6 Turing computability

In 1936, Alan Turing introduced a formal model of computation by defining a simple type of computer – now known as a *Turing machine* – that, he suggested, can implement any mechanical algorithm.

6.1 Turing machines

Let's think about what is involved in following an algorithm. An algorithm converts some input string into an output string. Let's assume that the input string is received on a piece of paper. The algorithm specifies what one should do with that string, providing step-by-step instructions to add, remove, or change symbols on the paper, until the output string is produced. At each step in the process, the algorithm clearly specifies what to do next, based on what's currently on the paper and on the current stage of the computation. When the computation is finished, the output string must be marked as such on the paper, perhaps by circling or underlining it. For definiteness, let's stipulate that the algorithm should contain instructions to erase everything else on the paper, leaving only the output.

A Turing machine is a machine that implements a process of this kind. It reads and writes symbols on a piece of paper, thereby converting an input string into an output string by following precise, step-by-step instructions.

Turing's key insight was that such a machine can be designed in a very simple way. To begin, we can assume that the paper on which the machine operates is a single strip of paper: any algorithm that requires writing symbols above or below other symbols can be reformulated as an algorithm that only requires writing symbols to the left or right of other symbols. A Turing machine therefore operates on a *tape* in which the symbols are always arranged in a single line. The tape is divided into "cells" or "squares", each of which can hold a single symbol. Since we're interested in what can be computed in principle, without worrying about practical limitations, we assume that the tape is unbounded. (If the machine reaches the end of the tape, the tape is automatically extended.)

Next, we make the steps in the computation as simple as possible. Evidently, any instruction for writing down a sequence of symbols can be broken down into a sequence

of instructions for writing down a single symbol. We'll therefore assume that at each step, a Turing machine writes at most one symbol onto its tape. Similarly, we assume that a Turing machine can only read a single cell on its tape at a time. Any instruction for reading larger chunks of the tape can be broken down into instructions for reading single cells.

At each step, a Turing machine therefore operates on a single cell of its tape. We say that it has a *head* that is positioned on this cell. At each step, the machine can read the content of the current cell; it can erase that content or replace it with a different symbol, and it can move its head left or right, by one cell.

For definiteness, we assume that each step involves all these actions. That is, each step in a Turing machine computation consists of three parts:

1. Read the content of the current cell;
2. Erase the content of the current cell and either leave it blank or write a new symbol onto it;
3. Move the head one cell to the left or right.

We can make one last simplification. Every known algorithm operates on strings from a finite alphabet. We can code these strings as numbers greater than 0. Each such number n can be written in *unary* notation, as a sequence of n *strokes*. Since there are effective algorithms for converting the original strings into sequences of strokes and back, any algorithm that operates on the original strings can be converted into an algorithm that operates on sequences of strokes. We'll therefore assume that the only symbol available to a Turing machine is the stroke.

In sum, a Turing machine has an unbounded tape, divided into cells, each of which can hold either a stroke or be blank. The machine works in steps, in accordance with a predefined program. At each step, the machine's head is positioned at a particular cell of the tape. A step involves reading the content of that cell, replacing it with either a blank or a stroke, and moving the head one cell to the left or right.

A program for a Turing machine specifies what the machine does at each step. This generally depends on the content of the current cell. A simple program might look as follows:

Step 1. If the current cell contains a stroke, erase it and move right; if the current cell is blank, write a stroke and move left.

Step 2. If the current cell contains a stroke, leave it and move left; if the current cell is blank, write a stroke and move left.

We're not assuming that a machine that executes this program would somehow read and understand the instructions. Turing machines only read strokes or blanks on their tape. A machine that executes the above program would simply be wired to follow the two instructions, one after the other, and then stop.

To build such a machine, we would need an internal switch or counter to keep track of where it is in the computation – whether it should follow instruction 1 or instruction 2. The machine might, for example, have a switch that can be in one of two positions, "up" and "down". We could then build it in such a way that it follows instruction 1 if the switch is up and instruction 2 if the switch is down. The switch would start in the up position and flip to down after the machine has finished following the first instruction.

To allow for programs with more than two instructions, we must allow the switch to have any finite number of positions. These switch positions are called *states* of the machine, and labelled $q_0, q_1, q_2, \ldots$. It doesn't matter how they are implemented. You can think of each state as indicating a "line" in the program the machine is executing.

Exercise 6.1 What does the above machine do if it starts (a) on a tape with a single stroke under its head? (b) on an empty tape?

Many algorithms involve repeating certain steps. The algorithms you've learned for written addition and multiplication, for instance, probably go through each digit in the decimal representation of the input numbers, asking you to perform the same operations for each digit. To implement such an algorithm, a Turing machine must be able to go into the same state more than once. Here is a program for a machine of this kind.

Instruction 1. If the current cell contains a stroke, erase it and move right, then process Instruction 2; if the current cell is blank, write a stroke, move right, and halt.

Instruction 2. If the current cell contains a stroke, erase it and move right; if the current cell is blank, write a stroke and move right; either way, continue with Instruction 1.

