Computability Theory and Applications: The Art of Classical Computability

Robert Irving Soare
Department of Mathematics
The University of Chicago

VOLUME I draft1246

December 22, 2011

Contents

Dedication to Alan M. Turing, 1912–1954

In 1935 a twenty-three year old graduate student heard the lectures of M.H.A. Newman at the University of Cambridge on Gödel's incompleteness theorem [1931] and on Hilbert's *Entscheidungsproblem* (decision problem). Turing gave his solution to the incredulous Newman on April 15, 1936. Turing conceived his *automatic machine (a-machine)* as an idealized typewriter with an infinite carriage, a reading head, and a very simple finite program. He produced noncomputable real numbers, and showed that the Hilbert's decision problem has no algorithm. Turing [1936] then gave a convincing demonstration that every effectively calculable function is Turing computable. Turing's extraordinary analysis of a computation process was enthusiastically accepted by the other founders of the subject, Gödel, Kleene, and Church. Later Turing [1939] proposed an oracle machine (o-machine) which led to relative computability (Turing reducibility), the single most important concept in computability theory and the one most relevant to modern computing.

> "That this really is the correct definition of mechanical computability was established beyond any doubt by Turing."
>
> *-Kurt Gödel* 193? *Notes in Gödel Nachlass [1935]*
>
> " But I was completely convinced only by Turing's paper."
>
> *-Gödel: letter to Kreisel of May 1, 1968 [Sieg, 1994, p. 88].*

> " . . . one [Turing] has for the first time succeeded in giving an absolute definition of an interesting epistemological notion, *i.e.*, one not depending on the formalism chosen."
>
> *-Gödel, Princeton Bicentennial, [1946, p. 84].*

> "Turing's computability is intrinsically persuasive" but "λ-definability is not intrinsically persuasive" and "general recursiveness scarcely so (its author Gödel being at the time not at all persuaded)."
>
> *-Stephen Cole Kleene [1981b, p. 49]*

> Of the three different notions: computability by a Turing machine, general recursiveness of Herbrand-Gödel-Kleene, and λ-definability, "The first has the advantage of making the identification with effectiveness in the ordinary (not explicitly defined) sense evident immediately—*i.e.*, without the necessity of proving preliminary theorems."
>
> *-Alonzo Church, in his review [1937] of Turing [1936].*

Preface

The subtitle of this book, *The Art of Classical Computability,* emphasizes three very important concepts: (1) *computability* (as opposed to recursion or induction); (2) *classical* computability (algorithmic functions on certain countable structures in the original sense of Turing and Post); and (3) the *art* of computability: a skill to be practiced, but also important an esthetic sense of beauty and taste in mathematics.

Classical Computability Theory

Classical computability theory is the theory of functions on the integers computable by a finite procedure. This includes computability on many countable structures since they can be coded by integers. Computable functions include the recursive functions of Gödel [1934], and the Turing machines presented in Turing [1936], as well as the advances in the 1930's by Church, Kleene, and Post. Turing [1939, §4] very briefly suggested the concept of an *oracle Turing machine (o-machine),* a kind of local Turing machine which can access an *external database,* which Turing called an *oracle.* Computability from a database is the model for much of our computing in the real world, such as accessing the internet.

Post [1944] developed this into *Turing reducibility (relative computability)* of a set B from a set A (written $B \leq_{\mathrm{T}} A$) if a Turing machine with A on its oracle tape can compute B. Post [1948] defined the *Turing degree* of a set A to be the collection of all sets Turing equivalent to A, *i.e.*, coding exactly the same information as A. This led to the important program of measuring the information content not only of sets of integers, but of real numbers, algebraic structures, models, and algorithmic complexity and Kolmogorov complexity, and computable combinatorics such as Ramsey's Theorem. Post [1944] simultaneously developed the theory of *computably enumerable (c.e.)* sets, those which can be effectively generated. The c.e. sets correspond to

many problems in mathematics in number theory, finitely presented groups, differential geometry and other areas.

The Art of Classical Computability

Mathematics is an art as well as a science. We use the word "art" in two senses. First "art" means a *skill* or craft which can be acquired and improved by practice. For example, Donald Knuth wrote *The Art of Computer Programming,* a comprehensive monograph in several volumes on programming algorithms and their analysis. Similarly, the present book is intended to be a comprehensive treatment of the *craft* of computability in the sense of knowledge, skill in solving problems, and presenting the solution in the most comprehensible, elegant form. The sections have been rewritten over and over in response to comments by hundreds of readers about what was clear and what was not, so as to achieve the most elegant and easily understood presentation.

However, in a larger sense this book is intended to develop the art of computability as an *artistic endeavor*, and with appreciation of its mathematical beauty. It is not enough to state a valid theorem with a correct proof. We must see a sense of beauty in how it relates to what came before, what will come after, the definitions, why it is the right theorem, with the right proof, in the right place.

One of the most famous art treasures is Michelangelo's statue of *David* displayed in the Accademia Gallery in Florence. The long aisle to approach the statue is flanked with the statues of Michelangelo's unfinished slaves struggling as if to emerge from the block of marble. There are practically no details, and yet they possess a weight and power beyond their physical proportions. Michelangelo thought of himself, not as carving a statue, but as seeing the figure within the marble and then chipping away the marble to release it. The unfinished slaves are perhaps a more revealing example of this talent than the finished statue of David.

Similarly, it was Alan Turing [1936] and [1939] who saw the figure of computability in the marble more clearly than anyone else. Finding a formal definition for effectively calculable functions was the first step, but *demonstrating* that it captured computability was as much an artistic achievement as a purely mathematical one. Gödel himself had expressed doubt that it would be possible to do so. The other researchers thought in terms of mathematical formalisms like recursive functions, λ-definable functions, and arithmetization of syntax. It was Turing who saw the computer itself

in the marble, a simple intuitive device equipped with only a finite program and using only a finite sequence of strokes at each stage in a finite computation, the vision closest to our modern computer. Even more remarkable, Turing saw how to explicitly *demonstrate* that this mechanical device captured all effectively calculable processes. Gödel immediately recognized this achievement in Turing and in no one else.

The first aim of this book is to present the *craft* of computability, but the second and more important goal is to teach the reader to see the figure inside the block of marble. It is to allow the reader to understand the nature of a computable process, of a set which can be computably enumerated, of the process by which one set B is computed relative to another set A, of a method by which we measure the information content of a set, an algebraic structure, or a model, and how we approximate these concepts at a finite stage in a computable process.

