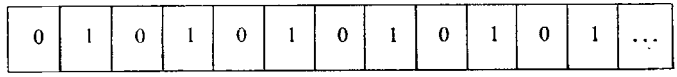
12

Algorithms

12.1. EXAMPLES OF ALGORITHMS

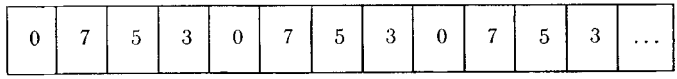
In our previous discussion we have examined many infinite sequences without stopping to think what it means "to examine" an infinite sequence. Obviously, we cannot scan it, and the only understanding of such a sequence which we can achieve derives from the analysis of its properties. We shall illustrate this concept by some examples.

Determination of a term in an infinite sequence. Consider the sequence



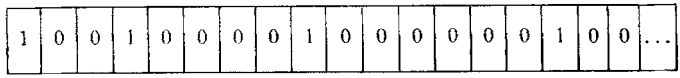
There is little that we can forecast about its further behavior. However, if we know that the symbols 0 and 1 always alternate, we can predict the term appearing in any position, because 0 always appears in an odd-numbered, and 1 in an even-numbered position.

Now consider the sequence



If we know that group 0, 7, 5, 3 is recurrent, and if we know the first term of this sequence, we can again determine any subsequent term. To find the term appearing in the nth position, we divide n by the number of terms in the recurrent group. The remainder obtained in this division indicates the position of the term in the recurrent group. If the remainder is 0, the nth term coincides with the last term of that group.

As a final example, consider the sequence



Here 1 appears only in positions whose "addresses" are squares of integers. If we know that, we know everything about the sequence. To find the nth term, we merely take the square root of n. If it is an integer, the nth term is 1. Otherwise, it is 0.

These three examples had one common "property" which enabled us to reconstitute the entire infinite sequence starting from a small segment of it. In all cases we had a "prediction procedure," that is, a procedure for determining any term, given its "address." To be more exact, in all three cases we dealt with an *algorithm* for finding the term, given the ordinal number of its position in the sequence.

An algorithm usually means a set of formal directions for obtaining the required solution. This formulation is not exact but rather expresses an intuitive concept which dates back to antiquity.*

To clarify the characteristic properties of an algorithm, let us consider some typical examples.

The Euclidean algorithm. This algorithm determines the greatest common divisor of two positive integers a and b, and may be described by the following sequence of directions:

1. Compare a and b ($a = b$, or $a < b$, or $a > b$). Go on to 2.

2. If $a = b$ then either is the greatest common divisor. Stop the computation. If $a \neq b$ go on to 3.

3. Subtract the smaller from the larger number and write down the subtrahend and the remainder. Go to the next instruction.

4. Assign symbol a to the subtrahend, and symbol b to the remainder. Return to direction 1.

The procedure is repeated until $a = b$. Then the computation is stopped.

The above set of directions, each consisting of a simple arithmetical operation (subtraction, comparison) can obviously be made more detailed, in which case the direction will be still simpler.

Algorithms which reduce the solution to arithmetical operations are termed *numerical algorithms*. Our three previous examples belonged to this class, as do formulas and procedures for solution of any class of problems, provided such formulas fully express both the nature of the operations (multiplication, subtraction, or division) and the order in which they must be performed.

A logical algorithm. Now consider an algorithm for solving a logical problem—that of finding a path in a finite labyrinth.

*The term "algorithm" itself derives from the name of the ninth century Uzbek mathematician al-Khuwārizmi, who formulated a set of formal directions, that is, rules for carrying out the four operations of arithmetic in the decimal system.

Imagine a finite system of rooms, each of which is the origin of one or more corridors. Each corridor joins two adjacent rooms, and a room may thus be connected to several other rooms. On the other hand, it may open only into a single corridor, in which case it will be a "dead-end" room. Graphically, the resulting labyrinth may be shown as a system of circles A, B, C, ..., joined by straight lines (Fig. 12.1). We shall say that room Y is accessible from room X if there exists a path leading from X to Y via intermediate corridors and rooms. This means that either X and Y are adjacent rooms or there exists a sequence of adjacent rooms X, X_1, X_2, X_3, ..., X_n, Y. If Y is accessible from X, then the path from X to Y must be simple (loopless); that is, each intermediate room is traversed only once. Thus, in the labyrinth of Fig. 12.1, one simple path from H to B is $HDCB$; but L is inaccessible from A.

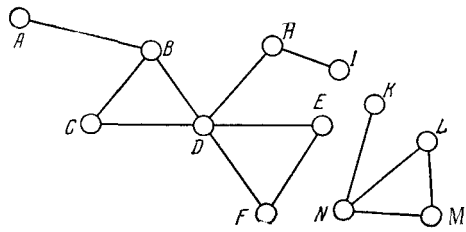


Fig. 12.1.

Suppose that we wish to ascertain whether F is accessible from A and that, if so, we wish to find a path from A to F; but, if F proves inaccessible, we wish to return to A at the end of the search. We have no map of this labyrinth and for that reason shall employ a general search procedure applicable to any labyrinth containing a finite number of rooms, with any mutual disposition of rooms A and F within it. In one such procedure, the searcher, like the mythical Theseus, holds a ball of thread, one end of which is tied down in the starting room A. In addition, the searcher can paint the corridors as he walks along them. He is thus able to distinguish those never passed before (green), those passed once (yellow), and those passed twice (red). The searcher can get from any room to an adjacent one by either of two moves:

a) He can unwind the thread. He thus stretches the thread along a "green" corridor, which then becomes "yellow."

b) He can wind the thread on the ball. He thus returns from a given room to an adjacent one, walking along a "yellow" corridor.

He picks up the thread as he walks along, and the corridor now becomes "red."*

Having arrived into any given room, the searcher may encounter any one of five possibilities:

1. This is room F, the object of the search.
2. At least two "yellow" corridors radiate from this room, that is, a thread is already stretched across it. The searcher now realizes that he has just traversed a loop.
3. At least one "green" corridor originates in this room.
4. This is the starting room A.
5. None of the above.

Now the search procedure can be specified by the following table:

Evidence in the room	*Next move*
1. Room F	Stop
2. Loop	Wind the thread
3. "Green" corridor	Unwind the thread
4. Room A	Stop
5. Fifth possibility	Wind the thread

In each room, the searcher must decide on his next move by scanning the table in numerical order and ascertaining which of the possibilities listed matches the evidence in the room. Having found the first match, he makes the necessary move without checking for other applicable possibilities. He continues to move in this fashion until the instructions are to stop.

For this procedure, we can prove the following:

1. After a finite number of moves, the searcher will stop at either A or F, regardless of the mutual dispositions of A and F.
2. If he stops at F, then he has reached his object, and the thread is now stretched along a simple path from A to F.
3. If he stops at A, then F is inaccessible.

Let us illustrate the procedure on the labyrinth of Fig. 12.1. We represent the procedure in the form of Table 12.1 and see that F is accessible from A. We write down those corridors of column 4 which remain "yellow" to the very end. These constitute the simple path from A to F.

This procedure involves an element of choice which did not exist in the previously discussed examples. Thus, two different calculators trying to find the greatest common divisor of two numbers by

*In fact, the searcher needs only two colors—green and red; this is because the "yellow" corridors contain the stretched thread.

Table 12.1

Move No.	Evidence in room determining next move	Next Move	Path (corridor) Taken	Corridor color after this move
1	Green corridor	Unwind thread	*AB*	Yellow
2	"	"	*BC*	"
3	"	"	*CD*	"
4	"	"	*DH*	"
5	"	"	*HI*	"
6	Fifth case	Wind thread	*IH*	Red
7	"	"	*HD*	"
8	Green corridor	Unwind thread	*DB*	Yellow
9	Loop	Wind thread	*BD*	Red
10	Green corridor	Unwind thread	*DF*	Yellow
11	Room *F*	STOP		

Euclid's algorithm performs operations coinciding in every detail: there is no room for individual judgment. But in the labyrinth procedure, two searchers may go from *A* to *F* via distinctly different paths.

Traditionally, the term "algorithm" is restricted to an exactly defined set of instructions. In this sense, the labyrinth search procedure is not an algorithm. To become one, it would have to be supplemented by an exact specification on what to do in the case of "green" corridors (for example, this supplemental instruction may specify that if several "green" corridors originate from the same room, the searcher must select the first one to the right of the entrance).

12.2. GENERAL PROPERTIES OF ALGORITHMS

The above examples point out some overall properties characteristic of any algorithm:

(a) *Determinancy.* The procedure is specified so clearly and precisely that there is no room for arbitrary interpretation. A procedure of this kind can be communicated to another person by a finite number of instructions. The operations described by these instructions do not depend on the whim of the operator and constitute a determinate process which is completely independent of the person carrying it out.

(b) *Generality.* An algorithm is applicable to more than just one specific problem: it is used for solving a class of problems, with the procedural instructions valid for any particular set of initial data. Thus, Euclid's algorithm is applicable to any pair of integers

$a > 0$, $b > 0$; the rules of arithmetic apply to all numbers; and the search rules hold for any finite labyrinth, however intricate.

In mathematics one considers a series of problems of a specific kind to be solved when an algorithm has been found (the finding of such algorithms is really the object of mathematics). But in the absence of an algorithm applicable to *all* problems of a given type, one is forced to devise a special procedure valid in some but not other cases. However, such a procedure is not an algorithm. For instance, there is no algorithm for finding out whether the solution of equation

$$x^n + y^n = z^n \tag{12.1}$$

is an integer at any $n = 1, 2, 3, 4, \ldots$. This problem may nevertheless be solved for particular values of n. Thus, for $n = 2$, we can easily find three numbers ($x = 3$, $y = 4$, $z = 5$) satisfying Eq. (12.1). And it may be proved that Eq. (12.1) has no integer solutions for $n = 3$. However, this proof cannot be extended to other values of n.

(c) *Efficacy.* This property, sometimes called the *directionality* of an algorithm, means that application of an algorithmic procedure to any problem of a given kind will lead to a "stop" instruction in a finite number of steps, at which point one must be able to find the required solution. Thus, no matter now intricate the (finite) labyrinth, the search algorithm must lead to a "stop" instruction in a finite number of steps. The stop will occur either at F or at A, enabling us to decide whether F is accessible or not. Again, the use of the Euclidean algorithm with any two numbers $a \geqslant 1$, $b \geqslant 1$ will sooner or later lead to a "stop" instruction, at which point one can determine the value of the greatest common divisor. However, nothing prevents us from using the Euclidean algorithm with $a \geqslant 0$ and $b \geqslant 0$, or with any pair of integers (positive or negative). There is no ambiguity at any step of the algorithm, but the procedure may not come to a stop. For example, if $a = 0$, $b = 4$, our sequence of instructions (1-4) gives the pairs 0, 4; 0, 4; 0, 4... and so on *ad infinitum*. The same will happen with the pair $a = -2$, $b = 6$.

Thus, the concept of efficacy of an algorithm naturally leads to the concept of its *range of application*. The range of application is the largest range of initial data for which the algorithm will yield results; in other words, if the problem is stated within the range of application, then the algorithm will work up the (given) conditions into a solution, after which the procedure will come to a stop; if, however, the problem conditions are outside this range, then either there will be no stop, or there will be a stop but we shall not be able to obtain a result. Thus, the range of application of the Euclidean

algorithm is the set of natural numbers {1, 2, 3, 4, . . .}, and the range of application of the search algorithm is the set of all finite labyrinths.

Now, the number of individual operations which must be performed in an algorithmic procedure is not known beforehand and depends on the choice of the initial data. For this reason, an algorithm should be understood primarily as a *potentially* feasible procedure. In some specific problems stated within the range of application of the algorithm there may be no *practical* solution: the procedure may be so long that the calculator will run out of paper, ink, or time, or the computer executing the algorithm may not have a large enough memory.

Determinancy, generality, and efficacy are empirical properties. They are present in all algorithms constructed so far. However, these empirical properties are too vague and inexact to be useful in a mathematical theory of algorithms. We shall try to refine them in subsequent sections.

12.3. THE WORD PROBLEM IN ASSOCIATIVE CALCULUS

The previously described search procedure was restricted to finite, though arbitrary, labyrinths. There exists, however, a far reaching generalization of this problem which, in a sense, constitutes a search in an infinite labyrinth. This is the *word problem*. It arose first in algebra, in the theory of associative systems, and in the group theory, but the conclusions derived from it have since transcended these specialized fields.*

Let any finite system of differing symbols constitute an alphabet, and let the constituent symbols be its characters. For instance, {а, ъ, ?, 7, *} is an alphabet, whereas а, ъ, ?, 7 and * are its characters. Any finite sequence of characters from an alphabet is called a *word* of that alphabet. Thus, the three-character alphabet {*a*, *b*, *c*} will yield words *ac*, *a*, *abbca*, *bbbbb*, *bbacab*, etc. An empty word, containing no characters, is denoted by Λ.

Consider two words L and M in some alphabet A. If L is a part of M, then we say that L occurs in M. For example, the word $L = ac$ occurs in $M = bbacab$. In general, L may occur in M many times; thus *bcb* occurs twice in *abcbcbab*.

*Important contributions to its solution were made by A.A. Markov, P.S. Novikov and their students. Familiarity with this problem will help us to understand the theory of recursive functions.

We shall now describe the process of transformation of words, whereby new words are obtained from given ones. We start by setting up a finite system of substitutions allowable in a given alphabet:

$$P - Q; \quad L - M; \quad \ldots; \quad S - T,$$

where P, Q, L, M, ..., S, T are words in this alphabet, and the dashes between them denote substitution. Thus, an L—M substitution in a word R of this alphabet may be defined as follows: if L occurs one or more times in R, then any one of these occurrences may be replaced with M; conversely, if M occurs in R, then it may be replaced with L. For example, there are four possible substitutions $ab - bcb$ in $abcbcbab$. Replacement of each subword bcb with ab yields $aabcbab$ and $abcabab$, respectively, whereas the replacement of each ab yields the words $bcbcbcbab$ and $abcbcbbcb$. However, the substitution ab—bcb in $bacb$ is not allowed since neither ab nor bcb occurs in it.

