computers Ltd.
what they really can't do
david harel
faculty of mathematics and computer science
the weizmann institute of science,
rehovot, israel

OXFORD UNIVERSITY PRESS
termination

An algorithm that completes its work but produces the wrong output is just one kind of worry. When it comes to the need for our algorithms and programs to do what we expect, there is something else we have to worry about — an algorithm that doesn’t terminate at all, but, rather, keeps running on its input forever. This is clearly an error too. We don’t want our programs to loop forever, i.e. to get stuck in an infinite non-terminating computation. The execution of a program on any one of its legal inputs should terminate within a finite amount of time, and its output must be the correct one.

Often, we can see rather easily how to make sure that our algorithm terminates. As a simple example, suppose we are devising an algorithm to check the primality of a number. We might have decided, rather stupidly, to base our approach directly on the definition of a prime number, verbatim. That is, in an attempt to find a factor (a divider) of the input number, we instruct our algorithm to try to divide it by each and every number from 2 on, in turn, with no bounds set. This rather silly algorithm would clearly loop indefinitely when run on a number that was indeed prime. Fortunately, as we have seen, there are obvious ways to bound the number of candidate divisors that need to be tested, and these guarantee termination.

Contrast this example with Problem 8 of the list given earlier, in which we don’t seem to be that lucky: a solution algorithm is required to give one answer if the input program $P$ behaves in some particular way, and another answer if it doesn’t. There appears to be no way for us to make the decision without actually running $P$, a process that can itself fail to terminate. Worse, it seems that we have to run $P$ on infinitely many inputs, not just on one or two.

We shall return to this example in the next chapter.

The message of this chapter is simple and clear. Computers are not omnipotent. They can’t do everything. Far from it.

We shall discuss problems that cannot be solved by any computer, past, present or future, running any program that can be devised, even if given unlimited amounts of time and even if endowed with unlimited storage space and other resources it might need. We still require, of course, that algorithms and programs terminate for each legal input in a finite amount of time, but we allow that time to be unlimited. The algorithm can take as long as it wishes, and can use whatever resources it asks for in the process, but it must eventually stop and produce the right output. Nevertheless, even under these generous conditions, we shall see interesting and important problems for which there simply are no algorithms, and it doesn’t matter how smart we are, or how sophisticated and powerful our computers, our software, our programming languages and our algorithmic methods. Figure 2.1 is intended to set the stage for what is to come.
finite problems are solvable

The first thing to notice is that any algorithmic problem with a finite set of inputs is solvable. That is, if all it will ever have to deal with is a finite, limited set of possible inputs, there is an algorithm to solve it. Suppose we have a decision problem whose sole legal inputs are the items input1, input2, ..., inputK. Then there is an algorithm that contains a table providing each of the K inputs with the appropriate output. It might look like this:

1. read the input;
2. if it is input1 then output ‘Yes’ and stop;
3. if it is input2 then output ‘Yes’ and stop;
4. if it is input3 then output ‘No’ and stop;
   ...
   ...
K + 1. if it is inputK then output ‘Yes’ and stop.

This works, of course, because it is possible to ‘hard-wire’ into an algorithm the entire algorithmic problem in all its glory by tabulating all the (finitely many) input–output pairs. It might be difficult to figure out the tabulation itself, that is, to construct such a table-driven algorithm, but we are not interested in this meta-difficulty here. For the present discussion, it suffices to know that for finitary problems solutions exist, and we ignore the issue of how to find them.

In contrast, algorithmic problems that have infinite sets of legal inputs are the really interesting ones. For these, we don’t even know if there exists a finite algorithm to tackle the infinitely many different cases, and it is those that will keep us busy.
The tiling problem

Our first example of a noncomputable problem involves covering large areas using colored tiles. A tile is defined to be 1 × 1 square, divided into four by the diagonals, each quarter colored with some color. We assume that the tiles have fixed orientation and cannot be rotated.¹

An input is a finite number of tile descriptions, collectively denoted by T. Each tile type in T is defined by its four colors in order. The problem asks whether any finite area, of any size (with integer dimensions, of course), can be covered using only tiles of the kinds described in T, but adhering to the following restriction: the colors on the touching edges of any two adjacent tiles must be identical. An unlimited number of tiles of each type is available, but in T there is only a finite, limited number of types of tiles.

Think of tiling your living room. The input T is a description of the various types of tile available, and the color-matching restriction reflects a rule enforced by your interior designer for esthetic reasons. The question we would like answered ahead of time is this: can any room, of any size, be tiled using only the available tile types, without violating the color-matching rule?

