(*e*) {what is the derivative of the function f? $|f$ is a polynomial function of one real variable}.

(*f*) {what is the nth odd prime? $|n \in D_N$}.

2 Prove that for any subset A of D_N, if both A and its complement are recursively enumerable, then A is recursive. (Hint: use Church's Thesis.) Deduce that there is a subset of D_N which is neither recursive nor recursively enumerable.

3 Prove that if an infinite set A is recursively enumerable in increasing order (i.e. if there is a recursive function f such that $f(0), f(1), f(2), \dots$ is a list of the members of A and $f(n) < f(n+1)$ for each $n \geq 0$), then A is recursive.

4 Show that the following functions are recursive.

(*a*) ϕ, where $\phi(n)=$ the number of positive integers p less than n such that p and n have no common factors $(n \in D_N)$.

(*b*) f, where $f(m, n) =$ smallest integer greater than m/n, $(m, n \in D_N)$.

(*c*) g, where

$$g(n)=\begin{cases} 0 \text{ if there is a sequence of at least } n \text{ 7s} \\ \quad \text{in the decimal expansion of } \pi, \\ 1 \text{ otherwise.} \quad (n \in D_N) \end{cases}$$

(*d*) q, where

$$q(m, n)=\begin{cases} \text{the Gödel number of } (\mathscr{A} \to \mathscr{B}) \text{ if } m \text{ and } n \text{ are} \\ \quad \text{the Gödel numbers of } wf\text{s. } \mathscr{A} \text{ and } \mathscr{B} \text{ of } \mathcal{N}, \\ 0 \text{ otherwise.} \quad (m, n \in D_N) \end{cases}$$

5 Give examples of

(*a*) A recursive set with a non-recursive subset.

(*b*) A non-recursive set with an infinite recursive subset.

6 Show that every infinite recursively enumerable set has an infinite recursive subset.

7.2 Turing machines

It is a valuable exercise actually to go into the details of one of the computational characterisations of the notion of algorithm. The Turing characterisation is the most useful and the most easily understood, and it can be applied directly to problems of decidability and solvability, as we shall see.

The reader should not be misled by the use of the word 'machine'. Turing machines are not actual working computing machines. They are abstract systems, precisely defined in a mathematical way so as to mirror computational procedures. The terminology that we use clearly indicates the 'machine' functions, and it is certainly true that actual machines can be built which follow the procedures of a 'notional' Turing machine.

Turing's purpose in describing his machines was to reduce computations to their barest essentials so as to describe in a simple way some basic procedures which are manifestly effective, and to which any effective procedure might be reduced. Let us now examine the technical details.

A Turing machine may be thought of as a black box through which passes a paper tape, which is divided into equal squares which may or may not have symbols printed on them. For a particular computation the machine will start with a finite amount of input information on the tape, i.e. with symbols printed on only finitely many tape squares. The machine processes the tape according to certain rules, and may or may not eventually come to a halt. If it does halt, then the output information is what remains on the tape. If it does not halt, then the computation is indeterminate and there is no output.

Before going further, there are two points raised above which require comment. The requirement that the input information be finite seems certainly to be a reasonable one. Indeed the reader may wonder why it is necessary to state it, for every existing paper tape is certainly finite in length and can consist of only finitely many equal squares. However it would be unreasonable to place a definite bound on the length of tape required for the input information. Also, as the machine is processing the tape, it may require more 'working space' than is provided on the original input tape, so we shall regard the tape as extendable indefinitely. Again, any computation will require only a finite amount of tape, but it would not be reasonable to put an absolute bound on the length of tape available, so we shall say that the tape is to be *potentially infinite*.

Besides placing no bounds on the tape, we shall place no bounds on the time that the machine requires for a particular computation. On a practical computer we would be forced to set a time limit, and, if no answer had been produced in that time, to give up and try to find a different program which would reduce the time required. However, for our abstract machines, it would be an artificial restriction to set an absolute limit on the number of steps or the time required to obtain an answer. All that we shall require is that if there is an answer, then the machine produces it in a finite time after a finite number of steps. Thus in the course of a computation we may not know (and in general we shall not know) whether that computation will terminate or not.

In order to study what such machines can do it is necessary to specify both the nature of the symbolic information which can appear on the tape and the ways in which the machine can process it.

A Turing machine has an *alphabet of tape symbols*, which may differ from machine to machine, but which is just a finite list of symbols. Each tape square may have printed on it at most one of these symbols at a time. Normally included in the list of symbols is the letter B, which will denote a blank square. The simplest Turing machine will have just two tape symbols, B and 1, say.

A Turing machine operates in the following way. At any given time the machine is 'reading' a single square of the tape. It may replace the symbol appearing in this square (if any) by another symbol, or print a symbol in it if it is blank, or leave the square unchanged, in which case it may shift its attention to the next square to the right or left on the tape. So we have:

Types of Operation:

(*a*) Print a symbol. (Printing a symbol includes first erasing the previous symbol.) Erasing a symbol, i.e. printing a B, is an operation of this type.

