2
Algorithms and Turing machines

Background to the algorithm concept

What precisely *is* an algorithm, or a Turing machine, or a universal Turing machine? Why should these concepts be so central to the modern view of what could constitute a 'thinking device'? Are there any absolute limitations to what an algorithm could in principle achieve? In order to address these questions adequately, we shall need to examine the idea of an algorithm and of Turing machines in some detail.

In the various discussions which follow, I shall sometimes need to refer to mathematical expressions. I appreciate that some readers may be put off by such things, oı perhaps find them intimidating. If you are such a reader, I ask your indulgence, and recommend that you follow the advice I have given in my 'Note to the reader' on p. viii! The arguments given here do not require mathematical knowledge beyond that of elementary school, but to follow them in detail, some serious thought would be required. In fact, most of the descriptions are quite explicit, and a good understanding can be obtained by following the details. But much can also be gained even if one simply skims over the arguments in order to obtain merely their flavour. If, on the other hand, you are an expert, I again ask your indulgence. I suspect that it may still be worth your while to look through what I have to say, and there may indeed be a thing or two to catch your interest.

The word 'algorithm' comes from the name of the ninth century Persian mathematician Abu Ja'far Mohammed ibn Mûsâ *al-Khowârizm* who wrote an influential mathematical textbook, in about 825 AD, entitled 'Kitab al jabr w'al-muqabala'. The way that the name 'algorithm' has now come to be spelt, rather than the earlier and more accurate 'algorism', seems to have been due to an association with the word 'arithmetic'. (It is noteworthy, also, that the word 'algebra' comes from the Arabic 'al jabr' appearing in the title of his book.)

Instances of algorithms were, however, known very much earlier than al-Khowârizm's book. One of the most familiar, dating from ancient Greek times (c. 300 BC), is the procedure now referred to as *Euclid's algorithm* for finding the highest common factor of two numbers. Let us see how this works.

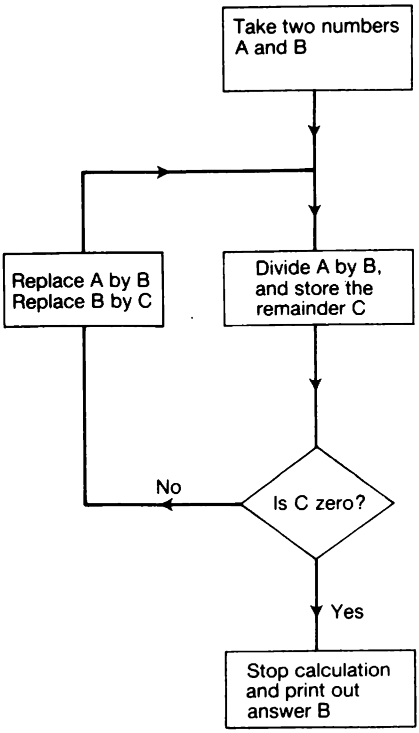
It will be helpful to have a specific pair of numbers in mind, say 1365 and 3654. The highest common factor is the largest single whole number that divides into each of these two numbers exactly. To apply Euclid's algorithm, we divide one of our two numbers by the other and take the remainder: 1365 goes twice into 3654, with remainder 924 (= 3654 − 2730). We now replace our original two numbers by this remainder, namely 924, and the number we just divided by, namely 1365, in that order. We repeat what we just did using this new pair: 924 goes once into 1365, with remainder 441. This gives another new pair, 441 and 924, and we divide 441 into 924 to get the remainder 42 (= 924 − 882), and so on until we get a division that goes exactly. Setting all this out, we get:

$$\begin{array}{rl}
3654 \div 1365 & \text{gives remainder } 924 \\
1365 \div 924 & \text{gives remainder } 441 \\
924 \div 441 & \text{gives remainder } 42 \\
441 \div 42 & \text{gives remainder } 21 \\
42 \div 21 & \text{gives remainder } 0.
\end{array}$$

The last number we divided by, namely 21, is the required highest common factor.

Euclid's algorithm itself is the *systematic procedure* by which we found this factor. We have just applied this procedure to a particular pair of numbers, but the procedure itself applies quite generally, to numbers of any size. For very large numbers, the procedure could take a very long time to carry out, and the larger the numbers, the longer the procedure will tend to take. But in any *specific* case the procedure will eventually terminate and a definite answer will be obtained in a finite number of steps. At each step it is perfectly clear-cut what the operation is that has to be performed, and the decision as to the moment at which the whole process has terminated is also perfectly clear-cut. Moreover, the description of the whole procedure can be presented in *finite* terms, despite the fact that it applies to natural numbers of unlimited size. (The 'natural numbers', are simply the ordinary non-negative[1] whole numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...). Indeed, it is easy to construct a (finite) 'flow chart' to describe the full logical operation of Euclid's algorithm (see next page).

It should be noted that this procedure is still not quite broken down into its most elementary parts since it has been implicitly assumed that we already 'know' how to perform the necessary basic operation of obtaining the remainder from a division, for two arbitrary natural numbers **A** and **B**. That operation is again algorithmic—performed by the very familiar procedure for division that we learn at school. That procedure is actually rather more complicated than the rest of Euclid's algorithm, but again a flow chart may be constructed. The main complication results from the fact that we would (presumably) be using the standard 'denary' notation for natural numbers, so

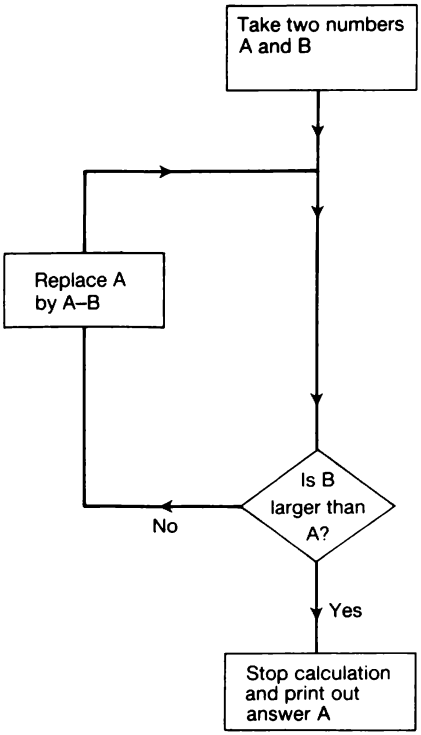

Take two numbers A and B
Replace A by B Replace B by C
Divide A by B, and store the remainder C
No
Is C zero?
Yes
Stop calculation and print out answer B

that we would need to list all our multiplication tables and worry about carrying, etc. If we had simply used a succession of *n* marks of some kind to represent the number *n*—for example, ●●●●● to represent five—then the forming of a remainder would be seen as a very elementary algorithmic operation. In order to obtain the remainder when **A** is divided by **B** we simply keep removing the succession of marks representing **B** from that representing **A** until there are not enough marks left to perform the operation again. The last remaining succession of marks provides the required answer. For example, to obtain the remainder when seventeen is divided by five we simply proceed by removing successions of ●●●●● from ●●●●●●●●●●●●●●●●● as follows:

```
●●●●●●●●●●●●●●●●●
     ●●●●●●●●●●●●
          ●●●●●●●
               ●●
```

and the answer is clearly two since we cannot perform the operation again.

A flow chart which finds the remainder in a division, achieved by this means of repeated subtraction, can be given as follows:



To complete the entire flow chart for Euclid's algorithm, we substitute the above chart for forming a remainder into the box at the centre right in our original chart. This kind of substitution of one algorithm into another is a common computer-programming procedure. The above algorithm for finding a remainder is an example of a *subroutine*, that is, it is a (normally previously known) algorithm which is called upon and used by the main algorithm as a part of its operation.

Of course the representation of the number *n* simply as *n* spots is very inefficient when large numbers are involved, which is why we normally use a more compact notation such as the standard (denary) system. However we shall not be much concerned with the *efficiency* of operations or notations here. We are concerned instead with the question of what operations can *in principle* be performed algorithmically. What is algorithmic if we use one notation for numbers is also algorithmic if we use the other. The only differences lie in the detail and complication in the two cases.

Euclid's algorithm is just one among the numerous, often classical, algorithmic procedures that are to be found throughout mathematics. But it is perhaps remarkable that despite the ancient historic origins of specific examples of algorithms, the precise formulation of the *concept of a general* *algorithm* dates only from *this* century. In fact, various alternative descriptions of this concept have been given, all in the 1930s. The most direct and persuasive of these, and also historically the most important, is in terms of the concept known as a *Turing machine*. It will be appropriate for us to examine these 'machines' in some detail.

One thing to bear in mind about a Turing 'machine' will be that it is a piece of 'abstract mathematics' and not a physical object. The concept was introduced by the English mathematician, code-breaker extraordinary and seminal computer scientist Alan Turing in 1935–6 (Turing 1937), in order to tackle a very broad-ranging problem, known as the *Entscheidungsproblem*, partially posed by the great German mathematician David Hilbert in 1900, at the Paris International Congress of Mathematicians ('Hilbert's tenth problem'), and more completely, at the Bologna International Congress in 1928. Hilbert had asked for no less than a general algorithmic procedure for resolving mathematical questions—or, rather, for an answer to the question of whether or not such a procedure might in principle exist. Hilbert also had a programme for placing mathematics on an unassailably sound foundation, with axioms and rules of procedure which were to be laid down once and for all, but by the time Turing produced his great work, that programme had already suffered a crushing blow from a startling theorem proved in 1931 by the brilliant Austrian logician Kurt Gödel. We shall be considering the Gödel theorem and its significance in Chapter 4. The problem of Hilbert's that concerned Turing (the *Entscheidungsproblem*) went beyond any particular formulation of mathematics in terms of axiomatic systems. The question was: is there some general mechanical procedure which could, *in principle*, solve all the problems of mathematics (belonging to some suitably well-defined class) one after the other?

Part of the difficulty in answering this question was to decide what is to be meant by a 'mechanical procedure'. The concept lay outside the normal mathematical ideas of the time. In order to come to grips with it, Turing tried to imagine how the concept of a 'machine' could be formalized, its operation being broken down into elementary terms. It seems clear that Turing also regarded a human brain to be an example of a 'machine' in his sense, so whatever the activities might be that are carried out by human mathematicians when they tackle their problems of mathematics, these also would have to come under the heading of 'mechanical procedures'.

Whilst this view of human thinking appears to have been valuable to Turing in the development of his highly important concept, it is by no means necessary for us to adhere to it. Indeed, by making precise what is meant by a mechanical procedure, Turing actually showed that there are some perfectly well-defined mathematical operations which cannot, in any ordinary sense, be called mechanical! There is perhaps some irony in the fact that this aspect of Turing's own work may now indirectly provide us with a possible loophole to his own viewpoint concerning the nature of mental phenomena. However, this is not our concern for the moment. We need first to find out what Turing's concept of a mechanical procedure actually is.

Turing's concept

Let us try to imagine a device for carrying out some (finitely definable) calculational procedure. What general form would such a device take? We must be prepared to idealize a little and not worry too much about practicalities: we are really thinking of a mathematically idealized 'machine'. We want our device to have a discrete set of different possible states, which are *finite* in number (though perhaps a very large number). We call these the *internal* states of the device. However we do not want to limit the size of the calculations that our device will perform in principle. Recall Euclid's algorithm described above. There is, in principle, no limit to the size of the numbers on which that algorithm can act. The algorithm—or the general calculational *procedure*—is just the same no matter how large the numbers are. For very large numbers, the procedure may indeed take a very long time, and a considerable amount of 'rough paper' may be needed on which to perform the actual calculations. But the *algorithm* is the same *finite* set of instructions no matter how big the numbers.

Thus, although it has a finite number of internal states, our device must be able to deal with an input which is not restricted in its size. Moreover, the device must be allowed to call upon an unlimited external storage space (our 'rough paper') for its calculations, and also to be able to produce an output of unlimited size. Since our device has only a finite number of distinct internal states, it cannot be expected to 'internalize' all the external data nor all the results of its own calculations. Instead it must examine only those parts of the data or previous calculations that it is *immediately* dealing with, and then perform whatever operation it is required to perform on them. It can note down, perhaps in the external storage space, the relevant results of that operation, and then proceed in a precisely determined way to the next stage of operation. It is the unlimited nature of the input, calculation space, and output which tells us that we are considering only a mathematical idealization rather than something that could be actually constructed in practice (see fig. 2.1). But it is an idealization of great relevance. The marvels of modern computer technology have provided us with electronic storage devices which can, indeed, be treated as unlimited for most practical purposes.

