3
Turing machines

No human being can write fast enough, or long enough, or small enough† to list all members of an enumerably infinite set by writing out their names, one after another, in some notation. But humans can do something equally useful, in the case of certain enumerably infinite sets: They can give explicit instructions for determining the nth member of the set, for arbitrary finite n. Such instructions are to be given quite explicitly, in a form in which they could be followed by a computing machine, or by a human who is capable of carrying out only very elementary operations on symbols. The problem will remain, that for all but a finite number of values of n it will be physically impossible for any human or any machine to actually carry out the computation, due to limitations on the time available for computation, and on the speed with which single steps of the computation can be carried out, and on the amount of matter in the universe which is available for forming symbols. But it will make for clarity and simplicity if we ignore these limitations, thus working with a notion of computability which goes beyond what actual men or machines can be sure of doing. The essential requirement is that our notion of computability not be too narrow, for shortly we shall use this notion to prove that certain functions are not computable, and that certain enumerable sets are not effectively (mechanically) enumerable – even if physical limitations on time, speed, and amount of material could somehow be overcome.

The notion of computation can be made precise in many different ways, corresponding to different possible answers to such questions as the following. 'Is the computation to be carried out on a linear tape, or on a rectangular grid, or what? If a linear tape is used, is the tape to have a beginning but no end, or is it to be endless in both directions? Are the squares into which the tape is divided to have addresses (like addresses of houses on a street) or are we to keep track of where we are by writing special symbols in suitable squares (as one might mark a trail in the woods by marking trees)?' And so on. Depending on how such questions are

† There is only a finite amount of paper in the world, so you'd have to write smaller and smaller without limit, to get an infinite number of symbols down on paper. Eventually, you'd be trying to write on molecules, on atoms, on electrons, ...

answered, computations will have one or another appearance, when examined in detail. But our object is not to give a faithful representation of the individual steps in computation, and in fact, *the very same set of functions turns out to be computable, no matter how these questions are answered.* We shall now answer these and other questions in a certain way, in order to get *a* characterization of the class of computable functions. A moderate amount of experience with this notion of computability will make it plausible that the net effect would have been the same, no matter in what plausible way the questions had been answered. Indeed, given any other plausible, precise characterization of computability that has been offered, one can prove by careful, laborious reasoning that our notion is equivalent to it in the sense that any function which is computable according to one characterization is also computable according to the other. But since there is no end to the possible variations in detailed characterizations of the notions of computability and effectiveness, one must finally accept or reject the *thesis* (which does not admit of mathematical proof) that the set of functions computable in our sense is identical with the set of functions that men or machines would be able to compute by whatever effective method, if limitations on time, speed, and material were overcome. Although this thesis ('Church's thesis') is unprovable, it *is* refutable, *if false*. For if it is false, there will be some function which is computable in the intuitive sense, but not in our sense. To show that such a function is computable in the intuitive sense, it would be sufficient to give instructions for computing its values for all (arbitrary) arguments, and to see that these instructions are indeed effective. (Presumably, we are capable of seeing *in particular cases*, that sets of instructions are effective.) To show that such a function is not computable in our sense, one would have to survey all possible Turing machines and verify (by giving a suitable proof) that none of them compute that function. (You will see examples of such proofs shortly, in Chapters 4 and 5; but the functions shown there not to be Turing computable have not been shown to be effectively computable in *any* intuitive sense, so *they* won't do as counterexamples to Church's thesis.) Then if Church's thesis is false, it is refutable by finding a counterexample; and the more experience of computation we have without finding a counterexample, the better confirmed the thesis becomes.

Now for the details. We suppose that the computation takes place on a tape, marked into squares, which is unending in both directions – either because it is actually infinite or because there is a man stationed at each

end to add extra blank squares as needed. With at most a finite number of exceptions, all squares are blank, both initially and at each subsequent stage of the calculation. If a square is not blank, it has exactly one of a finite number of symbols $S_1, \ldots, S_n$ written in it. It is convenient to think of a blank square as having a certain symbol – the *blank* – written in it. We designate the blank by 'S_0'.

