8. TURING MACHINES

§ 8.1. The Limits of Computing

Computers have been around for only just over 80 years. In fact the world's first electronic computer was built in the year I was born. This was at Bletchley Park, in England, and was designed by Alan Turing in order to crack the Enigma Code. You may have seen the film *The Imitation Game*.

During a university vacation in the early 1960's I worked on what was then one of only about six computers in Australia, at the Weapons Research Department associated with the Woomera Rocket Range. I can't tell you much about the project I was assigned to – it's probably still classified! Suffice to say that it had something to do with rockets and the chemistry of the upper atmosphere.

The computer occupied a large hall in the hut where I worked but I never got to see it. We apprentice programmers wrote out our programs on coding sheets. These were sent to a room full of punch girls who converted the program into punched cards. They didn't trust us to punch our own, and there was a sexist policy that men had the brains to write programs while girls could cope with repetitive tasks like typing. How the world has changed!

A day or so later we got back a deck of these punched cards. We had to check them and then submit them into a hole in the wall. On the other side they were taken out and loaded onto the computer. All we knew was that the next day, if we were lucky, we would find our deck of punched cards returned through the same hole in the wall, wrapped up in the printout from the computer.

All too often the printout merely said something like "syntax error, too many left parentheses". Then we had to find the offending card and have it re-punched. But, if we were lucky, the printout would contain rows and rows of numbers – the answers to our calculations.

As computer hardware and software continue to develop at an ever increasing rate we'd be forgiven for believing that no problem is too hard for a computer, at least in principle. Given enough memory, enough time and enough ingenuity on the part of the programmer no problem that can be precisely stated, is impossible.

After all, hasn't short-sighted man been proved wrong in the past when he declared that certain things are impossible? How can we place limits on what future generations can achieve? Yet there *are* problems that are inherently unsolvable by a computer (and so presumably by any other means). They are problems for which, if a program were to exist (whether or not there was a machine big enough and fast enough to actually perform it) a logical contradiction would result.

Those who train future computer scientists consider it desirable for their students to encounter the phenomenon of non-computability in order to give them some perspective on the work they will perform.

But computability is also relevant to the more logically fundamental parts of mathematics. Consider the real numbers, for example. Most of them are irrational and so can't be defined by writing out their complete decimal expansion. But we can get a hold of numbers like $\sqrt{2}$ and π to the extent that we could write a computer program which, if left to run forever, would print out all their digits. Unless we can do that there is a sense in which we don't really know the real number. But with uncountably many real numbers and only countably many potential computer programs, most real numbers are inaccessible to human thought! What is even more interesting is that there are some real numbers that can be precisely defined, yet no computer program can be devised to print out its digits.

§ 8.2. The Halting Problem

Anyone who has ever written a program for a computer will know that there's always the possibility for the program to get itself into an infinite loop. For example in a very old programming language called BASIC:

```
10: X = X + 1
20: GO TO 10
```

would cause the computer to carry out the same instruction over and over, for ever.

Such an obvious mistake is, of course, easy to avoid but loops can be created in all sorts of subtle ways. It would be nice if the compiler (a program which converts your high-level program into machine code) would alert you to the fact that your program will loop. After all it can tell you that you have left out a comma so why not check for loops.

Certainly it's easy to detect obvious loops like the one above, but how would the compiler search out *all* potential loops? Certainly not by trying to run the program! A program could run all day and all night and appear to be going around in circles, but so long as something is changing each time the program may halt eventually. Surely a clever programmer would be able to write some code that examines the structure of your program and find out, without actually running it, whether or not your program will ever halt.

The halting problem can be loosely stated as:

Halting Problem: Write a program that will determine whether or not any given program will halt when it starts with a given piece of data as input.

Now the problem can be solved for certain artificial or trivial languages. For example a program written in a language that has no branching or looping capabilities must inevitably terminate when it reaches the last instruction. That's an example of what I mean by a trivial language. But for all languages in which one can perform useful work, the problem is unsolvable. Such a program cannot logically exist.

Languages in which one can perform useful work include all languages that are actually used, ranging from BASIC with its notorious propensity for loops, to highly structured languages such as PASCAL, C++ or C#.

§ 8.3 Definition of a Turing Machine

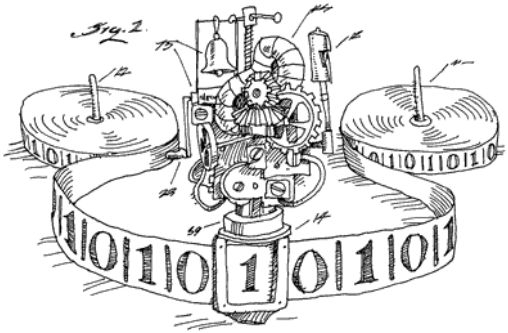
A little before the first computers were built, many mathematicians and logicians were trying to come up with a satisfactory definition of a 'computable function'. For them, a loose definition was a function that could be performed by an untiring and meticulous clerk who could accurately and tirelessly follow instructions involving a limited range of basic operations.

Several intuitively plausible definitions were considered for computability. While there is no way of deciding whether any of them is the 'right' one, the fact that they have all been shown to be equivalent suggests that, indeed, we do indeed have the right definition.

Alan Turing created a model that consists of a conceptual machine (i.e. one that exists in the mind only).