(*b*) Move one square left.

(*c*) Move one square right.

One *step* in the operation of the machine is a single operation of one of these types.

Next we must specify how the machine chooses, at each stage, which operation to perform. Its course of action is determined by the symbol which appears on the square being read and by the *internal state* of the machine. The machine is allowed to take on any of a finite number of internal states. In terms of actual computing machines the internal state may be thought of as the sum total of all the information stored in the machine at the given time. We are not concerned with the mechanics or electronics of storing information – we merely suppose that our black box has a finite number of different conditions which cause it to act in certain ways.

However, it is clear that provision must be made for the internal state of the Turing machine to change during a computation, so a single step in a computation must require specification of

(1) the present internal state of the machine,
(2) the contents of the square being read,
(3) the action taken by the machine, and
(4) the next internal state which the machine takes, in preparation for the next step of the computation.

Thus the internal state which the machine takes at a given time will be a consequence of the whole course of the previous computation, and in this sense it acts as a 'memory' for the machine.

Again, a comment is called for at this stage about the finiteness condition imposed on the number of internal states of a Turing machine. Actual digital computers do have only a finite number of different

internal configurations, though admittedly this number is usually exceedingly large. However, for the same reasons as previously, it would be unreasonable to place any specific bound, even a very large one, on the number of states permitted in a Turing machine. We therefore require the number merely to be finite.

The most convenient way to specify the procedure which a Turing machine will follow is by means of a finite set of quadruples of the form

(state, tape symbol, action taken, new state taken).

In order to trace a computation through, at each stage it is necessary to note the internal state and the tape symbol being read, to search through the set of quadruples to find one which starts with this pair, and to follow the action given and take the new state given by this quadruple. This brings out a restriction which must be placed on our set of quadruples: it must be consistent, i.e. for each pair (state, symbol) there must be at most one quadruple which starts with this pair, so that the Turing machine will have a well-defined procedure to follow.

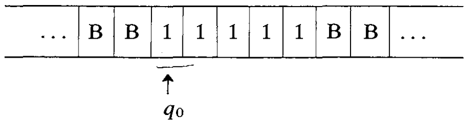
Since the number of states and the number of symbols are both finite, there is a limit on the number of quadruples for a particular machine. However, we shall not require that every (state, symbol) pair appears at the beginning of some quadruple. Some combinations may never occur in computations. More importantly, we shall say that a Turing machine computation *terminates* when and only when the current (state, symbol) pair does not occur in any of the quadruples, so that the machine has no instruction as to how to proceed.

Thus a Turing machine may transform an input tape into an output tape. The way in which information is symbolised on the tape must depend on the nature of the information, and we shall see by means of examples how it can be done. In all of our examples we shall use symbols $q_0, q_1, q_2, \ldots$ to denote internal states, and we shall make the convention that the machines start in state q_0 reading the leftmost non-blank square on the tape. (Some convention of this kind is clearly necessary – there is no particular significance to this one.)

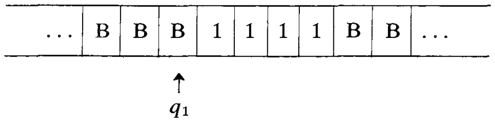
Example 7.12

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

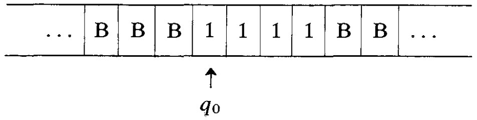
is the set of quadruples for a Turing machine with states q_0 and q_1 and alphabet of symbols B and 1. The quadruples are self explanatory except perhaps for the 'action taken' symbol. The third symbol in the quadruple can be L (for 'move left'), R (for 'move right') or just the symbol which is to replace the symbol being read. Observe the operation of this machine when the input tape contains a finite sequence of 1s, e.g.,



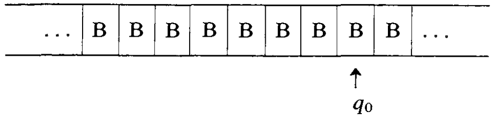
The machine starts in state q_0 reading the leftmost 1. It prints a B (erases the 1) and goes to state q_1.



The machine in state q_1 reading a B moves one square to the right and goes to state q_0.



Now, just as above, the 1 on the square being read is replaced by a B and the machine goes to state q_1. Thus the machine will proceed, moving to the right replacing each 1 by a B, until it reaches the situation



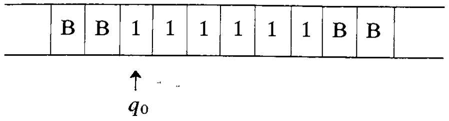
Now there is no quadruple to guide the machine in this situation, so the computation stops. This machine deletes a sequence of 1s and then stops.