The words obtained by means of allowable substitutions may be again replaced, which yields yet new words.

The aggregate of all the words in a given alphabet, together with an appropriate set of allowable substitutions, is called an ***associative calculus***. To define an associative calculus, it is sufficient to define the alphabet and the set of substitutions.

Two words P_1 and P_2 of an associative calculus are said to be *adjacent* if one may be transformed into the other by a single allowable substitution. A sequence of words P, P_1, P_2, P_3, ..., Q is called a *deductive chain* leading from P to Q if every two consecutive words in this chain are adjacent. Two words P and Q are said to be *equivalent* if there exists a deductive chain leading from P to Q. The equivalence relationship is shown as $P \sim Q$. It is obvious that if $P \sim Q$, then $Q \sim P$, since the allowable substitutions may be used in either direction.

Example. Assume we have the following associative calculus:

$\{a, b, c, d, e\}$ — alphabet

$ac - ca$
$ad - da$
$bc - cb$
$bd - db$
$abac - abacc$
$eca - ae$
$edb - be$

— allowable substitutions

Here, the words *abcde* and *acbde* are adjacent, since *abcde* may be transformed into *acbde* by the substitution *bc—cb*. However, *aaabb* has no adjacent words, since none of the given substitutions may be applied to it. The word *abcde* is equivalent to *cadedb*, since there exists a deductive chain of adjacent words: *abcde, acbde, cabde, cadbe, cadedb* , derived by successive use of the third, first, fourth, and fifth of the above substitutions.

An associative calculus may be put in correspondence with an infinite labyrinth by matching a specific room of the labyrinth with a word from the alphabet. Since the number of words which may be formed from the characters of a given alphabet is infinite, it follows that the labyrinth can have an infinite number of rooms.

Two adjacent rooms of the labyrinth correspond to adjacent words. Now, if two words P and Q are equivalent, then the labyrinth room corresponding to word Q is accessible from the room corresponding to P, that is, there exists a path from P to Q.*

In each associative calculus there occurs a specific *word problem*, whereby it is required to recognize *whether two words of this calculus are equivalent or not*. This problem is identical to our problem of accessibility in a labyrinth, except that here the labyrinth is infinite. But our previous algorithm is now useless because an infinite labyrinth cannot be searched in a finite time.

Since each associative calculus contains an infinite set of different words, it involves an infinite number of problems of equivalence between two words. All these problems must involve the same procedures, and we therefore naturally think of a solution consisting of an algorithm for recognizing the equivalence (or nonequivalence) of any pair of words. We shall see in the next section whether such a solution does exist.

However, we can immediately find the algorithm for the *restricted word problem*, where one wants to know whether a given word can be transformed into another by using the allowable substitutions a maximum of k times; here k is an arbitrary but fixed number. In the previous search problem, we imposed a restriction on the labyrinth, which had to be finite. Here, however, it is the number of moves which is restricted: we must inspect all those rooms of an finite labyrinth which are separated from the starting one by

*Sometimes one uses a form of associative calculus defined by an alphabet and a system of *oriented* substitutions $P \to Q$. The arrow means that only left-to-right substitution is allowed; that is, the word P may be replaced with Q, but not the other way around. This associative calculus may be graphically represented by an infinite labyrinth in which each of the corridors is unidirectional. Obviously, the equivalence $P \sim Q$ in no way implies that $Q \sim P$ in this case.

not more than k corridors; the remainder of the labyrinth is of no interest. In terms of associative calculus this means that one determines all words adjacent to one of the given words (by substitution); then for each of the new words so derived one determines all words adjacent to it, and so on, k times in all. We finally obtain a list of all words which may be derived from the given one by using the allowable substitutions at most k times. If the second given word appears in that list, then the answer to the restricted word problem is yes; if it does not appear, the answer is no. This scanning algorithm may then be further improved by removing from it all superfluous iterations (loops).

However, the solution of the restricted word problem does not bring us nearer to the solution of our basic "unlimited" problem. Here, the length of the deductive chain (if it exists) from word P to word Q may be extremely great (or infinite). For this reason, the scanning algorithm, restricted as it is to k substitutions, generally cannot tell whether an equivalence is present or, to put it another way, when to stop the search for such an equivalence. Thus we must turn to other, more sophisticated algorithms, as we shall do in later sections.

The reader will now recognize that *logical deductive* processes other than the search problem may also be treated as associative calculi. For instance, any logical formula may be interpreted as a word in some alphabet containing the characters denoting logical variables, logical functions, and logical connectives $\vee$, $\&$, $^-$, $\to$ (,), etc. The process of *inference* may be treated as a formal word transformation similar to the substitution in associative calculus, in which an elementary act of logical inference is made to correspond to a single act of substitution. The substitutions themselves may be written as logical rules or identities, for example, $\bar{\bar{x}}—x$ (which means "double negation may be removed"), or

$$\left.\begin{array}{l}(\forall x)\,[(A_1 \vee x)\,\&\,(A_2 \vee x)] — A_1\,\&\,A_2 \\ (\exists x)\,[(A_1 \vee x)\,\&\,(A_2 \vee x)] — A_1 \vee A_2\end{array}\right\} \text{ the exclusion rule, and so on.}$$

By making such substitutions in a given premise (that is, in the wording representing such a premise), we can obtain many further inferences (conclusions).

Now, in propositional calculus there exist methods for deriving *all* the conclusions which follow from a given set of axioms. There also exists an algorithm for recognizing deducibility, that is, a procedure for checking whether a given statement follows from a given axiom or not. However, propositional calculus cannot encompass both these procedures, since it is unable to express the relationship

between one object and another within the confines of a single statement; this calls for predicate calculus—which can also be interpreted as an associative calculus. As a result, we arrive at a variant of associative calculus—the logical calculus with a given system of allowable substitutions. The problem of recognition of deducibility now becomes one of existence of a deductive chain from the word representing the premise to the word representing the inference, a problem which the reader will recognize as that of equivalence of words in associative calculus.

In the same sense, all derivations of formulas, and all mathematical computations and transformations, are processes of constructing deductive chains in the corresponding associative calculi. And we shall prove in the next section that arithmetic itself may also be treated as an associative calculus.

Given the universal applicability of associative calculus, it would be natural to postulate it as a general method for defining determinate data processing procedures, that is, algorithms. However, before we can advance this postulate, we must state precisely what we mean by an algorithm in a given alphabet.

12.4. ALGORITHMS IN AN ALPHABET A. MARKOV'S NORMAL ALGORITHM

By analogy with the intuitive definition of Section 12.1 ff, we could intuitively define an "algorithm in alphabet A" as follows:

Definition I. An algorithm in alphabet A is a *universally understood exact instruction* specifying a potentially realizable operation on words from A; this operation admits any word from A as the initial one, and specifies the sequence in which it is transformed into new words of this alphabet. An algorithm is *applicable* to a word P if, starting from that word and acting in accordance with this instruction, we ultimately derive a new word Q, whereupon the process comes to a halt. We then say that the algorithm processes P into Q.

For example, the following instruction satisfies our definition:

Copy a given word, beginning from the end. The word so obtained is the result. Stop.

This algorithm is an exact instruction applicable to any word.

Nevertheless, Definition I is too broad, and we shall refine the concept "algorithm in alphabet A" by means of associative calculus.

Definition II. We shall say that *an algorithm in alphabet A is a set allowable substitutions, supplemented by a universally*

*understood exact instruction which specifies the order and manner of using these allowable substitutions and the conditions at which a stop occurs.**

The following is an example of an algorithm in the sense of Definition II.

Let alphabet A contain three characters: $A = \{a, b, c\}$, and let the algorithm be defined by a set of substitutions

$$\left.\begin{array}{l} cb - cc, \\ cca - ab, \\ ab - bca \end{array}\right\}$$

and the following instructions regarding the use of these substitutions:

Starting from any word P, one scans the above set of substitutions, in the order given, seeking the first formula whose left-hand part occurs in P. If there is no such formula, the procedure comes to a halt. Otherwise, one substitutes the *right-hand part* of the *first* such formula for the *first occurrence* of its left-hand part in P; this yields a new word P_1 of alphabet A. After this, the new word P_1 is used as starting one (P in the above), and the procedure is repeated. It comes to halt upon generation of a word P_n which does not contain any of the left-hand parts of the allowable substitutions.

This set of substitutions and the instructions for its use define an algorithm in alphabet A which processes the word *babaac* into the word *bbcaaac* by means of the third substitution, at which point the procedure comes to a halt. Similarly the word *cbacacb* may be successively transformed into words *ccacacb*, *ccacacc*, *abcacc*, and *bcacacc*, at which point the procedure again comes to an end. However, the word *bcacabc* generates the recurring sequence *bcacabc*, *bcacbcac*, *bcacccac*, *bcacabc*, and so on, where no stop can occur; therefore our algorithm is not applicable to the word *bcacabc*.

This algorithm is somewhat reminiscent of the following instructions for motion in an infinite labyrinth: having arrived into a room, go to the first corridor on your right, and so on. Here, a stop will occur when a dead end is reached and, as in the algorithm, there are three possibilities: starting from any room, we can either enter a dead end corridor (compare the case of word *babaac*), or move in a loop *ad infinitum* (compare the case of word *bcacabc*), or keep going for an infinitely long time without getting trapped in a loop.

*Since an alphabet and a system of allowable substitutions define an associative calculus which, as we know, can be placed into correspondence with an infinite labyrinth, that part of the definition which relates to instructions for using the substitutions may be treated as exact instructions for moving in an infinite labyrinth.

At first glance one may conclude that Definition II is narrower than Definition I. It turns out, however, that this is not so, since for any known algorithm defined in sense I we may construct an equivalent algorithm in sense II. This, of course, does not prove that Definitions I and II are equally strong; there can be no such proof, in view of the vagueness of both definitions (for instance, both contain the undefined phrase "universally understood exact instructions"). Still, Definition II is a substantial step forward, as we shall see below.

Now let us define *equivalence of algorithms*: two algorithms A_1 and A_2 in some alphabet are equivalent if their ranges of application coincide and if they process any word from their common range of application into the same result. In other words, if algorithm A_1 is applicable to a word P, then A_2 must also be applicable to that word, and conversely; also, both algorithms must transform the word P into the same word Q. If, however, one of the algorithms is not applicable to a word B, then the other algorithm must also be inapplicable.

At this point, Definition II may be transformed into an exact mathematical definition of an algorithm by a single step first proposed by A.A. Markov. His *normal* algorithm is identical to that of Definition II except that the "universally understood" instructions are replaced by a standard, once and for all fixed, and exactly specified procedure for the use of substitutions. This normal algorithm is specified as follows: To start with, the alphabet A is defined and the set of allowable substitutions is fixed. Then some word P in A is selected, and the substitution formulas are scanned (in the order given in the set) to find a formula whose left-hand part occurs in P. If there is no such formula, the procedure comes to a halt. Otherwise the right-hand member of the first of such formulas is substituted for the first occurrence of its left-hand member in P. This yields a new word P_1 in alphabet A. After this one proceeds to the second step, which differs from the first one only in that P_1 now acts as P. Then one goes to the third analogous step, and so on, until the process comes to a halt. However, the process can be terminated in only two ways: (1) when it generates a word P_n such that none of the left-hand parts of the formulas of the substitution set occurs in it; and (2) when the word P_n is generated by the last formula of the set.

We see that the algorithm of Definition II is an "almost normal" algorithm, the only difference being that it comes to a halt in only one case (when none of the allowable substitutions is applicable), whereas in the normal algorithm there are two possible causes for a "stop" instruction.

Two normal algorithms differ only in their alphabets and their set of allowable substitutions. Again, to define a normal algorithm it is sufficient to define its alphabet and its set of substitutions.

Examples of Normal Algorithms.

Let the alphabet A and the set of allowable substitutions be

$$A = \{1, +\} \qquad \begin{array}{l} 1 + \to + 1 \\ + 1 \to 1 \\ 1 \to 1 \end{array}$$

(the arrows are a convention denoting a Markov normal algorithm, to differentiate it from the usual associative calculus).

Now let us see how this algorithm transforms the word 1 1 1 1 + + 1 1 + 1 1 1. We obtain successively the words:

1 1 1 1 + 1 1 + 1 1 1
1 1 1 + 1 1 1 + 1 1 1
1 1 + 1 1 1 1 + 1 1 1
1 + 1 1 1 1 1 + 1 1 1
+ 1 1 1 1 1 1 + 1 1 1
+ 1 1 1 1 1 + 1 1 1 1
+ 1 1 1 1 + 1 1 1 1 1
+ 1 1 1 + 1 1 1 1 1 1
+ 1 1 + 1 1 1 1 1 1 1
+ 1 + 1 1 1 1 1 1 1 1
+ + 1 1 1 1 1 1 1 1 1
+ 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1

The procedure comes to a halt on the use of the last substitution $1 \to 1$, which processes the word 111111111 into itself.

Now let the set of substitutions (in the same alphabet) be

$$+ \longrightarrow \lambda$$
$$1 \longrightarrow 1$$

(Λ is again an empty word). Then 11 + 111 + 1 + 11 will be transformed as follows:

1 1 + 1 1 1 + 1 + 1 1
1 1 1 1 1 + 1 + 1 1
1 1 1 1 1 1 + 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1.

We see that both algorithms produce a sum of symbols 1; that is, they perform addition, and it not difficult to show that they are equivalent.