He defined a function to be computable if it can be computed by a Turing Machine. It's interesting that he described such imaginary machines in the 1930s and went on to become part of the team that built the world's first computer in the early 1940s. Turing is best known for having cracked the Enigma Code and for building the world's first computer. (The Americans claimed to have built the first computer but that's because the work at Bletchley Park was kept highly secret, even from the American government, until a couple of decades later.) Turing is less well known for his earlier work on computability, using his imaginary machines as an important tool. These are now known as Turing Machines.

A **Turing Machine** is an imaginary machine that can move along an infinitely long paper tape. The tape is marked off into squares and each square is either blank or contains a mark. Only finitely many squares are non-blank. We will denote a blank by '0' and the mark (non-blank) by '1'.



Please note that these machines are called **TURING** machines in honour of Alan Turing. Students often refer to them as ‘touring’ machines or ‘turning’ machines, probably because these adjectives are vaguely appropriate.

At any stage the machine can be in any one of a finite set of states. There’s a set of instructions (set out in a table) which defines the behaviour of the machine as it moves up and down the tape, reading and writing, and changing states.

If the machine is in a given state and reading a certain character there’s an instruction that dictates that the machine should print a certain character (erasing whatever the square previously contained), move in a certain direction (one square left or right), and go into a certain state. It’s like a Finite State Machine, but with the ability to read what it has written.

The ongoing behaviour of the machine is determined not only by its instruction set but also by the data on the tape (what is on the tape and the position of the head) and the current state. This combination of tape, position of the head and the state of the machine at any given stage is called an **instantaneous description** (**ID**). The TM can then be thought of as a function on the set of instantaneous descriptions.

An **instantaneous description** can be represented in following way:

[state] tape

where a fragment of the tape is shown that contains all the 1's. Anything to the left or right of the fragment is assumed to be 0. A dot "**.**" is interspersed to show the position of the head. The symbol being scanned is immediately to the right of the dot. As an aid we will often include the step number, to the left of the tape, though strictly speaking this doesn't form part of the ID.

Example 1: A typical ID is

[5] 00**.**1011101

indicating that the tape has the form:

….. 0 0 0 0 0 1 0 1 1 1 0 1 0 0 0 …..,

the head is scanning the left-most 1 and the machine is in state 5.

If this was the ID after 13 steps we might provide this additional information as

13⟩ [5] 00**.**1011101

The **trace** of a Turing machine, given certain input data, is the list of successive instantaneous descriptions, up to some point (usually until it halts – that is if it *does* halt).

If *N* is a natural number we define an **N state Turing Machine** (**TM** for short) as a table with two columns and N rows. The columns are labelled 0 and 1

and the rows are labelled, sequentially, 0 to N – 1. Each cell in the table contains an instruction of the form *m*D*n* where $m \in \{0, 1\}$, $n \in \{0, 1, 2, \ldots, N + 1\}$ and $D \in \{L. R\}$.

Example 2: The following is a 4 state TM.

	0	**1**
0	0 L 1	1 R 1
1	1 L 0	0 L 2
2	1 L 3	0 R 1

N-STATE TURING MACHINE
Start in state 0. Halt in state N.
If state = *s*, character scanned = *c*, obey the instruction in the *s*-*c* cell of the table.
***m*D*n* MEANS:**
- **PRINT *m***
- **MOVE one square in direction D (L = left, R = right)**
- **GO TO STATE *n***

Example 3: Let's operate the TM in example 1 with the tape … 1**.**1 0 1 …

We begin in state 0, scanning the symbol immediately to the right of the full stop. So we are in state 0 reading 1. We look up the 0-1 cell in the table and find the instruction 1R1. So we print 1 (it was already 1 so

there is no change to the tape), we move right and go to state 1.

The instantaneous description is now:

[1] … 1 1.0 1 …

Now we are in state 1, reading 0.
The instruction here is 1R3.

The trace is as follows (we provide the step number and the instruction obeyed for convenience):

0> [0] … 1.1 0 1 … [1R1]
1> [1] … 1 1.0 1 … [1L0]
2> [0] … 1.1 1 1 … [1R1]
3> [1] … 1 1.1 1 … [0L2]
4> [2] … 1.1 0 1 … [0R1]
5> [1] … 1 0.0 1 … [1L0]
6> [0] … 1.0 1 1 … [0L3]
7> [3] ….1 0 1 1 … HALT

Unlike finite state machines, which halt once they run out of input data, a Turing machine can chew over its data indefinitely. Only when it's sent to a non-existent state (represented by the next number after the last state) does it halt. If it never reaches such a state the machine runs forever.

AN N-STATE TURING MACHINE ALWAYS STARTS IN STATE 0 AND HALTS WHEN SENT TO STATE N

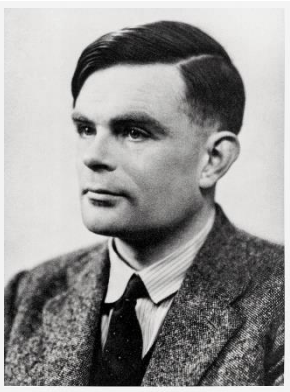
Example 4:
This TM has three states denoted by 0, 1 and 2 and two characters 0 and 1.

