This page intentionally left blank
Computability and Logic, Fourth Edition

This fourth edition of one of the classic logic textbooks has been thoroughly revised by John Burgess. The aim is to increase the pedagogical value of the book for the core audience of students of philosophy and for students of mathematics and computer science as well.

This book has become a classic because of its accessibility to students without a mathematical background, and because it covers not simply the staple topics of an intermediate logic course such as Gödel’s incompleteness theorems, but also a large number of optional topics from Turing’s theory of computability to Ramsey’s theorem. John Burgess has enhanced the book by adding a selection of problems at the end of each chapter and by reorganizing and rewriting chapters to make them more independent of each other and thus to increase the range of options available to instructors as to what to cover and what to defer.

“...gives an excellent coverage of the fundamental theoretical results about logic involving computability, undecidability, axiomatization, definability, incompleteness, etc.” American Mathematical Monthly

“The writing style is excellent: although many explanations are formal, they are perfectly clear. Modern, elegant proofs help the reader understand the classic theorems and keep the book to a reasonable length.” Computing Reviews

“...a valuable asset to those who want to enhance their knowledge and strengthen their ideas in the areas of artificial intelligence, philosophy, theory of computing, discrete structures, mathematical logic. It is also useful to teachers for improving their teaching style in these subjects.” Computer Engineering
For
SALLY
and
AIGLI
and
EDITH
# Contents

*Preface*  |  page x  
---|---

## COMPUTABILITY THEORY

1 Enumerability  | 3  
1.1 Enumerability  | 3  
1.2 Enumerable Sets  | 7  

2 Diagonalization  | 16  

3 Turing Computability  | 23  

4 Uncomputability  | 35  
4.1 The Halting Problem  | 35  
4.2 The Productivity Function  | 40  

5 Abacus Computability  | 45  
5.1 Abacus Machines  | 45  
5.2 Simulating Abacus Machines by Turing Machines  | 51  
5.3 The Scope of Abacus Computability  | 57  

6 Recursive Functions  | 63  
6.1 Primitive Recursive Functions  | 63  
6.2 Minimization  | 70  

7 Recursive Sets and Relations  | 73  
7.1 Recursive Relations  | 73  
7.2 Semirecursive Relations  | 80  
7.3 Further Examples  | 83  

8 Equivalent Definitions of Computability  | 88  
8.1 Coding Turing Computations  | 88  
8.2 Universal Turing Machines  | 94  
8.3 Recursively Enumerable Sets  | 96  

---

vii
BASIC METALOGIC

9 A Précis of First-Order Logic: Syntax 101
  9.1 First-Order Logic 101
  9.2 Syntax 106

10 A Précis of First-Order Logic: Semantics 114
  10.1 Semantics 114
  10.2 Metalogical Notions 119

11 The Undecidability of First-Order Logic 126
  11.1 Logic and Turing Machines 126
  11.2 Logic and Primitive Recursive Functions 132

12 Models 137
  12.1 The Size and Number of Models 137
  12.2 Equivalence Relations 142
  12.3 The Löwenheim–Skolem and Compactness Theorems 146

13 The Existence of Models 153
  13.1 Outline of the Proof 153
  13.2 The First Stage of the Proof 156
  13.3 The Second Stage of the Proof 157
  13.4 The Third Stage of the Proof 160
  13.5 Nonenumerable Languages 162

14 Proofs and Completeness 166
  14.1 Sequent Calculus 166
  14.2 Soundness and Completeness 174
  14.3 Other Proof Procedures and Hilbert’s Thesis 179

15 Arithmetization 187
  15.1 Arithmetization of Syntax 187
  15.2 Gödel Numbers 192
  15.3 More Gödel Numbers 196

16 Representability of Recursive Functions 199
  16.1 Arithmetical Definability 199
  16.2 Minimal Arithmetic and Representability 207
  16.3 Mathematical Induction 212
  16.4 Robinson Arithmetic 215

17 Indefinability, Undecidability, Incompleteness 221
  17.1 The Diagonal Lemma and the Limitative Theorems 221
  17.2 Undecidable Sentences 225
  17.3 Undecidable Sentences without the Diagonal Lemma 227

18 The Unprovability of Consistency 233
CONTENTS

FURTHER TOPICS

19 Normal Forms 243
  19.1 Disjunctive and Prenex Normal Forms 243
  19.2 Skolem Normal Form 247
  19.3 Herbrand’s Theorem 253
  19.4 Eliminating Function Symbols and Identity 255

20 The Craig Interpolation Theorem 260
  20.1 Craig’s Theorem and Its Proof 260
  20.2 Robinson’s Joint Consistency Theorem 264
  20.3 Beth’s Definability Theorem 265

21 Monadic and Dyadic Logic 270
  21.1 Solvable and Unsolvable Decision Problems 270
  21.2 Monadic Logic 273
  21.3 Dyadic Logic 275

22 Second-Order Logic 279

23 Arithmetical Definability 286
  23.1 Arithmetical Definability and Truth 286
  23.2 Arithmetical Definability and Forcing 289

24 Decidability of Arithmetic without Multiplication 295

25 Nonstandard Models 302
  25.1 Order in Nonstandard Models 302
  25.2 Operations in Nonstandard Models 306
  25.3 Nonstandard Models of Analysis 312

26 Ramsey’s Theorem 319
  26.1 Ramsey’s Theorem: Finitary and Infinitary 319
  26.2 König’s Lemma 322

27 Modal Logic and Provability 327
  27.1 Modal Logic 327
  27.2 The Logic of Provability 334
  27.3 The Fixed Point and Normal Form Theorems 337

Hints for Selected Problems 341

Annotated Bibliography 348

Index 349
The original authors of this work, the late George Boolos and my late colleague Richard Jeffrey, stated in the preface to the first edition that the work was intended for students of philosophy, mathematics, and other fields who desired a more advanced knowledge of logic than is supplied by an introductory course or textbook on the subject, and added the following:

The aim has been to present the principal fundamental theoretical results about logic, and to cover certain other meta-logical results whose proofs are not easily obtainable elsewhere. We have tried to make the exposition as readable as was compatible with the presentation of complete proofs, to use the most elegant proofs we knew of, to employ standard notation, and to reduce hair (as it is technically known).

Such have remained the aims of all subsequent editions, including the present one.

The ‘principal fundamental theoretical results about logic’ are primarily the theorems of Gödel—the completeness theorem and especially the incompleteness theorems—with their attendant lemmas and corollaries. The ‘other meta-logical results’ included have been of two kinds. On the one hand, filling roughly the first third of the book, there is an extended exposition by R.C.J. of the theory of Turing machines, a topic frequently alluded to in the literature of philosophy, computer science, and cognitive studies, but often omitted in textbooks on the level of this one. On the other hand, there is a varied selection of theorems on (in)definability, (un)decidability, (in)completeness, and related topics, to which G.S.B. added a few more items with each successive edition, until it came to fill about the last third of the book.

The special aim of the present edition has been to increase the pedagogical usefulness of the book by adding a selection of problems at the end of each chapter, and by making chapters more independent of each other, so as to increase the range of options available to the instructor or reader as to what to cover and what to defer. Pursuit of the latter aim has involved substantial rewriting, especially in the middle third of the book. A number of the new problems and one new section on undecidability have been taken from G.S.B.’s Nachlass, while the rewriting of the précis of first-order logic—summarizing the material typically covered in a more leisurely way in an introductory text or course, and introducing the more abstract modes of reasoning that distinguish intermediate- from introductory-level logic—was undertaken in consultation with R.C.J. Otherwise, the changes in the present edition are the sole responsibility of J.P.B.

The book runs now in outline as follows. The basic course in intermediate logic, culminating in the first incompleteness theorem, is contained in Chapters 1, 2, 6, 7, 9, 10, 12, 15,
16, and 17, minus any sections of these chapters starred as optional. Necessary background
on enumerable and nonenumerable sets is supplied in Chapters 1 and 2. All the material
on computability (recursion theory) that is strictly needed for the incompleteness theorems
has now been collected in Chapters 6 and 7, which may if desired be postponed until after
the needed background material in logic. That material is presented in Chapters 9, 10, and
12. (For readers who have not had an introductory course in logic including a proof of the
completeness theorem, Chapters 13 and 14 will also be needed.) The machinery needed
for the proof of the incompleteness theorems is contained in Chapter 15 on the arithme-
tization of syntax (though the instructor or reader willing to rely on Church’s thesis may
omit all but the first section of this chapter) and in Chapter 16 on the representability of
recursive functions. The first completeness theorem itself is proved in Chapter 17. (The
second incompleteness theorem is discussed in Chapter 18.)

A semester course should allow time to take up several supplementary topics in addition
to this core material. The topic given the fullest exposition is the theory of Turing machines
and their relation to recursive functions, which is treated in Chapters 3 through 5 and 8 (with
an application to logic in Chapter 11). This now includes an account of Turing’s theorem
on the existence of a universal Turing machine, one of the intellectual landmarks of the last
century. If this material is to be included, Chapters 3 through 8 would best be taken in that
order, either after Chapter 2 or after Chapter 12 (or 14).

Chapters 19 through 21 deal with topics in general logic, and any or all of them might
be taken up as early as immediately after Chapter 12 (or 14). Chapter 19 is presupposed by
Chapters 20 and 21, but the latter are independent of each other. Chapters 22 through 26,
all independent of each other, deal with topics related to formal arithmetic, and any of them
would most naturally be taken up after Chapter 17. Only Chapter 27 presupposes Chapter 18.
Users of the previous edition of this work will find essentially all the material in it still here,
though not always in the same place, apart from some material in the previous version of
Chapter 27 that has, since the last edition of this book, gone into The Logic of Provability.

On the one hand, it should go without saying that in a textbook on a classical subject,
only a small number of the results presented will be original with the authors. On the other
hand, a textbook is perhaps not the best place to go into the minutiae of the history of a field.
Apart from a section of remarks at the end of Chapter 18, we have indicated the history of
the field for the student or reader mainly by the names attached to various theorems. See
also the annotated bibliography at the end of the book.

There remains the pleasant task of expressing gratitude to those (beyond the dedicatees) to
whom the authors have owed more personal debts. Earlier editions of this work already cited
Paul Benacerraf, Burton Dreben, Hartry Field, Clark Glymour, Warren Goldfarb, Simon
Kochen, Saul Kripke, David Lewis, Paul Mellema, Hilary Putnam, W. V. Quine, T. M.
Scanlon, James Thomson, and Peter Tovey, and Michael J. Pendlebury for Figure 5-13. For
this edition further thanks are due to Caspar Hare for Proposition 4.3; to him, Mike Fara,
and Nick Smith, for assisting me in teaching from drafts of the revised material; and to
Sinan Dogramaci, Jacob Rosen, Evan Williams, Brad Monton, David Keyt, and especially
Warren Goldfarb for lists of errata.

January 2003

JOHN P. BURGESS
Computability Theory
Our ultimate goal will be to present some celebrated theorems about inherent limits on what can be computed and on what can be proved. Before such results can be established, we need to undertake an analysis of computability and an analysis of provability. Computations involve positive integers 1, 2, 3, . . . in the first instance, while proofs consist of sequences of symbols from the usual alphabet A, B, C, . . . or some other. It will turn out to be important for the analysis both of computability and of provability to understand the relationship between positive integers and sequences of symbols, and background on that relationship is provided in the present chapter. The main topic is a distinction between two different kinds of infinite sets, the enumerable and the nonenumerable. This material is just a part of a larger theory of the infinite developed in works on set theory: the part most relevant to computation and proof. In section 1.1 we introduce the concept of enumerability. In section 1.2 we illustrate it by examples of enumerable sets. In the next chapter we give examples of nonenumerable sets.

1.1 Enumerability

An enumerable, or countable, set is one whose members can be enumerated: arranged in a single list with a first entry, a second entry, and so on, so that every member of the set appears sooner or later on the list. Examples: the set \( P \) of positive integers is enumerated by the list

\[ 1, 2, 3, 4, \ldots \]

and the set \( N \) of natural numbers is enumerated by the list

\[ 0, 1, 2, 3, \ldots \]

while the set \( P^- \) of negative integers is enumerated by the list

\[ -1, -2, -3, -4, \ldots \]

Note that the entries in these lists are not numbers but numerals, or names of numbers. In general, in listing the members of a set you manipulate names, not the things named. For instance, in enumerating the members of the United States Senate, you don’t have the senators form a queue; rather, you arrange their names in a list, perhaps alphabetically. (An arguable exception occurs in the case where the members...
of the set being enumerated are themselves linguistic expressions. In this case we can plausibly speak of arranging the members themselves in a list. But we might also speak of the entries in the list as names of themselves so as to be able to continue to insist that in enumerating a set, it is names of members of the set that are arranged in a list.

By courtesy, we regard as enumerable the empty set, $\emptyset$, which has no members. (The empty set; there is only one. The terminology is a bit misleading: It suggests comparison of empty sets with empty containers. But sets are more aptly compared with contents, and it should be considered that all empty containers have the same, null content.)

A list that enumerates a set may be finite or unending. An infinite set that is enumerable is said to be enumerably infinite or denumerable. Let us get clear about what things count as infinite lists, and what things do not. The positive integers can be arranged in a single infinite list as indicated above, but the following is not acceptable as a list of the positive integers:

$$1, 3, 5, 7, \ldots, 2, 4, 6, \ldots$$

Here, all the odd positive integers are listed, and then all the even ones. This will not do. In an acceptable list, each item must appear sooner or later as the $n$th entry, for some finite $n$. But in the unacceptable arrangement above, none of the even positive integers are represented in this way. Rather, they appear (so to speak) as entries number $\infty + 1$, $\infty + 2$, and so on.

To make this point perfectly clear we might define an enumeration of a set not as a listing, but as an arrangement in which each member of the set is associated with one of the positive integers 1, 2, 3, \ldots. Actually, a list is such an arrangement. The thing named by the first entry in the list is associated with the positive integer 1, the thing named by the second entry is associated with the positive integer 2, and in general, the thing named by the $n$th entry is associated with the positive integer $n$.

