Europe strove unsuccessfully to discover the *quintic formula*, cracking equations of degree five, until the flowering of modern algebra brought the quest to a sudden, surprising resolution: A proof that *there is no quintic formula*.

This story, on first hearing, can engender a few natural reactions. Among them, surprise — what's the obstacle to a quintic formula? Why was it so hard to prove it didn't exist? And, more subtly, a mild sense of perplexity — what do we mean by a quintic formula anyway? Why can't we write "the largest x such that $ax^5 + bx^4 + cx^3 + dx^2 + ex + f = 0$" and declare this to be a formula?

So we back up. By a "formula" in this story, we meant a particular thing: a finite sequence of steps that begins with the given values of the coefficients and ends with a root x; each step consists of one of the basic arithmetic operations applied to certain of the quantities already computed, or else it consists of the extraction of a root of one of the quantities already computed. Now we can assert more precisely, thanks to the work of the nineteenth-century mathematicians Abel and Galois: there is no quintic formula.

Viewed from the safe distance of a few centuries, the story is clearly one about computation, and it contains many of the key ingredients that arise in later efforts to model computation: We take a computational process that we understand intuitively (solving an equation, in this case), formulate a precise model, and from the model derive some highly unexpected consequences about the computational power of the process. It is precisely this approach that we wish to apply to computation in general. But moving from this example

*Department of Computer Science, Cornell University, Ithaca NY 14853.
†Department of EECS, University of California at Berkeley, Berkeley, CA 94720.

to a fully general model of computation requires some further fundamental ideas, because the notion of a "formula" — a straight-line recipe of arithmetic operations — omits two of the crucial properties of general-purpose computation. First, computation can be repetitive; we should be able to perform some action over and over until a certain stopping condition is satisfied. Second, computation should contain "adaptive" steps of the following form: test whether a certain condition is satisfied; if it is, then perform action A; if it isn't, then perform action B. Neither of these is present in straight-line formulas; but a little reflection convinces one that they are necessary to specify many of the other activities that we would consider computational.

2 Computation as a Universal Technology

So, guided by this intuition, let us move beyond stylized forms of computation and seek to understand general-purpose computation in all its richness — for if we could do this, then we might find similarly suprising consequences that apply much more broadly. Such was the goal of Alan Turing in the 1930's; and such was also the goal of a host of other mathematical logicians at that time.

Turing's entry in this field is particularly compelling, not so much because of its mathematical elegance but because of its basic, common-sensical motivation and power. He sought a streamlined mathematical description of what goes on when a person is performing calculations in a large notebook: he or she writes down and erases symbols, turns the pages left or right, keeps a limited number of symbols in his or her memory. The computing device Turing proposed — the Turing machine — has access to an infinite sequence of "pages", each of which can hold only one symbol. At any time, the machine can be in one of a finite set of possible "states of mind" — its working memory. The flow of control proceeds simply as follows: based on its current state, and the symbol it is currently reading, the machine may write a new symbol on the current page (erasing the existing one), move to the next or preceding page, and change its state. Subject to these rules, the Turing machine processes the input it is given, and may eventually choose to halt — at which point the notebook contains its output.

Why should we accept this model? First, it accords well with common sense. It seems to be the way that symbolic computation, as performed slowly and painfully by humans, proceeds. Indeed, with some practice, one can implement seemingly any natural symbolic task on a Turing machine. Second, it is robust — a version of the model with very small sets of available symbols and states (say, eight of each) is, in a precise sense, just as powerful as a version with an arbitrary finite set of each — only the control rules become more complicated. Moreover, it does not matter that the "pages" on which the computation is performed are arranged in a linear sequence; it would not add to the power of the model if we arranged them instead in an arbitrary Web of connections. Finally, and most crucially, Turing's model is precisely equivalent to the other formalisms proposed in his day — among them, and preceding it, Alonzo Church's lambda calculus — and it is also equivalent to modern general-purpose programming languages such as C and Java (with access to an arbitrary amount of memory).