	0	**1**
0	0 L 1	1 R 0
1	1 R 1	0 R 2
2	1 R 3	0 R 2

If the machine begins with 11 on the tape, with all the remaining squares blank (represented by 0) and with the head on the leftmost "1", the trace is:

0⟩ [0] 0.110
1⟩ [0] 01.10
2⟩ [0] 011.0
3⟩ [1] 01.10
4⟩ [2] 010.00
5⟩ [3] 0101.00

At this stage the machine halts. Of course if a Turing machine fails to halt we can only ever record a finite portion of its trace.



§ 8.4. Old Notation

Alan Turing was born on 23 June 1912, in London. He studied mathematics at Kings College, Cambridge. During the war he

worked at the secret coding establishment at Bletchley Park, where he cracked the Enigma code and designed the world's first computer, Colossus.

He was a homosexual and was convicted under the Homosexuality Act. He chose chemical castration, rather than go to prison. He died on 7th June 19543 of cyanide poisononing. An inquest found that it was suicide. A half-eaten apple at his bedside was found to have traces of cyanide and he is said to have loved the fairy tale *Snowhite and the Seven Dwarfs* and had recently seen the Disney film.

But later evidence seems to go against that theory. One of his hobbies was electronics and he used cyanide in those experiments, so it is highly likely that the apple became contaminated.

In 2009 the then British Prime Minister made an official apology for his conviction, on behalf of the government and in 2013, after a campaign to clear his name, Queen Elizabeth granted him a posthumous pardon. A law that retrospectively pardoned all those who were convicted under the old Repression of Homosexuality law, is informally known as the Turing Law.

Alan Turing's paper, *On Computable Numbers with an Application to the Entscheidungsproblem*, was published in 1936. (The word *Entscheidingsproblem*) means 'decision problem'.)

His notation was much clumsier than that which we use here. You can Google his paper and see if you can

follow it. (I don't mean to belittle him. It's very common for a new mathematical idea to be somewhat obscure when it is first discovered, with later accounts being expressed much more simply and elegantly.)

Later accounts of what have become known as Turing Machines, many of which are still used in textbooks today, are based on systems of *quintuples*.

Here there are more than two symbols, called c_1, c_2, …, c_m and the states, or configurations, are labelled as s_1, s_2, …, s_n. Each instruction is coded as a 5-tuples, of the form $s_ic_jc_kLs_p$, $s_ic_jc_kRs_p$, or $s_ic_jc_kNs_p$, where $s_ic_jc_kLs_p$, for example, meant "if the machine is in state s_i, and is reading symbol c_j it moves one square to the left, and goes into state s_p". The symbol, R, represents 'right' and N represents 'no movement'.

The instructions could be in any order but, so that the machine is deterministic, the assumption is made that no two instructions began with the same two symbols. If, for a given combination of state, s_i, and symbol, c_j there's no quintuple starting with s_ic_j the machine halts.

Example 5: Using quintuples the TM in example 4 as follows:

	0	**1**
0	0 L 1	1 R 0
1	1 R 1	0 R 2
2	1 R 3	0 R 2

$s_1c_1c_1Ls_2$
$s_1c_2c_2Rs_1$
$s_2c_1c_2Rs_2$
$s_2c_2c_1Rs_3$
$s_3c_1c_2Rs_4$
$s_3c_2c_1Rs_3$

§ 8.5. Unary Representation of Numbers

In considering functions on the set of natural numbers we need a suitable format for them. An obvious choice is to use binary notation. After all, that's how numbers are stored in a real computer.

The problem is that we'd have no way of representing the last bit. Representing the number 9 we would have the binary representation for 9, that is 1001 with infinitely many 0's to the left and the right:

…….. 00**.**100100 ……..

The number 18 would consist of the binary representation of 18, that is 10010, preceded and followed by 0's:

…….. 00**.**10010 ……..

Both numbers would be represented identically. With binary notation we'd not be able to distinguish between 18, 36, 72 etc since doubling a number in binary means tacking an extra 0 onto the end. These extra 0's would get lost amongst the infinitely many 0's that would have to follow the binary representation.

But there's an even simpler representation than binary and that is the **unary** system. As well as solving

the above problem, it is simpler than binary. We just represent the natural number *n* by a string of *n* 1's? Certainly that will make adding 1 very easy – just write an extra ‘1’ at the end!

This might seem a clumsy arrangement, and certainly if we had to use TMs for any practical purpose we’d get sick of enormous strings of 1's (imagine writing the date in unary!). But our concern here is not convenience or speed but simply whether or not something can be done at all. The number 0 would be then represented by a completely blank tape.

One difficulty with this is that a TM has no way of recognising a blank tape. If the head is located on a blank square the number represented by the tape might be zero, or, there might be some 1’s located somewhere at some distant part of the tape. The TM can search the tape, alternately left and right, but after a finite number of steps it won’t not know whether the tape is really blank or whether it just hasn't gone far enough to locate the 1’s.

To overcome this difficulty we adopt the convention that with a non-blank tape, the head always starts and finishes on the first (left-most) ‘1’. (If the tape is blank then of course it doesn’t matter where the head starts.) So if the square being scanned at the start is blank, it is assumed that the entire tape is blank and that the input is the number zero.

THE NATURAL NUMBER *n* IS REPRESENTED BY A STRING OF *n* 1's AND IF *n* > 0, THE HEAD IS ON THE LEFTMOST 1

Example: … 000**.**1111111000 … is the unary representation of 7. Note the position of the dot, representing the position of the read/write head. And don't confuse it with a decimal point.
… 000**.**0000 … represents the number zero.

