N. J. Cutland

Computability

An introduction to recursive function theory

COMPUTABILITY

An introduction to recursive function theory

NIGEL CUTLAND

Department of Pure Mathematics, University of Hull

CAMBRIDGE UNIVERSITY PRESS

Cambridge

London New York New Rochelle

Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
32 East 57th Street, New York, NY 10022, USA
296 Beaconsfield Parade, Middle Park, Melbourne 3206, Australia

First published 1980

Printed in Great Britain by J. W. Arrowsmith Ltd., Bristol

Library of Congress Cataloguing in Publication Data

Cutland, Nigel.
Computability: an introduction to recursive function theory.

Bibliography: p.
Includes index.
1. Computable functions. 2. Recursion theory. I. Title.
QA9.59.C87 519.4 79-51823

ISBN 0 521 22384 9 hard covers
ISBN 0 521 29465 7 paperback

COMPUTABILITY

Preface

The emergence of the concept of a *computable function* over fifty years ago marked the birth of a new branch of mathematics: its importance may be judged from the fact that it has had applications and implications in fields as diverse as computer science, philosophy and the foundations of mathematics, as well as in many other areas of mathematics itself. This book is designed to be an introduction to the basic ideas and results of computability theory (or recursion theory, as it is traditionally known among mathematicians).

The initial purpose of computability theory is to make precise the intuitive idea of a computable function; that is, a function whose values can be calculated in some kind of automatic or effective way. Thereby we can gain a clearer understanding of this intuitive idea; and only thereby can we begin to explore in a mathematical way the concept of computability as well as the many related ideas such as decidability and effective enumerability. A rich theory then arises, having both positive *and* negative aspects (here we are thinking of *non*-computability and *un*decidability results), which it is the aim of this book to introduce.

We could describe computability theory, from the viewpoint of computer science, as beginning with the question What can computers do *in principle* (without restrictions of space, time or money)? – and, by implication – What are their inherent theoretical limitations? Thus this book is *not* about real computers and their hardware, nor is it about programming languages and techniques. Nevertheless, our subject matter is part of the theoretical background to the real world of computers and their use, and should be of interest to the computing community.

For the basic definition of computability we have used the 'idealised computer' or register machine approach; we have found that this is readily grasped by students, most of whom are aware of the idea of a computer. (We do not, however, assume such an awareness (although it is helpful) and even less do we assume any practical experience with computers or calculators.) Our approach is mathematically equivalent to the many others that have been discovered, including Turing machines, the favourite of many. (We discuss these equivalences in chapter 3.)

This text grew out of a course given to undergraduates in mathematics and computer science at the University of Hull. The reader envisaged is a mathematics student with no prior knowledge of this subject, or a student of computer science who may wish to supplement his practical expertise with something of the theoretical background to his subject. We have aimed at the second or third year undergraduate level, although the earlier chapters covering the basic theory (chapters 1–7) should be within the grasp of good students in sixth forms, high schools and colleges (and their teachers). The only prerequisites are knowledge of the mathematical language of sets and functions (reviewed in the Prologue) and the ability to follow a line of mathematical reasoning.

The later chapters (8–12) are largely independent of each other. Thus a short introductory course could consist of chapters 1–7 supplemented by selection according to taste from chapters 8–12. It has been our aim in these later chapters to provide an introduction to some of the ramifications and applications of basic computability theory, and thereby provide a stepping stone towards more advanced study. To this end, the final chapter contains a brief survey of possible directions for further study, and some suggestions for further reading. (The two main texts that might be regarded as natural sequels to this one are M. L. Minsky, *Computation: Finite and Infinite Machines*, which would complement the present volume by its broad and comprehensive study of *computation* (as opposed to computability), and H. Rogers, *Theory of Recursive Functions and Effective Computability*, which provides a more advanced treatment of recursion theory in depth.)