For the accumulation of these reasons, we are justified in believing that we have arrived

at the "right" model of computation; and this is the content of the Church-Turing thesis: a symbolic function is computable if and only if it can be computed on a Turing machine (or its equivalents). It is important to notice what is being claimed: we have not derived the notion of "computability" from a set of more primitive axioms; rather, after extensive thought experiments, we have asserted that "computability" corresponds precisely to what can be computed on a Turing machine.

Accepting the Turing machine as the basis for our precise definition of computability is a momentous step. Its first consequence is that of universality; there is a "universal Turing machine" U that does the following. As input, U receives the description of a Turing machine M (the "code" of M, written in U's book), and an input n to M (a later chapter in the same book). As output, U returns the result, if any, of running M on n. Today, we would think of U simply as an interpreter — it executes a step-by-step simulation of any Turing machine M presented to it. Indeed, our style of writing programs and then executing them is so ingrained in our view of computation that it takes a moment to appreciate the consequences that flow from a universal machine. It means that programs and data are really the same thing: a program is just a sequence of symbols that looks like any other piece of input; but when fed to a universal machine, this input wakes up and begins to compute. Think of mobile code, java applets, e-mail viruses: your computer downloads them as data, and then runs them as programs.

The principle of interchangeable parts — that components of a machine can be mass-produced independently, and a working whole can be assembled from a random selection of one part from each kind — was the disruptive insight that underpinned the Industrial Revolution. Universality is perhaps an even more radical approach to assembling systems: a single computer on your desk can run your word processor, your spreadsheet, and your on-line calendar, as well as new applications not yet conceived of or written. And while this may seem completely natural, most of technology in fact does not work this way at all. In most aspects of one's technological life, the device is the application; they are one and the same. If you own a radio and want to watch TV, you must buy a new device. If you want to drive, you use a car; but if you want to fly, you use an airplane. Your car cannot download a new set of instructions and suddenly know how to fly, or how to maneuver underwater. But the computational world has a flexibility of application that cannot really be imagined elsewhere — and that is because the world of computation is powered by universal machines.

We have all seen the consequences of this flexibility very clearly in the past decade, as the World Wide Web became a new medium within a mere seven years of its introduction. If we look at other media — at the phone, the radio, the television — it took a much longer time from their inceptions to widespread prominence; what was the difference? Of course there are many factors that one can point to, but mingled among them is the universality of the computers on which Web protocols and interfaces run. People had already bought personal computers for their homes, and built office information systems around them, before the Web ever existed; so when the Web arrived, it could spread through this infrastructure with amazing speed. One cannot really imagine an analogue of this process for the television — it would be as though millions of Americans had been induced to buy large inert boxes for their living rooms, and a decade later someone dreamed up the technology to begin broadcasting pictures to them. But this is more or less what happened with the Web.

3 The Limits of Computation

Computer science was born knowing its own limitations. For the strength of the universal machine leads directly to a second, and more negative, consequence — uncomputability, the fact that certain natural computational tasks cannot be carried out by Turing machines or, by extension, computer programs. The leap from the notion of universality to this impossibility result is surprisingly effortless, if ingenious. It is rooted in two basic issues — first, the surprising difficulty in determining the "ultimate" behavior of a program; and second, the self-referential character of the universal machine U.

To appreciate the first of these, recall that our universal machine U simply performed a step-by-step simulation of a Turing machine M on an input n. This means that if M computes forever, never halting, then U's simulation of M will run forever as well. This is the notion of an "infinite loop," familiar to beginning (and experienced) programmers everywhere — your program keeps running with no sign of any output; do you stop it and see what's wrong, or wait to see if it's just taking longer than expected to come up with an answer? We might well want something stronger than the blind simulation that U provides; we might want a "Universal Termination Detector" — a Turing machine D that behaves as follows. Given a description of a Turing machine M and an input n to M, the machine D performs a finite number of steps, and then correctly reports whether or not M will ever halt with a result when it is run on n. (So in particular, the machine D itself halts on every input.)

