EXCERPTED FROM

STEPHEN WOLFRAM

A NEW KIND OF SCIENCE

SECTION 12.8

Undecidability and Intractability

more input. And often the actual form of this train of thought is influenced by memory we have developed from inputs in the past—making it not necessarily repeatable even with exactly the same input.

But it seems likely that the individual steps in each train of thought follow quite definite underlying rules. And the crucial point is then that I suspect that the computation performed by applying these rules is often sophisticated enough to be computationally irreducible—with the result that it must intrinsically produce behavior that seems to us free of obvious laws.

Undecidability and Intractability

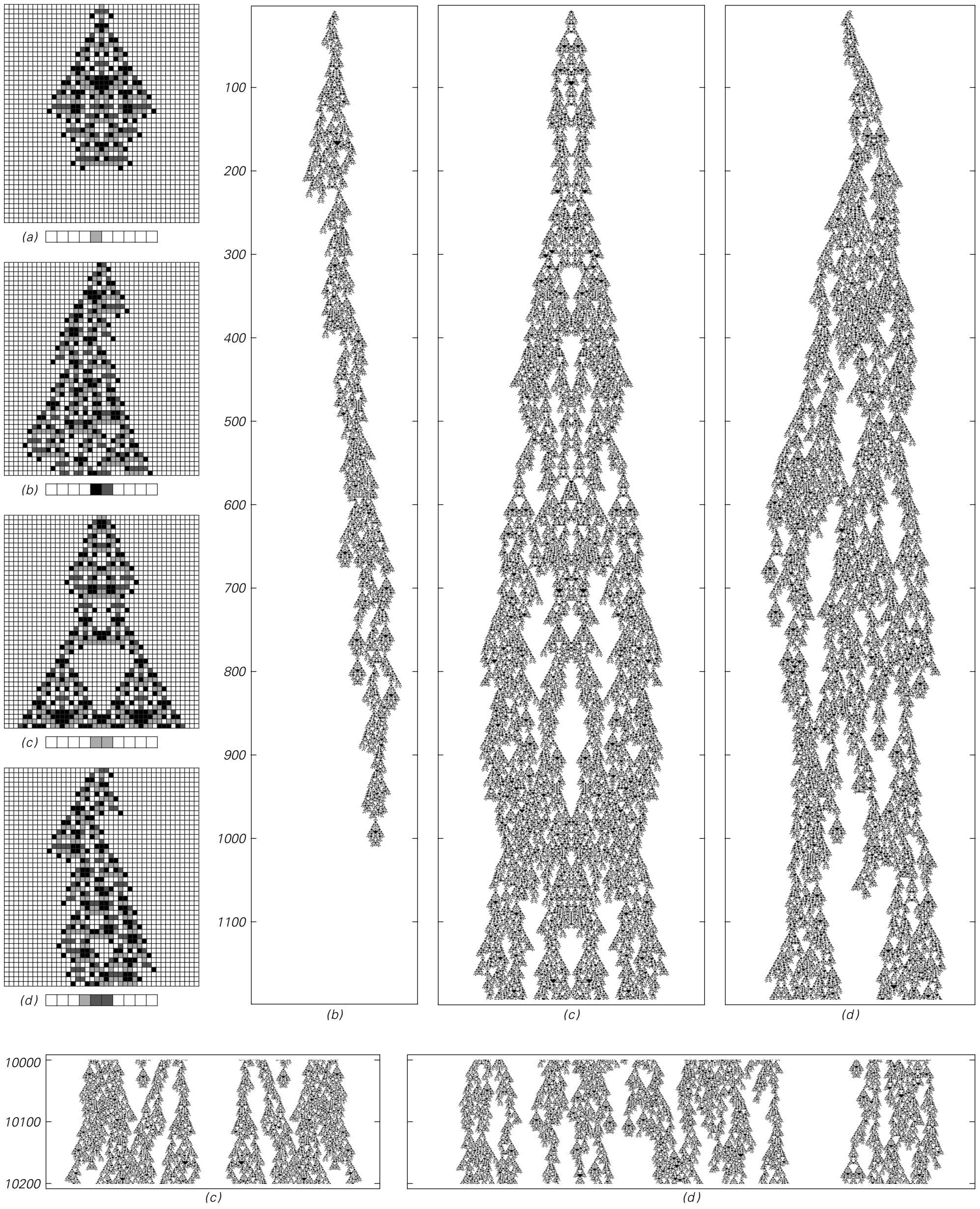
Computational irreducibility is a very general phenomenon with many consequences. And among these consequences are various phenomena that have been widely studied in the abstract theory of computation.

In the past it has normally been assumed that these phenomena occur only in quite special systems, and not, for example, in typical systems with simple rules or of the kind that might be seen in nature. But what my discoveries about computational irreducibility now suggest is that such phenomena should in fact be very widespread, and should for example occur in many systems in nature and elsewhere.

In this chapter so far I have mostly been concerned with ongoing processes of computation, analogous to ongoing behavior of systems in nature and elsewhere. But as a theoretical matter one can ask what the final outcome of a computation will be, after perhaps an infinite number of steps. And if one does this then one encounters the phenomenon of undecidability that was identified in the 1930s.

The pictures on the next page show an example. In each case knowing the final outcome is equivalent to deciding what will eventually happen to the pattern generated by the cellular automaton evolution. Will it die out? Will it stabilize and become repetitive? Or will it somehow continue to grow forever?

One can try to find out by running the system for a certain number of steps and seeing what happens. And indeed in example (a) this approach works well: in only 36 steps one finds that the pattern



Cellular automaton evolution illustrating the phenomenon of undecidability. Pattern (a) dies out after 36 steps; pattern (b) takes 1017 steps. But what the final outcome in cases (c) and (d) will be is not clear after even a million steps. And in general there appears to be no finite computation that can guarantee to determine the final outcome of the evolution after an infinite number of steps. The cellular automaton rule used is a 4-color totalistic one with code 1004600. Whether a pattern in a cellular automaton ever dies out can be viewed as analogous to a version of the halting problem for Turing machines.

dies out. But already in example (b) it is not so easy. One can go for 1000 steps and still not know what is going to happen. And only after 1017 steps does it finally become clear that the pattern in fact dies out.

So what about examples (c) and (d)? What happens to these? After a million steps neither has died out; in fact they are respectively 31,000 and 39,718 cells wide. And after 10 million steps both are still going, now 339,028 and 390,023 cells wide. But even having traced the evolution this far, one still has no idea what its final outcome will be.