The Mountaintop View

Understanding a mathematical book or paper or giving a paper or lecture related to what Leo Harrington calls *the mountaintop view*. As an example, suppose you are standing on the tenth floor of Evans Hall at the Berkeley Department of Mathematics, looking west over the San Franciso Bay. Someone asks you how to drive from the Berkeley campus to San Francisco. You do not give turn by turn instructions. You simply say: go west along University Avenue to the Bay; turn left on the Expressway; turn right on the Bay Bridge; and you are in San Francisco. Similarly, to explain a complicated mathematical paper you choose a small number of the most important elements, stress these, and let the other parts fall into place. Adopting this principle is closely tied to an artistic conception of the mathematical work, since this method is most likely to communicate the mathematical beauty to an audience.

Evolution of Terminology

The Term "Recursive"

When Dedekind [1888] proved that a definition by recursion uniquely defines a function, he called it "definition by induction." The term "recursive" arose in the first half of the 1900's. Hilbert [1904] used the term "rekurrent(e)," and in [1923] he used *"Rekursion"*. Skolem in [1923] showed that many

number-theoretic functions are primitive recursive, and he used *"rekurrierend."* According to Gandy the term *"recursive"* was apparently first used in English by Ramsey [1928].

The *concept* of recursion stems from the verb "recur," "to return to a place or status." The primary mathematical meaning of recursive has always been "defined by induction," as in §**??** and §**??** Scheme (V). Gödel [1931] used the German "rekursiv" to mean what we now call "primitive recursive," which we shall define in §**??**. After Gödel [1934] "recursive" formally meant "Herbrand-Gödel (general) recursive." The advantage of the Herbrand-Gödel definition of recursive function (§**??**) was that it encompassed recursion on an arbitrary number of arguments and allowed partial functions which primitive recursion did not.

The Term "Computable"

The term "computable" appears much earlier in 1646 in English usage according to the *Oxford English Dictionary* [O.E.D., 1989]. Both O.E.D. and *Webster's Third International Dictionary* [1993] give the definition of "computable" as roughly synonymous with "calculable," capable of being ascertained or determined by a mathematical process especially of some intricacy. The meaning of "calculate" is somewhat more general including "to figure out," "to design or adapt for a purpose," "to judge to be probable," while "compute" means more "to determine by a mathematical process," or "to determine or calculate by means of a computer." Turing [1936] used the term "computable" even in his title, and began his paper,

> "The 'computable numbers' may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means."

Gödel [1934] §9 also uses the term "computability" to explain the effective conditions he had added to the Herbrand definition, and uses the terms "computable" and "computability" in Gödel [1936] and [1946].

"Recursive" Acquires the Meaning "Computable"

From 1931 to 1934 Church and Kleene had been studying a class of effectively calculable functions called λ-*definable functions*. By 1934 Kleene had shown that a large class of number theoretic functions were λ-definable. On the strength of this evidence, Church proposed to Gödel around March, 1934

the first version of his Church's Thesis that the notion of "effectively calculable" be identified with "λ-definable," a suggestion which Gödel rejected as "thoroughly unsatisfactory."

In addition, it was difficult to present the results to a mathamatical audience in the formalism λ-definable functions, even though they became useful much later with the development of computers. Kleene [1981] wrote, "I myself, perhaps unduly influenced by rather chilly receptions from audiences around 1933–35 to disquisitions on λ-definability, chose, after general recursiveness had appeared, to put my work in that format. ...."

In 1935 Church changed the formal definition in Church's Thesis from "λ-definable" to "recursive," which was his abbreviation for the Herbrand-Gödel general recursive definition of Gödel [1934]. On April 19, 1935 Church presented to the American Mathematical Society his famous proposition published in Church [1936] and known since Kleene's book [1952] as *Church's Thesis* which asserts that the effectively calculable functions should be identified with the (Herbrand-Gödel) recursive functions. Gödel still refused to accept this identification even though it was phrased in terms of his own recursive functions.

In 1934 the terms and concepts had been completely clear and distinct. "Effectively calculable" meant the *informal* notion of "specified by a finite algorithm," and "recursive" meant the *formal* notion of "definable by the Herbrand-Gödel equations" and more generally "defined by some kind of induction." That distinction was about to become blurred for the next sixty years from 1936 to 1996. With the advent of Church [1936], Kleene [1936], Kleene [1943], and particularly the influential book Kleene [1952], "recursive" acquired the additional informal meaning "effectively calculable," and the field came to be known under Kleene as "recursive function theory" or simply "recursion theory" for short, even though Gödel objected that this term should be used with reference to the work Rosza Peter does (recursion on several arguments).

By 1936 Kleene and Church had begun thinking of the word "recursive" to mean "computable," or "calculable." Church had seen his first thesis rejected by Gödel and was heavily invested in the acceptance of his 1936 thesis in terms of recursive functions. Without the acceptance of this thesis Church had no unsolvable problem. Church wrote in [1936, p. 96], reprinted in Davis [1965], that a *"recursively enumerable set"* is one which is the range of a recursive function. This is apparently the first appearance of the term "recursively enumerable" in the literature and the first appearance of "recursively" as an adverb meaning "effectively" or "computably."

In the same year Kleene [1936, p. 238] cited in Davis [1965, p. 238]

mentioned a "recursive enumeration" and noted that there is no recursive enumeration of Herbrand-Gödel systems of equations which gives only the systems which define the (total) recursive functions. By a "recursive enumeration" Kleene states that he means "a recursive sequence (*i.e.*, the successive values of a recursive function of one variable)." Post [1944], under the influence of Church and Kleene, adopted this terminology of "recursive" and "recursively enumerable" over his own terminology of "effectively generated set," "normal set," "generated set." Thereafter, it became firmly established in the literature.