In mathematical parlance, an infinite list determines a function (call it $f$) that takes positive integers as arguments and takes members of the set as values. [Should we have written: ‘call it “$f$”’, rather than ‘call it $f$’? The common practice in mathematical writing is to use special symbols, including even italicized letters of the ordinary alphabet when being used as special symbols, as names for themselves. In case the special symbol happens also to be a name for something else, for instance, a function (as in the present case), we have to rely on context to determine when the symbol is being used one way and when the other. In practice this presents no difficulties.] The value of the function $f$ for the argument $n$ is denoted $f(n)$. This value is simply the thing denoted by the $n$th entry in the list. Thus the list

$$2, 4, 6, 8, \ldots$$

which enumerates the set $E$ of even positive integers determines the function $f$ for which we have

$$f(1) = 2, \quad f(2) = 4, \quad f(3) = 6, \quad f(4) = 8, \quad f(5) = 10, \ldots$$

And conversely, the function $f$ determines the list, except for notation. (The same list would look like this, in Roman numerals: II, IV, VI, VIII, X, \ldots, for instance.) Thus,
we might have defined the function \( f \) first, by saying that for any positive integer \( n \), the value of \( f \) is \( f(n) = 2n \); and then we could have described the list by saying that for each positive integer \( n \), its \( n \)th entry is the decimal representation of the number \( f(n) \), that is, of the number \( 2n \).

Then we may speak of sets as being enumerated by functions, as well as by lists. Instead of enumerating the odd positive integers by the list 1, 3, 5, 7, \ldots, we may enumerate them by the function that assigns to each positive integer \( n \) the value \( 2n - 1 \). And instead of enumerating the set \( P \) of all positive integers by the list 1, 2, 3, 4, \ldots, we may enumerate \( P \) by the function that assigns to each positive integer \( n \) the value \( n \) itself. This is the identity function. If we call it id, we have \( id(n) = n \) for each positive integer \( n \).

If one function enumerates a nonempty set, so does some other; and so, in fact, do infinitely many others. Thus the set of positive integers is enumerated not only by the function id, but also by the function (call it \( g \)) determined by the following list:

\[
2, 1, 4, 3, 6, 5, \ldots
\]

This list is obtained from the list 1, 2, 3, 4, 5, 6, \ldots by interchanging entries in pairs: 1 with 2, 3 with 4, 5 with 6, and so on. This list is a strange but perfectly acceptable enumeration of the set \( P \): every positive integer shows up in it, sooner or later. The corresponding function, \( g \), can be defined as follows:

\[
g(n) = \begin{cases} 
    n + 1 & \text{if } n \text{ is odd} \\
    n - 1 & \text{if } n \text{ is even}.
\end{cases}
\]

This definition is not as neat as the definitions \( f(n) = 2n \) and \( id(n) = n \) of the functions \( f \) and id, but it does the job: It does indeed associate one and only one member of \( P \) with each positive integer \( n \). And the function \( g \) so defined does indeed enumerate \( P \): For each member \( m \) of \( P \) there is a positive integer \( n \) for which we have \( g(n) = m \).

In enumerating a set by listing its members, it is perfectly all right if a member of the set shows up more than once on the list. The requirement is rather that each member show up \textit{at least once}. It does not matter if the list is redundant: All we require is that it be complete. Indeed, a redundant list can always be thinned out to get an irredundant list, since one could go through and erase the entries that repeat earlier entries. It is also perfectly all right if a list has gaps in it, since one could go through and close up the gaps. The requirement is that every element of the set being enumerated be associated with some positive integer, not that every positive integer have an element of the set associated with it. Thus flawless enumerations of the positive integers are given by the following repetitive list:

\[
1, 1, 2, 2, 3, 3, 4, 4, \ldots
\]

and by the following gappy list:

\[
1, -, 2, -, 3, -, 4, -, \ldots
\]

The function corresponding to this last list (call it \( h \)) assigns values corresponding to the first, third, fifth, \ldots entries, but assigns no values corresponding to the gaps
Thus we have \( h(1) = 1 \), but \( h(2) \) is nothing at all, for the function \( h \) is undefined for the argument 2; \( h(3) = 2 \), but \( h(4) \) is undefined; \( h(5) = 3 \), but \( h(6) \) is undefined. And so on: \( h \) is a partial function of positive integers; that is, it is defined only for positive integer arguments, but not for all such arguments. Explicitly, we might define the partial function \( h \) as follows:

\[
h(n) = \frac{n + 1}{2} \quad \text{if } n \text{ is odd.}
\]

Or, to make it clear we haven’t simply forgotten to say what values \( h \) assigns to even positive integers, we might put the definition as follows:

\[
h(n) = \begin{cases} 
(n + 1)/2 & \text{if } n \text{ is odd} \\
\text{undefined} & \text{otherwise.}
\end{cases}
\]

Now the partial function \( h \) is a strange but perfectly acceptable enumeration of the set \( P \) of positive integers.

It would be perverse to choose \( h \) instead of the simple function \( \text{id} \) as an enumeration of \( P \); but other sets are most naturally enumerated by partial functions. Thus, the set \( E \) of even integers is conveniently enumerated by the partial function (call it \( j \)) that agrees with \( \text{id} \) for even arguments, and is undefined for odd arguments:

\[
j(n) = \begin{cases} 
n & \text{if } n \text{ is even} \\
\text{undefined} & \text{otherwise.}
\end{cases}
\]

The corresponding gappy list (in decimal notation) is

\[-, 2, -, 4, -, 6, -, 8, \ldots.\]

Of course the function \( f \) considered earlier, defined by \( f(n) = 2n \) for all positive integers \( n \), was an equally acceptable enumeration of \( E \), corresponding to the gapless list 2, 4, 6, 8, and so on.

Any set \( S \) of positive integers is enumerated quite simply by a partial function \( s \), which is defined as follows:

\[
s(n) = \begin{cases} 
n & \text{if } n \text{ is in the set } S \\
\text{undefined} & \text{otherwise.}
\end{cases}
\]

It will be seen in the next chapter that although every set of positive integers is enumerable, there are sets of others sorts that are not enumerable. To say that a set \( A \) is enumerable is to say that there is a function all of whose arguments are positive integers and all of whose values are members of \( A \), and that each member of \( A \) is a value of this function: For each member \( a \) of \( A \) there is at least one positive integer \( n \) to which the function assigns \( a \) as its value.

Notice that nothing in this definition requires \( A \) to be a set of positive integers or of numbers of any sort. Instead, \( A \) might be a set of people; or a set of linguistic expressions; or a set of sets, as when \( A \) is the set \( \{ P, E, \emptyset \} \). Here \( A \) is a set with three members, each of which is itself a set. One member of \( A \) is the infinite set \( P \) of all positive integers; another member of \( A \) is the infinite set \( E \) of all even positive integers; and the third is the empty set \( \emptyset \). The set \( A \) is certainly enumerable, for example, by the following finite list: \( P, E, \emptyset \). Each entry in this list names a
member of $A$, and every member of $A$ is named sooner or later on this list. This list determines a function (call it $f$), which can be defined by the three statements: $f(1) = P$, $f(2) = E$, $f(3) = \emptyset$. To be precise, $f$ is a partial function of positive integers, being undefined for arguments greater than 3.

In conclusion, let us straighten out our terminology. A function is an assignment of values to arguments. The set of all those arguments to which the function assigns values is called the domain of the function. The set of all those values that the function assigns to its arguments is called the range of the function. In the case of functions whose arguments are positive integers, we distinguish between total functions and partial functions. A total function of positive integers is one whose domain is the whole set $P$ of positive integers. A partial function of positive integers is one whose domain is something less than the whole set $P$. From now on, when we speak simply of a function of positive integers, we should be understood as leaving it open whether the function is total or partial. (This is a departure from the usual terminology, in which function of positive integers always means total function.) A set is enumerable if and only if it is the range of some function of positive integers. We said earlier we wanted to count the empty set $\emptyset$ as enumerable. We therefore have to count as a partial function the empty function $e$ of positive integers that is undefined for all arguments. Its domain and its range are both $\emptyset$.

It will also be important to consider functions with two, three, or more positive integers as arguments, notably the addition function $\text{sum}(m,n) = m + n$ and the multiplication function $\text{prod}(m,n) = m \cdot n$. It is often convenient to think of a two-argument or two-place function on positive integers as a one-argument function on ordered pairs of positive integers, and similarly for many-argument functions. A few more notions pertaining to functions are defined in the first few problems at the end of this chapter. In general, the problems at the end should be read as part of each chapter, even if not all are going to be worked.

1.2 Enumerably Sets

We next illustrate the definition of the preceding section by some important examples. The following sets are enumerable.

1.1 Example (The set of integers). The simplest list is $0, 1, -1, 2, -2, 3, -3, \ldots$. Then if the corresponding function is called $f$, we have $f(1) = 0$, $f(2) = 1$, $f(3) = -1$, $f(4) = 2$, $f(5) = -2$, and so on.

1.2 Example (The set of ordered pairs of positive integers). The enumeration of pairs will be important enough in our later work that it may be well to indicate two different ways of accomplishing it. The first way is this. As a preliminary to enumerating them, we organize them into a rectangular array. We then traverse the array in Cantor’s zig-zag manner indicated in Figure 1.1. This gives us the list

$$(1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), (1, 4), (2, 3), (3, 2), (4, 1), \ldots.$$ 

If we call the function involved here $G$, then we have $G(1) = (1, 1)$, $G(2) = (1, 2)$, $G(3) = (2, 1)$, and so on. The pattern is: First comes the pair the sum of whose entries is 2, then
come the pairs the sum of whose entries is 3, then come the pairs the sum of whose entries is 4, and so on. Within each block of pairs whose entries have the same sum, pairs appear in order of increasing first entry.

As for the second way, we begin with the thought that while an ordinary hotel may have to turn away a prospective guest because all rooms are full, a hotel with an enumerable infinity of rooms would always have room for one more: The new guest could be placed in room 1, and every other guest asked to move over one room. But actually, a little more thought shows that with foresight the hotelier can be prepared to accommodate a busload with an enumerable infinity of new guests each day, without inconveniencing any old guests by making them change rooms. Those who arrive on the first day are placed in every other room, those who arrive on the second day are placed in every other room among those remaining vacant, and so on. To apply this thought to enumerating pairs, let us use up every other place in listing the pairs \((1, n)\), every other place then remaining in listing the pairs \((2, n)\), every other place then remaining in listing the pairs \((3, n)\), and so on. The result will look like this:

\[
(1, 1), (2, 1), (1, 2), (3, 1), (1, 3), (2, 2), (1, 4), (4, 1), (1, 5), (2, 3), \ldots
\]

If we call the function involved here \(g\), then \(g(1) = (1, 1)\), \(g(2) = (2, 1)\), \(g(3) = (1, 2)\), and so on.

Given a function \(f\) enumerating the pairs of positive integers, such as \(G\) or \(g\) above, an \(a\) such that \(f(a) = (m, n)\) may be called a code number for the pair \((m, n)\). Applying the function \(f\) may be called decoding, while going the opposite way, from the pair to a code for it, may be called encoding. It is actually possible to derive mathematical formulas for the encoding functions \(J\) and \(j\) that go with the decoding functions \(G\) and \(g\) above. (Possible, but not necessary: What we have said so far more than suffices as a proof that the set of pairs is enumerable.)

Let us take first \(J\). We want \(J(m, n)\) to be the number \(p\) such that \(G(p) = (m, n)\), which is to say the place \(p\) where the pair \((m, n)\) comes in the enumeration corresponding to \(G\). Before we arrive at the pair \((m, n)\), we will have to pass the pair whose entries sum to 2, the two pairs whose entries sum to 3, the three pairs whose entries sum to 4, and so on, up through the \(m + n - 2\) pairs whose entries sum to \(m + n - 1\).
The pair \((m, n)\) will appear in the \(m\)th place after all of these pairs. So the position of the pair \((m, n)\) will be given by

\[
[1 + 2 + \cdots + (m + n - 2)] + m.
\]

At this point we recall the formula for the sum of the first \(k\) positive integers:

\[
1 + 2 + \cdots + k = \frac{k(k + 1)}{2}.
\]

(Never mind, for the moment, where this formula comes from. Its derivation will be recalled in a later chapter.) So the position of the pair \((m, n)\) will be given by

\[
(m + n - 2)(m + n - 1)/2 + m.
\]

This simplifies to

\[
J(m, n) = (m^2 + 2mn + n^2 - m - 3n + 2)/2.
\]

For instance, the pair \((3, 2)\) should come in the place

\[
(3^2 + 2 \cdot 3 \cdot 2 + 2^2 - 3 - 3 \cdot 2 + 2)/2 = (9 + 12 + 4 - 3 - 6 + 2)/2 = 18/2 = 9
\]

as indeed it can be seen (looking back at the enumeration as displayed above) that it does: \(G(9) = (3, 2)\).

Turning now to \(j\), we find matters a bit simpler. The pairs with first entry 1 will appear in the places whose numbers are odd, with \((1, n)\) in place \(2n - 1\). The pairs with first entry 2 will appear in the places whose numbers are twice an odd number, with \((2, n)\) in place \(2(2n - 1)\). The pairs with first entry 3 will appear in the places whose numbers are four times an odd number, with \((3, n)\) in place \(4(2n - 1)\). In general, in terms of the powers of two \((2^0 = 1, 2^1 = 2, 2^2 = 4, \text{and so on})\), \((m, n)\) will appear in place \(j(m, n) = 2^{m-1}(2n - 1)\). Thus \((3, 2)\) should come in the place \(2^{3-1}(2 \cdot 2 - 1) = 2^2(4 - 1) = 4 \cdot 3 = 12\), as indeed it does: \(g(12) = (3, 2)\).

