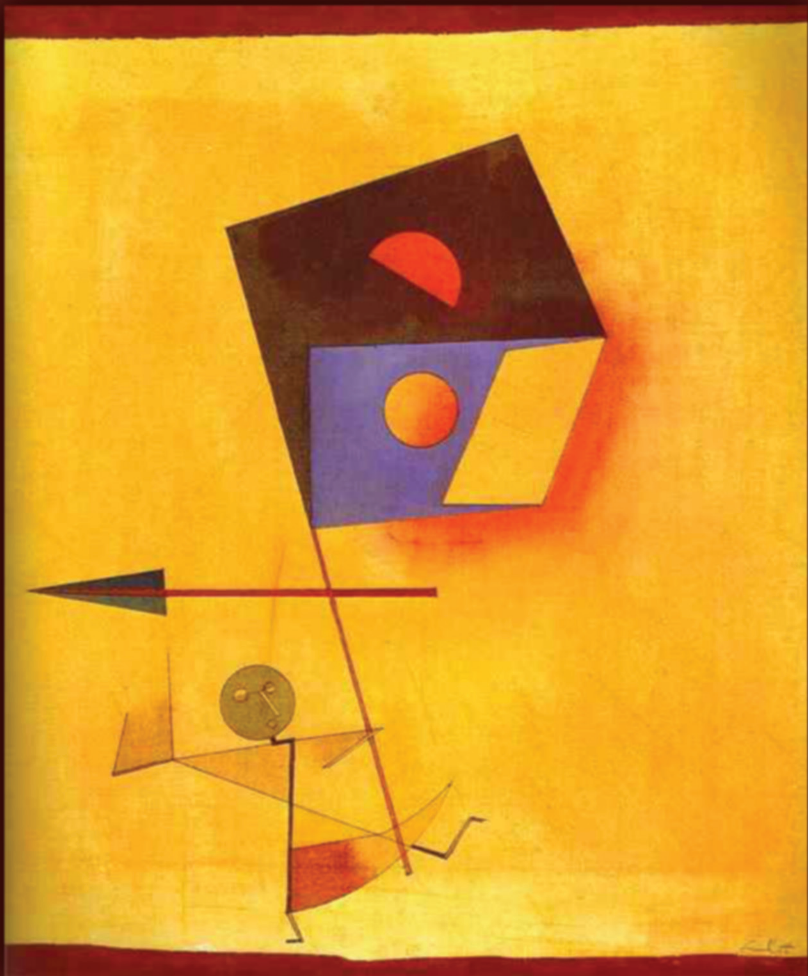
LIMITS OF COMPUTATION

An Introduction to the Undecidable and the Intractable



Edna E. Reiter
Clayton Matthew Johnson

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

LIMITS OF COMPUTATION

An Introduction to the Undecidable and the Intractable

LIMITS OF COMPUTATION

An Introduction to the Undecidable and the Intractable

Edna E. Reiter
Clayton Matthew Johnson

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20120808

International Standard Book Number-13: 978-1-4398-8207-8 (eBook - PDF)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Preface

To the Student: We think that the theory dealing with what is hard about computation (and what is impossible!) is challenging but fun. This book grows out of these ideas and our approach to teaching a course in computational complexity.

There is no doubt that some of the material in these chapters is what might be called "wrap your brain around it" material, where a first reaction might be that the authors are pulling off a trick like a magician pulling a rabbit out of a hat. For instance, consider the proof—using contradiction—that there can be no algorithm to tell whether a program written in C++ will go into an infinite loop. One reaction upon reaching the contradiction might be that there must be a misstep somewhere in the proof; another might be that there cannot really be a contradiction. Only after reading, rereading, and carefully considering each step can the student buy into the proof. There are no shortcuts here; this is not reading to be done with the television playing in the background.

There are also diversions here such as the bridges of Königsberg problem—interesting but easy, and useful to point out that there can be vast differences in the difficulty of problems that sound very much alike.

To the Instructor: We hope you have as much fun explaining the difficulties and complexities of computation as we do.

At California State University, East Bay, there is a required course for Computer Science Master of Science students in complexity theory.

This book grew out of the problems we had in choosing a text for this course. Sipser's *Introduction to the Theory of Computation* (Gale/Cengage Learning, 2006) has many good points, and we recommend it to all students as a reference—but it covers automata theory and formal languages (Part 1) as well as Computability Theory (Part 2) and Complexity (Part 3), which leads to less depth than we might like. And for a course that does not cover Part 1, the references back to it from Parts 2 and 3 are a bit problematic.

The Hopcroft, Motwani, and Ullman (2007) text is exemplary in many ways, and is another excellent reference; it is, however, more suitable to students in a doctoral program or those in an advanced course in complexity.