In fact, the type of storage space that has been referred to as 'external' in the above discussion could be regarded as actually part of the internal workings of a modern computer. It is perhaps a technicality whether a certain part of the storage space is to be regarded as internal or external. *One* way of

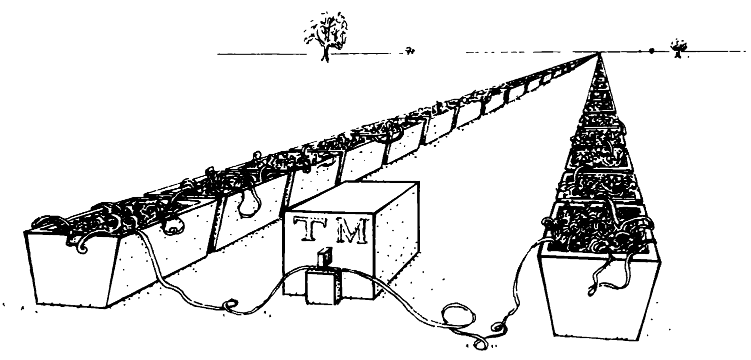


Fig. 2.1. A strict Turing machine requires an infinite tape!

referring to this division between the 'device' and the 'external' part would be in terms of *hardware* and *software*. The internal part could then be the hardware and the external part, the software. I shall *not* necessarily stick to this, but whichever way one looks at it, Turing's idealization is indeed remarkably well approximated by the electronic computers of today.

The way that Turing pictured the external data and storage space was in terms of a 'tape' with marks on it. This tape would be called upon by the device and 'read' as necessary, and the tape could be moved backwards or forwards by the device, as part of its operation. The device could also place new marks on the tape where required, and could obliterate old ones, allowing the *same* tape to act as external storage (i.e. 'rough paper') as well as input. In fact it is helpful not to make any clear distinction between 'external storage' and 'input' because in many operations the intermediate results of a calculation can play a role just like that of new data. Recall that with Euclid's algorithm we kept replacing our original input (the numbers **A** and **B**) by the results of the different stages of the calculation. Likewise, the same tape can be used for the final output (i.e. the 'answer'). The tape will keep running back and forth through the device so long as further calculations need to be performed. When the calculation is finally completed, the device comes to a halt and the answer to the calculation is displayed on that part of the tape which lies to one side of the device. For definiteness, let us suppose that the answer is always displayed on the left, while all the numerical data in the input, together with the specification of the problem to be solved, always goes in on the right.

For my own part, I feel a little uncomfortable about having our finite device moving a potentially infinite tape backwards and forwards. No matter how lightweight its material, an *infinite* tape might be hard to shift! Instead, I prefer to think of the tape as representing some external environment through which our finite device can move. (Of course, with modern electronics, neither the 'tape' nor the 'device' need actually 'move' in the ordinary physical sense, but such 'movement' is a convenient way of picturing things.) On this view, the device receives all its input from this environment. It uses the environment as its 'rough paper'. Finally it writes out its output on this same environment.

In Turing's picture the 'tape' consists of a linear sequence of squares, which is taken to be infinite in both directions. Each square on the tape is either blank or contains a single mark.* The use of marked or unmarked squares illustrates that we are allowing our 'environment' (i.e. the tape) to be broken down and described in terms of *discrete* (as opposed to continuous) elements. This seems to be a reasonable thing to do if we wish our device to function in a reliable and absolutely definite way. We are, however, allowing this 'environment' to be (potentially) infinite, as a feature of the mathematical idealization that we are using, but in any *particular* case the input, calculation and output must always be *finite*. Thus, although the tape is taken to be infinitely long, there must only be a finite number of actual marks on it. Beyond a certain point in each direction the tape must be entirely blank.

We indicate a blank square by the symbol '0' and a marked one by the symbol '1', e.g.:

....	0	0	0	1	1	1	1	0	1	0	0	1	1	1	0	0	1	0	0	1	0	1	1	0	1	0	0	....

We need our device to 'read' the tape, and we shall suppose that it does this *one* square at a time, and after each operation moves just *one* square to the right or left. There is no loss in generality involved in that. A device which reads *n* squares at a time or moves *k* squares at a time can easily be modelled by another device which reads and moves just one square at a time. A movement of *k* squares can be built up of *k* movements of one square, and by storing up *n* readings of one square it can behave as though it reads all *n* squares at once.

What, in detail, can such a device do? What is the most general way in which something that we would describe as 'mechanical' might function? Recall that the *internal states* of our device are to be finite in number. All we need to know, beyond this finiteness is that the behaviour of the device is completely determined by its internal state and by the input. This input we have simplified to being just one of the two symbols '0' or '1'. Given its initial state and this input, the device is to operate completely deterministically: it changes its internal state to some other (or possibly the same) internal state; it

* In fact, in his original descriptions Turing allowed his tape to be marked in more complicated ways, but this makes no real difference. The more complicated marks could always be broken down into successions of marks and blanks. I shall be taking various other unimportant liberties with Turing's original specifications.

replaces the 0 or 1 that it is just reading by the same or different symbol 0 or 1; it moves one square either to the right or left; finally, it decides whether to continue the calculation or to terminate it and come to a halt.

To define the operation of our device in an explicit way, let us first *number* the different internal states, say by the labels 0, 1, 2, 3, 4, 5, . . . ; the operation of the device, or *Turing machine*, would then be completely specified by some explicit list of replacements, such as:

00 → 00R
01 → 131L
10 → 651R
11 → 10R
20 → 01R.STOP
21 → 661L
30 → 370R
. .
. .
. .
2100 → 31L
. .
. .
. .
2581 → 00R.STOP
2590 → 971R
2591 → 00R.STOP

The *large* figure on the left-hand side of the arrow is the symbol on the tape that the device is in the process of reading, and the device replaces it by the large figure at the middle on the right. R tells us that the device is to move one square to the *right* along the tape and L tells us that it is to move by one step to the *left*. (If, as with Turing's original descriptions, we think of the tape moving rather than the device, then we must interpret R as the instruction to move the *tape* one square to the *left* and L as moving it one square to the *right*.) The word STOP indicates that the calculation has been completed and the device is to come to a halt. In particular, the second instruction 01 → 131L tells us that *if* the device is in internal state 0 and reads 1 on the tape then it must change to internal state 13, leave the 1 as a 1 on the tape, and move one square along the tape to the left. The last instruction 2591 → 00R.STOP tells us that if the device is in state 259 and reads 1 on the tape, then it must revert to state 0, erase the 1 to produce 0 on the tape, move one square along the tape to the right, and terminate the calculation.

Instead of using the numerals 0, 1, 2, 3, 4, 5, . . . for labelling the internal states, it would be somewhat more in keeping with the above notation for marks on the tape if we were to use symbols made up of just 0s and 1s. We could simply use a succession of n 1s to label the state n if we choose, but that is inefficient. Instead, let us use the *binary* numbering system, which is now a familiar mode of notation:

$$
\begin{aligned}
0 &\to 0,\\
1 &\to 1,\\
2 &\to 10,\\
3 &\to 11,\\
4 &\to 100,\\
5 &\to 101,\\
6 &\to 110,\\
7 &\to 111,\\
8 &\to 1000,\\
9 &\to 1001,\\
10 &\to 1010,\\
11 &\to 1011,\\
12 &\to 1100, \text{etc.}
\end{aligned}
$$

Here the final digit on the right refers to the 'units' just as it does in the standard (denary) notation, but the digit just before it refers to 'twos' rather than 'tens'. The one before that refers to 'fours' rather than 'hundreds' and before that, to 'eights' rather than 'thousands', and so on, the value of each successive digit, as we move to the left, being the successive *powers of two*: 1, 2, 4(=2×2), 8(= 2 × 2 × 2), 16(= 2 × 2 × 2 × 2), 32(= 2 × 2 × 2 × 2 × 2), etc. (For some other purposes that we shall come to later, we shall also sometimes find it useful to use a base other than two or ten to represent natural numbers: e.g. in base *three*, the denary number 64 would be written 2101, each digit having a value which is now a power of three: $64=(2 \times 3^3) + 3^2 + 1$; cf. Chapter 4, p. 106, footnote.)

Using such a binary notation for the internal states, the specification of the above Turing machine would now be:

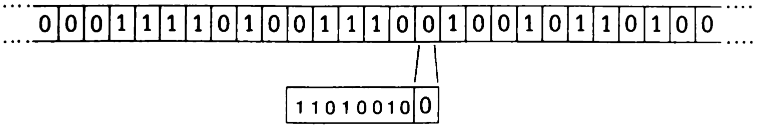
$$
\begin{aligned}
00 &\to 00\text{R}\\
01 &\to 11011\text{L}\\
10 &\to 10000011\text{R}\\
11 &\to 10\text{R}\\
100 &\to 01\text{STOP}\\
101 &\to 10000101\text{L}\\
110 &\to 1001010\text{R}\\
&\vdots\\
110100100 &\to 111\text{L}
\end{aligned}
$$

. .
. .
. .

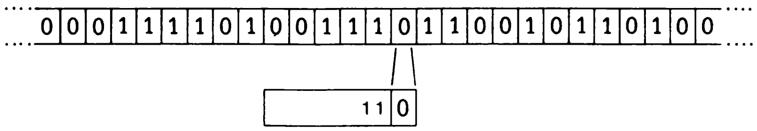
1000000101 → 00STOP
1000000110 → 1100001 1R
1000000111 → 00STOP

In the above, I have also abbreviated R.STOP to STOP, since we may as well assume that L.STOP never occurs so that the result of the final step in the calculation is always displayed at the left of the device, as part of the answer.

Let us suppose that our device is in the particular internal state represented by the binary sequence 11010010 and is in the midst of a calculation for which the tape is given as on p. 37, and we apply the instruction 110100100→111L:



The particular digit on the tape that is being read (here the digit '0') is indicated by a larger figure, to the right of the string of symbols representing the internal state. In the example of a Turing machine as partly specified above (and which I have made up more or less at random), the '0' which is being read would be replaced by a '1' and the internal state would be changed to '11'; then the device would be moved one step to the left:



The device is now ready to read another digit, again a '0'. According to the table, it now leaves this '0' unchanged, but replaces the internal state by '100101' and moves back along the tape to the right by one step. Now it reads '1', and somewhere down the table would be a further instruction as to what replacement to make in its internal state, whether it should change the digit it is reading, and in which direction it should move along the tape. It would continue this way until it reaches a STOP, at which point (after it moves one further step to the right) we imagine a bell ringing to alert the operator of the machine that the calculation has been completed.

We shall suppose that the machine is always started with internal state '0' and that all the tape to the left of the reading device is initially blank. The instructions and data are all fed in at the right. As mentioned earlier, this information which is fed in is always to take the form of a *finite* string of 0s and 1s, followed by blank tape (i.e. 0s). When the machine reaches STOP, the result of the calculation appears on the tape to the left of the reading device.

Since we wish to be able to include numerical data as part of our input, we shall want to have a way of describing ordinary numbers (by which I here mean the natural numbers 0, 1, 2, 3, 4, . . .) as part of the input. One way to do this might be simply to use a string of *n* 1s to represent the number *n* (although this could give us a difficulty with the natural number zero):

1→1, 2→11, 3→111, 4→1111, 5→11111, etc.

This primitive numbering system is referred to (rather illogically) as the *unary* system. Then the symbol '0' could be used as a space to separate different numbers from one another. It is important that we have such a means of separating numbers from one another since many algorithms act on *sets* of numbers rather than on just single numbers. For example, for Euclid's algorithm, our device would need to act on the *pair* of numbers A and B. Turing machines can be written down, without great difficulty, which effect this algorithm. As an exercise, some dedicated readers might perhaps care to verify that the following explicit description of a Turing machine (which I shall call **EUC**) does indeed effect Euclid's algorithm when applied to a pair of unary numbers separated by a 0:

00→00R, 01→11L, 10→101R, 11→11L, 100→10100R,
101→110R, 110→1000R, 111→111R, 1000→1000R, 1001→1010R,
1010→1110L, 1011→1101L, 1100→1100L, 1101→11L, 1110→1110L,
1111→10001L, 10000→10010L, 10001→10001L, 10010→100R,
10011→11L, 10100→00STOP, 10101→10101R.

Before embarking on this, however, it would be wise for any such reader to start with something much simpler, such as the Turing machine **UN+1**:

00→00R, 01→11R, 10→01STOP, 11→11R,

which simply adds one to a unary number. To check that **UN+1** does just that, let us imagine that it is applied to, say, the tape

. . . 00000111100000 . . .,

which represents the number 4. We take the device to be initially somewhere off to the left of the 1s. It is in internal state 0 and reads a 0. This it leaves as 0, according to the first instruction, and it moves off one step to the right, staying in internal state 0. It keeps doing this, moving one step to the right until it meets the first 1. Then the second instruction comes into play: it leaves the 1 as a 1 and moves to the right again, but now in internal state 1. In accordance with the fourth instruction, it stays in internal state 1, leaving the 1s alone, moving along to the right until it reaches the first 0 following the 1s.