The series of examples to follow shows how more and more complicated objects can be coded by positive integers. Readers may wish to try to find proofs of their own before reading ours; and for this reason we give the statements of all the examples first, and collect all the proofs afterwards. As we saw already with Example 1.2, several equally good codings may be possible.

1.3 Example. The set of positive rational numbers

1.4 Example. The set of rational numbers

1.5 Example. The set of ordered triples of positive integers

1.6 Example. The set of ordered \(k\)-tuples of positive integers, for any fixed \(k\)

1.7 Example. The set of finite sequences of positive integers less than 10

1.8 Example. The set of finite sequences of positive integers less than \(b\), for any fixed \(b\)

1.9 Example. The set of finite sequences of positive integers

1.10 Example. The set of finite sets of positive integers
1.11 **Example.** Any subset of an enumerable set

1.12 **Example.** The union of any two enumerable sets

1.13 **Example.** The set of finite strings from a finite or enumerable alphabet of symbols

**Proofs**

**Example 1.3.** A positive rational number is a number that can be expressed as a ratio of positive integers, that is, in the form \( m/n \) where \( m \) and \( n \) are positive integers. Therefore we can get an enumeration of all positive rational numbers by starting with our enumeration of all pairs of positive integers and replacing the pair \((m, n)\) by the rational number \(m/n\). This gives us the list

\[
1/1, 1/2, 2/1, 1/3, 2/2, 3/1, 1/4, 2/3, 3/2, 4/1, 1/5, 2/4, 3/3, 4/2, 5/1, 1/6, \ldots
\]

or, simplified,

\[
1, 1/2, 2, 1/3, 1, 3, 1/4, 2/3, 3/2, 4, 1/5, 1/2, 1, 2, 5/1, 1/6, \ldots.
\]

Every positive rational number in fact appears infinitely often, since for instance

\[
1/1 = 2/2 = 3/3 = \ldots \quad \text{and} \quad 1/2 = 2/4 = \ldots \quad \text{and} \quad 2/1 = 4/2 = \ldots
\]

and similarly for every other rational number. But that is all right: our definition of enumerability permits repetitions.

**Example 1.4.** We combine the ideas of Examples 1.1 and 1.3. You know from Example 1.3 how to arrange the positive rationals in a single infinite list. Write a zero in front of this list, and then write the positive rationals, backwards and with minus signs in front of them, in front of that. You now have

\[
\ldots, -1/3, -2, -1/2, -1, 0, 1, 1/2, 2, 1/3, \ldots
\]

Finally, use the method of Example 1.1 to turn this into a proper list:

\[
0, 1, -1, 1/2, -1/2, 2, -2, 1/3, -1/3, \ldots
\]

**Example 1.5.** In Example 1.2 we have given two ways of listing all pairs of positive integers. For definiteness, let us work here with the first of these:

\[
(1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), \ldots
\]

Now go through this list, and in each pair replace the second entry or component \( n \) with the pair that appears in the \( n \)th place on this very list. In other words, replace each 1 that appears in the second place of a pair by \((1, 1)\), each 2 by \((1, 2)\), and so on. This gives the list

\[
(1, (1, 1)), (1, (1, 2)), (2, (1, 1)), (1, (2, 1)), (2, (1, 2)), (3, (1, 1)), \ldots
\]

and that gives a list of triples

\[
(1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 1, 2), (3, 1, 1), \ldots
\]

In terms of functions, this enumeration may be described as follows. The original enumeration of pairs corresponds to a function associating to each positive integer \( n \)
a pair \( G(n) = (K(n), L(n)) \) of positive integers. The enumeration of triples we have just defined corresponds to assigning to each positive integer \( n \) instead the triple 
\[
(K(n), K(L(n)), L(L(n))).
\]
We do not miss any triples \((p, q, r)\) in this way, because there will always be an \( m = J(q, r) \) such that \((K(m), L(m)) = (q, r)\), and then there will be an \( n = J(p, m) \) such that \((K(n), L(n)) = (p, m)\), and the triple associated with this \( n \) will be precisely \((p, q, r)\).

**Example 1.6.** The method by which we have just obtained an enumeration of triples from an enumeration of pairs will give us an enumeration of quadruples from an enumeration of triples. Go back to the original enumeration pairs, and replace each second entry \( n \) by the triple that appears in the \( n \)th place in the enumeration of triples, to get a quadruple. The first few quadruples on the list will be
\[
(1, 1, 1, 1), (1, 1, 1, 2), (2, 1, 1, 1), (1, 2, 1, 1), (2, 1, 1, 2), \ldots.
\]
Obviously we can go on from here to quintuples, sextuples, or \( k \)-tuples for any fixed \( k \).

**Example 1.7.** A finite sequence whose entries are all positive integers less than 10, such as \((1, 2, 3)\), can be read as an ordinary decimal or base-10 numeral 123. The number this numeral denotes, one hundred twenty-three, could then be taken as a code number for the given sequence. Actually, for later purposes it proves convenient to modify this procedure slightly and write the sequence in reverse before reading it as a numeral. Thus \((1, 2, 3)\) would be coded by 321, and 123 would code \((3, 2, 1)\). In general, a sequence
\[
s = (a_0, a_1, a_2, \ldots, a_k)
\]
would be coded by
\[
a_0 + 10a_1 + 100a_2 + \cdots + 10^k a_k
\]
which is the number that the decimal numeral \( a_k \cdots a_2 a_1 a_0 \) represents. Also, it will be convenient henceforth to call the initial entry of a finite sequence the 0th entry, the next entry the 1st, and so on. To decode and obtain the \( i \)th entry of the sequence coded by \( n \), we take the quotient on dividing by \( 10^i \), and then the remainder on dividing by 10. For instance, to find the 5th entry of the sequence coded by 123 456 789, we divide by \( 10^5 \) to obtain the quotient 1234, and then divide by 10 to obtain the remainder 4.

**Example 1.8.** We use a decimal, or base-10, system ultimately because human beings typically have 10 fingers, and counting began with counting on fingers. A similar base-\( b \) system is possible for any \( b > 1 \). For a binary, or base-2, system only the ciphers 0 and 1 would be used, with \( a_k \cdots a_2 a_1 a_0 \) representing
\[
a_0 + 2a_1 + 4a_2 + \cdots + 2^k a_k.
\]
So, for instance, 1001 would represent \( 1 + 2^3 = 1 + 8 = 9 \). For a duodecimal, or base-12, system, two additional ciphers, perhaps \(*\) and \# as on a telephone, would be needed for ten and eleven. Then, for instance, 1*# would represent \( 1 + 12 \cdot 10 + 144 \cdot 1 = 275 \). If we applied the idea of the previous problem using base 12 instead
of base 10, we could code finite sequences of positive integers less than 12, and not just finite sequences of positive integers less than 10. More generally, we can code a finite sequence

\[ s = (a_0, a_1, a_2, \ldots, a_k) \]

of positive integers less than \( b \) by

\[ a_0 + ba_1 + b^2a_2 + \cdots + b^ka_k. \]

To obtain the \( i \)th entry of the sequence coded by \( n \), we take the quotient on dividing by \( b^i \) and then the remainder on dividing by \( b \). For example, when working with base 12, to obtain the 5th entry of the sequence coded by 123456789, we divide 123456789 by 125 to get the quotient 496. Now divide by 12 to get remainder 4. In general, working with base \( b \), the \( i \)th entry—counting the initial one as the 0th—of the sequence coded by \((b, n)\) will be

\[ \text{entry}(i, n) = \text{rem}(\text{quo}(n, b^i), b) \]

where \( \text{quo}(x, y) \) and \( \text{rem}(x, y) \) are the quotient and remainder on dividing \( x \) by \( y \).

**Example 1.9.** Coding finite sequences will be important enough in our later work that it will be appropriate to consider several different ways of accomplishing this task. Example 1.6 showed that we can code sequences whose entries may be of any size but that are of fixed length. What we now want is an enumeration of all finite sequences—pairs, triples, quadruples, and so on—in a single list; and for good measure, let us include the 1-tuples or 1-term sequences \((1), (2), (3), \ldots \) as well. A first method, based on Example 1.6, is as follows. Let \( G_1(n) \) be the 1-term sequence \((n)\). Let \( G_2 = G \), the function enumerating all 2-tuples or pairs from Example 1.2. Let \( G_3 \) be the function enumerating all triples as in Example 1.5. Let \( G_4, G_5, \ldots, \) be the enumerations of triples, quadruples, and so on, from Example 1.6. We can get a coding of all finite sequences by pairs of positive integers by letting any sequence \( s \) of length \( k \) be coded by the pair \((k, a)\) where \( G_k(a) = s \). Since pairs of positive integers can be coded by single numbers, we indirectly get a coding of sequences of numbers. Another way to describe what is going on here is as follows. We go back to our original listing of pairs, and replace the pair \((k, a)\) by the \( a \)th item on the list of \( k \)-tuples. Thus \((1, 1)\) would be replaced by the first item \((1)\) on the list of 1-tuples \((1), (2), (3), \ldots \); while \((1, 2)\) would be replaced by the second item \((2)\) on the same list; whereas \((2, 1)\) would be replaced by the first item \((1, 1)\) on the list of all 2-tuples or pairs; and so on. This gives us the list

\[
(1), (2), (1, 1), (3), (1, 2), (1, 1, 1), (4), (2, 1), (1, 1, 2), (1, 1, 1, 1), \ldots
\]

(If we wish to include also the 0-tuple or empty sequence \((\, )\), which we may take to be simply the empty set \( \emptyset \), we can stick it in at the head of the list, in what we may think of as the 0th place.)

Example 1.8 showed that we can code sequences of any length whose entries are less than some fixed bound, but what we now want to do is show how to code sequences of any length whose entries may be of any size. A second method, based
on Example 1.8, is to begin by coding sequences by pairs of positive integers. We take a sequence

\[ s = (a_0, a_1, a_2, \ldots, a_k) \]

to be coded by any pair \((b, n)\) such that all \(a_i\) are less than \(b\), and \(n\) codes \(s\) in the sense that

\[ n = a_0 + b \cdot a_1 + b^2 a_2 + \cdots + b^k a_k. \]

Thus \((10, 275)\) would code \((5, 7, 2)\), since \(275 = 5 + 7 \cdot 10 + 2 \cdot 10^2\), while \((12, 275)\) would code \((11, 10, 1)\), since \(275 = 11 + 10 \cdot 12 + 1 \cdot 12^2\). Each sequence would have many codes, since for instance \((10, 234)\) and \((12, 328)\) would equally code \((4, 3, 2)\), because \(4 + 3 \cdot 10 + 2 \cdot 10^2 = 234\) and \(4 + 3 \cdot 12 + 2 \cdot 12^2 = 328\). As with the previous method, since pairs of positive integers can be coded by single numbers, we indirectly get a coding of sequences of numbers.

A third, and totally different, approach is possible, based on the fact—proved in Euclid’s *Elements of Geometry*—that every positive integer can be written in one and only one way as a product of powers of larger and larger primes, a representation called its *prime decomposition*. This fact enables us to code a sequence \(s = (i, j, k, m, n, \ldots)\) by the number \(2^i 3^j 5^k 7^m 11^n \ldots\). Thus the code number for the sequence \((3, 1, 2)\) is \(2^3 3^1 5^2 = 8 \cdot 3 \cdot 25 = 600\).

*Example 1.10.* It is easy to get an enumeration of finite sets from an enumeration of finite sequences. Using the first method in Example 1.8, for instance, we get the following enumeration of sets:

\[
\{\}, \{1\}, \{1, 1\}, \{2\}, \{1, 2\}, \{1, 1, 1\}, \{3\}, \{1, 2, 1\}, \{1, 1, 2\}, \{1, 1, 1, 1\}, \ldots.
\]

The set \(\{1, 1\}\) whose only elements are 1 and 1 is just the set \(\{1\}\) whose only element is 1, and similarly in other cases, so this list can be simplified to look like this:

\[
\{\}, \{1\}, \{2\}, \{1, 3\}, \{1, 1, 2\}, \{1, 1, 1, 4\}, \{1, 1, 2, 1\}, \{1, 1, 1, 1, 5\}, \ldots.
\]

The repetitions do not matter.

*Example 1.11.* Given any enumerable set \(A\) and a listing of the elements of \(A\):

\[ a_1, a_2, a_3, \ldots \]

we easily obtain a gappy listing of the elements of any subset \(B\) of \(A\) simply by erasing any entry in the list that does not belong to \(B\), leaving a gap.

*Example 1.12.* Let \(A\) and \(B\) be enumerable sets, and consider listings of their elements:

\[ a_1, a_2, a_3, \ldots \quad b_1, b_2, b_3, \ldots \]

Imitating the *shuffling* idea of Example 1.1, we obtain the following listing of the elements of the union \(A \cup B\) (the set whose elements are all and only those items that are elements either of \(A\) or of \(B\) or of both):

\[ a_1, b_1, a_2, b_2, a_3, b_3, \ldots \]
If the intersection $A \cap B$ (the set whose elements of both $A$ and $B$) is not empty, then there will be redundancies on this list: If $a_m = b_n$, then that element will appear both at place $2m - 1$ and at place $2n$, but this does not matter.

**Example 1.13.** Given an ‘alphabet’ of any finite number, or even an enumerable infinity, of symbols $S_1, S_2, S_3, \ldots$ we can take as a code number for any finite string $S_{a_0}S_{a_1}S_{a_2} \cdots S_{a_k}$ the code number for the finite sequence of positive integers $(a_1, a_2, a_3, \ldots, a_k)$ under any of the methods of coding considered in Example 1.9. (We are usually going to use the third method.) For instance, with the ordinary alphabet of 26 symbols letters $S_1 = \text{‘A’}, S_2 = \text{‘B’},$ and so on, the string or word ‘CAB’ would be coded by the code for $(3, 1, 2)$, which (on the third method of Example 1.9) would be $2^3 \cdot 3 \cdot 5^2 = 600$.

