Chapter 3

A brief introduction to the theory of computation

It might surprise younger readers that there could even be questions that are *undecidable*, problems for which there can be no standardized procedure, ever, that provides an answer!

In fact, undecidability is ubiquitous: The generic mathematical subject is combinatorial— small repeating units interacting in highly restrained ways. One might think of tiles, but also group elements, connected by generators and relators; or of formal symbols, combined into well-formed logical expressions or equations. Even analysis, richly continuous, in many ways has a discrete and combinatorial nature, reflected in the very formation of the expressions we use in our work.

The theme of this book, more than a discussion of tilings, really, is that combinatorial structures are inherently, generically inscrutable, in a particular technical sense:

Given a particular class of object (say, "planar tiles") and a question (say, "does this tile admit a tiling?") we can ask, is there an algorithm, a procedure, to determine, upon being presented with an object in the class, after a finite period of time, the answer to our question?

That is, is the question *decidable*? If no such algorithm exists, we say the problem is *undecidable*. It is a wholly remarkable (even astounding!) fact that there are undecidable problems. It is even more incredible that these are in sense, generic![1]

[1]One little aside must be made: If the class of objects is *finite* every problem is in fact decidable. We may not know what the algorithm is, but it does exist: just look up the answer in a table. So for example, it is decidable whether a particular *given* tile admits a

When presented with a combinatorial problem, we might wonder whether or not it is decidable. If we find ourselves baffled, if there appears no correlation between the structure of the object and the outcome to the problem, we might apply **Conway's Presumption**:

[[**If a lot can happen, everything will happen.**]]

In other words, the betting man will gamble on undecidability. (Right or wrong, at least one is not likely to be contradicted!)

The "Halting Problem" is undecidable

Turing's **Halting Problem** [**?**] is the touchstone undecidable problem: *does a given computational procedure halt with an answer or run forever?*

We don't really need to go deeply into Turing's specific idea of what computation is; there are many fine sources available. In fact, it suffices to approach the whole topic rather naïvely: in essence, an **algorithm** is simply some sort of formalized procedure, precise instructions to be followed exactly. (We'll use the words "program", "procedure", "computation" and "algorithm" interchangeably.)

Unfortunately, most mathematics education, at least before our first abstract courses, consists of memorizing just these sorts of detailed instructions– for multiplication, adding fractions, plotting lines or conics, taking derivatives, et cetera– so we are all quite familiar with the notion.

These instructions assume we are capable only of keeping track of where we are and making simple notes to which we may refer as we go along. Now

an algorithm may be badly designed: for example

```
Start with 0;
   repeatedly add 1 until you reach -10 and then stop.
```

It is pretty clear that this algorithm will never halt. We could consider algorithms that allow some input, such as an arbitrary integer n:

tiling: the does exist an algorithm to answer the question. It is either the one that halts with an answer `yes` or halts with an answer `no`. Unfortunately, we might not know which is the correct algorithm!

```
Given an integer n;
   repeatedly add 1 until you reach -10 and then stop.
```

Obviously, this algorithm halts if we begin with $n < -10$ and does not halt if $n \geq -10$. But, in general, whether or not a given algorithm halts can be a quite subtle question!

It is known that the following algorithm halts for input $n < 13 \cdot 2^{58}$ and counting [8], but despite quite a lot of attention [9], no one is sure if it will always halt, on every input:

Given a counting number n,
   while $n \neq 1$
      let $n = \begin{cases} 3n+1 & \text{if } n \text{ is odd} \\ n/2 & \text{if } n \text{ is even} \end{cases}$

So for example, if we input $n = 5$, the algorithm next sets $n = 3 \cdot 5 + 1 = 16$ then $8, 4, 2$, and finally $n = 1$, at which point the algorithm stops.[2] If you doubt the trickiness of this algorithm, you might investigate its behavior beginning with $n = 27$.

Now procedures themselves can be encoded as integers. The simplest illustration of this lies in the hard drive of your computer. All of your programs, photos, data, music— everything— is encoded in one tremendous number. (On the computer I am now typing on, this number is something like $2^{\ 800\text{ billion}}$, as I have a 100-gigabyte hard drive, holding about 800 billion bits.)