The third instruction then tells it to change that 0 to a 1, move one further step to the right (recall that STOP stands for R.STOP) and then halt. Thus, another 1 has been added to the string of 1s, and the 4 of our example has indeed been changed to 5, as required.

As a somewhat more taxing exercise, one may check that the machine **UN×2**, defined by

$$00\rightarrow 00\text{R},\quad 01\rightarrow 10\text{R},\quad 10\rightarrow 101\text{L},\quad 11\rightarrow 11\text{R},\quad 100\rightarrow 110\text{R},\quad 101\rightarrow 1000\text{R},$$
$$110\rightarrow 01\text{STOP},\quad 111\rightarrow 111\text{R},\quad 1000\rightarrow 1011\text{L},\quad 1001\rightarrow 1001\text{R},\quad 1010\rightarrow 101\text{L},$$
$$1011\rightarrow 1011\text{L},$$

doubles a unary number, as it is intended to.

In the case of **EUC**, to get the idea of what is involved, some suitable explicit pair of numbers can be tried, say 6 and 8. The reading device is, as before, taken to be in state 0 and initially on the left, and the tape would now be initially marked as:

. . . 00000000000011111101111111100000 . . . .

After the Turing machine comes to a halt, many steps later, we get a tape marked

. . . 0000110000000000 0 . . .

with the reading device to the right of the non-zero digits. Thus the required highest common factor is (correctly) given as 2.

The full explanation of *why* **EUC** (or, indeed, **UN×2**) actually does what it is supposed to do involves some subtleties and would be rather more complicated to explain than the machine is complicated itself—a not uncommon feature of computer programs! (To understand fully why an algorithmic procedure does what it is supposed to involves *insights*. Are 'insights' themselves algorithmic? This is a question that will have importance for us later.) I shall not attempt to provide such an explanation here for the examples **EUC** or **UN×2**. The reader who does check them through will find that I have taken a very slight liberty with Euclid's actual algorithm in order to express things more concisely in the required scheme. The description of **EUC** is still somewhat complicated, comprising 22 elementary instructions for 11 distinct internal states. Most of the complication is of a purely organizational kind. It will be observed, for example, that of the 22 instructions, only 3 actually involve altering marks on the tape! (Even for **UN×2** I have used 12 instructions, half of which involve altering the marks.)

Binary coding of numerical data

The unary system is exceedingly inefficient for the representation of numbers of large size. Accordingly, we shall usually use the *binary* number system, as described earlier. However, we cannot just do this directly, attempting to read the tape simply as a binary number. As things stand, there would be no way of telling when the binary representation of the number has come to an end and the infinite succession of 0s representing the blank tape on the right begins. We need some notation for terminating the binary description of a number. Moreover, we shall often want to feed in *several* numbers, as with the *pair* of numbers[2] required for Euclid's algorithm. As things stand, we cannot distinguish the *spaces* between numbers from the 0s or strings of 0s that appear as parts of the binary representation of single numbers. In addition, we might perhaps also want to include all kinds of complicated instructions on the input tape, as well as numbers. In order to overcome these difficulties, let us adopt a procedure which I shall refer to as *contraction*, according to which any string of 0s and 1s (with a finite total number of 1s) is *not* simply read as a binary number, but is replaced by a string of 0s, 1s, 2s, 3s, etc., by a prescription whereby each digit of the second sequence is simply the number of 1s lying between successive 0s of the first sequence. For example, the sequence

01000101101010110100011101010111100110

would be replaced

```
       010 0 010110101011010 0 01110101011110 0110
       | | | | |   | | |   | | | |    | | |    | | |
thus:   1 0 0 1  2  1 1  2  1 0 0  3   1 1   4  0 2
```

We can now read the numbers 2, 3, 4, . . . as markers or instructions of some kind. Indeed, let us regard 2 as simply a 'comma', indicating the space between two numbers, whereas 3, 4, 5, . . . could, according to our wishes, represent various instructions or notations of interest, such as 'minus sign', 'plus', 'times', 'go to the location with the following number', 'iterate the previous operation the following number of times', etc. We now have various strings of 0s, and 1s which are separated by higher digits. The former are to represent ordinary numbers written in the binary scale. Thus, the above would read (with 'comma' for '2'):

(binary number 1001) comma (binary number 11) comma . . .

Using standard Arabic notation '9', '3', '4', '0' for the respective binary numbers 1001, 11, 100, 0, we get, for the entire sequence:

9, 3, 4 (instruction 3) 3 (instruction 4) 0,

In particular, this procedure gives us a means of terminating the description of a number (and thereby distinguishing it from an infinite stretch of blank tape on the right) simply by using a comma at the end. Moreover, it enables us to code any finite sequence of natural numbers, written in the binary notation, as a *single* sequence of 0s and 1s, where we use commas to separate the numbers. Let us see how this works in a specific case. Consider the sequence

5, 13, 0, 1, 1, 4,

for example. In binary notation this is

101, 1101, 0, 1, 1, 100,

which is coded on the tape, by *expansion* (i.e. the inverse of the above contraction procedure), as

. . . 000010010110101001011001101011010110100011000 . . . .

To achieve this coding in a simple direct way we can make replacements in our original sequence of binary numbers as follows

$$0 \rightarrow 0$$
$$1 \rightarrow 10$$
$$, \rightarrow 110$$

and then adjoin an unlimited supply of 0s at both ends. It is made clearer how this has been applied to the above tape, if we space it out:

0000 10 0 10 110 10 10 0 10 110 0 110 10 110 10 110 10 0 0 110 00

I shall refer to this notation for (sets of) numbers as the *expanded binary* notation. (So, in particular, the expanded binary form of 13 is 1010010.)

There is one final point that should be made about this coding. It is just a technicality, but necessary for completeness.[3] In the binary (or denary) representation of natural numbers there is a slight redundancy in that 0s placed on the far left of an expression do not 'count'—and are normally omitted, e.g. 00110010 is the same binary number as 110010 (and 0050 is the same denary number as 50). This redundancy extends to the number zero itself, which can be written 000 or 00 just as well as 0. Indeed a blank space should, logically, denote zero as well! In ordinary notation that would lead to great confusion, but it fits in well with the notation just described above. Thus a zero between two commas can just as well be written as two commas next to one another (,,) which would be coded on the tape as two pairs 11 separated by a single 0:

. . . 001101100. . . .

Thus the above set of six numbers can also be written in binary notation as

101,1101,,1,1,100,

and coded on the tape, in expanded binary form, as

. . . 00001001011010100101101101011010110100011000 . . .

(which has one 0 missing from the sequence that we had before).

We can now consider a Turing machine for effecting, say, Euclid's algorithm, applying it to pairs of numbers written in the expanded binary notation. For example, for the pair of numbers 6, 8 that we considered earlier, instead of using

. . . 0000000000011111101111111100000 . . . ,

as we did before, we consider the binary representations of 6 and 8, namely 110 and 1000, respectively. The *pair* is

6,8, i.e., in binary notation, 110,1000,

which, by expansion, is coded as the tape

. . . 00000101001101000011000000 . . .

For this particular pair of numbers there is no gain in conciseness from the unary form. Suppose, however, that we take, say, the (denary) numbers 1583169 and 8610. In binary notation these would be

110000010100001000001, 10000110100010,

so we have the pair coded as the tape

. . .
001010000001001000001000000101101000001010010000100110
. . .

which all fits on one line, whereas in the unary notation, the tape representing '1583169, 8610' would more than fill this entire book!

A Turing machine that effects Euclid's algorithm when the numbers are expressed in expanded binary notation could, if desired, be obtained simply by adjoining to **EUC** a suitable pair of subroutine algorithms which translate between unary and expanded binary. This would actually be extremely inefficient, however, since the inefficiency of the unary numbering system would still be 'internally' present and would show up in the slowness of the device and in the inordinate amount of external 'rough paper' (which would be on the left-hand part of the tape) that would be needed. A more efficient Turing machine for Euclid's algorithm operating entirely within expanded binary can also be given, but it would not be particularly illuminating for us here.

Instead, in order to illustrate how a Turing machine can be made to operate on expanded binary numbers, let us try something a good deal simpler than Euclid's algorithm, namely the process of simply *adding one* to a natural number. This can be effected by the Turing machine (which I shall call **XN+1**):

00→00R. 01→11R. 10→00R. 11→101R. 100→110L. 101→101R. 110→01STOP. 111→1000L. 1000→1011L. 1001→1001L. 1010→1100R. 1011→101R. 1101→1111R. 1110→111R. 1111→1110R.

Again, some dedicated readers might care to check that this Turing machine actually does what it is supposed to do, by applying it to, say, the number 167, which has binary representation 10100111 and so would be given by the tape

. . . 0000100100010101011000 . . . .

To add one to a binary number, we simply locate the final 0 and change it to 1 and then replace all the 1s which follow by 0s, e.g. 167 + 1 = 168 is written in binary notation as

$$10100111 + 1 = 10101000.$$

Thus our 'adding-one' Turing machine should replace the aforementioned tape by

. . . 0000100100100001100000 . . .

which indeed it does.

Note that even the very elementary operation of simply adding one is a bit complicated with this notation, using fifteen instructions and eight different internal states! Things were a lot simpler with the unary notation, of course, since 'adding one' then simply means extending the string of 1s by one further 1, so it is not surprising that our machine **UN+1** was more basic. However, for very large numbers, **UN+1** would be exceedingly slow because of the inordinate length of tape required, and the more complicated machine **XN+1**, which operates with the more compact expanded binary notation, would be better.

As an aside, I point out an operation for which the Turing machine actually looks simpler for expanded binary than for unary notation, namely *multiplying by two*. Here, the Turing machine **XN×2**, given by

00→00R. 01→10R. 10→01R. 11→100R. 100→111R. 110→01STOP.

achieves this in expanded binary, whereas the corresponding machine in unary notation, **UN×2**, which was described earlier, is a good deal more complicated!

This gives us some idea of what Turing machines can do at a very basic level. As might be expected, they can, and do, get vastly more complicated than this when operations of some complexity are to be performed. What is the ultimate scope of such devices? Let us consider this question next.

The Church–Turing Thesis

Once one has gained some familiarity with constructing simple Turing machines, it becomes easy to satisfy oneself that the various basic arithmetical operations, such as adding two numbers together, or multiplying them, or raising one number to the power of another, can indeed all be effected by specific Turing machines. It would not be too cumbersome to give such machines explicitly, but I shall not bother to do this here. Operations where the result is a *pair* of natural numbers, such as division with a remainder, can also be provided—or where the result is an arbitrarily large finite set of numbers. Moreover Turing machines can be constructed for which it is not specified ahead of time which arithmetical operation it is that needs to be performed, but the instructions for this are fed in on the tape. Perhaps the particular operation that has to be performed depends, at some stage, upon the result of some calculation that the machine has had to perform at some earlier stage. ('If the answer to that calculation was greater than so-and-so, do this; otherwise, do that.') Once it is appreciated that one can make Turing machines which perform arithmetic or simple logical operations, it becomes easier to imagine how they can be made to perform more complicated tasks of an algorithmic nature. After one has played with such things for a while, one is easily reassured that a machine of this type can indeed be made to perform *any mechanical operation whatever*! Mathematically, it becomes reasonable to *define* a mechanical operation to be one that can be carried out by such a machine. The noun 'algorithm' and the adjectives 'computable', 'recursive', and 'effective' are all used by mathematicians to denote the mechanical operations that can be performed by theoretical machines of this type—the Turing machines. So long as a procedure is sufficiently clear-cut and mechanical, then it is reasonable to believe that a Turing machine can indeed be found to perform it. This, after all, was the whole point of our (i.e. Turing's) introductory discussion motivating the very concept of a Turing machine.

On the other hand, it still could be felt that the design of these machines was perhaps unnecessarily restrictive. Allowing the device to read only one binary digit (0 or 1) at a time, and to move only one space at a time along only a *single* one-dimensional tape seems at first sight to be limiting. Why not allow four or five, or perhaps one thousand separate tapes, with a great number or interconnected reading devices running all at once? Why not allow a whole plane of squares of 0s and 1s (or perhaps a three-dimensional array) rather than insisting on a one-dimensional tape? Why not allow other symbols from some more complicated numbering system or alphabet? In fact, none of these changes makes the slightest difference to what can be in principle achieved, though some make a certain amount of difference to the economy of the operations (as would certainly be the case if we allowed more than one tape). The class of operations performed, and thus coming under the heading of 'algorithms' (or 'computations' or 'effective procedures' or 'recursive operations'), would be precisely the same as before even if we broadened the definition of our machines in all these ways at once!