At each stage of the computation, the human or mechanical computer is *scanning* some one square of the tape. The computer is capable of telling what symbol is written in the scanned square. It is capable of erasing the symbol in the scanned square and writing a symbol there. And it is capable of movement: one square to the right, or one square to the left. If you like, think of the machine quite crudely, as a box on wheels which, at any stage of the computation, is over some square of the tape. The tape is like a railroad track; the ties mark the boundaries of the squares; and the machine is like a very short car, capable of moving along the track in either direction, as in Figure 3-1. At the bottom of the car there is a device that can read what's written between the ties; erase such stuff; and write symbols there.

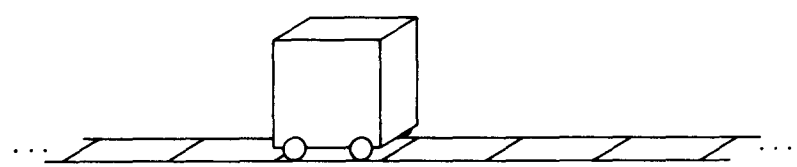


Figure 3-1

The machine is designed in such a way that at each stage of the computation it is in one of a finite number of internal *states*, $q_1, \ldots, q_m$. Being in one state or another might be a matter of having one or another cog of a certain gear uppermost, or of having the voltage at a certain terminal inside the machine at one or other of m different levels, or what have you: We are not concerned with the mechanics or the electronics of the matter. Perhaps the simplest way to picture the thing is quite crudely: Inside the box there is a man, who does all the reading and writing and erasing and moving. (The box has no bottom: the poor mug just walks along between the ties, pulling the box with him.) The man has a list of m instructions written down on a piece of paper. *He is in state q_i when he is carrying out instruction number i.*

Each of the instructions has conditional form: It tells the man what to

do, depending on whether the symbol being scanned (the symbol in the scanned square) is S_0 or S_1 or ... or S_n. Namely, there are $n+4$ things he can do:

(1) Halt the computation.
(2) Move one square to the right.
(3) Move one square to the left.
(4) Write S_0 in place of whatever is in the scanned square.
(5) Write S_1 in place of whatever is in the scanned square.
⋮
($n+4$) Write S_n in place of whatever is in the scanned square.

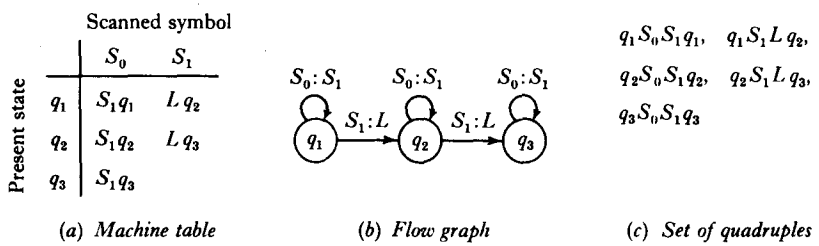
So, depending on what instruction he is carrying out (= what state he is in) and on what symbol he is scanning, the man will perform one or another of these $n+4$ overt acts. Unless he has halted (overt act number 1), he will also perform a covert act, in the privacy of his box, *viz.*, he will determine what the next instruction (*next* state) is to be. Thus, the *present state* and the presently *scanned symbol* determine what overt *act* is to be performed, and what the *next state* is to be.

The overall *program* of instructions that the man is to follow can be specified in various ways, e.g. by a *machine table* or by a *flow graph* or by a *set of quadruples*. The three sorts of descriptions are illustrated in Figure 3-2 for the case of a machine which writes three symbols S_1 on a blank tape and then halts, scanning the leftmost of the three.