Many people have helped towards the writing of this book. I would first thank John Cleave, who taught me recursive function theory in a graduate course at the University of Bristol in 1966, and introduced me to the register machine approach that I have used here. I have greatly appreciated the sustained interest and encouragement from Stan Wainer (who also made valuable suggestions for the material in chapters 10 and 12) and David Jordan: I thank them. I would also like to thank David Jordan and Dick Epstein for reading a draft of the manuscript and making many valuable comments and corrections. I am grateful to the Cambridge University Press for their interest and advice which has resulted in the emergence of the completed manuscript.

Finally, a big thank you to my wife Mary for her patience and encouragement during the many phases of writing and preparation of this book; her idealism and understanding have been a sustaining influence throughout.

Contents

Preface viii

Prologue. Prerequisites and notation 1

1 Sets 1
2 Functions 2
3 Relations and predicates 4
4 Logical notation 4
5 References 5

1 Computable functions 7

1 Algorithms, or effective procedures 7
2 The unlimited register machine 9
3 URM-computable functions 16
4 Decidable predicates and problems 22
5 Computability on other domains 23

2 Generating computable functions 25

1 The basic functions 25
2 Joining programs together 25
3 Substitution 29
4 Recursion 32
5 Minimalisation 42

3 Other approaches to computability: Church's thesis 48

1 Other approaches to computability 48
2 Partial recursive functions (Gödel–Kleene) 49
3 A digression: the primitive recursive functions 51
4 Turing-computability 52
5 Symbol manipulation systems of Post and Markov 57
6 Computability on domains other than $\mathbb{N}$ 65
7 Church's thesis 67

4 Numbering computable functions 72

1 Numbering programs 72
2 Numbering computable functions 76
3 Discussion: the diagonal method 79
4 The *s*–*m*–*n* theorem 81

5 Universal programs 85
1 Universal functions and universal programs 85
2 Two applications of the universal program 90
3 Effective operations on computable functions 93
Appendix. Computability of the function σ_n 95

6 Decidability, undecidability and partial decidability 100
1 Undecidable problems in computability 101
2 The word problem for groups 106
3 Diophantine equations 107
4 Sturm's algorithm 108
5 Mathematical logic 109
6 Partially decidable predicates 112

7 Recursive and recursively enumerable sets 121
1 Recursive sets 121
2 Recursively enumerable sets 123
3 Productive and creative sets 133
4 Simple sets 140

8 Arithmetic and Gödel's incompleteness theorem 143
1 Formal arithmetic 143
2 Incompleteness 146
3 Gödel's incompleteness theorem 149
4 Undecidability 155

9 Reducibility and degrees 157
1 Many–one reducibility 158
2 Degrees 161
3 m-complete r.e. sets 165
4 Relative computability 167
5 Turing reducibility and Turing degrees 174

10 Effective operations on partial functions 182
1 The second Recursion theorem 182
2 Effective operations on computable functions 189
3 The first Recursion theorem 192
4 An application to the semantics of programming languages 196

11 The second Recursion theorem 200
1 The second Recursion theorem 200
2 Discussion 207
3 Myhill's theorem 210

12 Complexity of computation 212
1 Complexity and complexity measures 213
2 The Speed-up theorem 218
3 Complexity classes 223
4 The elementary functions 225

13 Further study 236
Bibliography 239
Index of notation 241
Subject Index 246

Prologue
Prerequisites and notation

The only prerequisite to be able to read this book is familiarity with the basic notations of sets and functions, and the basic ideas of mathematical reasoning. Here we shall review these matters, and explain the notation and terminology that we shall use. This is mostly standard; so for the reader who prefers to move straight to chapter 1 and refer back to this prologue only as necessary, we point out that we shall use the word function to mean a *partial* function in general. We discuss this more fully below.