This algorithmic problem and its variants are commonly known as tiling problems, and are sometimes called domino problems because of the domino-like restriction on touching edges.

¹ After you finish reading this section you might want to try to convince yourself that this assumption is, in fact, necessary. We should add, however, that it is necessary only in the version we discuss here. It is easy to define a variant of the tiling problem, where instead of colors having to be identical, they have to match up in pairs (e.g. red against blue, green against orange, etc.). In such a version, the rotations-forbidden constraint is redundant, and the bad news is exactly the same.

In way of illustration, consider Fig. 2.2, which shows three tile types and a 5 × 5 tiling. The reader will have no difficulty verifying that the pattern in the figure can be extended in all directions, to yield a tiling of any sized room whatsoever. As can also be seen, this tiled portion uses only the three available types, and it adheres to the color-matching rule. However, exchange the bottom colors of tiles (2) and (3) as in Fig. 2.3, and the situation changes dramatically. It is now quite easy to show that even very small rooms cannot be tiled at all, since no matter how you attempt to lay down the tiles you will be forced very quickly into situations with mismatched colors. Figure 2.3 illustrates this. An algorithm for the tiling problem, thus should answer ‘Yes’ to the input consisting of the three tile types of Fig. 2.2, and ‘No’ to those of Fig. 2.3.

Can we somehow mechanize or ‘algorithmicize’ the reasoning employed in generating these answers?

![Fig. 2.2. Tile types that can tile any room, of any size.](image)
An algorithmic problem that admits no solution is termed non-computable; if it is a decision problem, as is the case here and in most of the examples that follow, it is termed undecidable. The tiling problem is thus undecidable: there is no way we can construct an algorithm, to be run on a computer, any computer, regardless of the amount of time and memory space required, that will be able to distinguish between tile types that can tile all areas and those that cannot.\(^3\) This problem, then, lies above the line of Fig. 2.1.

**do we really mean it?**

Are we really claiming that this problem has no algorithmic solution at all? How can we justify drawing the grand line of Fig. 2.1? What gives us the right to use such all-inclusive terms like non-computable and undecidable? 'Maybe', the reader might claim, 'you can't solve it, on your computer, with your ancient system software, mediocre programming language and old-fashioned algorithmic methods and tricks. But not me. I have an amazingly powerful supercomputer. I am smart and I work with incredibly sophisticated programming languages and state-of-the-art methodologies; I can surely do it! ...'.

---


\(^3\) There is a subtly different version of the tiling problem. We asked whether the tile set \(T\) can be used to tile any finite area, of any size. Instead, we could have asked whether \(T\) can be used to tile the entire infinite two-dimensional plane. Interestingly, these two problems are completely equivalent: a 'Yes' for the first version is a 'Yes' for the second version too, and a 'No' for the first is a 'No' for the second too. One direction of this equivalence (if we can tile the entire infinite plane then we can tile any finite area) is trivial, but the argument that establishes the other direction is quite delicate. You are encouraged to try to find it. Thus, the infinite-plane version is also undecidable.
Well, no, dear reader, you cannot. When we label a problem non-computable or undecidable, we really and truly mean it. You can't solve it, and neither can anyone else, no matter how rich or patient or smart.

Still, the claim does sound very strange if we don't restrict the basic operations. Surely, if anything is allowed, the following two-step procedure solves the tiling problem:

1. If the tile types in the input set $T$ can tile rooms of any size, output 'Yes' and stop;
2. Otherwise, output 'No' and stop.

So, is this not a solution? It consists of but two basic operations, and thus terminates in a finite amount of time, as it should. And surely it will always produce the correct output too.

Well, we must be a little more careful. Suppose we choose a fixed programming language $Lang$ as the medium for expressing algorithms, and a fixed computer $Comp$ as the machine on which they are to run (with the understanding that $Comp$ can grant any amount of time, additional storage space, and any other tangible resource requested by a program during a computation). Suppose that we also agree that, for the moment, whenever we talk about an algorithm we really mean a program written in $Lang$ and running on $Comp$. With this setup, when we say 'no algorithm exists' we really mean that no program can be written in the specific language $Lang$ for the specific computer $Comp$. This sounds a little less wild: it is conceivable that some problems will indeed be unsolvable if one is limited to working with a specific hardware/software framework (sometimes called a model of computation). In fact, a reasonable way to dismiss the above two-line 'solution' to the tiling problem is to convince its proposers that there is no way to implement the test in line 1 using their chosen language $Lang$ running on their machine $Comp$.