Example 3.1. Write $S_1S_1S_1$

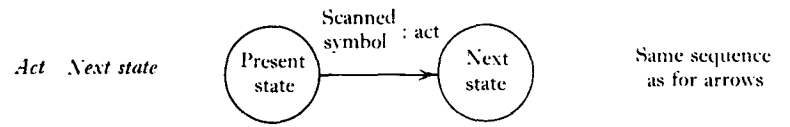
The machine will write an S_1 in the square it's initially scanning, move left one square, write an S_1 there, move left one more square, write a third S_1 there, and halt. (It halts when it has no further instructions.) There will be three states – one for each of the symbols S_1 that are to be written. In Figure 3-2, the entries in the top row of the machine table (under the horizontal line) tell the man that when he's following instruction q_1 he is to (1) write S_1 and repeat instruction q_1, if the scanned symbol is S_0, but (2) move left and follow instruction q_2 next, if the scanned symbol is S_1. The same information is given in the flow graph by the two arrows that emerge from the node marked 'q_1'; and the same information is also given by the first two quadruples. The significance of a table entry, of an arrow in a flow graph, and of a quadruple, is shown in Figure 3-3.

Unless otherwise stated, it is to be understood that a machine starts in its



(a) *Machine table* (b) *Flow graph* (c) *Set of quadruples*

Figure 3-2



(a) *Table entry* (b) *Arrow in a flow graph* (c) *Quadruple*

Figure 3-3

lowest-numbered state. The machine we have been considering *halts* when it is in state q_3 scanning S_1, for there is no table entry or arrow or quadruple telling it what to do in such a case.

A virtue of the flow graph as a way of representing the machine program is that if the starting state is indicated somehow (e.g. if it is understood that the leftmost node represents the starting state unless there is an indication to the contrary) then we can dispense with the names of the states: It doesn't matter what you call them. Then the flow graph could be drawn as in Figure 3-4. We can indicate how such a machine operates by writing down its sequence of *configurations*. Each configuration shows what's on the tape at some stage of the computation, shows what state the machine is in at that stage, and shows which square is being scanned. We'll do this by writing out what's on the tape and writing the name of the present state under the symbol in the scanned square:

```
0000000  000010 0  000010 0  00011 00  00011 00  00111 00
      1        1        2        2        3        3
```

Here we have written the symbols S_0 and S_1 simply as 0 and 1, and similarly we have written the states q_1, q_2, q_3 simply as 1, 2, 3 to save needless

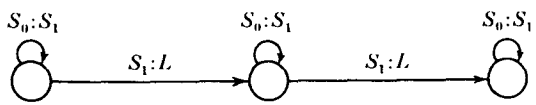


Figure 3-4. Write three S_1s.

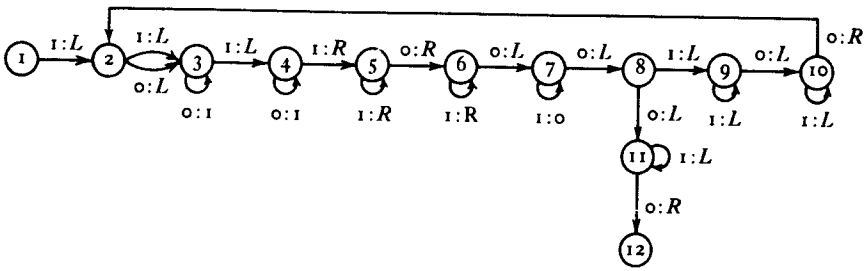
fuss. Thus, the last configuration is short for

$$\underset{q_3}{S_0S_0S_1S_1S_1S_0S_0}$$

Of course, the strings of S_0s at the beginning and end of the tape can be shortened or lengthened *ad lib.* without changing the significance: the tape is understood to have as many blanks as you please at each end. Now here is a more complex example.

