~~words, given a pair $\langle A, \mathbf{b} \rangle$ where A is an $m \times n$ rational matrix and $\mathbf{b}$ is an m dimensional rational vector, find out if there exists an n-dimensional vector $\mathbf{x}$ such that $A\mathbf{x} = \mathbf{b}$. The standard Gaussian elimination algorithm solves this problem in $O(n^3)$ *arithmetic operations*. But on a Turing machine, each arithmetic operation has to be done in the gradeschool fashion, bit by laborious bit. Thus, to prove that this decision problem is in **P**, we have to verify that Gaussian elimination (or some other algorithm for the problem) runs on a Turing machine in time that is polynomial in the number of bits required to represent $a_1, a_2, \ldots, a_n$. That is, in the case of Gaussian elimination, we need to verify that all the intermediate numbers involved in the computation can be represented by polynomially many bits. Fortunately, this does turn out to be the case (for a related result, see Exercise 2.3).~~

Arora Barak 2007

1.6.1 Why the model may not matter

We defined the classes of "computable" languages and **P** using Turing machines. Would they be different if we had used a different computational model? Would these classes be different for some advanced alien civilization, which has discovered computation but with a different computational model than the Turing machine?

We already encountered variations on the Turing machine model, and saw that the weakest one can simulate the strongest one with quadratic slow down. Thus *polynomial* time is the same on all these variants, as is the set of computable problems.

In the few decades after Church and Turing's work, many other models of computation were discovered, some quite bizarre. It was easily shown that the Turing machine can simulate all of them with at most polynomial slowdown. Thus, the analog of **P** on these models is no larger than that for the Turing machine.

Most scientists believe the **Church-Turing (CT) thesis**, which states that every physically realizable computation device—whether it's based on silicon, DNA, neurons or some other alien technology—can be simulated by a Turing machine. This implies that the set of *computable* problems would be no larger on any other computational model that on the Turing machine. (The CT thesis is not a theorem, merely a belief about the nature of the world as we currently understand it.)

However, when it comes to *efficiently* computable problems, the situation is less clear. The **strong form of the CT thesis** says that every physically realizable computation model can be simulated by a TM *with polynomial overhead* (in other words, t steps on the model can be simulated in t^c steps on the TM, where c is a constant that depends upon the model). If true, it implies that the class **P** defined by the aliens will be the same as ours. However, this strong form is somewhat controversial, in particular because of models such as *quantum computers* (see Chapter 10), which do not appear to be efficiently simulatable on TMs. However, it is still unclear if quantum computers can be physically realized.

~~1.6.2 On the philosophical importance of P~~

~~The class **P** is felt to capture the notion of decision problems with "feasible" decision procedures. Of course, one may argue whether $\mathbf{DTIME}(n^{100})$ really represents "feasible" computation in the real world since n^{100} is prohibitively huge even for moderate~~

~~DEFINITION 2.5. *An n-ary function $f(x_1, \ldots, x_n)$ is* **partially computable** *if there exists a Turing machine Z such that*~~

~~$f(x_1, \ldots, x_n) = \Psi_Z^{(n)}(x_1, \ldots, x_n).$~~

~~*In this case we say that Z computes f. If, in addition, $f(x_1, \ldots, x_n)$ is a total function, then it is called* **computable.**~~

It is the concept of *computable function* that we propose to identify with the intuitive concept of effectively calculable function. A partially computable function may be thought of as one for which we possess an algorithm which enables us to compute its value for elements of its domain, but which will have us computing forever in attempting to obtain a functional value for an element not in its domain, without ever assuring us that no value is forthcoming. In other words, when an answer is forthcoming, the algorithm provides it; when no answer is forthcoming, the algorithm has one spend an infinite amount of time in a vain search for an answer. We shall now comment briefly on the adequacy of our identification of effective calculability with computability in the sense of Definition 2.5. The situation is quite analogous to that met whenever one attempts to replace a vague concept, having a powerful intuitive appeal, with an exact mathematical substitute. (An obvious example is the area under a curve.) In such a case, it is, of course, pointless to demand a mathematical proof of the equivalence of the two concepts; the very vagueness of the intuitive concept precludes this. However, it is possible to present arguments, having strong intuitive appeal, which tend to make this identification extremely reasonable. We shall outline several arguments of this sort.