That's fine, but what if the input consists of more than one natural number? How do we represent the ordered pair (*m*, *n*)? In that case we write the numbers in unary and separate each of them from the next by a single 0. The head is to be placed scanning the left-most '1'.

THE *k*-TUPLE (n_1, n_2, …, n_k) IS WRITTEN IN UNARY AS FOLLOWS:

- **Remove the parentheses and any 0's**
- **Replace every comma by 0**
- **Convert every positive number to unary**
- **Put the dot for the head at the left – all other squares are blank (ie zeros)**

Example 6:

(0, 0, 3, 0, 1, 5) → , , 3 , , 1 , 5
→ 00300105
→ 00111001011111
→ **.**00111001011111
→ …000**.**00111001011111000 …

You may have spotted a potential problem with this arrangement. What if the last component is zero? What if we have several zero components at the end?

Example 7:

… 000**.**1110000 ...

could represent the number 3, or it could represent the pair (3, 0), or the triple (3, 0, 0) etc.

The answer is that the TM, as we have set it up, can't distinguish between these alternatives. We could introduce a comma as a third symbol. Or we could code a natural number *n* by a sequence of 1's of length *n* + 1. But that isn't necessary. We'll assume that anyone who's operating a TM knows the format of the input. After all a piece of music is coded in a real computer as a sequence of 0's and 1's and these might also represent an image. Feed the Moonlight Sonata into an imaging program and you might be able to see it! Or feed the Mona Lisa into a music program and you might be able to hear her!

In practice the binary representations of an image and a piece of music can actually be distinguished and

you'll probably just get an error message. But anyone who's inadvertently used Microsoft Word to open a PDF file will know what I mean.

So we don't worry about this ambiguity. If we have a TM to add two numbers we expect the TM to interpret the input as an ordered pair. If the TM is supposed to decide whether one number is greater than the sum of two others, it will assume that the input is a triple.

You may ask how do we represent decimal numbers ? We could devise some method for doing it. Any data that is input into a real computer is just a finite sequence of 0's and 1's. So all we need is to include a new symbol for the beginning and the end, such as '|'.

So do we need to generalise our TMs to use more than 0's and 1's? Not really. We can code a zero as 00 and a 1 as 11. That leaves 01 and 10 as possible codes for other symbols. So there's nothing to stop us coding the Mona Lisa, or the Moonlight Sonata on a TM. The only problem would be that the TM has no hardware to be able to see or hear them. But we could still construct a TM that can convert a piece of music to a different key, or rotate an image.

Example 8: We might code the binary sequence as 1001110 as 101100001111110010, where 10 represents the beginning and end of the sequence.

However we'll stick to Turing Machines that operate on binary strings. We have to convince ourselves

that whatever can be done on a real computing device can be done on a binary Turing Machine. In fact the humble TM, because of its infinitely long tape, can do things that no computer in the world can do. For example a TM could multiply two integers with 10^{80} digits. Since that is roughly the estimated number of atoms in the whole universe it's hard to imagine a real computer ever doing this.

But in a much larger universe it could be done. We don't want to base our definition of computability on the size of a computer's memory, so we define something to be computable if and only if it can be done on a Turing Machine.

Once you have examined a number of Turing Machines you will hopefully reach two conclusions:

(1) Turing Machines are totally impractical for any practical computing task. That's why they remain an intellectual tool – nobody actually builds them. There do exist some demonstration models but the problem is the infinitely long paper tape. Someone suggested using rolls of toilet paper joined one to another, but these Covid days it's getting hard to find them – and besides, even using all the toilet rolls in the world wouldn't give us an infinitely long tape.

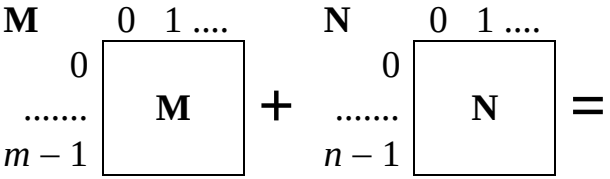
(2) Whatever can be computed, using any conceivable computer language can be done on a binary Turing Machine.

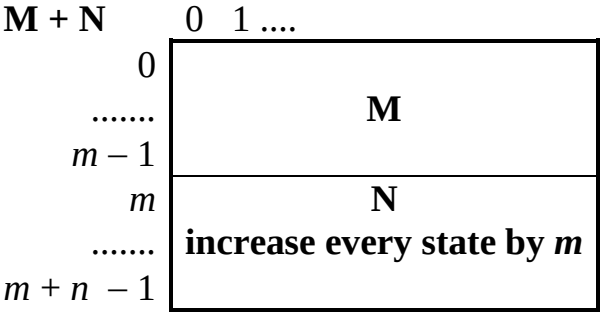
The latter may take some time for you to accept. But there is no way we could ever prove it, because of the difficulty of defining a 'conceivable computer language'. Instead we shall define a task to be **computable** if it can be done on a Turing Machine. Our next task is to provide evidence for (2) and for this we need to develop some tools for constructing more complex TMs from simpler ones. For the time being we shall assume that we are only dealing with binary TMs.

§ 8.6. Adding Turing Machines

Suppose we have two TMs M and N. The instruction table for the sum M + N is obtained by adjoining the table for N underneath that for M and increasing the number of every state in N by the number of states in M.