A third normal algorithm, equivalent to the above, is defined by a set

$$1+\to+1,$$
$$++\to+,$$
$$+\to\Lambda.$$

The reader is invited to verify that the normal algorithm $A=\{1, *, \vee, ?\}$

$$*11\to\vee*1,$$
$$*1\to\vee,$$
$$1\vee\to\vee1?,$$
$$?\vee\to\vee?,$$
$$?1\to1?,$$
$$\vee1\to\vee,$$
$$\vee?\to?,$$
$$?\to1,$$
$$1\to1$$

transform each word of the form $\underbrace{1111\ldots11}_{m\text{ times}}*\underbrace{111\ldots111}_{n\text{ times}}$ into the word $\underbrace{111\ldots111}_{m\cdot n\text{ times}}$; that is, it performs a multiplication.

Markov also refined the concept of an algorithm in an alphabet by postulating that *each algorithm in an alphabet is equivalent to some normal algorithm in the same alphabet*. This is a hypothesis which cannot be rigorously proved, since it contains both the vague statement "each algorithm" and the exact concept of a "normal algorithm." This statement may be regarded as a law which has not been proved but which has been confirmed by all accumulated experience. It is supported by the fact that no one has so far succeeded in formulating an algorithm for which there is no normal algorithm equivalent to it (in the same alphabet).

Now we can return to the problem of universal definition of algorithms (see end of Section 12.3). In view of what we said above, the Markov normal algorithm seems a convenient "standard form" for defining *any* algorithm, that is, we assume that *any* algorithm may be defined as a Markov normal algorithm. This is, of course, no more than a hypothesis and, at that, much less well-founded than the Markov hypothesis discussed above, since it cannot even be expressed in exact terms. However, its intuitive meaning is obvious.

As soon as we accept this hypothesis, we have a way of rigorously proving the algorithmic unsolvability of generalized problems. For example, we can prove the algorithmic unsolvability of the word problem; that is, we can prove that there is no algorithm applicable to all associative calculi and capable of determining whether two words P and Q are equivalent. All we have to do in order to prove this is to demonstrate the existence of one associative calculus in which there is no normal algorithm for recognizing the equivalence of words. Examples of such calculi were first given by A.A. Markov (1946) and E. Post (1947). After that it became clear that *a fortiori* there can be no algorithms capable of recognizing the equivalence of words in all associative calculi.

The examples of Markov and Post were unwieldy and comprised hundreds of allowable substitutions. Later, G. S. Tseytin exhibited an associative calculus containing only seven allowable substitutions, in which the problem of word equivalence was also algorithmically unsolvable.

As an illustration we shall show one proof of algorithmic unsolvability. Let U be a normal algorithm defined in an alphabet $A = \{a_1, a_2, \ldots, a_n\}$ with the aid of a set of substitutions. In addition to the characters of alphabet A, this algorithm also uses the symbols $\rightarrow$ and ,. By assigning to these symbols new characters a_{n+1} and a_{n+2}, we can represent U by a *word* in an expanded alphabet $\tilde{A} = \{a_1, a_2, \ldots, a_{n+2}\}$. Let us now apply U to the word representing it. If algorithm U transforms this word into another one, after which there is a stop, this means that U is applicable to its own representation—the algorithm is *self-applicable*. Otherwise, the algorithm is *nonself-applicable*. Now there arises the problem of recognition of self-applicability, that is, finding out from the representation of a given algorithm whether it is self-applicable or not.

This problem would be solved by a normal algorithm V, which, upon application to any representation of a self-applicable algorithm U, would transform that representation into a word M and which would transform all representations of a nonself-applicable algorithm U into another word L. Thus the application of the recognition algorithm V would show whether U is self-applicable or not.

However, it has been proved that such a normal algorithm V does not exist (see [64]), which also proves that the problem of recognition of self-applicability is algorithmically unsolvable. The proof, by *reductio ad absurdum*, is as follows. Suppose that we do have a normal algorithm V and that it transforms each representation of a self-applicable algorithm into M and each representation of a nonself-applicable algorithm into L. Then, by some modification of the

substitution system of the algorithm V, we may devise another algorithm $\tilde{V}$ which would again transform each representation of a nonself-applicable algorithm into L but which would be inapplicable to representations of a self-applicable algorithm (because the algorithm does not come to a stop). Such an algorithm $\tilde{V}$ leads to contradictions. Indeed,

1. Suppose $\tilde{V}$ is self-applicable (there is a stop), that is, it can be applied to its own representation (which is in the form of a word). But this simply means that $\tilde{V}$ is nonself-applicable.

2. Suppose $\tilde{V}$ is nonself-applicable. Then it can be applied to its own representation (since it is applicable to any representation of a nonself-applicable algorithm). But this simply means that $\tilde{V}$ is self-applicable.

The resulting contradiction proves the algorithmic unsolvability of the problem of recognition of self-applicability.

Thus there is no *a priori* proof for the existence or nonexistence of an algorithm for a given problem. But the nonexistence of an algorithm for a class of problems merely means that this class is so broad that there is no single effective method for the solution of all the problems contained in it. Thus, even though the generalized problem of recognition of word equivalence is algorithmically unsolvable in Tseytin's associative calculus, under normal conditions we can still find a way for proving the equivalence or nonequivalence of a specific pair of words.

There is an interesting history to the problem of algorithmic unsolvability. Prior to Markov's refinement of the concept of an algorithm, mathematicians held one or the other of the following points of view:

1. Problems for which there is no algorithm are still, in principle, algorithmically solvable; the desired algorithm is unavailable simply because the existing mathematical machinery is unequal to the task of devising this algorithm. In other words, our knowledge is insufficient to solve problems we call algorithmically unsolvable, but such algorithms will be found in the future.

2. There are classes of problems for which there are no algorithms. In other words, there are problems that cannot be solved mechanically by means of reasoning and computations and that require creative thinking.

This is a very strong statement because it says to all future mathematicians: Whatever the means at your disposal may be, do not waste your time searching for nonexistent algorithms!

But how does one *prove* the nonexistence of an algorithm? So long as the definition of an algorithm comprised the phrase

"universally understood instruction" such a proof was unthinkable, because one cannot conceive of all possible "universally understood instructions" and prove that none of these is applicable.

Thus the very survival of this second viewpoint is related to the daring hypotheses on the existence of "standard forms" for defining an algorithm (such as the Markov normal algorithm), that is, hypotheses permitting the formulation of the concepts of "algorithm" and "algorithmically unsolvable problem" in exact terms.

12.5. REDUCTION OF ANY ALGORITHM TO A NUMERICAL ALGORITHM. GÖDELIZATION

The advent of computers has prompted much work in the theory of numerical algorithms with which these machines operate. In the course of this work it has been shown that any logical algorithm can be reduced to a numerical algorithm. As the methods for doing this improved, it also became clear that *all* algorithms can be reduced to numerical ones, and thus the theory of numerical algorithms (which we shall also call the theory of computable functions) became a generalized mechanism for study of all algorithmic problems.

We shall now show how any algorithmic problem can be reduced to a computation of values of an integer-valued function of integer arguments.

Assume some algorithm is applicable to a range of data. We shall represent each set of data comprised in this range by means of a *unique* nonnegative integer A_n; when we have done this, we have, instead of the original data, or collection of numerals (labels) $A_0, A_1, A_2, \ldots, A_n, \ldots$ representing these data.

Similarly, we assign a unique numeral to each of the possible solutions derived with our algorithm from the above data and thus obtain a sequence of numerals (labels) $B_0, B_1, B_2, \ldots, B_m, \ldots$ representing these solutions.

Now this labeling or *numbering* permits us to dispense with the data and solutions themselves and to operate instead on *numerals representing* these quantities. For it is fairly obvious that if we have an algorithm processing a set of data into a solution, we can also devise an algorithm processing the *numeral* A_n *denoting* these data into *the numeral* B_m representing the corresponding solution.

It is also obvious that this algorithm must be a numerical one, of the type $m = \varphi(n)$.

In general, if there exists an algorithm for solving any given problem (that is, transforming a set of data into a solution), then

there must also exist an algorithm for computing the values of the corresponding function $m = \varphi(n)$. Indeed, to find the value of $\varphi(n)$ at $n = n^*$, one can reconstitute the set of data represented by n^* from a table of values of n vs. these data; then one can employ the (existing) algorithm to find a solution for the problem. Having the value of the solution, one can go to a table of values of solutions vs. m to find the numeral m^* representing m. Consequently,

$$\varphi(n^*) = m^*.$$

Conversely, if there exists an algorithm for conputing the values of $\varphi(n)$, there must be an algorithm for solving the given problem. Indeed, one can find from the table of the data vs. n the numerical n^* representing n. Then one can compute $m^* = \varphi(n^*)$; having m^*, one can determine the value of the actual solution from another table.

Let us now present a widely used method of *numbering* (that is, unique labeling), namely, the method of Gödel. Suppose we have a number n. Then, by virtue of the fact that any composite integer can be uniquely decomposed into prime factors, we have

$$n = 2^{a_1} \cdot 3^{a_2} \cdot 5^{a_3} \cdot 7^{a_4} \cdot \ldots \cdot p_{m-1}^{a_m},$$

where $p_0 = 2$, $p_1 = 3$, $p_2 = 5$, and, in general, p_m is the mth prime number. Thus n — the Gödel number — is the product of successive primes, which are raised to powers from the set $a_1, a_2, \ldots, a_m$. Any Gödel number n is uniquely related to a specific set $a_1, a_2, \ldots, a_m$, and, conversely, each set $a_1, a_2, \ldots, a_m$ is uniquely associated with a specific number n. For example, if $n = 60$, we have: $60 = 2^2 3^1 5^1$, that is, $a_1 = 2$, $a_2 = 1$, $a_3 = 1$.

Now, Gödel numbering (or gödelization) allows us to uniquely label any sequence of m members. Consider a few examples:

1. Any pair of numbers a_1 and a_2, for which we seek the greatest common divisor q, can be assigned a unique Gödel number $n = 2^{a_1} \cdot 3^{a_2}$. Now, Euclid's algorithm reduces to the computation of $q = \varphi(n)$.

2. We want to find the sumbol in the rth position of a purely periodic sequence produced by endless repetition of the numerical sequence $a_1, a_2, \ldots, a_m$. The statement of the problem can then be associated with the Gödel number

$$n = 2^r \cdot 3^{a_1} \cdot 5^{a_2} \cdot \ldots \cdot p_m^{a_m}.$$

The algorithm for finding the rth symbol then reduces to computation

of values of the function

$$q = \varphi(n),$$

where q may assume values only from the set $\{a_1, a_2, \ldots, a_m\}$.

3. An nth-degree equation

$$x^n + b_1 x^{n-1} + b_2 x^{n-2} + \ldots + b_n = 0$$

(where b_i are general symbols, not specific coefficients) can be assigned a number n; it is obvious that, knowing n, one can easily reconstruct the original equation.

When $n = 2$, the equation is

$$x^2 + b_1 x + b_2 = 0.$$

Its solution may be expressed in terms of coefficients b:

$$x = -\frac{b_1}{2} \pm \sqrt[2]{\left(\frac{b_1}{2}\right)^2 - b_2}. \qquad (12.2)$$

Let us rewrite Eq. (12.2) on one line

$$x = -b_1 : 2 + -\sqrt[2]{(b_1 \times b_1 : 4 - b_2)},$$

where the square root sign applies to the entire expression in parentheses. Assume that we intend to find an expression for the solution of the nth-degree equation in terms of the radical signs. It is obvious that, whatever the form of the solution, it may consist only of the following symbols:

$$+, -, \times, :, (,), 1, b_1, b_2, \ldots, b_n, \sqrt[2]{}, \sqrt[3]{}, \ldots, \sqrt[r]{}.$$

Also, we can use symbol 1 and the addition sign + to express any number which may be present as the sum $1 + 1 + 1 + \ldots + 1$. Let us code the above symbols by means of the following numerals:

$\sqrt[r]{}$	is assigned the number	$2r$	)	is assigned the number		13
$+$	»	» 3	1	»	»	15
$-$	»	» 5	b_1	»	»	17
$\times$	»	» 7	b_2	»	»	19
$:$	»	» 9	. . . . .			
$($	»	» 11	. . . . .			
			b_n	»	»	$(2n+15)$.

Then each expression consisting of these symbols is uniquely represented by a set of numerals. For example, the set

$$6, 11, 17, 3, 19, 13$$

corresponds to the expression

$$\sqrt[3]{(b_1 + b_2)}.$$

This set of numerals, as we already know, describes a Gödel number equal to

$$2^6 \cdot 3^{11} \cdot 5^{17} \cdot 7^3 \cdot 11^{19} \cdot 13^{13}.$$

Conversely, given a Gödel number, we can always reconstitute the corresponding set of numerals; each numeral can then be replaced by the symbol for which it stands. Thus, Gödel numbering permits us to code in a unique fashion any formula, any expression, whether it be composed of numbers or letters, signs denoting operations, or any combination of those above.

4. Assume we want to number all possible words which can be written in some alphabet A. This is easily done by matching a numeral with each character of the alphabet. Then each word will become a sequence of numerals, and we can obtain the Gödel number corresponding to each such word. And, if desired, we can also number all the sequences of such words (for example, all the deductive chains) to obtain Gödel numbers for the sequence of Gödel numbers of the individual words, as well as a Gödel number for the entire collection of such sequences.

We have now seen that the gödelization procedure reduces not only arithmetical algorithms but also any normal Markov algorithm to a computation of values of some integer-valued function. Therefore, the algorithm for such a computation is the universal algorithmic form we have sought from the beginning.

In concluding, we must point out that all of the above discussion was based on the assumption that, even though the set of data for a problem which can be processed by a given algorithm may be infinitely large, it is, nevertheless, countable. Our subsequent discussion of algorithms will also assume a countable set of conditions.

12.6. ELEMENTARY AND PRIMITIVE RECURSIVE FUNCTIONS

Functions such as $y = \varphi(x_1, x_2, \ldots, x_n)$ are called *arithmetical* if both the arguments and the functions themselves may assume values

only from the set $\{0, 1, 2, \ldots\}$. From now on, we shall discuss only arithmetical functions. Logical functions (Chapter 1) are a special case of arithmetical functions.