Historically, proposals were made by a number of different persons at about the same time (1936), mostly independently of one another, to identify the concept of effectively calculable function with various precise concepts. In this connection we may mention Church's notion of λ-*definability*,[1] the Herbrand-Gödel-Kleene notion of *general recursiveness*,[2] Turing's notion of *computability*[3] (defined in a manner differing somewhat from that of the present work), Post's notion of 1-*definability*,[4] and Post's notion of *binormality*.[5] These notions, which (except for the third and fourth) were quite different in formulation, have all been proved equivalent[6] in the sense that the classes of functions obtained are the

[1] Cf. Church [1, 3].

[2] Defined in Gödel [2]. The proposal to identify with effective calculability first appeared in Church [1]. Cf. also Kleene [1, 4, 6].

[3] Cf. Turing [1]. Davis 1958

[4] Cf. Post [1].

[5] Cf. Post [2, 3]; Rosenbloom [1].

[6] The equivalence of λ-definability with general recursiveness is proved in Kleene [2]. The equivalence of λ-definability with Turing's notion of computability is

same in each case. Now, the fact that these different concepts have turned out to be equivalent tends to make the identification of them with effective calculability the more reasonable.

Next, we may note that every computable function must surely be regarded as effectively calculable. For let $f(m)$ (for simplicity, we consider a singulary function) be computable, and let Z be a Turing machine which computes $f(m)$. Then, if we are given a number m_0, we may begin with the instantaneous description $\alpha_1 = q_1\overline{m_0} = q_1\underbrace{11\cdots1}_{m_0+1}$ and successively obtain instantaneous descriptions $\alpha_2, \alpha_3, \ldots, \alpha_p$, where $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \cdots \rightarrow \alpha_p$ and where α_p is terminal. Since $f(m)$ is ~~computable, such a terminal α_p must be obtainable in a finite number of steps. But then $f(m_0) = \langle\alpha_p\rangle$, and $\langle\alpha_p\rangle$ is simply the number of 1's in α_p. That this procedure would ordinarily be regarded as "effective" is clear when one realizes that to decide, for a given instantaneous description α of Z, whether or not there exists an instantaneous description β such that $\alpha \rightarrow \beta \quad (Z)$, and, if the answer is affirmative, to determine which β satisfies this condition, it suffices to write α in the form Pq_iS_jQ and to locate that quadruple of Z, if one exists, that begins $q_i\ S_j$.~~

This indicates, at any rate, that our definition is not too "wide." Is it, perhaps, too "narrow"? An answer as satisfying as the one given for the previous question is, presumably, not to be expected. For how can we ever exclude the possibility of our being presented, some day (perhaps by some extraterrestrial visitors), with a (perhaps extremely complex) device or "oracle" that "computes" a noncomputable function? However, there are fairly convincing reasons for believing that this will never happen. It is possible to show directly, for various possible theoretical computing devices which seem to possess greater power than Turing machines, that any functions computed by them are, in fact, computable. Thus, we might consider machines which could move any number of squares to the right or left, or which operate on a two-, three-, or one-hundred-dimensional tape, or which are capable of inserting squares into their own tape. Now, it is not very difficult to show that any computation that could be carried out by such a machine can also be performed by a Turing machine. This will be more readily

---

proved in Turing [2]. It follows at once, from the results of our Chap. 4 and the main result of Kleene [1], that our present notion of computability is equivalent to general recursiveness. Equivalence proofs for Post's notion of 1-definability and binormality do not appear in the published literature. 1-definability is quite similar, conceptually, to computability, and an equivalence proof is quite easy. A direct proof of the equivalence of binormality with general recursiveness is given in the author's dissertation (Davis [1]). This equivalence is, in fact, an immediate consequence of the results of our present Chap. 6.

CHAPTER 6

Hopcroft Ullman 1969

TURING MACHINES

6.1 INTRODUCTION

In this chapter we investigate a third type of recognizing device, the Turing machine. The Turing machine has been proposed as a mathematical model for describing procedures. Since our intuitive notion of a procedure as a finite sequence of instructions which can be mechanically carried out is not mathematically precise, we can never hope to show formally that it is equivalent to the precise notion of a Turing machine. However, from the definition of a Turing machine, it will be readily apparent that any computation that can be described by means of a Turing machine can be mechanically carried out. Thus the definition is not too broad. It can also be shown that any computation that can be performed on a modern-day digital computer can be described by means of a Turing machine. Thus if one ever found a procedure that fitted the intuitive notions, but could not be described by means of a Turing machine, it would indeed be of an unusual nature since it could not possibly be programmed for any existing computer. Many other formalizations of a procedure have been proposed, and they have been shown to be equivalent to the Turing machine formalization. This strengthens our belief that the Turing machine is general enough to encompass the intuitive notion of a procedure. It has been hypothesized by Church that any process which could naturally be called a procedure can be realized by a Turing machine. Subsequently, computability by a Turing machine has become the accepted definition of a procedure. We shall accept Church's hypothesis and simply substitute the formal definition of a Turing machine for the intuitive notion of a procedure.

