1
Computable functions

We begin this chapter with a discussion of the fundamental idea of an algorithm or effective procedure. In subsequent sections we describe the way in which this idea can be made precise using a kind of idealised computer; this lays the foundation for a mathematical theory of computability and computable functions.

1. Algorithms, or effective procedures

When taught arithmetic in junior school we all learnt to add and to multiply two numbers. We were not merely taught that any two numbers have a sum and a product – we were given methods or rules for finding sums and products. Such methods or rules are examples of *algorithms* or *effective procedures*. Their implementation requires no ingenuity or even intelligence beyond that needed to obey the teacher's instructions.

More generally, an *algorithm* or *effective procedure* is a mechanical rule, or automatic method, or programme for performing some mathematical operation. Some more examples of operations for which easy algorithms can be given are

(1.1) (*a*) given n, finding the nth prime number,
(*b*) differentiating a polynomial,
(*c*) finding the highest common factor of two numbers (the Euclidean algorithm),
(*d*) given two numbers x, y deciding whether x is a multiple of y.

Algorithms can be represented informally as shown in fig. 1*a*. The input is the raw data or object on which the operation is to be performed (e.g. a polynomial for (1.1) (*b*), a pair of numbers for (1.1) (*c*) and (*d*)) and the output is the result of the operation (e.g. for (1.1) (*b*), the derived polynomial, and for (1.1) (*d*), the answer yes or no). The output is produced mechanically by the black box – which could be thought of as a

Fig. 1a.

ılculating machine, a computer, or a schoolboy correctly taught – or ven a very clever dog trained appropriately. The algorithm is the rocedure or method that is carried out by the black box to obtain the utput from the input.

When an algorithm or effective procedure is used to calculate the alues of a numerical function then the function in question is described y phrases such as *effectively calculable*, or *algorithmically computable*, or *ffectively computable*, or just *computable*. For instance, the functions xy, ICF(x, y) = the highest common factor of x and y, and $f(n)$ = the nth rime number, are computable in this informal sense, as already ıdicated. Consider, on the other hand, the following function:

$$g(n) = \begin{cases} 1 & \text{if there is a run of exactly } n \text{ consecutive 7s} \\ & \text{in the decimal expansion of } \pi, \\ 0 & \text{otherwise.} \end{cases}$$

lost mathematicians would accept that g is a perfectly legitimate ınction. But is g computable? There *is* a mechanical procedure for enerating successive digits in the decimal expansion of π,[1] so the ɔllowing 'procedure' for computing g suggests itself.

'Given n, start generating the decimal expansion of π, one digit at a me, and watch for 7s. If at some stage a run of exactly n consecutive 7s as appeared, then stop the process and put $g(n) = 1$. If no such sequence f 7s appears put $g(n) = 0$.'

The problem with this 'procedure' is that, if for a particular n there is no equence of exactly n consecutive 7s, then there is no stage in the process ʼhere we can stop and conclude that this is the case. For all we know at ny particular stage, such a sequence of 7s could appear in the part of the xpansion of π that has not yet been examined. Thus the 'procedure' will o on for ever for inputs n such that $g(n) = 0$; so it is not an *effective* rocedure. (It is conceivable that there *is* an effective procedure for omputing g based, perhaps, on some theoretical properties of π. At the resent time, however, no such procedure is known.)

[1] This will be established in chapter 3 (example 7.1(3)).

This example pinpoints two features implicit in the idea of an effective procedure – namely, that such a procedure is carried out in a sequence of stages or steps (each completed in a finite time), and that any output should emerge after a finite number of steps.

So far we have described informally the idea of an algorithm, or effective procedure, and the associated notion of computable function. These ideas must be made precise before they can become the basis for a mathematical theory of computability – and *non*-computability.

We shall make our definitions in terms of a simple 'idealised computer' that operates programs. Clearly, the procedures that can be carried out by a real computer are examples of effective procedures. Any particular real computer, however, is limited both in the size of the numbers that it can receive as input, and in the amount of working space available; it is in these respects that our 'computer' will be idealised in accordance with the informal idea of an algorithm. The programs for our machine will be finite, and we will require that a completed computation takes only a finite number of steps. Inputs and outputs will be restricted to natural numbers; this is not a significant restriction, since operations involving other kinds of object can be coded as operations on natural numbers. (We discuss this more fully in § 5.)