The specific encoding depends entirely on our computing environment: how do we write down and interpret instructions to carry out our procedures? Different computer operating systems will encode procedures differently, but in a much deeper way, there are many different "models of computation"— different ways to conceive of what *is* computation, as well as encode it. We don't need to be too concerned with this here— suffice it to say that models all are functionally as powerful as one another.[3][[forward ref]] Always, we are assuming that we are working in some *specific, fixed model*; it is perfectly fine to imagine programming in your favorite language, on your favorite machine, but with no memory or time constraints.

[2] If we allowed it to continue, the algorithm would then loop forever: $3 \cdot 1 + 1 = 4$, then $2, 1, 4, \ldots$

[3] In essence, this assertion is the "Church-Turing thesis", the very foundation of the Theory of Computation. [[good source??]]

Typically, only special numbers will encode valid, working programs. The important thing, though, is that we can *enumerate* (list, by some procedure) *all* procedures, just by listing the counting numbers in turn and discarding any that don't encode a valid procedure. (Obviously, we will never be able to complete such a list— but any given procedure will eventually show up, if we wait long enough.)

Let $\mathcal{P}_n$ be the nth procedure in our enumeration. For simplicity, assume each $\mathcal{P}_n$ has one value it takes as input. Now we may formally ask the **Halting Problem**:

Does a given procedure halt on a given input ?

More importantly, is there a general technique to tell? *Is there a procedure $\mathcal{H}$ than can examine, as input, a given procedure $\mathcal{P}$ and counting number m, and determine whether $\mathcal{P}$ will, with input m, halt in a finite period of time?*

Is there a procedure $\mathcal{H}$ that can always settle the Halting Problem for a given $\mathcal{P}$ and m? *Is the Halting Problem* **decidable***?*

The obvious $\mathcal{H}$ to try is to run $\mathcal{P}$ on input m and see what happens. If $\mathcal{P}$ does eventually halt, sooner or later we will find out, though this may take a very long while. But what can we conclude if $\mathcal{P}$ has not halted after, say, one million years? Perhaps $\mathcal{P}$ will halt in a few more minutes, or a trillion years— *or not at all.* In other words, the obvious test, running the procedure to find out, is not guaranteed to give us a yes or no answer.

If we were more clever could we find an $\mathcal{H}$? Using a simple an ingenious trick, A. Turing proved that no such $\mathcal{H}$ exists, that the Halting Problem is undecidable!

Suppose that in fact there were a procedure $\mathcal{H}$ that could take as input n and, after a finite number of steps, determine whether the nth procedure, $\mathcal{P}_n$, halts on input n. We can modify $\mathcal{H}$, obtaining a new procedure $\mathcal{H}'$:

$\mathcal{H}'$ takes as input an integer n, and then carries out $\mathcal{H}$ to find out whether or not $\mathcal{P}_n$ does halt on input n. If it does, then $\mathcal{H}'$ goes into an infinite loop and if it does not, then $\mathcal{H}'$ halts.

Now then, $\mathcal{H}'$ is a procedure, and so appears somewhere in our initial enumeration— for some h, $\mathcal{H}' = \mathcal{P}_h$. But now consider, dear reader, what happens when $\mathcal{H}'$ is presented with h as input!

If, on this input, $\mathcal{H}'$ were to halt, it could have only been because $\mathcal{P}_h$ (which is of course $\mathcal{H}'$ itself) failed to halt on input h, an impossibility.

On the other hand, if $\mathcal{H}'$ fails to halt on input h, then $\mathcal{P}_h$— that is, $\mathcal{H}'$— halts on input h. That isn't possible either.

We have a contradiction— $\mathcal{H}'$ can neither halt nor not halt!— and there could have been no $\mathcal{H}$ in the first place! There is *no* algorithm to decide the Halting Problem.

But how special is this? Are only a few problems undecidable, in only a few parts of mathematics? To the contrary! One can argue that the generic mathematical problem is undecidable! (Though, naturally, the problems mathematicians actually solve tend not to be.)

Computable Functions

A **computable function** is simply a function, on the counting numbers say, for which we can compute its values. Pretty much any function on the counting numbers that you've every seen is computable— all this means is that you have an explicit description of how to work out its values.

It is not hard to give some computable functions that have astoundingly large growth. For example, take

$$f(n) = n^{n^{\cdot^{\cdot^{\cdot^{n}}}}}, \text{ where the tower of exponents is } n \text{ high}$$

Things start out slow: $f(1) = 1$ and $f(2) = 2^2 = 4$ but $f(3) = 3^{27}$ and $f(4) = 4^{4^{256}} = 4^{\text{a 154-digit number}}$. You might amuse yourself trying to understand the size of $f(5)$, $f(100)$ or $f(f(100))$.