Example 7.13

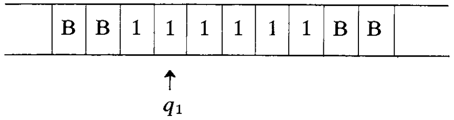
$$\{(q_0\ 1\ R\ q_1),\quad (q_1\ 1\ R\ q_0),\quad (q_1\ \mathrm{B}\ R\ q_2),\quad (q_2\ \mathrm{B}\ 1\ q_2)\}$$

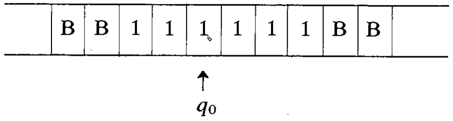
is the set of quadruples for a Turing machine which decides whether a number is odd or even, in the following way. A number may be input in

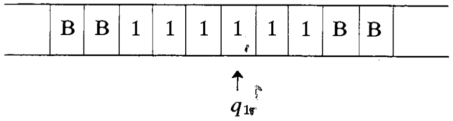
the form of a sequence of 1s on the tape. The machine starts in state q_0 reading the leftmost 1. E.g.,

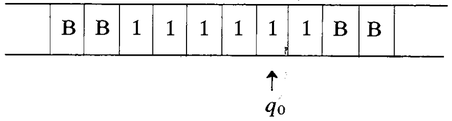


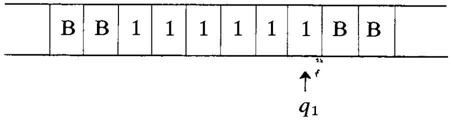
It proceeds as follows:

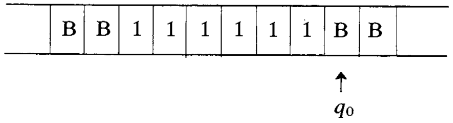




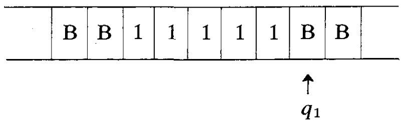




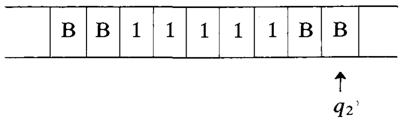


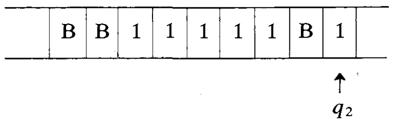


and halts there, since there is no quadruple starting with q_0B, and it will act similarly for any even input number. If the input number had been odd, we would have reached the situation (input 5)



and the computation would proceed thus:





and halts here. This machine will therefore print a 1 after a space of one square if and only if the input number is odd.

Example 7.14

$$\{(q_0\ 1\ X\ q_0),\ (q_0\ X\ R\ q_0),\ (q_0\ 0\ Y\ q_0),\ (q_0\ Y\ R\ q_0)\}$$

is the set of quadruples for a Turing machine which, when a sequence of 0s and 1s is input, will translate it into a sequence of Xs and Ys, e.g. 1 0 1 0 0 1 is translated into X Y X Y Y X. Question: does this machine halt when this translation is completed?

Example 7.15

Addition is almost trivial for a Turing machine. If m and n are input on the tape as sequences of 1s separated by a letter A, the machine with quadruples

$$\{(q_0\ 1\ B\ q_0),\ (q_0\ B\ R\ q_1),\ (q_1\ 1\ R\ q_1),\ (q_1\ A\ 1\ q_2)\}$$

will delete the leftmost 1 and replace the A by a 1 and then stop. When it stops the number $m+n$ is left on the tape as a sequence of $(m+n)$1s.

Example 7.16

A Turing machine can reproduce the contents of the input tape. For

example, if the input tape is blank except for a finite sequence of 0s and 1s, such a machine is specified by the set of quadruples:

q_0 0 X q_1	q_0 1 Y q_2
q_1 X R q_1	q_2 Y R q_2
q_1 0 R q_1	q_2 0 R q_2
q_1 1 R q_1	q_2 1 R q_2
q_1 B R q_3	q_2 B R q_4
q_3 0 R q_3	q_4 0 R q_4
q_3 1 R q_3	q_4 1 R q_4
q_3 B 0 q_5	q_4 B 1 q_5

q_5 0 L q_5

q_5 1 L q_5

q_5 B L q_5

q_5 X R q_0	q_5 Y R q_0

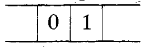
q_0 B L q_6

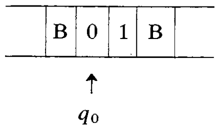
q_6 X 0 q_6

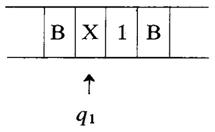
q_6 Y 1 q_6

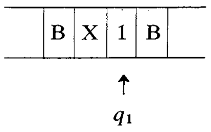
q_6 0 L q_6

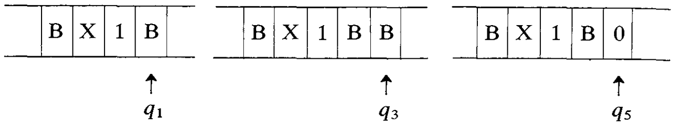
q_6 1 L q_6

This example is clearly more complicated than the previous ones, and the quadruples have been laid out in this way in order to make clearer the machine procedure. Let us follow through a simple example. Suppose that the input tape is blank except for  and that the machine starts in state q_0, reading the leftmost non-blank square. Then the machine will use first the left hand column of quadruples, and the tape will proceed thus:

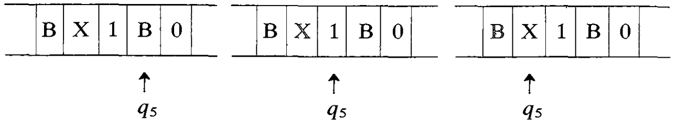




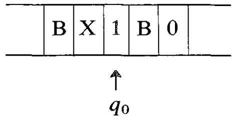




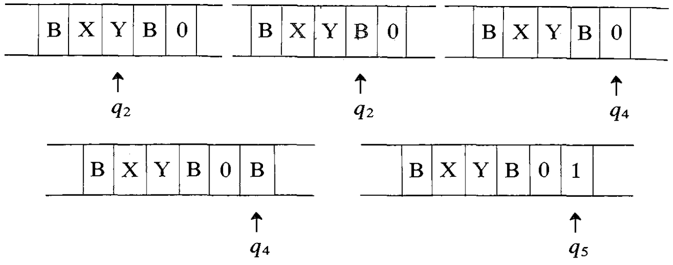
It will then use the centre column of quadruples, giving



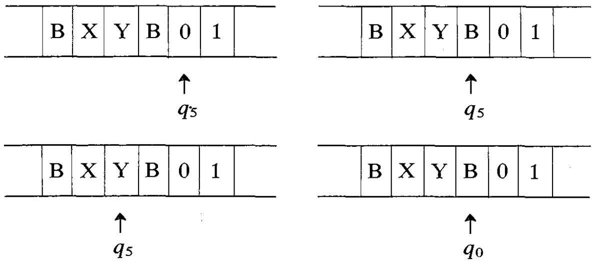
The single quadruple q_5 X R q_0 gives



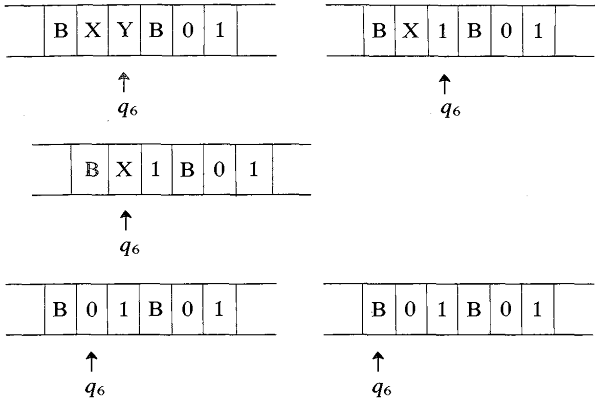
and we go to the top of the right hand column of quadruples



State q_5, as before, leads to



Now, state q_0 reading B uses the bottom centre column of quadruples, to give



and here the machine halts, since there is no quadruple which starts with q_6 B.

▷ This last example illustrates some of the elementary processes of searching, copying, memorising and repeating which a Turing machine can use in carrying out more complex procedures, and it also illustrates how instructions (i.e. quadruples) can be found for a particular Turing machine by analysing the procedure which it is required to follow into a sequence of these elementary processes. Note that the size of the alphabet of tape symbols has a direct effect on the number of quadruples required. In particular, a machine which would duplicate a string of 1s requires the same sort of procedures as in the above example, but would need many fewer quadruples.

Example 7.17

In this example we shall not go into the same amount of detail as in the previous one, but the reader is recommended to follow through at least one actual machine computation in order to observe the combination of elementary processes involved and in particular the way in which the copying procedure of the previous example is used. The following quadruples specify a Turing machine which multiplies two numbers m and n when input in the form of sequences of m and n 1s, separated by a blank square.

q_0 1 X q_1	q_5 B L q_5
q_1 X R q_1	q_5 Y 1 q_6
q_1 1 R q_1	q_6 1 R q_2 .
q_1 B R q_2 ✓	q_2 B L q_7.
q_2 1 Y q_3	q_7 1 L q_7
q_3 Y R q_3	q_7 B L q_7 ✓
q_3 1 R q_3	q_7 X B q_8
q_3 B R q_4	q_8 B R q_0
q_4 B 1 q_5	q_0 B R q_9 ✓
q_4 1 R q_4	q_9 1 B q_{10}
q_5 q L q_5	q_{10} B R q_9

This machine, with input tape, for example

...	B	1	1	B	1	1	1	B	...

when started in state q_0 reading the leftmost 1, will eventually halt in state q_9 with the tape thus:

B	B	1	1	1	1	1	1	B	
	↑ q_9								

This machine, though designed for a more complicated procedure than the previous example, nonetheless has fewer quadruples. This is because the alphabet of tape symbols is here more restricted.