Could one build such a thing? One's first reaction is to start dreaming up tricks by which one could look at a program and determine whether it will halt or not — looking for obviously repetitive behavior with no stopping condition. But gradually the problem begins to look hopelessly difficult. Maybe the program you're analyzing for termination is systematically enumerating the natural numbers, searching for a counter-example to a famous conjecture in mathematics; and it will only stop when it finds one. So a demonstration that this single program eventually terminates must implicitly resolve this mathematical conjecture! Could detecting the termination of programs really be as hard as automating mathematics?

This thought experiment raises the suggestion that we should perhaps be considering the problem from the other direction, trying to show that it is not possible to build a Universal Termination Detector. Another line of reasoning that might make us start considering such an impossibility result is, as suggested above, the self-referential nature of the universal machine U: U is a Turing machine that can simulate the behavior of any Turing machine. So, in particular, we could run U on a description of itself; what would happen? When you find yourself asking questions like this about a mathematical object — questions in which the object refers to itself — there are often explosive consequences lying just ahead. Indeed, the dangerous properties of self-reference appear in ordinary discourse. From ancient Greece we have Eubulides' paradox, which asserts, "This statement is false" — is this a true statement or a false one? Or consider Bertrand Russell's hypothetical barber, who shaves the set of all people who do not shave themselves — who then shaves the barber himself?

In the case at hand, we can exploit the self-reference inherent in universality to prove that there is no Universal Termination Detector.

There is no Universal Termination Detector

We begin by observing that the set of all Turing machines, while clearly infinite, can be enumerated in a list $M_1, M_2, M_3, \ldots$ in such a way that each Turing machine appears once in the list. To do this, we can write a description of each Turing machine as a sequence of symbols, and then order these descriptions in alphabetical order; we first list all descriptions with one symbol (if there are any), then all descriptions with two symbols, and so forth.

Our impossibility proof proceeds by assuming that there exists a Universal Termination Detector; we then show how this leads to a contradiction, establishing that our initial assumption cannot be valid. So to begin, let D be a Turing machine that is a Universal Termination Detector.

We construct, from D, a Turing machine X that will lead to the contradiction. On input n, here is what X does. It first invokes the Termination Detector D to determine whether the machine M_n will ever halt when run with input n. (This is the core of the self-reference: we investigate the behavior of M_n on its own position n in the alphabetical listing of Turing machines.) If it turns out that M_n will never halt on n, then X halts. But if it turns out that M_n will halt when run on n, then X gratuitously throws itself into an infinite loop, never halting.

X is not a very useful program; but that is not the point. The point is to notice that X is itself a Turing machine, and hence is one of the machines from our list; let us suppose that X is really M_k. The self-reference paradoxes we mentioned above — Eubulides' and Russell's — were both triggered by asking a natural question that exposed the latent contradiction. Our proof here employs such a question, and it is this: does X halt when it is run on input k?

We consider this question as follows. Suppose that X halts when it is run on k. Then, since we know X is really M_k, it follows that M_k halts on k; and so, by our construction of X, X should not halt when it is run on k. On the other hand, suppose that X does not halt when it is run on k. Then, again using the fact that X is really M_k, it follows that M_k does not halt on k; and so X should halt on k. So neither outcome is possible — X cannot halt on k, and it cannot fail to halt on k! This is a contradiction, so there cannot be such a machine X, and hence there cannot be a Universal Termination Detector D.

This style of proof is often referred to as *diagonalization*, and it was introduced by Cantor to show that one cannot put the real numbers in one-to-one correspondence with the integers. The term "diagonalization" here comes from the following intuition. We imagine drawing an infinite two-dimensional table whose rows correspond to the Turing machines $M_1, M_2, M_3, \ldots$ and whose columns correspond to all the possible inputs $1, 2, 3, \ldots$ Each entry of this table — say the entry at the meeting of row i and column j — indicates whether or not machine M_i halts when it is run on input j. Viewed in these terms, our supposed machine X "walks down the diagonal" of this table; on input k, it consults the table entry at the meeting of row k and column k, and essentially inverts the answer that it finds there.