We shall also introduce the following system of notation:

Variables will be denoted by lower-case Latin letters:

$$a,\ b,\ c,\ \ldots,\ m,\ n,\ \ldots,\ x,\ y,\ z \quad \text{or} \quad x_1,\ x_2\ , \text{etc.}$$

Functions will be denoted by lower-case Greek letters:

$$\varphi,\ \psi,\ \chi,\ \xi,\ \ldots,\ \alpha,\ \beta,\ \gamma \quad \text{or} \quad \varphi_1,\ \varphi_2,\ \varphi_3\ , \text{etc.}$$

Predictions will be denoted by Latin capitals:

$$A,\ B,\ P,\ Q,\ R,\ S\ , \text{etc.}$$

Specific numbers or constants will also be denoted by lower-case Latin letters but will carry an asterisk:

$$a^*,\ b^*,\ x^*,\ y^*, \text{etc.}$$

We shall now define a computable arithmetical function and a solvable predicate.

A function $y = \varphi(x_1, x_2, \ldots, x_n)$ *is said to be algorithmically computable (*or just *computable) if there exists an algorithm for finding the value of this function at all values of variables* $x_1,\ x_2,\ \ldots\ x_n$.

A predicate $P(x_1, x_2, \ldots, x_n)$ *defined on the set of integers is said to be algorithmically solvable (*or just *solvable) if there exists an algorithm for finding the value of this predicate at all values of variables* $x_1, x_2, \ldots, x_n$.

These definitions are intuitive and inexact, since we have not yet defined a computing algorithm. In order to refine them, we shall have to develop the class of computable functions, starting with the most elementary computable functions.

We shall call *elementary* those arithmetical functions which can be obtained from nonnegative integers and variables by means of a finite number of additions, arithmetical subtractions (by which we mean obtaining $|x - y|$), multiplications, arithmetical divisions (by which we mean deriving the integer part of the quotient $\left[\frac{a}{b}\right]$ for $b \neq 0$), as well as from constructions involving sums and products. The computability of elementary functions is undisputable since there are algorithms for all the separate operations involved in such functions, and thus the aggregate function must also be algorithmically computable.

To construct elementary functions we need only one number, namely, 1, since

$$0 = |1 - 1|, \quad 2 = 1 + 1, \quad 3 = (1 + 1) + 1, \text{ etc.}$$

Now let us see what functions are elementary.

1. All the simple functions such as

$$\varphi(x) \equiv x + 1, \quad \varphi(y) \equiv 12y, \quad \psi(a, b, c) \equiv ab + c, \quad \chi(b) \equiv b^2$$

(since $b^2 = b \cdot b$), etc., are elementary.

2. Many of the frequently employed functions of number theory are elementary. For example:

a) $\min(x, y) = \left[\frac{|(x+y) - |x-y||}{2}\right]$;

b) $\operatorname{sg}(x) = \begin{cases} 1 & \text{when } x \geqslant 1, \\ 0 & \text{when } x = 0. \end{cases}$

Function $\operatorname{sg}(x)$ may be expressed by means of function $\min(x, y)$:

$$\operatorname{sg}(x) = \min(x, 1).*$$

From $\operatorname{sg}(x)$ one can obtain $\overline{\operatorname{sg}}(x)$:

$$\overline{\operatorname{sg}}(x) = |1 - \operatorname{sg}(x)| = \begin{cases} 1 & \text{when } x = 0, \\ 0 & \text{when } x \geqslant 1. \end{cases}$$

3. The inequality $x \leqslant y$ is equivalent to the $\min(x, y) = x$ or $|\min(x, y) - x| = 0$. Then the predicate "x is smaller than or equal to y" may be written as

$$P(x, y) = \overline{\operatorname{sg}}(|\min(x, y) - x|).$$

Indeed, if $x^* \leqslant y^*$, then $P(x^*, y^*) = 1$, that is, it is a true statement; otherwise $P = 0$.

4. Later we shall use the function

$$y \dot{-} x \begin{cases} |y - x|, & \text{if } y \geqslant x, \\ 0, & \text{if } y < x. \end{cases}$$

This is also an elementary function since it can be expressed as

$$y \dot{-} x = |y - x| \overline{\operatorname{sg}}(|\min(x, y) - x|).$$

5. The residue obtained upon division of a by n

$$\operatorname{res}(a, n) = \left|a - n\left[\frac{a}{n}\right]\right|$$

is again an elementary function.

Now, is the class of all computable functions broader than that of elementary functions? In other words, is there a computable function which is not elementary? To answer, let us follow some

*See Section 9.2f for notation.

elementary functions to see at what point they cease to be elementary.

Of the simple elementary functions, the one that increases the most rapidly is the product. The product an arises from adding a to itself n times; thus, multiplication is iterated addition.

Raising to a power is, in turn, iterated multiplication:

$$a^n = a \cdot a \cdot a \cdot \ldots \cdot a = \prod_1^n a.$$

This function is still elementary since it is expressed by a product. It increases very rapidly with a and n.

A still more rapidly increasing function involves the iteration of the operation of raising to a power:

$$\psi(0, a) = a, \quad \psi(1, a) = a^a, \quad \psi(2, a) = a^{(a^a)}$$

and, in general,

$$\psi(n+1, a) = a^{\psi(n, a)}. \tag{12.3}$$

Here the increase is so rapid that it becomes impossible to "keep pace" with the increase in $\psi(n, a)$ by devising elementary functions (for proof, see [77]); to be more precise, this function, starting from some $a = a^*$, majorizes all elementary functions; that is, for any elementary function $\varphi(a)$ there is always some m^* such that the inequality

$$\varphi(a) < \psi(m^*, a)$$

will be satisfied at all $a \geqslant a^*$. But if this is the case, then it is easy to show that the function $\gamma(n) = \psi(n, n)$ is not elementary.

Indeed, if $\psi(n, n)$ were an elementary function, then we could find some m^* (it may be assumed that $m^* \geqslant 2$ because the function $\psi(n, a)$ is monotonic) such that $\psi(n, n) < \psi(m^*, n)$ for $n \geqslant 2$. This inequality would, in particular, hold when $n = m^*$ since $m^* \geqslant 2$. We then get $\psi(m^*, m^*) < \psi(m^*, m^*)$, which is impossible.

Thus, iteration of the operation of raising to a power gives a nonelementary function. But, at the same time, $\psi(n, a)$ is *a priori* known to be computable. Indeed, suppose we want to compute $\psi(n, a)$ for any $n = n^*$, $a = a^*$. For $a = a^*$, Eq. (12.3) becomes

$$\psi(n+1, a^*) = (a^*)^{\psi(n, a^*)}.$$

Let us denote $(a^*)^m = \chi(m)$; here $\chi(m)$ is an elementary single-valued computable function, whose computation algorithm merely consists

of multiplication by a^*, repeated m times. The formula

$$\psi(n+1,\ a^*)=\chi(\psi(n,\ a^*)) \tag{12.4}$$

relates the value of ψ at a point with its value at the preceding point. Now it is sufficient to specify the initial value $\psi(0,\ a^*)=a^*$ in order to obtain a computing procedure, which successively yields

$$\begin{aligned}
&\psi(1,\ a^*)=\chi(\psi(0,\ a^*))=\chi(a^*),\\
&\psi(2,\ a^*)=\chi(\psi(1,\ a^*))=\chi(\chi(a^*)),\\
&\psi(3,\ a^*)=\chi(\psi(2,\ a^*))=\chi(\chi(\chi(a^*))),\\
&\cdots\cdots\cdots\cdots\cdots\cdots
\end{aligned}$$

This process is continued until the value of $\psi(n^*,\ a^*)$ is obtained. It is obvious that this method specifies ψ at all points and does so uniquely, since the computation of its values reduces to the computation of $\chi(m)$, which is a determinate and unique function at all points.

Our previous example has shown that computable functions need not be elementary. To continue our delineation of the class of computable functions, let us examine the method of defining $\psi(n, a)$. This function was defined by induction: that is, we were given the initial value of the function, namely, $\psi(0, a)$, and were told by what allowable operations the successive values of this function are derived from their predecessors. Now we shall use this induction method for specifying all computable functions. But first we must refine and broaden this method.

Derivation by induction can, generally speaking, be used with any ordered set in which the concepts of "predecessor" and "successor" are meaningful. Let us denote by x' the *successor function* which describes the transition to the next member of the given set. We shall assume that our given set is always $\{0, 1, 2, \ldots\}$, so that $0'=1$; $1'=2$; $2'=3$, or, in general, $x'\equiv x+1$.

The generalized procedure for defining a function $\varphi(x)$ can now be precisely defined as follows:

1. Give the value of $\varphi(0)$.

2. Specify the manner of expressing $\varphi(x')$ in terms of x and $\varphi(x)$ at any x:

$$\left.\begin{aligned}
&\varphi(0)=q,\\
&\varphi(x')=\chi(x,\ \varphi(x)).
\end{aligned}\right\} \tag{12.5}$$

In a more general case, there may also occur parameters $x_2, x_3, \ldots, x_n$, which remain unaltered in the induction process. Then the defining equation (12.5) is modified to

$$\left.\begin{aligned}\varphi(0, x_2, x_3, \ldots, x_n) &= \psi(x_2, x_3, \ldots, x_n),\\ \varphi(y', x_2, x_3, \ldots, x_n) &= \chi(y, \varphi(y, x_2, x_3, \ldots, x_n), x_2, x_3, \ldots, x_n).\end{aligned}\right\} \quad (12.6)$$

If ψ and χ are known and computable functions, then the scheme of Eq. (12.6) may be used to develop a computation procedure, which will give, consecutively, $\varphi(1, x_2^*, \ldots, x_n^*)$, $\varphi(2, x_2^*, \ldots, x_n^*)$, and so on. Consequently, this scheme does indeed specify a computable function.

Let us now see which arithmetical functions can be derived by induction and how broad is the class of such functions. To state the problem exactly, we must specify which function we consider initially (that is, *a priori*) known and which operations (in addition to the above-described induction procedure) are allowable in the derivation of subsequent functions.

We shall consider the following functions as initially known (fundamental) or *primitive*:

I. $\varphi(x) = x'$, the *successor function* described above, applied to the set consisting of 0 and all natural numbers. Its abbreviated notation is S.

II. $\varphi(x_1, x_2, \ldots, x_n) = q$, where $q = \text{const}$, a *constant function*. It is denoted by C_q^n.

III. $\varphi(x_1, x_2, \ldots, x_n) = x_i$, the *identity function*.* It is denoted by U_i^n.

In addition to the induction procedure [Eqs. (12.5) or (12.6)], we shall include the *substitution procedure* IV among the allowable operations.

IV. $\varphi(x_1, x_2, \ldots, x_n) = \psi(\chi_1(x_1, x_2, \ldots, x_n), \chi_2(x_1, x_2, \ldots x_n), \ldots, \chi_m(x_1, x_2, \ldots, x_n))$.

Let us now write out all the allowable operations into one column:

$$\begin{aligned}
&\text{I. } \varphi(x) = S(x) = x'.\\
&\text{II. } \varphi(x_1, x_2, \ldots, x_n) = C_q^n = q.\\
&\text{III. } \varphi(x_1, x_2, \ldots, x_n) = U_i^n = x_i.\\
&\text{IV. } \varphi(x_1, x_2, \ldots, x_n) = \psi(\chi_1(x_1, x_2, \ldots, x_n),\\
&\qquad \chi_2(x_1, x_2, \ldots, x_n), \ldots, \chi_m(x_1, x_2, \ldots, x_n)),\\
&\text{V, a. } \begin{cases} \varphi(0) = q,\\ \varphi(x') = \chi(x, \varphi(x)).\end{cases}
\end{aligned}$$

*Rather than employ an identity function, we could consider the variables themselves originally known, as we have done in defining elementary functions. The identity function is introduced here merely for the sake of uniform exposition. Again, instead of the constant function, the null-function $\phi(x_1, x_2, \ldots, x_n) \equiv 0$ could have been used as a fundamental function, since repeated applications of the successor function then gives all the constants: $1 = 0'$, $2 = 1'$, etc.

$$\text{V, b} \quad \begin{cases} \varphi(0, x_2, \ldots, x_n) = \psi(x_2, \ldots, x_n), \\ \varphi(y', x_2, \ldots, x_n) = \chi(y, \varphi(y, x_2, \ldots \\ \qquad \ldots, x_n), x_2, x_3, \ldots, x_n). \end{cases}$$

Operations I-III specify the primitive functions and assume the role of axioms, whereas IV and V act as rules of inference.

Definition. *A function $\varphi(x_1, x_2, \ldots, x_n)$ is a primitive recursive function if it can be defined by means of finite number of applications of operations I-V.*

We shall say that a function φ *depends directly* on other functions if, for any given m and n, it satisfies operation IV for some $\psi, \chi_1, \chi_2, \ldots, \chi_m$ (in this case, φ is directly dependent on $\psi, \chi_1, \chi_2, \ldots, \chi_m$) or if, for any given q it satisfies V, a or V, b for some ψ, χ (here, φ is directly dependent on ψ and χ).

Definition. *A sequence of functions $\varphi_1, \varphi_2, \ldots, \varphi_k$ such that each function of the sequence either is primitive or depends directly on the preceding functions of the sequence while the last function φ_k is φ is called a primitive recursive description of the primitive recursive function $\varphi(x_1, x_2, \ldots, x_n)$. We shall call k the depth of the primitive recursive description of the function φ.*

A primitive recursive description is simply the series of functions obtained by successive applications of operations I-V in the definition of function φ. Indeed, we start from the initial (starting) functions (which become the beginning of our train of functions) and then proceed step by step toward the function φ.

We shall now show examples of derivation of some primitive recursive functions.