'OK', those proposing the two-line solution might say, 'so we can't solve the problem on this particular computer and with this particular language, but we could solve it had we a more powerful computer and a more sophisticated language.' Isn't the issue merely a question of coming up with the right algorithmic idea, designing the corresponding software and running it on a sufficiently powerful piece of hardware?4

No, it isn't. Not at all.

Actually, the situation is far more striking. It is not only that each model of computation can be shown to be fallible, by exhibiting some special problem it cannot solve, but there are fixed problems (the tiling problem is one of them) that are bad news for each and every model. That is, these problems are noncomputable regardless of the model chosen. They are thus inherently noncomputable. Worse, we computer scientists believe that this applies not only to currently known models, but to any effectively implementable language, running on any computer of any type, size or shape, now or at any time in the future. And this is what we mean when we say that a problem is noncomputable.

Amazingly, all that is needed in order to establish that a problem is noncomputable in this all-embracing sense is to show that it can't be solved within an extremely simple-looking model of computation, which we now set out to describe. That it actually can't be solved in any known model whatsoever, including the most

---

4 This is probably what the TIME magazine interviewee quoted in the Preamble had in mind.
powerful computers invented and those that will be invented in the future, will follow from this modest-looking fact.

**elementary computing devices**

Let us see how simple we can make a general computing model.

The first thing to notice is that any item of data used by an algorithm can be viewed as a string of symbols. An integer is but a string of digits, and a fractional number is a string of digits with a decimal point. A word in English is a string of letters, and an entire text is really just a string of symbols consisting of letters, blanks, and punctuation marks. More complicated objects, such as lists, tables, city-connection networks, graphs, pictures, video sequences, and even whole databases, can also be encoded this way, by using special delimiting symbols to signify new items, line breaks, file borderlines, and so on.

The number of different symbols used in all such encodings is actually finite, and can always be fixed ahead of time. This is the ingenuity of a standard numbering scheme, such as the decimal system: we do not need infinitely many symbols, one for each number — 10 symbols suffice to encode them all.\(^5\) The same obviously applies to words, texts, and pictures, since only a finite number of letters, punctuation marks, color codes, and special symbols are used in writing or in rendering images for computerization. Consequently, in principle, we can write any data of interest along a one-dimensional tape, perhaps a long one, which consists of a sequence of squares, each containing a single symbol taken from some finite alphabet. In order to allow for additional 'scrap paper'

---

\(^5\) The binary system uses just two, 0 and 1.

---

sometimes we can't do it 37
to contemplate and possibly change the symbol it finds on the current square. It then 'changes gear', and hops over to a neighboring square for its next step. That's all.

Here is an informal description of a Turing machine that has been programmed to add two decimal numbers $X$ and $Y$. (You can skip to the next section if you feel you might be bored by a rather tedious description of how a primitive-looking machine adds numbers.) The input numbers are given on the tape, separated by the symbol $+$, and the rest of the tape contains blanks, which are denoted here by $\#$. See Fig. 2.5, which shows, from top to bottom, some snapshots of the tape as the computation proceeds.

Initially, the head is positioned at the leftmost symbol of the first number $X$ — in this case, it is the 7. The machine then travels to the rightmost digit of $X$ — the 6 — one square at a time, without making changes, until it reaches the separating symbol $+$, and then moves one square to its left. It then erases this digit, i.e. replaces it with a blank, while 'remembering' the erased digit as its state; it will need 10 different states for this, one for each digit. The machine then travels over to the rightmost digit of $Y$ — the 9 — and erases it too, entering a state that remembers the sum digit of the two numbers, and whether or not there is a carry. This state depends only on the current digit and the memorized one, and hence 20 different states are needed — one for each of the possible combinations of the 10 sum digits and the carry/no-carry indication. The machine then moves to the left of what remains of $X$ and writes the sum digit down — a 5 in this case — having prepared a new separating symbol, say, an exclamation mark, '!'. This situation is illustrated in the second line of the figure.

The next step is similar, but involves the currently rightmost digits (which were second from the right in the original numbers — here the 3 and the 1), and takes the carry into account, if there is one. The new sum digit — here 5 because of the carry — is written down to the left of the previous one, and the process continues. Of course, any one of the two input numbers might run out of digits before the other, in which case, after adding the carry (if there is one) to the remaining portion of the larger number, that portion is
just copied down on the left, tediously, digit by digit. Finally, a second exclamation mark is written down on the extreme left, to identify the machine's output as consisting of the portion of the tape enclosed by the two exclamation marks, and the machine halts.