6.2 DEFINITIONS AND NOTATION

Specifications for the Turing machine have been given in various ways in the literature. We begin with the discussion of a basic model, as shown in Fig. 6.1. Later we investigate other models of the Turing machine, and show that all these models are equivalent. The basic model has a *finite control*, an input tape which is divided into cells, and a *tape head* which scans one cell of the tape at a time. The tape has a leftmost cell but is infinite to the right. Each cell of the tape may hold exactly one of a finite number of *tape*

5 Un~~decidability~~

Lewis Papadimitriou 1981

5.1 THE CHURCH-TURING THESIS

In this book we address this question: What can be computed? (And, more intriguingly, what *cannot* be computed?) We have introduced various and diverse mathematical models of computational processes that accomplish concrete computational tasks —in particular, decide, semidecide, or generate languages, and compute functions. In the previous chapter we saw that Turing machines can carry out surprisingly complex tasks of this sort. We have also seen that certain additional features that we might consider adding to the basic Turing machine model, including a random access capability, do not increase the set of tasks that can be accomplished. Also, following a completely different path (namely, trying to generalize context-free grammars), we arrived at a class of language generators with precisely the same power as Turing machines. Finally, by trying to formalize our intuitions on which numerical functions can be considered computable, we defined a class of functions that turned out to be precisely the recursive ones.

All this suggests that we have reached a natural upper limit on what a computational device can be designed to do; that our search for the ultimate and most general mathematical notion of a computational process, of an *algorithm*, has been concluded successfully —and the Turing machine is the right answer. However, we have also seen in the last chapter that not all Turing machines deserve to be called "algorithms:" We argued that Turing machines that semidecide languages, and thus reject by never halting, are not useful computational devices, whereas Turing machines that decide languages and compute functions (and therefore halt at all inputs) are. Our notion of an algorithm must exclude Turing machines that may not halt on some inputs.

We therefore propose to adopt the Turing machine that halts on all inputs as the precise formal notion corresponding to the intuitive notion of an "algorithm." Nothing will be considered an algorithm if it cannot be rendered as a Turing machine that is guaranteed to halt on all inputs, and all such machines will be rightfully called algorithms. This principle is known as the **Church-Turing thesis**. It is a thesis, not a theorem, because it is not a mathematical result: It simply asserts that a certain informal concept (algorithm) corresponds to a certain mathematical object (Turing machine). Not being a mathematical statement, the Church-Turing thesis cannot be proved. It is theoretically possible, however, that the Church-Turing thesis could be *dis*proved at some future date, if someone were to propose an alternative model of computation that was publicly acceptable as a plausible and reasonable model of computation, and yet was provably capable of carrying out computations that cannot be carried out by any Turing machine. No one considers this likely.

~~Adopting a precise mathematical notion of an algorithm opens up the intriguing possibility of formally proving that certain computational problems *cannot* be solved by *any* algorithm. We already know enough to expect this. In Chapter 1 we argued that if strings are used to represent languages, not every language can be represented: there are only a countable number of strings over an alphabet, and there are uncountably many languages. Finite automata, pushdown automata, context-free grammars, unrestricted grammars, and Turing machines are all examples of finite objects that can be used for specifying languages, and that can be themselves described by strings (in the next section we develop in detail a particular way of representing Turing machines as strings). Accordingly, there are only countably many recursive and recursively enumerable languages over any alphabet. So although we have worked hard to extend the capabilities of computing machines as far as possible, in absolute terms they can be used for semideciding or deciding only an infinitesimal fraction of all the possible languages.~~