Ambiguity in the Term "Recursive"

The Kleene-Church assignment of "recursive" to mean "algorithmic" or "calculable" led to ambiguity. On one hand Kleene identified recursion theory with algorithmic functions and he wrote in Kleene [1988, p. 19], "The recognition of algorithms goes back at least to Euclid (c. 330 B.C.)." On the other hand, Kleene [1981b, p. 44] wrote of Dedekind [1988], (where induction was used to define addition and multiplication), "I think we can say that recursive function theory was born there ninety-two years ago with Dedekind's Theorem 126 ('Satz der Definition durch Induktion') that functions can be defined by primitive recursion." Did Kleene mean that recursion and inductive definitions began with Dedekind in 1888 or that computability and algorithms began there and not with Euclid? When Kleene used the term "recursive" to also mean "algorithmic," one was never sure whether a particular instance meant "algorithmic" or "inductive." Our language had become indistinct. When a speaker used the word "recursive" in 1990 in front of a general audience, did he mean "defined by induction," "related to fixed points and reflexive program calls," or did he mean "computable?"

Difficulty in Communication

The term "recursive" may have originally been an appropriate term for the subject in 1936 because some mathematicians and scientists understood that term in the sense of "inductive" and because the term "computer" had not yet become known to the general public. By 1990 it was the reverse. Those who had heard the term "recursive" associated it with the second of the elementary programming methods of iteration and recursion (*i.e.*, defined by induction). Very few associated it with the extended Kleene meaning of "defined by a finite procedure" or "calculable." In contrast, Turing had used his Turing machine model to construct a real digital computer during

1940-1945 for cryptanalysis, and John von Neumann used it to design the EDVAC computer architecture where the data and program are stored in the same address space. With the arrival of IBM's Personal Computer (PC) in 1981 the computer was transformed from an arcane object in a special room accessed by a stack of IBM cards in 1960 to a kind of electronic typewriter, capable of all kinds of calculations, compositions, and communication via the ethernet with other computers.

By 1990 the situation had become very difficult. Many people had access to a personal computer on their desks and the terminology of computing was familiar to the general population, but the term "recursive" was limited to very small number who mainly understood the wrong meaning. They mainly associated it with a first year programming course or a definition by induction on mathematics and almost never with computability. So few people understood the meaning of "recursive" that by 1990 we had to begin papers with, "Let f be a recursive function (that is, a computable function)," as if we were writing in Chinese and then translating back into English.

Furthermore, university students interested in computability failed to recognize the content from the course title "Recursive Function Theory" in the catalogue description, and never took the course. Writing letters of recommendation to place graduate students became more difficult because virtually no one on the hiring committee knew what "recursion theory" was.

The founders of the two key definitions of computabilty, Turing and Gödel, never used the word "recursive" to mean "computable," and objected when it was so used. When others did, Gödel reacted sharply negatively, as related by Martin Davis.

> In a discussion with Gödel at the Institute for Advanced Study in Princeton about 1952–54, I [Martin Davis] casually used the term "recursive function theory" as it was used then. "To my surprise, Gödel reacted sharply, saying that the term in question should be used with reference to the kind of work Rosza Peter did." (See R. Peter's work on recursions in [1934] and [1951].)

Making the Change to Computability

By 1995 the confusion had become great. Soare [1996] wrote an article *Computability and Recursion* for the *Bulletin of Symbolic Logic* on the history and scientific reasons why one should use "computable" and not "recursive" to mean "calculable." "Recursive" should mean "inductive" as it had for Dedekind and Hilbert. At first, few were willing to make such a dramatic

change, overturning a sixty year old tradition of Kleene, the most influential leader in computability after 1940. In 1996 the terms *"computability theory"* and *"computably enumerable (c.e.) set"* did not come "trippingly on the tongue" (to use the phrase in Shakespeare's Hamlet Act 3, Scene 2, 1–4) as they do now. However, in a few months more people were convinced by the logic of the situation. Three years later the A.M.S. conference in Boulder, Colorado had the title, *Computability Theory and Its Applications: Current Trends and Open Problems,* a title that would have been unthinkable a few years earlier. At that meeting, referenced in Soare [2000], most researchers had adopted the new terminology and conventions. Changing back from "recursive" to "computable" during 1996–1999 has had a number of advantages. Simultaneously, the Elsevier volume *Handbook of Recursion Theory* changed its name to *Handbook of Computability Theory* and solicited Soare's paper [1999b] as its lead paper.

Epilogue on Computability

Today the term "recursive" means "inductive" as it always has. Few use it to also mean "computable." The change was not merely a change of names but also a change of attitudes. Ironically, by 1990 the subject of Recursion Theory had been moving ever more inward and away from the general scientific public in spite of the applications mentioned above. Leo Harrington said, "Model Theory is about models, Set Theory is about sets, but Recursion Theory is not about recursion."

The change of names signalled a return to the concepts of Turing and Gödel with an emphasis on computability not induction. Books on computability sprang up with a different orientation, such as Cooper [2004] and Enderton [2011]. In the last decade Barry Cooper formed the organization *Computability in Europe (CiE)* which meets every summer at a different location and which several hundred people attend. The fields represented in the 2010 meeting included: Proof Theory and Computation, Computational Complexity, Computability of the Physical, Reasoning and Computation from Liebniz to Boole, Biological Computing, Web Applications and Computation. These are diverse fields but all have the concept of computability in common. The change of terminology has announced to those inside computability and those outside that we are open for communication. Cooper said it would not have been possible to have achieved such a large and diverse audience for the CiE meetings under the name "Recursion in Europe."

The Great Papers of Computability

During the 1930's, educators suggested that college students should read the great books of Western culture in the original. At the University of Chicago the principal proponents were President Robert Maynard Hutchins and his colleague Professor Mortimer Adler. The curriculum relied on *primary* sources as much as possible and a discussion under the supervision of a professor. For decades the Great Books Program became a hallmark of a University of Chicago education.

In the first two decades of Computability Theory from 1930 to 1950 the primary sources were papers not books. Most were reprinted in the book by Martin Davis [1965] *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems, and Computable Functions.* Of course, all of these papers are important, shaped the subject, and should be read by the serious scholar. However, many of these papers are written in a complicated mathematical style which is difficult for a beginner to comprehend. Nevertheless, at least two of these papers are of fundamental importance and are easily accessible to a beginning student. My criteria for selecting these papers are the following.

1. The paper must have introduced and developed a topic of fundamental importance to computability.

2. The topic and its development must be as important today as then.

3. The paper must be written in a clear, informal style, so appealing that any beginning student will enjoy reading it.

There are two papers in computability which meet these criteria.

Turing [1936] Especially §9

Turing's [1936] paper is probably the single most important paper in computability. It introduces the Turing machine, the universal machine, and demonstrates the existence of undecidable problems. It is most often used in mathematics and computer science to define computable functions. It is perhaps Turing's best known and most influential paper.