Phew ...

the church-turing thesis

This example is a little surprising. Turing machines have only finitely many states, that is, a finite 'brain', and the only thing they can do is to rewrite symbols on a linear tape one at a time. Nevertheless, they can be programmed to add numbers of any size and shape. The task can be frustrating and thankless, and the machine's method of execution can be painfully slow and simple-minded (try to describe a Turing machine to multiply numbers or to compute the average of N salaries), yet it gets the job done.

With this in mind, let us forget about tedious, frustration and efficiency for the moment, and ask ourselves what indeed can be done with 'Turing machines', for whatever cost and no matter how painstakingly? Which algorithmic problems can be solved by an appropriately programmed Turing machine?

The answer to this is not a little surprising, but very surprising indeed: Turing machines are capable of solving any effectively solvable algorithmic problem! Put differently, any algorithmic problem for which we can find an algorithm that can be programmed in some programming language, any language, running on some computer, any computer — even one that has not been built yet (but, in principle, can be built), and even one that requires unbounded amounts of time and memory space for ever-larger inputs — is also solvable by a Turing machine!

This statement is one version of the so-called Church-Turing thesis, after Alonzo Church and Turing, who arrived at it independently in the 1930s, following the work of Kurt Gödel on the incompleteness of mathematics.7

It is important to realize that the CT thesis, as we shall call it (both for Church-Turing and for computability theory), is a thesis, not a theorem, since part of it cannot be proved mathematically. The reason for this is that one of the concepts it involves is informal and imprecise, namely, effective solvability, or effective computability. The thesis equates the mathematically precise notion of 'solvable by a Turing machine' with the informal, intuitive notion of 'effectively solvable', which alludes to all real computers and all programming languages, past, present, and future. It thus sounds more like a wild speculation than what it really is: a deep and far-reaching statement, put forward by two of the most respected pioneers of the science of computing. And, as we shall see, while its futuristic facet cannot be proved until the future materializes, its past and present facts have been proved.

Turing machines are a little like typewriters. A typewriter is also a very primitive kind of machine. All it enables us to do is to type sequences of symbols on blank paper. Yet despite this, any

---

computability is robust

Why should we believe the CT thesis, when even its proponents admit that the yet-to-be-seen parts of it can’t be proven? What evidence is there for it, and how does that evidence fare in an age of incredible day-to-day advances in both hardware and software?

Let us go back to the 1930s. At that time, several researchers were busy devising various algorithmic models, with the goal of trying to capture the slippery and elusive notion of effective computability, i.e. the ability to compute mechanically or electronically. Long before the first actual computers were invented, Turing suggested his limited-looking machines and Church devised a simple mathematical formalism of functions called the lambda calculus.

Around the same time, Emil Post defined certain symbol-manipulating mechanisms called production systems, and Stephen Kleene defined a class of mathematical objects called recursive functions. All these people tried, and succeeded, in showing that their models were able to solve many algorithmic problems for which ‘effectively executable’ algorithms were known. Actually, collectively, they also succeeded in showing that their formalisms were all equivalent in terms of the class of problems they could solve. Other people have since proposed numerous different models for the universal algorithmic device. Some of these models underly real computers, and some are purely mathematical in nature. But the crucial fact is that they have all been proven to be computationally equivalent; the class of algorithmic problems they can solve is the same. And this fact is still true today, even for the most powerful models conceived.8

Thus, the strongest, most powerful computer you know, coupled with the richest, most sophisticated programming language it supports, cannot do any more than can be done with a

---

simple laptop and a very modest language. Or, for that matter, any more than can be done by the ultimate in computational simplicity — the ever so primitive Turing machine model.\(^9\)

Noncomputable (or undecidable) problems, such as the tiling problem, are solvable on neither, and computable (or decidable) problems, such as sorting words or testing a number for primality, are solvable on both. All this, mind you, on condition that running time and memory space are not an issue: there must be as much of those available as is needed.