A machine that implements this program still needs two states, q_0 and q_1. In state q_0, it follows instruction 1; in state q_1, it follows instruction 2. Each instruction effectively specifies three actions:

- what to write into the current cell (a stroke or a blank);
- whether to move left or right;
- what state to go into next.

We can introduce a compact notation for these instructions, using, for example, '$1, \mathrm{R}, q_1$' to mean that the machine should replace the content of the current cell by a stroke, move right, and go into state q_1, and '$\mathrm{B}, \mathrm{L}, q_0$' for "empty the current cell, move left, and go into state q_0". The above program can then be written as a table:

	1	B
q_0	$\mathrm{B}, \mathrm{R}, q_1$	$1, \mathrm{R}, q_2$
q_1	$\mathrm{B}, \mathrm{R}, q_0$	$1, \mathrm{R}, q_0$
q_2		

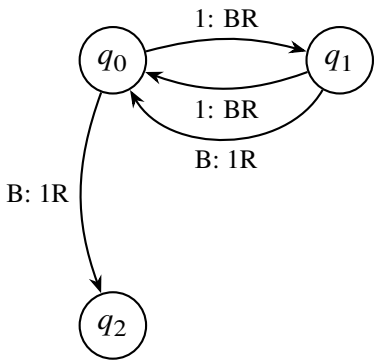
Each cell in the table holds the instruction for what to do in a given state (the row) when reading a given symbol (the column). I've added a third state, q_2, so that the directive to halt can be represented as the directive to go into a new state for which there are no instructions.

There are other ways to represent the same program. We could, for example, package it into a list of quintuples:

$$(q_0, 1, \mathrm{B}, \mathrm{R}, q_1), \quad (q_0, \mathrm{B}, 1, \mathrm{R}, q_2), \quad (q_1, 1, \mathrm{B}, \mathrm{R}, q_0), \quad (q_1, \mathrm{B}, 1, \mathrm{R}, q_0).$$

Here, $\langle q_0, 1, \mathrm{B}, \mathrm{R}, q_1 \rangle$ means that if the machine is in state q_0 and reads a stroke, then it should erase the stroke ("write a blank"), move right, and go into state q_1. Similarly for the other entries in the list.

Another popular way to represent Turing machine programs is as a flow chart. Here is the same machine again:



The nodes in the chart represent the states. Each arrow represents an instruction. '1 : BR' means: 'if you read 1, write a blank, and move right'. The new state is given by the node to which the arrow points.

Exercise 6.2 Can you figure out what this machine does if it starts at the left end of a sequence of strokes on an otherwise empty tape?

Let's do this exercise together. The machine starts in state q_0, reading the first stroke. It erases the stroke and moves right, entering state q_1. It reads the second stroke, erases it, moves right, and goes back into state q_0. It keeps alternating between q_0 and q_1 in this way, moving right and erasing strokes, until it reaches the first blank. At that point, the machine is either in state q_0 or q_1, depending on whether the original tape had an even or an odd number of strokes. If the number of strokes was even, the machine is now in state q_0; it reads the blank, prints a stroke, moves right and halts (in state q_2). If the tape originally had an odd number of strokes, the machine is in state q_1 when it reaches the first blank. It prints a stroke, moves right, and goes into q_0. It then reads another blank, prints another stroke, moves right, and halts.

The machine implements an algorithm for deciding whether the input sequence has an even or odd number of strokes. If even, the output is a single stroke; if odd, the output is two strokes.

Exercise 6.3 In computer programming, it is important to check for edge cases. Does the program correctly classify the empty input as having an even number of strokes?

Exercise 6.4 Design a Turing machine that extends any input sequence of strokes by one stroke: when starting on the left-most stroke of a sequence of n strokes on an otherwise blank tape, the machine should eventually halts on an otherwise blank tape with a sequence of $n + 1$ strokes.

Exercise 6.5 Draw a flow chart for the machine given by the following quintuples: $(q_0, \mathrm{B}, 1, \mathrm{R}, q_1)$, $(q_0, 1, 1, \mathrm{L}, q_2)$, $(q_1, \mathrm{B}, 1, \mathrm{L}, q_0)$, $(q_1, 1, 1, \mathrm{R}, q_1)$, $(q_2, \mathrm{B}, 1, \mathrm{L}, q_1)$. Can you figure out what this machine does, when started on a blank tape?

6.2 Computing arithmetical functions

A Turing machine converts a pattern of strokes and blanks on its tape into another pattern of strokes and blanks. To compute a function that doesn't take such patterns as input or output, we must code the inputs and outputs as patterns of strokes and blanks.

Let's look at functions on the natural numbers. In the previous section, I suggested that we could code each number n as a sequence of n strokes. This works, but it has the downside that we can't distinguish between empty cells and cells that store the number 0. In this section, I'll therefore use a slightly different coding scheme, in which each number n is represented by a sequence of $n + 1$ strokes: the number 0 is coded by a single stroke, 1 by a sequence of two strokes, and so on.