We can see that there is no *necessity* to have more than one tape, so long as the device can keep finding new space on the given tape, as required. For this, it may need to keep shunting data from one place to another on the tape. This may be 'inefficient', but it does not limit what can be in principle achieved.[4] Likewise, using more than one Turing device in *parallel action*—which is an idea that has become fashionable in recent years, in connection with attempts to model human brains more closely—does *not* in principle gain anything (though there may be an improved speed of action under certain circumstances). Having two separate devices which do not directly communicate with one another achieves no more than having two which *do* communicate; and *if* they communicate, then, in effect, they are just a single device!

What about Turing's restriction to a one-dimensional tape? If we think of this tape as representing the 'environment', we might prefer to think of it as a planar surface rather than as a one-dimensional tape, or perhaps as a three-dimensional space. A planar surface might seem to be closer to what is needed for a 'flow chart' (as in the above description of the operation of Euclid's algorithm) than a one-dimensional tape would be*. There is, however, no difficulty in principle about writing out the operation of a flow diagram in a 'one-dimensional' form (e.g. by the use of an ordinary verbal description of the chart). The two-dimensional planar display is only for our own convenience and ease of comprehension and it makes no difference to what can in principle be achieved. It is always possible to code the location of a mark or an object on a two-dimensional plane, or even in a three-dimensional space, in a straightforward way on a one-dimensional tape. (In fact, using a two-dimensional plane is completely equivalent to using *two* tapes. The two tapes would supply the two 'coordinates' that would be needed for specifying a point on a two-dimensional plane; likewise *three* tapes can act as 'coordinates' for a point in a three-dimensional space.) Again this one-dimensional coding may be 'inefficient', but it does not limit what can be achieved in principle.

Despite all of this, we might still question whether the concept of a Turing machine really does incorporate *every* logical or mathematical operation that

* As things have been described here, this flow chart itself would actually be part of the 'device' rather than of the external environment 'tape'. It was the actual numbers **A**, **B**, **A**−**B**, etc, which we represented on the tape. However, we shall be wanting also to express the specification of the *device* in a linear one-dimensional form. As we shall see later, in connection with the *universal* Turing machine, there is an intimate relation between the specification for a particular 'device' and the specification possible 'data' (or 'program') for a given device. It is therefore convenient to have *both* of these in one-dimensional form.

we would wish to call 'mechanical'. At the time that Turing wrote his seminal paper, this was considerably less clear than it is today, so Turing found it necessary to put his case in appreciable detail. Turing's closely argued case found additional support from the fact that, quite independently (and actually a little earlier), the American logician Alonzo Church (with the help of S. C. Kleene) had put forward a scheme—the lambda calculus—also aimed at resolving Hilbert's *Entscheidungsproblem*. Though it was much less obviously a fully comprehensive mechanical scheme than was Turing's, it had some advantages in the striking economy of its mathematical structure. (I shall be describing Church's remarkable calculus at the end of this chapter.) Also independently of Turing there were yet other proposals for resolving Hilbert's problem (see Gandy 1988), most particularly that of the Polish-American logician Emil Post (a little later than Turing, but with ideas considerably more like those of Turing than of Church). All these schemes were soon shown to be completely equivalent. This added a good deal of force to the viewpoint, which became known as the *Church–Turing Thesis*, that the Turing machine concept (or equivalent) actually does define what, mathematically, we mean by an algorithmic (or effective or recursive or mechanical) procedure. Now that high-speed electronic computers have become such a familiar part of our lives, not many people seem to feel the need to question this thesis in its original form. Instead, some attention has been turned to the matter of whether actual *physical* systems (presumably including human brains)—subject as they are to precise *physical* laws—are able to perform more than, less than, or precisely the same logical and mathematical operations as Turing machines. For my own part, I am very happy to accept the original *mathematical* form of the Church–Turing Thesis. Its relation to the behaviour of actual physical systems, on the other hand, is a separate issue which will be a major concern for us later in this book.

Numbers other than natural numbers

In the discussion given above, we considered operations on *natural numbers*, and we noted the remarkable fact that single Turing machines can handle natural numbers of arbitrarily large size, despite the fact that each machine has a fixed *finite* number of distinct internal states. However, one often needs to work with more complicated kinds of number than this, such as negative numbers, fractions, or infinite decimals. Negative numbers and fractions (e.g. numbers like −597/26) can be easily handled by Turing machines, and the numerators and denominators can be as large as we like. All we need is some suitable coding for the signs '−' and '/', and this can easily be done using the expanded binary notation described earlier (for example, '**3**' for '−' and '**4**' for '/'—coded as `1110` and `11110`, respectively, in the expanded binary notation). Negative numbers and fractions are thus handled in terms of finite sets of natural numbers, so with regard to general questions of computability they give us nothing new.

Likewise, *finite* decimal expressions of unrestricted length give us nothing new, since these are just particular cases of fractions. For example, the finite decimal approximation to the irrational number π, given by 3.14159265, is simply the fraction 314159265/100000000. However, *infinite* decimal expressions, such as the *full* non-terminating expansion

$$\pi = 3.14159265358979 \ldots$$

present certain difficulties. Neither the input nor the output of a Turing machine can, strictly speaking, be an infinite decimal. One might think that we could find a Turing machine to churn out *all* the successive digits, 3,1,4,1,5,9, . . ., of the above expansion for π one after the other on the output tape, where we simply allow the machine to run on forever. But this is *not allowed* for a Turing machine. We must wait for the machine to halt (indicated by the bell ringing!) before we are allowed to examine the output. So long as the machine has not reached a STOP order, the output is subject to possible change and so cannot be trusted. After it has reached STOP, on the other hand, the output is necessarily finite.

There is, however, a procedure for *legitimately* making a Turing machine produce digits one after the other, in a way very similar to this. If we wish to generate an infinite decimal expansion, say that of π, we could have a Turing machine produce the whole-number part, 3, by making the machine act on 0, then we could produce the first decimal digit, 1, by making the machine act on 1, then the second decimal digit, 4, by making it act on 2, then the third, 1, by making it act on 3, and so on. In fact a Turing machine for producing the entire decimal expansion of π in *this* sense certainly *does* exist, though it would be a little complicated to work it out explicitly. A similar remark applies to many other irrational numbers, such as $\surd 2 = 1.414213562 \ldots$ It turns out, however, that some irrationals (remarkably) cannot be produced by any Turing machine at all, as we shall see in the next chapter. The numbers that *can* be generated in this way are called *computable* (Turing 1937). Those that cannot (actually the vast majority!) are *non*-computable. I shall come back to this matter, and related issues, in later chapters. It will have some relevance for us in relation to the question of whether an *actual physical object* (e.g. a human brain) can, according to our physical theories, be adequately described in terms of computable mathematical structures.

The issue of computability is an important one generally in mathematics. One should not think of it as a matter which applies just to *numbers* as such. One can have Turing machines which operate directly on *mathematical formulae*, such as algebraic or trigonometric expressions, for example, or which carry through the formal manipulations of the calculus. All that one needs is some form of precise coding into sequences of 0s and 1s, of all the mathematical symbols that are involved, and then the Turing machine concept can be applied. This, after all, was what Turing had in mind in his attack on the *Entscheidungsproblem*, which asks for an algorithmic procedure for answering mathematical questions of a *general* nature. We shall be coming back to this shortly.

The universal Turing machine

I have not yet described the concept of a *universal* Turing machine. The principle behind this is not too difficult to give, even though the details are complicated. The basic idea is to code the list of instructions for an arbitrary Turing machine T into a string of 0s and 1s that can be represented on a tape. This tape is then used as the initial part of the input for some *particular* Turing machine U—called a universal Turing machine—which then acts on the remainder of the input just as T would have done. The universal Turing machine is a universal mimic. The initial part of the tape gives the universal machine U the full information that it needs for it to imitate any given machine T exactly!

To see how this works we first need a systematic way of *numbering* Turing machines. Consider the list of instructions defining some particular Turing machine, say one of those described above. We must code this list into a string of 0s and 1s according to some precise scheme. This can be done with the aid of the 'contraction' procedure that we adopted before. For if we represent the respective symbols R, L, STOP, the arrow (→), and the comma as, say, the numerals 2, 3, 4, 5, and 6, we can code them as contractions, by 110, 1110, 11110, 111110, and 1111110. Then the digits 0 and 1, coded as 0 and 10, respectively, can be used for the actual strings of these symbols appearing in the table. We do not need to have a different notation to distinguish the large figures 0 and 1 in the Turing machine table from the smaller arabic ones, since the position of the large digits at the end of the binary numbering is sufficient to distinguish them from the others. Thus, for example, 1101 would be read as the binary number 1101 and coded on the tape as 1010010. In particular, 00 would be read as 00, which can, without ambiguity, be coded 0, or as a symbol omitted altogether. We can economize considerably by not actually bothering to code any arrow nor any of the symbols immediately preceding it, relying instead upon the numerical ordering of instructions to specify what those symbols must be—although to adopt this procedure we must make sure that there are no gaps in this ordering, supplying a few extra 'dummy' orders where required. (For example, the Turing machine **XN+1** has no order telling us what to do with 1100, since this combination never occurs in the running of the machine, so we must insert a 'dummy' order, say 110→0R, which can be incorporated into the list without changing anything. Similarly we should insert 101→0R into the machine **XN×2**.) Without such 'dummies', the coding of the subsequent orders in the list would be spoiled. We do not actually need the comma at the end of each instruction, as it turns out, since the symbols L or R suffice to separate the instructions from one another. We therefore simply adopt the following coding:

0 for 0 or 0, 10 for 1 or 1, 110 for R, 1110 for L,
11110 for STOP

As an example, let us code the Turing machine **XN+1** (with the 110→0R instruction inserted). Leaving out the arrows, the digits immediately preceding them, and also the commas, we have

00R 11R 00R 101R 110L 101R 01STOP 1000L 1011L 1001L
1100R 101R 00R 1111R 111R 1110R.

We can improve on this by leaving out every 00 and replacing each 01, by simply 1, in accordance with what has been said earlier, to get

R11RR101R110L101R1STOP1000L1011L1001L1100R101RR1111R111R1110R.

This is coded as the tape sequence

11010101101101001011010100111010010110101111010000111O
10010101110100010111010100011010010110110101010101101O
10101101010100110.

As two further minor economies, we may as well always delete the initial 110 (together with the infinite stretch of blank tape that precedes it) since this denotes 0R, which represents the initial instruction 00→0R that I have been implicitly taking to be common to *all* Turing machines—so that the device can start arbitrarily far to the left of the marks on the tape and run to the right until it comes up to the first mark—and we may as well always delete the final 110 (and the implicit infinite sequence of 0s which is assumed to follow it) since all Turing machines must have their descriptions ending this way (because they all end with R, L, or STOP). The resulting *binary number* is the *number* of the Turing machine, which in the case of **XN+1** is:

101011011010010110101001110100101101011110100001110100
101011101000101110101000110100101101101010101011010101
01101010100.

In standard denary notation, this particular number is

450 813 704 461 563 958 982 113 775 643 437 908.

We sometimes loosely refer to the Turing machine whose number is n as the nth Turing machine, denoted T_n. Thus **XN+1** is the 450813704461563958982113775643437908th Turing machine!

It is a striking fact that we appear to have to go this far along the 'list' of Turing machines before we find one that even performs so trivial operation as adding one (in the expanded binary notation) to a natural number! (I do not think that I have been grossly inefficient in my coding, though I can see room for some minor improvements.) Actually there are some Turing machines with smaller numbers which are of interest. For example, **UN+1** has the binary number

101011010111101010

which is merely 177642 in denary notation! Thus the particularly trivial Turing machine **UN+1**, which merely places an additional 1 at the end of a sequence of 1s, is the 177642nd Turing machine. For curiosity's sake, we may note that 'multiplying by two' comes somewhere between these two in the list of Turing machines, in either notation, for we find that the number for **XN×2** is 10389728107 while that of **UN × 2** is 1492923420919872026917547669.

It is perhaps not surprising to learn, in view of the sizes of these numbers, that the vast majority of natural numbers do not give working Turing machines at all. Let us list the first thirteen Turing machines according to this numbering:

T_0:	00→00R,	01→00R,	
T_1:	00→00R,	01→00L,	
T_2:	00→00R,	01→01R,	
T_3:	00→00R,	01→00STOP,	
T_4:	00→00R,	01→10R,	
T_5:	00→00R,	01→01L,	
T_6:	00→00R,	01→00R,	10→00R,
T_7:	00→00R,	01→???,	
T_8:	00→00R,	01→100R,	
T_9:	00→00R,	01→10L,	
T_{10}:	00→00R,	01→11R,	
T_{11}:	00→00R,	01→01STOP,	
T_{12}:	00→00R,	01→00R,	10→00R.