1. We define the function $\varphi(x, y)$ as:

$$\varphi(0, x) = x,$$
$$\varphi(y', x) = [\varphi(y, x)]'.$$

We thus have

$$\varphi(1, x) = x' = x + 1,$$
$$\varphi(2, x) = [\varphi(1, x)]' = (x + 1)' = x + 1 + 1 = x + 2,$$
$$\varphi(3, x) = [\varphi(2, x)]' = (x + 2)' = x + 2 + 1 = x + 3,$$

or, in general,

$$\varphi(y, x) = x + y.$$

To get a primitive recursive description of this function, write out in full the operation V, b as applied to $\varphi(y, x) = x + y$:

$$\left.\begin{aligned} \varphi(0, x) &= \psi(x), \\ \varphi(y', x) &= \chi(y, \varphi(y, x), x). \end{aligned}\right\} \tag{12.7}$$

Here, $\psi(x)$ assumes the form $\psi(x) \equiv x$, that is, it is the original identity function $\psi(x) \equiv U_1^1(x)$.

The function $\chi(y, z, x) \equiv z'$ can be obtained from the initial function $U_2^3(y, z, x) \equiv z$ by means of operation IV, where the successor function $S(z) \equiv z'$ is taken as ψ. One can, therefore, write

$$\chi(y, z, x) = S[U_2^3(y, z, x)].$$

One primitive recursive description of the function χ will be the sequence U_2^3, S, χ. Adding to it the function $\psi(x) \equiv U_1^1(x)$, on which $\varphi(y, x)$ depends directly in accordance with Eq. (12.7), we get the primitive recursive description of $\varphi(y, x)$:

$$U_2^3, S, \chi, U_1^1, \varphi.$$

2. In order to define the next primitive recursive function, we shall use the fact that the sum $x + y$ has already been defined as a primitive recursive function. We set

$$\varphi(0, x) = 0,$$
$$\varphi(y', x) = \varphi(y, x) + x.$$

Then, we obtain in succession

$$\varphi(1, x) = \varphi(0, x) + x = 0 + x = x,$$
$$\varphi(2, x) = \varphi(1, x) + x = x + x = 2x,$$
$$\varphi(3, x) = \varphi(2, x) + x = 2x + x = 3x$$
$$\cdots\cdots\cdots\cdots\cdots\cdots$$

or, in general,

$$\varphi(y, x) = yx.$$

Consequently, the product is also a primitive recursive function.

3. Using the result of Example 2, let us define

$$\varphi(0, x) = 1,$$
$$\varphi(y', x) = \varphi(y, x)\, x.$$

It is easily shown that this function means raising to a power: $\varphi(y, x) = x^y$.

4. $\varphi(0) = 1$. $\varphi(x') = \varphi(x)x'$. It can be seen easily that $\varphi(x) = x!$.

5. The function "predecessor of x"

$$\mathrm{pd}(x) = \begin{cases} 0 & \text{if} \quad x = 0, \\ |x - 1| & \text{if} \quad x > 0 \end{cases}$$

is a primitive recursive function since it is defined by operation V, a

$$\mathrm{pd}(0) = 0$$
$$\mathrm{pd}(x') = x.$$

6. The previously encountered function $x \dot{-} y$ is defined as

$$x \dot{-} 0 = x,$$
$$x \dot{-} y' = \mathrm{pd}(x \dot{-} y).$$

7. The function $\min(x, y)$ can now be defined by using operation IV:

$$\min(x, y) = y \dot{-} (y \dot{-} x).$$

8. $\max(x, y) = (x + y) \dot{-} \min(x, y)$.
9. $\mathrm{sg}(x) = \min(x, 1)$.
10. $\overline{\mathrm{sg}}(x) = 1 \dot{-} x$.
11. $|x - y| = (x \dot{-} y) + (y \dot{-} x)$.
12. The remainder obtained upon division of y by x [this function is denoted by $\mathrm{res}(y, x)$] is defined as

$$\mathrm{res}(0, x) = 0,$$
$$\mathrm{res}(y', x) = (\mathrm{res}(y, x))' \cdot \mathrm{sg}|x - (\mathrm{res}(y, x))'|.$$

13. $\left[\frac{y}{x}\right]$ is defined as

$$\left[\frac{0}{x}\right] = 0,$$
$$\left[\frac{y'}{x}\right] = \left[\frac{y}{x}\right] + \overline{\mathrm{sg}}|x - (\mathrm{res}(y, x))'|.$$

14. Primitive recursion may be used to define finite sums and products such as

$$\sum_{i=0}^{y} \varphi(i, x) \quad \text{and} \quad \prod_{i=0}^{y} \varphi(i, x).$$

Indeed,

$$\left.\begin{aligned} \sum_{i=0}^{0} \varphi(i, x) &= \varphi(0, x), \\ \sum_{i=0}^{y'} \varphi(i, x) &= \sum_{i=0}^{y} \varphi(i, x) + \varphi(y', x) \end{aligned}\right\}$$
$$\left.\begin{aligned} \prod_{i=0}^{0} \varphi(i, x) &= \varphi(0, x), \\ \prod_{i=0}^{y} \varphi(i, x) &= \prod_{i=0}^{y} \varphi(i, x) \cdot \varphi(y', x) \end{aligned}\right\}.$$

Among the primitive recursive functions just defined we find the sum $x + y$, the absolute difference $|x - y|$, the product xy, the quotient $\left[\frac{y}{x}\right]$, as well as finite sums and products. Consequently, all the elementary functions discussed at the beginning of this section are

primitive recursive functions—which is the same as saying that elementary functions are a subclass of primitive recursive functions.

12.7. PREDICATES. MINIMALIZATION

In logic, predicates are introduced whenever it is necessary to represent symbolically a relationship between several objects (see Chapter 1). In general, a predicate is defined on a set (finite or infinite) of objects and may assume two values: true or false (1 or 0). However, we shall discuss only arithmetic predicates defined on the set $\{0, 1, 2, \ldots\}$.

The predicate $P(x_1, x_2, \ldots, x_n)$ depends on n variables (it is an n-place predicate). The variables appearing under the quantifier signs in front of the predicate are *bound*; the other variables are *free*. For example, the predicate $P(x, y, z, t)$ is dependent on four variables. In $(\forall x)(\exists y)P(x, y, z, t)$,* however, x and y are bound and the predicate depends on the free variables z and t. For this reason, the expression $(\forall x)(\exists y)P(x, y, z, t)$ really represents the predicate $Q(z, t)$.

$$(\forall x)(\exists y)P(x, y, z, t) \equiv Q(z, t).$$

Indeed, this notation means the following: depending on the z^* and t^* values, there may exist for all x a y such that $P(x, y, z^*, t^*)$; or this may not be the case. In the first case Q is true (or $Q = 1$), and in the second, it is false (or $Q = 0$).

Just as a function, a predicate may be specified by induction. For example, the operation

$$E(0) \quad (\text{or} \quad E(0) = 1),$$
$$E(a') = \bar{E}(a)$$

defines the predicate $E(a) \equiv$ "a is even." By analogy with the derivation of primitive recursive functions, this points to a procedure for deriving predicates and to the concept of a "primitive recursive predicate." However, we shall not follow this path. Instead, we shall show another definition of a primitive recursive predicate—that proposed by Gödel in 1931. To start with, we define a representative function of a predicate $P(x_1, x_2, \ldots, x_n)$ as a function $\varphi(x_1, x_2, \ldots, x_n)$ which vanishes at those $x_1, x_2, \ldots, x_n$ for which

*It is read as: "for all x there exists a y such that $P(x, y, z, t)$ is true." The phase "is true" is often omitted.

$P(x_1, x_2, \ldots, x_n)$ is true and only at those. Then the assertion that $P(x_1, x_2, \ldots, x_n)$ is true may be expressed

$$\varphi(x_1, x_2, \ldots, x_n) = 0.$$

Obviously, a single predicate may have several representative functions, the zeros of which coincide.

Definition. A predicate is primitive recursive if there exists a primitive recursive function representing that predicate.

Let us assume that the predicate $Q(x_1, x_2, \ldots, x_n)$ is defined by a primitive recursive predicate $P(x_1, x_2, \ldots, x_n, y)$, using a bounded universal quantifier

$$Q(x_1, x_2, \ldots, x_n) = (\forall y)_{y \leqslant z} P(x_1, x_2, \ldots, x_n, y) \tag{12.8}$$

or, more explicitly,

$$Q(x_1, x_2, \ldots, x_n) = (\forall y)[y \leqslant z \to P(x_1, x_2, \ldots, x_n, y)].$$

The predicate $Q(x_1, x_2, \ldots, x_n)$ corresponds to the statement that, given $x_1, x_2, \ldots, x_n$, the predicate $P(x_1, x_2, \ldots, x_n, y)$ is true for all $y \leqslant z$. The predicate $Q(x_1, x_2, \ldots, x_n)$ so defined is primitive recursive since its representative function $\varphi(x_1, x_2, \ldots, x_n)$ can be expressed in terms of the representative function $\varphi(x_1, x_2, \ldots, x_n, y)$ of predicate P:

$$\varphi(x_1, x_2, \ldots, x_n) = \sum_{y=0}^{z} \varphi(x_1, x_2, \ldots, x_n, y).$$

Let us note there that z may also depend on $x_1, x_2, \ldots, x_n$; if all the inferences are to remain valid, this dependence must also be primitive recursive.

An analogous conclusion may be drawn with respect to a predicate Q defined by using a bounded existential quantifier

$$Q(x_1, x_2, \ldots, x_n) = (\exists y)_{y \leqslant z} P(x_1, x_2, \ldots, x_n, y) \tag{12.9}$$

or, more explicitly,

$$Q(x_1, x_2, \ldots, x_n) = (\exists y)[y \leqslant z \,\&\, P(x_1, x_2, \ldots, x_n, y)].$$

Here the representative function of the predicate Q is given by the product

$$\varphi(x_1, x_2, \ldots, x_n) = \prod_{y=0}^{z} \varphi(x_1, x_2, \ldots, x_n, y).$$

Now let us introduce the *minimalization operator*.

Assume that we are given a primitive recursive predicate $P(x_1, x_2, \ldots, x_n, y)$ and that it is known *a priori* that the condition

$$(\forall x_1)(\forall x_2)\ldots(\forall x_n)(\exists y)_{y\leqslant z}P(x_1, x_2, \ldots, x_n, y), \tag{12.10}$$

is satisfied; that is, for any set $x_1, x_2, \ldots, x_n$ there exists at least one $y \leqslant z$ such that $P(x_1, x_2, \ldots, x_n, y)$ will be true.

Then the predicate P may be used to define a function $\psi(x_1, x_2, \ldots, x_n)$ in the following manner: given $x_1^*, x_2^*, \ldots, x_n^*$ the value of function $\psi(x_1^*, x_2^*, \ldots, x_n^*)$ is the smallest number y^* at which $P(x_1^*, x_2^*, \ldots, x_n^*, y^*)$ is true. We shall indicate this by writing

$$\psi(x_1, x_2, \ldots, x_n) = \mu y_{y\leqslant z}P(x_1, x_2, \ldots, x_n, y). \tag{12.11}$$

By virtue of Eq. (12.10), such a y exists for all $x_1, x_2, \ldots, x_n$; consequently, $\psi(x_1, x_2, \ldots, x_n)$ is defined at all points.

If we deal with the representative function $\varphi(x_1, x_2, \ldots, x_n, y)$ of the predicate $P(x_1, x_2, \ldots, x_n, y)$ rather than the predicate itself, then Eq. (12.11) becomes

$$\psi(x_1, x_2, \ldots, x_n) = \mu y_{y\leqslant z}[\varphi(x_1, x_2, \ldots, x_n, y) = 0],$$

that is, the smallest y at which $\varphi(x_1, x_2, \ldots, x_n, y)$ vanishes is taken as the value of the function $\psi(x_1, x_2, \ldots, x_n)$.

Thus, the minimalization operator is a means for deriving new functions, starting from primitive recursive ones.

We shall now show that the function $\psi(x_1, x_2, \ldots, x_n)$, defined by means of the minimalization operator, is primitive recursive. For this purpose, we shall explicitly express the $\psi(x_1, x_2, \ldots, x_n)$ in terms of the representative function $\varphi(x_1, x_2, \ldots, x_n, y)$ of the predicate P

$$\psi(x_1, x_2, \ldots, x_n) = \sum_{k=0}^{z} \operatorname{sg}\left(\prod_{y=0}^{k}\varphi(x_1, x_2, \ldots, x_n, y)\right). \tag{12.12}$$

That Eq. (12.12) does indeed express $\psi(x_1, x_2, \ldots, x_n)$ can be verified in the following manner: let us expand expression (12.12):

$$\begin{aligned}\psi(x_1, x_2, \ldots, x_n) = {} & \operatorname{sg}[\varphi(x_1, x_2, \ldots, x_n, 0)] + \\ & + \operatorname{sg}[\varphi(x_1, x_2, \ldots, x_n, 0)\cdot\varphi(x_1, x_2, \ldots, x_n, 1)] + \\ & + \operatorname{sg}[\varphi(x_1, x_2, \ldots, x_n, 0)\cdot\varphi(x_1, x_2, \ldots, x_n, 1)\cdot \\ & \qquad \cdot\varphi(x_1, x_2, \ldots, x_n, 2)] + \ldots\end{aligned}$$

All of the above summands which include terms preceding $\varphi(x_1, x_2, \ldots, x_n, y) = 0$ are equal to 1; all succeeding summands are 0. Thus, the entire operation amounts to adding 1 to itself y times; that is, the addition gives the number y.

Since $\psi(x_1, x_2, \ldots, x_n)$ is defined in terms of sums, products, and the function $\operatorname{sg}(x)$, it is a primitive recursive function.