And in any system the only way to be able to guarantee to know this in general is to have some way to shortcut the evolution of the system, and to be able to reduce to a finite computation what takes the system an infinite number of steps to do.

But if the behavior of the system is computationally irreducible—as I suspect is so for the cellular automaton on the facing page and for many other systems with simple underlying rules—then the point is that ultimately no such shortcut is possible. And this means that the general question of what the system will ultimately do can be considered formally undecidable, in the sense there can be no finite computation that will guarantee to decide it.

For any particular initial condition it may be that if one just runs the system for a certain number of steps then one will be able to tell what it will do. But the crucial point is that there is no guarantee that this will work: indeed there is no finite amount of computation that one can always be certain will be enough to answer the question of what the system does after an infinite number of steps.

That this is the case has been known since the 1930s. But it has normally been assumed that the effects of such undecidability will rarely be seen except in special and complicated circumstances. Yet what the picture on the facing page illustrates is that in fact undecidability can have quite obvious effects even with a very simple underlying rule and very simple initial conditions.

And what I suspect is that for almost any system whose behavior seems to us complex almost any non-trivial question about what the system does after an infinite number of steps will be undecidable. So, for example, it will typically be undecidable whether the evolution of the system from some particular initial condition will ever generate a specific arrangement of cell colors—or whether it will yield a pattern that is, say, ultimately repetitive or ultimately nested.

And if one asks whether any initial conditions exist that lead, for example, to a pattern that does not die out, then this too will in general be undecidable—though in a sense this is just an immediate consequence of the fact that given a particular initial condition one cannot tell whether or not the pattern it produces will ever die out.

But what if one just looks at possible sequences—as might be used for initial conditions—and asks whether any of them satisfy some constraint? Even if the constraint is easy to test it turns out that there can again be undecidability. For there may be no limit on how far one has to go to be sure that out of the infinite number of possible sequences there are really none that satisfy the constraint.

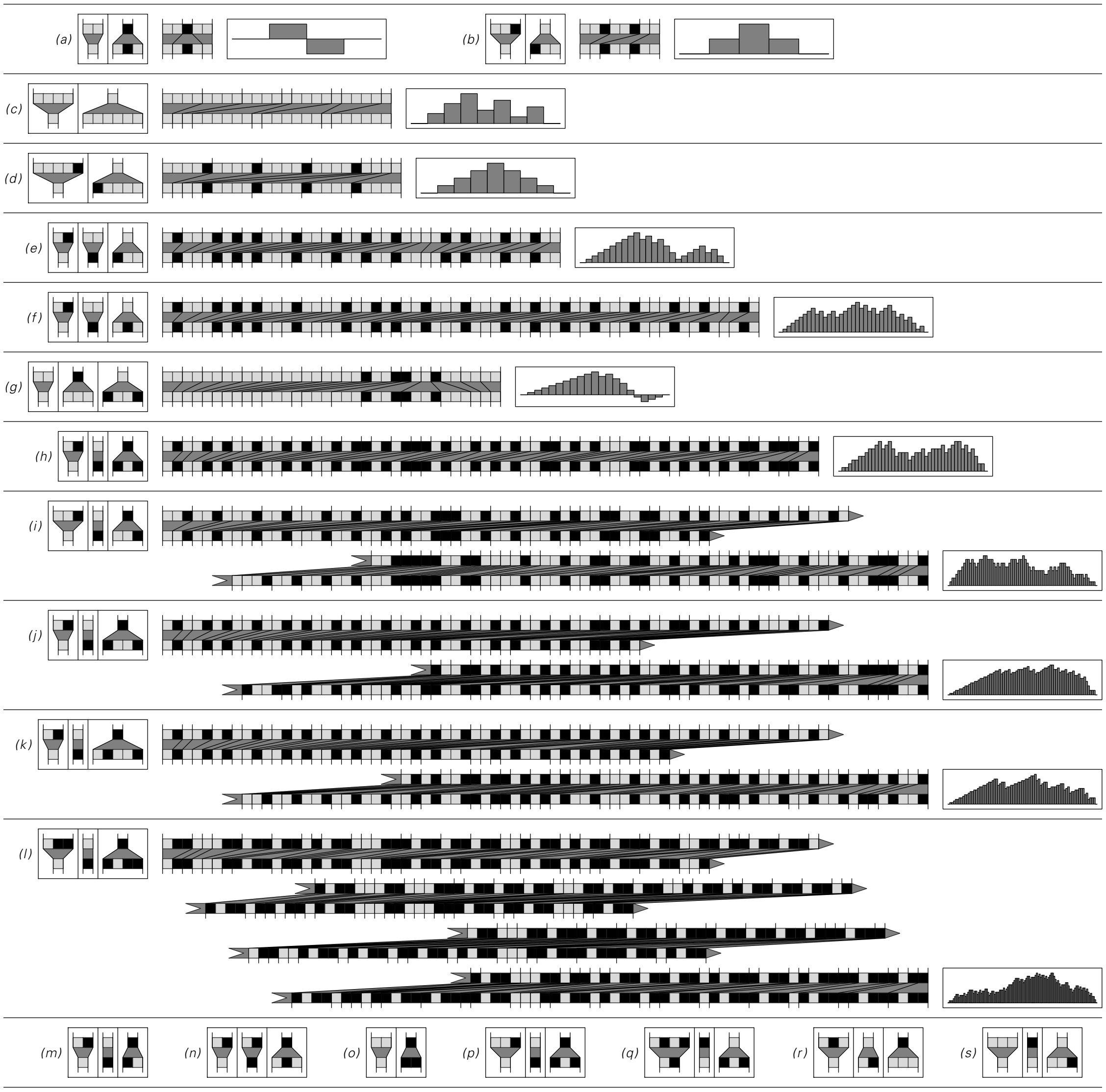
The pictures on the facing page show a simple example of this. The idea is to pick a set of pairs of upper and lower blocks, and then to ask whether there is any sequence of such pairs that satisfies the constraint that the upper and lower strings formed end up being in exact correspondence.

When there are just two kinds of pairs it turns out to be quite straightforward to answer this question. For if any sequence is going to satisfy the constraint one can show that there must already be a sequence of limited length that does so—and if necessary one can find this sequence by explicitly looking at all possibilities.

But as soon as there are more than two pairs things become much more complicated, and as the pictures on the facing page demonstrate, even with very short blocks remarkably long and seemingly quite random sequences can be required in order to satisfy the constraints.