Remark 7.18

At any stage of a Turing machine computation, only a finite part of the tape is non-blank. We can therefore represent the condition of the machine by an 'instantaneous description', for example such as

$$1\ 1\ \mathrm{B}\ 1\ \mathrm{B}\ q_2\ \mathrm{B}\ 1\ \mathrm{B}\ \mathrm{B}\ \mathrm{B}\ 1.$$

This includes the entire non-blank section of the tape and the square being read by the machine. The state symbol is included in the sequence, and is placed immediately to the left of the symbol being read. It is

important to note that an instantaneous description need not contain *only* the non-blank part of the tape. For example,

$$q_5\ \mathrm{B}\ \mathrm{B}\ \mathrm{B}\ \mathrm{B}\ 1\ 1\ \mathrm{B}\ 1 \quad \text{and} \quad 1\ \mathrm{B}\ 1\ 1\ \mathrm{B}\ \mathrm{B}\ \mathrm{B}\ q_3\ \mathrm{B}$$

are proper instantaneous descriptions where the machine is reading a square on a blank part of the tape. However, we never include any more blanks than are necessary to include the square being read.

Remark 7.19

A Turing machine is specified by its list of quadruples. By a coding system similar to the Gödel numbering of Chapter 6, it is possible to assign code numbers first to the quadruples themselves and second to finite lists of quadruples. This is because (we may suppose) all possible state symbols and tape symbols constitute a countable set, and each Turing machine uses only finitely many. To any Turing machine we can therefore assign a code number, and if the method of description (regarding initial state and how inputs and outputs are coded) is standardised, the code numbers will be such that all the relevant information about the corresponding Turing machines will be effectively recoverable. Just as previously, we choose the code numbers so that different code numbers correspond to different machines. Also it will be possible to decide effectively, given any natural number, whether it is the code number of a Turing machine. We shall not prove this, because it depends on the detail of the system of code numbers chosen, but it yields an important result.

Proposition 7.20

The collection of (code numbers of) Turing machines is effectively enumerable.

Proof. List all natural numbers, and as they appear delete all numbers which are not code numbers of Turing machines.

▷ This effective enumeration enables us to associate the list of Turing machines effectively with the set of all natural numbers. Each Turing machine will correspond to the number given by its position in the list. We can therefore take the set of code numbers of Turing machines to be the whole set of natural numbers. We have the following proposition.

Proposition 7.21 (The Enumeration Theorem)

The set of all Turing machines may be listed $T_0, T_1, T_2, \ldots$ in such a way that each suffix determines effectively and completely the instructions for the corresponding machine.

▷ Our examples of Turing machines have shown that they can perform various kinds of computation. We are interested in one particular kind of computation, namely computation of the values of functions. Now *any* Turing machine can be said to compute the values of a function provided that we specify the way in which the contents of the input and output tapes are interpreted as representing elements of the domain and range of the function. Our functions will be number theoretic functions, i.e. whose domains and ranges consist of natural numbers. An element n of D_N may be input to any Turing machine as a sequence of n 1s on an otherwise blank tape. When and if the machine halts subsequently, the output number may be taken to be the number of non-blank squares on the tape at that time. There is nothing special about these conventions. There is no reason why we should choose these rather than others, but it is important to be specific at this point, because we require to associate with each Turing machine a unique (partial) function. Note that these conventions yield functions of one variable. If we considered the computations of Turing machines when their input tapes have a pair of numbers represented on them (for example as described in Example 7.15), then the machines would be associated with functions of two variables. Clearly we can treat larger numbers of variables similarly.

Remark 7.22

(*a*) It is known that for any number theoretic function f which is computable by Turing machine, there is a Turing machine which computes the values of f and whose alphabet of tape symbols is {B, 1}. We shall not prove this, but it is a consequence of the fact that the alphabet of any Turing machine is finite, and its symbols can be encoded as numbers in binary notation, using symbol B in place of 0.

(*b*) It is known also that for any number theoretic function f which is computable by Turing machine there is a Turing machine which computes f and which has only two internal states. Again we shall not prove this but we should observe that this reduction in the number of states is possible only by considerably enlarging the alphabet of tape symbols. Either the number of states may be reduced to two, or the number of symbols may be, but not in general both. (See Minsky, Ch. 6.)

Definition 7.23

A number theoretic (partial) function is *Turing computable* if there is a Turing machine which computes its values, under the above specified conventions regarding input and output.