Then, if machine M halts, it passes straight into machine N with the data that M left behind. Of course if M doesn't halt then N never gets a look in!





Example 9:

M	0	1
0	0 L 1	1 R 0
1	1 L 2	0 R 1

+

N	0	1
0	1 R 1	0 L 2
1	1 L 0	1 R 1

=

M + N	0	1
0	0 L 1	1 R 0
1	1 L 2	0 R 1
2	1 R 3	0 L 4
3	1 L 2	1 R 3

§ 8.7. Sample Turing Machines

The following are some examples of TMs. They demonstrate how some of the fundamental tasks of a computer can be done by a TM.

Example 10:
This TM computes the function ZERO(n) = 0 for all n.

ZERO⟩ n = 0

	0	**1**
0	0R1	0R0

Example 11:
This TM computes the function INC(n) = n + 1.

	0	**1**	
0	1L1	1L0	Add 1 to left
1	0R2		Reset head

We shall use an empty cell for a combination which will never be reached. Also, we'll often provide some explanation for what is happening in each state.

Example 12:
If the tape contains a single '1' (all the rest blank) somewhere to the right of the head, this TM will find the '1', halting on that square.

	0	1	
0	0R0	1L1	Found the "1"
1	0R2		Halt

Note that because we have to move left or right every time we need state 1 in order to halt at the correct square.

Example 13: This TM computes the function REPEAT(n) = (n, n).

	0	1	
0	0L5	0R1	Raise flag
1	0R2	1R1	Copy a 1
2	1L3	1R2	
3	0L4	1L3	Lower flag
4	1R0	1L4	
5	0R6	1L5	Reset head

Example 14: This TM computes the function ADD(m, n) = $m + n$.

	0	1	
0	1R1	1R0	Plug the gap
1	0L2	1R1	Find the right hand end
2		0L3	Remove 1 from right
3	0R4	1L3	Reset the head

Plugging up the gap between the two parameters will give $m + n + 1$, so we need to take off a '1' from one

end. Instead we could also have removed the ‘1’ at the left-hand end – this would have resulted in a smaller Turing Machine:

	0	**1**	
0	0R3	0R1	Remove 1 from the left
1	1L2	1R1	Plug up the gap
2	0R3	1L2	Reset the head

As a stand-alone program this would be fine. But if we wish to add this to another machine there could be problems if there was another parameter before the two that are involved in the addition. The second version would move the sum $m + n$ away from the earlier parameter. So if the input was (r, m, n) the output would be (r, 0, $m + n$) not (r, $m + n$). For example if the input was (4, 3, 2) the second version would give

[3] 111100**.**11111

as the output while the first version gives

[4] 11110**.**11111

This becomes an issue if we wanted to calculate 2^n by repeated doubling (see example 13).

To avoid this problem with arithmetic programs we should adopt the convention that, not only should the head start and finish on the left-most ‘1’, but that this position

should be the same for input and output and nothing to the left of this position should be changed.

This could be remedied by insisting that the infinitely long paper tape is only infinite at one end, and that the head is assumed to start and finish on the leftmost square. But there would be a problem in knowing when we were on that square. So we'll stick to a 2-way infinite tape.

Example 15: This TM computes the function DOUBLE(n) = 2n.

DOUBLE = REPEAT + ADD
Writing DOUBLE out in full we get:

	0	**1**	
0	0L5	0R1	**REPEAT**
1	0R2	1R1	
2	1L3	1R2	
3	0L4	1L3	
4	1R0	1L4	
5	0R6	1L5	
6	1R7	1R6	**ADD**
7	0L8	1R7	
8		0L9	
9	0R10	1L9	

Example 16: This TM computes the function HALVE(n) = the integer part of n/2.

	0	1	
0	0L4	0R1	Raise flag
1	0L2	1R1	Remove 1 from right
2	0L4	0L3	
3	1R0	1L3	Lower flag
4	0R5	1L4	Reset head

Example 17: This TM computes the function QUARTER(*n*) = the integer part of *n*/4.
Since INT(*n*/4) = INT(INT(*n*/2)/2) we can construct the machine QUARTER to compute INT(*n*/4) as HALVE + HALVE, as follows:

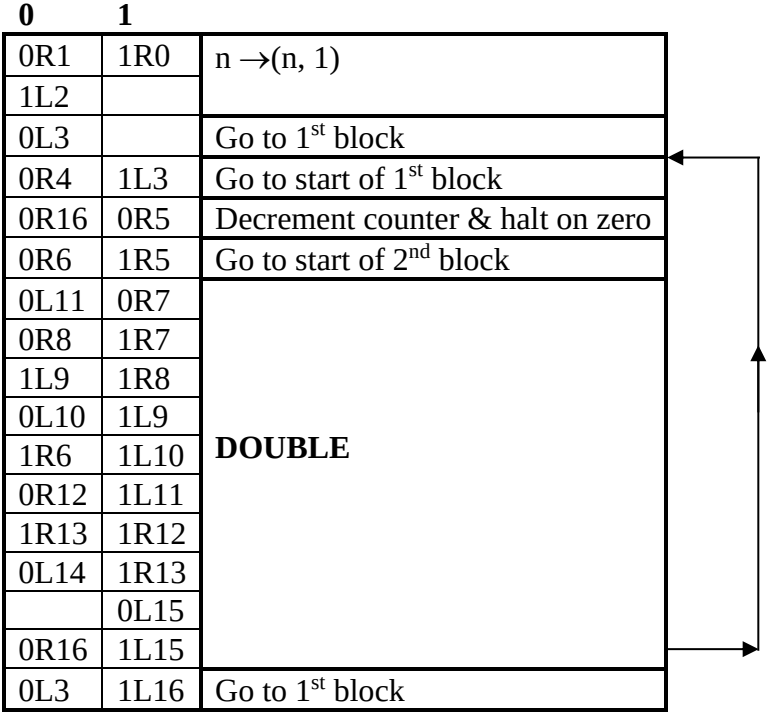
	0	1	
0	0L4	0R1	first copy of HALVE
1	0L2	1R1	
2	0L4	0L3	
3	1R0	1L3	
4	0R5	1L4	
5	0L9	0R6	second copy of HALVE
6	0L7	1R6	
7	0L9	0L8	
8	1R5	1L8	
9	0R10	1L9	