And in fact I strongly suspect that even with just three pairs there is already computational irreducibility, so that in effect the only way to answer the question of whether the constraints can be satisfied is explicitly to trace through some fraction of all arbitrarily long sequences—making this question in general undecidable.

And indeed whenever the question one has can somehow involve looking at an infinite number of steps, or elements, or other things, it



Examples of a class of one-dimensional constraints where it is in general undecidable whether they can be satisfied. The constraints require that concatenating in some order the blocks shown should yield identical upper and lower strings. In cases (a)–(l) the constraints can be satisfied, and the minimal strings which do so are shown. The plots to the right give the successive differences in length between upper and lower strings when each new block is added; that this difference reaches zero reflects the fact that the constraint is satisfied. Cases (m)–(s) show constraints that cannot be satisfied by strings of any finite length. When the constraints involve more than two blocks there seems in general to be no upper limit on how long a string one may need to consider to tell whether the constraints can be satisfied. Pictures (a), (b), (h) and (j) show the longest minimal strings needed for any of the 4096, 16384, 65536 and 262144 constraints involving blocks with totals of 7, 8, 9 and 10 elements. The general problem of satisfying constraints of the kind shown here is known as the Post Correspondence Problem. Finding the systems on this page required constructing—by computer and otherwise—an immense number of proofs of the impossibility of satisfying particular constraints.

turns out that such a question is almost inevitably undecidable if it is asked about a system that exhibits computational irreducibility.

So what about finite questions?

Such questions can ultimately always be answered by finite computations. But when computational irreducibility is present such computations can be forced to have a certain level of difficulty which sometimes makes them quite intractable.

When one does practical computing one tends to assess the difficulty of a computation by seeing how much time it takes and perhaps how big a program it involves and how much memory it needs.

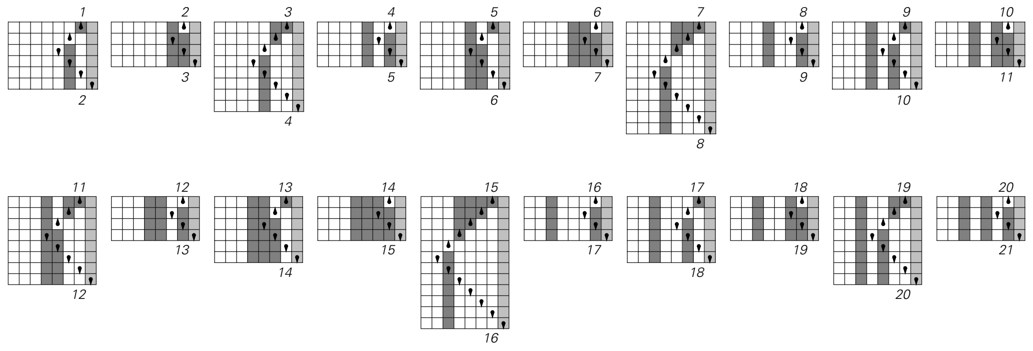
But normally one has no way to tell whether the scheme one has for doing a particular computation is the most efficient possible. And in the past there have certainly been several instances when new algorithms have suddenly allowed all sorts of computations to be done much more efficiently than had ever been thought possible before.

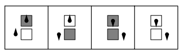
Indeed, despite great efforts in the field of computational complexity theory over the course of several decades almost no firm lower bounds on the difficulty of computations have ever been established. But using the methods of this book it turns out to be possible to begin to get at least a few results.

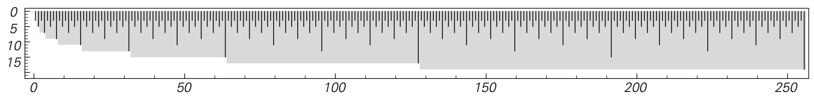
The key is to consider very small programs. For with such programs it becomes realistic to enumerate every single one of a particular kind, and then just to see explicitly which is the most efficient at performing some specific computation.

In the past such an approach would not have seemed sensible, for it was normally assumed that programs small enough to make it work would only ever be able to do rather trivial computations. But what my discoveries have shown is that in fact even very small programs can be quite capable of doing all sorts of sophisticated computations.

As a first example—based on a rather simple computation—the picture at the top of the facing page shows a Turing machine set up to add 1 to any number. The input to the Turing machine is the base 2 digit sequence for the number. The head of the machine starts at the right-hand end of this sequence, and the machine runs until its head first goes further to the right—at which point the machine stops, with







Examples of the behavior of a simple Turing machine that does the computation of adding 1 to a number. The number is given as a base 2 digit sequence; the Turing machine runs until its head hits the gray stripe on the right. The plot shows the number of steps that this takes as a function of the input number *x*. The result turns out to be given by *2 IntegerExponent[x + 1, 2] + 3*, which has a maximum of *2 n + 3*, where *n* is the length of the digit sequence of *x*, or *Floor[Log[2, x]]*. The average for a given length of input does not increase with *n*, and is always precisely 5.

whatever sequence of digits are left behind being taken to be the output of the computation.

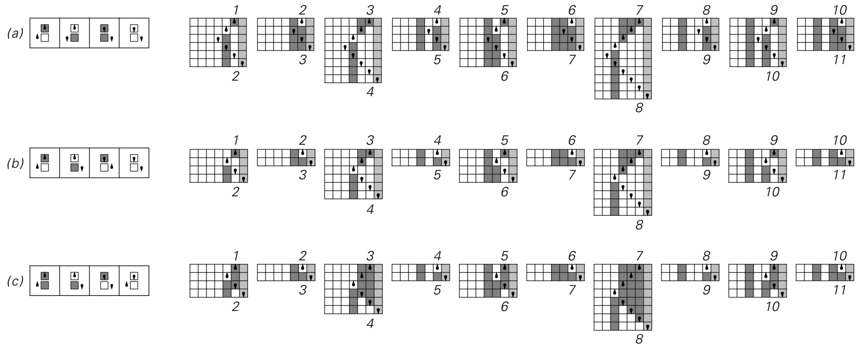
And what the pictures above show is that with this particular machine the number of steps needed to finish the computation varies greatly between different inputs. But if one looks just at the absolute maximum number of steps for any given length of input one finds an exactly linear increase with this length.

So are there other ways to do the same computation in a different number of steps? One can readily enumerate all 4096 possible Turing machines with 2 states and 2 colors. And it turns out that of these exactly 17 perform the computation of adding 1 to a number.