We have no information as yet regarding the possibility of a single function corresponding in this way to more than one Turing machine, but we can now be precise.

Proposition 7.24

For each Turing computable (partial) function f there are infinitely many Turing machines which compute the values of f.

Proof. Given f and a Turing machine T which computes its values, suppose that $q_0, \ldots, q_k$ are the internal states of T. Adjoin the quadruple $q_{k+1}\ 1\ 1\ q_{k+1}$ to obtain a new Turing machine T^1. T^1 computes the values of f, since the new quadruple has no effect on any of the computations (state q_{k+1} is never entered). Similarly we can generate a sequence T^2, $T^3, \ldots$ of Turing machines by adjoining the quadruples $q_{k+2}\ 1\ 1\ q_{k+2}$, $q_{k+3}\ 1\ 1\ q_{k+3}, \ldots$ successively. Each of these computes the values of f.

Remark. It is important to note the distinction which arises here between a function and a set of instructions for computing its values. The Turing machines referred to in the above proof are different, since they have different sets of quadruples, but the differences are inessential in that they do not affect the computations carried out.

▷ Turing postulated (in 1936) that he had succeeded in doing what he set out to do, namely to characterise in a mathematically precise way, by means of his machines, the class of functions computable by algorithm, and the following statement is now known as *Turing's Thesis*:

> The class of Turing computable (partial) functions is the same as the class of (partial) functions computable by algorithm.

An equivalent way of stating this is to say that any algorithm (or set of instructions) for computing the values of a partial function f may be translated (effectively) into a set of quadruples for a Turing machine which computes the values of f.

As remarked previously, we cannot hope to prove Turing's Thesis, for it involves an intuitive notion (that of algorithm). However, a large amount of research work has led to many results which support Turing's Thesis. One of these is the following.

Proposition 7.25

A number theoretic (partial) function is Turing computable if and only if it is a recursive (partial) function.

Proof. The proof of this is highly technical, since it necessarily involves all the detail of the definitions of Turing machine and of recursive function. It is too lengthy to be included here, but the interested reader is referred to the book by Minsky.

▷ In the light of this, of course, Turing's Thesis is seen to be equivalent to Church's Thesis.

Proposition 7.26

There is an effective enumeration $\phi_0, \phi_1, \phi_2, \ldots$ of recursive partial functions of one variable, in which each recursive partial function occurs infinitely many times.

Proof. This is an immediate consequence of Propositions 7.21, 7.24 and 7.25.

▷ So much for the actual description of Turing machines and what they do. Let us now take things a stage further and discover where they lead us in regard to decision problems and solvability.

Consider the following algorithm: given any pair of numbers m, $n \in D_N$, enumerate the list $T_0, T_1, T_2, \ldots$ of Turing machines until T_m is found, and follow through the computation of T_m applied to the input n. This is an algorithm for computing the values of a (partial) function of two variables. By Turing's Thesis, then, there is a Turing machine which computes the values of this function. This may be put in a proposition.

Proposition 7.27

There exists a *universal Turing machine*. That is, there is a Turing machine T, which when interpreted as computing the values of a function of two variables m and n, carries out the computation of the machine T_m with input n.

▷ The universal machine can thus carry out the procedures of any of the machines $T_0, T_1, \ldots$ in computing functions of one variable. This is an indication of both the power and the limitations of Turing machines. The universal machine is apparently a complex and powerful object, for it embodies the capabilities of every one variable machine. On the other hand, the complexities of $T_0, T_1, \ldots$ are limited by the complexity of the universal machine.

Consider the following algorithm: given any $n \in D_N$, find ϕ_n in the list of recursive partial functions, and follow the computation (using T_n) of $\phi_n(n)$. If and when a result is obtained, add 1 to that result. By Church's Thesis, this algorithm defines a recursive partial function, say ϕ_{k_0}. Now let us ask what is the result of the computation of the value of $\phi_{k_0}(k_0)$? Follow the algorithm, and we obtain the contradiction

$$\phi_{k_0}(k_0) = \phi_{k_0}(k_0) + 1.$$

We have missed a point, however, in obtaining this, and that is that the

computation of $\phi_{k_0}(k_0)$ may not terminate. Indeed we can deduce from the above that it does not, for otherwise there is no way out of the contradiction.

The above illustrates how we can use the enumerations $T_0, T_1, \ldots$ and $\phi_0, \phi_1, \ldots$ in describing algorithms. This is a procedure which leads to interesting problems. It also can be used in the following way.

Proposition 7.28

There is no effective enumeration $f_0, f_1, \ldots$ of all recursive (total) functions of one variable.

Proof. Suppose the contrary, i.e. let $f_0, f_1, f_2, \ldots$ be an effective enumeration of all the recursive total functions of one variable (possibly with repetitions). Consider the following algorithm: given $n \in D_N$, enumerate $f_0, f_1, \ldots$ until f_n is obtained. Compute $f_n(n)$, and add 1. By Church's Thesis, this function h, with

$$h(n) = f_n(n) + 1 \quad \text{for each } n \in D_N,$$

is recursive, and it is total, since each f_n is total. So $h = f_k$ for some k. Thus

$$h(k) = f_k(k) = f_k(k) + 1.$$

This time there is no escape from the contradiction, and our result is proved.