Of these, T_0 simply moves on to the right obliterating everything that it encounters, never stopping and never turning back. The machine T_1 ultimately achieves the same effect, but in a clumsier way, jerking backwards after it obliterates each mark on the tape. Like T_0, the machine T_2 also moves on endlessly to the right, but is more respectful, simply leaving everything on the tape just as it was before. None of these is any good as a Turing machine since none of them ever stops. T_3 is the first respectable machine. It indeed stops, modestly, after changing the first (leftmost) 1 into a 0.

T_4 encounters a serious problem. After it finds its first 1 on the tape it enters an internal state for which there is no listing, so it has no instructions as to what to do next. T_8, T_9, and T_{10} encounter the same problem. The difficulty with T_7 is even more basic. The string of 0s and 1s which codes it involves a sequence of *five* successive 1s: 110111110. There is no interpretation for such a sequence, so T_7 will get stuck as soon as it finds its first 1 on the tape. (I shall refer to T_7, or any other machine T_n for which the binary expansion of n contains a sequence of more than four 1s as being *not correctly specified*.) The machines T_5, T_6, and T_{12} encounter problems similar to those of T_0, T_1, and T_2. They simply run on indefinitely without ever stopping. All of the machines T_0, T_1, T_2, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, and T_{12} are duds! Only T_3 and T_{11} are working Turing machines, and not very interesting ones at that. T_{11} is even more modest than T_3. It stops at its first encounter with a 1 and it doesn't change a thing!

We should note that there is also a redundancy in our list. The machine T_{12} is identical with T_6, and also identical in action with T_0, since the internal state 1 of T_6 and T_{12} is never entered. We need not be disturbed by this redundancy, nor by the proliferation of dud Turing machines in the list. It would indeed be possible to improve our coding so that a good many of the duds are removed and the redundancy considerably reduced. All this would be at the expense of complicating our poor universal Turing machine which has to decipher the code and pretend to be the Turing machine T_n whose number n it is reading. This might be worth doing if we could remove *all* the duds (or the redundancy). But this is *not* possible, as we shall see shortly! So let us leave our coding as it is.

It will be convenient to interpret a tape with its succession of marks, e.g.

. . . 0001101110010000 . . .

as the binary representation of some number. Recall that the 0s continue indefinitely at both ends, but that there is only a finite number of 1s. I am also assuming that the number of 1s is *non-zero* (i.e. there is at least one 1). We could choose to read the finite string of symbols between the first and last 1 (inclusive), which in the above case is

110111001,

as the binary description of a natural number (here 441, in denary notation). However, this procedure would only give us *odd* numbers (numbers whose binary representation ends in a 1) and we want to be able to represent *all* natural numbers. Thus we adopt the simple expedient of removing the final 1 (which is taken to be just a marker indicating the termination of the expression) and reading what is left as a binary number.[5] Thus, for the above example, we have the binary number

11011100,

which, in denary notation, is 220. This procedure has the advantage that zero is also represented as a marked tape, namely

. . . 0000001000000. . . .

Let us consider the action of the Turing machine T_n on some (finite) string of 0s and 1s on a tape which we feed in on the right. It will be convenient to regard this string also as the binary representation of some number, say *m*, according to the scheme given above. Let us assume that after a succession of steps the machine T_n finally comes to a halt (i.e. reaches STOP). The string of binary digits that the machine has now produced at the left is the answer to the calculation. Let us also read this as the binary representation of a number in the same way, say *p*. We shall write this relation, which expresses the fact that when the n^{th} Turing machine acts on *m* it produces *p*, as:

$$T_n(m) = p.$$

Now let us look at this relation in a slightly different way. We think of it as expressing one particular operation which is applied to the *pair* of numbers *n* and *m* in order to produce the number *p*. (Thus: given the *two* numbers *n* and *m*, we can work out from them what *p* is by seeing what the *n*th Turing machine does to *m*.) This particular operation is an entirely algorithmic procedure. It can therefore be carried out by *one particular* Turing machine *U*; that is, *U* acts on the *pair* (*n*, *m*) to produce *p*. Since the machine *U* has to act on *both* of *n* and *m* to produce the single result *p*, we need some way of coding the pair (*n*, *m*) on the *one* tape. For this, we can assume that *n* is written out in ordinary binary notation and then immediately terminated by the sequence 111110. (Recall that the binary number of every correctly specified Turing machine is a sequence made up just of 0s, 10s, 110s, 1110s, and 11110s, and it therefore contains no sequence of more than four 1s. Thus if T_n is a correctly specified machine, the occurrence of 111110 indeed signifies that the description of the number *n* is finished with.) Everything following it is to be simply the tape represented by *m* according to our above prescription (i.e. the binary number *m* immediately followed by 1000 . . .). Thus this second part is simply the tape that T_n is supposed to act on.

As an example, if we take $n = 11$ and $m = 6$, we have, for the tape on which *U* has to act, the sequence of marks

. . . 000101111111011010000. . . .

This is made up as follows:

. . . 0000 (initial blank tape)
1011 (binary representation of 11)
111110 (terminates n)
110 (binary representation of 6)
10000 . . . (remainder of tape)

What the Turing machine U would have to do, at each successive step of the operation of T_n on m, would be to examine the structure of the succession of digits in the expression for n so that the appropriate replacement in the digits for m (i.e. T_n's 'tape') can be made. In fact it is not difficult in principle (though decidedly tedious in practice) to see how one might actually construct such a machine. Its own list of instructions would simply be providing a means of reading the appropriate entry in that 'list' which is encoded in the number n, at each stage of application to the digits of the 'tape', as given by m. There would admittedly be a lot of dodging backwards and forwards between the digits of m and those of n, and the procedure would tend to be exceedingly slow. Nevertheless, a list of instructions for such a machine can certainly be provided; and we call such a machine a *universal* Turing machine. Denoting the action of this machine on the pair of numbers n and m by $U(n,m)$, we have:

$$U(n,m) = T_n(m)$$

for each (n, m) for which T_n is a correctly specified Turing machine.[6] The machine U, when first fed with the number n, precisely imitates the nth Turing machine!

Since U is a Turing machine, it will itself have a number; i.e. we have

$$U = T_u,$$

for some number u. How big is u? In fact we can take *precisely*

u = 7244855335339317577198395039615711237952360672556559631108144796606505059404241090310483613632359365644443458382226883278767626556
144692814117715017842551707554085657689753346356942478488597046934725739988582283827795294683460521061169835945938791885546326440925525505820555989451890716537414896033096753020431553625034984529832320651583047664142130708819329717234151056980262734686429921838172157333482823073453713421475059740345184372359593090640024321077342178851492760797597634415123079586396354492269159479654614711345700145048
167337562172573464522731054482980784965126988788964569760906634204477989021914437932830019493570963921703904833270882596201301773727202718625919914428275437422351355675134084222299889374410534305471044368695876405178128019437530813870639942772823156425289237514565443899052780793241144826142357286193118332610656122755531810207511085337633806031082361675045635852164214869542347187426437544428790062485

```
8270912404220765387542644541334517485662915742999095026230097337381
3772416217274772361020678685400289356608569682262014198248621698902
6091309402985706001743006700868967590344734174127874255812015493663
9389969058177385916540553567040928213322216314109787108145997866959
9704509681841906299443656015145490488092208448003482249207730403043
1884298993931352668823496621019471619107014619685231928474820344958
9770955356110702758174873332729667899879847328409819076485127263100
1740166787363477605857245036964434897992034489997455662402937487668
8397514044516657077500605138839916688140725455446652220507242623923
7921152531816251253630509317286314220040645713052758023076651833519
9568913974813750492642960501001365198018694563949
```

(or some other possibility of at least that kind of size). This number no doubt seems alarmingly large! Indeed it *is* alarmingly large, but I have not been able to see how it could have been made significantly smaller. The coding procedures and specifications that I have given for Turing machines are quite reasonable and simple ones, yet one is inevitably led to a number of this kind of size for the coding of an actual universal Turing machine.[7]

I have said that all modern general purpose computers are, in effect, universal Turing machines. I do not mean to imply that the logical design of such computers need resemble at all closely the kind of description for a universal Turing machine that I have just given. The point is simply that, by supplying any universal Turing machine first with an appropriate program (initial part of the input tape), it can be made to mimic the behaviour of any Turing machine whatever! In the description above, the program simply takes the form of a single number (the number n), but other procedures are possible, there being many variations on Turing's original theme. In fact in my own descriptions I have deviated somewhat from those that Turing originally gave. None of these differences is important for our present needs.

The insolubility of Hilbert's problem

We now come to the purpose for which Turing originally put forward his ideas, the resolution of Hilbert's broad-ranging *Entscheidungsproblem*: is there some mechanical procedure for answering all mathematical problems, belonging to some broad, but well-defined class? Turing found that he could phrase his version of the question in terms of the problem of deciding whether or not the n^{th} Turing machine would actually ever *stop* when acting on the number m. This problem was referred to as the *halting problem*. It is an easy matter to construct an instruction list for which the machine will not stop for *any* number m (for example, $n = 1$ or 2, as given above, or any other case where there are no STOP instructions whatever). Also there are many instruction lists for which the machine would always stop, whatever number it is given (e.g. $n = 11$); and some machines would stop for some numbers but not for others. One could fairly say that a putative algorithm is not much use when it runs forever without stopping. That is no algorithm at all. So an important question is to be able to decide whether or not T_n applied to m actually ever gives any answer! If it does *not* (i.e. if the calculation does *not* stop), then I shall write

$$T_n(m) = \square.$$

(Included in this notation would be those situations where the Turing machine runs into a problem at some stage because it finds no appropriate instruction to tell it what to do—as with the dud machines such as T_4 and T_7 considered above. Also, unfortunately, our seemingly successful machine T_3 must now also be considered a dud: $T_3(m) = \square$, because the result of the action of T_3 is always just blank tape, whereas we need at least one 1 in the output in order that the result of the calculation be assigned a number! The machine T_{11} is, however, legitimate since it produces a single 1. This output is the tape numbered 0, so we have $T_{11}(m) = 0$ for all m.)

It would be an important issue in mathematics to be able to decide when Turing machines stop. For example, consider the equation:

$$(x + 1)^{w+3} + (y + 1)^{w+3} = (z + 1)^{w+3}.$$

(If technical mathematical equations are things that worry you, don't be put off! This equation is being used only as an example, and there is no need to understand it in detail.) This particular equation relates to a famous unsolved problem in mathematics—perhaps the most famous of all. The problem is this: is there *any* set of natural numbers w, x, y, z for which this equation is satisfied? The famous statement known as 'Fermat's last theorem', made in the margin of Diophantus's *Arithmetica*, by the great seventeenth century French mathematician Pierre de Fermat (1601–1665), is the assertion that the equation is *never* satisfied.[*8] Though a lawyer by profession (and a contemporary of Descartes), Fermat was the finest mathematician of his time. He claimed to have 'a truly wonderful proof' of his assertion, which the margin was too small to contain; but to this day no-one has been able to reconstruct such a proof nor, on the other hand, to find any counter-example to Fermat's assertion!

It is clear that *given* the quadruple of numbers (w, x, y, z), it is a mere matter of computation to decide whether or not the equation holds. Thus we could imagine a computer algorithm which runs through all the quadruples of

[*] Recall that by the *natural* numbers we mean 0, 1, 2, 3, 4, 5, 6, . . . The reason for the '$x + 1$' and '$w + 3$', etc., rather than the more familiar form ($x^w + y^w = z^w$; $x, y, z > 0$, $w > 2$) of the Fermat assertion, is that we are allowing *all* natural numbers for x, w, etc., starting with zero.

numbers one after the other, and stops only when the equation is satisfied. (We have seen that there are ways of coding finite sets of numbers, in a computable way, on a single tape, i.e. simply as single numbers, so we can 'run through' all the quadruples by just following the natural ordering of these single numbers.) If we could establish that this algorithm does *not* stop, then we would have a proof of the Fermat assertion.

In a similar way it is possible to phrase many other unsolved mathematical problems in terms of the Turing machine halting problem. Such an example is the 'Goldbach conjecture', which asserts that every even number greater than 2 is the sum of two prime numbers.* It is an algorithmic process to decide whether or not a given natural number is prime since one needs only to test its divisibility by numbers *less* than itself, a matter of only *finite* calculation. We could devise a Turing machine which runs through the even numbers 6, 8, 10, 12, 14, . . . trying all the different ways of splitting them into pairs of odd numbers

$$6 = 3 + 3, \qquad 8 = 3 + 5, \qquad 10 = 3 + 7 = 5 + 5, \qquad 12 = 5 + 7,$$
$$14 = 3 + 11 = 7 + 7, \ldots$$

and testing to make sure that, for *each* such even number, it splits to *some* pair for which *both* members are prime. (Clearly we need not test pairs of *even* summands, except 2 + 2, since all primes except 2 are odd.) Our machine is to stop only when it reaches an even number for which *none* of the pairs into which that number splits consists of two primes. In that case we should have a counter-example to the Goldbach conjecture, namely an even number (greater than 2) which is *not* the sum of two primes. Thus if we could decide whether or not this Turing machine ever stops, we should have a way of deciding the truth of the Goldbach conjecture also.