2. The unlimited register machine

Our mathematical idealisation of a computer is called an *unlimited register machine* (URM); it is a slight variation of a machine first conceived by Shepherdson & Sturgis [1963]. In this section we describe the URM and how it works; we begin to explore what it can do in § 3.

The URM has an infinite number of *registers* labelled $R_1, R_2, R_3, \ldots$, each of which at any moment of time contains a natural number; we denote the number contained in R_n by r_n. This can be represented as follows

R_1	R_2	R_3	R_4	R_5	R_6	R_7	…
r_1	r_2	r_3	r_4	r_5	r_6	r_7	…

The contents of the registers may be altered by the URM in response to certain *instructions* that it can recognise. These instructions correspond to very simple operations used in performing calculations with numbers. A finite list of instructions constitutes a *program*. The instructions are of four kinds, as follows.

o instructions For each $n = 1, 2, 3, \ldots$ there is a *zero instruction* ı). The response of the URM to the instruction Z(n) is to change the :tents of R_n to 0, leaving all other registers unaltered.

Example Suppose that the URM is in the following ıfiguration

R_1	R_2	R_3	R_4	R_5	R_6	
9	6	5	23	7	0	...

l obeys the zero instruction Z(3). Then the resulting configuration is

9	6	0	23	7	0	...

: response of the URM to a zero instruction Z(n) is denoted by $0 \to R_n$, $r_n := 0$ (this is read r_n *becomes* 0).

:cessor instructions For each $n = 1, 2, 3, \ldots$ there is a *successor 'ruction* S(n). The response of the URM to the instruction S(n) is to :rease the number contained in R_n by 1, leaving all other registers ıltered.

Example Suppose that the URM is in the configuration (*) ›ve and obeys the successor instruction S(5). Then the new conıration is

R_1	R_2	R_3	R_4	R_5	R_6	
9	6	0	23	8	0	...

)

: effect of a successor instruction S(n) is denoted by $r_n + 1 \to R_n$, or $= r_n + 1$ (r_n *becomes* $r_n + 1$).

ınsfer instructions For each $m = 1, 2, 3, \ldots$ and $n = 1, 2, 3, \ldots$ there a *transfer instruction* T(m, n). The response of the URM to the truction T(m, n) is to replace the contents of R_n by the number r_m ıtained in R_m (i.e. transfer r_m into R_n); all other registers (including) are unaltered.

Example Suppose that the URM is in the configuration (**) ›ve and obeys the transfer instruction T(5, 1). Then the resulting configuration is

R_1	R_2	R_3	R_4	R_5	R_6	
8	6	0	23	8	0	...

The response of the URM to a transfer instruction T(m, n) is denoted by $r_m \to R_n$, or $r_n := r_m$ (r_n *becomes* r_m).

Jump instructions In the operation of an informal algorithm there may be a stage when alternative courses of action are prescribed, depending on the progress of the operation up to that stage. In other situations it may be necessary to repeat a given routine several times. The URM is able to reflect such procedures as these using *jump instructions*; these will allow jumps backwards or forwards in the list of instructions. We shall, for example, be able to use a jump instruction to produce the following response:

> 'If $r_2 = r_6$, go to the 10th instruction in the program; otherwise, go on to the next instruction in the program.'

The instruction eliciting this response will be written J(2, 6, 10).

Generally, for each $m = 1, 2, 3, \ldots, n = 1, 2, 3, \ldots$ and $q = 1, 2, 3, \ldots$ there is a *jump instruction* J(m, n, q). The response of the URM to the instruction J(m, n, q) is as follows. Suppose that this instruction is encountered in a program P. The contents of R_m and R_n are compared, but all registers are left unaltered. Then

> if $r_m = r_n$, the URM proceeds to the qth instruction of P;
>
> if $r_m \neq r_n$, the URM proceeds to the next instruction in P.

If the jump is impossible because P has less than q instructions, then the URM stops operation.

Zero, successor and transfer instructions are called *arithmetic* instructions.

We summarise the response of the URM to the four kinds of instruction in table 1.