And of course, there is the wonderful Garey and Johnson (1979) text on NP-completeness. We love to point out to our computer science students that this book is still important and that every computer scientist should own a copy. Can you say that about any other book on technology of that vintage? There are also good, older books available, although not at a level appropriate for an M.S. student without a strong background in automata. (See Brainerd and Landweber, 1974; Harrison, 1978; Lewis and Papadimitriou, 1997; and Papadimitriou, 1994.)

We also think that the more popular books that mention Turing machines (such as those by Douglas Hofstader, 1979 and Penrose, 1989) are nice supplemental extracurricular material, although students must become adept at reading different models of Turing machines.

This book, then, is intended for advanced undergraduates or beginning graduate students who may not have a strong background in theoretical computer science and who do not plan to become experts in the area.

The book is designed so that essentially all of it can be covered in a one-quarter (4 hour/week) or one-semester (3 hour/week) course, with roughly half the course devoted to what is undecidable, and half to what is intractable. To do this, it may be necessary to omit some proofs (such as the proof that if $\mathbf{P} \neq \mathbf{NP}$, then there are problems in **NP** that are not **NP**-complete), but we feel that such proofs must be included in the text for the interested reader.

The authors have included a wide range of exercises. There is no way to learn this material without doing it. We feel strongly that the student should read this book with pencil in hand, filling in any missing details ("it is easily shown that …"). The student then needs to test his or her understanding by doing a variety of exercises, and he or she needs feedback that his or her approach to solving the exercises was (or was not) correct. We give weekly graded written assignments, and either one or two midterm exams in a 10-week quarter, plus a final exam. We have read at least one study on pedagogy stating that something must be learned three times in order to be mastered and for the learning to last: we hope that in doing a homework set, in studying for a midterm, and then again in studying for a final exam, we have forced our students into rethinking the material at least three times.

Acknowledgments

The authors wish to thank the many people who have helped in the path toward publication. First and foremost, there is the California State University, East Bay (CSUEB) office staff—Richard Uhler, who helped with many technical problems, and Susan Foye, Kimberly Cherry, and Erendira McDunn, who helped with the administrative details and kept the department running. We thank our colleagues Shirley Veomett, who helped with the illustration on function growth, and Steve Simon, who also teaches complexity. We greatly appreciate the efforts of Stan Wakefield, literary agent, and Amy Blalock, project coordinator and Linda Leggio, project editor at Taylor & Francis/CRC Press. We also thank our families and friends, particularly Jim Reiter and Grant Petersen, who supported us during the writing (and rewriting) of the book.

About the Authors

Drs. Edna Reiter and Clayton Matthew Johnson are faculty in the Department of Mathematics and Computer Science at California State University, East Bay (CSUEB) in the San Francisco Bay area. Together, they developed the subject matter of the CSUEB course Computation and Complexity, required for all students in the Master of Science program. The course covers the hard problems of computer science—those that are intractable or undecidable. The text and the exercises in the text have been tested on multiple sections of CSUEB students.

Edna E. Reiter, Ph.D., received M.S. degrees from the University of Michigan and the University of California at Davis, and a Ph.D. from the University of Cincinnati. Her initial research interests were in noncommutative ring theory, but she then became interested in the theoretical aspects of computer science. Dr. Reiter has discovered that she enjoys introducing students to these subjects. She is currently Chair of the Department of Mathematics and Computer Science at CSUEB..

Clayton Matthew Johnson, Ph.D., received an M.S. degree from Michigan State University, and holds a Ph.D. from the College of William and Mary in Virginia. His current research interests are genetic algorithms and machine learning. He teaches many of the core courses for both graduate and undergraduate students including data structures, automata theory, analysis of algorithms, and complexity. Dr. Johnson is the graduate coordinator for all M.S. students at CSUEB; he is also the incoming Chair of the Department of Mathematics and Computer Science.

Introduction

ALGORITHM: WHAT IS IT, WHEN DOES ONE EXIST?

Most of the computer science courses in the first several years of a student's experience are concerned with teaching algorithms, and getting students to implement them. The student of computer science spends many hours in many courses learning how to write programs to solve problems. She learns about sorting data, saving data in various data structures, maintaining databases, managing networks, creating graphics, and more. She learns algorithms for supporting an operating system, maintaining database purity, parsing source files, and so on. The question of unsolvable problems may not even appear—the instructor, the text, and the student are too busy learning the efficient algorithms that are in use.

Even the word *algorithm* is often used without a good definition—and expressing that definition will be a major theme of this text, one that cannot be answered in just a few paragraphs.

IMPORTANT QUESTIONS

Only in a few places does the computer science student study such questions as: What is computation? What is an algorithm? How do I know that this problem has a solution? If there is a solution, will it answer the problem fast enough? An answer in the next century is no better than no answer at all.