▷ It would be convenient if for a given Turing machine computation we could tell in advance whether it will terminate. Thus the problem of deciding, given any pair m, $n \in D_N$, whether the Turing machine T_m will halt with input n, is a problem which has been examined. It is called the *Halting Problem* for Turing machines. Its importance is due in no small measure to the next result.

Proposition 7.29

The Halting Problem for Turing machines is unsolvable, i.e., there is no algorithm which provides answers to questions from the set

$$\{\text{does machine } T_m \text{ halt with input } n? \mid m, n \in D_N\}.$$

Proof. By Turing's Thesis it will be sufficient if we can show that the function $f: D_N \to D_N$ given by

$$f(n) = \begin{cases} 0 \text{ if } T_n \text{ halts on input } n \\ 1 \text{ if } T_n \text{ does not halt on input } n \end{cases}$$

is not Turing computable. For suppose that there is an algorithm for answering questions from the set given above. Then there is an algorithm for answering questions from the set

$$\{\text{does machine } T_n \text{ halt with input } n? \mid n \in D_N\},$$

and hence there is an algorithm for computing the values of the function f above. This cannot be the case if f is not Turing computable.

Let us concentrate on f, then, and suppose that it is Turing computable, and that the Turing machine T computes its values. Then, for input n, T will halt, with output 0 or 1 depending on whether T_n halts with input n or not. Let us now modify T to obtain a Turing machine T' such that for any n,

$$*\begin{cases}\text{if } T_n \text{ halts with input } n,\ T' \text{ does not halt with input } n, \text{ and}\\ \text{if } T_n \text{ does not halt with input } n,\ T' \text{ halts with input } n.\end{cases}$$

T' is obtained from T by including new states and quadruples which have the effect of appending, on to any computation by T which halts, a further search for a non-blank square on the output tape. If the search is successful, T' is to halt, otherwise T' is to continue searching indefinitely. The search can be successful (when T' has been started with input n) if and only if there is a 1 on the tape at this last stage, i.e. if and only if T_n does not halt with input n. (In detail, we could include two new states, q_α and q_β, say, and the following quadruples: $q_i\ S\ R\ q_\alpha$, for each pair $q_i\ S$ which do not occur as the first pair in any quadruple of T (S a tape symbol), and q_α B A q_β, q_α A R q_α, q_β A L q_β, q_β B A q_α.)

The machine T' must occur in the list $T_0, T_1, T_2, \ldots$. Say T' is T_{n_0}. Now let us ask the crucial question. Does T_{n_0} halt with input n_0? Return to * above. What it says is: T_{n_0} halts with input n_0 if and only if T_{n_0} does not halt with input n_0. This clear contradiction is sufficient to tell us that f cannot be Turing computable and that the result of the proposition is demonstrated.

▷ Consider now the following algorithm: given any $n \in D_N$, search in the list $T_0, T_1, T_2, \ldots$ until T_n is reached, and then follow the computation of the machine T_n with input n. If and when the computation halts, give output 1. By Turing's Thesis, the function whose values are computed by this algorithm is Turing computable. This function, ϕ, say, is given by

$$\phi(n) = \begin{cases}1 \text{ if } T_n \text{ halts with input } n,\\ \text{undefined, otherwise.}\end{cases}$$

It is clear that ϕ is a partial function, for certainly there are Turing machines which halt for no input and others which halt for only some

inputs. The domain of ϕ is an important set in this field of study, and is usually denoted by K.

$$K = \{n \in D_N : T_n \text{ halts with input } n\}.$$

Now it is a consequence of the proof of Proposition 7.29 that K is not a recursive set (under the assumption of Church's Thesis, of course), for if it were there would be an algorithm for answering questions of the form 'is $n \in K$?' for $n \in D_N$, i.e. questions from the set

$$\{\text{does machine } T_n \text{ halt with input } n? \mid n \in D_N\}.$$

Proposition 7.30

K is a recursively enumerable, non-recursive set.

Proof. K is not recursive, by the above. That K is effectively enumerable may be shown by giving an algorithm for enumerating it.

Step 1. Follow one step in the computation of T_0 with input 0.
Step 2. Follow one step in the computation of T_1 with input 1 and the second step in the computation of T_0 with input 0.
Step 3. Follow one step in the computation of T_2 with input 2 and the second step in the computation of T_1 with input 1 and the third step in the computation of T_0 with input 0.

and so on. Whenever one of the machines T_i halts, put i in the enumeration of K and disregard the references to T_i in the subsequent steps of the algorithm. For each $n \in K$ there will come a step when T_n with input n is found to halt, and n is put into the enumeration of K. K is thus effectively enumerable, and, by Church's Thesis, recursively enumerable.