~~Using cardinality arguments to establish the limitation of our approach is trivial; finding particular examples of computational tasks that cannot be accomplished within a model is much more interesting and rewarding. In earlier chapters we did succeed in finding certain languages that are not regular or context-free; in this chapter we do the same for the recursive languages. There are two major differences, however. First, these new negative results are not just temporary setbacks, to be remedied in a later chapter where an even more powerful computational device will be defined: according to the Church-Turing thesis, computational tasks that cannot be performed by Turing machines are impossible, hopeless, *undecidable*. Second, our methods for proving that languages are not recursive will have to be different from the "pumping" theorems we used for exploiting the weaknesses of context-free grammars and finite au-~~

~~cursive functions. By functions we mean functions of one or more variables, hence, what we usually call arithmetical *operations* are also functions. Arithmetical functions are not only important examples of possibly computable objects, but also they have a sufficiently rich structure. Thus it is possible to encode other computations only using operations on natural numbers. In general we have to choose a suitable domain for data. In the case of Turing machines the domain consists of sequences of symbols, while the domain of recursive functions is the set of all natural numbers.~~

~~In logic it is very common to define a class inductively by saying that some initial elements belong to the class and giving some operators that produce new elements in the class. The class consists of those elements that can be produced in this way. (Think of, say, proofs as defined by axioms and derivation rules.) Kleene, as a logician, used this approach. He took some basic functions and considered several operators producing new functions from given ones. The basic functions are quite simple, such as the constant function 0, the successor function $x + 1$, etc., so they clearly should be considered computable. We can add other functions, for example, addition and multiplication, that we surely consider computable, but it is not necessary as the operators enable us to produce them from the basic ones. The operators are also simple. In particular, we take the operator of *composition* (also called *substitution*) of functions. Having two functions f and g of one variable, we may first apply f to the input number and then apply g to the value produced by f. The resulting function is the composition of f and g. We may also compose binary functions. Thus we can get, for example, the function $x + yz$ from addition and multiplication.~~

~~If we started with the basic arithmetic operations (addition, multiplication and constants) and only used composition we would only obtain the functions that can be expressed by algebraic terms (polynomials). Therefore, we need more operators. Another basic operator is the operator of *recursion*. The name '*recursive functions*' comes from this operator, but it is misleading, as this operator is still too weak to produce all computable functions. A special case of it is the operator of *iteration*, by which we compose a function with itself a given number of times. The most powerful operator is the *minimization* operator. It allows us to search for the smallest number satisfying a condition.~~

~~We can imagine recursive functions as follows. We expand our "algebraic" language by taking more operators on top of the composition. Having a sufficiently powerful set of basic functions and operators enables us to define all computable functions by an expression in this language.~~

~~A formal definition of recursive functions is in Notes.~~

Pudlak 2013

The Church-Turing Thesis

Having definitions of computable functions the next natural question is how good these definitions are. What seems clear is that each of the definitions only describes functions that can be (at least in principle) computed. But do these concepts (Turing machines, programs, recursive functions, etc.) cover all computable functions? This is the same kind of a property that we studied in logical calculi and called completeness, but here we have a problem: we do not have a class of functions that we would consider as the computable functions and that would be defined independently of a concrete computational model. We do not have a purely semantical definition of computable functions. We do have a fairly clear idea about computable functions and all the computation models are in good agreement with it, (the idea is that a computation should use a finite number of elementary operations), but to make this idea precise we have to opt for one of the computation models. Before we choose one, it is, surely, worthwhile to compare them. In particular, is the class of numeric functions computable by Turing machines the same as the class of recursive functions? I have already said that all algorithms can be done on Turing machines, so it should not come as a surprise that the two classes coincide. In fact, if you think about these concepts, after a while you will realize that they are not so different as they appear at first glance. If you try to implement algorithms on a Turing machine you will soon realize that a programming language for it would be handy. That is exactly like asking for a higher level language instead of a machine code for your computer. So isn't a Turing machine rather a primitive programming language? In some sense it surely is. It is a programming language with a single data structure which is a linear array and a single pointer whose position can be incremented or decremented only by one. If you analyze it more, you may find an even closer correspondence, for example, the program lines correspond to the state of the control in the Turing machine, etc.