Any "educated" computer scientist needs to know something about the answers to these questions. There are many easy-to-state questions in computer science that either have no algorithm at all, or have no practical algorithm. These include:

- Given a context-free grammar G, is it ambiguous?
- What is the shortest route for a salesman to take, starting at his home, visiting all the cities on his route?
- Given a program written by another computer science student, will this program terminate? Or will it go into an infinite loop?
- Given a set of processes running on a system, will they end in deadlock?

These questions have intrigued computer scientists and others. Some popular books address some of the same questions as this text—see Hofstadter (1979) and Penrose (1991). Some questions have monetary prizes for anyone who can solve them (see Devlin, 2002).

DOES MY PROBLEM HAVE A SOLUTION? A GOOD SOLUTION?

The computer scientist should be aware of questions like these, and be suspicious of new assignments—if told to write code to solve problem X, it would be nice to know that X has a solution and that this solution will not require centuries or millennia to execute.

Questions to consider before beginning are: Is there a known good algorithm for this problem? Or, are there algorithms that work, but take too long? A problem that used to be difficult, but now has become easier, is the forecasting of weather. Weather (like many physical systems) obeys a complicated set of differential equations, and it can be forecast by getting initial data points—the current weather—and solving these equations. However, if it takes longer to solve the equations than for weather to happen, these forecasts are not useful. It does not do much good to have the forecast for Tuesday on the following Wednesday. Problems like this one generated research in methods for solving differential equations, and considerable progress has been made.

But what problems are like this—solvable with good methods? Are there any problems that are difficult now and that we cannot expect ever to have good solutions?

Of course, if one needs a solution to a problem that has no solution—or no reasonable solution—then what? Here too, some knowledge is useful. It is not necessary just to give up and say, “Can’t be done.” Knowing the difficulties (and the options around them) is the first step.

- What is an algorithm?
- What is a computation?
- What is a computer?
- When does an algorithm exist?
- When do reasonable algorithms exist?
- What is meant by *reasonable*?

FIGURE I.1 Questions for this course.

In short, an educated computer scientist needs more than tools of programming. He or she needs to understand what is possible to program—which is the topic of this book.

THE "BIG" IDEAS

The two big questions that this book deals with are:

> Which problems have no algorithm at all (and what does that mean)?
>
> Which problems cannot be solved efficiently (and what does that mean)?

Answering these questions—and even having the machinery to properly pose them as questions—will take some time and effort.

This can be summarized in Figure I.1.

CHAPTER 1

Set Theory

Students will have seen set theory before and thus, the following is a brief review. Some important ideas, though, may be new and will be covered in more detail.

1.1 SETS—BASIC TERMS

Definition 1.1

A **set** is a collection of objects. ■

Definition 1.2

A **member** or **element** is an object in a set. A set is said to **contain** its elements. ■

Elements in a set are listed in braces.

Examples

$$S_1 = \{1, 2, 3, 4\}$$

$$S_2 = \{a, b, c\}$$

$$S_3 = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$$

Repetition does not matter in a set and ordering means nothing, so {a, b, c} = {b, a, c, b}.

Sets can be finite or infinite. Ellipses can be used in set notation once a pattern of membership has been established.

Examples

$$S_4 = \{1, 2, 3, \ldots, 98, 99, 100\}$$

$$S_5 = \{1, 2, 3, \ldots\}$$

$$S_6 = \{\ldots, -3, -2, -1\}$$

$$S_7 = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

Sets can also be described using Peano's notation.

$$S = \{x \mid x \text{ satisfies some condition}\}$$

Examples

$\{x \mid x = y^2 \text{ and } y \text{ is an integer}\}$ {squares}

$\{x \mid x = 2y \text{ and } y \text{ is an integer}\}$ {even numbers}

There must always be an underlying **universal set** *U*, either specifically stated or implicit. Some common universal sets include:

$\mathbf{N} = \{0, 1, 2, 3, \ldots\}$	(natural or counting numbers)
$\mathbf{Z} = \{\ldots, -2, --1, 0, 1, 2, \ldots\}$	(integers)
$\mathbf{Z}^+ = \{1, 2, 3, \ldots\}$	(positive integers)
$\mathbf{Z}^- = \{\ldots, -3, -2, -1\}$	(negative integers)
$\mathbf{Q} = \{x : x = m/n, m, n$ are integers, $n \neq 0\}$	(rational numbers)
$\mathbf{R}$ = real numbers	

Set membership is indicated by the $\in$ symbol, and **set exclusion** (is not a member) by $\notin$.

