8. TURING MACHINES

§ 8.1. The Limits of Computing

As computer hardware and software continue to develop at an ever increasing rate we would 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 non-computability in order to give them some perspective in the work they will perform.

But computability is also relevant for the more logically fundamental parts of mathematics. Consider the real numbers, for example. Most of them are irrational and so cannot be defined by writing out their 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 that, if left to run to eternity, 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 uncountably many potential computer programs, most real numbers are inaccessible to human thought! What is even more interesting is that we can define quite precisely a real number, yet never be able to compute it in the above way.

§ 8.2. The Halting Problem

Anyone who has ever written a program for a computer will know that there is always the possibility for the program to get itself into an infinite loop. For example, in BASIC:

```
10: 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 but how would the compiler search out *all* potential loops. Certainly not by trying out 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.

Now the problem can be solved for certain artificial or trivial languages. For example a program written in a language that had no branching or looping capabilities must inevitably terminate when it reaches the last instruction. That is an example of what is meant 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.

A language in which one can perform useful work includes all languages that are actually used, ranging from BASIC with its notorious propensity for loops, to highly structured languages such as PASCAL and C++. But what precisely is meant by these terms? You cannot prove that something is impossible until you precisely define the problem.

§ 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 accurate 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 have the right definition.

Alan Turing created a model that consists of a conceptual machine (i.e. one which exists in the mind only). He defined a function to be computable if it can be computed by a Turing Machine.

A **Turing Machine** is an imaginary machine which 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 with only finitely many squares being non-blank. We shall 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. They are *not* "touring" machines or "turning" machines, even though these adjectives might seem appropriate.

At any stage the machine can be in any one of a finite set of states. There is 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.

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.

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

[state] .....0 tape • tape 0 ...

where the portion of the tape shown contains all the 1's and where the character being scanned by the head lies to the immediate right of the full-stop.. As an aid we shall sometimes include the step number, although strictly speaking this does not form part of the ID.

Defn: 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 halts).

Example 1:

17> [2] ... 0 • 1 1 1 0 1 1 1 1 1 0 ...

represent the fact that after 17 steps th non-blank portion of the tape is 111011111 with the head on the left-most 1 and with the machine in state 2.

The instructions for a TM are set out in a table, with two columns, one for each character 0 or 1. The rows correspond to the states and these will be numbered 0, 1, 2, ... The machine always begins in state 0.

	0	**1**
0		
1		
2	1 L 4	0 R 2
3		

If the machine is in state 2 and is reading a 1 it obeys the instruction in the corresponding row and column. In this case it is 0R2. This means that it prints a 0 (erasing the "1" that was occupying the square) moves right one square and stays in state 2.

If after 17 steps the ID was as above this is what would happen. Moreover it continues moving right, erasing the 1's, until it reaches a 0. This time the machine takes its instruction from the first column and obeys the instruction 1L4.

In this example there is no state 4, so the machine halts. This is the device that is used to stop the processing. 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 is sent to a non-existent state (normally represented by the next number after the last state) does it halt. If it never reaches such an instruction the machine runs forever, never producing any final output.

AN n-STATE TM ALWAYS STARTS IN STATE 0 AND HALTS WHEN SENT TO STATE n

§ 8.4. Example of a Turing Machine

Example 2:

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 • 1 1 0 ...

1> [0] ... 0 1 • 1 0 ...

2> [0] ... 0 1 1 • 0 ...

3> [1] ... 0 1 • 1 0 ...

4> [2] ... 0 1 0 • 0 0 ...

5> [3] ... 0 1 0 1 • 0 0 ...

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

§ 8.5. Unary Representation of Numbers

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

But there's an even simpler representation than binary and that's the *unary* system. Why not represent the number n by a string of n 1's? Certainly that will make adding 1 very easy – just write an extra "1" at one end!

This might seem a clumsy arrangement, and certainly if we had to use TMs for any practical purpose we would get sick of enormous strings of 1's (imagine writing the date!). 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 will 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.)

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