Each of them works in a slightly different way, but all of them follow one of the three schemes shown at the top of the next page—and all of them end up exhibiting the same overall linear increase in number of steps with length of input.

So what about other computations?

It turns out that there are 351 different functions that can be computed by one or more of the 4096 Turing machines with 2 states



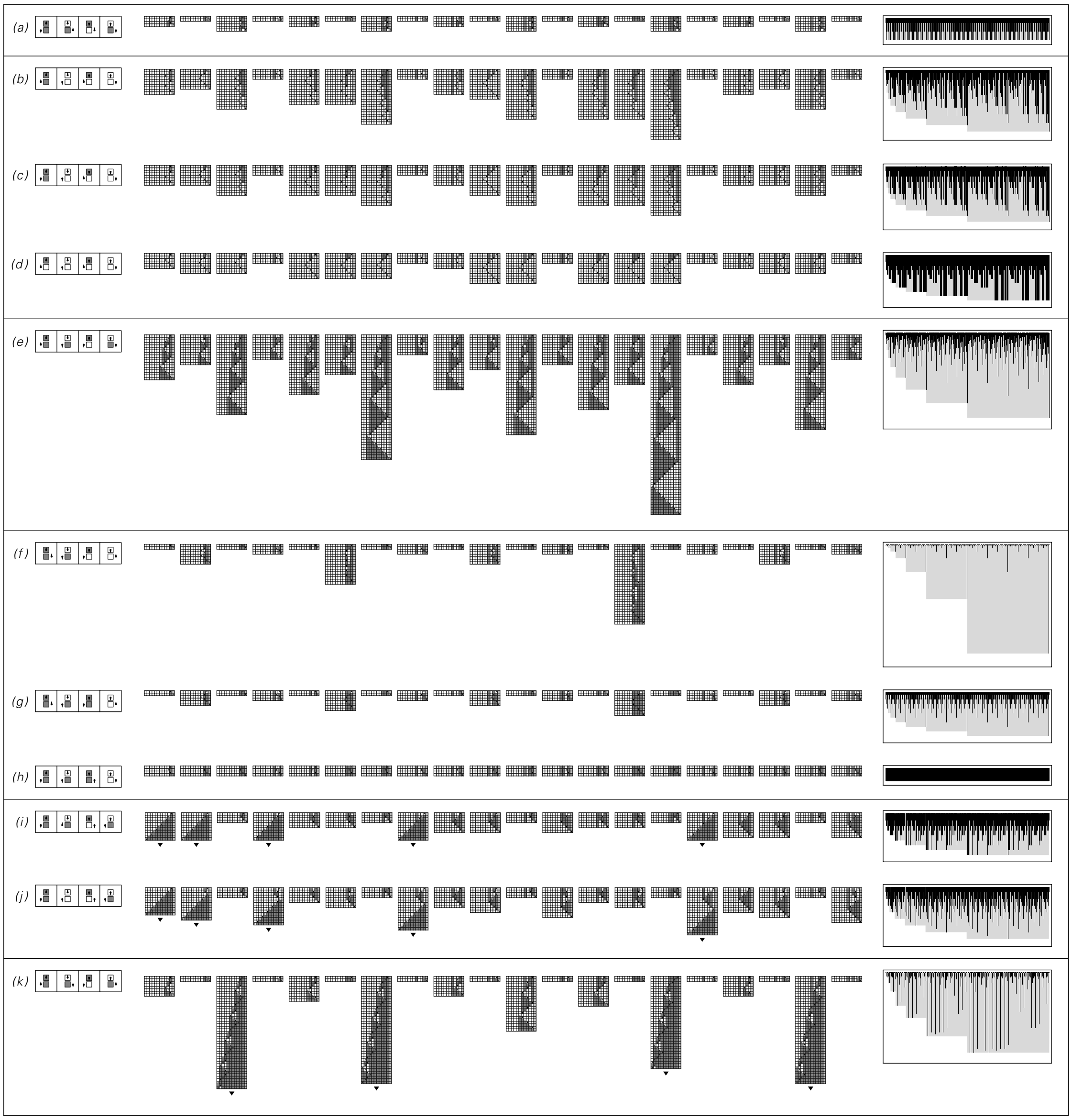
The three schemes for adding 1 to a number that are used by Turing machines with 2 states and 2 colors. All show the same linear growth in maximum number of steps as their size of input increases. This growth can be viewed as a consequence of potentially having to propagate carry digits from one end of the input number to the other. The machines shown are numbered 445, 461 and 1512.

and 2 colors. And as the pictures on the facing page show, different Turing machines can take very different numbers of steps to do the computations they do.

Turing machine (a), for example, always finishes its computation after at most 5 steps, independent of the length of its input. But in most of the other Turing machines shown, the maximum number of steps needed generally increases with the length of the input.

Turing machines (b), (c) and (d) are ones that always compute the same function. But while this means that for a given input each of them yields the same output, the pictures demonstrate that they usually take a different number of steps to do so. Nevertheless, if one looks at the maximum number of steps needed for any given length of input one finds that this still always increases exactly linearly—just as for the Turing machines that add 1 shown at the top of this page.

So are there cases in which there is more rapid growth? Turing machine (e) shows an example in which the maximum number of steps grows like the square of the length of the input. And it turns out that at least among 2-state 2-color Turing machines this is the only one that computes the function it computes—so that at least if one wants to use a program this simple there is no faster way to do the computation.



Examples of computations being done by Turing machines with two states and two colors. Evolution from a succession of initial conditions is shown corresponding to inputs of numbers from 1 to 20. Each block of Turing machines yields the same output for a given input. A computation is taken to be complete when the head of the Turing machine goes further to the right than it was at the beginning. The plots show how many steps this takes for successive inputs with lengths up to 9. The maximum for input of length *n* is (a) *5*, (b) $6n+3$, (c) $4n+3$, (d) $2n+3$, (e) $2n^2+8n+7$, (f) $2^{n+1}-1$ (though the average is $n+2$), (g) $2n+1$, (h) *3*, (i) $2n+1$, (j) $4n-1$, (k) roughly $2.5n^2$. In cases (i), (j) and (k) there are some inputs for which the head goes further and further to the left, and the Turing machine never halts. The machines shown are numbered 3279, 1285, 3333, 261, 1447, 1953, 1969, 3517, 3246, 3374, 1507.