Examples

$$a \in \{a, b, c\}$$

$$d \notin \{a, b, c\}$$

Definition 1.3

The set A is a **subset** of set B, denoted $A \subseteq B$, if and only if (iff) every member of A is also a member of B. ■

Example

{a}, {b, c}, and {c, b, a} are some of the subsets of {a, b, c}.

Definition 1.4

The **empty set**, denoted $\varnothing$, is the set {}. It contains no elements. ■

Definition 1.5

The set A is a **proper subset** of set B iff every member of A is also a member of B and $A \neq B$, denoted $A \subset B$. ■

Example

{a}, {b, c} are some proper subsets of {a, b, c}.

The empty set is a subset of every set, and a proper subset of every set except itself.

Definition 1.6

The standard set operations are union, intersection, difference, and complement. They are defined as:

The **union** of two sets A and B, denoted $A \cup B$, is the set $\{x \mid x \in A \text{ or } x \in B\}$.

The **intersection** of two sets A and B, denoted $A \cap B$, is the set $\{x \mid x \in A \text{ and } x \in B\}$.

The **difference** of two sets A and B, denoted $A - B$, is the set $\{x \mid x \in A \text{ and } x \notin B\}$.

The **complement** of a set A, denoted $\bar{A}$ or A^c, is the set $\{x \mid x \notin A \text{ and } x \in \mathbf{U}\}$. ■

Definition 1.7

A **multiset** is a set in which the repetition of elements is important. Order is still irrelevant in a multiset. ■

Example

$\{4, 1, 2, 4, 1\} \neq \{4, 1, 2\}$	(for multisets)
$\{4, 1, 2, 4, 1\} = \{4, 1, 2\}$	(for sets)
$\{4, 1, 2, 4, 1\} = \{1, 1, 2, 4, 4\}$	(for multisets and sets)

Definition 1.8

A **well-ordered set** is a set in which there is a natural ordering of the elements such that for any two distinct elements e_1 and e_2 in the set, either $e_1 < e_2$ or $e_1 > e_2$. For example, the English language alphabet {a, b, c, …, x, y, z} is a well-ordered set. We rely on this fact when we alphabetize. ■

Definition 1.9

A **sequence** is a list of objects in an order. Elements in a sequence are listed in parentheses. ■

Example

(a, b, r, a, c, a, d, a, b, r, a)

(3, 1, 4, 1, 5, 9, 2)

Repetition and order both matter in a sequence, so $(1, 2, 3) \neq (1, 1, 2, 3) \neq (2, 1, 3)$.

Definition 1.10

An **empty sequence** is the sequence (). ■

As with sets, a sequence can be finite or infinite. The set of natural numbers can be viewed as a sequence $(0, 1, 2, 3, \ldots)$.

Finite sequences have particular names.

Definition 1.11

A **tuple** is a finite sequence.

An ***n*-tuple** is a sequence containing exactly n elements. The sequence (a, b, c) is therefore a 3-tuple, and the sequence (1, 2, 3, 4) is a 4-tuple.

An **ordered pair** is a 2-tuple. ■

Definition 1.12

The **power set** of A, denoted $P(A)$, is the set of all subsets of A. ■

Examples

$$P(\{a, b, c\}) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

$$P(\{1, 2\}) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$$

$$P(\emptyset) = \{\emptyset\}$$

Definition 1.13

The **Cartesian product** or **cross-product** of two sets A and B, denoted $A \times B$, is the set $\{(x, y): x \in A \text{ and } y \in B\}$. ■

Example

$$\{a, b\} \times \{c, d\} = \{(a, c), (a, d), (b, c), (b, d)\}$$

$$\{1, 2, 3\} \times \{1, 2\} = \{(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)\}$$

$$\{a, b, c\} \times \varnothing = \varnothing$$

1.2 FUNCTIONS

Again, functions are a concept quite familiar to computer science.

Definition 1.14

A **function** or **mapping** from set A to set B (written f: A → B) is a subset of A × B such that each x ∈ A is associated with a *unique* y ∈ B.

For f: A → B:
- A is called the **domain** of f.
- B is called the **codomain** of f.

If f(x) = y:
- y is called the **image** of x under f.
- x is the **preimage** of y under f. ■

Thus, the mapping from a person to his or her mother is a function (assuming exactly one mother per person), but the mapping from a person to his or her child is not. The mapping (person x, mother of X) has a domain of all people—since every person has a mother, and a codomain of the set of women who have children.

Definition 1.15

A function f from a set A to a set B is an **injection** if no two values from A are mapped to the same element of B (f(x) = f(y) implies that x = y). It is a **surjection** if it is onto B (for every b ∈ B, there is an x ∈ A such that f(x) = b). It is a **bijection** or **one-to-one correspondence** if it is both an injection and a surjection (one-to-one and onto). ■