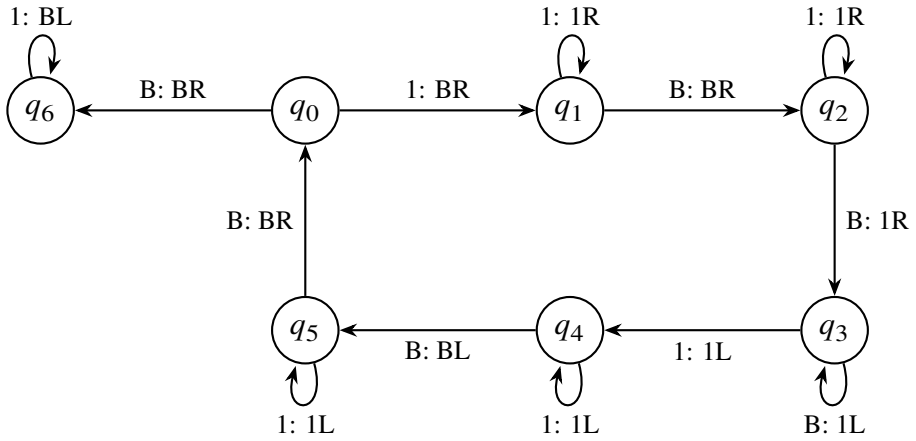
We say that a Turing machine *computes a function f from $\mathbb{N}$ to $\mathbb{N}$* if, whenever it starts on the left-most stroke of a sequence of $n + 1$ strokes on an otherwise blank tape, it eventually halts on a tape with a sequence of $f(n) + 1$ strokes on an otherwise blank tape. A function f from $\mathbb{N}$ to $\mathbb{N}$ is *Turing-computable* if it is computed by some Turing machine.

These definitions can obviously be generalized to functions with more than one argument. If a function takes n numbers as input, we stipulate that these numbers must be represented by n blocks of strokes, separated by a blank. For example, if we want a Turing machine to add the numbers 2 and 3, we would start it on a tape that looks like this:

Exercise 6.6 Since blanks and strokes effectively give us two symbols, one might suggest that we could code numbers in binary, so that 0 is coded as a blank (B) 1 as a stroke (1) 2 as 1B, 3 as 11, 4 as 1BB, and so on. Explain why this doesn't work.

In exercise 6.4, you designed a Turing machine that adds a single stroke to the sequence of strokes at which it starts. By our present conventions, this machine computes the successor function that takes a number as input and returns that number plus 1.

Here is a machine that computes the "times 2" function: it converts a sequence of $n+1$ strokes into a sequence of $2n + 1$ strokes.



The output sequence is constructed to the right of the input, with a blank as a separator. The machine successively removes one stroke from the input, then moves right past the first blank after the input, then moves right past all strokes that follow the blank, prints two strokes, and returns to the left-most stroke on what remains of the input sequence, until that sequence is completely erased. At this point, the machine has created a block of $2n + 2$ strokes. It removes the left-most stroke of this block and halts.

Exercise 6.7 Design a Turing machine that computes addition: when started at the left end of a sequence of $n + 1$ strokes followed by a blank followed by $m + 1$ strokes, the machine halts on a tape with $n + m + 1$ consecutive strokes.

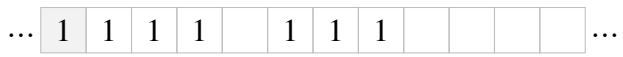
To implement more complex algorithms, it helps to think in terms of subroutines. Let's tackle multiplication. A Turing machine that computes multiplication would start at the left end of a block of $n + 1$ strokes, followed by a blank, followed by another block of $m + 1$ strokes, and eventually halt on a tape with $n \times m + 1$ consecutive strokes. How could we design such a machine?

We could use the first block of strokes as a counter, as in the doubling machine: we'd erase one stroke at a time from the left block; for each stroke that's erased, we add m strokes to the second block; we do this until the counter block has only two strokes left, at which point we erase these strokes and halt. (Do you understand why we'd stop when there are two strokes left in the counter?)

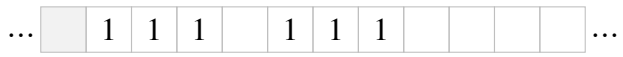
But how can we repeatedly add m strokes to the second block? After i iterations, the first block would have $n + 1 - i$ strokes and the second $i \times m + 1$. It's hard to extract from this the original value m. So here's a better idea: instead of directly adding m strokes to the second block, we insert *m blanks inside* the second block, after its first stroke. That

is, we shift the last m strokes of the second block m squares to the right. When we're done, we fill all these blanks with strokes.