Example 3.2. Double the number of 1s

This machine starts off scanning the leftmost of a string of 1s on an otherwise blank tape, and winds up scanning the leftmost of a string of twice that many 1s on an otherwise blank tape. Here is the flow graph:



How does it work? In general, by writing double 1s at the left, and erasing single 1s from the right. In particular, suppose that the initial configuration is $\underset{1}{111}$, so that we start in state 1, scanning the leftmost of a string of three 1s on an otherwise blank tape. The next few configurations are

$$\underset{2}{0111},\quad \underset{3}{00111},\quad \underset{3}{10111},\quad \underset{4}{01011},\quad \underset{4}{11011}.$$

So we have written our first double 1 at the left – separated from the original string, 111, by a 0. Next we go right, past the 0 to the right-hand end of the original string, and erase the rightmost 1. Here is how that works, in two stages. Stage 1:

$$\underset{5}{110111},\quad \underset{5}{110111},\quad \underset{6}{110111},\quad \underset{6}{110111},\quad \underset{6}{110111},\quad \underset{6}{110110}.$$

Now we know that we have passed the last of the original string of 1s, so (stage 2) we back up and erase one:

$$\underset{7}{110111},\quad \underset{7}{110110}.$$

Now we hop back to our original position, scanning the leftmost 1 in what remains of the original string. When we have gone round the loop once more we shall be in this configuration: $\underset{7}{111101}0$. We'll then go through these:

$$\underset{8}{111101},\quad \underset{9}{111101},\quad \underset{10}{111101},\quad \underset{10}{111101},\quad \underset{10}{111101},\quad \underset{10}{111101},\quad \underset{10}{011111}0,$$

$$\underset{2}{111101},\quad \underset{3}{011111}0,\quad \underset{3}{111111}0,\quad \underset{4}{011111}00,\quad \underset{4}{111111}0.$$

We now have our six 1s on the tape; but we want to erase the last of the original three and then halt, scanning the leftmost of the six that we wrote down. Here's how:

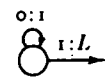
$$\underset{5}{1111110}1,\quad \underset{5}{1111110}1,\quad \ldots,\quad \underset{5}{1111110}1,\quad \underset{5}{1111110}1,\quad \underset{6}{1111110}1,$$

$$\underset{6}{11111010},\quad \underset{7}{11111010},\quad \underset{7}{11111100},\quad \underset{8}{111111}0,\quad \underset{11}{111111},$$

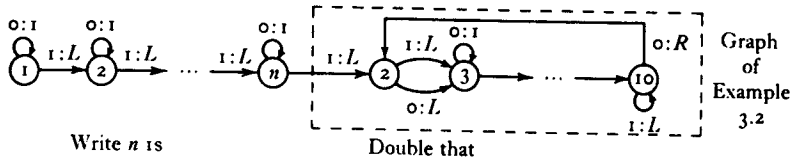
$$\underset{11}{111111},\quad \ldots,\quad \underset{11}{111111},\quad \underset{11}{011111}1,\quad \underset{12}{111111}.$$

Now we are in state 12, scanning a 1. Since there is no arrow from that node telling us what to do in such a case, we halt. The machine performs as advertised. (*Note*: the fact that the machine doubles the number of 1s when the original number is three is not a proof that the machine performs as advertised. But our examination of the special case in which there are three 1s initially made no essential use of the fact that the initial number was three: it is readily converted into a proof that the machine doubles the number of 1s no matter how long the original string may be.)

Example 3.3. Write $2n$ 1s on a blank tape and halt, scanning the leftmost 1

There is a straightforward way, using $2n$ states. We made use of the method in Example 3.1. In general, the flow graph would be obtained by stringing together $2n$ replicas of this: 

and lopping off the last arrow. But there is another way, using $n+11$ states: write n 1s on the tape in the straightforward way, and then use the method of Example 3.2 to double *that*. Schematically, we put the two graphs together by identifying node 1 in Example 3.2 with node n in the graph of the machine that writes n 1s on a blank tape. If n is large, the straightforward way of writing $2n$ 1s will require many more states than the indirect way which we have just



described. (If n is greater than 11, $2n$ is greater than $n+11$.) Of course, we could save even more states by repeating the doubling trick, e.g. to write 100 1s we could use 25 states to write 25 of them, use another 11 to double that, and use yet another 11 to get the full 100 by doubling *that*. This does it with 47 states instead of the 100 we would need by the direct method, or the 61 we would need if we used the doubling trick only once.