I am especially recommending Turing [1936] *The extent of the computable numbers,* §9, pp. 249–254 in Turing [1936]. Here Turing gives a demonstration that the numbers computable by a Turing machine "include all numbers which would naturally be regarded as computable." This is a brilliant demonstration and is necessary for the argument. Without it

we do not know that we have diagonalized against *all* potential decidable procedures and therefore we have no undecidable problems. Books on computability (including mine) rarely give this demonstration even though it is critical, perhaps because of its nonmathematical nature. Every student of computability should read this very short section.

Post [1944] Especially §11

Turing [1939] very briefly introduced the notion of an "oracle machine," a Turing machine which could consult an oracle tape (database), but he did not develop the idea. In his paper *Recursively enumerable sets of positive integers and their decision problems,* Emil Post Post [1944] developed two crucial ideas, the structure and information content of computably enumerable (c.e.) sets, and the idea of a set B being *reducible* to another set A.

Turing [1939] never thought of his oracle machine as a device for reducing one set to another. It was simply a local machine interacting with an external database as a laptop might query the Internet. Post was the first to turn the oracle machines into a reducibility of a set B to a set A, written $B \leq_{\mathrm{T}} A$, which Post generously called *Turing reducibility.* Post's entire paper is wonderfully written and easily accessible to a beginner. He begins with simpler reduciblities such as many-one and truth-table and works up to Turing reducibility which was not understood at the time.

The last section §11 *General (Turing) reducibility* is especially recommended. Here Post explored informally the idea of a c.e. set B being Turing reducible to another c.e. set A. For the next decade 1944–1954 Post continued to develop the notions of Turing reducibility and information content. Post [1948] introduced the idea of *degrees of unsolvability*, now called *Turing degrees*, which are the key to measuring information content of a set or algebraic structure. Post gave his notes to Kleene before his death in 1954. Kleene revised them and published Kleene-Post [1954] introducing a finite forcing argument as in Chapter 6 to define Turing incomparable sets each Turing computable in K. These two notions: *computability* by Turing's automatic machine (a-machine) in [1936]; and *reducibility* of one set B to another set A in Turing [1939] §4, and Post [1944]; are two of the *two most important* ideas in computability theory. Therefore, these papers are required should be read by anyone taking a course from this book.

There are other excellent computability papers reprinted in Martin Davis [1965] especially the Gödel incompleteness theorem [1931] with the improvement by Rosser, and the computability papers by Church [1936], Kleene [1936] and [1943]. Some of these papers may be difficult for a beginner to

read, but they will be more accessible after a first course in computability.

Acknowledgements

Among others I would like to thank colleagues and students for careful reading of preliminary versions of this book and for their suggestions and corrections: including my students at the University of Chicago: Russell Miller, Ken Harris, Chris Conidis, David Diamondstone, Damir Dzhafarov, Rachel Epstein, Karen Lange, Matthew Wright, Jonathan Stephenson, Eric Astor, William Chan; my University of Chicago colleagues: Denis Hirschfeldt, Joseph Mileti, Antonio Montalban, and Maryanthe Mallarias; colleagues at other universities: Ted Slaman, Richard Shore, Carl Jockusch, Douglas Cenzer, Leo Harrington, Manuel Lerman; to Steffen Lempp, and the Wisconsin faculty: Bart Kastermans, Arnie Miller, Joe Miller, and Wisconsin students: Asher Kach, Dan Turetsky; to Barbara Csima and her students at the University of Waterloo: Vladimir Soukharev, Jui-Yi Kao, Atul Sivaswamy, David Belanger, Carolyn Knoll. I am grateful to Piet Rodenburg, his colleague Tom Sterkenburg, and the students in Amsterdam: including Frank Nebel; to University of Notre Dame colleagues: Julia Knight and Peter Cholak; and their students: Joshua Cole, Yang Lu, Stephen Flood, Quinn Culver and John Pardo; to Valentina Harizanov and her graduate students: Jennifer Chubb and Sarah Pingrey; to Aaron Sterling at Iowa State University.

I am especially grateful to Barbara Csima of Waterloo, to Nathan Collins of the University of Wisconsin, and to Rebecca Steiner of Queens College, CUNY, for a very thorough and painstaking reading chapter by chapter and for detailed suggestions and corrections. Carl Jockusch read several of the advanced chapters in great detail and made a number of important mathematical corrections and suggestions. Damir Dzhafarov did an excellent job of drawing the diagrams in tikz.

[OTHERS will be added as the book progresses.]

Introduction

Hilbert's Famous Programs Set the Stage

By the 1890's Georg Cantor had published several papers on his new set theory, originally invented to solve a problem in Fourier analysis. Mathematicians were interested, but concerned over the famous paradoxes which soon emerged. More rigorous formulations of set theory with restricted rules of set formation avoided these paradoxes, but mathematicians feared that new contradictions might arise and they searched for a proof that these formulations of set theory were consistent.

In his epochal address to the *International Congress of Mathematicians* in 1900, David Hilbert, one of the two foremost mathematicians of the first third of the twentieth century, reduced the question of the consistency of geometry to that of the real numbers, and in 1904 posed the question of proving the consistency of arithmetic. This was to become one of his two main programs in logic and the foundations of mathematics from 1900 to 1930. The other was the *Entscheidungsproblem* (decision problem) which was to give a decision procedure for all valid sentences of first order logic (the predicate calculus), a program which emerged over several decades but was formulated clearly by Hilbert and Ackermann in 1928. A decision procedure here meant a decision procedure for much of mathematics which could be formulated in first order logic.

Gödel's Incompleteness Theorem [1931]

Hilbert retired in 1930, and gave a special address in Königsberg, his birth city, in which he repeated his insistence that there are no unsolvable problems. At the same conference, a quiet, unknown young man, Kurt Gödel, from Vienna, only a year past his Ph.D. announced a result which would seal the death of Hilbert's finite consistency program and forever transform the foundations of mathematics. Few understood the significance of Gödel's announcement, but one who did was John von Neumann. Already by 1930

John von Neumann was one of the leading mathematicians in Hilbert's group in Göttingen and after Hilbert's retirement was soon to become the most influential mathematician on a world scale.