**Problems**

1.1 A (total or partial) function $f$ from a set $A$ to a set $B$ is an assignment for (some or all) elements $a$ of $A$ of an associated element $f(a)$ of $B$. If $f(a)$ is defined for **every** element $a$ of $A$, then the function $f$ is called total. If every element $b$ of $B$ is assigned to some element $a$ of $A$, then the function $f$ is said to be onto. If no element $b$ of $B$ is assigned to more than one element $a$ of $A$, then the function $f$ is said to be one-to-one. The inverse function $f^{-1}$ from $B$ to $A$ is defined by letting $f^{-1}(b)$ be the one and only $a$ such that $f(a) = b$, if any such $a$ exists; $f^{-1}(b)$ is undefined if there is no $a$ with $f(a) = b$ or more than one such $a$. Show that if $f$ is a one-to-one function and $f^{-1}$ its inverse function, then $f^{-1}$ is total if and only if $f$ is onto, and conversely, $f^{-1}$ is onto if and only if $f$ is total.

1.2 Let $f$ be a function from a set $A$ to a set $B$, and $g$ a function from the set $B$ to a set $C$. The composite function $h = gf$ from $A$ to $C$ is defined by $h(a) = g(f(a))$. Show that:

(a) If $f$ and $g$ are both total, then so is $gf$.
(b) If $f$ and $g$ are both onto, then so is $gf$.
(c) If $f$ and $g$ are both one-to-one, then so is $gf$.

1.3 A correspondence between sets $A$ and $B$ is a one-to-one total function from $A$ onto $B$. Two sets $A$ and $B$ are said to be equinumerous if and only if there is a correspondence between $A$ and $B$. Show that equinumerosity has the following properties:

(a) Any set $A$ is equinumerous with itself.
(b) If $A$ is equinumerous with $B$, then $B$ is equinumerous with $A$.
(c) If $A$ is equinumerous with $B$ and $B$ is equinumerous with $C$, then $A$ is equinumerous with $C$.

1.4 A set $A$ has $n$ elements, where $n$ is a positive integer, if it is equinumerous with the set of positive integers up to $n$, so that its elements can be listed as $a_1, a_2, \ldots, a_n$. A nonempty set $A$ is finite if it has $n$ elements for some positive integer $n$. Show that any enumerable set is either finite or equinumerous with
the set of all positive integers. (In other words, given an enumeration, which is to say a function from the set of positive integers onto a set $A$, show that if $A$ is not finite, then there is a correspondence, which is to say a one-to-one, total function, from the set of positive integers onto $A$.)

1.5 Show that the set of all finite subsets of an enumerable set is enumerable.

1.6 Show that the following sets are equinumerous:

(a) The set of rational numbers with denominator a power of two (when written in lowest terms), that is, the set of rational numbers $\pm m/n$ where $n = 1$ or 2 or 4 or 8 or some higher power of 2.

(b) The set of those sets of positive integers that are either finite or cofinite, where a set $S$ of positive integers is cofinite if the set of all positive integers $n$ that are not elements of $S$ is finite.

1.7 Let $A = \{ A_1, A_2, A_3, \ldots \}$ be an enumerable family of sets, and suppose that each $A_i$ for $i = 1, 2, 3$, and so on, is enumerable. Let $\cup A$ be the union of the family $A$, that is, the set whose elements are precisely the elements of the elements of $A$. Is $\cup A$ enumerable?
In the preceding chapter we introduced the distinction between enumerable and nonenumerable sets, and gave many examples of enumerable sets. In this short chapter we give examples of nonenumerable sets. We first prove the existence of such sets, and then look a little more closely at the method, called diagonalization, used in this proof.

Not all sets are enumerable: some are too big. For example, consider the set of all sets of positive integers. This set (call it \( P^* \)) contains, as a member, each finite and each infinite set of positive integers: the empty set \( \emptyset \), the set \( P \) of all positive integers, and every set between these two extremes. Then we have the following celebrated result.

2.1 Theorem (Cantor’s Theorem). The set of all sets of positive integers is not enumerable.

Proof: We give a method that can be applied to any list \( L \) of sets of positive integers in order to discover a set \( \Delta(L) \) of positive integers which is not named in the list. If you then try to repair the defect by adding \( \Delta(L) \) to the list as a new first member, the same method, applied to the augmented list \( L^* \) will yield a different set \( \Delta(L^*) \) that is likewise not on the augmented list.

The method is this. Confronted with any infinite list \( L \)

\[
S_1, S_2, S_3, \ldots
\]

of sets of positive integers, we define a set \( \Delta(L) \) as follows:

\((*)\) For each positive integer \( n \), \( n \) is in \( \Delta(L) \) if and only if \( n \) is not in \( S_n \).

It should be clear that this genuinely defines a set \( \Delta(L) \); for, given any positive integer \( n \), we can tell whether \( n \) is in \( \Delta(L) \) if we can tell whether \( n \) is in the \( n \)th set in the list \( L \). Thus, if \( S_3 \) happens to be the set \( E \) of even positive integers, the number 3 is not in \( S_3 \) and therefore it is in \( \Delta(L) \). As the notation \( \Delta(L) \) indicates, the composition of the set \( \Delta(L) \) depends on the composition of the list \( L \), so that different lists \( L \) may yield different sets \( \Delta(L) \).

To show that the set \( \Delta(L) \) that this method yields is never in the given list \( L \), we argue by *reductio ad absurdum*: we suppose that \( \Delta(L) \) does appear somewhere in list \( L \), say as entry number \( m \), and deduce a contradiction, thus showing that the
supposition must be false. Here we go. *Supposition:* For some positive integer $m$,

$$S_m = \Delta(L).$$

[Thus, if 127 is such an $m$, we are supposing that $\Delta(L)$ and $S_{127}$ are the same set under different names: we are supposing that a positive integer belongs to $\Delta(L)$ if and only if it belongs to the 127th set in list $L$.] To deduce a contradiction from this assumption we apply definition (*) to the particular positive integer $m$: with $n = m$, (*) tells us that

$$m \text{ is in } \Delta(L) \text{ if and only if } m \text{ is not in } S_m.$$  

Now a contradiction follows from our supposition: if $S_m$ and $\Delta(L)$ are one and the same set we have

$$m \text{ is in } \Delta(L) \text{ if and only if } m \text{ is in } S_m.$$  

Since this is a flat self-contradiction, our supposition must be false. For no positive integer $m$ do we have $S_m = \Delta(L)$. In other words, the set $\Delta(L)$ is named nowhere in list $L$.

So the method works. Applied to any list of sets of positive integers it yields a set of positive integers which was not in the list. Then no list enumerates all sets of positive integers: the set $P^*$ of all such sets is not enumerable. This completes the proof.

Note that results to which we might wish to refer back later are given reference numbers 1.1, 1.2, ... consecutively through the chapter, to make them easy to locate. Different words, however, are used for different kinds of results. The most important general results are dignified with the title of ‘theorem’. Lesser results are called ‘lemmas’ if they are steps on the way to a theorem, ‘corollaries’ if they follow directly upon some theorem, and ‘propositions’ if they are free-standing. In contrast to all these, ‘examples’ are particular rather than general. The most celebrated of the theorems have more or less traditional names, given in parentheses. The fact that 2.1 has been labelled ‘Cantor’s theorem’ is an indication that it is a famous result. The reason is not—we hope the reader will agree!—that its proof is especially difficult, but that the *method* of the proof (*diagonalization*) was an important innovation. In fact, it is so important that it will be well to look at the proof again from a slightly different point of view, which allows the entries in the list $L$ to be more readily visualized.

Accordingly, we think of the sets $S_1$, $S_2$, ... as represented by functions $s_1$, $s_2$, ... of positive integers that take the numbers 0 and 1 as values. The relationship between the set $S_n$ and the corresponding function $s_n$ is simply this: for each positive integer $p$ we have

$$s_n(p) = \begin{cases} 1 & \text{if } p \text{ is in } S_n \\ 0 & \text{if } p \text{ is not in } S_n. \end{cases}$$

Then the list can be visualized as an infinite rectangular array of zeros and ones, in which the $n$th row represents the function $s_n$ and thus represents the set $S_n$. That is,
Figure 2-1. A list as a rectangular array.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$s_1(1)$</td>
<td>$s_1(2)$</td>
<td>$s_1(3)$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_2(1)$</td>
<td>$s_2(2)$</td>
<td>$s_2(3)$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$s_3(1)$</td>
<td>$s_3(2)$</td>
<td>$s_3(3)$</td>
</tr>
<tr>
<td>$s_4$</td>
<td>$s_4(1)$</td>
<td>$s_4(2)$</td>
<td>$s_4(3)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The entries in the diagonal of the array (upper left to lower right) form a sequence of zeros and ones:

$s_1(1) s_2(2) s_3(3) s_4(4) \ldots$

This sequence of zeros and ones (the _diagonal sequence_ ) determines a set of positive integers (the _diagonal set_ ). The diagonal set may well be among those listed in $L$. In other words, there may well be a positive integer $d$ such that the set $S_d$ is none other than our diagonal set. The sequence of zeros and ones in the $d$th row of Figure 2-1 would then agree with the diagonal sequence entry by entry:

$s_d(1) = s_1(1), \quad s_d(2) = s_2(2), \quad s_d(3) = s_3(3), \ldots$

That is as may be: the diagonal set may or may not appear in the list $L$, depending on the detailed makeup of the list. What we want is a set we can rely upon not to appear in $L$, no matter how $L$ is composed. Such a set lies near to hand: it is the _antidiagonal set_, which consists of the positive integers not in the diagonal set. The corresponding _antidiagonal sequence_ is obtained by changing zeros to ones and ones to zeros in the diagonal sequence. We may think of this transformation as a matter of subtracting each member of the diagonal sequence from 1: we write the antidiagonal sequence as

$1 - s_1(1), \ 1 - s_2(2), \ 1 - s_3(3), \ 1 - s_4(4), \ldots$

This sequence can be relied upon not to appear as a row in Figure 2-1, for if it did appear—say, as the $m$th row—we should have

$s_m(1) = 1 - s_1(1), \quad s_m(2) = 1 - s_2(2), \ldots, \quad s_m(m) = 1 - s_m(m), \ldots$

But the $m$th of these equations cannot hold. [Proof: $s_m(m)$ must be zero or one. If zero, the $m$th equation says that $0 = 1$. If one, the $m$th equation says that $1 = 0$.] Then the antidiagonal sequence differs from every row of our array, and so the antidiagonal set differs from every set in our list $L$. This is no news, for the antidiagonal set is simply the set $\Delta(L)$. We have merely repeated with a diagram—Figure 2-1—our proof that $\Delta(L)$ appears nowhere in the list $L$.

Of course, it is rather strange to say that the members of an infinite set ‘can be arranged’ in a single list. By whom? Certainly not by any human being, for nobody
has that much time or paper; and similar restrictions apply to machines. In fact, to call a set enumerable is simply to say that it is the range of some total or partial function of positive integers. Thus, the set \(E\) of even positive integers is enumerable because there are functions of positive integers that have \(E\) as their range. (We had two examples of such functions earlier.) Any such function can then be thought of as a program that a superhuman enumerator can follow in order to arrange the members of the set in a single list. More explicitly, the program (the set of instructions) is: ‘Start counting from 1, and never stop. As you reach each number \(n\), write a name of \(f(n)\) in your list. [Where \(f(n)\) is undefined, leave the \(n\)th position blank.]’ But there is no need to refer to the list, or to a superhuman enumerator: anything we need to say about enumerability can be said in terms of the functions themselves; for example, to say that the set \(P^*\) is not enumerable is simply to deny the existence of any function of positive integers which has \(P^*\) as its range.

Vivid talk of lists and superhuman enumerators may still aid the imagination, but in such terms the theory of enumerability and diagonalization appears as a chapter in mathematical theology. To avoid treading on any living toes we might put the whole thing in a classical Greek setting: Cantor proved that there are sets which even Zeus cannot enumerate, no matter how fast he works, or how long (even, infinitely long).

If a set is enumerable, Zeus can enumerate it in one second by writing out an infinite list faster and faster. He spends \(1/2\) second writing the first entry in the list; \(1/4\) second writing the second entry; \(1/8\) second writing the third; and in general, he writes each entry in half the time he spent on its predecessor. At no point during the one-second interval has he written out the whole list, but when one second has passed, the list is complete. On a time scale in which the marked divisions are sixteenths of a second, the process can be represented as in Figure 2-2.

```
0  1/16  2/16  3/16  4/16  5/16  6/16  7/16  8/16  9/16  10/16  11/16  12/16  13/16  14/16  15/16  1
```

Zeus makes 1st entry 2nd entry 3rd entry &c.

Figure 2-2. Completing an infinite process in finite time.

To speak of writing out an infinite list (for example, of all the positive integers, in decimal notation) is to speak of such an enumerator either working faster and faster as above, or taking all of infinite time to complete the list (making one entry per second, perhaps). Indeed, Zeus could write out an infinite sequence of infinite lists if he chose to, taking only one second to complete the job. He could simply allocate the first half second to the business of writing out the first infinite list (\(1/4\) second for the first entry, \(1/8\) second for the next, and so on); he could then write out the whole second list in the following quarter second (\(1/8\) for the first entry, \(1/16\) second for the next, and so on); and in general, he could write out each subsequent list in just half the time he spent on its predecessor, so that after one second had passed he would have written out every entry in every list, in order. But the result does not count as a
single infinite list, in our sense of the term. In our sort of list, each entry must come some finite number of places after the first.

As we use the term ‘list’, Zeus has not produced a list by writing infinitely many infinite lists one after another. But he could perfectly well produce a genuine list which exhausts the entries in all the lists, by using some such device as we used in the preceding chapter to enumerate the positive rational numbers. Nevertheless, Cantor’s diagonal argument shows that neither this nor any more ingenious device is available, even to a god, for arranging all the sets of positive integers into a single infinite list. Such a list would be as much an impossibility as a round square: the impossibility of enumerating all the sets of positive integers is as absolute as the impossibility of drawing a round square, even for Zeus.