A natural question arises: how are we to decide whether or not any particular Turing machine (when fed with some specific input) will ever stop? For many Turing machines this might not be hard to answer; but occasionally, as we have seen above, the answer could involve the solution of an outstanding mathematical problem. So, is there some *algorithmic* procedure for answering the general question—the halting problem—completely automatically? Turing showed that indeed there is not.

His argument was essentially the following. We first suppose that, on the contrary, there *is* such an algorithm.† Then there must be some Turing machine H which 'decides' whether or not the nth Turing machine, when acting on the number m, eventually stops. Let us say that it outputs the tape numbered 0 if it does not stop and 1 if it does:

* Recall that the *prime* numbers 2, 3, 5, 7, 11, 13, 17, . . are those natural numbers divisible, separately, only by themselves and by unity. Neither 0 nor 1 is considered to be a prime.

† This is the familiar—and powerful—mathematical procedure known as *reductio ad absurdum*, whereby one first assumes that what one is trying to prove is false; and from that one derives a contradiction, thus establishing that the required result is actually *true*.

$$H(n;m) = \begin{cases} 0 & \text{if } T_n(m) = \square \\ 1 & \text{if } T_n(m) \text{ stops.} \end{cases}$$

Here, one might take the coding of the pair (n, m) to follow the same rule as we adopted for the universal machine U. However this could run into the technical problem that for some number n (e.g. $n = 7$), T_n is not correctly specified; and the marker 111110 would be inadequate to separate n from m on the tape. To obviate this problem, let us assume that n is coded using the *expanded* binary notation rather than just the binary notation, with m in ordinary binary, as before. Then the marker 110 will actually be sufficient to separate n from m. The use of the *semicolon* in $H(n; m)$, as distinct from the *comma* in $U(n, m)$, is to indicate this change.

Now let us imagine an infinite array, which lists all the outputs of all possible Turing machines acting on all the possible different inputs. The nth row of the array displays the output of the nth Turing machine, as applied to the various inputs 0, 1, 2, 3, 4, . . . :

$m \rightarrow$	0	1	2	3	4	5	6	7	8	. . .
n ↓										
0	□	□	□	□	□	□	□	□	□	. . .
1	0	0	0	0	0	0	0	0	0	. . .
2	1	1	1	1	1	1	1	1	1	. . .
3	0	2	0	2	0	2	0	2	0	. . .
4	1	1	1	1	1	1	1	1	1	. . .
5	0	□	0	□	0	□	0	□	0	. . .
6	0	□	1	□	2	□	3	□	4	. . .
7	0	1	2	3	4	5	6	7	8	. . .
8	□	1	□	□	1	□	□	□	1	. . .
.	.				.				.	. . .
.	.				.				.	
.	.				.				.	
197	2	3	5	7	11	13	17	19	23	. . .
.	.				.				.	
.	.				.				.	
.	.				.				.	

In the above table I have cheated a little, and not listed the Turing machines as they are *actually* numbered. To have done so would have yielded a list that looks much too boring to begin with, since all the machines for which n is less than 11 yield nothing but □s, and for $n = 11$ itself we get nothing but 0s. In order to make the list look initially more interesting, I have assumed that some much more efficient coding has been achieved. In fact I have simply made up the entries fairly randomly, just to give some kind of impression as to what its general appearance could be like.

I am not asking that we have actually *calculated* this array, say by some algorithm. (In fact, there is no such algorithm, as we shall see in a moment.) We are just supposed to *imagine* that the *true* list has somehow been laid out before us, perhaps by God! It is the occurrence of the □s which would cause the difficulties if we were to attempt to calculate the array, for we might not know for sure when to place a □ in some position since those calculations simply run on forever!

However, we *could* provide a calculational procedure for generating the table if we were allowed to use our putative H, for H would tell us where the □s actually occur. But instead, let us use H to *eliminate* every □ by replacing each occurrence with 0. This is achieved by preceding the action of T_n on m by the calculation $H(n; m)$; then we allow T_n to act on m only if $H(n; m) = 1$ (i.e. only if the calculation $T_n(m)$ actually gives an answer), and simply write 0 if $H(n; m) = 0$ (i.e. if $T_n(m) = \square$). We can write our new procedure (i.e. that obtained by preceding $T_n(m)$ by the action of $H(n; m)$) as

$$T_n(m) \times H(n; m).$$

(Here I am using a common mathematical convention about the ordering of mathematical operations: the one on the *right* is to be performed *first*. Note that, symbolically, we have $\square \times 0 = 0$.)

The table for this now reads:

$n \downarrow$ / $m \rightarrow$	0	1	2	3	4	5	6	7	8	...
0	0	0	0	0	0	0	0	0	0	...
1	0	0	0	0	0	0	0	0	0	...
2	1	1	1	1	1	1	1	1	1	...
3	0	2	0	2	0	2	0	2	0	...
4	1	1	1	1	1	1	1	1	1	...
5	0	0	0	0	0	0	0	0	0	...
6	0	0	1	0	2	0	3	0	4	...
7	0	1	2	3	4	5	6	7	8	...
8	0	1	0	0	1	0	0	0	1	...
.	.				.				.	
.	.				.				.	
.	.				.				.	

Note that, assuming H exists, the rows of this table consist of *computable sequences*. (By a computable sequence I mean an infinite sequence whose successive values can be generated by an algorithm; i.e. there is some Turing machine which, when applied to the natural numbers $m = 0, 1, 2, 3, 4, 5, \ldots$ in turn, yields the successive members of the sequence.) Now, we take note of two facts about this table. In the first place, *every* computable sequence of natural numbers must appear somewhere (perhaps many times over) amongst its rows. This property was already true of the original table with its □s. We have simply *added* some rows to replace the 'dud' Turing machines (i.e. the ones which produce at least one □). In the second place, the assumption having been made that the Turing machine H actually exists, the table has been *computably generated* (i.e. generated by some definite algorithm), namely by the procedure $T_n(m) \times H(n; m)$. That is to say, there is some Turing machine Q which, when acting on the pair of numbers (n, m) produces the appropriate entry in the table. For this, we may code n and m on Q's tape in the same way as for H, and we have

$$Q(n; m) = T_n(m) \times H(n; m).$$

We now apply a variant of an ingenious and powerful device, the 'diagonal slash' of Georg Cantor. (We shall be seeing the original version of Cantor's diagonal slash in the next chapter.) Consider the elements of the main diagonal, marked now with **bold** figures:

$$\begin{array}{ccccccccc}
\mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 2 & 0 & \mathbf{2} & 0 & 2 & 0 & 2 & 0 \\
1 & 1 & 1 & 1 & \mathbf{1} & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 2 & 0 & \mathbf{3} & 0 & 4 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & \mathbf{7} & 8 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \mathbf{1} \\
\vdots & & & & \vdots & & & & \vdots
\end{array}$$

These elements provide some sequence 0, 0, 1, 2, 1, 0, 3, 7, 1, . . . to each of whose terms we now add 1:

$$1, 1, 2, 3, 2, 1, 4, 8, 2, \ldots$$

This is clearly a computable procedure and, given that our table was computably generated, it provides us with some new computable sequence, in fact with the sequence $1 + Q(n; n)$, i.e.

$$1 + T_n(n) \times H(n; n)$$

(since the diagonal is given by making m equal to n). But our table contains *every* computable sequence, so our new sequence must be somewhere in the list. Yet this cannot be so! For our new sequence differs from the first row in the first entry, from the second row in the second entry, from the third row in the third entry, and so on. This is manifestly a contradiction. It is the contradiction which establishes what we have been trying to prove, namely that the Turing machine H does not in fact exist! *There is no universal algorithm for deciding whether or not a Turing machine is going to stop.*

Another way of phrasing this argument is to note that, on the assumption that H exists, there is some Turing machine number, say k, for the algorithm (diagonal process!) $1 + Q(n;\, n)$, so we have

$$1 + T_n(n) \times H(n; n) = T_k(n).$$

But if we substitute $n = k$ in this relation we get

$$1 + T_k(k) \times H(k; k) = T_k(k).$$

This is a contradiction because if $T_k(k)$ stops we get the impossible relation

$$1 + T_k(k) = T_k(k)$$

(since $H(k;\, k) = 1$), whereas if $T_k(k)$ does not stop (so $H(k;\, k) = 0$) we have the equally inconsistent

$$1 + 0 = \square.$$

The question of whether or not a particular Turing machine stops is a perfectly well-defined piece of mathematics (and we have already seen that, conversely, various significant mathematical questions can be phrased as the stopping of Turing machines). Thus, by showing that no algorithm exists for deciding the question of the stopping of Turing machines, Turing showed (as had Church, using his own rather different type of approach) that there can be no general algorithm for deciding mathematical questions. Hilbert's *Entscheidungsproblem* has no solution!

This is not to say that in any *individual* case we may not be able to decide the truth, or otherwise, of some particular mathematical question; or decide whether or not some given Turing machine will stop. By the exercise of ingenuity, or even of just common sense, we may be able to decide such a question in a given case. (For example, if a Turing machine's instruction list contains *no* STOP order, or contains *only* STOP orders, then common sense alone is sufficient to tell us whether or not it will stop!) But there is no one algorithm that works for *all* mathematical questions, nor for *all* Turing machines and all numbers on which they might act.

It might seem that we have now established that there are at least *some* undecidable mathematical questions. However, we have done nothing of the kind! We have *not* shown that there is some especially awkward Turing machine table for which, in some absolute sense, it is impossible to decide whether or not the machine stops when it is fed with some especially awkward number—indeed, quite the reverse, as we shall see in a moment. We have said nothing whatever about the insolubility of *single* problems, but only about the *algorithmic* insolubility of *families* of problems. In any single case the answer is either 'yes' or 'no', so there certainly *is* an algorithm for deciding that particular case, namely the algorithm that simply says 'yes', when presented with the problem, or the one that simply says 'no', as the case may be! The difficulty is, of course, that we may not know *which* of these algorithms to use. That is a question of deciding the mathematical truth of a single statement, not the systematic decision problem for a family of statements. It is important to realize that algorithms do not, in themselves, decide mathematical truth. The *validity* of an algorithm must always be established by external means.

How to outdo an algorithm

This question of deciding the truth of mathematical statements will be returned to later, in connection with Gödel's theorem (see Chapter 4). For the moment, I wish to point out that Turing's argument is actually a lot more constructive and less negative than I have seemed to imply so far. We have certainly *not* exhibited a specific Turing machine for which, in some absolute sense, it is undecidable whether or not it stops. Indeed, if we examine the argument carefully, we find that our very procedure has actually implicitly *told us the answer* for the seemingly 'especially awkward' machines that we construct using Turing's procedure!

Let us see how this comes about. Suppose we have some algorithm which is *sometimes* effective for telling us when a Turing machine will not stop. Turing's procedure, as outlined above, will *explicitly* exhibit a Turing machine calculation for which that particular algorithm is not able to decide whether or not the calculation stops. However, in doing so, it actually enables *us* to see the answer in this case! The particular Turing machine calculation that we exhibit will indeed *not* stop.

To see how this arises in detail suppose we have such an algorithm that is sometimes effective. As before, we denote this algorithm (Turing machine) by H, but now we allow that the algorithm may not always be sure to tell us that a Turing machine will actually not stop:

$$H(n; m) = \begin{cases} 0 \textit{ or } \square & \text{if } T_n(m) = \square \\ 1 & \text{if } T_n(m) \text{ stops,} \end{cases}$$

so $H(n; m) = \square$ is a possibility when $T_n(m) = \square$. Many such algorithms $H(n; m)$ actually exist. (For example, $H(n; m)$ could simply produce a 1 as soon as $T_n(m)$ stops, although *that* particular algorithm would hardly be of much practical use!)