What about recursive functions? Also this definition can be interpreted as a kind of a programming language. It is a programming language for computations with natural numbers. It has some simple functions as primitive concepts, as most programming languages do. Then it has certain operators that we can interpret as possible constructions that can be used in a program. One of these is *composition*, that is used to form terms; this is also available in most programming languages. Another is *recursion*, again this construction is very common in programming languages. It helps when writing very short programs, but professional programmers try to avoid it whenever possible, as it does not give them good control of the computation. *Minimization* is not present in programming languages as a basic construct (it gives even less control of what is going on during the computation), but it can be programmed easily. On the other hand, *iteration* corresponds to a simple loop in a program and this is the most frequent construction.

Such mutual interpretations were found not only for the aforementioned three concepts, but for all that had been proposed. This is not a proof, but sufficiently good evidence that the concepts have been chosen correctly. The claim that these concepts characterize computable functions is called the *Church-Turing Thesis* (although neither Church nor Turing stated the thesis precisely in the way it is presented nowadays).

Can the Church-Turing Thesis be proved or disproved? Firstly, it cannot be proved or disproved as a *mathematical* statement because it is not a mathematical statement. It relates mathematical concepts to part of our practical experience for which we do not have a rigorous definition. Theoretically, it is possible that somebody might come up with a programming trick, an operator on functions, etc. that we would like to call computable but that would not be covered by the current definitions of computations. In such a case we would have a reason to abandon the thesis, but strictly speaking this would not be disproving. Considering the years of experience with programming, such a possibility is almost excluded.

The only way to make this thesis a little more formal statement is to interpret it as a postulate in *physics*. It perfectly makes sense to take the current physical theories and look whether the phenomena there have a computable nature and how they can be used for computations. The conjecture that all *physically* computable functions are computable on Turing machines, or their equivalents, is called the *Physical Church-Turing Thesis*. This question has been studied and lead to the new concept of quantum computing. Quantum Turing machines are a new important concept and it seems that they can solve some problems faster than classical Turing machines (as we will see in Chap. 5), but when computational complexity is ignored, the two models are equivalent. This is not a proof, but a strong evidence that the Physical Church-Turing Thesis cannot be disproved using quantum theory.

Naturally, also general relativity was used in the attempts to refute the Physical Church-Turing Thesis. There the situation is less clear. There are enthusiasts who believe that some likely occurring phenomena, such as black holes, can be used to solve problems that are not computable on Turing machines, others are more sceptical. In any case, this is only a theoretical discussion; nobody believes that such schemes can ever be used to help people compute. This is in contrast with quantum computing, where several teams of experimental physicists are working on constructing quantum computers. Nevertheless, the research into relativistic computations is extremely important because it concerns the fundamental question of what can physically be computed. (For more about it, see Notes.)

It has also been suggested that a noncomputable function could be encoded in fundamental physical constants. For example, the decimal digits of some constant could define a noncomputable set. If there also were a way to measure the value of the constant with arbitrary high precision, we would obtain a refutation of the Physical Church-Turing Thesis. In the most optimistic scenario we would also know that the digits of the constant encode a particular noncomputable set and we could use it to decide Church-Turing undecidable problems.

It should be noted that the Physical Church-Turing Thesis is meaningful only if the universe is infinite, which we still do not know. If space and time were finite, then there could only be finite computations. In such a case the fundamental role of computability would be replaced by the role of computational complexity. But I believe that computational complexity is equally important for physics even if the universe is infinite.

~~*The Syntax and the Semantics of Computations*~~

~~The distinction between syntax and semantics is essentially the same as in logic. Syntax is (descriptions of) Turing machines and programs in programming lan-~~

Rogers 1968

§1.7 CHURCH'S THESIS

The claim that each of the standard formal characterizations provides satisfactory counterparts to the informal notions of *algorithm* and *algorithmic function* cannot be proved. It must be accepted or rejected on grounds that are, in large part, empirical. (That the claim for one characterization is equivalent to the claim for another follows from Parts I and III of the Basic Result.) The Basic Result provides impressive evidence that the class of partial functions defined is a natural one (Part I) and that it is sufficiently inclusive (Parts I and II). The Turing characterization provides convincing evidence that every partial function in the class is computable by a procedure that is, intuitively, "mechanical." (In §1.10 we shall discuss further the possibility that the formal class is too inclusive; see question *10 in §1.1.) On the basis of this evidence, many mathematicians have accepted the claim that the standard characterizations give a satisfactory formalization, or "rational reconstruction," of the (necessarily vague) informal notions. This claim is often referred to as *Church's Thesis.* Church's Thesis may be viewed as a *proposal* as well as a claim, a proposal that we agree henceforth to supply certain previously intuitive terms (e.g., "function computable by algorithm") with certain precise meanings.