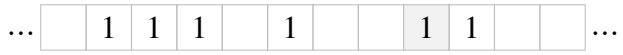
For example, consider the case of $n = 3$ and $m = 2$. The input tape is



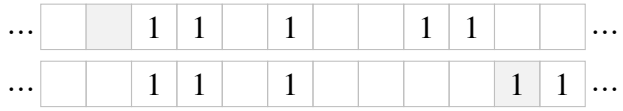
The desired output is a sequence of $3 \times 2 + 1 = 7$ strokes. We begin by erasing the first stroke in the left block (the counter block). We now have



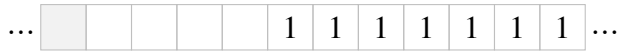
Then we shift the two last strokes in the right block two squares to the right:



Then we start over, erasing another stroke in the counter block and shifting the two strokes on the right by two more squares:

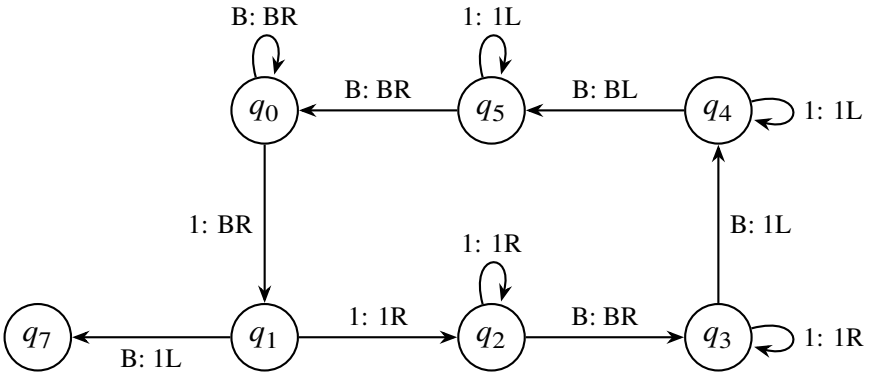


Now there are only two strokes left in the counter. We erase these two strokes and fill in all the blanks we've inserted in the right block:

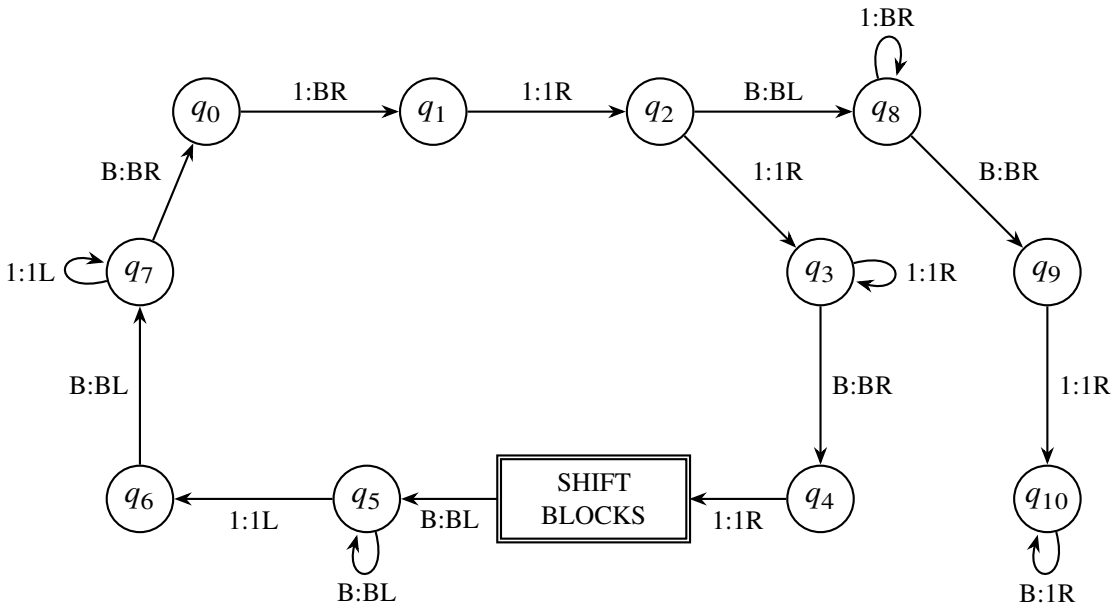


We have the desired output of seven strokes.

Most of this is straightforward to implement. The only slightly tricky part is the subroutine for shifting the strokes in the second block. Let's think of this as a separate task. Let's assume that the head is at the first of the m strokes that we want to shift by m squares to the right. The machine that achieves this should not change anything to the left of its starting position. Here is a machine that does the job. Let's call it 'SHIFT BLOCKS'.



We can now plug this into a multiplication machine, using the algorithm I've just described:



In q_0, this machine removes the current stroke in the counter block. It then moves right twice. If it lands on a blank, there is only one stroke left in the counter block, and the machine goes into the cleanup routine q_8–q_{10}, where it erases the remaining counter stroke and fills the blanks in the right block. Alternatively, if there are further strokes in the counter block, the machine moves right past the counter block, past the separator blank, and past the first stroke of the right block. It then calls the 'SHIFT BLOCKS' subroutine (which I've conveniently defined so that if it starts on a blank then it first

moves right until it finds a stroke). After the subroutine, the machine moves back to the left end of the counter block.

It takes some practice and patience to design Turing machines that compute arithmetical functions. In the next chapter, we'll show with one very general argument that a wide range of arithmetical functions can be computed by Turing machines.