I find the following function particularly appealing:

$$h(1) := 1, \quad h(n) := n \underbrace{!\ldots!}_{h(n-1)}$$

$h(2) = 2! = 2$, not terribly exciting to be sure, but $h(3) = 3!! = 6! = 720$ and then $h(4) = 4\underbrace{!\ldots!}_{720} = 24\underbrace{!\ldots!}_{719} = 620,448,401,733,239,439,360,000\underbrace{!\ldots!}_{718} =$??. I shudder to think of $h(5)$.

But then consider $h(h(n))$, or worse $\underbrace{h \circ \ldots \circ h}_{h(n)}$, etc. One may amuse oneself all day in this way. These functions are all computable, for the simple reason that we have managed to describe them explicitly. For any of these, it is no great trick to write a short program that, in principle, could calculate its values.

And yet, and yet there are functions that are unbounded by *any* computable function, functions that have *no computable bound!*

Take, for example, the **Halting Function** $T : \{1, 2, \ldots\} \to \{-1, 0, 1 \ldots\}$, defined by

$$T(n) = \begin{cases} \text{N} & \text{if procedure } \mathcal{P}_n \text{ halts on input } n \text{ after } N \text{ steps} \\ \text{-1} & \text{if procedure } \mathcal{P}_n \text{ does not halt on input } n \end{cases}$$

Not only is there no way to compute the values of this function, T cannot be bounded by computable function! (On the other hand, we could say T is "semicomputable": there is a simple procedure that can decide in finite time whether $T(n) \neq -1$: just run the nth program. The trouble is that this procedure won't ever tell you if the answer *is* -1.)

Suppose there were a computable function b so that $b(n) > T(n)$ for all n. Then we would have a simple procedure for deciding whether the procedure $\mathcal{P}_n$ halts on input n: calculate $b(n)$ and run $\mathcal{P}_n$; if it hasn't halted by the time we've executed $b(n)$ steps, then we're in the clear— the procedure will never halt. But we know the Halting Problem is undecidable; there can't be an algorithm to decide this— so there can't have been a computable bound on T.

Think for a moment what this means! If we list out procedures, say in the order of their length, we're going to have some pretty lame ones for quite a long while. We cannot expect to see anything interesting until we get quite far along. By this time, $f(n), h(n)$ and the rest will have popped off beyond the mathematical stratosphere.

Somehow, incredibly, T, on occasion, has to beat out every one of these, and we can show, has to do so infinitely often.

The complexity of an integer

It is also worth considering the **program-size complexity** of a given integer. Every integer can be computed by a program that runs, and then halts, with output N: one such program is N lines long: begin with 0 and then add one, add one, add one ,... add one. Fixing a specific integer N, we can obviously do better with

```
Calculate N
   Set i = 0; while i < N
      Increment i
Output i.
```

This is much shorter but still takes about $\log N$ symbols to write out—because N itself has to be written out as part of the program; this takes $\log_b N$ symbols in whatever base b we are using to represent integers. Since b is fixed (we have a fixed system for writing things out), $\log_b N = \frac{1}{\log b}\log N$, a constant multiple of $\log N$. Of course the specific number of symbols depends on our language and numbering system, but up to a constant, the length is about right.

How much better can we do? We can obtain some truly staggering numbers with some very simple procedures: let us calculate Graham's Number[**?**], famously the largest number ever used in a mathematical theorem [**?**].

Graham's number is so huge we must use some special symbols even to describe it– the Knuth arrow notation. Just as multiplication is iterated addition, and exponentiation $\uparrow$ is iterated multiplication, the operator $\uparrow\uparrow$ is iterated exponentiation, $\uparrow\uparrow\uparrow$ is iterated $\uparrow\uparrow$'ing, and so on. The function $f(n)$ in Section 3 is merely $f(n) = n\uparrow\uparrow n$. You might work out a few of these: $3\uparrow\uparrow\uparrow 3 = 3\uparrow\uparrow 3\uparrow\uparrow 3 = 3\uparrow\uparrow(3\uparrow 3\uparrow 3) = 3\uparrow\uparrow(3^{27}) = \underbrace{3\uparrow 3\ldots 3\uparrow 3}_{3^{27}}$, a tower of exponents 3^{27} tall! And this is just the beginning![4]