Alternatively, we can design it from scratch in a way that could be easily adapted to INT(*n*/*k*) for any fixed *k*.

	0	1	
0	0R8	0R1	Erase a group of 4
1	0R8	0R2	

2	0R8	0R3	Halt if all gone
3	0R8	0R4	
4	0R5	1R4	Go to end of 2nd block and add 1
5	1L6	1R5	
6	0L7	1L6	Return to start of 1st block and continue
7	0R0	1L7	

Example 18: This TM computes the function 2^n.



	0	**1**	
0	0R1	1R0	n →(n, 1)
1	1L2		
2	0L3		Go to 1st block
3	0R4	1L3	Go to start of 1st block
4	0R16	0R5	Decrement counter & halt on zero
5	0R6	1R5	Go to start of 2nd block
6	0L11	0R7	**DOUBLE**
7	0R8	1R7	
8	1L9	1R8	
9	0L10	1L9	
10	1R6	1L10	
11	0R12	1L11	
12	1R13	1R12	
13	0L14	1R13	
14		0L15	
15	0R16	1L15	
16	0L3	1L16	Go to 1st block

Example 19:

In example 7 we were able to locate a single '1' on an otherwise blank tape if we could be sure it was at or to the right of the head. But what if we don't know that? In

that case we need to look both left and right in ever-increasing sweeps.

We'd need to be able to know, when we're looking in one direction, how far we got last time. We do this by making clever use of pointers. We put down two extra 1's next to each other on the tape and gradually move them apart, in both directions. The portion between the two 1's represents the territory we've examined and which contains no '1'. As we move the left pointer to the left or the right marker to the right we check whether there's already a '1' on that square. If so, this is the '1' we had to find. But finding the '1' is not quite enough because we will then have to erase the remaining pointer and then return to the '1' that we found.

FIND:

	0	**1**	
0	1R1	1L7	find 1 straight away
1	1L2	1L8	find 1 on right
2	0L2	0L3	
3	1R4	1R5	find 1 on left
4	0R4	0R1	
5	0R5	0L6	tidy up right pointer
6	0L6	1L7	
7	0R10		halt on the '1'
8	0L8	0R9	tidy up left pointer
9	0R9	1L7	

Of course if the tape is completely blank this TM will never halt.

EXERCISES FOR CHAPTER 8

EXERCISES 8A (Operating TMs)

Ex 8A1: Describe the behaviour of the following Turing Machine when started with a blank tape (a blank is represented here by the symbol '0') and show that it halts.Do this by giving the successive instantaneous descriptions of the machine, showing at each stage the contents of the tape and the position of the head.

	0	**1**
0	1L1	1R0
1	1R2	0L2
2	1R3	0L0

Ex 8A2: Run the following TM, starting with a blank tape, until it halts.

	0	**1**
0	1R1	1R2
1	1R2	1R3
2	1R3	0R1
3	1L4	1R5
4	0R0	1L4

Ex 8A3: (a) Describe the behaviour of the following TM, when started with a blank tape.

	0	**1**
0	1R1	1R0
1	1R2	1R3
2	1R3	0R1
3	1L4	1R5
4	0R0	1L4

(b) Find a string, α, of length 7 such that the above TM will halt when it starts with α on the tape with the head on the leftmost '1'.

Ex 8A4: Using the unary representation for natural numbers carry out the following arithmetic Turing Machine on various input of the form (*m*, *n*) for small values of *m* and *n*. State in words what function *f* (*n*) this machine appears to be carrying out.

	0	**1**
0	0R6	0R1
1	0R2	1R1
2	0L3	1R2
3	0L7	0L4
4	0L5	1L4

	0	**1**
5	0R0	1L5
6	0R9	1L8
7	0L8	0L7
8	0R9	1L8

EXERCISES 8B (Constructing TMs)

Ex 8B1: Construct a Turing Machine to implement the following arithmetic function:

$$f(n) = \begin{cases} n-2 & \text{if } n \geq 2 \\ 0 & \text{if } n = 0 \text{ or } 1 \end{cases}$$

The input n is given by a sequence of n consecutive 1's with all other squares blank (0 = blank). The head must start and finish on the left-most 1 (anywhere if there are no 1's).