Exercise 6.8 My multiplication machine has a bug: it doesn't correctly deal with certain edge cases. Can you find the bug? Can you fix it?

Exercise 6.9 Design a Turing machine that computes the function $\max(x, y)$ that takes two numbers as input and returns the larger of the two.

6.3 Universal Turing machines

Every Turing machine computes a particular function. There is a machine for computing addition, another for multiplication, and so on. As Turing pointed out, one can also design a "universal" Turing machine that can compute *any* computable function. Such a machine takes as input an algorithm for computing a function, as well as the arguments to that function. For example, if we supply the machine with an algorithm for addition and the numbers 2 and 3, it would compute the output 5. If we give it an algorithm for multiplication and the numbers 2 and 3, it would compute the output 6.

There are different ways of representing an algorithm. A natural choice, in the present context, is to use Turing machine specifications. Our universal Turing machine U will therefore take as input a specification of a Turing machine M, as well as some input I for M. Its output will be the output produced by M on input I.

Let's think about how we could build such a machine. To begin, we need to code specifications of Turing machines as patterns of strokes and blanks, so that we can feed them as input to the universal machine.

We know that every Turing machine can be represented as a list of quintuples of the form

$$\langle q_i, s, s', d, q_j \rangle,$$

where q_i and q_j are states, s and s' are tape symbols (stroke or blank), and d is a direction (left or right). We can code each of these components by a string of strokes, using (say) $i + 1$ strokes for state q_i, one stroke for the blank and for 'left' and two strokes for the stroke and for 'right'. We can then represent a quintuple by putting its component codes

end to end, separated by a blank. The quintuple $\langle q_0, 1, \mathrm{B}, \mathrm{R}, q_1 \rangle$, for example, would be coded by 1 B 1 1 B 1 B 1 1 B 1 1. To represent an entire machine, we put all the codes for its quintuples end to end, separated by (say) three blanks. We'll call this pattern of strokes and blanks the *machine code* of the machine.

Next, we need to design a Turing machine U that can read the code of a machine M and simulate the behaviour of that machine for any input I. The input for U is the machine code of M, followed by, say, four blanks, followed by the input I for M.

While simulating M, U will divide its tape into three parts. The left part will store the code of M. The right part is a simulation of M's tape. The middle part is a working area.

MACHINE CODE WORK AREA SIMULATED TAPE AREA

U is going to simulate each step of running M on I. To this end, U needs to keep track of M's position on its tape. We achieve this by adding a marker for the position of M's head in the simulated tape area. To make space for the marker, we begin by inserting a blank in between any two cells of M's input, so that the original input lies in the odd-numbered squares. For example, if the original input I was

1	1		1		1	1							

then this is converted to

1		1				1				1		1	

in the simulated tape area. The even-numbered cells can now be used to mark the position of M's head. At the beginning, M's head is positioned on the first cell of its input; U marks this by putting a stroke into the first even-numbered cell of the simulated tape area:

1	1	1				1				1		1	

U also needs to keep track of M's current state. To this end, it simply stores the code of the state (a single stroke for q_0, two strokes for q_1, and so on) in the work area. Initially, U writes a single stroke there, assuming that every machine starts in q_0.

After this preparatory work, the simulation begins. It goes as follows.

Stage 1. Find the active position in the simulated tape area, by moving right until you meet the first even square with a stroke. The cell to your left holds the symbol currently scanned by M. Remember this symbol by going into distinct states depending on whether it is a blank or a stroke.

Stage 2. Either way, move left to the work area and print, to the right of the code for M's current state, a blank, followed by the code of the currently scanned symbol (1 or 11).

Remember that the machine code stored in the left part of the tape divides into quintuple blocks, each of which begins with a state code followed by a "current symbol" code. The work area therefore now contains the first two items of the quintuple that holds the instruction for what to do in the current state when reading the currently scanned symbol.

Stage 3. Move left to find the position in the machine code that matches the string in the work area, preceded by three blanks. If there's no match, the simulation is finished. In this case, erase everything but the simulated tape area, as well as all the even-numbered squares in that area, and halt. If there is a match, continue to stage 4.

Stage 4. Scan the instructions in the matched quintuple: remember the symbol to be written onto the tape and the direction to move by going into a different state depending on which symbol is to be written and in which direction to move.

Stage 5. Copy the last element of the quintuple (the new state) into the work area, erasing the previous content of the work area. Then move to the marked position in the simulated tape area, insert the remembered symbol into the cell before the marker stroke. Move the marker in the remembered direction by two steps (because the simulated tape contains the marker spaces). Return to stage 1.

All this is relatively straightforward, albeit tedious, to implement. The most fiddly part is stage 3, where we need to find the position in the machine code matching the string in the work area. This requires keeping track of positions in both strings, which can be done by storing the positions in the work area. If the work area runs out of space, or the simulated machine runs off the left edge of the simulated tape area, a subroutine has to be called that moves the entire content of the simulated tape area to the right.

As described, this design is highly inefficient. But it illustrates a profound fact: a single mechanical architecture can in principle carry out any computation. All modern computers are based on this insight. You don't have to re-wire your laptop or phone whenever you want to run a new program. Instead, you load the program code (the algorithm) into memory and tell the processor to read and execute that code.