This function $u(a,b,n)$ eventually returns $a\underbrace{\uparrow\ldots\uparrow}_{n}b$ (after a very very long while):

`Calculate` $u(a,b,n)$
`Given counting numbers` a,b,n

$$u(a,b,n) = \begin{cases} u(a,u(a,b-1,n),n-1) & \texttt{for } b,n>1 \\ a & b=1 \\ ab & n=0 \end{cases}$$

But we have barely begun to construct Graham's number! Let $g_1 := 3\uparrow\uparrow\uparrow\uparrow 3$ (an incomprehensibly huge number already). Then $g_2 := 3\underbrace{\uparrow\ldots\uparrow}_{g_1}3$ (beyond incomprehensible) and so on: $g_i := 3\underbrace{\uparrow\ldots\uparrow}_{g_{i-1}}3$. Graham's number is (merely) g_{64}:

`Calculate Graham's Number`
`Set` $i = 1, N = u(3,3,4)$

[4]Amusingly, just as $2+2 = 2\times 2 = 2\uparrow 2 = 4$, no matter how many $\uparrow$'s we use, $2\uparrow\ldots\uparrow 2 = 4$ as well.

```
While i < 64
   Set N = u(3,3,N) and increment i
Output N
```

All in all, the full algorithm took a total of just a few dozen characters to write out, and one could expect that we might do better with adjustments to the language we used. Graham's number has *low* program-size complexity.

Fixing a model of computation, we can ask, what is the length $F(n)$ of the shortest program that calculates a given integer n? We have an upper bound: $F(n)$ is no longer than the number of digits in n, in whatever base we are using. And in some cases, $F(n)$ can be miniscule, compared with n.

But amusingly, on average, $F(n)$ is a constant multiple of $\log n$, for the very simple reason that there are only about n programs of length $\log n$! Fixing a language and number base k, the generic integer has no shorter representation than its expansion base k. The generic integer is fundamentally complicated to describe!

Turing machines

Let us return to the notion of "procedure" or "algorithm". What really is a "computation"?

In 1936, Alan Turing proposed a simple model of computation which has stood the test of time. Many other mathematical systems have been shown to be equivalently powerful (and simpler to exploit) but Turing's model remains among the most intuitively clear.

How do you, the reader, carry out a computation? You follow some specified, finite list of instructions (which you might have memorized, or you might be reading), and have some capacity for keeping track of information, such as a limitless pad of paper.

At each stage, we might be read off of the notepad, follow an instruction, perhaps write something new, and find the next instruction. Turing's insight was that this is pretty much the whole story.

Whatever we can accomplish by this means, a "Turing machine" can do just as well. Imagine first a mechanical insect that can walk about the notepad, its instructions hard-wired into its brain. At each step, the insect is in a certain "state", prepared to carry out a particular act, such as moving a little ways, reading, erasing or writing a symbol, or changing over into another state. This is exactly what we ourselves do when we carry out a

mechanical procedure, such as an algebraic manipulation, or long division.

Our states are thoughts such as "I am now carrying a 1" or "I am adding these two numbers, getting ready to write the answer down" or "I am moving my pen over to the edge of the paper, prepared to write a 7 when I get there".

With a little imagination, we can do away with our notepad and make do with a long tape: we arrange the rows of the notepad into a line, perhaps placing marks for the end of each row. When our mechanical insect needs more rows, it can mark off another bit of the tape; if it needs to extend a row, it can mechanically scoot the other rows over to make more room.

We have reduced the original vision of a person, calculating on a pad of paper, to a model of a simple machine, with a "head" that is in one of a finite number of states, that reads a symbol on the tape; depending on what the head is reading and its state, it may erase the symbol and write a different one, step to the right or to the left, and change into a different state. We also include a special "halt" state: if the machine ever reaches this state, it stops and the computation is ended.

Sometimes we might require that a machine start on a blank tape, or sometimes we will allow a machine to accept input, by starting on a tape with some symbols already written on it.

This simple vision of computation is as powerful as any other, but in practice, Turing machines are slow and inefficient. Their great advantage is in their simplicity.

Here is a simple example, which merely "adds" two numbers, written as strings of tick-marks on the tape. The machine has five states A, B, C, D and E, plus the halt state H. We specify how the machine acts by the following table:

	A	B	C	D	E
0	$0RA$	$1RC$	$0LD$	$0LD$	$0RH$
1	$1RB$	$1RB$	$1RC$	$0LE$	$1LE$

That is, if the machine is in state B, reading a 0, it writes a 1, moves one step to the right, and enters state C. Here is a portion of a run of the machine, starting on a specially prepared tape.