We can follow through Turing's procedure in detail just as given above, except that instead of replacing *all* the $\square$s by 0s, we now have some $\square$s left. As before, our diagonal procedure has provided us with

$$1 + T_n(n) \times H(n; n),$$

as the *n*th term on the diagonal. (We shall get a $\square$ whenever $H(n; n) = \square$. Note that $\square \times \square = \square$, $1 + \square = \square$.) This is a perfectly good computation, so it is achieved by some Turing machine, say the *k*th one, and now we *do* have

$$1 + T_n(n) \times H(n; n) = T_k(n).$$

We look at the *k*th diagonal term, i.e. $n = k$, and obtain

$$1 + T_k(k) \times H(k; k) = T_k(k).$$

If the computation $T_k(k)$ stops, we have a contradiction (since $H(k; k)$ is supposed to be 1 whenever $T_k(k)$ stops, and the equation then gives inconsistency: $1 + T_k(k) = T_k(k)$). Thus $T_k(k)$ cannot stop, i.e.

$$T_k(k) = \square.$$

But the algorithm cannot 'know' this, because if it gave $H(k; k) = 0$, we should again have a contradiction (symbolically, we should have the invalid relation: $1 + 0 = \square$).

Thus, if we can find *k* we shall know how to construct our specific calculation to defeat the algorithm but for which *we* know the answer! How do we find *k*? That's hard work. What we have to do is to look in detail at the construction of $H(n; m)$ and of $T_n(m)$ and then see in detail how $1 + T_n(n) \times H(n; n)$ acts as a Turing machine. We find the number of this Turing machine, which is *k*. This would certainly be complicated to carry out in detail, but it could be done.* Because of the complication, we would not be at all interested in the calculation $T_k(k)$ were it not for the fact that we have specially produced it in order to defeat the algorithm *H*! What is important is that we have a well-defined procedure, whichever *H* is given to us, for finding a corresponding *k* for which *we* know that $T_k(k)$ defeats *H*, and for which we can therefore do better than the algorithm. Perhaps that comforts us a little if we think we are better than mere algorithms!

In fact the procedure is so well defined that we could find an *algorithm* for generating *k*, given *H*. So, before we get too complacent, we have to realize that *this* algorithm can improve[9] on *H* since, in effect, it 'knows' that $T_k(k) = \square$—or does it? It has been helpful in the above description to use the anthropomorphic term 'know' in reference to an algorithm. However, is it not *we* who are doing the 'knowing', while the algorithm just follows the rules we have told it to follow? Or are we ourselves merely following rules that we have been programmed to follow from the construction of our brains and from our environment? The issue is not really simply one of algorithms, but also a question of how one judges what is true and what is not true. These are central issues that we shall have to return to later. The question of mathematical truth (and its non-algorithmic nature) will be considered in Chapter 4. At least we should now have some feeling about the *meanings* of

* In fact, the hardest part of this is already achieved by the construction of the universal Turing machine *U* above, since this enables us to write down $T_n(n)$ as a Turing machine acting on *n*.

the terms 'algorithm' and 'computability', and an understanding of some of the related issues.

Church's lambda calculus

The concept of computability is a very important and beautiful mathematical idea. It is also a remarkably recent one—as things of such a fundamental nature go in mathematics—having been first put forward in the 1930s. It is an idea which cuts across *all* areas of mathematics (although it may well be true that most mathematicians do not, as yet, often worry themselves about computability questions). The power of the idea lies partly in the fact that some well-defined operations in mathematics are actually *not* computable (like the stopping, or otherwise, of a Turing machine; we shall see other examples in Chapter 4). For if there were no such non-computable things, the concept of computability would not have much mathematical interest. Mathematicians, after all, like puzzles. It can be an intriguing puzzle for them to decide, of some mathematical operation, whether or not it is computable. It is especially intriguing because the general solution of *that* puzzle is itself non-computable!

One thing should be made clear. Computability is a genuine 'absolute' mathematical concept. It is an abstract idea which lies quite beyond any particular realization in terms of the 'Turing machines' as I have described them. As I have remarked before, we do not need to attach any particular significance to the 'tapes' and 'internal states', etc., which characterize Turing's ingenious but particular approach. There are also other ways of expressing the idea of computability, historically the first of these being the remarkable 'lambda calculus' of the American logician Alonzo Church, with the assistance of Stephen C. Kleene. Church's procedure was quite different, and distinctly more abstract from that of Turing. In fact, in the form that Church stated his ideas, there is rather little obvious connection between them and anything that one might call 'mechanical'. The key idea lying behind Church's procedure is, indeed, *abstract* in its very essence—a mathematical operation that Church actually referred to as 'abstraction'.

I feel that it is worth while to give a brief description of Church's scheme not only because it emphasizes that computability is a mathematical idea, independent of any particular concept of computing machine, but also because it illustrates the power of abstract ideas in mathematics. The reader who is not readily conversant with mathematical ideas, nor intrigued by such things for their own sake, may, at this stage, prefer to move on to the next chapter—and there would not be significant loss in the flow of argument. Nevertheless, I believe that such readers might benefit by bearing with me for a while longer, and thus witnessing some of the magical economy of Church's scheme (see Church 1941).

In this scheme one is concerned with a 'universe' of objects, denoted by say

$$a, b, c, d, \ldots, z, a', b', \ldots, z', a'', b'', \ldots, a''', \ldots, a'''', \ldots$$

each of which stands for a mathematical operation or *function*. (The reason for the primed letters is simply to allow an unlimited supply of symbols to denote such functions.) The 'arguments' of these functions—that is to say, the things on which these functions act—are other things of the same kind, i.e. also functions. Moreover, the result (or 'value') of one such function acting on another is to be again a function. (There is, indeed, a wonderful economy of concepts in Church's system.) Thus, when we write*

$$a = bc$$

we mean that the result of the function b acting on the function c is another function a. There is no difficulty about expressing the idea of a function of two or more variables in this scheme. If we wish to think of f as a function of two variables p and q, say, we may simply write

$$(fp)q$$

(which is the result of the function fp as applied to q). For a function of three variables we consider

$$((fp)q)r,$$

and so on.

Now comes the powerful operation of *abstraction*. For this we use the Greek letter λ (lambda) and follow it immediately by a letter standing for one of Church's functions, say x, which we consider as a 'dummy variable'. Every occurrence of the variable x in the square-bracketed expression which immediately follows is then considered merely as a 'slot' into which may be substituted anything that follows the whole expression. Thus if we write

$$\lambda x.[fx],$$

we mean the function which when acting on, say, a produces the result fa. That is to say,

$$(\lambda x.[fx])a = fa.$$

In other words, $\lambda x.[fx]$ is simply the function f, i.e.

$$\lambda x.[fx] = f.$$

This bears a little thinking about. It is one of those mathematical niceties that seems so pedantic and trivial at first that one is liable to miss the point

* A more familiar form of notation would have been to write $a = b(c)$, say, but these particular parentheses are not really necessary and it is better to get used to their omission. To include them consistently would lead to rather cumbersome formulae such as $(f(p))(q)$ and $((f(p))(q))(r)$, instead of $(fp)q$ and $((fp)q)r$ respectively.

completely. Let us consider an example taken from familiar school mathematics. We take the function f to be the trigonometrical operation of taking the sine of an angle, so the abstract function 'sin' is defined by

$$\lambda x.[\sin x] = \sin.$$

(Do not worry about how the 'function' x may be taken to be an angle. We shall shortly see something of the way that numbers may be regarded as functions; and an angle is just a kind of number.) So far, this *is* indeed rather trivial. But let us imagine that the notation 'sin' had not been invented, but that we are aware of the power series expression for sin x:

$$x - \tfrac{1}{6}x^3 + \tfrac{1}{120}x^5 - \ldots.$$

Then we could define

$$\sin = \lambda x.[x - \tfrac{1}{6}x^3 + \tfrac{1}{120}x^5 - \ldots].$$

Note that, even more simply, we could define, say, the 'one-sixth cubing' operation for which there is no standard 'functional' notation:

$$Q = \lambda x.[\tfrac{1}{6}x^3]$$

and find, for example,

$$Q(a + 1) = \tfrac{1}{6}(a + 1)^3 = \tfrac{1}{6}a^3 + \tfrac{1}{2}a^2 \tfrac{1}{2}a + \tfrac{1}{6}.$$

More pertinent to the present discussion would be expressions made up simply from Church's elementary functional operations, such as

$$\lambda f.[f(fx)].$$

This is the function which, when acting on another function, say g, produces g iterated twice acting on x, i.e.

$$(\lambda f.[f(fx)])\mathrm{g} = \mathrm{g}(\mathrm{g}x).$$

We could also have 'abstracted away' the x first, to obtain

$$\lambda f.[\lambda x.[f(fx)]],$$

which we may abbreviate to

$$\lambda fx.[f(fx)].$$

This is the operation which, when acting on g, produces the function 'g iterated twice'. In fact this is the very function that Church identifies with the natural number 2:

$$\mathbb{2} = \lambda fx.[f(fx)],$$

so $(\mathbb{2}\ \mathrm{g})y = \mathrm{g}(\mathrm{g}y)$. Similarly he defines:

$$\mathbb{3} = \lambda fx.[f(f(fx))], \qquad \mathbb{4} = \lambda fx.[f(f(f(fx)))], \qquad \text{etc.,}$$

together with

$$\mathbb{1} = \lambda fx.[fx], \qquad \mathbb{0} = \lambda fx.[x].$$

Really, Church's '$\mathbb{2}$' is more like 'twice' and his '$\mathbb{3}$' is 'thrice', etc. Thus, the action of $\mathbb{3}$ on a function f, namely $\mathbb{3}\,f$, is the operation 'iterate f three times'. The action of $\mathbb{3}\,f$ on y, therefore, would be $(\mathbb{3}\,f)y = f(f(f(y)))$.

Let us see how a very simple arithmetical operation, namely the operation of adding $\mathbb{1}$ to a number, can be expressed in Church's scheme. Define

$$S = \lambda abc.[b((ab)c)].$$

To illustrate that S indeed simply adds $\mathbb{1}$ to a number described in Church's notation, let us test it on :

$$\begin{aligned} S\,\mathbb{3} &= \lambda abc.[b((ab)c)]\,\mathbb{3} = \lambda bc.[b((\mathbb{3}\,b)c)] \\ &= \lambda bc.[b(b(b(bc)))] = \mathbb{4}\ , \end{aligned}$$

since $(\mathbb{3}\,b)c = b(b(bc))$. Clearly this applies equally well to any other natural number. (In fact $\lambda abc.[(ab)(bc)]$ would also have done just as well for S.)

How about multiplying a number by two? This doubling can be achieved by

$$D = \lambda abc.[(ab)((ab)c)],$$

which is again illustrated by its action on $\mathbb{3}$:

$$\begin{aligned} D &= \lambda abc.[(ab)((ab)c)]\,\mathbb{3} = \lambda bc.[(\mathbb{3}\,b)((\mathbb{3}\,b)c)] \\ &= \lambda bc.[(\mathbb{3}\,b)(b(b(bc)))] = \lambda bc.[b(b(b(b(b(bc)))))] = \mathbb{6}\ . \end{aligned}$$

In fact, the basic arithmetical operations of addition, multiplication, and raising to a power can be defined, respectively, by:

$$A = \lambda fgxy.[((fx)(gx))y],$$

$$M = \lambda fgx.[f(gx)],$$

$$P = \lambda fg.[fg].$$

The reader may care to convince herself or himself—or else to take on trust—that, indeed,

$$(A\mathbb{m})\mathbb{n} = \mathbb{m}+\mathbb{n}\ , \qquad (M\mathbb{m})\,\mathbb{n} = \mathbb{m}\times\mathbb{n}\ , \qquad (P\mathbb{m})\,\mathbb{n} = \mathbb{n}^{\mathbb{m}}\ ,$$

where $\mathbb{m}$ and $\mathbb{n}$ are Church's functions for two natural numbers, $\mathbb{m} + \mathbb{n}$ is his function for their sum, and so on. The last of these is the most astonishing. Let us just check it for the case $\mathbb{m} = \mathbb{2}$, $\mathbb{n} = \mathbb{3}$:

$$\begin{aligned} (P\mathbb{2}\,)\mathbb{3} &= ((\lambda fg.[fg])\,\mathbb{2}\,)\,\mathbb{3} = (\lambda g.[\,\mathbb{2}\,g])\,\mathbb{3} \\ &= (\lambda g.[\lambda fx.[f(fx)]g])\,\mathbb{3} = \lambda gx.[g(gx)]\mathbb{3} \\ &= \lambda x.[\,\mathbb{3}\,(\mathbb{3}\,x)] = \lambda x.[\lambda fy.[f(f(fy))](\mathbb{3}\,x)] \\ &= \lambda xy.[(\mathbb{3}\,x)((\mathbb{3}\,x)((\mathbb{3}\,x)y))] \end{aligned}$$

$$\begin{aligned}&= \lambda xy.[(\mathbb{3}\,x)((\mathbb{3}\,x)(x(x(xy))))]\\ &= \lambda xy.[(\mathbb{3}\,x)(x(x(x(x(x(xy))))))]\\ &= \lambda xy.[x(x(x(x(x(x(x(x(xy)))))))))] = \mathbb{9} = \mathbb{3}^{\mathbb{2}}\end{aligned}$$

The operations of subtraction and division are not so easily defined (and, indeed, we need some convention about what to do with '$\mathbb{m} - \mathbb{n}$' when $\mathbb{m}$ is smaller than $\mathbb{n}$ and with '$\mathbb{m} \div \mathbb{n}$' when $\mathbb{m}$ is not divisible by $\mathbb{n}$). In fact a major landmark of the subject occurred in the early 1930s when Kleene discovered how to express the operation of subtraction within Church's scheme! Other operations then followed. Finally, in 1937, Church and Turing independently showed that every computable (or algorithmic) operation whatever—now in the sense of Turing's machines—can be achieved in terms of one of Church's expressions (and vice versa).