Exercise 6.10 Show that if a two-place function f is Turing-computable, then so is the one-place function g such that $g(x) = f(x, x)$.

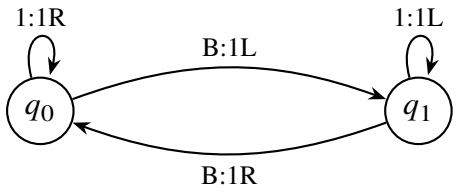
Exercise 6.11 Can the universal Turing machine simulate itself? What happens if you feed U its own machine code as input, together with some further input I for U?

Exercise 6.12 According to the Church-Turing Thesis, any effective, mechanical algorithm can be implemented by a Turing machine. Use the Church-Turing Thesis to argue that there is a universal Turing machine.

6.4 Uncomputability

In the previous chapter, I argued that the set of algorithms is computably enumerable, and inferred that there can be no algorithm that detects whether any given algorithm halts on a given input. We can now see how this plays out for Turing machines.

Every Turing machine has a finite number of states. A Turing machine can still run forever: by going into an infinite loop. Here, for example, is a machine that, when started on a consecutive string of strokes, keeps expanding that string on both ends, without ever halting. (A real computer would "crash" when running this kind of program.)



From the flow chart, it is easy to see that this machine will never halt, no matter its input. But is there a general recipe for determining whether a given Turing machine will halt, on a given input? This is the *halting problem* for Turing machines.

To be clear, the problem is not to determine, for a fixed machine M and input I, whether M will halt on I. *This* problem is trivially decidable. Rather, the problem is to find a general algorithm that decides, for any Turing machine M and any input I, whether M halts on I.

Exercise 6.13 Why is it trivially decidable whether a fixed Turing machine M halts on a fixed input I?

We can show that the halting problem can't be solved by a Turing machine. A Turing machine H that solves the halting problem would take the code of a Turing machine M and an input I for M as input and would output (say) two strokes if M halts on I and one stroke if M doesn't halt on I. We can show by a diagonal argument that such a machine H can't exist.

Theorem 6.1

The Halting Problem is undecidable by a Turing machine.

Proof. Suppose for reductio that there is a Turing machine H that decides the Halting Problem. We could then plug H into a larger machine D that takes the code for a machine M as input and halts iff M halts when given *its own code* (the code of M) as input.

This machine D would be constructed as follows. When started on the code of a machine M, it first creates a copy of the input. It then runs H on the contents of the tape, to determine if M halts on its own code. If the answer is yes, D goes into an infinite loop. If the answer is no, D halts.

Now we get a contradiction if we ask whether D halts on its own code: by design, D halts on the code of M iff M does not halt on its own code; so D halts on its own code iff D does not halt on its own code. It follows that D can't exist, and therefore that H can't exist. □

Exercise 6.14 Can a universal Turing machine get stuck in an infinite loop? If so, how? Could we prevent it by, say, keeping a counter of the number of simulated steps and abort the simulation if that counter exceeds some fixed limit?

The unsolvability of the halting problem can be used to show that various other functions are not Turing-computable. A neat example is the *Busy Beaver function* Σ, introduced by Tibor Rado in 1962. This function takes a number n as input and returns the largest number of strokes that can be printed by a Turing machine with n states before halting, when started on a blank tape.

For example, it is easy to see that $\Sigma(1) = 1$. Let M be any machine with just one state, q_0. When started on a blank tape, the first instruction M executes is the one for q_0 and a blank cell. The machine can either print a stroke or leave the cell blank; then it moves either left or right, to another blank cell. If the machine doesn't halt at this point, it will

again be in state q_0, reading a blank cell; it will repeat the same action, moving in the same direction, without end. So the only way M can halt is by halting after the first step. The most it can print in that step is a single stroke. So the largest number of strokes that a 1-state machine can print before halting (when started on an empty tape) is 1.

A somewhat more involved argument along the same lines shows that $\Sigma(2) = 4$ and $\Sigma(3) = 6$. (In exercise 6.5, I asked you to draw the flow chart for the 3-state machine that prints 6 strokes on an empty tape and then halts.)

If the halting problem were decidable, we could easily compute the Busy Beaver function. For any input number n, we would simply enumerate all Turing machines with n states, use the halting algorithm to discard the non-halting machines, and run the remaining machines (on an empty tape) to see how many strokes they print before halting. Due to the undecidability of the halting problem, this algorithm doesn't work. In fact, there is no algorithm that computes the Busy Beaver function. More precisely, there is no Turing machine that computes the Busy Beaver function. This can be shown by showing that any machine that computes the Busy Beaver function could be used to solve the halting problem. But it can also be shown directly:

Theorem 6.2: (Rado 1962)

The Busy Beaver function is not Turing-computable.