Exercises. (Solutions follow)

3.1 Design a Turing machine which, started scanning the leftmost of an unbroken string of 1s on an otherwise blank tape, eventually halts, scanning a square on an otherwise blank tape – where the scanned square contains a blank or a 1 depending on whether there were an even or an odd number of 1s in the original string.

3.2 Design a Turing machine which, started scanning the leftmost of an unbroken string of 1s and 2s on an otherwise blank tape (in any order, and perhaps with all 1s or all 2s) eventually halts – at which point the contents of the tape indicate (following some convention which you are free to stipulate) whether or not there were exactly as many 1s as 2s in the original string.

3.3 *Add 1 in decimal notation.* Initially the machine scans the rightmost of an unbroken string of decimal digits on an otherwise blank tape. When it halts, the tape holds the decimal representation of 1 + the original number. Design a simple machine which does this. (Give the machine table – a graph would be messy.)

3.4 *Multiply in monadic notation.* Initially, the tape is blank except for two solid blocks of 1s, separated by a single blank. Say there are p 1s in the first block and q in the second. Design a machine which, started scanning the leftmost 1 on the tape, eventually halts, scanning the leftmost of an unbroken block of pq 1s on an otherwise blank tape.

3.5 *Add in in monadic notation.* (Identical with Exercise 3.4 but with '$p+q$' in place of 'pq'.)

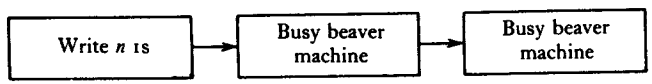
3.6 *Not all functions from positive integers to positive integers are Turing computable.* You know via an Exercise in Chapter 2 that there are more such functions than there are Turing machines, so there must be some that no Turing machines compute. Explain a bit more fully.

3.7 *Productivity.* Consider Turing machines which use only the symbol 1 in addition to the blank. Initially the tape is blank. If the machine eventually halts, scanning the leftmost of an unbroken string of 1s on an otherwise blank tape, its *productivity* is said to be the length of that string. But if the machine never halts, or halts in some other configuration, its productivity is said to be 0. Now define $p(n)$ = *the productivity of the most productive n-state Turing machines.* Prove that

(*a*) $p(1) = 1$;
(*b*) $p(47) \geqslant 100$;
(*c*) $p(n+1) > p(n)$;
(*d*) $p(n+11) \geqslant 2n$.

(For solutions, see Chapter 4.)

3.8 *The busy beaver problem* (after Tibor Rado, *Bell System Technical Journal*, May 1962). According to Exercise 3.6, uncomputable functions exist. But what would such a thing look like? Actually, you have just met one: the function p defined in Exercise 3.7. The proof that it is uncomputable turns on the fact that the 'busy beaver problem' is unsolvable. This is the problem of designing a Turing machine which uses only the symbol 1 in addition to the blank and which, started in state 1 scanning the leftmost of an unbroken string of n 1s on an otherwise blank tape, eventually halts scanning the leftmost of an unbroken string of $p(n)$ 1s on an otherwise blank tape. In fact, no such machine can exist. To prove that, assume that there *is* such a machine (with k states, say) and deduce a contradiction: That takes some ingenuity. *Hint*: if there were such a machine, then by stringing two replicas of it onto an n-state machine which writes n 1s on a blank tape (right to left) you can get an $n+2k$ state machine which has productivity $p(p(n))$; and from this, together with the last two things you proved in Exercise 3.7, you can deduce a contradiction.



(For solution, see Chapter 4.)

3.9 *Uncomputability via diagonalization.* Every Turing machine which uses only the symbol 1 in addition to the blank computes some function

from positive integers to positive integers according to the following convention. To compute the value which the function assigns to the argument n, start the machine in its lowest-numbered state, scanning the leftmost of a string of n 1s on an otherwise blank tape. The function is defined for the argument n if and only if the machine eventually halts, scanning the leftmost of an unbroken string of 1s on an otherwise blank tape, in which case the value of the function is the length of the string. Since the Turing machines are enumerable (cf. Exercise 3.6), so are the functions they compute. Define an 'antidiagonal' function and prove that no Turing machine computes it. (For solution, see Chapter 5.)