1. Sets

Generally we shall use capital letters $A, B, C, \ldots$ to denote sets. We write $x \in A$ to mean that x is a member of A, and we write $x \notin A$ to mean that x is not a member of A. The notation $\{x: \ldots x \ldots\}$ where $\ldots x \ldots$ is some statement involving x means the set of all objects x for which $\ldots x \ldots$ is true. Thus $\{x$: x is an even natural number$\}$ is the set $\{0, 2, 4, 6, \ldots\}$.

If A, B are sets, we write $A \subseteq B$ to mean that A is contained in B (or A is a *subset* of B); we use the notation $A \subset B$ to mean that $A \subseteq B$ but $A \neq B$ (i.e. A is a *proper subset* of B). The *union* of the sets A, B is the set $\{x: x \in A$ or $x \in B$ (or both)$\}$, and is denoted by $A \cup B$; the *intersection* of A, B is the set $\{x: x \in A$ and $x \in B\}$ and is denoted by $A \cap B$. The *difference* (or relative complement) of the sets A, B is the set $\{x: x \in A$ and $x \notin B\}$ and is denoted by $A \setminus B$.

The empty set is denoted by $\emptyset$. We use the standard symbol $\mathbb{N}$ to denote the set of natural numbers $\{0, 1, 2, 3, \ldots\}$. If A is a set of natural numbers (i.e. $A \subseteq \mathbb{N}$) we write $\bar{A}$ to denote the complement of A relative to $\mathbb{N}$, i.e. $\mathbb{N} \setminus A$. We write $\mathbb{N}^+$ for the set of positive natural numbers $\{1, 2, 3, \ldots\}$, and as usual $\mathbb{Z}$ denotes the set of integers.

We write (x, y) to denote the *ordered pair* of elements x and y; thus $(x, y) \neq (y, x)$ in general. If A, B are sets, the *Cartesian product* of A and B is the set $\{(x, y): x \in A \text{ and } y \in B\}$, and is denoted by $A \times B$.

More generally, for elements $x_1, \ldots, x_n$ we write $(x_1, \ldots, x_n)$ to denote the *ordered n-tuple* of $x_1, \ldots, x_n$; an n-tuple is often represented by a single boldfaced symbol such as $\boldsymbol{x}$. If $A_1, \ldots, A_n$ are sets we write $A_1 \times \ldots \times A_n$ for the set of n-tuples $\{(x_1, \ldots, x_n): x_1 \in A_1 \text{ and } x_2 \in A_2 \ldots x_n \in A_n\}$. The product $A \times A \times \ldots \times A$ (n times) is abbreviated by A^n; A^1 means A.

2. **Functions**

We assume familiarity with the basic idea of a function, and the distinction between a function f and a particular value $f(x)$ at any given x where f is defined.[1] If f is a function, the *domain* of f is the set $\{x: f(x) \text{ is defined}\}$, and is denoted $\mathrm{Dom}(f)$; we say that $f(x)$ is *undefined* if $x \notin \mathrm{Dom}(f)$. The set $\{f(x): x \in \mathrm{Dom}(f)\}$ is called the *range* of f, and is denoted by $\mathrm{Ran}(f)$. If A and B are sets we say that f is a function *from A to B* if $\mathrm{Dom}(f) \subseteq A$ and $\mathrm{Ran}(f) \subseteq B$. We use the notation $f: A \rightarrow B$ to mean that f is a function from A to B with $\mathrm{Dom}(f) = A$.

A function f is said to be *injective* if whenever $x, y \in \mathrm{Dom}(f)$ and $x \neq y$, then $f(x) \neq f(y)$. If f is injective, then f^{-1} denotes the *inverse* of f, i.e. the unique function g such that $\mathrm{Dom}(g) = \mathrm{Ran}(f)$ and $g(f(x)) = x$ for $x \in \mathrm{Dom}(f)$. A function f from A to B is *surjective* if $\mathrm{Ran}(f) = B$.

If $f: A \rightarrow B$, we say that f is an *injection* (from A to B) if it is injective, and a *surjection* (from A to B) if it is surjective. It is a *bijection* if it is both an injection and a surjection.