This means that the class of computable, effectively solvable, or decidable algorithmic problems is, in fact, extremely robust. It is invariant under changes in the computer model, the operating system, the programming language, the software development methodology, etc. Proponents of a particular computer architec-

\(^9\) Another extremely primitive model of computation that is nevertheless as powerful as Turing machines and is therefore also of universal power and subject to the CT thesis, is that of counter programs, or counter machines. A counter program is a sequence of simple instructions on non-negative integers that can assign 0 to a variable \((X \leftarrow 0)\), and can increase or decrease a variable by one \((X \leftarrow Y + 1\) and \(X \leftarrow Y - 1 \)). It can also branch conditionally, based on the zero-ness of a variable \((\text{if } X = 0 \text{ goto } G, \text{ where } G \text{ labels some other instruction in the sequence})\). Surprisingly, merely incrementing and decrementing integers by 1 and testing values against 0 can be used to do anything any computer can do. Turing machines and counter programs are dual models in the following interesting sense: they both have access to unlimited amounts of memory, but in different ways. With Turing machines, the number of memory items (the tape's squares) is unlimited, but the amount of information in each is finite and is bounded ahead of time (one symbol from a fixed and finite alphabet). With counter programs it is the other way around: there are only finitely many variables in a given program, but each can contain an arbitrarily large number as its value, thus encoding a potentially unlimited amount of information.

structure or programming discipline must find reasons other than raw solving power to justify their recommendations, since anything doable with one is also doable with the other, and all are equivalent to the primitive machines of Turing or the various formalisms of Church, Post, Kleene, and others.

That so many people, working with such a diversity of tools and concepts, captured the very same notion (long before any actual computers were built, we should add!), is evidence for the profundity of that notion. That they were all after the same intuitive concept and ended up with different-looking, but equivalent, models, is justification for equating that intuitive notion with those precise models. Hence the CT thesis.

Thus, if we set efficiency aside for now, not caring about how much time or space an algorithm actually requires, but simply giving it anything it wants, the line drawn between the computable and the noncomputable in Fig. 2.1 is fully justified. Moreover, as we proceed in our discussions, we can safely allude to some favorite computer Comp and programming language \(\text{Lang}\) as the model on which algorithmic problems are to be solved, just as we did earlier on a temporary basis, because it makes no difference! Nevertheless, it is intellectually satisfying to be able to point to a most simple model — Turing machines — that is as powerful as anything of its kind.\(^{10}\)

\(^{10}\) Another advantage of knowing that simple-looking models like Turing machines or counter programs are universally powerful, is that they are better suited for establishing bad news. As stated earlier, to prove that a problem is undecidable, for example, all one has to show is that it cannot be solved using Turing machines. That it cannot be solved on any model will then follow from the CT thesis.
domino snakes

Let us return for a moment to the tiling problem. Some people react to its undecidability by saying: 'well, obviously the problem is undecidable, since a single input can give rise to a potentially infinite number of cases to check; and there is no way you can get an infinite job done by an algorithm that has to terminate after finitely many steps.' Indeed, a single input (that is, a single set \( T \) of tile types) apparently requires all rooms of all sizes to be checked (or, equivalently, a single infinite 'room'), and there appears to be no way to set a bound on the number of cases that have to be considered.

This unboundedness-implies-undecidability hypothesis is unbased, and can be very misleading. In fact, it is often simply wrong. To drive the point home, here is a similar tiling problem, whose status violates this hypothesis in a surprising way. As before, the input contains a finite set \( T \) of tile types, but here it also contains the coordinates of two points on the infinite plane, \( V \) and \( W \). The problem doesn't talk about tiling whole rooms, but, in the spirit of real domino games, it asks if it is possible to connect \( V \) and \( W \) by a 'domino snake' consisting of tiles from \( T \), and with the same color-matching restriction: every two adjacent edges must have identical colors (see Fig. 2.6). Note that a snake originating at \( V \) might twist and turn erratically, reaching unboundedly distant points before converging to \( W \). So, to decide whether or not there is such a snake, we might have to check ever-larger portions of the infinite plane (the infinitely large room) — perhaps all of it — before we either find such a snake or conclude that none exists. Hence, this problem also seems to require an infinite search, prompting us to presume that it too is undecidable.

Curiously, the decidability of the domino snake problem depends on the portion of the plane available for placing tiles, and in a very counter-intuitive fashion. If snakes are allowed to go anywhere (that is, if the allowed portion is the entire infinite plane), the problem is decidable; but if the allowed area is limited to, say, the upper half of the plane, the problem becomes undecidable! That is, if snakes can run around anywhere, with no limitations, there is an algorithm to decide whether there is a snake going from \( V \) to \( W \), but if we do limit its habitat, there is no algorithm. Surprising, right?