In recent theoretical work, the phrase "Church's Thesis" has come to play a somewhat broader role than that indicated above. In Parts II and III of the Basic Result, we noted that a number of powerful techniques have been developed for showing that partial functions with informal algorithms are in fact partial recursive and for going from an informal set of instructions to a formal set of instructions. These techniques have been developed to a point where (*a*) a mathematician can recognize whether or not an alleged informal algorithm provides a partial recursive function, much as, in other parts of mathematics, he can recognize whether or not an alleged informal proof is valid, and where (*b*) a logician can go from an informal definition for an algorithm to a formal definition, much as, in other parts of mathematics, he can go from an informal to a formal proof. Recursive-function theory, of course, deals with a precise subject matter: the class of partial functions defined in §1.5. Researchers in the area, however, have been using informal methods with increasing confidence. We shall rely heavily on such methods in this book. They permit us to avoid cumbersome detail and to isolate crucial mathematical ideas from a background of routine manipulation. We shall see that much profound mathematical substance can be discussed, proved, and communicated in this way. *We continue to claim, however, that our results have concrete mathematical status as results about the class of partial functions formally characterized in* §1.5. *Of course, any investigator who uses informal methods and makes such a claim must be prepared to supply formal details if challenged.*

Proofs which rely on informal methods have, in their favor, all the evidence accumulated in favor of Church's Thesis. Such proofs will be called *proofs by Church's Thesis.*

We meet our first examples of such informal methods in the remaining sections of this chapter. Almost all the proofs in this book will use Church's Thesis to some extent. The analogy to informal methods of proof in other parts of mathematics is instructive. In both cases, the use of informal methods is a matter not of extremes but of degree. The degree of formalization of a proof usually depends upon the complexity and abstraction (what might be called the "danger") of the argument. The degree of formal detail we employ in this book will similarly vary with circumstances.

The beginning reader, who does not possess first-hand knowledge of the evidence for Church's Thesis, may be troubled by our arguments. To whatever extent he experiences doubt, we urge him to use the books of Davis and Kleene, in which he will find the tools needed to formalize our arguments fully.

§1.8 GÖDEL NUMBERS, UNIVERSALITY, *s-m-n* THEOREM

~~We have adopted the Turing-machine characterization as basic. We saw in §1.5 that a set of instructions is a set of quadruples satisfying the consistency restriction. It is possible to list all sets of instructions by a procedure similar to that indicated in §1.4 for listing all primitive recursive derivations. This procedure is itself algorithmic (in our first, unrestricted, informal sense of that word). It can be viewed as a procedure which associates with each integer x the set of instructions falling at the $(x + 1)$st place in the list of all sets of instructions. We assume now that we have selected one such listing procedure. We keep it fixed for the remainder of the book. We do not give formal details.~~

~~***Definition*** P_x is the set of instructions associated with the integer x in the fixed listing of all sets of instructions. x is called the *index* or *Gödel number* of P_x.~~

~~$\varphi_x^{(k)}$ is the partial function of k variables determined by P_x. x is also called an *index* or *Gödel number* for $\varphi_x^{(k)}$. (We shall drop the superscript (k) when its value is clear from context or when $k = 1$. We shall be most often concerned with functions of one variable.†)~~

~~Clearly the listing procedure gives us *both* (a) an algorithm for going from any x to the corresponding P_x, and (b) an algorithm for going from any consistent set of quadruples P to a corresponding integer x such that P is P_x.~~

~~† The notation $\{x\}$ (for our φ_x) also appears in the literature. We use the φ_x and P_x notations to emphasize further the distinction between extension and name, i.e., between partial function and set of instructions.~~

OG Alan Turing 1936 at 22 years old!!!!!

9. *The extent of the computable numbers.*

No attempt has yet been made to show that the "computable" numbers include all numbers which would naturally be regarded as computable. All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically. The real question at issue is "What are the possible processes which can be carried out in computing a number?"

The arguments which I shall use are of three kinds.

(*a*) A direct appeal to intuition.

(*b*) A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).

(*c*) Giving examples of large classes of numbers which are computable.

Once it is granted that computable numbers are all "computable", several other propositions of the same character follow. In particular, it follows that, if there is a general process for determining whether a formula of the Hilbert function calculus is provable, then the determination can be carried out by a machine.