Here is the initial run of the machine:

[[Figure]]

You might follow this to its conclusion– the machine will stop when it reaches its halt state H.

This example is not particularly inspiring, but remember, as we described above, every procedure can be encoded as the action of a Turing machine. In practice, it's not terribly difficult to construct a machine to carry out a particular task, but usually [[the machine will be pretty ugly]].

Busy Beaver Candidates

On the other hand, some small machines can behave in the most amazing ways. Consider the behavior of this remarkable machine [6]

	A	B	C	D	E
0	$1RB$	$1RC$	$1RD$	$1LA$	$1RH$
1	$1LC$	$1RB$	$0LE$	$1LD$	$0LA$

If it begins on a blanked-out tape, marked with all 0's, it will run for a whopping $47,176,870$ steps, printing out 4098 1's on the tape before halting[**?**, **?**, **?**].

It is not so interesting that such a machine might print out 1's forever— so what? On the other hand, it is *truly incredible* that a simple machine can print *exactly* this many and just then come crashingly to a halt.

In 1962, Tibor Radó asked, for each given n, what is the greatest number of steps that a machine with n states, and symbols 0, 1, beginning on a blanked out tape, can run and then halt? How many 1's can such a machine print before halting?

(More generally, we might ask how long a machine with n states and m symbols might run, and how many non-zero symbols it might print, before halting)

Radó called the longest running machine of a given size a "Busy Beaver" and the number of steps it runs, the **Busy Beaver Function** $\Sigma(n)$.[5]

Precisely because the Halting Problem is undecidable, there can be no computable function bounding $\Sigma(n)$. For if there were, we could check whether a given machine halts or not— calculate this bound and try to run the machine for that many steps. If you succeed, then it will never halt, and if not, then it did halt.

[5]Though there are a finite number of distinct Turing machines with n states, in general there is no way to find a Busy Beaver among them. In practice, the number of machines of a given size is astronomical. But more importantly, there will be good candidates that one will never know whether they halt or not, whether they are in fact potential Busy Beavers.

There should be small machines that run for a great long while, then halt. And indeed there are: T. and S. Ligocki found this amazing machine in 2007 [**?**] which runs for more than 7.6×10^{868} steps, printing out more than 4.6×10^{434} non-zero symbols, then grinding to a halt.

	A	B	C
0	$1RB$	$0RC$	$1LB$
1	$0RB$	$1RH$	$2LA$
2	$3LC$	$2RC$	$3LA$
3	$1RC$	$3RC$	$2RB$

H. Marxen and J. Buntrock, discovered a six-state machine, using just the two symbols $0, 1$, that runs for an astounding 3×10^{1730} steps and then halts, printing out more than 10^{865} 1's [].

	A	B	C	D	E	F
0	$1RB$	$0RC$	$1LD$	$0LE$	$0RA$	$1LA$
1	$0LF$	$0RD$	$1RE$	$0LD$	$1RC$	$1RH$

Even a machine with a small number of states, working with just a few symbols can behave utterly inscrutably.

Seeing such examples, a mathematician asks *why?* Why do these behave as they do? Can we design even stranger machines? (Or for that matter, understand the limits of how strangely a machine can behave?) This is fundamentally what mathematicians try to do: reduce complicated, unruly structure to simplified observations— theorems— describing how things are.

As we shall see in a moment, the underlying undecidability of the Halting Problem implies that these machines behave as they do *for no good reason at all!*

That a given machine halts in after certain number of steps is a mathematical statement, and can be proven (simply by running the machine and checking). Nontheless, as we shall discuss further, in some fundamental, real sense, the behavior of these machines, in general, is beyond mathematical description, is beyond mathematics.

The Halting Problem for Turing machines

We first pause to recast the Halting Problem in terms of Turing machines: Is there a procedure for deciding whether a given Turing machine eventually reaches its halt state? The same argument we gave before still applies.

Just as we enumerated procedures, we can enumerate Turing machines, listing all possible machines, in some order, one after the other. We could pick some convention, maybe even the one I used to type the tables above, describing each machine as a string of symbols. We can then enumerate all possible strings, keeping only those that are not malformed and do describe working machines.