Previously (Section 12.6), we have cited $\operatorname{res}(a, n)$ as an example of a primitive recursive function. This function also gives the number of divisors of a. Let us divide a successively by 1, 2, 3, 4, ... and count the number of times the division gives no remainder. This will give the number of divisors of a, which we denote by $\rho(a)$. The function $\rho(a)$ is primitive recursive since

$$\rho(a) = \sum_{i=1}^{a} \overline{\operatorname{sg}}(\operatorname{res}(a, i)).$$

If a is a prime number, then $\rho(a) = 2$, since a prime number is divisible only by 1 and by itself. Then

$$\overline{\operatorname{sg}}(|\rho(a) - 2|) = \begin{cases} 1, \text{if } a \text{ is a prime number,} \\ 0 \text{ in all other cases.} \end{cases}$$

It is now easy to add up the number of prime numbers which do not exceed y. Let this number be $\pi(y)$:

$$\pi(y) = \sum_{a=2}^{y} \overline{\operatorname{sg}}(|\rho(a) - 2|).$$

The addition starts at $a = 2$, since we do not consider 0 and 1 as prime numbers: the zeroth prime number will then be 2, the first will be 3, and so on:

$$p_0 = 2,\ p_1 = 3,\ p_2 = 5,\ p_3 = 7,\ \ldots$$

Now let us tabulate some values of $\pi(y)$:

$$\begin{aligned} &\pi(2) = 1, \\ &\pi(3) = 2, \\ &\pi(4) = 2, \\ &\pi(5) = 3, \\ &\pi(6) = 3, \\ &\pi(7) = 4, \\ &\pi(8) = 4, \\ &\pi(9) = 4, \\ &\pi(10) = 4, \\ &\pi(11) = 5 \quad , \text{ etc.} \end{aligned}$$

We shall now define $p_n = \varphi(n)$ as a function which, for given n, yields the nth prime number. It is known from number theory that the nth prime number does not exceed $2^{2^{n+1}}$. We can, therefore, write

$$p_n = \mu y \left[y \leqslant 2^{2^{n+1}} \,\&\, \pi(y) = n + 1 \right],$$

since the nth prime number is the smallest such that the numbers not greater than it include exactly $n + 1$ primes. For example,

$$\varphi(1) = p_1 = \mu y [y \leqslant 16 \,\&\, \pi(y) = 2] = 3,$$
$$\varphi(2) = p_2 = \mu y [y \leqslant 256 \,\&\, \pi(y) = 3] = 5.$$

The function $p_n = \varphi(n)$ is primitive recursive since it is defined by means of the minimalization operator [the primitive recursive function $z(n) = 2^{2^{n+1}}$ acts as a bound $z(n)$ in this instance], as well as the primitive recursive relationship of $\pi(y) = n + 1$.

Let us define still another function—that giving the number of times a prime p_a occurs in the decomposition of n, and let us denote the function by $p_a(n)$. Obviously, the value of $\operatorname{ex} p_a(n)$ is the largest y for which p_a^y is still a divisor of n, or, alternatively, the smallest y at which p_a^{y+1} fails to be a divisor of n. We can then write

$$\operatorname{ex} p_a(n) = \mu y [y \leqslant n \,\&\, p_a^{y+1} \text{ is not a divisor of } n]$$

or

$$\operatorname{ex} p_a(n) = \mu y [y \leqslant n \,\&\, \operatorname{res}(n,\ p_a^{y+1}) \neq 0]. \tag{12.13}$$

Inspection of Eq. (12.13) shows that $\operatorname{ex} p_a(n)$ is a primitive recursive function. Now, the reader will recall that gödelization is associated with the decomposition of a given number into prime factors and determination of the exponent with which the prime number p_a occurs in this decomposition. Consequently, *gödelization is associated only with primitive recursive functions.*

In conclusion we shall cite, without proof, two additional primitive recursive functions. Let $m_1, m_2, \ldots, m_r$ be a set of numbers whose Gödel number is a, and let $n_1, n_2, \ldots, n_s$ be a set whose Gödel number is b. We shall now form a new sequence $m_1, m_2, \ldots, m_r, n_1, n_2, \ldots, n_s$ by appending the sequence $n_1, n_2, \ldots, n_s$ to the sequence $m_1, m_2, \ldots, m_r$. We want to determine, from the known Gödel numbers a and b, the Gödel number y for the composite sequence. The function thus defined is denoted by writing $y = a \circ b$, and is primitive recursive.

Now let $m_1, m_2, \ldots, \underline{m_i, m_{i+1}, \ldots, m_j}, m_{j+1}, \ldots, m_n$, be a sequence of numbers whose Gödel number is a. We shall cut out from this sequence the segment beginning with m_i and ending with m_j (this segment is underlined in the above expression), and insert in its place another sequence whose Gödel number is b. We want to determine the Gödel number y for the new sequence. The function giving this number in terms of known a, i, j, and b is denoted by

$$y = \operatorname{subst}_a\binom{i, j}{b}$$

and is primitive recursive.

Now recall the transformation of words in associative calculus. The operation of substitution following gödelization of an associative calculus reduces to the above inclusion operation. Consequently, transformation of words in associate calculus is also associated only with primitive recursive functions.

These conclusions will be useful in the discussion of general recursive functions.

12.8. A COMPUTABLE BUT NOT PRIMITIVE RECURSIVE FUNCTION

So far, we have dealt with primitive recursive functions. The very nature of the derivation of such functions shows that all primitive recursive functions are computable. But is the converse true? Are all computable functions primitive recursive? The answer is no. We know this from the work Péter and Ackermann who, almost simultaneously and in entirely different ways, constructed examples of a computable but not primitive recursive function. Let us follow Péter's reasoning.

Péter was the first to notice that the set of primitively recursive functions is countable. Indeed, the class of primitive functions is countable (since the number of different variables x_i and constants q is countable). Consequently, the class of primitive recursive functions, derived by a single application of operations IV or V of Section 12.6, is also countable, since the set of the sets $\psi, \chi_1, \chi_2, \ldots, \chi_m$ used in operation IV is countable, as is the set of pairs ψ, χ for operation V; this must be so since these sets are formed from elements of a countable class.

Further, the set of primitively recursive functions derived by means of two applications of operations IV or V is countable, and so on. By the same reasoning, the set of primitive recursive functions is, in general, countable. In particular, the set of primitive functions of one variable is countable (because it is contained in this countable set).

Péter succeeded in *actually numbering* all the primitive recursive functions of one variable, that is, in arranging them into a sequence

$$\varphi_0(x),\ \varphi_1(x),\ \varphi_2(x),\ \varphi_3(x),\ \ldots \tag{12.14}$$

so that from the form of a function one can determine its number, while (conversely) the form of the function is given by the

corresponding number. Then it became possible to construct an example of a computable function which is not primitively recursive.

Suppose we have a function $\psi(y, x) \equiv \varphi_y(x)$; $\psi(yx)$ is countable [since from the value $y = y^*$ one can find the corresponding function $\varphi_{y^*}(x)$ and compute its value for a given $x = x^*$; this would automatically give the value of $\psi(y^*, x^*)$]. This function is not primitive recursive. Indeed, if $\psi(y, x)$ were primitive recursive, so would $\psi(x, x)$ be, which is a function of one variable. Then $\psi(x, x) + 1$, would also be primitive recursive, since the addition of 1 constitutes an allowable operation of "succession." But since the series (12.14) contains *all* the primitive recursive functions of one variable, there would exist a number y^*, such that $\psi(x, x) + 1 \equiv \varphi_{y^*}(x)$ for all x. In other words, $\psi(x, x) + 1 \equiv \psi(y^*, x)$. Since this identity must hold for all x, it holds, in particular, for $x = y^*$. But then

$$\psi(y^*, y^*) + 1 \equiv \psi(y^*, y^*),$$

which is impossible. It means that the enumerating function $\psi(y, x)$ is not primitive recursive. This function is known, however, to be computable. Consequently, the class of primitive recursive functions does not encompass all computable functions. It must be broadened to serve our purposes.

Whereas in the case of elementary functions we were limited by the fact that we were unable to construct very rapidly increasing functions by means of allowable operations, in the case of primitive recursive functions we are limited by our *form of induction*. The trouble is that we have fixed in advance the operation (V), that is, the form in which the induction must appear.

Extension of the class of primitively recursive functions was proposed by Gödel in 1934, based on a bold idea of Herbrand.

12.9. GENERAL RECURSIVE FUNCTIONS

The Herbrand-Gödel Definition

So far, we have dealt with recursive functions, where a function φ was defined in terms of several functions χ and ψ, assumed to be known *a priori*. Now let us examine two computations using only one auxiliary function χ.

Example 1. Assume we are given the system

$$\chi(0, 4) = 7, \tag{12.15}$$
$$\chi(1, 7) = 7, \tag{12.16}$$

$$\varphi(0) = 4, \tag{12.17}$$

$$\varphi(y') = \chi(y, \varphi(y)). \tag{12.18}$$

It is required to find the chain of formal inferences which yields $\varphi(2) = 7$, starting from Eqs. (12.15) to (12.18).

1. In (12.18) we set $y = 0$:

$$\varphi(1) = \chi(0, \varphi(0)). \tag{12.19}$$

2. In (12.19) we replace $\varphi(0)$ by 4, in accordance with (12.17):

$$\varphi(1) = \chi(0, 4). \tag{12.20}$$

3. We then use (12.15):

$$\varphi(1) = 7.$$

Continuing in a similar manner, we get successively:

4. $\varphi(2) = \chi(1, \varphi(1))$.
5. $\varphi(2) = \chi(1, 7)$.
6. $\varphi(2) = 7$

Example 2. Assume that function $\varphi(n, a)$ is given by

$$\varphi(0, a) = a, \tag{12.21}$$

$$\varphi(n+1, a) = \varphi(n, a) + 1 \tag{12.22}$$

and that we want to find the value of $\varphi(3, 5)$. By a formal analysis, we shall find the operations needed for computing $\varphi(3, 5)$ by means of Eqs. (12.21) and (12.22).

1. In Eq. (12.22), we set $n = 2$, $a = 5$. Then

$$\varphi(3.5) = \varphi(2.5) + 1. \tag{12.221}$$

2. Now we set $n = 1$, $a = 5$ in (12.22) and determine $\varphi(2,5)$:

$$\varphi(2.5) = \varphi(1.5) + 1. \tag{12.222}$$

3. Again

$$\varphi(1.5) = \varphi(0.5) + 1. \tag{12.223}$$

4. We set $a = 5$ in Eq. (12.21). Then

$$\varphi(0.5) = 5. \tag{12.224}$$

5. We substitute this value of φ (0,5) into (12.223) and get

$$\varphi(1.5) = 5 + 1 = 6. \tag{12.225}$$

6. Now, substituting this value of $\varphi(1.5)$ into (12.223), we get

$$\varphi(2.5) = 6 + 1 = 7. \tag{12.226}$$

7. Finally, substituting this value of φ (2.5) into (12.221), we get

$$\varphi(3.5) = 7 + 1 = 8. \tag{12.227}$$

We required only two operations to compute the answers for the above two examples. These operations were (1) replacement of symbols (variables) by numbers and (2) substitution of equivalents, whereby we used one side of an equation as a replacement for the other (see steps 5 and 6 above).

If one can, by means of these two operations, deduce another equation from a given system of equations E, then this equation is said to be *deducible in the system E.* Since deducibility is crucial to the theory of general recursive functions, we shall consider it in detail. First, we shall introduce a broad and exact definition of deducibility. We begin by defining a "term," an "equation," and an "inference."

The letters used so far to denote functions φ, χ, $\varkappa$, σ, φ_1, φ_2, φ_3, φ_4, ... shall be called the *functional signs* (the list of functional signs is infinite). The variables will again be denoted by x, y, z, t, m, n, a, b, c, x_1, x_2, x_3, ...

We shall define a "term" by induction:

1. 0 is a term.
2. Each variable is a term.
3. R' is a term if R is a term.
4. $\varphi(R_1, R_2, \ldots, R_n)$ is a term if φ is a functional sign and $R_1, R_2, \ldots, R_n$ are terms.
5. There are no other terms.

The following are examples of forms:

1. The number 3 is a term (because 0 is a term, hence $0' = 1$ is a term, hence $1' = 2$ is a term, so that $2' = 3$ is a term).
2. Any constant is a term. Again, constants shall be denoted as x^*, y^*, z^*, t^*, m^*, n^*, ...
3. φ (2) is a term. The following are also terms:
4. $\varphi(x, y)$.
5. $\varphi(x, \chi_1(8, y), \chi_2(3, 5), \chi_3(y, z))$.
6. $\chi(x', y)$.
7. $\psi(y', x, (\varphi(z))')$, etc.

The following are not terms:

1. $\varphi(\psi)$; 2. $5(x)$; 3. $y(\varphi)$; 4. $\varphi(z, \psi)$

and so on. Thus, terms are specific expressions which are composed of symbols denoting variables, constants, and functional signs by means of brackets and primes.

Now let us define equations. *An equation shall be an expression* $R = S$, *where* R *and* S *are terms*. New equations shall be deducible from a given system of equations E by means of the following operations:

1. Substitution of numbers for symbols of variables.

2. Transformation of expressions $R = S$ and $H = P$ which do not contain variables (where R, S, H, P are terms) into an expression derived from $R = S$ by one of more simultaneous substitutions of P for occurrences of H.

The reader will recall that these were the only two operations used in the two examples considered above.

Now, our scheme for deriving primitive recursive functions involved the following "properties":

a) The values of the functions were derived from equations by a method which can be formally analyzed.

b) Each definition was arrived at by mathematical induction.