Computations To perform a computation the URM must be provided with a program P and an *initial configuration* – i.e. a sequence $a_1, a_2, a_3, \ldots$ of natural numbers in the registers $R_1, R_2, R_3, \ldots$. Suppose that P consists of s instructions $I_1, I_2, \ldots, I_s$. The URM begins the computation by obeying I_1, then I_2, I_3, and so on unless a jump

Table 1

Type of instruction	Instruction	Response of the URM
Zero	Z(n)	Replace r_n by 0. ($0 \to R_n$, or $r_n := 0$)
Successor	S(n)	Add 1 to r_n. ($r_n + 1 \to R_n$, or $r_n := r_n + 1$)
Transfer	T(m, n)	Replace r_n by r_m. ($r_m \to R_n$, or $r_n := r_m$)
Jump	J(m, n, q)	If $r_m = r_n$, jump to the qth instruction; otherwise go on to the next instruction in the program.

instruction, say J(m, n, q), is encountered. In this case the URM proceeds to the instruction prescribed by J(m, n, q) and the current contents of the registers R_m and R_n. We illustrate this with an example.

2.1. *Example*

Consider the following program:

I_1 J(1, 2, 6)
I_2 S(2)
I_3 S(3)
I_4 J(1, 2, 6)
I_5 J(1, 1, 2)
I_6 T(3, 1)

Let us consider the computation by the URM under this program with initial configuration

R_1	R_2	R_3	R_4	R_5	
9	7	0	0	0	...

(We are not concerned at the moment about what function this program actually computes; we wish to illustrate the way in which the URM operates programs in a purely mechanical fashion *without* needing to understand the algorithm that is being carried out.)

We can represent the progress of the computation by writing down the successive configurations that occur, together with the next instruction to be obeyed at the completion of each stage.

	R_1	R_2	R_3	R_4	R_5		Next instruction
Initial configuration	9	7	0	0	0	...	I_1
	9	7	0	0	0		I_2 (since $r_1 \neq r_2$)
	9	8	0	0	0		I_3
	9	8	1	0	0		I_4
	9	8	1	0	0		I_5 (since $r_1 \neq r_2$)
	9	8	1	0	0		I_2 (since $r_1 = r_1$)

and so on. (We shall continue this computation later.)

We can describe the operation of the URM under a program $P = I_1, I_2, \ldots, I_s$ in general as follows. The URM starts by obeying instruction I_1. At any future stage in the computation, suppose that the URM is obeying instruction I_k. Then having done so it proceeds to the *next instruction in the computation*, defined as follows:

if I_k is not a jump instruction, the *next instruction* is I_{k+1};

if $I_k = J(m, n, q)$ the *next instruction* is $\begin{cases} I_q & \text{if } r_m = r_n, \\ I_{k+1} & \text{otherwise,} \end{cases}$

where r_m, r_n are the current contents of R_m and R_n.

The URM proceeds thus as long as possible; the computation *stops* when, and only when, there is no next instruction; i.e. if the URM has just obeyed instruction I_k and the 'next instruction in the computation' according to the above definition is I_v where $v > s$. This can happen in the following ways:

(i) if $k = s$ (the last instruction in P has been obeyed) and I_s is an arithmetic instruction,
(ii) if $I_k = J(m, n, q)$, $r_m = r_n$ and $q > s$,
(iii) if $I_k = J(m, n, q)$, $r_m \neq r_n$ and $k = s$.

We say then that the computation stops after instruction I_k; the *final configuration* is the sequence $r_1, r_2, r_3, \ldots$, the contents of the registers at this stage.

Let us now continue the computation begun in example 2.1.

Example 2.1 (continued)

	R_1	R_2	R_3	R_4	R_5		Next instruction
	9	8	1	0	0	...	I_2
	9	9	1	0	0		I_3
	9	9	2	0	0		I_4
	9	9	2	0	0		I_6 (since $r_1 = r_2$)
Final configuration	2	9	2	0	0		I_7: STOP.

This computation stops as indicated because there is no seventh instruction in the program.