The latter case is 'more bounded' than the former, and therefore should be 'more decidable'. The facts, however, are quite the other way around.\(^{11}\)

\(^{11}\) If the available portion of the infinite plane is finite, the problem is trivially decidable, since only finitely many possible snakes can be positioned in a given finite area, and an algorithm can be easily designed.
program verification

In Chapter 1 we discussed the need for algorithms and programs to be correct. Establishing the fact that a candidate program indeed solves the algorithmic problem you are working on is no easy feat. So it is tempting to ask whether computers can do this for us. We would really like an automatic verifier, a piece of software whose input consists of (the description of) an algorithmic problem and (the text of) an algorithm, or program. We would like the verifier to determine algorithmically whether the given program solves the given problem. In other words, we want a 'Yes' if for each of the input problem's legal inputs the input program, had we run it on that legal input, would terminate with the correct output, and a 'No' if for even a single legal input the input program would either fail to terminate or would terminate with the wrong output (see Fig. 2.7). The verifier must be able to do this for every choice of algorithmic problem and for every choice of a candidate program.\(^\text{12}\)

As a particularly pressing example, wouldn't it be nice if someone were to establish a start-up company and construct a general-purpose Y2K verifier? We could have then subjected any piece of software to the verifier, and found out whether what it would have done on 1 January 2000 is the same as what it did on 31 December 1999. Is this possible?

\(^{12}\)Here too, it is convenient to fix a computer model and programming language in advance. Actually, since in this case programs are part of the input, we must adopt a language with well-defined syntax and semantics, so as to be able to hand the program verifier a real, tangible object as input. By the CT thesis, however, such a choice does not detract from the generality of what we have to say here.
Well, the general verification problem is undecidable, as is the special case of verifying compliance with the year 2000. A candidate verifier might work nicely for many of its inputs; it might be able to verify certain kinds of programs against certain limited kinds of specifications, but as a general verifier it is bound to fail. There will always be algorithms or programs that such a verifier will not be able to verify. We can thus forget about a computerized solution to the Y2K problem or any other such sweeping attempt at establishing the correctness of software by computer.

In contrast to tiling and snake domino problems, which you might dismiss as toy problems of no practical value, program verification is an extremely important computer-related task, coming from the real world. The fact that it is unsolvable dashes our hope for a software system that would make sure that our computers do what we want them to.

the halting problem

It turns out that the news is as bad already for a lot less than the full correctness of programs. We cannot even decide whether a program merely terminates on its inputs. Worse, it is not even decidable whether it terminates on one specific input! These issues of termination, or halting, are at the heart of Problem 8 in the list given in Chapter 1, and they deserve special attention.

Consider the following algorithm (call it A):

1. while \( X \neq 1 \) do the following: set \( X \leftarrow X - 2 \);
2. stop.

In words, the algorithm A repeatedly decreases its input number \( X \) by 2 until it becomes equal to 1. Assuming that the legal inputs consist of the positive integers 1, 2, 3, etc., it is quite obvious that

A halts precisely for the odd numbers. An even number will be decreased repeatedly by 2, will 'miss' the 1, running forever through 0, -2, -4, -6, etc. Hence the problem of deciding whether a legal input will cause this particular algorithm to halt is trivial: all we have to do is to check whether the input is odd or even, and answer accordingly.

Here is a slightly more complicated algorithm, B:

1. while \( X \neq 1 \) do the following:
   1.1. if \( X \) is even, set \( X \leftarrow X/2 \);
   1.2. otherwise (i.e. \( X \) is odd), set \( X \leftarrow 3X + 1 \);
2. stop.

This algorithm repeatedly halves its input if it is even, but increases it more than threefold if it is odd. And it too halts if and when it reaches the value 1. For example, if B is run on the number 7, the sequence of values is: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, as you can easily check, following which execution halts. Actually, if we try running algorithm B on an arbitrary positive integer, even using a powerful computer, we will find that it either terminates, or progresses through an erratic-looking sequence, reaching surprisingly high values, and fluctuating unpredictably. In the latter case, one gives up after a while, having not observed either termination or a periodic sequence of values (which, of course, would have indicated that the computation will not terminate). Indeed, over the years, B has been tested on numerous inputs, and on large and fast computers. On the one hand, no periodicity has been observed, and no-one has been able to come up with an input for which B can be proved not to terminate. On the other hand, no-one has been able to prove that B terminates for all positive integers (although people believe it does). Which of these is the case is actually a difficult unresolved question in the branch
of mathematics known as number theory, and it has been open for some 60 years.\textsuperscript{13}

So, here we are, with two algorithms, the uninteresting $A$ and the far more interesting $B$. While some mathematicians in the field of number theory would probably give a lot to find out whether $B$ halts on all its inputs, $B$ is still but one specific algorithm. In the study of algorithmics we are not that interested in the halting behavior of specific programs, even tantalizing ones like $B$. Rather, we are interested in the generic problem of determining the halting behavior of an unknown given algorithm or program. This general decision problem is called the halting problem.