We have already established above that primitive recursive functions are a restricted class because of the mode of induction which was fixed in advance. The *a priori* fixing of the inductive method is the root of the difficulty. Were we to adopt another, possibly even a broader induction method, we would still have no guarantee that the new method would not lead to a quite restricted class of recursive functions. Herbrand therefore suggested that the induction method be left open (not fixed) and that property (a) itself be used as a definition. The Herbrand-Gödel definition of the general recursive function is as follows:

A function $\varphi(x_1, x_2, \ldots, x_n)$ *is general recursive if there exists a finite system of equations* E *such that, for any set of* $x_1^*, x_2^*, \ldots, x_n^*$, *there is one and only one* y^*, *such that the equation* $\varphi(x_1^*, x_2^*, \ldots, x_n^*) = y$ *can be deduced from* E *by a finite number of applications of operations 1 and 2* (that is, replacement of variables by numbers and substitution of equivalents).

The system E is the *defining system* of equations; one also says that E *defines the function* φ *recursively*.

This definition does not require that a function be computable from its values at preceding points; it does not require that the auxiliary functions contained in the system E be computable at all points; and no induction method is fixed *a priori*. The only requirement is that the system E define a particular value of φ (with the aid of other values of φ and values of auxiliary functions) in such a

way that φ will be uniquely computable from E at all points. Uniqueness in this instance means that E does not simultaneously yield two contradictory equations.

This definition of a general recursive function is not by itself a computation procedure. The definition merely says that if a given system of equations E recursively defines a function φ, then for any $x_1^*, x_2^*, \ldots, x_n^*$ there exists a y^* such that the equation

$$\varphi(x_1^*, x_2^*, \ldots, x_n^*) = y^*$$

can be deduced from E. But how does one go about such a deduction? How does one find y^*? One obvious way is to keep on deducing equations derivable from E until a suitable equation comes up. But that may take an infinite time. The reader will recall that a poorly organized search can lead to infinitely long wandering and no result even in a finite labyrinth. Thus some organization is a necessity if equation $\varphi(x_1^*, x_2^*, \ldots, x_n^*) = y^*$ is to be deduced in a finite, though not *a priori* bounded, number of steps. We shall not dwell on the description of the techniques employed. Suffice it to say that gödelization allows us to reduce the scanning of all the possible deductions to the application of the operator minimalization. This operator also permits another method of defining recursive functions.

12.10. EXPLICIT FORM OF GENERAL RECURSIVE FUNCTIONS

In Section 12.7 we introduced the bounded smallest-number operator which places a primitive recursive predicate $P(x_1, x_2, \ldots, x_n, y)$, or a primitive recursive function $\varphi(x_1, x_2, \ldots, x_n, y)$ representing P, into correspondence with a primitive recursive function $\psi(x_1, x_2, \ldots, x_n)$:

$$\psi(x_1, x_2, \ldots, x_n) = \mu y_{y \leqslant z} P(x_1, x_2, \ldots, x_n, y) = \tag{12.23}$$
$$= \mu y_{y \leqslant z} [\varphi(x_1, x_2, \ldots, x_n, y) = 0]$$

provided

$$(\forall x_1)(\forall x_2) \ldots (\forall x_n)(\exists y)_{y \leqslant z} P(x_1, x_2, \ldots, x_n, y)$$

or

$$(\forall x_1)(\forall x_2) \ldots (\forall x_n)(\exists y)_{y \leqslant z} [\varphi(x_1, x_2, \ldots, x_n, y) = 0],$$

where z may, in general, be a primitive recursive function of $x_1, x_2, \ldots, x_n$:

$$z = z(x_1, x_2, \ldots, x_n).$$

Let us now consider a case where the operator is not bounded. Let $\varphi(x_1, x_2, \ldots, x_n, y)$ be a primitive recursive function such that

$$(\forall x_1)(\forall x_2)\ldots(\forall x_n)(\exists y)[\varphi(x_1, x_2, \ldots, x_n, y) = 0]. \qquad (12.24)$$

Here there is no upper bound for y. The only stipulation is that for all $x_1, x_2, \ldots, x_n$ there exist a y such that

$$\varphi(x_1, x_2, \ldots, x_n, y) = 0.$$

In this case the function $\psi(x_1, x_2, \ldots, x_n)$, defined by means of the minimalization operator

$$\psi(x_1, x_2, \ldots, x_n) = \mu y[\varphi(x_1, x_2, \ldots, x_n, y) = 0], \qquad (12.25)$$

is *a priori* computable. Indeed, to compute its values at a point $x_1^*, x_2^*, \ldots, x_n^*$, it is sufficient to compute successively $\varphi(x_1^*, x_2^*, \ldots, x_n^*, 0)$, $\varphi(x_1^*, x_2^*, \ldots, x_n^*, 1)$, $\varphi(x_1^*, x_2^*, \ldots, x_n^*, 2)$ and so on, until one obtains a y^* such that $\varphi(x_1^*, x_2^*, \ldots, x_n^*, y^*) = 0$. The value of y^* is then the value of ψ at the point under consideration.

This computation procedure must end in a finite number of steps, because Eq. (12.24) indicates the existence of a y^* at which $\varphi = 0$. Now we want to know whether the computable function $\psi(x_1, x_2, \ldots, x_n)$ defined by Eq. (12.25) subject to condition (12.24) is general recursive. It turns out that there is a system of equations E which recursively defines ψ, that is, ψ is a general recursive function. To simplify the derivation, we shall consider a function of one variable

$$\left.\begin{aligned} &\psi(x) = \mu y[\varphi(x, y) = 0] \\ \text{and} \quad &(\forall x)(\exists y)[\varphi(x, y) = 0]. \end{aligned}\right\} \qquad (12.26)$$

Here, the system E, which defines $\psi(x)$ recursively, is as follows:

$$\left.\begin{aligned} &1.\ \sigma(0, x, y) = y, \\ &2.\ \sigma(z+1, x, y) = \sigma[\varphi(x, y+1), x, y+1], \\ &3.\ \psi(x) = \sigma[\varphi(x, 0), x, 0]. \end{aligned}\right\} \quad (E)$$

Let us prove that E does indeed define $\psi(x)$ recursively. Suppose that x^* is a number and we want to determine $\psi(x^*)$. According to Eq. 3 of E,

$$\psi(x^*) = \sigma[\varphi(x^*, 0), x^*, 0].$$

Now there are two possibilities: either $\varphi(x^*, 0)$ vanishes or it does not. If $\varphi(x^*, 0) = 0$, we can only use the first equation of E:

$$\sigma(0, x^*, 0) = 0, \text{ that is, } \psi(x^*) = 0.$$

But then the value of ψ must also be zero in accordance with (12.26). If, however, $\psi(x^*, 0) \neq 0$, its value may be represented as $z + 1$, and we can use the second defining equation of E:

$$\sigma[\varphi(x^*, 0), x^*, 0] = \sigma[\varphi(x^*, 1), x^*, 1].$$

Here again there are two possibilities: either $\varphi(x^*, 1)$ vanishes or it does not. If $\varphi(x^*, 1) = 0$, then we can use only Eq. 1 of E:

$$\sigma[0, x^*, 1] = 1$$

and, consequently, $\psi(x^*) = 1$. In this case Eq. 1 of E does in fact yield the value of ψ indicated by Eq. (12.26). If, however, $\varphi(x^*, 1) \neq 0$, we may represent it as $z + 1$ and again use Eq. 2 of E:

$$\sigma[\varphi(x^*, 1), x^*, 1] = \sigma[\varphi(x^*, 2), x^*, 2].$$

We continue this procedure until we find a y^* such that $\varphi(x^*, y^*) = 0$. That value of y^* will be the value of $\psi(x^*)$.

Therefore, the system E does indeed recursively define the function

$$\psi(x) = \mu y[\varphi(x, y) = 0]$$

and consequently $\psi(x)$ is a general recursive function.

In the above proof we did not use the fact that the function $\varphi(x_1, x_2, \ldots, x_n, y)$ of Eq. (12.25) is primitive recursive. For this reason, the argument holds completely even if function $\varphi(x_1, x_2, \ldots, x_n, y)$ is assumed to be general recursive.

Thus, if condition (12.24) is satisfied, the minimalization operator μy permits us to derive general recursive functions from primitive recursive functions (predicates). Further work has also shown that the difference between primitive recursive and general recursive functions resides entirely in the operator μy. Thus it has been proved that *any general recursive function* $\varphi(x_1, x_2, \ldots, x_n)$ *may be represented as*

$$\varphi(x_1, x_2, \ldots, x_n) = \psi\{\mu y[\tau(x_1, x_2, \ldots, x_n, y) = 0]\}, \qquad (12.27)$$

where ψ and τ are primitive recursive functions, while the following statement holds for the function τ:

$$(\forall x_1)(\forall x_2) \ldots (\forall x_n)(\exists y)[\tau(x_1, x_2, \ldots, x_n, y) = 0].$$

Equation (12.27) is the explicit form of general recursive functions. Let us sketch out the proof of the above statement. Assume $E = \{e_0, e_1, \ldots, e_s\}$ is a system of equations defining a function $\varphi(x_1, x_2, \ldots, x_n)$ recursively. Each equation has a Gödel number m_i.

Then the Gödel number for the entire system E is

$$w = p_0^{m_0} p_1^{m_1} p_2^{m_2} \dots p_s^{m_s}.$$

Now, we shall deduce new equations from the system E. This means that we shall successively obtain equations

$$\tilde{e}_0, \tilde{e}_1, \tilde{e}_2, \dots, \tilde{e}_r, \dots. \tag{12.28}$$

If n_i is the Gödel number of equation $\tilde{e}_i$, then each inference [that is, each string such as (12.28)] can be put into correspondence with the Gödel number of this inference

$$z = p_0^{n_0} p_1^{n_1} \dots p_r^{n_r}.$$

Suppose we want to evaluate φ at point $x_1^*, x_2^*, \dots, x_n^*$, that is, we wish to derive from the system E an equation of the form

$$\varphi(x_1^*, x_2^*, \dots, x_n^*) = y^*. \tag{12.29}$$

What are the properties of the Gödel number z of this inference?

1) Equations can be inferred from other equations, as we already know, by substitution of numbers for variables and replacement of occurrences. In these procedures, the Gödel numbers of the resulting new equations are primitive recursive functions of the Gödel numbers of the starting equations, since the operations of replacement of occurrences, substitution, and the determination of the Gödel number are associated only with primitive recursive functions. Some of these functions were already considered above [$\operatorname{ex} p_i(x)$, $a \circ b$, $\operatorname{subst}_a \binom{i, j}{b}$, $p_n = \varphi(n)$, etc.].

Therefore, the first requirement which z must satisfy is this: each of the exponents $n_0, n_1, n_2, \dots$ of the decomposition of z into primes must be either the Gödel number of one of the defining equations e_i or the value of some primitive recursive function of these (Gödel) numbers.

2) The last exponent n_r of the decomposition of z must be the Gödel number of an equation such as (12.29).

It turns out that the predicate

$$T(x_1^*, x_2^*, \dots, x_n^*, z) \equiv \left\{ \begin{array}{l} z \text{ is the Gödel number of the} \\ \text{inference of the value of} \\ \varphi(x_1^*, x_2^*, \dots, x_n^*) \end{array} \right\}$$

is a primitive recursive predicate. Consequently, its representing function $\tau(x_1^*, x_2^*, \dots, x_n^*, z)$ is also primitive recursive, and it is equal to zero for those z which are the Gödel numbers of inferences

terminating in the equation

$$\varphi(x_1^*, x_2^*, \ldots, x_n^*) = y^*$$

and only for those.

For this reason, our problem of finding the desired inference may be formulated as follows: find at least one number z^*, such that

$$\tau(x_1^*, x_2^*, \ldots, x_n^*, z^*) = 0. \tag{12.30}$$

Since an inference must exist at all points (by definition, E recursively defines φ recursively), the function τ has the property

$$(\forall x_1)(\forall x_2) \ldots (\forall x_n)(\exists z)[\tau(x_1, x_2, \ldots, x_n, z) = 0]. \tag{12.31}$$

Having found a z^* which Eq. (12.30) is satisfied, we can decode this Gödel number and get

$$y^* = \psi(z^*),$$

whereby ψ also turns out to be a primitive recursive function, since the decoding reduces to the following primitive recursive operations: determination of the last exponent n_r in the decomposition of z^* followed by decoding of the number n_r [which is the Gödel number of our Eq. (12.29)]. Moreover, $\psi(z)$ turns out to be a universal primitive recursive function, identical for all systems E (that is, for all general recursive functions φ), since the decoding of the Gödel number of the inference always proceeds in a standard way.

If, we we have established, any z for which

$$\tau(x_1^*, x_2^*, \ldots, x_n^*, z) = 0,$$

is the Gödel number of the desired inference, then

$$\mu z[\tau(x_1^*, x_2^*, \ldots, x_n^*, z) = 0]$$

is also the Gödel number of this inference. We shall, therefore, finally get

$$y = \varphi(x_1, x_2, \ldots, x_n) = \psi\{\mu z[\tau(x_1, x_2, \ldots, x_n, z) = 0]\},$$

where ψ and τ are primitive recursive functions and the condition (12.31) is satisfied for τ.

It should be pointed out that from the form of expression (12.30) immediately indicates that all general recursive functions form a countable set.* This conclusion arises from the fact that the number

*This conclusion could have been arrived at earlier, by observing that the set of different systems E recursively defining functions φ is countable, since all these systems E can be tagged by means of Gödel numbers.

of different recursive functions τ defining general recursive functions by means of the scheme (12.30) is also countable (denumerable).

However, in contrast with primitive recursive functions, the set of general recursive functions is not effectively countable (for further details, see Section 12.12), and, consequently, Péter's method does not allow us to construct an example of an enumerable function more general than the general recursive.