Once we have one example of a nonenumerable set, we get others.

2.2 Corollary. The set of real numbers is not enumerable.

Proof: If $\xi$ is a real number and $0 < \xi < 1$, then $\xi$ has a decimal expansion $x_1x_2x_3\ldots$ where each $x_i$ is one of the cyphers 0–9. Some numbers have two decimal expansions, since for instance .2999... = .3000...; so if there is a choice, choose the one with the 0s rather than the one with the 9s. Then associate to $\xi$ the set of all positive integers $n$ such that a 1 appears in the $n$th place in this expansion. Every set of positive integers is associated to some real number (the sum of $10^{-n}$ for all $n$ in the set), and so an enumeration of the real numbers would immediately give rise to an enumeration of the sets of positive integers, which cannot exist, by the preceding theorem.

Problems

2.1 Show that the set of all subsets of an infinite enumerable set is nonenumerable.

2.2 Show that if for some or all of the finite strings from a given finite or enumerable alphabet we associate to the string a total or partial function from positive integers to positive integers, then there is some total function on positive integers taking only the values 1 and 2 that is not associated with any string.

2.3 In mathematics, the real numbers are often identified with the points on a line. Show that the set of real numbers, or equivalently, the set of points on the line, is equinumerous with the set of points on the semicircle indicated in Figure 2-3.

Figure 2-3. Interval, semicircle, and line.
2.4 Show that the set of real numbers $\xi$ with $0 < \xi < 1$, or equivalently, the set of points on the interval shown in Figure 2-3, is equinumerous with the set of points on the semicircle.

2.5 Show that the set of real numbers $\xi$ with $0 < \xi < 1$ is equinumerous with the set of all real numbers.

2.6 A real number $x$ is called algebraic if it is a solution to some equation of the form

$$c_dx^d + c_{d-1}x^{d-1} + c_{d-2}x^{d-2} + \cdots + c_2x^2 + c_1x + c_0 = 0$$

where the $c_i$ are rational numbers and $c_d \neq 0$. For instance, for any rational number $r$, the number $r$ itself is algebraic, since it is the solution to $x - r = 0$; and the square root $\sqrt{r}$ of $r$ is algebraic, since it is a solution to $x^2 - r = 0$.

(a) Use the fact from algebra that an equation like the one displayed has at most $d$ solutions to show that every algebraic number can be described by a finite string of symbols from an ordinary keyboard.

(b) A real number that is not algebraic is called transcendental. Prove that transcendental numbers exist.

2.7 Each real number $\xi$ with $0 < \xi < 1$ has a binary representation $0 \cdot x_1x_2x_3 \ldots$ where each $x_i$ is a digit 0 or 1, and the successive places represent halves, quarters, eighths, and so on. Show that the set of real numbers, $\xi$ with $0 < \xi < 1$ and $\xi$ not a rational number with denominator a power of two, is equinumerous with the set of those sets of positive integers that are neither finite nor cofinite.

2.8 Show that if $A$ is equinumerous with $C$ and $B$ is equinumerous with $D$, and the intersections $A \cap B$ and $C \cap D$ are empty, then the unions $A \cup B$ and $C \cup D$ are equinumerous.

2.9 Show that the set of real numbers $\xi$ with $0 < \xi < 1$ (and hence by an earlier problem the set of all real numbers) is equinumerous with the set of all sets of positive integers.

2.10 Show that the following sets are equinumerous:

(a) the set of all pairs of sets of positive integers

(b) the set of all sets of pairs of positive integers

(c) the set of all sets of positive integers.

2.11 Show that the set of points on a line is equinumerous with the set of points on a plane.

2.12 Show that the set of points on a line is equinumerous with the set of points in space.

2.13 (Richard’s paradox) What (if anything) is wrong with the following argument?

The set of all finite strings of symbols from the alphabet, including the space, capital letters, and punctuation marks, is enumerable; and for definiteness let us use the specific enumeration of finite strings based on prime decomposition. Some strings amount to definitions in English of sets of positive integers and others do not. Strike out the ones that do not, and we are left with an enumeration of all definitions in English of sets of positive integers, or, replacing each definition by the set it defines, an enumeration of all sets of positive integers that have definitions in English. Since some sets have more than one definition, there will be redundancies in this enumeration. 
of sets. Strike them out to obtain an irredundant enumeration of all sets of positive integers that have definitions in English. Now consider the set of positive integers defined by the condition that a positive integer \( n \) is to belong to the set if and only if it does not belong to the \( n \)th set in the irredundant enumeration just described.

This set does not appear in that enumeration. For it cannot appear at the \( n \)th place for any \( n \), since there is a positive integer, namely \( n \) itself, that belongs to this set if and only if it does not belong to the \( n \)th set in the enumeration. Since this set does not appear in our enumeration, it cannot have a definition in English. And yet it does have a definition in English, and in fact we have just given such a definition in the preceding paragraph.
A function is effectively computable if there are definite, explicit rules by following which one could in principle compute its value for any given arguments. This notion will be further explained below, but even after further explanation it remains an intuitive notion. In this chapter we pursue the analysis of computability by introducing a rigorously defined notion of a Turing-computable function. It will be obvious from the definition that Turing-computable functions are effectively computable. The hypothesis that, conversely, every effectively computable function is Turing computable is known as Turing’s thesis. This thesis is not obvious, nor can it be rigorously proved (since the notion of effective computability is an intuitive and not a rigorously defined one), but an enormous amount of evidence has been accumulated for it. A small part of that evidence will be presented in this chapter, with more in chapters to come. We first introduce the notion of Turing machine, give examples, and then present the official definition of what it is for a function to be computable by a Turing machine, or Turing computable.

A superhuman being, like Zeus of the preceding chapter, could perhaps write out the whole table of values of a one-place function on positive integers, by writing each entry twice as fast as the one before; but for a human being, completing an infinite process of this kind is impossible in principle. Fortunately, for human purposes we generally do not need the whole table of values of a function \( f \), but only need the values one at a time, so to speak: given some argument \( n \), we need the value \( f(n) \). If it is possible to produce the value \( f(n) \) of the function \( f \) for argument \( n \) whenever such a value is needed, then that is almost as good as having the whole table of values written out in advance.

A function \( f \) from positive integers to positive integers is called effectively computable if a list of instructions can be given that in principle make it possible to determine the value \( f(n) \) for any argument \( n \). (This notion extends in an obvious way to two-place and many-place functions.) The instructions must be completely definite and explicit. They should tell you at each step what to do, not tell you to go ask someone else what to do, or to figure out for yourself what to do: the instructions should require no external sources of information, and should require no ingenuity to execute, so that one might hope to automate the process of applying the rules, and have it performed by some mechanical device.

There remains the fact that for all but a finite number of values of \( n \), it will be infeasible in practice for any human being, or any mechanical device, actually to carry
out the computation: in principle it could be completed in a finite amount of time if we stayed in good health so long, or the machine stayed in working order so long; but in practice we will die, or the machine will collapse, long before the process is complete. (There is also a worry about finding enough space to store the intermediate results of the computation, and even a worry about finding enough matter to use in writing down those results: there’s only a finite amount of paper in the world, so you’d have to writer smaller and smaller without limit; to get an infinite number of symbols down on paper, eventually you’d be trying to write on molecules, on atoms, on electrons.) But our present study will ignore these practical limitations, and work with an idealized notion of computability that goes beyond what actual people or actual machines can be sure of doing. Our eventual goal will be to prove that certain functions are not computable, even if practical limitations on time, speed, and amount of material could somehow be overcome, and for this purpose the essential requirement is that our notion of computability not be too narrow.

So far we have been sliding over a significant point. When we are given as argument a number $n$ or pair of numbers $(m, n)$, what we in fact are directly given is a numeral for $n$ or an ordered pair of numerals for $m$ and $n$. Likewise, if the value of the function we are trying to compute is a number, what our computations in fact end with is a numeral for that number. Now in the course of human history a great many systems of numeration have been developed, from the primitive monadic or tally notation, in which the number $n$ is represented by a sequence of $n$ strokes, through systems like Roman numerals, in which bunches of five, ten, fifty, one-hundred, and so forth strokes are abbreviated by special symbols, to the Hindu–Arabic or decimal notation in common use today. Does it make a difference in a definition of computability which of these many systems we adopt?

Certainly computations can be harder in practice with some notations than with others. For instance, multiplying numbers given in decimal numerals (expressing the product in the same form) is easier in practice than multiplying numbers given in something like Roman numerals. Suppose we are given two numbers, expressed in Roman numerals, say XXXIX and XLVIII, and are asked to obtain the product, also expressed in Roman numerals. Probably for most us the easiest way to do this would be first to translate from Roman to Hindu–Arabic—the rules for doing this are, or at least used to be, taught in primary school, and in any case can be looked up in reference works—obtaining 39 and 48. Next one would carry out the multiplication in our own more convenient numeral system, obtaining 1872. Finally, one would translate the result back into the inconvenient system, obtaining MDCCCLXXII. Doing all this is, of course, harder than simply performing a multiplication on numbers given by decimal numerals to begin with.

But the example shows that when a computation can be done in one notation, it is possible in principle to do in any other notation, simply by translating the data from the difficult notation into an easier one, performing the operation using the easier notation, and then translating the result back from the easier to the difficult notation. If a function is effectively computable when numbers are represented in one system of numerals, it will also be so when numbers are represented in any other system of numerals, provided only that translation between the systems can itself be
carried out according to explicit rules, which is the case for any historical system of numeration that we have been able to decipher. (To say we have been able to decipher it amounts to saying that there are rules for translating back and forth between it and the system now in common use.) For purposes of framing a rigorously defined notion of computability, it is convenient to use monadic or tally notation.

A Turing machine is a specific kind of idealized machine for carrying out computations, especially computations on positive integers represented in monadic notation. We suppose that the computation takes place on a tape, marked into squares, which is unending in both directions—either because it is actually infinite or because there is someone stationed at each end to add extra blank squares as needed. Each square either is blank, or has a stroke printed on it. (We represent the blank by $S_0$ or 0 or most often $B$, and the stroke by $S_1$ or | or most often 1, depending on the context.) And with at most a finite number of exceptions, all squares are blank, both initially and at each subsequent stage of the computation.

At each stage of the computation, the computer (that is, the human or mechanical agent doing the computation) is scanning some one square of the tape. The computer is capable of erasing a stroke in the scanned square if there is one there, or of printing a stroke if the scanned square is blank. And he, she, or it is capable of movement: one square to the right or one square to the left at a time. If you like, think of the machine quite crudely, as a box on wheels which, at any stage of the computation, is over some square of the tape. The tape is like a railroad track; the ties mark the boundaries of the squares; and the machine is like a very short car, capable of moving along the track in either direction, as in Figure 3-1.

At the bottom of the car there is a device that can read what’s written between the ties, and erase or print a stroke. The machine is designed in such a way that at each stage of the computation it is in one of a finite number of internal states, $q_1, \ldots, q_m$. Being in one state or another might be a matter of having one or another cog of a certain gear uppermost, or of having the voltage at a certain terminal inside the machine at one or another of $m$ different levels, or what have you: we are not concerned with the mechanics or the electronics of the matter. Perhaps the simplest way to picture the thing is quite crudely: inside the box there is a little man, who does all the reading and writing and erasing and moving. (The box has no bottom: the poor mug just walks along between the ties, pulling the box along.) This operator inside the machine has a list of $m$ instructions written down on a piece of paper and is in state $q_i$when carrying out instruction number $i$.

Each of the instructions has conditional form: it tells what to do, depending on whether the symbol being scanned (the symbol in the scanned square) is the blank or
stroke, $S_0$ or $S_1$. Namely, there are five things that can be done:

1. **Erase**: write $S_0$ in place of whatever is in the scanned square.
2. **Print**: write $S_1$ in place of whatever is in the scanned square.
3. **Move one square to the right**.
4. **Move one square to the left**.
5. **Halt the computation**.

[In case the square is already blank, (1) amounts to doing nothing; in case the square already has a stroke in it, (2) amounts to doing nothing.] So depending on what instruction is being carried out (= what state the machine, or its operator, is in) and on what symbol is being scanned, the machine or its operator will perform one or another of these five overt acts. Unless the computation has halted (overt act number 5), the machine or its operator will perform also a covert act, in the privacy of box, namely, the act of determining what the next instruction (next state) is to be. Thus the present state and the presently scanned symbol determine what overt act is to be performed, and what the next state is to be.

The overall program of instructions can be specified in various ways, for example, by a *machine table*, or by a *flow chart* (also called a *flow graph*), or by a *set of quadruples*. For the case of a machine that writes three symbols $S_1$ on a blank tape and then halts, the scanning of the leftmost of the three, three sorts of descriptions are illustrated in Figure 3-2.

![Figure 3-2. A Turing machine program.](image)

**3.1 Example** (Writing a specified number of strokes). We indicate in Figure 3-2 a machine that will write the symbol $S_1$ three times. A similar construction works for any specified symbol and any specified number of times. The machine will write an $S_1$ on the square it’s initially scanning, move left one square, write an $S_1$ there, move left one more square, write an $S_1$ there, and halt. (It halts when it has no further instructions.) There will be three states—one for each of the symbols $S_1$ that are to be written. In Figure 3-2, the entries in the top row of the machine table (under the horizontal line) tell the machine or its operator, when following instruction $q_1$, that (1) an $S_1$ is to be written and instruction $q_1$ is to be repeated, if the scanned symbol is $S_0$, but that (2) the machine is to move left and follow instruction $q_2$ next, if the scanned symbol is $S_1$. The same information is given in the flow chart by the two arrows that emerge from the node marked $q_1$; and the same information is also given by the first two quadruples. The significance
in general of a table entry, of an arrow in a flow chart, and of a quadruple is shown in Figure 3-3.