As input, the halting problem is fed the text of a legal program $A$ in our chosen programming language $Lang$, and a potential input $X$, which is really nothing more than a string of symbols. The problem asks whether or not $A$ would have terminated had we run it on the input $X$ (see Figure 2.8).

The halting problem, just like the more demanding verification problem, cannot be solved by algorithmic means; it is undecidable. This means that there is no way to tell, in general, and in a finite amount of time, whether the execution of a given program will terminate on a given input.$^\text{14}$

\textsuperscript{13} J. C. Lagarias (1985). 'The $3x + 1$ Problem and its Generalizations', \textit{Amer. Math. Monthly} 92, 3–23. This is perhaps the simplest-to-describe open problem in mathematics. To understand it you need to know nothing except basic arithmetic symbols. Is it or is it not the case that any positive integer eventually reaches 1 if it is repeatedly halved when even and tripled and increased by one when odd?

\textsuperscript{14} This is due to Turing. See his 1936 paper referenced in footnote 6 of this chapter. See also G. Rozenberg and A. Salomaa (1994). \textit{Cornerstones of Undecidability}. Prentice Hall, New York, NY.

Fig. 2.8. The halting problem.

It is tempting to try to solve the problem by a simulation algorithm that simply mimics running the program $A$ on the input $X$ and waits to see what happens. The point is that if and when execution terminates we can justifiably stop and conclude that the answer is 'Yes'; had we indeed run $A$ on $X$ it would have terminated. The difficulty is in deciding when to stop waiting and say 'No'. We cannot simply give up after a long wait and conclude that since the simulation has not yet terminated it never will. Perhaps if we had left it to run just a little longer — maybe one more microsecond would do it — it \textit{would} have terminated. Simulating the given program's behavior on the given input, therefore, does not do the job, and, as stated, nothing can do the job, since the problem is undecidable.

\textbf{nothing about computation can be computed!}

This phenomenon is actually much deeper and more devastating. There is a remarkable result, called \textit{Rice's theorem}, that shows that
not only can we not verify programs or determine their halting status, but we can't really figure out anything about them. No algorithm can decide any nontrivial property of computations. More precisely, let us say we are interested in deciding some property of programs, which is (i) true of some programs but not of others, and (ii) insensitive to the syntax of the program, that is, it is a property of the underlying algorithm and not of the particular form it takes in a programming language. For example, we might want to know whether a program runs in less than a particular amount of time, whether it ever outputs a 'yes', whether it always produces numbers, whether it is equivalent to some other program, etc., etc.

What Rice's theorem tells us is that no such properties of programs can be decided. They are all undecidable. We can really forget about being able to reason automatically about programs. Virtually nothing about computation is computable!

some problems are even worse

As it turns out, three of the undecidable problems mentioned so far — the tiling problem, the domino snake problem on the half-plane, and the halting problem — are computationally equivalent. This is not a simple notion, since obviously these problems look very different: tiling rooms and determining whether programs terminate, for example, don't seem to have anything to do with each other. In fact, they have everything to do with each other.

What exactly do we mean by two undecidable problems being computationally equivalent? Well, the key notion is inter-reducibility. Each one of the two problems is reducible to the other, in the sense that one can be decided with the aid of an imaginary solution, or oracle, for the other. Thus, if we had an algorithm to decide, in general, whether programs halt on inputs (we can't have a real algorithm for this because the problem is undecidable, but say we had a hypothetical one, the oracle) we could use it to decide whether tiles can tile living rooms. And perhaps more surprisingly, vice versa: if we could decide about tiling living rooms, we could decide about computer programs halting. Imagine that!

Having an imaginary solution is like having an immortal oracle who gives you answers to certain questions for free. Thus, if you had an oracle who could answer tiling questions whenever asked, you could solve the halting problem.