This is a first, fundamental impossibility result for computation — a natural problem that cannot be solved computationally. And starting with this result, impossibility spreads like a shock wave through the space of problems. We might want a Universal Equivalence Tester: given two Turing machines M and M', are they equivalent? Do they produce the same result on every input? But it is easy to convince yourself that if we could build an Equivalence Tester, we could use it to build a Termination Detector, which we know cannot exist. And so: There is no Universal Equivalence Tester. We might want a Universal Code Verifier: given a Turing machine M and an "undesirable" output n, is there any input that

will cause M to produce the output n? But again, from a Code Verifier we could easily build a Termination Detector. No Termination Detector, no Code Verifier.

Suppose you want to verify that the program you've just written will never access a restricted area of memory; or suppose you want to ensure that a certain revision — a transformation of the program — will not cause it to change its behavior. Research in the area of programming languages has developed powerful techniques for program verification and transformation tasks, and they are used effectively to analyze complex programs in practice [see the essay by Al Aho and Jim Larus in this volume]. Such techniques, however, are developed in a constant struggle against the negative results discussed above: over the years researchers have carved out broader and broader tractable special cases of the problem, but to solve these verification and transformation tasks in their full generality — to perform them correctly on all possible programs — is provably impossible. Such results impose fundamental limitations on our ability to implement and reason about complex pieces of software; they are among the laws that constrain our world.

4 When Finite is not Good Enough

Computers, as we think of them now, did not exist when Turing carried out his seminal work. But by the 1950's and 1960's, as truly automated computation became increasingly available, a growing amount of attention was devoted to the study and development of *algorithms* — step-by-step computational procedures, made precise by the notion of computability. And as this development began to gather force, it became clear that uncomputability was only one of the laws that constrained our ability to solve problems. The world abounded in problems whose solvability was not in doubt — but for which solving any but the smallest instances seemed practically infeasible.

Some of the most vivid of these problems came from the area of operations research, a field that sprang in large part from the epiphany — conceived during World War II, and spilling into civilian life ever after — that there was a science to the efficient coordinated movement of armies and organizations, the efficient allocation of supplies and raw materials. Thus, we can consider the Traveling Salesman Problem: You are given a map of N cities and the distances between them, as well as a "budget" B; you must visit all the cities via a tour whose total length is at most B. Or consider the Matching Problem: You must pair up $2N$ newly admitted college students — some of whom don't like one another — into N pairs of roommates, so that each pair consists of students who will get along.

In the 1960's, Jack Edmonds came up with a beautiful and efficient algorithm to solve the Matching Problem; and he wrote a paper describing the method. But how should one describe the result, actually? "A computational solution to the Matching Problem"? — this is not quite right, since there's an obvious way to solve it: try all possible pairings, and see if any of them works. There was no question that the Matching Problem had a computable solution in Turing's sense. The crux of the result was in the efficiency of the solution. Jack Edmonds understood this difficulty very clearly: "I am claiming, as a mathematical result, the existence of a *good* algorithm for finding a ... matching in a graph. There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph. It is by no means obvious whether *or not* there exists an algorithm whose difficulty increases only algebraically with the size of the graph."

It is hard to find much to add to this. There is clearly an algorithm that solves the Matching Problem in a number of steps that is exponential in N — at least N factorial. But try to imagine how long this would take, even on the fastest computers we have today — the problem of forming 30 pairs could require an amount of time comparable to the age of the universe. Yes, the solution is computable; yes, we can even imagine how the whole computation will proceed; but such a method is of no real value to us at all if we are seeking a solution to a problem of even moderate size. We need to find a qualitatively faster method; and this is exactly what Jack Edmonds had accomplished.