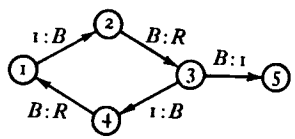
3.10 *The halting problem.* Let $M_1, M_2, \ldots$ be an enumeration of all Turing machines which use only the symbol 1 in addition to the blank. The *self-halting problem* is that of specifying a uniform effective procedure in some format or other for computing a total or partial function s for which we have $s(n) = 1$ if and only if M_n never halts, after being started in its initial state, scanning the leftmost of an unbroken string of n 1s on an otherwise blank tape. The full *halting problem* is that of specifying a uniform effective procedure for computing some function h for which we have $h(m, n) = 1$ if and only if M_m never halts, after being started in its initial state, scanning the leftmost of an unbroken string of n 1s on an otherwise blank tape.

(*a*) Prove that the self-halting problem is unsolvable.

(*b*) Prove the corollary: the halting problem is unsolvable.

Hint: Church's thesis must be used, sooner or later. Sooner *and* later: prove that one of the functions shown in Exercise 3.6 to be computable by no Turing machine is computable in an unusual format (using two Turing machines at once) if the self-halting problem is solvable.

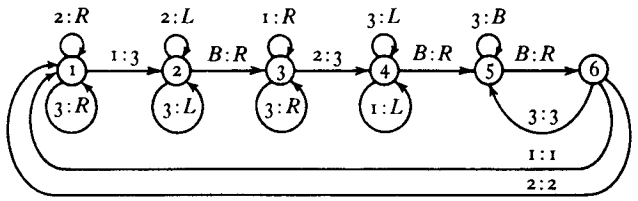
Solutions



(We have written 'B' for S_0, the blank.)

3.1 If there were 0 or 2 or 4 or ... 1s to begin with, this machine halts in state 1, scanning a blank on a blank tape; if there were 1 or 3 or 5 or ..., it halts in state 5, scanning a 1 on an otherwise blank tape.

3.2 When this machine halts, the tape is blank if and only if there were equal numbers of 1s and 2s in the original string.



Change leftmost 1 to 3 and return to the left

Change leftmost 2 to 3 and return to the left

Erase all 3s from the extreme left

There are other ways of doing this, e.g. ways in which the machine uses no symbols other than 1 and 2. The simplest way of doing it which is compatible with the letter, but not the spirit of the problem statement, is to adopt the following convention regarding the way in which the contents of the tape when the machine halts indicate whether or not there were equal numbers of 1s and 2s initially: there were equal numbers of 1s and 2s initially if and only if there are equal numbers of 1s and 2s on the tape when the machine halts. The machine can then halt immediately, so that the final tape contents are identical with the initial contents. The graph of such a machine would look like this: (1)

3.3 There are 11 possible scanned symbols, among which B is the blank and 0 is not the blank but the cipher zero; $1, \ldots, 9$ are the usual ciphers. In the body of the table, each pair represents the *act to be per-*

Machine table for adding 1 *in decimal notation*

Scanned symbol	Present state 1	Present state 2
B	1 2	
0	1 2	L 1
1	2 2	
2	3 2	
3	4 2	
4	5 2	
5	6 2	
6	7 2	
7	8 2	
8	9 2	
9	0 2	

formed and the *next state*, in that order. The many blanks in column 2 represent situations in which the machine has no instructions, and therefore halts.

3.4 There are lots of equally efficient ways of multiplying in monadic notation. Many of them involve use of extra symbols beyond the B and 1 that appear on the tape initially and finally, and that is perfectly all right: they are symbols which the machine uses to keep track of where it is in the process, and which it erases before halting. But the method shown uses only B and 1 throughout the process. We have done that in order to illustrate this point: *any function from positive integers to positive integers which is computable at all is computable in monadic notation by a machine which uses only the symbols B and 1*.

