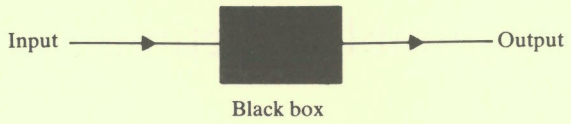
Fig. 1*a*.



calculating machine, a computer, or a schoolboy correctly taught – or even a very clever dog trained appropriately. The algorithm is the procedure or method that is carried out by the black box to obtain the output from the input.

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

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

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

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

The problem with this 'procedure' is that, if for a particular n there is no sequence of exactly n consecutive 7s, then there is no stage in the process where we can stop and conclude that this is the case. For all we know at any particular stage, such a sequence of 7s could appear in the part of the expansion of π that has not yet been examined. Thus the 'procedure' will go on for ever for inputs n such that $g(n) = 0$; so it is not an *effective* procedure. (It is conceivable that there *is* an effective procedure for computing g based, perhaps, on some theoretical properties of π. At the present 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 $\mathrm{R}_1, \mathrm{R}_2, \mathrm{R}_3, \ldots$, 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.

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

Example Suppose that the URM is in the following configuration

R_1	R_2	R_3	R_4	R_5	5_6	
9	6	5	23	7	0	...

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

(*)

9	6	0	23	7	0	...

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

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

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

(**)

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

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

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

Example Suppose that the URM is in the configuration (**) above 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.