We'll show that every Turing-computable total function f on the natural numbers is eventually overtaken by the Busy Beaver function Σ. That is, for every Turing-computable total function f on $\mathbb{N}$, there is a number k such that $\Sigma(k) > f(k)$. If Σ were Turing-computable, there would be a number k such that $\Sigma(k) > \Sigma(k)$. This is impossible. So Σ is not Turing-computable.

Let f be any Turing-computable total function from $\mathbb{N}$ to $\mathbb{N}$. Then the following function g is also Turing-computable and total:

$$g(x) = \max(f(2x), f(2x+1)) + 1.$$

To compute $g(x)$ for any x, we first create a copy of the input x at a sufficient distance from the original input. Then we use the "times 2" machine to convert the input x into $2x$, and run the machine that computes f on the resulting block of strokes. We then have $f(2x)$ on that part of the tape. Next, we use the "times 2" machine and the "add 1" machine to convert the copy of x into $2x + 1$, and run the machine that computes f on the resulting block. We now have $f(2x)$ and $f(2x + 1)$ on the tape. To finish the

computation of $g(x)$, we run your algorithm for computing max from exercise 6.9, add a single stroke to the result, and halt.

Let M be some such machine for computing g. If M has k states, we can define, for any input x, a machine N_x with $x + k$ states that first writes x strokes on the tape and then imitates M. (No more than x states are needed to write x strokes.)

When started on a blank tape, N_x writes $g(x)$ strokes and then halts. So there is a machine with $x + k$ states that prints $g(x)$ strokes on the empty tape and then halts. By definition of the Busy Beaver function, this means that $\Sigma(x + k) \geq g(x)$. By definition of g, both $f(2x)$ and $f(2x + 1)$ are less than $g(x)$. So we have

$$\Sigma(x + k) \geq g(x) > f(2x);$$
$$\Sigma(x + k) \geq g(x) > f(2x + 1).$$

But obviously, if $x \geq k$ then

$$\Sigma(2x + 1) \geq \Sigma(2x) \geq \Sigma(x + k).$$

Combining these inequalities, we infer that $f(x) < \Sigma(x)$ for $x \geq 2k$. □

I mentioned above that the first few values of the Busy Beaver function are not hard to determine: $\Sigma(0) = 0$, $\Sigma(1) = 1$, $\Sigma(2) = 4$, and $\Sigma(3) = 6$. It is also known that $\Sigma(4) = 13$ and $\Sigma(5) = 4098$. As of 2025, the value of $\Sigma(6)$ is not known exactly; but it is known that there is a 6-state machine that prints $2 \uparrow\uparrow (2 \uparrow\uparrow (2 \uparrow\uparrow 10))$ strokes. So $\Sigma(6)$ is at least $2 \uparrow\uparrow (2 \uparrow\uparrow (2 \uparrow\uparrow 10))$. The up-arrow stands for repeated exponentiation: $2 \uparrow\uparrow 10$ is $2^{2^{\cdot^{\cdot^{2}}}}$ with ten twos in the tower. This number is *much* larger than, say, the number of atoms in the observable universe. $2 \uparrow\uparrow (2 \uparrow\uparrow 10)$ is a power tower of $2 \uparrow\uparrow 10$ twos. You couldn't write down all the twos in this tower even if you managed to write a '2' onto each atom in the universe. $2 \uparrow\uparrow (2 \uparrow\uparrow (2 \uparrow\uparrow 10))$ is a power tower of $2 \uparrow\uparrow (2 \uparrow\uparrow 10)$ twos. $\Sigma(7)$ is known to be at least $2 \uparrow^{11} (2 \uparrow^{11} 3)$, which I won't even try to explain. It is an incomprehensibly large number. You can inspect the machine tables for the known Busy Beaver champions at bbchallenge.org/~pascal.michel/bbc.

As Turing realised, we can also use the undecidability of the halting problem to show that Hilbert's Entscheidungsproblem is unsolvable by a Turing machine: there can be no Turing machine that decides whether any given first-order sentence is valid. The idea is that for any Turing machine M and input I, we can construct a first-order sentence $S_{M,I} \to H_{M,I}$ that is valid iff M halts on input I. The antecedent $S_{M,I}$ is a first-order description of the machine and its input; the consequent $H_{M,I}$ says that the machine

halts. If we could decide whether $S_{M,I} \to H_{M,I}$ is valid (or equivalently, whether $S_{M,I}$ entails $H_{M,I}$), we could decide whether M halts on input I.

To explain what $S_{M,I}$ and $H_{M,I}$ look like, let the *configuration* of a machine M with input I at step n consist of the machine's state, the position of its head on the tape, and the tape's content at step n. $S_{M,I}$ will specify the initial configuration of M on input I, at step 0. It will also describe how the configuration changes from one step to the next, in accordance with the machine table of M. We'll need some non-logical vocabulary to spell this out.

I'll use '0' and 's' to create terms for the computation steps: '0' denotes step 0, 's(0)' step 1, and so on. For the tape positions, I use a constant 'o' ("origin") for the square at which the machine starts, and two unary function symbols 'l' and 'r' that move one square to the left and right, respectively. So '$l(l(o))$', for example, denotes the square two to the left of the starting square. I'll also use a predicate 'Q_i' for each state q_i of M, so that $Q_i(n)$ means that at step n the machine is in state q_i. Finally, I'll use two binary predicates '@' and '1', where $@(n, x)$ means that at step n the machine is positioned on square x, and $1(n, x)$ that at step n there is a stroke in square x.