Edmonds's concern with "good algorithms" fit naturally into a research agenda that was being pursued contemporaneously by Juris Hartmanis and Richard Stearns — that of determining the intrinsic *computational complexity* of natural problems, by determining the smallest number of computational steps that are required to produce a solution. And so, following this line of attack, we seek algorithms that require only "polynomial time" — on an input of size N, a "good" algorithm should use a number of steps that is bounded by a polynomial function of N such as N^2 or N^5. Clearly polynomial functions grow much more slowly than exponential ones, and they have a very desirable scaling property — if you increase your input size by a constant multiplicative factor, the running time goes up by a predictable constant factor as well. But however much one tries to justify our interest in polynomial time on theoretical grounds, its primary justification follows the same lines as the Church-Turing thesis that we saw earlier: it accords well with our experience from practice. Problems solvable by polynomial-time algorithms tend overwhelmingly to be efficiently solvable in practice; and for problems that lack polynomial-time algorithms, one tends to encounter enormously difficult instances with some regularity.

The choice of polynomial time as a mathematical surrogate for efficiency has served computer science very well — it has been a powerful guide to the design of good algorithms. And algorithmic efficiency has proved to be a far more subtle concept than we could have imagined. Consider again the Traveling Salesman Problem and the Matching Problem. For each, the "search space" is enormous: for the Traveling Salesman, any ordering of the N cities forms a tour that must in principle be considered; for the Matching Problem, any possible pairing of the individuals is a candidate solution. And yet, despite similarities at this level, their behavior from the point of view of computational difficulty seems to be utterly divergent. Matching has a polynomial-time algorithm, and very large problems are solved every day in practice. For the Traveling Salesman Problem, on the other hand, we are famously without a polynomial-time algorithm, and the solution of relatively small instances can still require a major computational effort.

Where does this enormous difference in computational complexity lie? What features of a computational problem determine its underlying difficulty? The ongoing effort to resolve these questions is a core research activity in computer science today; it has led to a rich theory of computational intractability — including the notion of NP-completeness, which has spread from computer science into the physical, natural, and social sciences. It has also led to the celebrated "P versus NP" question, which has drawn the attention of mathematicians as well as computer scientists, and is now featured on several lists of the foremost open questions in mathematics.

Does P equal NP?

We have been concerned with the set of all problems that can be solved by a polynomial-time algorithm; let's use P to denote this set of problems.

Now, we believe that the Traveling Salesman Problem is very difficult to solve computationally; it is likely that this problem does not belong to the set P we have just defined. But there is at least one good thing we can say about its tractability. Suppose we are given N cities, the distances between them, and a budget B; and suppose that in fact there exists a tour through all these cities of length at most B. Then there exists a way for someone to prove this to us fairly easily: he or she could simply show us the order in which we should visit the cities; we would then tabulate the total distance of this tour and verify that it is at most B.

So the Traveling Salesman Problem may not be efficiently solvable, but it is at least efficiently verifiable: if there is a short tour among the cities we are given, then there exists a "certificate" — the tour itself — that enables us to verify this fact in polynomial time. This is the crux of the Traveling Salesman Problem's complexity, coiled like the "trick" that helps you solve a difficult puzzle: it's hard to find a short tour on your own, but it's easy to be convinced when the short tour is actually revealed to you. This notion of a certificate — the extra piece of information that enables you to verify the answer — can be formalized for computational problems in a general way. As a result, we can consider the set of all problems that are efficiently verifiable in this sense. This is the set NP — the set of all problems for which solutions can be checked (though not necessarily solved) in polynomial time.

It is easy to show that any problem in P must also belong to NP; essentially, this is as easy as arguing that if we can solve a problem on our own in polynomial time, then we can verify any solution in polynomial time as well — even without the help of an additional certificate. But what about the other side of the question: is there a problem that belongs to NP but not to P, a problem for which verifying is easy but solving is hard? Although there is widespread belief that such problems must exist, the issue remains unresolved; this is the famous "P versus NP" question.

To address this question, it is natural to seek out the "hardest" problems in NP, for they are the best candidates for problems that belong to NP but not to P. How can we formalize the notion that one problem is at least as hard as another? The answer lies in reducibility, an idea that came up implicitly when we discussed computational impossibility. We say that a problem A is "reducible" to a problem B if, given a "black box" capable of solving instances of B in polynomial time, we can design a polynomial-time algorithm for A. In other words, we are able to solve A by drawing on a solution to B as a "resource." It follows that if we actually had a polynomial-time algorithm for problem B, we could use this as our "black box," and hence design a polynomial-time algorithm for problem A. Or, simply running this reasoning backward, if there is no polynomial-time algorithm for A, then there cannot be a polynomial-time algorithm for B: problem B is at least as hard as problem A.