So are there computations that take still longer to do? In Turing machine (f) the maximum number of steps increases exponentially with the length of the input. But unlike in example (e), this Turing machine is not the only one that computes the function it computes. And in fact both (g) and (h) compute the same function—but in a linearly increasing and constant number of steps respectively.

So what about other Turing machines? In general there is no guarantee that a particular Turing machine will ever even complete a computation in a finite number of steps. For as happens with several inputs in examples (i) and (j) the head may end up simply going further and further to the left—and never get to the point on the right that is needed for the computation to be considered complete.

But if one ignores inputs where this happens, then at least in examples (i) and (j) the maximum number of steps still grows in a very systematic linear way with the length of the input.

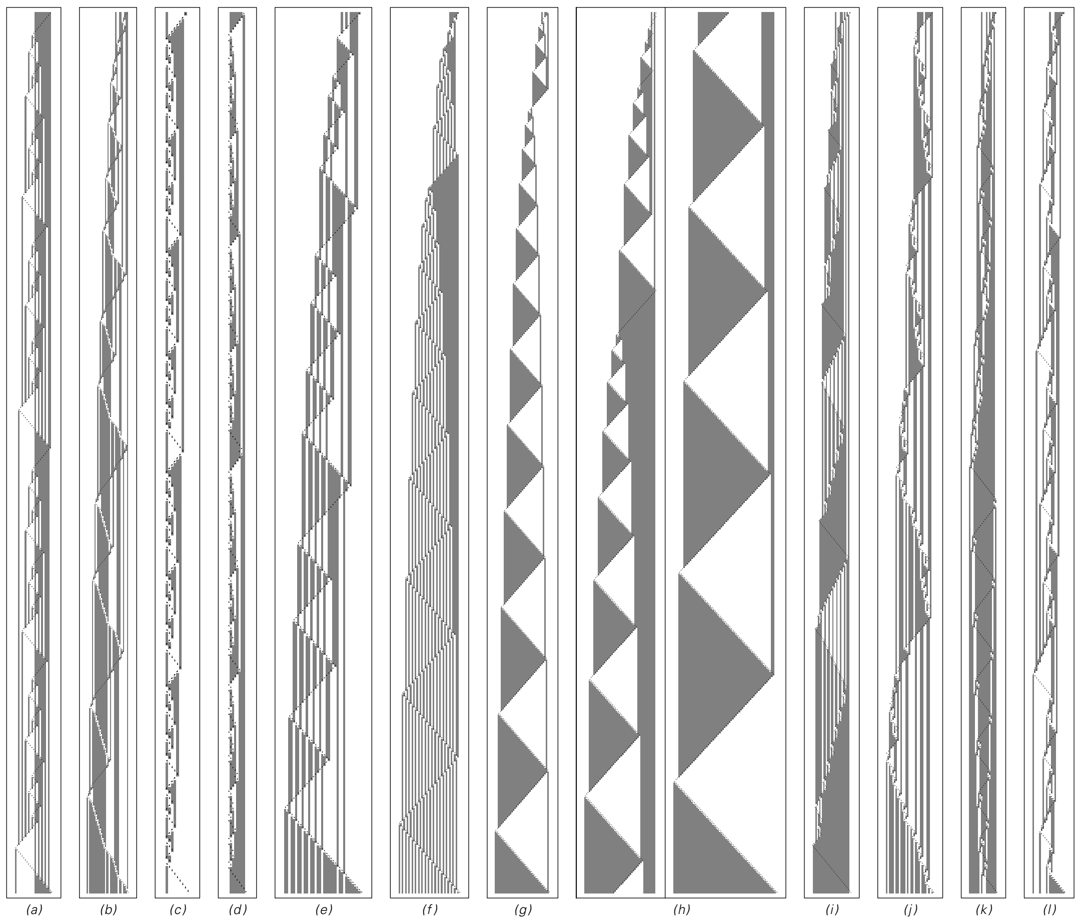
In example (k), however, there is more irregular growth. But once again the maximum number of steps in the end just increases like the square of the length of the input. And indeed if one looks at all 4096 Turing machines with 2 states and 2 colors it turns out that the only rates of growth that one ever sees are linear, square and exponential.

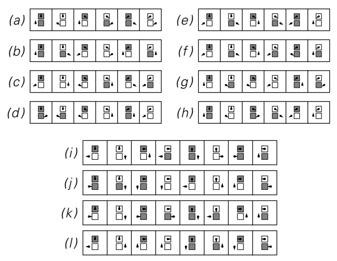
And of the six examples where exponential growth occurs, all of them are like example (f) above—so that there is another 2-state 2-color Turing machine that computes the same function, but without the maximum number of steps increasing at all with input length.

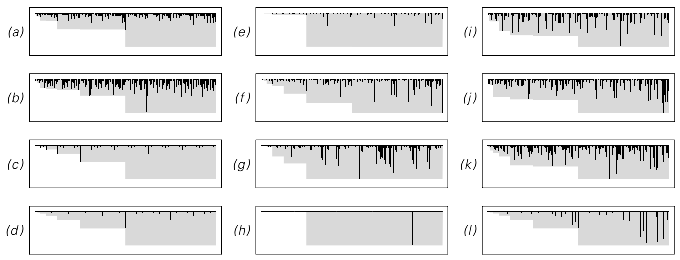
So what happens if one considers more complicated Turing machines? With 3 states and 2 colors there are a total of 2,985,984 possible machines. And it turns out that there are about 33,000 distinct functions that one or more of these machines computes.

Most of the time the fastest machine at computing a given function again exhibits linear or at most quadratic growth. But the facing page shows some cases where instead it exhibits exponential growth.

And indeed in a few cases the growth seems to be even faster. Example (h) is the most extreme among 3-state 2-color Turing machines: with the size 7 input 106 it already takes 1,978,213,883 steps







Examples of Turing machines with 3 and 4 states in which the maximum number of steps before a computation is finished grows at least exponentially with the length of the input. In all cases no Turing machines with the same number of states compute the same functions in fewer steps. In case (h) the number of steps grows so rapidly that only two peaks are seen in the plot. The top row of pictures are all scaled to be exactly the same height, even though the initial conditions cannot be chosen to make the number of steps in each case anything more than roughly the same. The machines have numbers: 582285, 657939, 2018806, 2868668, 2138664, 2139050, 132527, 600720, 3374234978, 1806221583, 1232059922, 3238044559. Cases like (c) and (d) show nested behavior reminiscent of a counter which generates digit sequences of successive integers.

to generate its output, and in general with size n input it may be able to take more than 2^{2^n} steps.