With this vocabulary, we can express the configuration of M on input I at every step n. For example, suppose M starts in state q_0 on input 11. Then the initial configuration can be expressed as follows.

$$Q_0(0) \wedge @(0, o) \wedge 1(0, o) \wedge 1(0, r(o)) \wedge \forall y(y \neq o \wedge y \neq r(o) \to \neg 1(0, y)).$$

We can also express how the configuration changes from one step to the next. For example, if M has an entry $\langle q_0, 1, \mathrm{B}, \mathrm{R}, q_1 \rangle$ in its machine table, then $S_{M,I}$ would have the following conjunct:

$$\begin{aligned}&\forall x \forall y((Q_0(x) \wedge @(x, y) \wedge 1(x, y)) \to \\ &\quad (Q_1(s(x)) \wedge @(s(x), r(y)) \wedge \neg 1(s(x), y) \wedge \forall z(z \neq y \to (1(s(x), z) \leftrightarrow 1(x, z))))).\end{aligned}$$

This says that if at some step x the machine is in state q_0 and positioned at some square y that contains a stroke, then at step $s(x)$ the machine is in state q_1, positioned at the square to the right of y, where the square y is now blank, and all other squares have the same content as before.

$S_{M,I}$ will be a big conjunction containing, first, the initial configuration of M on input I, then all the transition rules for M, and finally some background "axioms" to fix the intended interpretation of the non-logical symbols:

T1 $\forall x\, s(x) \neq 0$
T2 $\forall x \forall y(s(x) = s(y) \to x = y)$
T3 $\forall y(r(l(y)) = y \wedge l(r(y)) = y)$
T4 $\forall x(Q_0(x) \vee Q_1(x) \vee \cdots \vee Q_n(x))$
T5 $\forall x \forall y(Q_i(x) \to \neg Q_j(x))$ for $i \neq j$
T6 $\forall x \exists y @(x, y)$
T7 $\forall x \forall y \forall z((@(x, y) \wedge @(x, z)) \to y = z)$.

Let's turn to $H_{M,I}$. This is meant to say that M halts on input I. It is a disjunction, each disjunct of which corresponds to a state/symbol combination for which there is no entry in the machine table. For example, if there's no entry in the table for what to do in state q_1 when reading a blank, then $H_{M,I}$ will have the following as a disjunct:

$$\exists x \exists y(Q_1(x) \wedge @(x, y) \wedge \neg 1(x, y)).$$

This says that at some step x the machine is in state q_1 and positioned at a square y that is blank.

Theorem 6.3: Turing-undecidability of first-order logic (Turing 1936)

No Turing machine can decide whether any given first-order sentence is valid.

Proof sketch. Let M be any Turing machine and I any input for M. Let $S_{M,I}$ and $H_{M,I}$ be as above. We show that M halts on I iff $S_{M,I} \models H_{M,I}$. It follows that any Turing machine that solves the Entscheidungsproblem could be used to solve the halting problem.

Left to right. Suppose M halts on I after n steps. Let $\mathfrak{M}$ be any model of $S_{M,I}$. One can show by induction on n that $\mathfrak{M}$ satisfies the sentence describing the configuration of M on input I at step n. Since M halts on I at step n, it follows that $\mathfrak{M}$ satisfies $H_{M,I}$.

Right to left. Suppose M does not halt on I. We can then build a model $\mathfrak{M}$ of $S_{M,I}$ that does not satisfy $H_{M,I}$, by giving all the non-logical symbols their intended interpretation. □

I've omitted a lot of details here. Filling them in would take a few more pages. Since I'll give a full proof of Theorem 6.3, via a rather different route, in Chapter **??**, I'll save us the labour.

Exercise 6.15 The proof of Theorem 6.3 shows that M halts on I iff $S_{M,I} \models H_{M,I}$. Is it also true that M *doesn't* halt on I iff $S_{M,I} \models \neg H_{M,I}$? (a) Explain why this would contradict the undecidability of the halting problem, given the completeness of first-order logic. (b) Explain informally how it can be that M doesn't halt on I, but $S_{M,I} \not\models \neg H_{M,I}$.

The proof of Theorem 6.3 doesn't just show that no Turing machine can solve the Entscheidungsproblem. It shows more concretely that any Turing machine that could solve the Entscheidungsproblem could be converted into a Turing machine that solves the halting problem: if you gave a Turing machine an "oracle" for deciding validity in predicate logic – a magical subroutine that decides validity – then that machine could solve the halting problem. In this sense, the halting problem *reduces to* the Entscheidungsproblem. Many other problems have been revealed as undecidable in this way, by showing that their solution would yield a solution to the halting problem.

Exercise 6.16 Could the oracle Turing machine just described solve the halting problem for oracle Turing machines, or only for ordinary Turing machines?