Suppose that f is a function and X is a set. The *restriction* of f to X, denoted by $f|X$, is the function with domain $X \cap \mathrm{Dom}(f)$ whose value for $x \in X \cap \mathrm{Dom}(f)$ is $f(x)$. We write $f(X)$ for $\mathrm{Ran}(f|X)$. If Y is a set, then the *inverse image of Y under f* is the set $f^{-1}(Y) = \{x: f(x) \in Y\}$. (Note that this is defined even when f is not injective.)

If f, g are functions, we say that g *extends* f if $\mathrm{Dom}(f) \subseteq \mathrm{Dom}(g)$ and $f(x) = g(x)$ for all $x \in \mathrm{Dom}(f)$: in short, $f = g|\mathrm{Dom}(f)$. This is written $f \subseteq g$.

[1] Usually in mathematical texts a function f is defined to be a set of ordered pairs such that if $(x, y) \in f$ and $(x, z) \in f$, then $y = z$, and $f(x)$ is defined to be this y. We do not insist on this definition of a function, but our exposition is consistent with it.

The *composition* of two functions f, g is the function whose domain is the set $\{x: x \in \mathrm{Dom}(g) \text{ and } g(x) \in \mathrm{Dom}(f)\}$, and whose value is $f(g(x))$ when defined. This function is denoted $f \circ g$.

We denote by $f_\emptyset$ the function that is defined nowhere; i.e. $f_\emptyset$ has the property that $\mathrm{Dom}(f_\emptyset) = \mathrm{Ran}(f_\emptyset) = \emptyset$. Clearly $f_\emptyset = g \mid \emptyset$ for any function g.

Often in computability we shall encounter functions, or expressions involving functions, that are not always defined. In such situations the following notation is very useful. Suppose that $\alpha(\boldsymbol{x})$ and $\beta(\boldsymbol{x})$ are expressions involving the variables $\boldsymbol{x} = (x_1, \ldots x_n)$. Then we write

$$\alpha(\boldsymbol{x}) \simeq \beta(\boldsymbol{x})$$

to mean that for any $\boldsymbol{x}$, the expressions $\alpha(\boldsymbol{x})$ and $\beta(\boldsymbol{x})$ are either both defined, or both undefined, and if defined they are equal. Thus, for example, if f, g are functions, writing $f(x) \simeq g(x)$ is another way of saying that $f = g$; and for any number y, $f(x) \simeq y$ means that $f(x)$ is defined and $f(x) = y$ (since y is always defined).

Functions of natural numbers For most of this book we shall be concerned with functions of natural numbers; that is, functions from $\mathbb{N}^n$ to $\mathbb{N}$ for various n, most commonly $n = 1$ or 2.

A function f from $\mathbb{N}^n$ to $\mathbb{N}$ is called an *n-ary* function. The value of f at an n-tuple $(x_1, \ldots, x_n) \in \mathrm{Dom}(f)$ is written $f(x_1, \ldots, x_n)$, or $f(\boldsymbol{x})$, if $\boldsymbol{x}$ represents $(x_1, \ldots, x_n)$. In some texts the phrase *partial function* is used to describe a function from $\mathbb{N}^n$ to $\mathbb{N}$ whose domain is not necessarily the whole of $\mathbb{N}^n$. For us the word function *means* partial function. On occasion we will, nevertheless, write *partial function* to emphasise this fact. A *total function* from $\mathbb{N}^n$ to $\mathbb{N}$ is a function whose domain is the whole of $\mathbb{N}^n$.

Particularly with number theoretic functions, we shall blur the distinction between a function and its particular values in two fairly standard and unambiguous ways. First we shall allow a phrase such as 'Let $f(x_1, \ldots x_n)$ be a function ...' as a means of indicating that f is an n-ary function. Second, we shall often describe a function in terms of its general value when this is given by a formula. For instance, '*the function* x^2' means 'the unary function f whose value at any $x \in \mathbb{N}$ is x^2'; similarly, '*the function* $x + y$' is the binary function whose value at $(x, y) \in \mathbb{N}^2$ is $x + y$.