2.2. *Exercise*

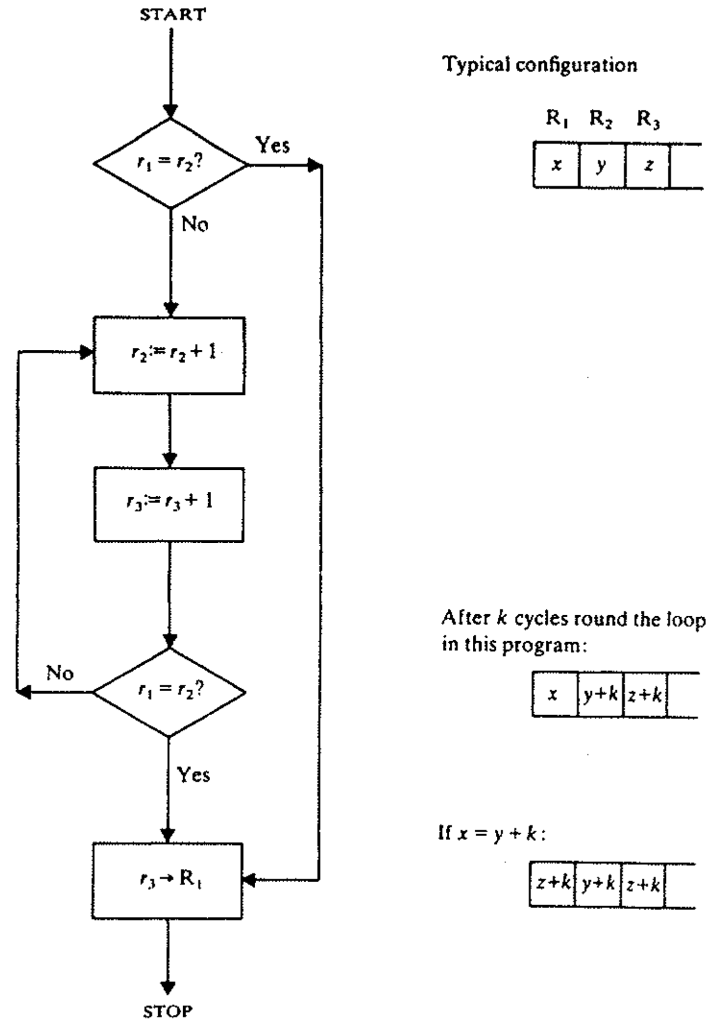
Carry out the computation under the program of example 2.1 with initial configuration 8, 4, 2, 0, 0, . . .

The essence of a program and the progress of computations under it is often conveniently described informally using a *flow diagram*. For example, a flow diagram representing the program of example 2.1 is given in fig. 1*b*. (We have indicated alongside the flow diagram the typical configuration of the registers at various stages in a computation.) Note the convention that tests or questions (corresponding to jump instructions) are placed in diamond shaped boxes.

The translation of this flow diagram into the program of exercise 2.1 is almost self-explanatory. Notice that the backwards jump on answer 'No' to the second question '$r_1 = r_2$?' is achieved by the fifth instruction J(1, 1, 2) which is an *unconditional* jump: we always have $r_1 = r_1$, so this instruction causes a jump to I_2 whenever it is encountered.

When writing a program to perform a given procedure it is often helpful to write an informal flow diagram as an intermediate step: the translation of a flow diagram into a program is then usually routine.

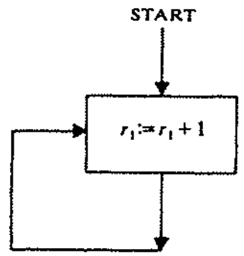
Fig. 1*b*. Flow diagram for the program of example 2.1.



There are, of course, computations that never stop: for example, no computation under the simple program S(1), J(1, 1, 1) ever stops. Computation under this program is represented by the flow diagram in fig. 1*c*. The jump instruction invariably causes the URM to return, or loop back, to the instruction S(1).

There are more sophisticated ways in which a computation may run for ever, but always this is caused essentially by the above kind of repetition or looping back in the execution of the program.

Fig. 1c.



2.3 *Exercise*

Show that the computation under the program of example 2.1 with initial configuration 2, 3, 0, 0, 0, ... never stops.

The question of deciding whether a particular computation eventually stops or not is one to which we will return later.