I. [Type (*a*)]. This argument is only an elaboration of the ideas of § 1.

Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, *i.e.* on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent†. The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as

† If we regard a symbol as literally printed on a square we may suppose that the square is $0 \leqslant x \leqslant 1$, $0 \leqslant y \leqslant 1$. The symbol is defined as a set of points in this square, viz. the set occupied by printer's ink. If these sets are restricted to be measurable, we can define the "distance" between two symbols as the cost of transforming one symbol into the other if the cost of moving unit area of printer's ink unit distance is unity, and there is an infinite supply of ink at $x = 2$, $y = 0$. With this topology the symbols form a conditionally compact space.

17 or 999999999999999 is normally treated as a single symbol. Similarly in any European language words are treated as single symbols (Chinese, however, attempts to have an enumerable infinity of symbols). The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same.

The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused. Again, the restriction is not one which seriously affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape.

Let us imagine the operations performed by the computer to be split up into "simple operations" which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer (possibly with a special order), and the state of mind of the computer. We may suppose that in a simple operation not more than one symbol is altered. Any other changes can be split up into simple changes of this kind. The situation in regard to the squares whose symbols may be altered in this way is the same as in regard to the observed squares. We may, therefore, without loss of generality, assume that the squares whose symbols are changed are always "observed" squares.

Besides these changes of symbols, the simple operations must include changes of distribution of observed squares. The new observed squares must be immediately recognisable by the computer. I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount. Let us say that each of the new observed squares is within L squares of an immediately previously observed square.

In connection with "immediate recognisability", it may be thought that there are other kinds of square which are immediately recognisable. In particular, squares marked by special symbols might be taken as immediately recognisable. Now if these squares are marked only by single symbols there can be only a finite number of them, and we should not upset our theory by adjoining these marked squares to the observed squares. If, on the other hand, they are marked by a sequence of symbols, we cannot regard the process of recognition as a simple process. This is a fundamental point and should be illustrated. In most mathematical papers the equations and theorems are numbered. Normally the numbers do not go beyond (say) 1000. It is, therefore, possible to recognise a theorem at a glance by its number. But if the paper was very long, we might reach Theorem 157767733443477; then, further on in the paper, we might find "... hence (applying Theorem 157767733443477) we have ...". In order to make sure which was the relevant theorem we should have to compare the two numbers figure by figure, possibly ticking the figures off in pencil to make sure of their not being counted twice. If in spite of this it is still thought that there are other "immediately recognisable" squares, it does not upset my contention so long as these squares can be found by some process of which my type of machine is capable. This idea is developed in III below.

The simple operations must therefore include:

(*a*) Changes of the symbol on one of the observed squares.

(*b*) Changes of one of the squares observed to another square within L squares of one of the previously observed squares.

It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:

(*A*) A possible change (*a*) of symbol together with a possible change of state of mind.

(*B*) A possible change (*b*) of observed squares, together with a possible change of state of mind.

The operation actually performed is determined, as has been suggested on p.136, by the state of mind of the computer and the observed symbols. In particular, they determine the state of mind of the computer after the operation is carried out.

We may now construct a machine to do the work of this computer. To each state of mind of the computer corresponds an "m-configuration" of the machine. The machine scans B squares corresponding to the B squares observed by the computer. In any move the machine can change a symbol on a scanned square or can change any one of the scanned squares to another square distant not more than L squares from one of the other scanned squares. The move which is done, and the succeeding configuration, are determined by the scanned symbol and the m-configuration. The machines just described do not differ very essentially from computing machines as defined in § 2, and corresponding to any machine of this type a computing machine can be constructed to compute the same sequence, that is to say the sequence computed by the computer.

II. [Type (b)].

If the notation of the Hilbert functional calculus† is modified so as to be systematic, and so as to involve only a finite number of symbols, it becomes possible to construct an automatic‡ machine $\mathcal{K}$, which will find all the provable formulae of the calculus§.

Now let α be a sequence, and let us denote by $G_\alpha(x)$ the proposition "The x-th figure of α is 1", so that|| $-G_\alpha(x)$ means "The x-th figure of α is 0". Suppose further that we can find a set of properties which define the sequence α and which can be expressed in terms of $G_\alpha(x)$ and of the propositional functions $N(x)$ meaning "x is a non-negative integer" and $F(x, y)$ meaning "$y = x+1$". When we join all these formulae together conjunctively, we shall have a formula, $\mathfrak{A}$ say, which defines α. The terms of $\mathfrak{A}$ must include the necessary parts of the Peano axioms, viz.,

$$(\exists u) N(u) \,\&\, (x)\big(N(x) \to (\exists y) F(x, y)\big) \,\&\, \big(F(x, y) \to N(y)\big),$$

which we will abbreviate to P.