A rather striking addendum to the equivalence between these noncomputable problems is that some problems — program verification for example — are even less decidable. What on earth can we mean by this? What can be worse for an algorithmic problem than to have no solution at all? Here too, the key is reducibility: the halting problem can be reduced to program verification, meaning that an imaginary solution to the latter can be used to solve the former. The converse, however, is not true. Even with a free (imaginary) oracle for the halting problem, or the tiling problem, or the half-plane snake domino problem (or even with oracles for all of these) we could not verify programs. The verification problem is thus harder than the halting problem; it is less decidable, so to speak.
This oracle-based way of comparing undecidable problems, making some of them 'better' than others, induces a classification of algorithmic problems into levels of undecidability, or levels of noncomputability. Layers upon layers of problems exist, coming with worse and worse news! The three equivalent problems we mentioned, halting, tiling, and half-plane snakes, turn out to be on one of the lowest such levels. You might say that they are *almost* decidable. Sadly, however, many problems reside far higher up in the hierarchies of ever more terrible news, so that they are much less decidable than the ones lower down.

One interesting level is sometimes termed high noncomputability, or high undecidability, and it deserves a zone of its own in the sphere of algorithmic problems (see Fig. 2.9). Highly noncomputable problems are much, much worse than the 'ordinarily' noncomputable ones we have discussed. In fact, they are *infinitely* worse. Even an infinite lineup of increasingly more sophisticated oracles wouldn't suffice to solve them. Thus, above the almost computable, or almost decidable problems (tiling, halting, and their friends) there are infinitely many different problems, each more difficult than the ones lower down, and each one not computable even with the aid of oracles for all those below it. The problems we have termed highly undecidable are even worse than all those.

---

17 S. C. Kleene (1943). 'Recursive predicates and Quantifiers', *Trans. Amer. Math. Soc.* 53, 41–73. The high undecidability we discuss here is called the $\Sigma_1^\forall$ level, in technical terminology. A simple example of a highly undecidable problem is the following variant of the tiling problem (we use the version of tiling that asks whether the set $T$ of tile types can tile the entire infinite plane, rather than the one that asks about tiling all finite areas). The new variant adds but a small requirement: we want to know whether $T$ can tile the infinite plane, but in such a way that the tiling contains infinitely many copies of the first tile listed in $T$; i.e., a recurrence of the designated tile type. We want a 'Yes' if there is a $T$-tiling of the plane containing a recurrence of this particular tile, and a 'No' if no such tiling exists. Note that the answer must be 'No' even if there are legal tilings of the whole plane using the tiles in $T$, but none of them has the first tile of $T$ recurring infinitely often. This extra requirement doesn't look as though it should make a big difference, because if you can tile the infinite plane using a finite set of tile types, then *some* of the types must occur in the tiling infinitely often. The difference is that here we want a *specific* tile to recur. Despite the apparent similarity, this recurring dominoes problem, as it is called, is highly undecidable. It is not decidable even with the use of imaginary solutions to the infinitely many other problems residing on lower levels of the undecidability hierarchies. See D. Harel (1986). 'Effective Transformations on Infinite Trees, with Applications to High Undecidability, Dominoes, and Fairness', *J. Assoc. Comput. Mach.* 33, 224–48. But don't think that this is as bad as it can get. Some problems are even worse than the highly undecidable ones, but we will ease off now, and let it go at that.

---

Fig. 2.9. The sphere of algorithmic problems: Version II.
In summary, we have learned that the world of algorithmic/computational problems is divided into the computable, or decidable, vs. the noncomputable, or undecidable, and that among themselves the problems in the latter class exhibit various degrees of hardness. We have also seen that these facts are extremely robust and lasting: the dividing lines of Fig. 2.9 are mathematically precise and firmly defined, and are insensitive to variations in computational models, languages, methodologies, hardware or software.

So our hopes for computer omnipotence are shattered. We now know that not all algorithmic problems are solvable by computers, even with unlimited access to resources like time and memory space.

Can we finish our story here? Isn’t this the bad news alluded to in the Preamble? What else can go wrong?

chapter 3

sometimes we can’t afford to do it

The fact that some tasks cannot be computerized is bad enough already. But we are not done yet. Let us now concentrate on the ones that can.

Say we are asked to construct a bridge over a river. The bridge could be ‘incorrect’, it might not be wide enough for the required lanes, it might not be strong enough to carry rush-hour traffic, or it might not reach the other side at all! However, even a ‘correct’ design may be unacceptable. It might call for too large a workforce, or too many materials or components. It might also require far too much time to bring to completion. In other words, although it will result in a good bridge, a design might be too expensive.

The field of algorithmics is susceptible to similar concerns. Even if a problem is computable, or decidable, and a correct solution algorithm is found, that algorithm might be far too costly in its use of resources, and hence impractical. The term ‘impractical’ sounds mild, but it’s not: we shall discuss problems that require such