Gödel's remarkable proof introduced for the first time the arithmetization of syntax (see §**??**), *i.e.*, the assignment of a code number (Gödel number) to every element of the syntax. This enabled him to demonstrate for any effectively axiomatized formal system T extending Peano Arithmetic (PA) a sentence which asserted its own unprovability. Therefore, T must be incomplete or inconsistent. Gödel used primitive recursive functions (see §**??**) to do the coding. Kleene later used Gödel's methods including primitive recursive functions for coding and arithmetization of syntax to prove the Enumeration Theorem **??** and the closely related Normal Form Theorem stated in Exercise **??**. Turing [1936] independently gave a more direct proof of the Enumeration Theorem by defining a *universal Turing machine* (**??**) which could simulate the behavior of any Turing machine and therefore gave a computable enumeration of partial computable functions. The Enumeration Theorem is essential for computability theory as we discuss in Remark **??**.

Recursive Functions Emerge

Although Gödel had used *primitive* recursive functions (§**??**) in his 1931 paper, he knew that Ackermann [1928] had shown the existence of a function defined by simultaneous recursion on *two* variables which was not primitive recursive but was clearly computable (see §**??**.) Building on a suggestion of Herbrand, Gödel [1934] developed a formal system called the *Herbrand-Gödel (HG) recursive functions* (**??**) which now are called simply the *recursive functions*. Later Kleene introduced an equivalent formalism, the *μ-recursive functions* (see §**??**) based on the five schemata for primitive recursive functions plus a sixth schema for unbounded search, whose usefulness depended heavily on Gödel's method for arithmetization of syntax in §**??**.

After seeing Gödel's definition, Church announced to the American Mathematical Society [1935] what Kleene later in his very influential book [1952] called *Church's Thesis* (§**??**), that the effectively calculable functions should be indentified with the (Herbrand-Gödel) recursive functions. On the basis of this thesis, Church [1936] announced the unsolvability of Hilbert's *Entscheidungsproblem*. Gödel rejected Church's thesis even though it was phrased in terms of Gödel's own formal model of HG-recursive functions and was partly based on his own arithmetization of syntax. Gödel was not

convinced that his recursive functions "comprised all possible recursions," and he thought that perhaps the thesis could not be proved but rather is a "heuristic principle."

Capturing Computability

In the spring of 1935 Alan Turing had attended lectures by Max Newman on Hilbert's *Entscheidungsproblem* and Gödel's Incompleteness Theorem. He worked intensely on the problem and brought his solution to Newman on April 15, 1936. Turing's *automatic machine (a-machine)* differed greatly from the other models, such as λ-definable functions or recursive functions, and was much closer to modern digital computers.

A Turing machine was a kind of idealized typewriter with an infinite carriage and a reading head moving back and forth one cell at a time (§**??**). Equally important was the informal but precise analysis in Turing [1936] of a human being calculating and how to simulate it with his a-machine. Gödel's reaction was immediate and overwhelmingly positive as described in the dedication. Turing [1937] proved that his a-machines were equivalent to the λ-definable functions and hence to the other formal models of computability. After 1936 most people accepted what we now call the *Church-Turing Thesis*: a function is intuitively computable iff it is Turing computable or equivalently if it is recursive (see §**??**).

Turing's Oracle Machines (o-Machines)

Since no one at Cambridge was working in computability and mathematical logic, Newman suggested that Turing go to Princeton and complete a Ph.D. with Church, which Turing did from 1936–1938. Turing [1939] contains his thesis. It was mainly about ordinal logics, a topic suggested by Church, but one page described an *oracle machine (o-machine)* which is of the greatest importance. The description was very brief and sketchy without detail or applications. With the start of World War II in September, 1939, Turing left academia for the British cryptographic world and never returned to computability theory.

Emil Post Develops Turing Reducibility

Turing's oracle machine concept lay dormant for five years until Emil Post's magnificent paper [1944] revived it, greatly expanded it, and cast the subject in an informal, intuitive light. This has been one of the most influential

papers on computability theory since 1936. We may imagine an oracle machine (o-machine) as an ordinary Turing a-machine with a read only *"oracle tape"* on which can be written the characteristic function of some set A as defined precisely in §**??**. The set B computed by this machine is said to be A-*computable*, or *Turing reducible* to A, written $B \leq_{\mathrm{T}} A$ (§**??**). We write $\Phi_e^A = B$ and regard Φ_e^A as the Turing functional computed by the e^{th} oracle program $\widetilde{P}_e$ (§**??**) with A on the oracle tape.

The oracle machines subsume the ordinary a-machines because if φ_e is computed by ordinary Turing program P_e, then there is an oracle machine Φ_i with oracle program $\widehat{P}_i$ such that $\varphi_e(x) = \Phi_i^A(x)$ where $A = \emptyset$ the empty set. But the oracle machines do much more. Turing reducibility is the *single most important concept* in computability theory because in the theory and applications we rarely prove results about computable functions on computable sets. We compare *noncomputable* (undecidable) sets B and A with respect to their relative information content. Turing reducibility gives us a precise measure of the information they encode relative to other sets and the *Turing degrees* (§**??**) are equivalence classes containing sets with the same information content. The thesis which was initiated briefly by Turing [1939, §4] and developed so eloquently by Post [1944, §11] is the most important thesis in the subject because it subsumes the others, and it gets to the heart of what researchers do in practice.

Post-Turing Thesis, Post [1944, §11]. A set B is effectively reducible to a set A in the intuitive sense if and only if B is Turing reducible to A.

Post Studies Computably Enumerable Sets

Turing and Church concentrated on formalisms for defining computable *functions,* but Post's formalism of production systems led him naturally to study *effectively generated sets,* those which could be effectively listed. The formal equivalent in modern terminology is that of a *computably enumerable (c.e.) set* defined as $W_e = \text{domain } \varphi_e$ in §**??** or equivalently one which can be listed (§**??**) by a computable function.

Post's Thesis (equivalent to the Church-Turing Thesis) is that a set is effectively generated if and only if it is c.e. Post recognized the widespread occurrence of c.e. sets in other branches of mathematics and asked whether (up to Turing equivalence) there was just one unsolvable problem for c.e. sets equivalent to the Gödel diagonal set $K = \{e : e \in W_e\}$, called the *complete set. Post's Problem* (see Definition **??**) was whether there is a noncomputable c.e. set $A <_{\mathrm{T}} K$. Post's problem and the results in his

paper [1944] had a great influence for decades.