Figure 3-3. A Turing machine instruction.

Unless otherwise stated, it is to be understood that a machine starts in its lowest-numbered state. The machine we have been considering halts when it is in state \( q_3 \) scanning \( S_1 \), for there is no table entry or arrow or quadruple telling it what to do in such a case. A virtue of the flow chart as a way of representing the machine program is that if the starting state is indicated somehow (for example, if it is understood that the leftmost node represents the starting state unless there is an indication to the contrary), then we can dispense with the names of the states: It doesn’t matter what you call them. Then the flow chart could be redrawn as in Figure 3-4.

![Figure 3-4. Writing three strokes.](image)

We can indicate how such a Turing machine operates by writing down its sequence of configurations. There is one configuration for each stage of the computation, showing what’s on the tape at that stage, what state the machine is in at that stage, and which square is being scanned. We can show this by writing out what’s on the tape and writing the name of the present state under the symbol in the scanned square; for instance,

\[
1100111
\]

shows a string or block of two strokes followed by two blanks followed by a string or block of three strokes, with the machine scanning the leftmost stroke and in state 2. Here we have written the symbols \( S_0 \) and \( S_1 \) simply as 0 and 1, and similarly the state \( q_2 \) simply as 2, to save needless fuss. A slightly more compact representation writes the state number as a subscript on the symbol scanned: \( 1_2 100111 \).

This same configuration could be written \( 01_2 100111 \) or \( 1_2 1001110 \) or \( 01_2 1001110 \) or \( 001_2 100111 \) or \( \ldots \) —a block of 0s can be written at the beginning or end of the tape, and can be shortened or lengthened \textit{ad lib.} without changing the significance: the tape is understood to have as many blanks as you please at each end.

We can begin to get a sense of the power of Turing machines by considering some more complex examples.
3.2 Example (Doubling the number of strokes). The machine starts off scanning the leftmost of a block of strokes on an otherwise blank tape, and winds up scanning the leftmost of a block of twice that many strokes on an otherwise blank tape. The flow chart is shown in Figure 3-5.

![Figure 3-5. Doubling the number of strokes.](image)

How does it work? In general, by writing double strokes at the left and erasing single strokes at the right. In particular, suppose the initial configuration is 1_11, so that we start in state 1, scanning the leftmost of a block of three strokes on an otherwise blank tape. The next few configurations are as follows:

\[
\begin{align*}
0_2111 & \quad 0_30111 \quad 1_30111 \quad 0_410111 \quad 1_410111. \\
\end{align*}
\]

So we have written our first double stroke at the left—separated from the original block 111 by a blank. Next we go right, past the blank to the right-hand end of the original block, and erase the rightmost stroke. Here is how that works, in two phases. Phase 1:

\[
\begin{align*}
1150111 & \quad 1101111 \quad 1101611 \quad 1101161 \quad 11011161 \quad 11011106. \\
\end{align*}
\]

Now we know that we have passed the last of the original block of strokes, so (phase 2) we back up, erase one of them, and move one more square left:

\[
\begin{align*}
1101117 & \quad 1101107 \quad 1101180. \\
\end{align*}
\]

Now we hop back left, over what is left of the original block of strokes, over the blank separating the original block from the additional strokes we have printed, and over those additional strokes, until we find the blank beyond the leftmost stroke:

\[
\begin{align*}
110191 & \quad 110911 \quad 1110111 \quad 11010111. \\
\end{align*}
\]

Now we will print another two new strokes, much as before:

\[
\begin{align*}
0121011 & \quad 0_311011 \quad 1_311011 \quad 0_4111011 \quad 1_4111011. \\
\end{align*}
\]

We are now back on the leftmost of the block of newly printed strokes, and the process that led to finding and erasing the rightmost stroke will be repeated, until we arrive at the following:

\[
\begin{align*}
11110117 & \quad 11110107 \quad 11110180. \\
\end{align*}
\]

Another round of this will lead first to writing another pair of strokes:

\[
141111101. 
\]
It will then lead to erasing the last of the original block of strokes:

\[
\begin{align*}
11111101_7 & \quad 11111100_7 & \quad 1111110_8 0.
\end{align*}
\]

And now the endgame begins, for we have what we want on the tape, and need only move back to halt on the leftmost stroke:

\[
\begin{align*}
11111111 & \quad 11111111 & \quad 111111 & \quad 1111 & \quad 111 & \quad 11111 & \quad 1 & \quad 1111111 & \quad 0 \quad 1111111 & \quad 1 \quad 1111111.
\end{align*}
\]

Now we are in state 12, scanning a stroke. Since there is no arrow from that node telling us what to do in such a case, we halt. The machine performs as advertised.

(Note: The fact that the machine doubles the number of strokes when the original number is three is not a proof that the machine performs as advertised. But our examination of the special case in which there are three strokes initially made no essential use of the fact that the initial number was three: it is readily converted into a proof that the machine doubles the number of strokes no matter how long the original block may be.)

Readers may wish, in the remaining examples, to try to design their own machines before reading our designs; and for this reason we give the statements of all the examples first, and collect all the proofs afterward.

3.3 Example (Determining the parity of the length of a block of strokes). There is a Turing machine that, started scanning the leftmost of an unbroken block of strokes on an otherwise blank tape, eventually halts, scanning a square on an otherwise blank tape, where the square contains a blank or a stroke depending on whether there were an even or an odd number of strokes in the original block.

3.4 Example (Adding in monadic (tally) notation). There is a Turing machine that does the following. Initially, the tape is blank except for two solid blocks of strokes, say a left block of \(p\) strokes and a right block of \(q\) strokes, separated by a single blank. Started on the leftmost blank of the left block, the machine eventually halts, scanning the leftmost stroke in a solid block of \(p + q\) strokes on an otherwise blank tape.

3.5 Example (Multiplying in monadic (tally) notation). There is a Turing machine that does the same thing as the one in the preceding example, but with \(p \cdot q\) in place of \(p + q\).

Proofs

Example 3.3. A flowchart for such a machine is shown in Figure 3-6.

![Figure 3-6. Parity machine.](image)

If there were 0 or 2 or 4 or \(\ldots\) strokes to begin with, this machine halts in state 1, scanning a blank on a blank tape; if there were 1 or 3 or 5 or \(\ldots\), it halts in state 5, scanning a stroke on an otherwise blank tape.
Example 3.4. The object is to erase the leftmost stroke, fill the gap between the two blocks of strokes, and halt scanning the leftmost stroke that remains on the tape. Here is one way of doing it, in quadruple notation: $q_1S_1S_0q_1; q_1S_0Rq_2; q_2S_1Rq_2; q_2S_0S_1q_3; q_3S_1Lq_3; q_3S_0Rq_4$.

Example 3.5. A flow chart for a machine is shown in Figure 3-7.

At this point the machine is scanning the leftmost 1 on the tape.

Figure 3-7. Multiplication machine.

Here is how the machine works. The first block, of $p$ strokes, is used as a counter, to keep track of how many times the machine has added $q$ strokes to the group at the right. To start, the machine erases the leftmost of the $p$ strokes and sees if there are any strokes left in the counter group. If not, $pq = q$, and all the machine has to do is position itself over the leftmost stroke on the tape, and halt.
But if there are any strokes left in the counter, the machine goes into a *leapfrog routine*: in effect, it moves the block of *q* strokes (the *leapfrog group*) *q* places to the right along the tape. For example, with *p* = 2 and *q* = 3 the tape looks like this initially:

$$11B111$$

and looks like this after going through the leapfrog routine:

$$B1BBBB111.$$  

The machine will then note that there is only one 1 left in the counter, and will finish up by erasing that 1, moving right two squares, and changing all *B*s to strokes until it comes to a stroke, at which point it continues to the leftmost 1 and halts.

The general picture of how the leapfrog routine works is shown in Figure 3-8.

![Figure 3-8. Leapfrog.](image)

In general, the leapfrog group consists of a block of 0 or 1 or . . . or *q* strokes, followed by a blank, followed by the remainder of the *q* strokes. The blank is there to tell the machine when the leapfrog game is over: without it the group of *q* strokes would keep moving right along the tape forever. (In playing leapfrog, the portion of the *q* strokes to the left of the blank in the leapfrog group functions as a counter: it controls the process of adding strokes to the portion of the leapfrog group to the right of the blank. That is why there are two big loops in the flow chart: one for each counter-controlled subroutine.)

We have not yet given an official definition of what it is for a numerical function to be computable by a Turing machine, specifying how inputs or arguments are to be represented on the machine, and how outputs or values represented. Our specifications for a *k*-place function from positive integers to positive integers are as follows:

(a) The arguments $m_1, \ldots, m_k$ of the function will be represented in monadic notation by blocks of those numbers of strokes, each block separated from the next by a single blank, on an otherwise blank tape. Thus, at the beginning of the computation of, say, $3 + 2$, the tape will look like this: $111B111$.

(b) Initially, the machine will be scanning the leftmost 1 on the tape, and will be in its initial state, state 1. Thus in the computation of $3 + 2$, the initial configuration will be $1_111B111$. A configuration as described by (a) and (b) is called a *standard initial configuration* (or position).

(c) If the function that is to be computed assigns a value *n* to the arguments that are represented initially on the tape, then the machine will eventually halt on a tape
containing a block of that number of strokes, and otherwise blank. Thus in the
computation of 3 + 2, the tape will look like this: 11111.

(d) In this case, the machine will halt scanning the leftmost 1 on the tape. Thus in the
computation of 3 + 2, the final configuration will be $q_n1111$, where $n$th state is one
for which there is no instruction what to do if scanning a stroke, so that in this
configuration the machine will be halted. A configuration as described by (c) and
(d) is called a standard final configuration (or position).

(e) If the function that is to be computed assigns no value to the arguments that are
represented initially on the tape, then the machine either will never halt, or will
halt in some nonstandard configuration such as $B_n11111$ or $B11_n111$ or $B11111_n$.

The restriction above to the standard position (scanning the leftmost 1) for starting
and halting is inessential, but some specifications or other have to be made about
initial and final positions of the machine, and the above assumptions seem especially
simple.

With these specifications, any Turing machine can be seen to compute a function of
one argument, a function of two arguments, and, in general, a function of $k$ arguments
for each positive integer $k$. Thus consider the machine specified by the single quadruple $q_111q_2$. Started in a standard initial configuration, it immediately halts, leaving the
tape unaltered. If there was only a single block of strokes on the tape initially, its final
configuration will be standard, and thus this machine computes the identity function id
of one argument: $id(m) = m$ for each positive integer $m$. Thus the machine computes
a certain total function of one argument. But if there were two or more blocks of
strokes on the tape initially, the final configuration will not be standard. Accordingly,
the machine computes the extreme partial function of two arguments that is undefined
for all pairs of arguments: the empty function $e_2$ of two arguments. And in general,
for $k$ arguments, this machine computes the empty function $e_k$ of $k$ arguments.

By contrast, consider the machine whose flow chart is shown in Figure 3-9. This
machine computes for each $k$ the total function that assigns the same value, namely 1,
to each $k$-tuple. Started in initial state 1 in a standard initial configuration, this machine
erases the first block of strokes (cycling between states 1 and 2 to do so) and goes to
state 3, scanning the second square to the right of the first block. If it sees a blank there,
it knows it has erased the whole tape, and so prints a single 1 and halts in state 4, in
a standard configuration. If it sees a stroke there, it re-enters the cycle between states
1 and 2, erasing the second block of strokes and inquiring again, in state 3, whether
the whole tape is blank, or whether there are still more blocks to be dealt with.

![Figure 3-9](image-url)
A numerical function of \( k \) arguments is Turing computable if there is some Turing machine that computes it in the sense we have just been specifying. Now computation in the Turing-machine sense is certainly one kind of computation in the intuitive sense, so all Turing-computable functions are effectively computable. Turing’s thesis is that, conversely, any effectively computable function is Turing computable, so that computation in the precise technical sense we have been developing coincides with effective computability in the intuitive sense.

It is easy to imagine liberalizations of the notion of the Turing machine. One could allow machines using more symbols than just the blank and the stroke. One could allow machines operating on a rectangular grid, able to move up or down a square as well as left or right. Turing’s thesis implies that no liberalization of the notion of Turing machine will enlarge the class of functions computable, because all functions that are effectively computable in any way at all are already computable by a Turing machine of the restricted kind we have been considering. Turing’s thesis is thus a bold claim.

It is possible to give a heuristic argument for it. After all, effective computation consists of moving around and writing and perhaps erasing symbols, according to definite, explicit rules; and surely writing and erasing symbols can be done stroke by stroke, and moving from one place to another can be done step by step. But the main argument will be the accumulation of examples of effectively computable functions that we succeed in showing are Turing computable. So far, however, we have had just a few examples of Turing machines computing numerical functions, that is, of effectively computable functions that we have proved to be Turing computable: addition and multiplication in the preceding section, and just now the identity function, the empty function, and the function with constant value 1.

Now addition and multiplication are just the first two of a series of arithmetic operations all of which are effectively computable. The next item in the series is exponentiation. Just as multiplication is repeated addition, so exponentiation is repeated multiplication. (Then repeated exponentiation gives a kind of super-exponentiation, and so on. We will investigate this general process of defining new functions from old in a later chapter.) If Turing’s thesis is correct, there must be a Turing machine for each of these functions, computing it. Designing a multiplier was already difficult enough to suggest that designing an exponentiator would be quite a challenge, and in any case, the direct approach of designing a machine for each operation would take us forever, since there are infinitely many operations in the series. Moreover, there are many other effectively computable numerical functions besides the ones in this series. When we return, in the chapter after next, to the task of showing various effectively computable numerical functions to be Turing computable, and thus accumulating evidence for Turing’s thesis, a less direct approach will be adopted, and all the operations in the series that begins with addition and multiplication will be shown to be Turing computable in one go.