To conclude this section, let us write out the operations defining general recursive functions. Here, operations I - V are the already familiar schemes for defining primitively recursive functions, while operation VI is the explicit form of a general recursive function:

$$\begin{aligned}
&\text{I. } \varphi(x) = x', \\
&\text{II. } \varphi(x_1, x_2, \ldots, x_n) = q, \\
&\text{III. } \varphi(x_1, x_2, \ldots, x_n) = x_i, \\
&\text{IV. } \varphi(x_1, x_2, \ldots, x_n) = \psi(\chi_1(x_1, x_2, \ldots \\
&\quad \ldots, x_n), \chi_2(x_1, x_2, \ldots, x_n), \ldots, \chi_m(x_1, x_2, \ldots, x_n)), \\
&\text{Va. } \varphi(0) = q, \quad \varphi(y') = \chi(y, \varphi(y)), \\
&\text{Vb } \varphi(0, x_2, \ldots, x_n) = \psi(x_2, \ldots, x_n), \\
&\varphi(y', x_2, \ldots, x_n) = \chi(y, \varphi(y, x_2, \ldots, x_n), x_2, \ldots, x_n), \\
&\text{VI. } \varphi(x_1, x_2, \ldots, x_n) = \psi[\mu y[\tau(x_1, x_2, \ldots, x_n, y) = 0]],
\end{aligned}$$

whereby

$$(\forall x_1)(\forall x_2) \ldots (\forall x_n)(\exists y)[\tau(x_1, x_2, \ldots, x_n, y) = 0].$$

Now we can define a general recursive function: *A function $\varphi(x_1, x_2, \ldots, x_n)$ is said to be general recursive if it can be defined by using operations I - IV a finite number of times.*

Since operations I - V defining the primitive recursive functions are encompassed by operations I - VI defining general recursive functions, primitive recursive functions are a special case of general recursive functions; every primitive recursive function is a general recursive function. However, the converse is not true.

12.11. CHURCH'S THESIS

Let us now return to our initial problem, that of defining the class of computable functions. In solving this problem, we have defined in succession, the class of elementary functions, then the class of primitive recursive functions, and finally the broad class of general recursive functions. Now we must ask: is this the final solution? Or must the class of general recursive functions be further broadened?

The many attempts at broadening the class of general recursive functions have all ended in failure. And in 1936 Church suggested that *every effectively countable* function (or effectively solvable predicate) is general recursive (see [110]). By virtue of this thesis, *the class of computable functions coincides with the class of general recursive functions.*

Church's thesis cannot be proved, since it contains, on the one hand, the vague concept of a computable function and, on the other, the mathematically exact concept of a general recursive function. The thesis is a hypothesis supoorted by several valid arguments which no one has so far succeeded in refuting. One such argument is that the various refinements of the concept of an algorithm turn out to be equivalent. Thus, for instance, Markov's normal algorithm proved to be reducible to general recursive functions.

Previously we said that an "algorithm" and a "computation of the values of an arithmetical function" are identical concepts. In the light of Church's thesis a problem is algorithmically solvable only if the arithmetical function to the computation of which we reduce our problem is general recursive.

To sum up, an algorithm can exist only if a corresponding general recursive function can be constructed.

Conversely, by virtue of Church's thesis, the algorithmic unsolvability of a problem means that the arithmetical function to the computation of which the problem is reduced is not general recursive.

The proof of algorithmic unsolvability is often as involved, difficult and time-consuming as the search for an algorithm. However, algorithmic unsolvability can be proved in some cases. We shall give, without proof, two examples of this type:

Example 1. If we had an algorithm which, given the Gödel number w of an equation system E, would be capable of deciding by inspection of w whether E defines a general recursive function, then we could define once and for all which systems E define general recursive functions, and we could effectively number all such functions. In other words, we need a general recursive function $\psi(w)$ such that:

$$\psi(w) \begin{cases} = 0 \text{ if } w \text{ is the Gödel number of} \\ \quad \text{system } E \text{ defining a general} \\ \quad \text{recursive function,} \\ > 0 \text{ in all other cases.} \end{cases}$$

It has been proved [42] that such a general recursive function $\psi(w)$ does not exist. Therefore, the problem of recognizing those

systems E which define general recursive functions is algorithmically unsolvable. The set of general recursive functions turns out to be countable, but not effectively so.

Example 2. The following problem turns out to be algorithmically unsolvable: it is required to find an algorithm for recognizing, for any primitive recursive function $\tau(x, y)$ [or for any primitive recursive predicate $T(x, y)$] whether that function has the property

$$(\forall x)(\exists y)[\tau(x, y) = 0] \tag{12.32}$$

or in the case of a predicate, whether $(\forall x)(\exists y)\, T(x, y)]$.

Since primitive recursive functions can be effectively and distinctively labeled (numbered), the problem reduces to finding a computable function $\psi(r)$ such that:

$$\psi(r) \begin{cases} = 0 & \text{if } r \text{ is the number of a primitive recursive function having the property (12.32).} \\ > 0 & \text{in all other cases.} \end{cases}$$

This function proved not to be general recursive and, consequently, is noncomputable.

Even if condition (12.32) is somewhat weakened, the problem is still algorithmically unsolvable. Thus the following simple problem is algorithmically unsolvable: given a primitive recursive function $\varphi(x, y)$ it is required to find, for any x^*, whether the following condition holds for that x^*:

$$(\exists y)[\varphi(x^*, y) = 0].$$

Yet another algorithmically unsolvable problem is this: for any primitive recursive predicate $P(y)$, it is required to find whether it is true that

$$(\exists y)\, P(y).$$

In all cases, the proof reduces to proving that the corresponding recognition function is not general recursive.

One often proves the algorithmic unsolvability of a given problem by showing that it reduces to another problem, whose algorithmic unsolvability has already been proved. Sometimes it suffices to show that a narrower problem, which is a special case of the given problem, is algorithmically unsolvable. In Chapters 8 and 9, we used the method in proving algorithmic unsolvability of the two basic problems of the theory of finite automata and sequential machines.

12.12. RECURSIVE REAL NUMBERS

Recursive real numbers occur in *constructive* mathematics, an approach which has developed from the desire to avoid getting entrapped in logical contradictions (antinomies). In this approach, a proof is considered complete if, in addition to establishing a mathematical fact, one is able to demonstrate that the corresponding mathematical objects can also be computed.

The machinery of recursive functions plays an important role in constructive mathematics: it is in terms of these functions that the algorithms for effective construction of required objects are usually defined.

Consider a typical constructive formulation of a frequent practical problem. In analysis one often comes across the statement "for every small $\varepsilon > 0$ there exists a number N such that some quantity (which is a function of n) becomes smaller than ε for $n \geqslant N$." Now, what is the constructive variant of this statement? In order to arrive at it, we require:

1. A more precise definition of what we understand by "every small ε." To define such ε, we may assume, for example, $\varepsilon = \frac{1}{m}$, where m is a positive integer which may be as large as desired.

2. An effective method for determining N, starting from ε (that is, from m).

Therefore, an effective formulation of the above statement is: "there exists a general recursive function $\nu(m)$ such that some quantity which is a function of n becomes smaller than $\frac{1}{m}$ for $n \geqslant \nu(m)$."

This kind of formulation can be related to convergence of a sequence of rational numbers. Let us say that a sequence of rational numbers $a_0, a_1, a_2, \ldots, a_n, \ldots$ is recursive if there exist general recursive functions $\alpha(n), \beta(n)$ [where $\gamma \geqslant 1$] such that

$$a_n = \frac{\alpha(n) - \beta(n)}{\gamma(n)}.$$

We shall say that the *sequence converges recursively* (or *effectively)* if there exists a general recursive function $\nu(m)$ such that for any arbitrarily large $m > 0$,

$$|a_n - a_{n^*}| < \frac{1}{m}, \quad \text{if} \quad n, n^* \geqslant \nu(m).$$

The number r, defined by this effectively convergent sequence, is called *a recursive real number*.

It can be shown that the recursivity of r (that is, the fact that there exists a sequence of rational numbers which, in the limit, recursively approaches r) does not imply that r can be expanded

into a recursive decimal fraction. That is, the recursivity does not imply the existence of a general recursive function $\psi(n) \leqslant 9$ such that

$$r = \sum_{k=0}^{\infty} \frac{\psi(k)}{10^k}.$$

However, there exists one special sequence whose recursive convergence implies the possibility of recursive expansion of r in any number systems. This is the factorial expansion

$$r = \frac{a_1}{1!} + \frac{a_2}{2!} + \frac{a_3}{3!} + \ldots + \frac{a_n}{n!} + \ldots$$

where $a_n \leqslant n - 1$ for large n. If there exists a general recursive function $\chi(n)$ such that $a_n = \chi(n)$, this series always converges recursively and defines a recursive real number, which can be expanded into a recursive fraction in any number system.

Let us also mention here that the set of recursive real numbers is not larger than the set of general recursive functions; that is, this set is not larger than a countable set, whereas the set of all real numbers is a continuum. In this sense, only a very small fraction of all real numbers are recursive.

To summarize, a recursive real number is really a number which may be computed to any degree of accuracy by means of an algorithm. All numbers usually employed in mathematical analysis (e, π, $\sqrt{2}$, and so on) are recursive real numbers.

12.13. RECURSIVELY ENUMERABLE AND RECURSIVE SETS

There are several equivalent formulations defining recursive and recursively enumerable sets of integers. For convenience, we shall assemble these definitions into a table:

Recursively enumerable sets of numbers	Recursive sets of numbers
1A. A set is said to be recursively enumerable if it consists of values of some general recursive function if, alternatively, if there exists a general recursive function enumerating it, even if this involves repetitions. The empty set is deemed to be recursively enumerable. 2A. A set C containing at least one element is recursively enumerable if and only if the predicate "$y \epsilon C$" can be expressed in the form $(\exists x)P(x,y)$, where $P(x, y)$ is general recursive.	1B. A set C is said to be recursive if there exists an algorithm for determining whether a given number y belongs to C. 2B. A set C is said to be recursive if the predicate "$y \epsilon C$" can be expressed in the form $P(y)$, where P is general recursive. 3B. A set C is said to be recursive if both the set and its complement $\bar{C}$ are recursively enumerable. 4B. An infinite set C is recursive if and only if it can be enumerated by a general recursive function without repetition and in increasing order of its elements [that is if $\varphi(0)$, $\varphi(1)$, $\varphi(2)$, ... give the elements of C in increasing order].

Now, a few remarks regarding these definitions:

Note on 1A. It has been proved that, if a set can be enumerated be a general recursive function with repetition, it can also be enumerated without repetition (by another general recursive function).

Note on 2A. We shall show that definition 2A follows from 1A. Let C be the set of values of a general recursive function $\varphi(x)$. Then, the fact that a number y belongs to C means that there exists an x such that

$$y = \varphi(x),$$

or that

$$(\exists x)[y = \varphi(x)].$$

Equation $y = \varphi(x)$ may be regarded as a general recursive relationship of equality between two general recursive functions

$$P(x, y) \equiv [\chi_1(x, y) = \chi_2(x, y)],$$

where

$$\chi_1(x, y) \equiv y, \quad \chi_2(x, y) \equiv \varphi(x).$$

Note on 2B. Definition 2B is merely a more exact version of definition 1B.

Note on 3B. If condition 3B is satisfied, then it follows that condition 1B is also satisfied. Indeed, let C be enumerated by function $\varphi_1(x)$, and set $\bar{C}$ by the function $\varphi_2(x)$. To ascertain whether a given number belongs to C, we shall compute the parallel sequences

$$\varphi_1(0), \quad \varphi_1(1), \quad \varphi_1(2), \quad \varphi_1(3), \ldots \tag{I}$$

$$\varphi_2(0), \quad \varphi_2(1), \quad \varphi_2(2), \quad \varphi_2(3), \ldots \tag{II}$$

Since y belongs either to C or to $\bar{C}$, sooner or later it will appear either in row I or in row II. If it appears in row I, then $y \in C$, and if in row II, then $y \in \bar{C}$. Thus, there exists an algorithm for determining whether any y belongs to C.

Note on 4B. When condition 4B holds, there also exists an algorithm for recognizing the membership of any y in set C. Indeed, let us compute the sequence $\varphi(0), \varphi(1), \varphi(2), \ldots$. If, at some n, we arrive at $\varphi(n) > y$, then there is no need to continue the computation, and we may conclude that $y \in \bar{C}$; if, however, we find $m \leq n$ such that $\varphi(m) = y$, then $y \in C$.

We shall give a few examples of recursive sets:

1) The two—element set $\{0, 1\}$ is recursive by virtue of condition 1B or 2B.

2) Any finite set is recursive by virtue of condition 1B or 2B.

3) The set of even numbers $\{0, 2, 4, 6, 8, \ldots\}$ is recursive. Here $y \in C = [\text{res}(y, 2) = 0]$, and the set is recursive by virtue of 2B; or $\varphi(x) = 2x$, and the set is recursive by virtue of 4B.

Now, let us give examples of sets that are and sets that are not recursively enumerable.

According to definition 2A, the set of all y for which $(\exists x) P(x, y)$ at some general recursive $P(x, y)$ is recursively enumerable. One can so select $P(x, y)$ that the set $\{y\}$ will be recursively enumerable, but not recursive; its complement $\{\bar{y}\}$ will be the set of those y for which

$$\overline{(\exists x) P(x, y)} = (\forall x) \bar{P}(x, y) = (\forall x) Q(x, y)$$

(where Q is a general recursive predicate); this set $\{\bar{y}\}$ will be neither recursive nor recursively enumerable.

The set of Gödel numbers z of systems E which define a general recursive function is neither recursive nor recursively enumerable.

In conclusion, let us point out that the comparison of 1A and 4B, as well as the fact that any finite set is both recursive and recursively enumerable imply that any recursive set is recursively enumerable. The converse, however, is not true.

The concept of recursive real numbers and recursively enumerable sets is important in determining whether a machine "can do" more than just realize a given algorithm. For we have shown above that any algorithm reduces to the computation of the values of a computable integer function. *Thus, if a device generates an output of a set of numbers and that set is not recursively enumerable, we immediately know that the operation of this device cannot be represented by an algorithm; that is, this device "does more" than just realize an algorithm.*