But what if one allows Turing machines with more complicated rules? With 4-state 2-color rules it turns out to be possible to generate the same output as examples (c) and (d) in just a fixed number of steps. But for none of the other 3-state 2-color Turing machines shown do 4-state rules offer any speedup.

Nevertheless, if one looks carefully at examples (a) through (h) each of them shows large regions of either repetitive or nested behavior. And it seems likely that this reflects computational reducibility that should make it possible for sufficiently complicated programs to generate the same output in fewer than exponentially many steps.

But looking at 4-state 2-color Turing machines examples (i) through (l) again appear to exhibit roughly exponential growth. Yet now—much as for the 4-state Turing machines in Chapter 3—the actual behavior seen does not show any obvious computational reducibility.

So this suggests that even though they may be specified by very simple rules there are indeed Turing machine computations that cannot actually be carried out except by spending an amount of computational effort that can increase exponentially with the length of input.

And certainly if one allows no more than 4-state 2-color Turing machines I have been able to establish by explicitly searching all 4 billion or so possible rules that there is absolutely no way to speed up the computations in pictures (i) through (l).

But what about with other kinds of systems?

Once one has a system that is universal it can in principle be made to do any computation. But the question is at what rate. And without special optimization a universal Turing machine will for example typically just operate at some fixed fraction of the speed of any specific Turing machine that it is set up to emulate.

And if one looks at different computers and computer languages practical experience tends to suggest that at least at the level of issues like exponential growth the rate at which a given computation can be done is ultimately rather similar in almost every such system.

But one might imagine that across the much broader range of computational systems that I have considered in this book—and that presumably occur in nature—there could nevertheless still be great differences in the rates at which given computations can be done.

Yet from what we saw in Chapter 11 one suspects that in fact there are not. For in the course of that chapter it became clear that almost all the very varied systems in this book can actually be made to emulate each other in a quite comparable number of steps.

Indeed often we found that it was possible to emulate every step in a particular system by just a fixed sequence of steps in another system. But if the number of elements that can be updated in one step is sufficiently different this tends to become impossible.

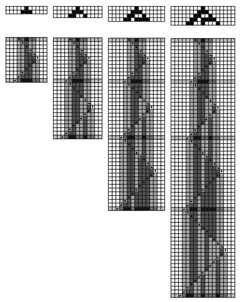
And thus for example the picture on the right shows that it can take t^2 steps for a Turing machine that updates just one cell at each step to build up the same pattern as a one-dimensional cellular automaton builds up in t steps by updating every cell in parallel.

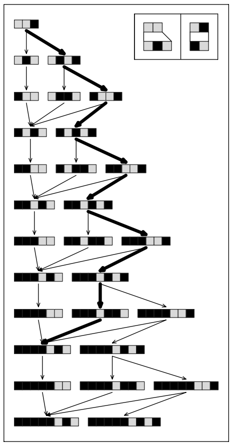
And in d dimensions it is common for it to take, say, t^{d+1} steps for one system to emulate t steps of evolution of another.

But can it take an exponential number of steps? Certainly if one has a substitution system that yields exponentially many elements then to reproduce all these elements with an ordinary Turing machine will take exponentially many steps. And similarly if one has a multiway system that yields exponentially many strings then to reproduce all these will again take exponentially many steps.

But what if one asks only about some limited feature of the output—say whether some particular string appears after t steps of evolution of the multiway system? Given a specific path like the one in the picture on the right it takes an ordinary Turing machine not much more than t steps to test whether the path yields the desired string.

But how long can it take for a Turing machine to find out whether any path in the multiway system manages to produce the string? If the Turing machine in effect had to examine each of the perhaps exponentially many paths in turn then this could take exponentially many steps. But the celebrated P=NP question in computational complexity theory asks whether in general there is some


To emulate t steps in the evolution of the cellular automaton takes the Turing machine $2t^2 + 5t - 6$ steps.


A Turing machine can quickly test the highlighted path but could take exponentially long to test all paths.

way to get such an answer in a number of steps that increases not exponentially but only like a power.

And although it has never been established for certain it seems by now likely that in most meaningful senses there is not. So what this implies is that to answer questions about the t-step behavior of a multiway system can take any ordinary Turing machine a number of steps that increases faster than any power of t.

So how common is this kind of phenomenon? One can view asking about possible outcomes in a multiway system as like asking about possible ways to satisfy a constraint. And certainly a great many practical problems can be formulated in terms of constraints.

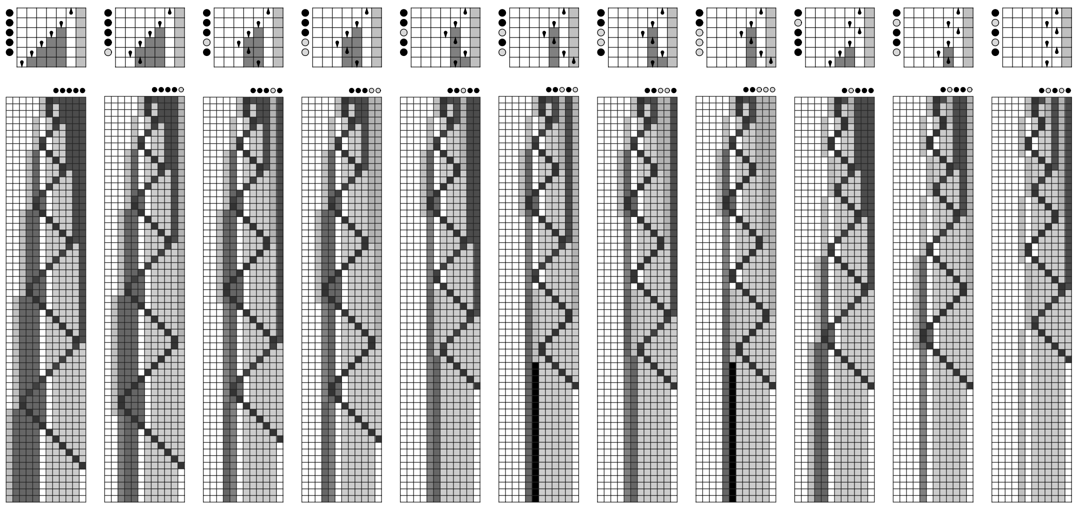
But how do such problems compare to each other? The Principle of Computational Equivalence suggests that those that seem difficult should somehow tend to be equivalent. And indeed it turns out that over the course of the past few decades a rather large number of such problems have in fact all been found to be so-called NP-complete.