Ex 8B2: Construct a Turing Machine that erases a non-blank binary tape. The head starts somewhere to the left of the data. You may assume that the tape is not blank and that if ever you read 000 on three successive squares you've reached the end of the non-blank portion of the tape.

Ex 8B3: Construct a Turing Machine to compute the function $f(x) = x + 2$. (The input and output is to be in unary.)

Ex 8B4: Construct a Turing Machine to compute the function $f(x, y, z) = x + y + z$. (The input and output is to be in unary.)

Ex 8B5: Construct a Turing Machine to compute the function $f(x) = (x, x, x)$. (The input and output is to be in unary.) [**HINT:** Adapt the machine REPEAT as given in the notes. Once a second copy of x has been made, instead

of the head returning to the very beginning it should return to the beginning of the second copy. Then use the same instructions (but with a fresh set of states) to copy the second copy, making the third.]

Ex 8B6: Construct a Turing Machine to compute the function $f(x) = 3x$. (The input and output is to be in unary.) [**HINT:** Combine two of the machines you have already created.]

Ex 8B7: Construct a Turing Machine which takes a sequence 1111...1 of n consecutive 1's, with $n \geq 1$, and halts with the pattern 101010....01 consisting of n 1's with a single 0 between each pair of consecutive 1's. The head should start and finish on the left-most 1. All squares to the left and right of those indicated are assumed to be blank (i.e. 0).

Ex 8B8: Construct a Turing Machine which will compute the function

$$f(x) = \begin{cases} x & \text{if } x \text{ is even} \\ x + 1 & \text{if } x \text{ is odd} \end{cases}$$

Input and output are to be in unary form.

Ex 8B9: Construct a Turing Machine to compute the function $f(x) = (x + 1, 2)$.

Ex 8B10: (a) Construct a Turing Machine to calculate the function $f(m, n) = |m - n|$ with input and output in unary notation.

(b) Operate your Turing Machine for the following cases showing the successive instantaneous descriptions until the machine halts (or for 30 steps, whichever comes first). If you manage to get your machine to halt within 30 steps write down the number of steps taken.
(i) $m = 3, n = 1$; (ii) $m = 1, n = 2$; (iii) $m = 1, n = 1$.
HINT: Take 1's alternately from m and n. When one of them runs out, the other becomes the absolute value of the difference. It then remains to move the head to the right place. But there are several ways you can chop off the 1's. You could take them from the left-hand end of each, or the right-hand end of each, or from the middle, or from the outside in. Three of these four possibilities might lead to difficulties. The other works smoothly. Experiment!

Ex 8B11: Construct a Turing Machine to compute the function $f(x, y) = (x + y, 1)$.

Ex 8B12: Design a Turing Machine to compute the following functions:

(i) $f(n) = \text{INT}(n/3)$;
(ii) $g(m, n) = m + n + 3$;
(iii) $h(n) = n - 1$ (if $n > 0$) and 3 if $n = 0$.

Ex 8B13: Design a 23-state Turing Machine to compute the function $f(n) = 2^{2^n}$. Input and output are to be in unary notation.
[**HINT:** Use the 17 state machine that computes 2^n in example 19.]

Ex 8B14: Construct a binary TM to calculate the function $f(a, b) = a + b + 2$.

Ex 8B15: Construct a binary TM to calculate the function $f(n) = \text{INT}(n/3)*3$ with n in unary notation. In other words the TM takes off one or two 1's, where necessary, to get a multiple of 3. The output should be in unary form with the head positioned appropriately.

Ex 8B16: Design a Turing Machine to compute the function $f(x, y) = \text{INT}(\text{AVERAGE}(x, y)) + 2$.

Ex 8B17: Design a Turing Machine to compute the function: $f(m, n) = \begin{cases} m + n \text{ if } m + n \text{ is even} \\ 0 \text{ otherwise} \end{cases}$

SOLUTIONS FOR CHAPTER 8

Ex 8A1:

0] [0] 0.0
1] [1] 0.01
2] [2] 1.1
3] [0] 0.1
4] [0] 1.0
5] [1] 0.11
6] [2] 0.001
7] [3] 1.01

Ex 8A2:

0] [0] 0.0
1] [1] 1.0
2] [2] 11.0
3] [3] 111.0
4] [4] 11.11
5] [4] 1.111
6] [4] 0.1111
7] [4] 0.01111
8] [0] 0.1111
9] [2] 1.111
10] [1] 10.11
11] [3] 101.1
12] [5] 1011.0 HALTS

Ex 8A3: (a) Since the only difference between this and the TM in the previous example is the instruction for state 0, reading a "1" the first 8 steps will be as above. Continuing:

9] [0] 11**.**11
10] [0] 111**.**1
11] [0] 1111**.**0

During these 11 steps the head never moves to the left of its initial position so these four 1's will be left undisturbed and in the next 11 steps a further eleven 1's will appear. Clearly this TM will never halt when started with a blank tape.
(b) The TM halts if it reads a '1' in state 3. Examining the trace for a blank tape we see that if we started 0**.**0001 we would be reading the '1' while in state 3, and so halt. Moreover, since the instruction for state 0, reading a '1' is 1R0 we could insert any number of 1's in front of the 0001, so that the TM will halt when started with 0.1^n0001, for any $n \geq 0$. To have $|\alpha| = 7$ we need $n = 3$, so a suitable α is 1110001.