This is a truly remarkable fact, and it serves to emphasize the fundamentally objective and mathematical character of the notion of computability. Church's notion of computability has, at first sight, very little to do with computing machines. Yet it has, nevertheless, some fundamental relations to practical computing. In particular, the powerful and flexible computer language LISP incorporates, in an essential way, the basic structure of Church's calculus.

As I indicated earlier, there are also other ways of defining the notion of computability. Post's concept of computing machine was very close to Turing's, and was produced independently, at almost the same time. There was currently also a rather more usable definition of computability (recursiveness) due to J. Herbrand and Gödel. H. B. Curry in 1929, and also M. Schönfinkel in 1924, had a different approach somewhat earlier, from which Church's calculus was partly developed. (See Gandy 1988.) Modern approaches to computability (such as that of an *unlimited register machine*, described in Cutland 1980) differ considerably in detail from Turing's original one, and they are rather more practical. Yet the *concept* of computability remains the same, whichever of these various approaches is adopted.

Like so many other mathematical ideas, especially the more profoundly beautiful and fundamental ones, the idea of computability seems to have a kind of *Platonic reality* of its own. It is this mysterious question of the Platonic reality of mathematical concepts generally that we must turn to in the next two Chapters.

1. I am adopting the usual modern terminology which now includes zero among the 'natural numbers'.
2. There are many other ways of coding pairs, triples, etc., of numbers as single numbers, well known to mathematicians, though less convenient for our present

purposes. For example, the formula $\frac{1}{2}((a + b)^2 + 3a + b)$ represents the pairs (a, b) of natural numbers uniquely as a single natural number. Try it!

3. I have not bothered, in the above, to introduce some mark to *initiate* the sequence of numbers (or instructions, etc.). This is not necessary for the input, since things just start when the first 1 is encountered. However, for the output something else may be needed, since one may not know *a priori* how far to look along the output tape in order to reach the first (i.e. leftmost) 1. Even though a long string of 0s may have been encountered going off to the left, this would be no guarantee that there would not be a 1 still *farther* off on the left. One can adopt various viewpoints on this. One of these would be always to use a special mark (say, coded by 6 in the contraction procedure) to initiate the entire output. But for simplicity, in my descriptions I shall take a different point of view, namely that it is always 'known' how much of the tape has actually been encountered by the device (e.g. one can imagine that it leaves a 'trail' of some kind), so that one does not, in principle, have to examine an infinite amount of tape in order to be sure that the entire output has been surveyed.
4. One way of coding the information of two tapes on a single tape is to interleave the two. Thus, the odd-numbered marks on the single tape could represent the marks of the first tape, whilst the even-numbered ones could represent the marks of the second tape. A similar scheme works for three or more tapes. The 'inefficiency' of this procedure results from the fact that the reading device would have to keep dodging backwards and forwards along the tape and leaving markers on it to keep track of where it is, at both even and odd parts of the tape.
5. This procedure refers only to the way in which a marked tape can be interpreted as a natural number. It does not alter the numbers of our specific Turing machines, such as **EUC** or **XN+1**.
6. If T_n is *not* correctly specified, then U will proceed as though the number for n has terminated as soon as the first string of more than four 1s in n's binary expression is reached. It will read the rest of this expression as part of the tape for m, so it will proceed to perform some nonsensical calculation! This feature could be eliminated, if desired, by arranging that n be expressed in *expanded* binary notation. I have chosen not to do this so as not to complicate further the description of the poor universal machine U!
7. I am grateful to David Deutsch for deriving the denary form of the binary description for u which I had worked out below. I am grateful to him also for checking that this binary value of u actually *does* give a universal Turing machine! The binary value for u is in fact:

```
10000000010111010011010001001010101101000110100010100000110101001101000101
01001011010000110100010100101011010010011101001010010010111010100011101010
10010010101110101010011010001010001010110100000110100100000101011010001001
11010010100001010111010010001110100101010000101110100101001101000010000111
01010000111010100001001001110100010101011010100101011010000011010101001011
01001001000110100000000110100000011101010010101010111010000100111010010101
01010101011101000010101011101000010100010111010001010011010010000101001101
00101001001101001000101101010001011101001001010111010010100011101010010100
10011101010101000011010010101010111010100100010110101000010110101000100110
```

```
10101010100010110100101010010010110101001001011101010100101011101010010100
11010101000011101000100100101011101010100101011101010100000111010100100000
11010101010010111010100101011010001001000111010000000111010010100101010101
11010010100100101011101000001010111010000100011101000001010100111010000101
00111010000010001011101000100001110100001001010011101000100001011010001010
01011101000101001011010010000010110100010101001001101000101010101110100100
00011101001001010101011101010101001101001000101011010010010010110100000001
01101000001000110100000100101101000000000110100101000101110100101010001101
00101001010110100000100111010010101001011010010011101010000001010111010100
00001101010100010101011010010101011010100001010111010100100101011101010001
00101101010010000101110100000011101010010001011010100101001101010100010111
01010010100101110101010000010111010101000001011101000000111010101000010101
11010010101011010101000010111010100010101011101010100100101110101010100001
11010100000001110100100100001101001001000101101010101010011101000000001011
01001000011010101010100101110100100001101001000101010111010000100011101000
10000111010000110100000000101101000001001011101010100101010110100010001001
11101000001001110101010011010000010101011010000100001110100100001000111010
10101010100111010000100100111010001001000011101000010100101101000010100001
11010101010101011101000100100110100010010011010100101001011101000100010101
11010000000111010001001001011101001101001001000010110101010100110100010100
01011101000011010100001000101101010011010101001010010110101010011010010010
10111010011010010000010110100010101010001110100100001010110100000010011010
01000100101110100100001101010000010010111010010010100110100100101010110100
11010010010100101101001101001010000010110100100000111010100100110101010100
00101110100101000010111010010101010111010100010010110100100111010010101000
10111010001001110101000010110100100111010010101010101110100100011101001010
10100101110100100011101010000010101011100110101000001011010010011101010000
00101110100101101010000010101101001010010111010100001001011101000011010100
01000010110101001101010001000101101010101001011101010001010010110100010101
01011101001000010101101010001011101010010010101011101010100100101110101000
11101010001110101001001001011101010001110101001010001011101010001011101010
00010010111010100011101000101000101110100101001011101010010101001011101001
01010101010110101000010101010110100001001110100001010101010111010101000101
01110101010001010111010000001110101010001001011101000000111010101001000101
11010100000011010100001011010000001110100100000010111010100011101010010001
01011101010011010101010001010110100000110101010100101010110100000010011010
10101001001110101001101010101001001011010100110100100100111010000011010101
01010010101101010001001101000101001010101110100000110101010101010010110100
01000111010001010101010101101000100011101000010101110100010010000111010011
01000000010011101000000100101110100010001010011101000000100101110100101010
10100101101000010101010111010001001010010111010000010001011101010100101101
00010001001110100000100101011101000000101010110100001000111001111010000100
00011101000010010011101000001010010111010000010100101101000010001010111010
00010001001101000100001110101111010000100100101110100001001001011101000000
01010111010000101010001101000100101110100001000001110100001001110100010000
01011101010100101101000100000101110100001010101011101000000101010111010001
00001010111010001000010101110100100000111010100100100110100000010101110100
```

```
0100010010111010101000011101010010101101001010101000011010000010100110100
0000111010000010010011101001011010010001010010110101010011010001010010010
1010101001101000101010001011001101010010010111010101001101000101010101011
0110101000101010110011010010001010101011101000100011101001001010101010110
0010100101000110100100000010111010000011010101001010101011010010101011010
1000100010111010001010101101010000010101101000100000110100100010101101000
1001110101001010101010111010010110100100100010101100110100100100101010111
1001101001001001010110100101101001001001001011010010110100100101000101100
1010010010100101011101000101011101001001011100110100100101010010111001101
0101000101010111010001000111010000101001011010010100010111010010100010101
0100010011101001010001001011101000100111010010100100010111001101001000100
1110100010011101001010010101011100110100101000001110011010101010101101000
0001110100101010010101011101001000111010010101001010111001101000010100100
1001101010000011010000000111010010101010010101110011010100010000110100000
0111010001001010101011101000100011101010101010101011010000100111010010000
1001010111010010101000100110101000000010110100100111010100001010111010010
0011010100000001011010010001110101001001011101000011010100001010101101010
0101110101000010100101110101000101110101000101010101110011010100010101101
0001101010001001010
```

The enterprising reader, with an effective home computer, may care to check, using the prescriptions given in the text, that the above code does in fact give a universal Turing machine's action, by applying it to various simple Turing machine numbers!

Some lowering of the value of *u* might have been possible with a different specification for a Turing machine. For example, we could dispense with STOP and, instead, adopt the rule that the machine stops whenever the internal state 0 is re-entered after it has been in some other internal state. This would not gain a great deal (if anything at all). A bigger gain would have resulted had we allowed tapes with marks other than just 0 or 1. Very concise-looking universal Turing machines have indeed been described in the literature, but the conciseness is deceptive, for they depend upon exceedingly complicated codings for the descriptions of Turing machines generally.

8. For a non-technical discussion of matters relating to this famous assertion, see Devlin (1988).
9. We could, of course, defeat this improved algorithm too, by simply applying the foregoing procedure all over again. We can then use this new knowledge to improve our algorithm still further; but we could defeat that one also, and so on. The kind of consideration that this iterative procedure leads us into will be discussed in connection with Gödel's theorem, in Chapter 4, cf. p. 109.

3
Mathematics and reality

The land of Tor'Bled-Nam

Let us imagine that we have been travelling on a great journey to some far-off world. We shall call this world Tor'Bled-Nam. Our remote sensing device has picked up a signal which is now displayed on a screen in front of us. The image comes into focus and we see (Fig. 3.1):

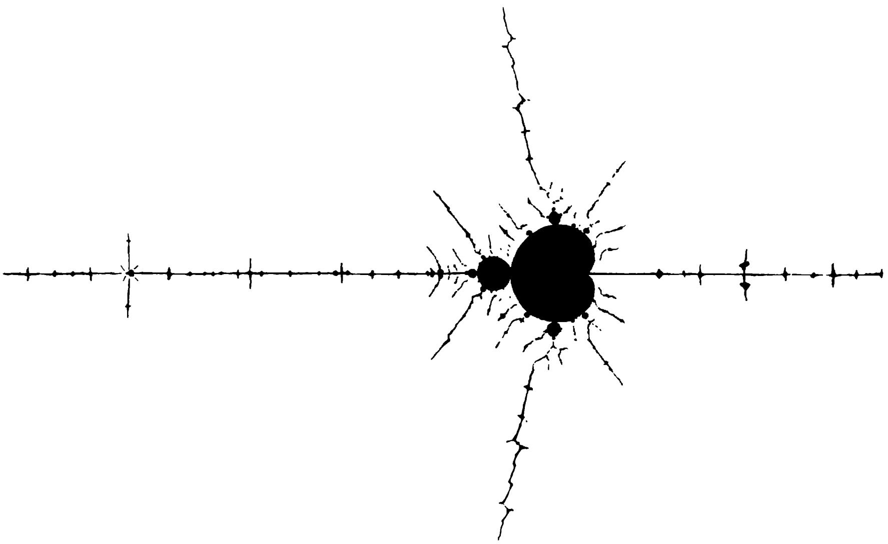


Fig. 3.1. A first glimpse of a strange world.

What can it be? Is it some strange-looking insect? Perhaps, instead, it is a dark-coloured lake, with many mountain streams entering it. Or could it be some vast and oddly shaped alien city, with roads going off in various directions to small towns and villages nearby? Maybe it is an island—and then let us try to find whether there is a nearby continent with which it is associated. This we can do by 'backing away', reducing the magnification of our sensing device by a linear factor of about fifteen. Lo and behold, the entire world springs into view (Fig. 3.2):