Proposition 7.31

For any Turing machine T, the domain of T, i.e. the set of all $n \in D_N$ for which T halts with input n, is a recursively enumerable set (which may also be recursive, of course).

Proof. The proof is very similar to the above proof, following the computations, this time of the same machine T, on different inputs, simultaneously making a list of all the input numbers for which T halts.

▷ K is a concrete example of a non-recursive set which arises from consideration of Turing machines and the Enumeration Theorem.

There are many others. For example:

Example 7.32

(*a*) $\{n \in D_N : T_n$ halts for every input number$\}$ is not recursive, nor recursively enumerable.

(*b*) $\{n \in D_N : T_n$ halts for no input number$\}$ is not recursive, nor recursively enumerable.

(*c*) For each fixed n_0, the set $\{n \in D_N : T_n$ halts with input $n_0\}$ is not recursive, but is recursively enumerable.

These examples also give us classes of questions which are recursively unsolvable, according to Definition 7.11.

(*d*) $\{$is $n \in K$? $|n \in D_N\}$

(*e*) $\{$does T_n halt for every input number? $|n \in D_N\}$

(*f*) $\{$does T_n halt for some input number? $|n \in D_N\}$

(*g*) $\{$does T_n halt for input n_0? $|n \in D_N\}$, where n_0 is any fixed number.

These classes of problems are all recursively unsolvable.

The method used to verify all these results is similar. It is to assume that the set is recursive (or that the class of problems is recursively solvable), and to deduce that the set K is recursive (or that some other known non-recursive set is recursive). This method may be viewed in two ways. First, it can be seen as a straight proof by contradiction. Second, it can be seen in a wider context as introducing a new idea, that of *reducibility*.

Definition 7.33

The set A is *reducible* to the set B if the existence of an algorithm for deciding membership of B would guarantee the existence of an algorithm for deciding membership of the set A.

Neither algorithm may exist, but it is of interest often to discover whether two sets are related in this way. An obvious example of such reducibility is the result that $D_N \backslash K$ is reducible to K, although neither set has an algorithm for deciding membership. These ideas will not concern us here, however, and the interested reader is referred to the book by Rogers for further information.

▷ This concludes our investigation of Turing machines. Now we shall proceed to use the ideas and techniques we have developed in a discussion of unsolvability and undecidability results.

Exercises

7 Modify the Turing machine of Example 7.12 so that it will delete every symbol occurring on the input tape. (As it stands the machine will delete 1s until it comes to a B, when it will stop, so that if the input tape has a number of 1s scattered along it, not all will be deleted.)

8 Modify the Turing machine of Example 7.13 so that it halts with a completely blank tape if the input number is even and halts with a single 1 on the tape if the input number is odd.

9 Construct a Turing machine with alphabet of tape symbols {B, 1} which does not halt for any input tape.

10 Construct a Turing machine which when the input tape contains a single sequence of 1s will halt with the tape containing two sequences of 1s of that same length, separated by a symbol X.

11 Construct a Turing machine T which will compute the values of the partial function f, where

$$f(n) = \begin{cases} n & \text{if } n \text{ is odd} \\ \text{undefined} & \text{if } n \text{ is even.} \end{cases}$$

Modify T to obtain a Turing machine T' which behaves as T does if the input number is odd, but which halts with a blank tape if the input number is even. Give an example of a Turing computable partial function for which the above procedure is not possible.

12 Let T be a Turing machine whose set of quadruples contains no instruction to move left. Devise an effective procedure for deciding in advance for any input tape, whether T will eventually halt.

13 A Turing machine whose instructions permit it to move in only one direction along the tape is sometimes called a finite state machine. Show that the halting problem for finite state machines is solvable.

14 Prove that $\bar{K}$ is not recursively enumerable.

15 Let A be a recursively enumerable subset of $\bar{K}$ and suppose that for some Turing machine T_n, $A = \{x : T_n$ halts with input $x\}$. Show that $n \in \bar{K} \backslash A$.

16 Show that every recursively enumerable set is the domain of some Turing machine.

17 Show that each of the classes of problems in Example 7.32 is recursively unsolvable.

18 Let K_0 be the set $\{(m, n)$: Turing machine T_m halts with input $n\}$. Show that K is reducible to K_0, and that K_0 is reducible to K.

7.3 Word problems

One area where algebra and logic impinge on one another is in consideration of word problems for algebraic systems such as groups, semigroups and abelian groups. The case of semigroups is the simplest to describe, so we shall use it as an illustration.

Let $A = \{a_1, \ldots, a_k\}$ be a set of formal symbols. This set is to be regarded as an alphabet, and the *words* in this alphabet are just the finite strings of symbols from the alphabet (there are no restrictions on which strings are words). Denote by S_A the set of all words in the alphabet A.