We describe the zero function $\mathbb{N} \to \mathbb{N}$ by $\mathbf{0}$; and generally, for $m \in \mathbb{N}$, we denote the function $\mathbb{N} \to \mathbb{N}$ whose value is always m by the boldface symbol $\boldsymbol{m}$.

3. Relations and predicates

If A is a set, a property $M(x_1, \ldots, x_n)$ that holds (or is true) for some n-tuples from A^n and does not hold (or is false) for all other n-tuples from A is called an n-ary *relation* or *predicate* on A.[2]

For example, the property $x < y$ is a binary relation (or predicate) on $\mathbb{N}$; $2 < 3$ holds (or is true) whereas $9 < 5$ does not hold (or is false). As another example, any n-ary function f from $\mathbb{N}^n$ to $\mathbb{N}$ gives rise to an $(n+1)$-ary predicate $M(\mathbf{x}, y)$ given by

$$M(x_1, \ldots, x_n, y) \text{ if and only if } f(x_1, \ldots, x_n) \simeq y.$$

Equivalence relations and orders (The student unfamilar with these notions may prefer to delay reading this paragraph until it is needed in chapter 9.) In chapter 9 we shall encounter two special kinds of relations on a set A.

(*a*) A binary relation R on a set A is called an *equivalence relation* if it has the following properties for all $x, y, z \in A$:

(i) (reflexivity) $R(x, x)$;
(ii) (symmetry) if $R(x, y)$ then $R(y, x)$;
(iii) (transitivity) if $R(x, y)$ and $R(y, z)$ then $R(x, z)$.

We think of $R(x, y)$ as saying that x, y are equivalent (in some particular sense). Then we define the *equivalence class of* x as the set $\{y : R(x, y)\}$, consisting of all things equivalent to x.

(*b*) A binary relation R on a set A is called a *partial order* if, for all $x, y, z \in A$,

(i) (irreflexivity) not $R(x, x)$;
(ii) (transitivity) if $R(x, y)$ and $R(y, z)$ then $R(x, z)$.

A partial order is usually denoted by the symbol $<$, and we write $x < y$ rather than $<(x, y)$. A partial order is often defined by first defining $\leq$ (meaning $<$ or $=$), with the properties

(i) $x \leq y$;
(ii) if $x \leq y$ and $y \leq x$ then $x = y$;
(iii) $\leq$ is transitive;

and then defining $x < y$ to mean $x \leq y$ and $x \neq y$.

4. Logical notation

Our logical notation and usage will be standard throughout. We use the word *iff* as an abbreviation for if and only if. The symbol $\equiv$

[2] Often an n-ary relation or predicate $M(\mathbf{x})$ on a set A is identified with the set $\{\mathbf{x} : \mathbf{x} \in A^n$ and $M(\mathbf{x})$ holds$\}$. We do not insist on this identification here, although our exposition is consistent with this approach.

denotes definitional equivalence, while $\Rightarrow$ denotes implies, and $\Leftrightarrow$ denotes implies and is implied by. We use the symbols $\forall$, $\exists$ to mean 'for all' and 'there exists' in the standard way.

The symbol $\square$ is used in the text to indicate the end of a proof.

5. **References**

Each chapter is divided into sections, and items in each section are numbered consecutively. A reference such as theorem 5-1.4 means theorem 1.4 of chapter 5: this is the fourth numbered item of § 1 in that chapter. When referring within a chapter the number of the chapter is omitted. Exercises are included in this system of numbering. Thus exercise 6-1.8(2) means the second exercise of exercises 1.8, found in chapter 6.

Reference to entries in the bibliography is made by citing the author and year of publication of the work referred to.

1
Computable functions

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

1. Algorithms, or effective procedures

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

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

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

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