When we say "$\mathfrak{A}$ defines α", we mean that $-\mathfrak{A}$ is not a provable formula, and also that, for each n, one of the following formulae (A_n) or (B_n) is provable.

$$\mathfrak{A} \,\&\, F^{(n)} \to G_\alpha(u^{(n)}), \qquad (\mathrm{A}_n)\P$$

$$\mathfrak{A} \,\&\, F^{(n)} \to \big(-G_\alpha(u^{(n)})\big), \qquad (\mathrm{B}_n),$$

where $F^{(n)}$ stands for $F(u, u') \,\&\, F(u', u'') \,\&\, \ldots F(u^{(n-1)}, u^{(n)})$.

† The expression "the functional calculus" is used throughout to mean the *restricted* Hilbert functional calculus.

‡ It is most natural to construct first a choice machine (§ 2) to do this. But it is then easy to construct the required automatic machine. We can suppose that the choices are always choices between two possibilities 0 and 1. Each proof will then be determined by a sequence of choices $i_1, i_2, \ldots, i_n$ ($i_1 = 0$ or 1, $i_2 = 0$ or $1, \ldots, i_n = 0$ or 1), and hence the number $2^n + i_1 2^{n-1} + i_2 2^{n-2} + \ldots + i_n$ completely determines the proof. The automatic machine carries out successively proof 1, proof 2, proof 3, ...

§ The author has found a description of such a machine.

|| The negation sign is written before an expression and not over it.

¶ A sequence of r primes is denoted by $^{(r)}$.

Yasuhara 1971

~~$g_{p'}(n) = \begin{cases} g_p(n) & \text{if } g_p(n) \text{ is defined} \\ \text{something} & \text{otherwise} \end{cases}$~~

~~By the definition of g_p,~~

~~$g_p(p') = \begin{cases} g_{p'}(p') + 1 & \text{if } g_{p'}(p') \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$~~

~~Since $g_{p'}$ is total, it is defined and so we have $g_p(p') = g_{p'}(p') + 1$; but by the definition of $g_{p'}$, $g_p(p') = g_{p'}(p')$. Contradiction.~~ ▮

Whatever class of functions may be encompassed by the idea of total effectively computable functions, surely the class of recursive functions must be included. The facts that the recursive functions and the Turing-computable functions are the same class of functions, and that no one has yet produced a function which all agree to as being a total effectively computable function but not a recursive function, are often taken as weighty evidence in support of the following conjecture, known as *Church's thesis*†: Any total effectively computable function is a recursive function. Of course, "total effectively computable function" is not defined precisely; thus, perhaps the thesis should not be called a conjecture, since it cannot be proved. It might better be referred to as a working hypothesis. The student is encouraged to use Church's thesis in proving theorems and solving problems. However, he should not use it without giving some thought to backing up his arguments with actual Tm's or recursive functions. Let us give the name *the extended Church thesis*‡ to that version of Church's thesis that says that the class of partial effectively computable functions coincides with the class of partial recursive functions. Of course, this cannot be proved either, but we have as evidence in favor of its truth that the class of partial recursive functions coincides with the class of functions that are Turing computable in general. The reader is also encouraged to use the extended Church thesis, but to do so with considerable caution, as there are serious pitfalls. One in particular is of the following nature: since we are dealing with procedures that may not terminate, we can safely say, "*if* the procedure terminates, do so-and-so"; but we *cannot* say, "if the procedure does not terminate, do so′-and-so′." For examples of the possible difficulties, do exercises 5.9 and 5.10 below.

† For Church's original discussion, see Church (1936), sections 1 and 7. Further interesting discussions of this topic can be found in Davis (1958, p. 10), Kleene (1952, pp. 300, 331, see his index for more), Rogers (1967, section 1.7), and Shoenfield (1967, section 6.5).

‡ This is not a standard name. "Church's thesis" is sometimes used to include what is here called "the extended Church thesis." Consult the references of the previous footnote, in particular Kleene (1952, p. 331).