For the moment, we set aside the positive task of showing functions to be Turing computable and instead turn to examples of numerical functions of one argument that are Turing uncomputable (and so, if Turing’s thesis is correct, effectively uncomputable).
Problems

3.1 Design a Turing machine that will do the following. Given a tape containing a block of strokes, and otherwise blank, if the machine is started scanning the leftmost stroke on the tape, it will eventually halt scanning the rightmost stroke on the tape, having neither printed nor erased anything.

3.2 Design a Turing machine that will do the following. Given a tape containing a block of strokes, followed by a blank, followed by another block of strokes, and otherwise blank, if the machine is started scanning the leftmost stroke on the tape, it will eventually halt scanning the rightmost stroke on the tape, having neither printed nor erased anything.

3.3 Design a Turing machine that will do the following. Given a tape containing a block of \( n \) strokes, followed by a blank, followed by a block of \( m \) strokes, followed by a blank, followed by a block of \( k \) strokes, and otherwise blank, if the machine is started scanning the rightmost stroke on the tape, it will eventually halt with the tape containing a block of \( n - 1 \) strokes, followed by a blank, followed by a block of \( m + 1 \) strokes, followed by a blank, followed by a block of \( k + 1 \) strokes, and otherwise blank, with the machine scanning the rightmost stroke on the tape.

3.4 Design a Turing machine that will do the following. Given a tape containing a block of \( n \) strokes, followed by a blank, followed by a block of \( m \) strokes, followed by a blank, followed by a block of \( k \) strokes, and otherwise blank, if the machine is started scanning the rightmost stroke on the tape, it will eventually halt with the tape containing a block of \( n - 1 \) strokes, followed by a blank, followed by a block of \( m - 1 \) strokes, followed by a blank, followed by a block of \( k + 1 \) strokes, and otherwise blank, with the machine scanning the rightmost stroke on the tape.

3.5 Design a Turing machine to compute the function \( \min(x, y) = \) the smaller of \( x \) and \( y \).

3.6 Design a Turing machine to compute the function \( \max(x, y) = \) the larger of \( x \) and \( y \).
In the preceding chapter we introduced the notion of Turing computability. In the present short chapter we give examples of Turing-uncomputable functions: the halting function in section 4.1, and the productivity function in the optional section 4.2. If Turing’s thesis is correct, these are actually examples of effectively uncomputable functions.

4.1 The Halting Problem

There are too many functions from positive integers to positive integers for them all to be Turing computable. For on the one hand, as we have seen in Chapter 2, the set of all such functions is nonenumerable. And on the other hand, the set of Turing machines, and therefore of Turing-computable functions, is enumerable, since the representation of a Turing machine in the form of quadruples amounts to a representation of it by a finite string of symbols from a finite alphabet; and we have seen in Chapter 1 that the set of such strings is enumerable. These considerations show us that there must exist functions that are not Turing computable, but they do not provide an explicit example of such a function. To provide explicit examples is the task of this chapter.

We begin simply by examining the argument just given in slow motion, with careful attention to details, so as to extract a specific example of a Turing-uncomputable function from it.

To begin with, we have suggested that we can enumerate the Turing-computable functions of one argument by enumerating the Turing machines, and that we can enumerate the Turing machines using their quadruple representations. As we turn to details, it will be convenient to modify the quadruple representation used so far somewhat. To indicate the nature of the modifications, consider the machine in Figure 3-9 in the preceding chapter. Its quadruple representation would be

\[ q_1 S_0 R q_3, q_1 S_1 S_0 q_2, q_2 S_0 R q_1, q_3 S_0 S_1 q_4, q_3 S_1 S_0 q_2. \]

We have already been taking the lowest-numbered state \( q_1 \) to be the initial state. We now want to assume that the highest-numbered state is a halted state, for which there are no instructions and no quadruples. This is already the case in our example, and if it were not already so in some other example, we could make it so by adding one additional state.
We now also want to assume that for every state \( q_i \) except this highest-numbered halted state, and for each of the two symbols \( S_j \) we are allowing ourselves to use, namely \( S_0 = B \) and \( S_1 = 1 \), there is a quadruple beginning \( q_i S_j \). This is not so in our example as it stands, where there is no instruction for \( q_2 S_1 \). We have been interpreting the absence of an instruction for \( q_i S_j \) as an instruction to halt, but the same effect could be achieved by giving an explicit instruction to keep the same symbol and then go to the highest-numbered state. When we modify the representation by adding this instruction, the representation becomes

\[
q_1 S_0 R q_3, \ q_1 S_0 q_2, \ q_2 S_0 R q_1, \ q_2 S_1 S_1 q_4, \ q_3 S_0 S_1 q_4, \ q_3 S_1 S_0 q_2.
\]

Now taking the quadruples beginning \( q_1 S_0, q_1 S_1, q_2 S_0, \ldots \) in that order, as we have done, the first two symbols of each quadruple are predictable and therefore do not need to be written. So we may simply write

\[
R q_3, \ S_0 q_2, \ S_1 q_4, \ S_1 q_4, \ S_0 q_2.
\]

Representing \( q_i \) by \( i \), and \( S_j \) by \( j + 1 \) (so as to avoid 0), and \( L \) and \( R \) by 3 and 4, we can write still more simply

\[
4, \ 3, \ 1, \ 2, \ 4, \ 1, \ 2, \ 4, \ 2, \ 4, \ 1, \ 2.
\]

Thus the Turing machine can be completely represented by a finite sequence of positive integers—and even, if desired, by a single positive integer, say using the method of coding based on prime decomposition:

\[
2^4 \cdot 3^3 \cdot 5 \cdot 7^2 \cdot 11^4 \cdot 13 \cdot 17^2 \cdot 19^4 \cdot 23^2 \cdot 29^4 \cdot 31 \cdot 37^2.
\]

Not every positive integer will represent a Turing machine: whether a given positive integer does or not depends on what the sequence of exponents in its prime decomposition is, and not every finite sequence represents a Turing machine. Those that do must have length some multiple 4\( n \) of 4, and have among their odd-numbered entries only numbers 1 to 4 (representing \( B, L, R \)) and among their even-numbered entries only numbers 1 to \( n + 1 \) (representing the initial state \( q_1 \), various other states \( q_i \), and the halted state \( q_{n+1} \)). But no matter: from the above representation we at least get a gappy listing of all Turing machines, in which each Turing machine is listed at least once, and on filling in the gaps we get a gapless list of all Turing machines, \( M_1, M_2, M_3, \ldots \), and from this a similar list of all Turing-computable functions of one argument, \( f_1, f_2, f_3, \ldots \), where \( f_i \) is the total or partial function computed by \( M_i \).

To give a trivial example, consider the machine represented by \( (1, 1, 1, 1) \), or \( 2 \cdot 3 \cdot 5 \cdot 7 = 210 \). Started scanning a stroke, it erases it, then leaves the resulting blank alone and remains in the same initial state, never going to the halted state, which would be state 2. Or consider the machine represented by \( (2, 1, 1, 1) \) or \( 2^2 \cdot 3 \cdot 5 \cdot 7 = 420 \). Started scanning a stroke, it erases it, then prints it back again, then erases it, then prints it back again, and so on, again never halting. Or consider the machine represented by \( (1, 2, 1, 1) \), or \( 2 \cdot 3^2 \cdot 5 \cdot 7 = 630 \). Started scanning a stroke, it erases it, then goes to the halted state 2 when it scans the resulting blank, which means halting in a nonstandard final configuration. A little thought shows that 210, 420, 630 are the smallest numbers that represent Turing machines, so the three
machines just described will be $M_1$, $M_2$, $M_3$, and we have $f_1 = f_2 = f_3 =$ the empty function.

We have now indicated an explicit enumeration of the Turing-computable functions of one argument, obtained by enumerating the machines that compute them. The fact that such an enumeration is possible shows, as we remarked at the outset, that there must exist Turing-uncomputable functions of a single argument. The point of actually specifying one such enumeration is to be able to exhibit a particular such function. To do so, we define a diagonal function $d$ as follows:

$$d(n) = \begin{cases} 1 & \text{if } f_n(n) \text{ is defined and } = 1 \\ 2 & \text{otherwise.} \end{cases}$$

Now $d$ is a perfectly genuine total function of one argument, but it is not Turing computable, that is, $d$ is neither $f_1$ nor $f_2$ nor $f_3$, and so on. \textit{Proof:} Suppose that $d$ is one of the Turing computable functions—the $m$th, let us say. Then for each positive integer $n$, either $d(n)$ and $f_m(n)$ are both defined and equal, or neither of them is defined. But consider the case $n = m$:

$$f_m(m) = d(m) = \begin{cases} 2 & \text{if } f_m(m) \text{ is defined and } = 1 \\ 1 & \text{otherwise.} \end{cases}$$

Then whether $f_m(m)$ is or is not defined, we have a contradiction: Either $f_m(m)$ is undefined, in which case (2) tells us that it is defined and has value 1; or $f_m(m)$ is defined and has a value $\neq 1$, in which case (2) tells us it has value 1; or $f_m(m)$ is defined and has value 1, in which case (2) tells us it has value 2. Since we have derived a contradiction from the assumption that $d$ appears somewhere in the list $f_1, f_2, \ldots, f_m, \ldots$, we may conclude that the supposition is false. We have proved:

4.1 Theorem. The diagonal function $d$ is not Turing computable.

According to Turing’s thesis, since $d$ is not Turing computable, $d$ cannot be effectively computable. Why not? After all, although no Turing machine computes the function $d$, we were able compute at least its first few values. For since, as we have noted, $f_1 = f_2 = f_3 =$ the empty function we have $d(1) = d(2) = d(3) = 1$. And it may seem that we can actually compute $d(n)$ for any positive integer $n$—if we don’t run out of time.

Certainly it is straightforward to discover which quadruples determine $M_n$ for $n = 1, 2, 3$, and so on. (This is straightforward in principle, though eventually humanly infeasible in practice because the duration of the trivial calculations, for large $n$, exceeds the lifetime of a human being and, in all probability, the lifetime of the human race. But in our idealized notion of computability, we ignore the fact that human life is limited.)

And certainly it is perfectly routine to follow the operations of $M_n$, once the initial configuration has been specified; and if $M_n$ does eventually halt, we must eventually get that information by following its operations. Thus if we start $M_n$ with input $n$ and it does halt with that input, then by following its operations until it halts, we can see whether it halts in nonstandard position, leaving $f_n(n)$ undefined, or halts in standard
position with output $f_n(n) = 1$, or halts in standard position with output $f_n(n) \neq 1$. In the first or last cases, $d(n) = 1$, and in the middle case, $d(n) = 2$.

But there is yet another case where $d(n) = 1$; namely, the case where $M_n$ never halts at all. If $M_n$ is destined never to halt, given the initial configuration, can we find that out in a finite amount of time? This is the essential question: determining whether machine $M_n$, started scanning the leftmost of an unbroken block of $n$ strokes on an otherwise blank tape, does or does not eventually halt.

Is this perfectly routine? Must there be some point in the routine process of following its operations at which it becomes clear that it will never halt? In simple cases this is so, as we saw in the cases of $M_1$, $M_2$, and $M_3$ above. But for the function $d$ to be effectively computable, there would have to be a uniform mechanical procedure, applicable not just in these simple cases but also in more complicated cases, for discovering whether or not a given machine, started in a given configuration, will ever halt.

Thus consider the multiplier in Example 3.5. Its sequential representation would be a sequence of 68 numbers, each $\leq 18$. It is routine to verify that it represents a Turing machine, and one can easily enough derive from it a flow chart like the one shown in Figure 3-7, but without the annotations, and of course without the accompanying text. Suppose one came upon such a sequence. It would be routine to check whether it represented a Turing machine and, if so, again to derive a flow chart without annotations and accompanying text. But is there a uniform method or mechanical routine that, in this and much more complicated cases, allows one to determine from inspecting the flow chart, without any annotations or accompanying text, whether the machine eventually halts, once the initial configuration has been specified?

If there is such a routine, Turing’s thesis is erroneous: if Turing’s thesis is correct, there can be no such routine. At present, several generations after the problem was first posed, no one has yet succeeded in describing any such routine—a fact that must be considered some kind of evidence in favor of the thesis.

Let us put the matter another way. A function closely related to $d$ is the halting function $h$ of two arguments. Here $h(m, n) = 1$ or 2 according as machine $m$, started with input $n$, eventually halts or not. If $h$ were effectively computable, $d$ would be effectively computable. For given $n$, we could first compute $h(n, n)$. If we got $h(n, n) = 2$, we would know that $d(n) = 1$. If we got $h(n, n) = 1$, we would know that we could safely start machine $M_n$ in standard initial configuration for input $n$, and that it would eventually halt. If it halted in nonstandard configuration, we would again have $d(n) = 1$. If it halted in standard final configuration giving an output $f_n(n)$, it would have $d(n) = 1$ or 2 according as $f_n(n) \neq 1$ or $= 1$.

This is an informal argument showing that if $h$ were effectively computable, then $d$ would be effectively computable. Since we have shown that $d$ is not Turing computable, assuming Turing’s thesis it follows that $d$ is not effectively computable, and hence that $h$ is not effectively computable, and so not Turing computable. It is also possible to prove rigorously, though we do not at this point have the apparatus needed to do so, that if $h$ were Turing computable, then $d$ would be Turing computable, and since we have shown that $d$ is not Turing computable, this would show that $h$ is not
4.1. THE HALTING PROBLEM

Finally, it is possible to prove rigorously in another way, not involving $d$, that $h$ is not Turing computable, and this we now do.

4.2 Theorem. The halting function $h$ is not Turing computable.

Proof: By way of background we need two special Turing machines. The first is a copying machine $C$, which works as follows. Given a tape containing a block of $n$ strokes, and otherwise blank, if the machine is started scanning the leftmost stroke on the tape, it will eventually halt with the tape containing two blocks of $n$ strokes separated by a blank, and otherwise blank, with the machine scanning the leftmost stroke on the tape. Thus if the machine is started with

$$\ldots BBB1111BBB \ldots$$

it will halt with

$$\ldots BBB1111B1111BBB \ldots$$

eventually. We ask you to design such a machine in the problems at the end of this chapter (and give you a pretty broad hint how to do it at the end of the book).