Bounded Reductions

At first Post did not make much progress on the general case of Turing reducibility. To progress toward it he considered various stronger reducibilities. A Turing reduction $\Phi_e^A = B$ is a *bounded Turing* reduction (§**??**), written $B \leq_{\mathrm{bT}} A$, if there is a computable function $h(x)$ bounding the use function, *i.e.*, $\varphi_e^A(x) \leq h(x)$ where the use function $\varphi_e^A(x)$ is the maximum element *used* (scanned) during the computation (§**??**). For example, if B is many-one reducible to A, $B \leq_{\mathrm{m}} A$ by a computable function f, *i.e.*, $x \in B$ iff $f(x) \in A$, then the reduction is bounded by $h(x) = f(x)$. Post also introduced a *truth-table* reduction $B \leq_{\mathrm{tt}} A$ (§**??**) the case where Φ_e is not only a bounded Turing reduction but is also defined on all initial segments of $[0, h(x)]$. Along with various bounded reducibilities Post introduced c.e. sets A which, although coinfinite, had ever thinner complements, such as simple (§**??**) and hypersimple sets (§**??**).

Finally Understanding Turing Reducibility

Post realized that the bounded reducibilities and thin sets would not solve his problem. It required a deeper understanding of Turing reducibility. His understanding increased over the next decade after 1944. Post [1948] introduced the notion of *d*egree of unsolvability to collect into one equivalence class sets coding the *same information content* (§**??**). He wrote notes on his work. As he became terminally ill in 1954 Post gave them to Kleene who expanded them and published Kleene-Post [1954]. This paper was a fundamental advance toward solving Post's problem and toward understanding Turing reducibility. The key idea was the continuity of Turing functionals (§**??**) (also called the Use Principle Theorem **??**) that if $\Phi_e^A(x) = y$ then $\Phi_e^\sigma(x) = y$ for some finite initial segment $\sigma \prec A$ and then if $B \succ \sigma$ then $\Phi_e^B(x) = y$ also.

Using this, Kleene and Post constructed sets A and B in K such that $A \not\leq_{\mathrm{T}} B$ and $B \not\leq_{\mathrm{T}} A$. Hence, $\emptyset <_{\mathrm{T}} A <_{\mathrm{T}} K$ (see §**??**). This did not explicitly solve Post's Problem because the sets were not c.e., but Kleene and Post divided the conditions into requirements of the form $\Phi_e^A \neq B$ as in (**??**), which could be arranged in a priority list of order type ω and processed one at a time as in (**??**) using the Use Principle. This became the model for most arguments in the subject. It became the key step in

the later solution of Post's Problem by Friedberg [1957] and Muchnik[1956] since they combined this strategy with a computable approximation stage by stage as in §**??**.

From these ideas emerged the understanding that a Turing functional Φ_e is continuous as a map on Cantor space 2^ω and is not only continuous but *effectively continuous* (see §**??**) because the inverse image of a basic open set is the *computable* union of basic open sets (§**??**). Conversely, any classically continuous functional Ψ on Cantor space is effectively continuous (*i.e.*, a Turing functional) relative to some real parameter $X \subseteq \omega$ (see §**??**). This links the fundamental notion from computability theory to one of the most basic notions of analysis.

Computing in the Modern World

A Turing a-machine is an example of a *closed computing* process, like a calculator, which is given a program and an initial input but no further data during its computation. In contrast a Turing o-machine is an example of an *open computing* process which can receive additional data during the computation, just as a modern computer can access a database (oracle) such as the World Wide Web which is too large to be incorporated into the machine itself. Many modern computing processes are open computing processes and can be simulated only by an o-machine but not an a-machine.

Priority Arguments

The Kleene-Post construction had constructed finite initial segments $\sigma \prec A$ and $\tau \prec B$ such that for some x, $\Phi_e^A(x) \neq \tau(x)$. Hence $\Phi_e^A(x) \neq B(x)$. To make the sets A and B *computably enumerable* Muchnik [1956] and Friedberg [1957] had to abandon the K-oracle and computably enumerate the sets letting A_s be the finite set of elements enumerated in A by the end of stage s and likewise for B_s. They attempted to preserve strings $\sigma \prec A_s$ and $\tau \prec B_s$ when it seemed to give $\Phi_e^\sigma(x) \neq \tau(x)$. This action might later be *injured* because action by a higher priority requirement forces $\sigma \not\prec A_{s+1}$ causing this condtion for e to begin all over again. This finite injury method is presented in §**??**. These results led to much more complicated infinite injury arguments presented in Part II. To answer the riddle of how a requirement can be injured infinitely often and still be satisfied in the end go to Chapter 8.

Games in Computability Theory

In Part III we consider some games which shed light on computability and serve as a useful tool to discover and present theorems. The *Banach-Mazur games* in Chapter 11 had been used decades ago in point set topology to study dense open sets. They are used here to study the finite extension constructions presented in Chapter 6 and to help give a paradigm there to analyze extension constructions. Banach-Mazur games are useful when we are doing a construction computably in an *oracle*, but when we are constructing *computably enumerable* sets the construction must be completely decidable with *no oracle*.

Lachlan [1970] invented a new kind of game, called a *Lachlan game* to handle effective constructions which we present in Chapter 13. In a Lachlan game there are two players, Player I (RED), and Player II (BLUE), each of whom plays a finite or infinite collection of sets. Certain requirements are specified in advance. Each player may enumerate into the sets he is playing finitely many elements at every stage. At the end the game is won according to whether the sets satisfy the predetermined requirements or not. We present Lachlan games in Chapter 13, but the reader is advised to begin reading Chapter 13 as soon as possible because the methods will be helpful in most of the exercises as illustrated in Chapter 13 with references back to the earlier chapters.

Terminology and Notation

Most mathematicians believe that a fitting choice of terminology and notation is essential to mathematics. Georg Cantor, the founder of modern set theory wrote this.

> "I am extremely careful with the choice of those [*i.e.*, new notions], as I take the position that the development and propagation in no small degree depends on a fortunate and properly fitting terminology."

Philosopher Charles S. Peirce described the importance of language for science in *The Ethics of Terminology, Volume II, Elements of Logic,* p. 129.

> "... the woof and warp of all thought and all research is symbols, and the life of thought and science is the life inherent in symbols; so that it is wrong to say that a good language is *important* to good thought, merely; for it is of the essence of it."