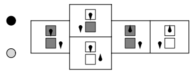
What this means is that these problems exhibit a kind of analog of universality which makes it possible with less than exponential effort to translate any instance of any one of them into an instance of any other. So as an example the picture on the facing page shows how one type of problem about a so-called non-deterministic Turing machine can be translated to a different type of problem about a cellular automaton.

Much like a multiway system, a non-deterministic Turing machine has rules that allow multiple choices to be made at each step, leading to multiple possible paths of evolution. And an example of an NP-complete problem is then whether any of these paths satisfy the constraint that, say, after a particular number of steps, the head of the Turing machine has ever gone further to the right than it starts.

The top row in the picture on the facing page shows the first few of the exponentially many possible paths obtained by making successive sequences of choices in a particular non-deterministic Turing machine. And in the example shown, one sees that for two of these paths the head goes to the right, so that the overall constraint is satisfied.

So what about the cellular automaton below in the picture? Given a particular initial condition its evolution is completely





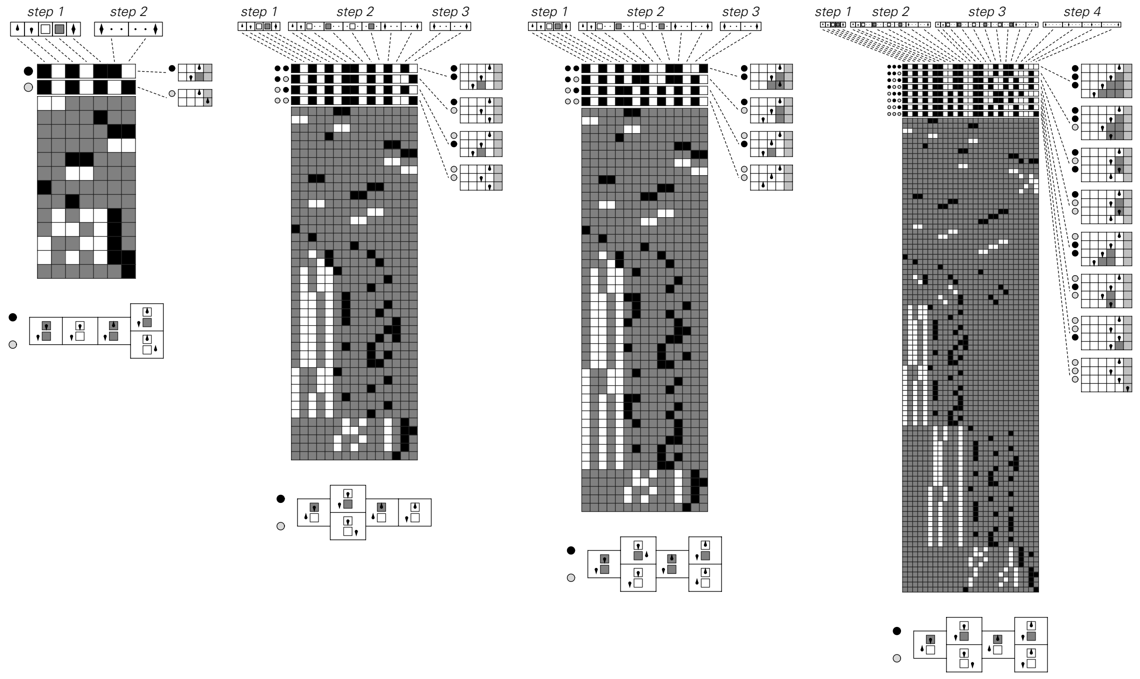
Translation between an NP-complete problem about non-deterministic Turing machines and about cellular automata. The top row shows how a particular non-deterministic Turing machine behaves with successive sequences of choices for rules to apply. The bottom row shows how a cellular automaton can be made to emulate this behavior when given a succession of different initial conditions. The cellular automaton is set up to produce a vertical black stripe if the head of the Turing machine ever goes further to the right than it starts—as it does in cases 6 and 8. The left part of each cellular automaton configuration emulates the actual evolution of the Turing machine; a specification of which rules should be applied at each step is progressively fetched from the right and delivered to the position of the head. Given particular initial conditions for the Turing machine the problem of whether the head ever goes further to the right than it starts is thus equivalent to the problem of whether the cellular automaton ever produces a vertical black stripe given particular initial conditions on its left. The cellular automaton takes $2t^2+t$ steps to emulate t steps of evolution in the Turing machine. It involves a total of 19 colors.

deterministic. But what the picture shows is that with successive initial conditions it emulates each possible path in the non-deterministic Turing machine.

And so what this means is that the problem of finding whether initial conditions exist that make the cellular automaton produce a certain outcome is equivalent to the non-deterministic Turing machine problem above—and is therefore in general NP-complete.

So what about other kinds of problems?

The picture on the next page shows the equivalence between the classic problem of satisfiability and the non-deterministic Turing machine problem at the top of this page. In satisfiability what one does is to start with a collection of rows of black, white and gray squares. And then what one asks is whether any sequence of just black and



Translation between the NP-complete problem of halting in a non-deterministic Turing machine and the classic NP-complete problem of satisfiability. In satisfiability one sets up a collection of rows of black, white and gray squares, then asks whether there exists any sequence of black and white squares that satisfies the constraint that on every row the color of at least one square agrees with the color of the corresponding square in the sequence. Each row can be viewed as a term in a conjunctive normal form Boolean expression, with each column corresponding to a different variable. When a given square on a particular row is black or white it indicates that a variable or its negation appear in that term. The translation from the Turing machine problem is achieved by representing the behavior of the Turing machine by saying which of a sequence of elementary statements are true about it at each step: whether the head is in one state or another, whether the cell under the head is black or white, and whether the head is at each of the possible positions it can be in. The Boolean expression then gives constraints on which of these statements can simultaneously be true. In the first two pictures, for example, the first row corresponds to the constraint that on the first step of Turing machine evolution, the head cannot simultaneously be in an up and a down state. About the first half of the terms in each Boolean expression correspond to similar general constraints about the operation of Turing machines. There are then a few terms that specify the particular initial conditions used here, followed by terms that give the rule for the Turing machine that is used. The very last term makes the statement that the Turing machine halts. As the pictures indicate, each possible path of evolution for the Turing machine then corresponds to a possible assignment of truth values to the variables associated with each elementary statement. And if there is any path that leads the Turing machine to halt the Boolean expression will be satisfiable. This is the case in the first and fourth examples shown, but not in the other two. In general, it is possible to represent t steps in the evolution of a non-deterministic Turing machine by a Boolean expression with at most t^3 terms in t^2 variables. A version of the translation shown here was what launched the study of NP completeness in the early 1970s.

white squares exists that satisfies the constraint that on every row there is at least one square whose color agrees with the color of the corresponding square in the sequence.