We can ask, does the nth machine $\mathcal{M}_n$ eventually halt, if it begins to run a tape marked with the number n (written out as a string of symbols)?

Is there a machine $\mathcal{H}$ that can decide if the nth machine eventually halts or not? That is, is there a machine that, if it begins on a tape marked n, eventually halts, writing "Y E S" on the tape if $\mathcal{M}_n$ halts on input n, or "N O" if it does not?

As before, no such machine can exist, for if there were, we could design a new machine $\mathcal{H}'$ that instead of writing out an answer and halting, either fails to halt (if $\mathcal{M}_n$ halts on n), or does halt (if $\mathcal{M}_n$ does not). As before, $\mathcal{H}'$ appears somewhere on our list of all machines; and so is $\mathcal{M}_h$ for some h. And exaclty as before, we have a contradiction: $\mathcal{H}'$ halts on h if and only if it does not. No such $\mathcal{H}'$, nor $\mathcal{H}$, can exist and the Halting Problem is undecidable for Turing machines.

All we have really done is to formalize, a little more, the ideas we laid out in Section 3.

If a Turing machines does halt, why does it do so?

Why do the strange Turing machines we have just seen halt as they do?

It is productive to take a step back and ask: *What do we mean when we say that something happens for a reason?*

If we are discussing a mathematical fact, we might just mean that the fact can be proven— the proof itself is the "reason" that the fact is true.

But if a proof is very, very long, perhaps so long that no human could really understand it in any detail, then the proof is certainly not a very *good* reason. A "good" reason, fundamentally, is a short proof, some argument that one human being can convey to another that a fact is really true.

Let's fix some sort of reasonable system of logic— the specific details are not so important for us here and we really can approach this topic fairly naïvely. But we mean a way of writing down logical expressions, and

a collection of formal rules for manipulating them. In such a scheme, if we can manipulate an expression— a theorem— eventually obtaining an expression which is patently true, then we have a proof.

In order to be of any use, a logical system should be reasonably powerful, but free of inconsistencies.[6] Any reasonable logical system, first of all, should at least be powerful enough to be able to prove that a given Turing machine halts, if in fact it does, just by tracking the machine's behavior. [[This might need more elaboration]]

And we should be able to design a Turing machine that can check to see whether a proof in our reasonable system is correct, just by making sure that all of our logical steps have been carried out appropriately. [[This might need more elaboration]]

Since our system is written in only a finite number of symbols, we can enumerate all possible strings. Since in our reasonable system we check whether a string encodes a valid proof, we can discard those that don't, and so have a procedure for enumerating all possible proofs. That is, there is some Turing machine that generates, in turn all possible proofs.

And of course, we also want our reasonable system to only include statements that are true or are false but are never both! A system with inherent contradictions is not of much use.

It is not easy to really pin down all the details— designing such systems was a great endeavor involving many mathematicians of the early 20th Century [[ref and cite; also Turing's 1936 paper]]. But it is easy to believe that it can be done, and implicitly we use this belief in every proof we write.

Once we have fixed a reasonable logical system, we can define the **proof complexity** of a provable statement as the length of its shortest proof; informally, then, a statement has a "good reason" if its proof complexity is relatively low.

If a Turing machine halts, we can prove this (just by running the machine, described as a series of logical statements). But this proof may be much longer than the shortest possible proof— there might be a much simpler reason why the machine gets around to halting. So what can we say about the proof complexity of a given machine's halting?

In our enumeration of all possible Turing machines, some will halt, and some will not (starting, say, on a blank tape). Let $\mathcal{M}'_n$ be the nth Turing

[6] As we shall shortly see, [[Gödel]]

machine that does halt— we have no way of knowing which machine this is, in general, since the Halting problem is undecidable, but $\mathcal{M}'_n$ is well-defined— and let $P(n)$ be the proof complexity of the fact that it halts.

Amazingly, $P(n)$ cannot be bounded by any computable function! In other words, *many machines halt for no good reason at all!* Indeed, since $P(n)$ is no greater than $H(n)$, we may say that, in a sense, the machines that run the longest are the least reasonable in doing so!

Suppose $P(n)$ were bounded by a computable function, say $b(n)$. Then we would have a procedure for deciding whether the nth machine halts:

The nth halting machine is also the mth machine overall, for some $m \geq n$. Now we can calculate $B = \max_{k \leq m} b(k)$ and enumerate all possible proofs of length up to B. We have assumed that we can check whether a proof is valid, and so whether a proof demonstrates a given theorem, namely that $\mathcal{M}'_n$ halts.