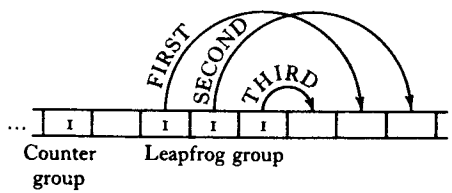
Here is how this machine works. The first block, of p 1s, is used as a *counter*, to keep track of how many times the machine has added q 1s to the group at the right. To start, the machine erases the leftmost of the p 1s and sees if there are any 1s left in the counter group. If not, $pq = q$, and all the machine has to do is position itself over the leftmost 1 on the tape, and halt. But if there are any 1s left in the counter the machine goes into its *leapfrog routine*: in effect, it moves the block of q 1s (the 'leapfrog group') q places to the right along the tape, e.g. with $p = 2$ and $q = 3$ the tape looks like this initially

$$11B111$$

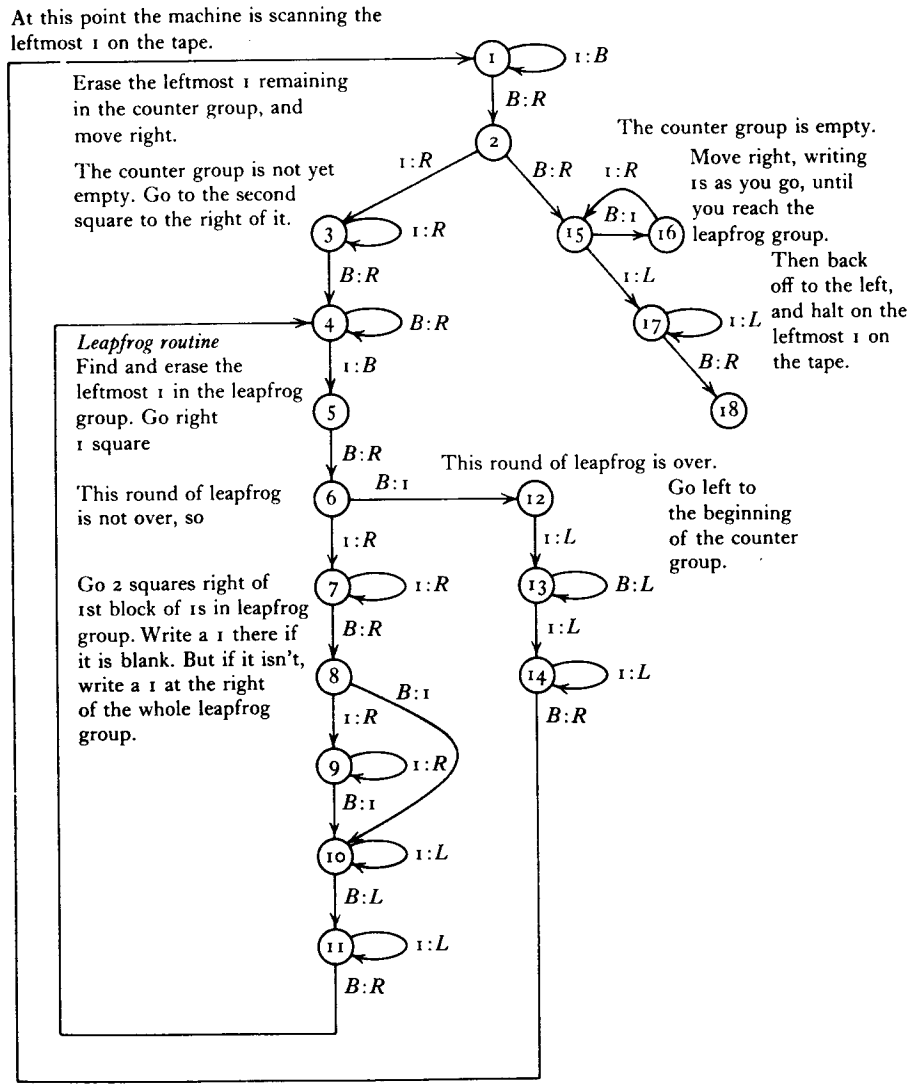
and looks like this after going through the leapfrog routine:

$$B1BBBB111$$

The machine will then note that there is only one 1 left in the counter, and will finish up by erasing that 1, moving right two squares, and changing all Bs to 1s until it comes to a 1, at which point it continues on to the leftmost 1 and halts. This is how the leapfrog routine works:



In general, the leapfrog group consists of a block of 0 or 1 or ... or q 1s, followed by a blank, followed by the remainder of the q 1s. The blank is there to tell the machine when the leapfrog game is over: without it the



Flow graph for multiplying in monadic notation

group of q 1s would keep moving right along the tape forever. (In playing leapfrog, the portion of the q 1s to the left of the blank in the leapfrog group functions as a counter: it controls the process of adding 1s to the portion of the leapfrog group to the right of the blank. That is why there are two big loops in the flow graph: one for each counter-controlled subroutine.)

3.5 The object is to erase the leftmost 1, fill the gap between the two blocks of 1s, and halt scanning the leftmost 1 that remains on the tape. Here is one way of doing it, in quadruple notation: $q_1S_1S_0q_1$; $q_1S_0Rq_2$; $q_2S_1Rq_2$; $q_2S_0S_1q_3$; $q_3S_1Lq_3$; $q_3S_0Rq_4$.

3.6 According to Exercise 2.3 the set of all functions from positive integers to positive integers is not enumerable. On the other hand, the set of all Turing machines *is* enumerable, for any Turing machine is describable by a finite string of letters of the infinite alphabet

$$;,\ R,\ L,\ S_0,\ q_1,\ S_1,\ q_2,\ S_2,\ q_3,\ \ldots$$

and according to Exercise 2.4(d) (with $a_1 = ;$, $a_2 = R$, $a_3 = L$, $a_4 = S_0$, $a_5 = q_1, \ldots$) the set of all such strings is enumerable. (We need a convention for identifying the starting state, e.g. we might require that the starting state be assigned the lowest q-number.) Then whatever convention one may adopt, for deciding what (if anything) the function is which is computed by each Turing machine in our list, our list of Turing machines yields a (possibly gappy) list of *all* the functions from positive integers to positive integers which are computed by Turing machines according to our convention. One possible convention is the one specified in Exercise 3.4, according to which we may take it that Turing machines whose quadruples contain any of the symbols S_2, S_3, ... compute *no* functions from positive integers to positive integers. Note that if $f_1, f_2, f_3, \ldots$ is a gapless list of all functions computed by Turing machines according to our convention, the antidiagonal function u of Exercise 2.3 is an example of a function from positive integers to positive integers which is computed in our format by no Turing machine. Another example is the partial function t which we define:

$$t(n) = \begin{cases} 1 & \text{if } f_n(n) \text{ is undefined,} \\ \text{undefined} & \text{if } f_n(n) \text{ is defined.} \end{cases}$$

(For further details, see Chapter 5.)

3.10(a) If the self-halting problem were solvable, then by Church's thesis there would be some Turing machine S which computes s in some format, e.g. in the format of Exercise 3.9. The function t of Exercise 3.6 would then be computable in the following format. To determine $t(n)$, start machines S and M_n in their initial states, scanning the leftmost of unbroken strings of n 1s on their otherwise blank tapes.

Case 1: M_n *never halts*. Then S will eventually halt, scanning a 1 on its otherwise blank tape. At that time we shall know that M_n will never halt, and thus we shall know that the corresponding function f_n is undefined for the argument n, so that $t(n) = 1$.

Case 2: M_n *eventually halts*. When it does, we can determine by examining the tape whether or not $f_n(n)$ is defined, and will thus know whether $t(n)$ is undefined or equal to 1. Then if the self-halting problem is solvable, t is computable in an unusual format, and hence, by Church's thesis, t is computed by some Turing machine, in contradiction to what we have proved in Exercise 3.6.

3.10(b) If the halting problem were solvable, i.e., if h were computable, then the self-halting problem would be solvable, i.e., s would be computable, for we have $s(n) = h(n, n)$ for every positive integer n.