So here is a natural thing we might search for: a single problem B in NP with the property that *every* problem in NP is reducible to B. Such a problem would, quite conclusively, be among the hardest problems in NP — a solution to it would imply a solution to everything in NP. But do such problems exist? Why should there be a single problem that is this powerful?

In the early 1970's, Steve Cook in North America and Leonid Levin in the Soviet Union independently made the crucial breakthrough, discovering a number of natural problems in NP with precisely this property: everything in NP can be reduced to them. Today we say that such a problem is "NP-complete," and research over the past decades has led to the discovery that there are literally thousands of natural NP-complete problems, across all the sciences. For example, determining the winner(s) under a wide variety of election and auction schemes is NP-complete. Optimizing the layout of the gates in a computer chip is NP-complete. Finding the folding of minimum energy for discrete models of proteins is NP-complete. The Traveling Salesman Problem — the tough nut that started us on this road — is NP-complete. And an important thing to bear in mind is this: because *every* problem in NP is reducible to any of these, they are all reducible to each other. There is a polynomial-time algorithm for one if and only if there is a polynomial-time algorithm for all. So we have come to realize that researchers in a host of different areas, struggling over a spectrum of computationally intractable problems, have in a fairly precise sense all been struggling over the same problem; this is the great insight that NP-completeness has given us.

Exponential growth is a recurring theme in computer science, and it is revealing to juxtapose two of its fundamental roles: in Moore's Law, which charts the exponential growth in the power of computing machinery [see Mark Hill's essay in this volume], and in computational complexity, which asserts that the effort needed to solve certain problems grows exponentially in their size. Do these two principles cancel out? If our computing power is growing exponentially over time, will we really be bothered by problems of exponential complexity? The answer is that Moore's Law does not make our concerns about exponential complexity go away — and it is important to realize why. The full search space for a 17-city instance of the Traveling Salesman Problem is 16 times larger than the search space for a 16-city instance. So if exhaustively searching the space of solutions for a 16-city problem is at the limit of your current computer's abilities, and if computing power doubles every year and a half, then you'll need to wait six years before your new computer can tackle a 17-city problem by brute force — six years to be able to solve a problem that is only one city larger! Waiting for Moore's Law to deliver better computing power only gets you so far, and it does not beat down the exponential complexity of a deeply intractable problem. What is needed is not just better hardware on which to apply brute force, but a better *algorithm* for finding a solution — something like what Edmonds found for the Matching Problem.

The fact that simply-stated problems can have enormous complexity, with solutions that are computationally very difficult to find, has led to new perspectives on a number of well-studied ideas. Cryptography has been revolutionized by the theory of complexity, for the design of secure communication protocols is a field that exists in a mirror-world where difficult computational problems — codes that are easy to apply and hard to break — are resources to be cultivated [see Madhu Sudan's essay in this volume]. The RSA public-key cryptosystem was inspired by the presumed difficulty of factoring large integers, with the prime factors of a number N forming the hidden "key" whose knowledge allows for easy decryption. The age-old notion of *randomness* — a concept that is intuitively apparent but notoriously tricky to define — has been given an appealing formalization based on computational complexity: a sequence of digits appears "random" to an observer if it is computationally difficult for the observer to predict the next digit with odds significantly better than guessing.

For designers of algorithms, we have seen that their struggle with the complexity of computation has proceeded at a number of different levels. One boundary divides the computable from the uncomputable — it is feasible to build a step-by-step interpreter for computer programs, but one cannot design an algorithm that decides whether arbitrary programs will terminate. Another boundary divides polynomial-time solvability from the exponential growth of brute-force search. But while polynomial time is indeed a good high-level means for gauging computational tractability, there are an increasing number of applications, typically involving very large datasets, where simply having a polynomial-time algorithm is far from adequate. Suppose the size of the input is measured in gigabytes; an algorithm that takes a number of steps equal to the cube of the input size is no more useful in practice than one that never terminates.