We can check, one by one, whether any of the proofs we've listed proves that $\mathcal{M}'_n$ halts. If any does, of course, then the machine does halt. Conversely, if $\mathcal{M}'_n$ does halt, one of the proofs we listed will have to prove this is so, since $b(n) \leq \max_{k \leq m} b(k) = B$.

Consequently, no such $b(n)$ can exist and $P(n)$ cannot be bounded by any computable function!

Gödel's Theorem

At the beginning of the 20th century, many mathematicians attempted to construct a solid logical foundation for all of mathematics, a systematic collection of logical rules from which all of mathematics could be built, [[the kinds of "reasonable logical systems" we discussed above]].

A reasonable logical system should be powerful enough to capture at least simple mathematical truths, such as the arithmetic properties of the counting numbers, or proving a given Turing machine halts or not. Its statements and proofs should be in some regular syntax, and thus possible, with some procedure, to check to see whether they are correct and to list out.

And it should be free of inconsistencies. We would also hope, though, that a reasonable system would be "complete" as well, able to prove any statement that happened to be true within it. In other words, we would not expect that there might be *true* but *unprovable* theorems!

In 1931, Kurt Gödel proved a remarkable theorem, in effect that there

can be no reasonable logical system that is complete, that *every* reasonable logical system must have true but unprovable theorems! And indeed, Gödel *proved* this.

Indeed, there must be an infinite collection of such theorems, as the Halting Problem shows us:

In our enumeration of all possible Turing machines, let $T(n)$ be the statement that the nth machine halts. Either $T(n)$ is true (the machine does halt), or its negation $T'(n)$ is true (the machine does not actually halt). If $T(n)$ is true, there is a proof: run the machine. As we discussed above, this may not be a particularly satisfying proof, and might be very long, but $T(n)$ is provable.

But there are some n for which $T'(n)$ is true, but has no proof! Suppose that this were not the case, that every true $T'(n)$ could be proven. Then for all n, there would be a proof of either $T(n)$ or $T'(n)$. But then this gives a procedure for solving the Halting Problem: Examine every single proof in turn; eventually one will come across a proof of $T(n)$ or $T'(n)$, depending on whether the nth machine halts or not. In either case, the problem is settled. But as we know, the Halting Problem is undecidable, and so there must be n for which $T'(n)$ is undecidable.[7]

One final note

Before getting back to geometry, we pause for one final technical note. It would be crazy to have a different computer for each possible task that we might face! Instead, of course, we use a general purpose machine that can be programmed to carry out any computation we might wish to (assuming we have sufficient memory and time!)

In the same way, there are more general purpose, "universal" Turing machines that can be "programmed" to carry out any computation. In effect, the programmed instructions it should follow appear in a special region of the tape. In fact, if you think about it, this is exactly the way that you or I carried out the instructions given in the tables above: we have one region for writing and reading. At each step, we keep track of where we are on this portion of the tape, but make a quick run over to the table to look up what we should do next based on where we are now. It is not so difficult, really, to create a machine that can carry this out— if we are not trying to be terribly efficient about it.

[7] In fact, there must be infinitely many such n. Can you see why?

However, there are some remarkably simple universal Turing machines. In 2007, Alex Smith proved that this very simple machine is universal, winning a prize offered by Stephen Wolfram: [**?**]

	A	B
0	$1LA$	$2RA$
1	$0LA$	$0RB$
2	$1RB$	$0LA$

[[Consequently, all of our results carry over to this simple machine: no halt state, so halting problem doesn't quite apply]]

But what does this have to do with mathematics as a whole? Are these phenomena special and isolated? One of the aims of this book is to explore how these issues play out in what might be considered recreational mathematics. After all, the topics in the first two chapters of this book are not so difficult— and yet, as we shall see, they touch on the foundation of mathematics.

And, really, there is nothing particularly special about tilings (or for that matter Turing machines). Everywhere in mathematics we see the same phenomenon rearing up— inscrutability and undecidability lurk in every corner.

Mathematicians, though, necessarily avoid such intractable topics; after all, what is the point of asking questions for which there can be no answer?

In some real way, though, it is as if there are small islands of decidable truth in vast unknowable seas. The strength of a mathematician, in large part, is measured by an ability to navigate towards shore. Nonetheless, some are attracted to the untamable stormy waters.