If the data consists of more than one number, each number is separated by a single 0.

With these conventions there is ambiguity as to the number of items in the data list. However the TM simply takes as many arguments as it needs.

Example 3:

... 0 • 1 1 1 0 ...

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

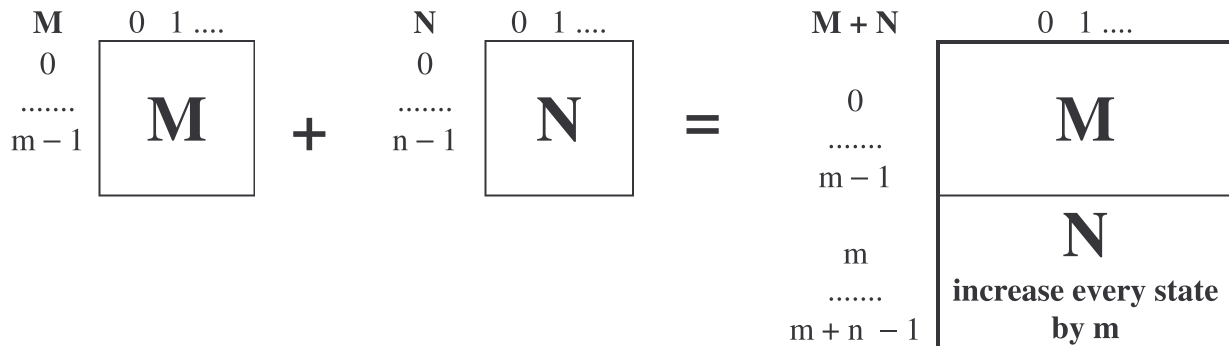
... 0 • 1 1 0 1 0 0 1 1 1 0 ...

represents the list (2, 1, 0, 3, 0, 0, ...)

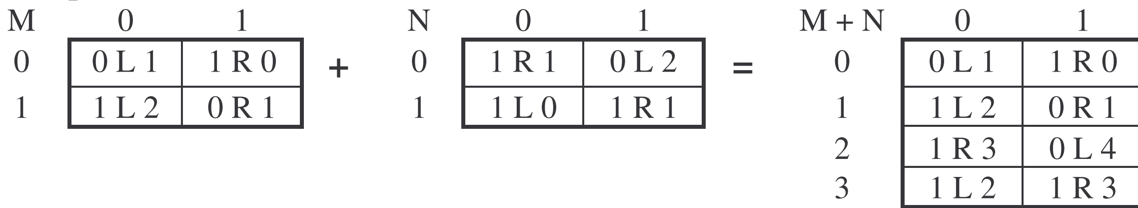
§ 8.6. Adding Turing Machines

More complex TMs can be built up from simpler ones. Suppose we have two TMs M and N on the same alphabet. 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 does not halt then N never gets a look in!)



Example 4:



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 that a computer is called upon to do, can be done by a TM.

Example 5:
This TM computes the function ZERO(n) = 0.
ZERO> n = 0

	0	**1**
0	0R1	0R0

Example 6:
This TM computes the function INC(n) = n + 1.

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

N.B. An empty cell indicates a combination which will never be reached.

Example 7:
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 8: 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 9: This TM computes the function ADD(m, n) = m + n.

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

Example 10: This TM computes the function DOUBLE(n) = 2n.

DOUBLE = REPEAT + ADD

Writing DOUBLE out in full we get:

	0	**1**	
0	0L5	0R1	
1	0R2	1R1	
2	1L3	1R2	**REPEAT**
3	0L4	1L3	
4	1R0	1L4	
5	0R6	1L5	
6	0R9	0R7	**ADD**
8	1L8	1R7	
9	0R9	1L8	

Example 11: 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 12: 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	
1	0L2	1R1	
2	0L4	0L3	HALVE
3	1R0	1L3	
4	0R5	1L4	
5	0L9	0R6	
6	0L7	1R6	
7	0L9	0L8	HALVE
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 Halt if all gone
1	0R8	0R2	
2	0R8	0R3	
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 13: 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	0R9	0R7	DOUBLE
7	1L8	1R7	
8	0R9	1L8	
9	0L14	0R10	
10	0R11	1R10	
11	0L14	1R11	
12	0L13	1L12	
13	0R9	1L13	
14	0R15	1L14	
15	0L3	1L15	Go to 1st block

Example 14:

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 need to be able to know, when we are looking in one direction far we got last time. We do this by making clever use of markers. 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 have examined and which contains no "1". As we move the left marker to the left or the right marker to the right we check whether there is 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 be left with the other marker to erase.

	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 marker
6	0L6	1L7	
7	0R10		halt on the “1”
8	0L8	0R9	tidy up left marker
9	0R9	1L7	

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

EXERCISES 8B (Constructing TMs)

Ex 8B1: Construct a Turing Machine which implements 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 at the start and finish must be 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 have 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 ≥ 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 following function: f(x) = the integer part of (x/2).

Ex 8B10: Construct a Turing Machine which computes the function f(x) = (x + 1, 2).

Ex 8B11: (a) Construct a binary 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 8B12: Construct a Turing Machine which will compute the function

$$f(x, y) = (x + y, 1)$$

for unary input and output.

Ex 8B13: Design a Turing Machine to compute the following functions:
(i) f(n) = INT(n/3);
(ii) g(m, n) = m + n + 3;
(iii) h(n) = n − 1 (if n > 0) and 3 if n = 0.

Ex 8B14: 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 16 state machine that computes 2^n.

Ex 8B15: Construct a binary TM to calculate the function f(a, b) = a + b + 2 with n in unary notation.

Ex 8B16: Construct a binary TM to calculate the function f(n) = 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 8B17: Design a Turing Machine to compute the function f(x, y) = INT(AVERAGE(x, y)) + 2.

Ex 8B18: 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 11 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 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 way to construct this machine is to 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. (It is important that we check that running the first copy of ADD does not disturb the third summand z and leaves the pair (x + y, z) on the tape.) This gives:

	0	1
0	0R3	0R1
1	1L2	1R1
2	0R3	1L2
3	0R6	0R4
4	1L5	1R4

5	0R6	1L5

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

Ex 8B9:

	0	1
0	0L4	0R1
1	0L2	1R1
2	0L4	0L3
3	1R0	1L3
4	0R5	1L4

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

Ex 8B10:

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

Ex 8B11:
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 8B12:

	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 8B13:
(i)

	0	1	
0	0R7	0R1	Erase a group of 3 Halt if all gone
1	0R7	0R2	
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 8B14:

	0	1
0	0R1	1R1
1	1L2	
2	0L3	
3	0R4	1L3
4	0R23	0R5
5	0R6	1R5
6	2^n	
…		
21		
22	0L3	1L22

Ex 8B15:

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

Ex 8B16:

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

Ex 8B17:

	0	1	
0	0R3	0R1	ADD
1	1L2	1R1	
2	0R3	1L2	
3	0L7	0R4	HALVE Slightly modified to add 1 as well
4	0L5	1R4	
5	0L7	0L6	
6	1R3	1L6	
7	1L8	1L7	
8	1L9		Add another 1
9	0R10		Halt

Ex 8B18:

	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