The second is a dithering machine $D$. Started on the leftmost of a block of $n$ strokes on an otherwise blank tape, $D$ eventual halts if $n > 1$, but never halts if $n = 1$. Such a machine is described by the sequence

$$1, 3, 4, 2, 3, 1, 3, 3.$$

Started on a stroke in state 1, it moves right and goes into state 2. If it finds itself on a stroke, it moves back left and halts, but if it finds itself on a blank, it moves back left and goes into state 1, starting an endless back-and-forth cycle.

Now suppose we had a machine $H$ that computed the function $h$. We could combine the machines $C$ and $H$ as follows: if the states of $C$ are numbered 1 through $p$, and the states of $H$ are numbered 1 through $q$, renumber the latter states $p + 1$ through $r = p + q$, and write these renumbered instructions after the instructions for $C$. Originally, $C$ tells us to halt by telling us to go into state $p + 1$, but in the new combined instructions, going into state $p + 1$ means not halting, but beginning the operations of machine $H$. So the new combined instructions will have us first go through the operations of $C$, and then, when $C$ would have halted, go through the operations of $H$. The result is thus a machine $G$ that computes the function $g(n) = h(n, n)$.

We now combine this machine $G$ with the dithering machine $D$, renumbering the states of the latter as $r + 1$ and $r + 2$, and writing its instructions after those for $G$. The result will be a machine $M$ that goes through the operations of $G$ and then the operations of $D$. Thus if machine number $n$ halts when started on its own number, that is, if $h(n, n) = g(n) = 1$, then the machine $M$ does not halt when started on that number $n$, whereas if machine number $n$ does not halt when started on its own number, that is, if $h(n, n) = g(n) = 2$, then machine $M$ does halt when started on $n$. But of course there can be no such machine as $M$. For what would it do if started with input its own number $m$? It would halt if and only if machine number $m$, which is
to say itself, does \textit{not} halt when started with input the number $m$. This contradiction shows there can be no such machine as $H$.

The \textit{halting problem} is to find an effective procedure that, given any Turing machine $M$, say represented by its number $m$, and given any number $n$, will enable us to determine whether or not that machine, given that number as input, ever halts. For the problem to be solvable by a Turing machine would require there to be a Turing machine that, given $m$ and $n$ as inputs, produces as its output the answer to the question whether machine number $m$ with input $n$ ever halts. Of course, a Turing machine of the kind we have been considering could not produce the output by printing the word ‘yes’ or ‘no’ on its tape, since we are considering machines that operate with just two symbols, the blank and the stroke. Rather, we take the affirmative answer to be presented by an output of 1 and the negative by an output of 2. With this understanding, the question whether the halting problem can be solved by a Turing machine amounts to the question whether the halting function $h$ is Turing computable, and we have just seen in Theorem 4.2 that it is not. That theorem, accordingly, is often quoted in the form: ‘The halting problem is not solvable by a Turing machine.’ Assuming Turing’s thesis, it follows that it is not solvable at all.

Thus far we have two examples of functions that are not Turing computable—or problems that are not solvable by any Turing machine—and if Turing’s thesis is correct, these functions are not effectively computable. A further example is given in the next section. Though working through the example will provide increased familiarity with the potential of Turing machines that will be desirable when we come to the next chapter, and in any case the example is a beautiful one, still none of the material connected with this example is strictly speaking indispensable for any of our further work; and therefore we have starred the section in which it appears as optional.

4.2* \textbf{The Productivity Function}

Consider a $k$-state Turing machine, that is, a machine with $k$ states (not counting the halted state). Start it with input $k$, that is, start it in its initial state on the leftmost of a block of $k$ strokes on an otherwise blank tape. If the machine never halts, or halts in nonstandard position, give it a score of zero. If it halts in standard position with output $n$, that is, on the leftmost of a block of $n$ strokes on an otherwise blank tape, give it a score of $n$. Now define $s(k) =$ the highest score achieved by any $k$-state Turing machine. This function can be shown to be Turing uncomputable.

We first show that if the function $s$ were Turing computable, then so would be the function $t$ given by $t(k) = s(k) + 1$. For supposing we have a machine that computes $s$, we can modify it as follows to get a machine, having one more state than the original machine, that computes $t$. Where the instructions for the original machine would have it halt, the instructions for the new machine will have it go into the new, additional state. In this new state, if the machine is scanning a stroke, it is to move one square to the left, remaining in the new state; while if it is scanning a blank, it is to print a stroke and halt. A little thought shows that a computation of the new machine will
4.2. THE PRODUCTIVITY FUNCTION

If the new machine would halt on the leftmost of a block of \( n \) strokes, the new machine will go through two more steps of computation (moving left and printing a stroke), leaving it halted on the leftmost of a block of \( n + 1 \) strokes. Thus its output will be one more than the output of the original machine, and if the original machine, for a given argument, computes the value of \( s \), the new machine will compute the value of \( t \).

Thus, to show that no Turing machine can compute \( s \), it will now be enough to show that no Turing machine can compute \( t \). And this is not hard to do. For suppose there were a machine computing \( t \). It would have some number \( k \) of states (not counting the halted state). Started on the leftmost of a block of \( k \) strokes on an otherwise blank tape, it would halt on the leftmost of a block of \( t(k) \) strokes on an otherwise blank tape. But then \( t(k) \) would be the score of this particular \( k \)-state machine, and that is impossible, since \( t(k) > s(k) = \) the highest score achieved by any \( k \)-state machine. Thus we have proved:

**4.3 Proposition.** The scoring function \( s \) is not Turing computable.

Let us have another look at the function \( s \) in the light of Turing’s thesis. According to Turing’s thesis, since \( s \) is not Turing computable, \( s \) cannot be effectively computable. Why not? After all there are (ignoring labelling) only finitely many quadruple representations or flow charts of \( k \)-place Turing machines for a given \( k \). We could in principle start them all going in state 1 with input \( k \) and await developments. Some machines will halt at once, with score 0. As time passes, one or another of the other machines may halt; then we can check whether or not it has halted in standard position. If not, its score is 0; if so, its score can be determined simply by counting the number of strokes in a row on the tape. If this number is less than or equal to the score of some \( k \)-state machine that stopped earlier, we can ignore it. If it is greater than the score of any such machine, then it is the new record-holder. Some machines will run on forever, but since there are only finitely many machines, there will come a time when any machine that is ever going to halt has halted, and the record-holding machine at that time is an \( k \)-state machine of maximum score, and its score is equal to \( s(k) \). Why doesn’t this amount to an effective way of computing \( s(k) \)?

It would, if we had some method of effectively determining which machines are eventually going to halt. Without such a method, we cannot determine which of the machines that haven’t halted yet at a given time are destined to halt at some later time, and which are destined never to halt at all, and so we cannot determine whether or not we have reached a time when all machines that are ever going to halt have halted. The procedure outlined in the preceding paragraph gives us a solution to the scoring problem, the problem of computing \( s(n) \), only if we already have a solution to the halting problem, the problem of determining whether or not a given machine will, for given input, eventually halt. This is the flaw in the procedure.

There is a related Turing-uncomputable function that is even simpler to describe than \( s \), called the *Rado* or *busy-beaver* function, which may be defined as follows. Consider a Turing machine started with the tape blank (rather than with input equal to the number of states of the machine, as in the scoring-function example). If the
machine eventually halts, scanning the leftmost of an unbroken block of strokes on an otherwise blank tape, its \textit{productivity} is said to be the length of that block. But if the machine never halts, or halts in some other configuration, its productivity is said to be 0. Now define $p(n) =$ the productivity of the most productive Turing machine having no more than $n$ states (not counting the halted state).

This function also can be shown to be Turing uncomputable.

The facts needed about the function $p$ can be conveniently set down in a series of examples. We state all the examples first, and then give our proofs, in case the reader wishes to look for a proof before consulting ours.

4.4 Example. $p(1) = 1$

4.5 Example. $p(n + 1) > p(n)$ for all $n$

4.6 Example. There is an $i$ such that $p(n + i) \geq 2p(n)$ for all $n$

Proofs

Example 4.4. There are just 25 Turing machines with a single state $q_1$. Each may be represented by a flow chart in which there is just one node, and 0 or 1 or 2 arrows (from that node back to itself). Let us enumerate these flow charts.

Consider first the flow chart with no arrows at all. (There is just one.) The corresponding machine halts immediately with the tape still blank, and thus has productivity 0.

Consider next flow charts with two arrows, labelled ‘$B$:—’ and ‘$1$: . . . ’, where each of ‘—’ and ‘. . . ’ may be filled in with $R$ or $L$ or $B$ or 1. There are $4 \cdot 4 = 16$ such flow charts, corresponding to the 4 ways of filling in ‘—’ and the 4 ways of filling in ‘. . . ’. Each such flow chart corresponds to a machine that never halts, and thus has productivity 0. The machine never halts because no matter what symbol it is scanning, there is always an instruction for it to follow, even if it is an instruction like ‘print a blank on the (already blank) square you are scanning, and stay in the state you are in’.

Consider flow charts with one arrow. There are four of them where the arrow is labelled ‘$1$: . . . ’. These all halt immediately, since the machine is started on a blank, and there is no instruction what to do when scanning a blank. So again the productivity is 0.

Finally, consider flow charts with one arrow labelled ‘$B$:—’. Again there are four of them. Three of them have productivity 0: the one ‘$B:B$’, which stays put, and the two labelled ‘$B:R$’ and ‘$B:L$’, which move endlessly down the tape in one direction or the other (touring machines). The one labelled ‘$B:1$’ prints a stroke and then halts, and thus has productivity 1. Since there is thus a 1-state machine whose productivity is 1, and every other 1-state machine has productivity 0, the most productive 1-state machine has productivity 1.

Example 4.5. Choose any of the most productive $n$-state machines, and add one more state, as in Figure 4-1.

The result is an $(n + 1)$-state machine of productivity $n + 1$. There may be $(n + 1)$-state machines of even greater productivity than this, but we have established that
4.2. THE PRODUCTIVITY FUNCTION

the productivity of the most productive \((n + 1)\)-state machines is at least greater by 1 than the productivity of the most productive \(n\)-state machine.

Example 4.6. We can take \(i = 11\). To see this, plug together an \(n\)-state machine for writing a block of \(n\) strokes (Example 3.1) with a 12-state machine for doubling the length of a row of strokes (Example 3.2). Here ‘plugging together’ means superimposing the starting node of one machine on the halting node of the other: identifying the two nodes. [Number the states of the first machine 1 through \(n\), and those of the second machine \((n - 1) + 1\) through \((n - 1) + 12\), which is to say \(n\) through \(n + 11\). This is the same process we described in terms of lists of instructions rather than flow charts in our proof of Theorem 4.2.] The result is shown in Figure 4-2.

The result is an \((n + 11)\)-state machine with productivity \(2n\). Since there may well be \((n + 11)\)-state machines with even greater productivity, we are not entitled to conclude that the most productive \((n + 11)\)-state machine has productivity exactly \(2n\), but we are entitled to conclude that the most productive \((n + 11)\)-state machine has productivity at least \(2n\).

So much for the pieces. Now let us put them together into a proof that the function \(p\) is not Turing computable. The proof will be by reductio ad absurdum: we deduce an absurd conclusion from the supposition that there is a Turing machine computing \(p\).

The first thing we note is that if there is such a machine, call it \(BB\), and the number of its states is \(j\), then we have

\[
P(n + 2j) \geq P(p(n))
\]

for any \(n\). For given a \(j\)-state machine \(BB\) computing \(p\), we can plug together an \(n\)-state machine writing a row of \(n\) strokes with two replicas of \(BB\) as in Figure 4-3.

The result is an \((n + 2j)\)-state machine of productivity \(P(p(n))\). Now from Example 4.5 above it follows that if \(a < b\), then \(p(a) < p(b)\). Turning this around,
if \( p(b) \leq p(a) \), we must have \( b \leq a \). Applying this observation to (1), we have

\[
(2) \quad n + 2j \geq p(n)
\]

for any \( n \). Letting \( i \) be as in Example 4.6 above, we have

\[
(3) \quad p(m + i) \geq 2m
\]

for any \( m \). But applying (2) with \( n = m + i \), we have

\[
(4) \quad m + i + 2j \geq p(m + i)
\]

for any \( m \). Combining (3) and (4), we have

\[
(5) \quad m + i + 2j \geq 2m
\]

for any \( m \). Setting \( k = i + 2j \), we have

\[
(6) \quad m + k \geq 2m
\]

for any \( m \). But this is absurd, since clearly (6) fails for any \( m > k \). We have proved:

4.7 Theorem. The productivity function \( p \) is Turing uncomputable.

Problems

4.1 Is there a Turing machine that, started anywhere on the tape, will eventually halt if and only if the tape originally was not completely blank? If so, sketch the design of such a machine; if not, briefly explain why not.

4.2 Is there a Turing machine that, started anywhere on the tape, will eventually halt if and only if the tape originally was completely blank? If so, sketch the design of such a machine; if not, briefly explain why not.

4.3 Design a copying machine of the kind described at the beginning of the proof of theorem 4.2.

4.4 Show that if a two-place function \( g \) is Turing computable, then so is the one-place function \( f \) given by \( f(x) = g(x, x) \). For instance, since the multiplication function \( g(x, y) = xy \) is Turing computable, so is the square function \( f(x) = x^2 \).

4.5 A universal Turing machine is a Turing machine \( U \) such that for any other Turing machine \( M_n \) and any \( x \), the value of the two-place function computed by \( U \) for arguments \( n \) and \( x \) is the same as the value of the one-place function computed by \( M_n \) for argument \( x \). Show that if Turing’s thesis is correct, then a universal Turing machine must exist.