To see the equivalence to questions about Turing machines one imagines breaking the description of the behavior of a Turing machine into a sequence of elementary statements: whether the head is in a particular state on a particular step, whether a certain cell has a particular color, and so on. The underlying rules for the Turing machine then define constraints on which sequences of such statements can be true. And in the picture on the facing page almost every row of black, white and gray squares corresponds to one such constraint.

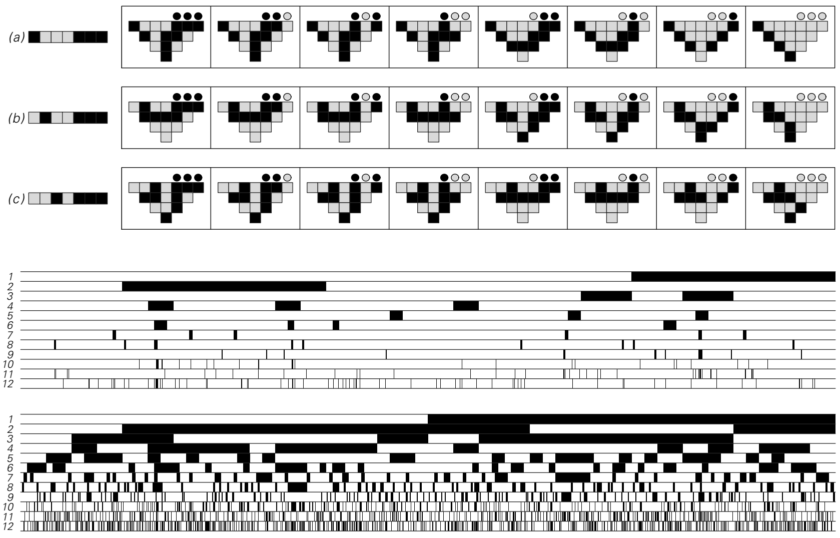
The last row, however, represents the further constraint that the head of the Turing machine must at some point go further to the right than it starts. And this means that to ask whether there is any sequence in the satisfiability problem that obeys all the constraints is equivalent to finding the answer to the Turing machine problem described above.

Starting from satisfiability it is possible to show that all sorts of well-known computational problems in discrete mathematics are NP-complete. And in addition almost any undecidable problem that involves simple constraints—such as the correspondence problem on page 757—turns out to be NP-complete if restricted to finite cases.

In studying the phenomenon of NP completeness what has mostly been done in the past is to try to construct particular instances of rather general problems that exhibit equivalence to other problems. But almost always what is actually constructed is quite complicated—and certainly not something one would expect to occur at all often.

Yet on the basis of intuition from the Principle of Computational Equivalence I strongly suspect that in most cases there are already quite simple instances of general NP-complete problems that are just as difficult as any NP-complete problem. And so, for example, I suspect that it does not take a cellular automaton nearly as complicated as the one on page 767 for it to be an NP-complete problem to determine whether initial conditions exist that lead to particular behavior.

Indeed, my expectation is that asking about possible outcomes of t steps of evolution will already be NP-complete even for the rule 30 cellular automaton, as illustrated below.



Example of a simple problem that I suspect is NP-complete. The problem is to determine whether right-hand cells in the initial conditions for rule 30 can be filled in so as to produce a vertical black stripe of a certain height at the bottom of the center column formed after t steps of evolution. The pictures at the top show that in case (a) stripes up to height 3 can be produced, in case (b) up to height 2, and in case (c) only up to height 1. The pictures at the bottom indicate in black for which of the 2^{t+1} successive left-hand sequences of $t + 1$ cells it is impossible to get stripes of respectively heights 1 and 2. The apparent randomness of these patterns reflects the likely difficulty of the problem. The problem is related to issues of rule 30 cryptanalysis discussed on page 603.

Just as with the Turing machines of pages 761 and 763 there will be a certain density of cases where the problem is fairly easy to solve. But it seems likely that as one increases t, no ordinary Turing machine or cellular automaton will ever be able to guarantee to solve the problem in a number of steps that grows only like some power of t.

Yet even so, there could still in principle exist in nature some other kind of system that would be able to do this. And for example one might imagine that this would be possible if one were able to use exponentially small components. But almost all the evidence we have suggests that in our actual universe there are limits on the sizes and densities of components that we can ever expect to manipulate.

In present-day physics the standard mathematical formalism of quantum mechanics is often interpreted as suggesting that quantum systems work like multiway systems, potentially following many paths in parallel. And indeed within the usual formalism one can construct quantum computers that may be able to solve at least a few specific problems exponentially faster than ordinary Turing machines.

But particularly after my discoveries in Chapter 9, I strongly suspect that even if this is formally the case, it will still not turn out to be a true representation of ultimate physical reality, but will instead just be found to reflect various idealizations made in the models used so far.

And so in the end it seems likely that there really can in some fundamental sense be an almost exponential difference in the amount of computational effort needed to find the behavior of a system with given particular initial conditions, and to solve the inverse problem of determining which if any initial conditions yield particular behavior.

In fact, my suspicion is that such a difference will exist in almost any system whose behavior seems to us complex. And among other things this then implies many fundamental limits on the processes of perception and analysis that we discussed in Chapter 10.

Such limits can ultimately be viewed as being consequences of the phenomenon of computational irreducibility. But a much more direct consequence is one that we have discussed before: that even given a particular initial condition it can require an irreducible amount of computational work to find the outcome after a given number of steps of evolution.

One can specify the number of steps t that one wants by giving the sequence of digits in t. And for systems with sufficiently simple behavior—say repetitive or nested—the pictures on page 744 indicate that one can typically determine the outcome with an amount of effort that is essentially proportional to the length of this digit sequence.

But the point is that when computational irreducibility is present, one may in effect explicitly have to follow each of the t steps of evolution—again requiring exponentially more computational work.