Some notation will help us now in our discussion. Let $a_1, a_2, a_3, \ldots$ be an infinite sequence from $\mathbb{N}$ and let P be a program; we will write

(i) $P(a_1, a_2, a_3, \ldots)$ for the computation under P with initial configuration $a_1, a_2, a_3, \ldots$;

(ii) $P(a_1, a_2, a_3, \ldots)\downarrow$ to mean that the computation $P(a_1, a_2, a_3, \ldots)$ eventually stops;

(iii) $P(a_1, a_2, a_3, \ldots)\uparrow$ to mean that the computation $P(a_1, a_2, a_3, \ldots)$ never stops.

In most initial configurations that we shall consider, all but finitely many of the a_i will be 0. Thus the following notation is useful. Let $a_1, a_2, \ldots, a_n$ be a finite sequence of natural numbers; we write

(iv) $P(a_1, a_2, \ldots, a_n)$ for the computation $P(a_1, a_2, \ldots, a_n, 0, 0, 0, \ldots)$,

Hence

(v) $P(a_1, a_2, \ldots, a_n)\downarrow$ means that $P(a_1, a_2, \ldots, a_n, 0, 0, 0, \ldots)\downarrow$;

(vi) $P(a_1, a_2, \ldots, a_n)\uparrow$ means that $P(a_1, a_2, \ldots, a_n, 0, 0, 0, \ldots)\uparrow$.

Often a computation that stops is said to *converge*, and one that never stops is said to *diverge*.

3. URM-computable functions

Suppose that f is a function from $\mathbb{N}^n$ to $\mathbb{N}$ ($n \geq 1$); what does it mean to say that f is computable by the URM? It is natural to think in terms of computing a value $f(a_1, \ldots, a_n)$ by means of a program P on initial configuration $a_1, a_2, \ldots, a_n, 0, 0, \ldots$. That is, we consider computations of the form $P(a_1, a_2, \ldots, a_n)$. If any such computation

stops, we need to have a single number that we can regard as the output or result of the computation; we make the convention that this is the number r_1 finally contained in R_1. The final contents of the other registers can be regarded as rough work or jottings, that can be ignored once we have the desired result in R_1.

Since a computation $P(a_1, \ldots, a_n)$ may not stop, we can allow our definition of computability to apply to functions f from $\mathbb{N}^n$ to $\mathbb{N}$ whose domain may not be all of $\mathbb{N}^n$; i.e. partial functions. We shall require that the relevant computations stop (and give the correct result!) *precisely* for inputs from the domain of f. Thus we make the following definitions.

3.1 *Definitions*

Let f be a partial function from $\mathbb{N}^n$ to $\mathbb{N}$.

(*a*) Suppose that P is a program, and let $a_1, a_2, \ldots, a_n, b \in \mathbb{N}$.

(i) The computation $P(a_1, a_2, \ldots, a_n)$ *converges to* b if $P(a_1, a_2, \ldots, a_n)\downarrow$ and in the final configuration b is in R_1. We write this $P(a_1, \ldots, a_n)\downarrow b$;

(ii) P *URM-computes* f if, for every $a_1, \ldots, a_n, b$ $P(a_1, \ldots, a_n)\downarrow b$ if and only if $(a_1, \ldots, a_n) \in \mathrm{Dom}(f)$ and $f(a_1, \ldots, a_n) = b$. (In particular, this means that $P(a_1, \ldots, a_n)\downarrow$ if and only if $(a_1, \ldots, a_n) \in \mathrm{Dom}(f)$.)

(*b*) The function f is *URM-computable* if there is a program that URM-computes f.

The class of URM-computable functions is denoted by $\mathscr{C}$, and n-ary URM-computable functions by $\mathscr{C}_n$. From now on we will use the term *computable* to mean URM-computable, except in chapter 3 where other notions of computability are discussed.

We now consider some easy examples of computable functions.

3.2 *Examples*

(*a*) $x + y$.

We obtain $x + y$ by adding 1 to x (using the successor instruction) y times. A program to compute $x + y$ must begin on initial configuration $x, y, 0, 0, 0, \ldots$; our program will keep adding 1 to r_1, using R_3 as a counter to keep a record of how many times r_1 is thus increased. A typical configuration during the computation is

R_1	R_2	R_3	R_4	R_5	
$x + k$	y	k	0	0	...