Terminology

A function is *computable* if it is defined by a Turing machine and *recursive* if it is general recursive, *i.e.*, defined by a set of Herbrand-Gödel equations, or equivalently if it is μ-recursive as defined by Kleene. By the Church-Turing Thesis the Turing computable functions coincide with the algorithmically computable, *i.e.*, intuitively computable, or *effectively calculable* functions. We use the term *"computable"* in either of the two meanings: effectively calculable in the intuitive sense; or formally computable by a Turing machine.

We use the term "recursive" informally to mean "defined by induction" in the sense of §**??** and §**??**, especially by a procedure using the Primitive Recursion Scheme V of Definition **??**. We take the formal meaning of "recursive" to be Herbrand-Gödel recursive (§**??**) or equivalently Kleene μ-recursive (§**??**). We regard these as reflexive program calls, in which the program calls itself. We extend the meaning of reflexive program calls to include the Kleene Recursion Theorem **??**.

We do *not* use the term "recursive" to mean "effectively calculable" or "Turing computable," although the latter is extensionally (but not intensionally) equivalent to recursive. This attention to the *intensional meaning* of terms allows a more precise and historically more accurate usage of terms such as recursive and computable.

Notation

Notation will be defined when introduced. The Quick Finder Index is on page xxxiii. The Notation Index is in the Appendix Part **??**. We now summarize the most common notation.

Notation for Sets.

The universe is the set of nonnegative integers $\omega = \{0, 1, 2, 3, \dots\}$ which sometimes appears in the literature as $\mathbb{N}$. Most of the objects we study can be associated with some $n \in \omega$ called a "code number" or "Gödel number." We can think of operations on these objects as being presented by a corresponding function on these numbers and our functions will have domain and range contained in ω.

Upper case Latin letters A, B, C, D and X, Y, Z normally represent subsets of $\omega = \{0, 1, 2, 3, \dots\}$ with the usual set operations $A \cup B$, $A \cap B$; $|A|$, or $\mathrm{card}(A)$ denotes the cardinality of A; $\max(A)$ denotes the maximum element $x \in A$ if A is finite; $A \subseteq B$ denotes that A is a subset of B, and

$A \subset B$ that it is a *proper* subset; $A - B$ denotes the set of elements in A but not in B; $\overline{A} = \omega - A$, the complement of A; $A \sqcup B$ denotes the *disjoint union*, *i.e.*, $A \cup B$ provided that $A \cap B = \emptyset$; the *symmetric difference* is $A \,\Delta\, B = (A - B) \cup (B - A)$; a, b, c, $\ldots x$, y, z, $\ldots$ represent integers in ω; $A \times B$ is the cartesian product of A and B, the set of ordered pairs (x, y) such that $x \in A$ and $y \in B$; $\langle x, y \rangle$ is the integer which is the image of the pair (x, y) under the standard pairing function from $\omega \times \omega$ onto ω; $A \subseteq^* B$ denotes that $|A - B| < \infty$; $A =^* B$ denotes that $A \,\Delta\, B$ is finite; $A \subset_\infty B$ denotes that $|B - A| = \infty$. Given a simultaneous enumeration (see p. **??**) of A and B let $A \setminus B$ denote the set of elements enumerated in A before B and $A \searrow B = (A \setminus B) \cap B$, the set of elements appearing in A and later in B.

Logical Notation

We form predicates with the usual notation of logic where &, $\vee$, $\neg$, $\Longrightarrow$, $\exists$, $\forall$, μx denote respectively: and, or, not, implies, there exists, for all, and the least x: $(\mu x)\, R(x)$ denotes the least x such that $R(x)$ if it exists, and is undefined otherwise; $(\exists^\infty x)$ denotes "there exist infinitely many x," and $(\forall^\infty x)$ denotes "for almost all x" as in Definition **??**. These quantifiers are dual to each other. The latter is written $(\exists x_0)(\forall x \geq x_0)$. We use $x, y, z < w$ to abbreviate $x < w$, $y < w$, and $z < w$. In a partially ordered set we let $x \mid y$ denote that x and y are incomparable, *i.e.*, $x \not\leq y$ and $y \not\leq x$. We often use dots to abbreviate brackets before and after the principal connective of a logical expression. For example, $[\alpha \,.\Longrightarrow.\, \beta]$ abbreviates $[[\, \alpha \,] \Longrightarrow [\, \beta \,]]$. From the first expression, insert a right bracket to the left of $\Longrightarrow$ and a left bracket to the right and insert matching brackets at the beginning and end of formulas α and β. The periods increase readability of a long expression. TFAE abbreviates "The following are equivalent."

We use the usual Church *lambda notation* for defining partial functions. Suppose $[\ldots x \ldots]$ is an expression such that for any integer x the expression has at most one corresponding value y. Then $\lambda x\, [\ldots x \ldots]$ denotes the associated partial function $\theta(x) = y$, for example $\lambda x\, [\, x^2 \,]$. The expression $\lambda x\, [\uparrow]$ denotes the partial function which is undefined for all arguments. We also use the lambda notation for partial functions of k variables, writing $\lambda x_1 x_2 \ldots x_k$ in place of λx. An expression such as $\lambda x\, y\, [\, x + y \,]$, denotes addition as a function of x and y. However, $\lambda x\, [\, x + y \,]$, indicates that the expression is viewed as a function of x with y as a parameter, such as $\lambda x\, [x + 2]$. One advantage is that with an expression of several arguments, such as in the *s-n-m* Theorem **??** (Parameter Theorem) we can make clear which ar-

guments are variables and which are parameters, for example as explained in Remark **??**.

Notation for Strings and Functionals

We let $2^{<\omega}$ denote the the set of all finite sequences of 0's and 1's called strings and denoted by σ, ρ, and τ. Let 2^ω denote the set of all functions f from ω to $2 = \{0,1\}$, and ω^ω the set of all functions f from ω to ω. The integers $n \in \omega$ are *type 0* objects, (partial) functions $f \in 2^\omega$ or subsets $A \subseteq \omega$ (which are identified with their characteristic function $\chi_A \in 2^\omega$) are *type 1* objects, a (partial) *functional* Ψ is a map from type 1 objects to type 1 objects, *i.e.*, a map from 2^ω to 2^ω and is called a *type 2* object. Identifying a set A with its *characteristic function* χ_A we often write $A(x)$ for $\chi_A(x)$. Upper case Latin letters A, B, C, ..., represent subsets of ω. Script letters $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ represent subsets of 2^ω and are called *classes* to distinguish them from sets.