Ex 8A4: This TM computes the function:

$$f(m, n) = \begin{cases} n - m & \text{if } n \geq m \\ 0 & \text{if } n < m \end{cases}.$$

Ex 8B1:

	0	**1**
0	0R2	0R1
1	0R2	0R2

Ex 8B2:

	0	**1**
0	0R0	0R1
1	0R2	0R1
2	0R3	0R1

Ex 8B3: This is a typical solution, working from scratch.

	0	**1**	
0	1L2	1L1	move left
1	1L2		add 1
2	1L3		add 1
3	0R4		reset head

However, since $f(x)$ = INC(INC(x)), the simplest thing to do is construct INC + INC.

	0	**1**
0	1L2	1L1
1	1L2	
2	1L4	1L3
3	1L4	

Ex 8B4: This is a typical solution, working from scratch.

	0	**1**	
0	1R1	1R0	remove 1st separator
1	1R2	1R1	remove 2nd separator
2	0L3	1R2	move to end
3		0L4	take off 1
4		0L5	take off 1
5	0R6	1L5	reset head

However note that ADD + ADD will achieve the same result.

Ex 8B5: Version A uses two copies of REPEAT with a little extra head-moving. It clearly involves some unnecessary head-moving and could be improved on slightly. Version B is slightly more efficient.

A	**0**	**1**	
0	0L5	0R1	REPEAT
1	0R2	1R1	
2	1L3	1R2	
3	0L4	1L3	
4	1R0	1L4	
5	0R6	1L5	
6	0R7	1R6	Move head to 2nd copy
7	0L11	0R8	REPEAT
8	0R9	1R8	
9	1L10	1R9	
10	0L11	1L10	
11	1R7	1L11	
12	0R13	1L12	
13	0L14	1L13	Move head to start of original
14	0R15	1L14	

B	**0**	**1**
0	0R5	0R1
1	0R2	1R1
2	1L3	1R2
3	0L4	1L3
4	1R0	1L4
5	0L10	0R6
6	0R7	1R6
7	1L8	1R7
8	0L9	1L8
9	1R5	1L9
10	0L11	1L10
11	0R12	1L11

Ex 8B6: Add two copies of ADD to the above TM. Then $x \to (x, x, x) \to (2x, x) \to 3x$.

Ex 8B7:

	0	**1**
0	0R6	0R1
1	0R2	1R1
2	1L3	1R1
3	0L4	1L5
4	0R0	1L3
5	0R0	1L5

Ex 8B8:

	0	**1**
0	0L2	1R1
1	1L2	1R0
2	0R3	1L2

Each step consists of removing a 1 from the right and advancing the marker (the marker is a 1 which is temporarily erased).

Ex 8B9:

	0	**1**
0	1R1	1R0
1	0R2	
2	1R3	
3	1L4	
4	0L5	1L4
5	0R6	1L5

Ex 8B10:

We alternately take 1's from *m* and *n*, working from the outside in, starting with *n*.

	0	**1**	
0	0R1	1R0	take 1 off RH end
1	0L2	1R1	
2	0L6	0L3	
3	0L4	1L3	take 1 off LH end
4	0R5	1L4	
5	1L8	0R0	
6	0R7	1L6	when these ends meet
7	0R8		reset the head

Ex 8B11:

	0	**1**	
0	1R1	1R0	insert '1' in gap
1	0L2	1R1	go to extreme right
2	1L3	0R2	move final '1' one square to right
3	0L4		reset
4	0L5	1L4	head

Ex 8B12:

(i)

	0	**1**	
0	0R7	0R1	Erase a group of 3
1	0R7	0R2	Halt if all gone
2	0R7	0R3	
3	0R4	1R3	Go to end of 2nd block and add 1
4	1L5	1R4	
5	0L6	1L5	Return to start of 1st block and continue
6	0R0	1L6	

(ii)

	0	**1**
0	1R1	1R0
1	1R2	1R1
2	1L3	
3	0R4	1L3

(iii)

	0	**1**
0	1R1	0R4
1	1R2	
2	1L3	
3	0R4	1L3

Ex 8B13:

	0	**1**
0	0R1	1R1
1	1L2	
2	0L3	
3	0R4	1L3
4	0R23	0R5
5	0R6	1R5
6 … **22**	2^n	
23	0L3	1L22

Ex 8B14:

	0	**1**	
0	1L1	1R0	Plug up gap
1	1L2	1L1	Go left and add 1
2	0R3		Halt

Ex 8B15:

	0	**1**	
0	0L3	1R1	Count out a
1	0L4	1R2	block of
2	0L5	1R0	three 1's
3	0R6	1L3	Reset head
4		0L3	Erase
5		0L4	superfluous 1's

Ex 8B16:

	0	1	
0	1R1	1R0	ADD
1	0L2	1R1	
2		0L3	
3	0R4	1L3	
4	0L8	0R5	HALVE Slightly modified to add 1 as well
5	0L6	1R5	
6	0L8	0L7	
7	1R4	1L7	
8	1L9	1L8	
9	1L10		Add another 1
10	0R11		Halt

Ex 8B17:

	0	1	
0	1R1	1R0	Go to very end and fill in gap
1	0L2	1R1	
2	0R4	1L3	Go left, testing parity
3	0R5	1L2	
4		0R6	Subtract 1 if $m + n$ is even
5	0R6	0R5	Erase if $m + n$ is odd