None of this is to say that polynomial time has lost its relevance to the design of algorithms. But for many large-scale problems, we are faced with a reprise of the situation in the 1950's and 1960's, when we tumbled from a concern with computability to a concern with computational complexity: as our real-world computational needs expand, our guidelines for

what constitutes an "acceptable" algorithm become increasingly stringent. And this in turn has led to new lines of research, focusing for example on algorithms that must run in time very close to the size of the input itself; algorithms that must "stream" through the input data in one pass, unable to store significant parts of it for post-processing.

While algorithm design has deepened into the study of increasingly time-efficient techniques, it has also opened outward, revealing that running time is just one of many sources of complexity that must be faced. In many applications — scheduling under real-time conditions, or managing a busy network — the input is not a static object but a dynamic one, with new data arriving continuously over time. Decisions must be made and solutions must be constructed adaptively, without knowledge of future input. In other applications, the computation is *distributed* over many processors, and one seeks algorithms that require as little communication, and synchronization, as possible. And through all these settings, from NP-complete problems onward, algorithms have been increasingly designed and analyzed with the understanding that the *optimal* solution may be unattainable, and that the optimum may have to serve only as an implicit benchmark against which the quality of the algorithm's solution can be measured.

5 The Lens of Computation

Our contemplation of computation has led us quickly to the "P versus NP" question, now considered to be among the deepest open problems in mathematics. More recently, our views on complexity have been influenced by the striking confluence of computation and quantum physics: What happens to our standard notions of running time and complexity when the computation unfolds according to the principles of quantum mechanics? It is now known that access to such a hypothetical "quantum computer" would yield polynomial-time algorithms for certain problems (including integer factorization and the breaking of the RSA cryptosystem) that are believed to be intractable on standard computers. What are the ultimate limits of quantum computers? And are there theoretical obstacles to building them (in addition to the practical ones currently braved in labs all over the world)? These questions, purely computational in their origin, present some of the most daunting challenges facing theoretical physics today. [See Charles Bennett's essay in this volume.]

Biology is a field where the synergy with computation seems to go ever deeper the more we look. Let us even leave aside all the ways in which sophisticated algorithmic ideas are transforming the practice of biology — with vast genomic and structural databases, massive numerical simulations of molecules, the fearsome symbol-crunching of the Human Genome Project — let us leave all this aside and consider how we might view molecular biology itself through a computational lens, with the cell as an information-processing engine. For fundamentally, a biological system like a cell is simply a chemical system in which the information content is *explicit*. The genome is part of the overall chemical system, but it is not there to take part in the chemistry itself — rather, it is there as a sophisticated encoding of the chemical processes that will take place. It is a programming abstraction — it is the representation of the real thing, co-existing with the thing itself.

Just as computation distinguishes itself from the rest of technology, so are biological systems intrinsically different from all other physical and chemical systems — for they too

have separated the application from the device. We can take a cell and change a few symbols in its genome — splice in a new gene or two — and we can cause its chemistry to change completely. We have replaced one piece of code with another; the device — the cellular hardware — has been left alone, while we simply changed the application running on it. This phenomenon really seems to have no parallel in the other sciences. Surely, the non-biological world obeys fundamental laws, but it does not contain — and actually implement — an explicit representation of these laws. Where in the solar system are the few molecules, encoding the laws of gravity and motion, which we could modify to cause the planets to follow more eccentric orbits?

Computation as a technology that follows its own laws; computation as the quintessence of universality; computation as a powerful perspective on the world and on science— these are issues that still drive our study of the phenomenon today. And the more we grapple with the underlying principles of computation, the more we see their reflections and imprints on all disciplines — in the way structured tasks can be cast as stylized computational activities; in the surprising complexity of simple systems; and in the rich and organic interplay between information and code.