Partial Computable (P.C.) Functions

Let $\{P_e\}_{e\in\omega}$ be an effective numbering of all Turing machine programs (as in Definition **??**). We write $\varphi_e(x) = y$ if program P_e with input x halts and yields output y, in which case we say that $\varphi_e(x)$ *converges* (written $\varphi_e(x)\downarrow$), and otherwise $\varphi_e(x)$ *diverges* (written $\varphi_e(x)\uparrow$); $\{\varphi_e\}_{e\in\omega}$ is an effective listing of all *partial computable (p.c.)* functions; the domain and range of φ_e are denoted by $\mathrm{dom}(\varphi_e)$ and $\mathrm{rng}(\varphi_e)$. A set A is *computably enumerable (c.e.)* if it can be effectively listed, *i.e.*, if $A = \mathrm{dom}(\varphi_e)$ for some e.

If $\mathrm{dom}(\varphi_e) = \omega$ then φ_e is a *total computable function* (abbreviated *computable function*); we let f, g, h, ... denote total functions; $f \circ g$ or fg denotes the composition of functions, applying first g to an argument x and then applying f to $g(x)$. Let $f \restriction x$ denote the restriction to elements $y < x$ and $f \mathbin{\restriction\!\restriction} x$ the restriction to elements $y \leq x$.

Turing Functionals Φ_e^A

Let $\{\widetilde{P}_e\}_{e\in\omega}$ be an effective numbering of all Turing machine oracle programs, finite sets of sextuples defined in §**??**. Write $\Phi_e^A(x) = y$ if oracle program $\widetilde{P}_e$ with A on its oracle tape and input x halts and yields output y. Let the use function $\varphi_e^A(x)$ be the greatest element z for which the computation scanned the square $A(z)$ on the oracle tape. We regard Φ_e as a (partial) functional (type 2 object) from 2^ω to 2^ω mapping A to B if $\Phi_e^A = B$.

The use function $\varphi_e^A(x)$ has an exponent A to distinguish from the p.c. function $\varphi(x)$. They usually come in matched pairs, $\Psi^A(x)$ and $\psi^A(x)$, $\Theta^A(x)$ and $\theta^A(x)$, where the lower case function denotes the use function for the upper case functional. See Definition **??** (vi) for a function f as oracle in place of the set A.

Lachlan notation.

When $E(A_s, x_s, y_s, \ldots)$ is an expression with a number of arguments subscripted by s denoting their value at stage s, Lachlan has introduced the notation $E(A, x, y, \ldots)[\, s\,]$ to denote the evaluation of E where all arguments are taken with their values at the end of stage s.

(1) $\Phi_e^A(x)\,[\,s\,]$ denotes $\Phi_{e,s}^{A_s}(x_s)$ and $\varphi_e^A(x)\,[\,s\,]$ denotes $\varphi_{e,s}^{A_s}(x_s)$.

This Lachlan notation has become very popular and is now used in most papers and books.

Sections Marked $^{\oslash}$ and Exercises Marked $^{\diamond}$

Sections marked with the latex symbol $^{\oslash}$ (oslash) should be omitted in a first course simply to streamline the course so the core material can be completed in a quarter or semester. However, bright and highly motivated students are encouraged to study these sections anyway because they require no more mathematics than has been presented up to that point and are usually no more difficult than the rest. Sections in Chapters 1–7 *not* marked with $^{\oslash}$ are part of the *core* and normally should be covered in sequence in a course.

The symbol *diamond* $\diamond$ is used to denote greater *difficulty* in an exercise and should not be confused with the section symbol $^{\oslash}$. An exercise with no $\diamond$ is usually straightforward from the definitions and preceding text, and is often included to have a reference for an explicit statement without writing out fairly obvious proofs. An exercise with *one* diamond$^{\diamond}$ is harder and usually requires some thought. An exercise with two diamonds$^{\diamond\diamond}$ is harder still, and should be avoided in routine homework assignments. Hints to various exercises will be given in the back of the book in Part VI Appendix B on hints.

Quick Finder Index

[NOTE. THIS QUICK FINDER INDEX HAS BEEN ONLY BEGUN, AND WILL BE COMPLETED LATER.]

Introduction

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Part I

Foundations of Computability

Chapter 1

Defining Computability

Chapter 2

Computably Enumerable Sets

Chapter 3

Turing Reducibility

Chapter 4

The Arithmetical Hierarchy

Chapter 5

Classifying C. E. Sets

Chapter 6

Oracle Constructions and Forcing

Chapter 7

The Finite Injury Priority Method

Part II

Infinite Injury and the Tree Method

Chapter 8

Infinite Injury

Chapter 9

The Tree Method and Minimal Pairs

Chapter 10

More Tree Proofs

Part III

Games in Computability

Chapter 11

Banach-Mazur Games

Chapter 12

Gale-Stewart Games

Chapter 13

Lachlan Games

Chapter 14

Index Sets of C.E. Sets

Chapter 15

Pinball Machines

Part IV

Definability and Automorphisms

Chapter 16

Definable Properties of C.E. Sets

Chapter 17

Automorphisms of C.E. Sets

Part V

The History and Art of Computability

Chapter 18

The History of Computability

Chapter 19

Classical Computability and Classical Art

Part VI

Definitions and Hints

Appendix A

Standard Definitions

Appendix B

Hints for Exercises

Part VII

Bibliography

Part VIII

Indices

Computability theory, also known as recursion theory, is a branch of mathematical logic, of computer science, and of the theory of computation that originated in the 1930s with the study of computable functions and Turing degrees. The field has since expanded to include the study of generalized computability and definability. In these areas, recursion theory overlaps with proof theory and effective descriptive set theory. Basic questions addressed by recursion theory include Turing machines form the core of computability theory, or recursion theory as it is also known. This chapter introduces basic notions and results and readers already familiar with them can safely skip it. The exposition is based on standard references [18, 39, 67, 83, 109].Â In particular, if we choose classical general relativity theory as our background theory, then the above mentioned limitations (predicted by these Theses) become no more necessary, hence certain forms of the Church-Turing Thesis cease to be valid (in general relativity). (For other choices of the background theory the answer might be different.)