Automata, Computability and Complexity

THEORY AND APPLICATIONS

Elaine Rich

PEARSON
Prentice Hall
Upper Saddle River NJ 07458

CONTENTS

PART I

INTRODUCTION

C H A P T E R 1

Why Study the Theory of Computation?

In this book, we present a theory of what can be computed and what cannot. We also sketch some theoretical frameworks that can inform the design of programs to solve a wide variety of problems. But why do we bother? We don't we just skip ahead and write the programs that we need? This chapter is a short attempt to answer that question.

1.1 The Shelf Life of Programming Tools

Implementations come and go. In the somewhat early days of computing, programming meant knowing how to write code like:[1]

```
ENTRY     SXA    4,RETURN
          LDQ    X
          FMP    A
          FAD    B
          XCA
          FMP    X
          FAD    C
          STO    RESULT
RETURN    TRA    0

A         BSS    1
B         BSS    1
C         BSS    1
X         BSS    1
TEMP      BSS    1
STORE     BSS    1
          END
```

[1]This program was written for the IBM 7090. It computes the value of a simple quadratic $ax^2 + bx + c$.

In 1957, Fortran appeared and made it possible for people to write programs that looked more straightforwardly like mathematics. By 1970, the IBM 360 series of computers was in widespread use for both business and scientific computing. To submit a job, one keyed onto punch cards a set of commands in OS/360 JCL (Job Control Language). Guruhood attached to people who actually knew what something like this meant:[2]

```
//MYJOB     JOB (COMPRESS),'VOLKER BANDKE', CLASS=P,COND=(0,NE)
//BACKUP   EXEC PGM=IEBCOPY
//SYSPRINT DD   SYSOUT=*
//SYSUT1    DD  DISP=SHR,DSN=MY.IMPORTNT.PDS
//SYSUT2    DD  DISP=(,CATLG),DSN=MY.IMPORTNT.PDS.BACKUP,
//              UNIT=3350,VOL=SER=DISK01,
//              DCB=MY.IMPORTNT.PDS,SPACE=(CYL,(10,10,20))
//COMPRESS EXEC PGM=IEBCOPY
//SYSPRINT DD   SYSOUT=*
//MYPDS     DD  DISP=OLD,DSN=*.BACKUP.SYSUT1
//SYSIN     DD  *
COPY INDD=MYPDS,OUTDD=MYPDS
//DELETE2 EXEC PGM=IEFBR14
//BACKPDS   DD  DISP=(OLD,DELETE,DELETE),DSN=MY.IMPORTNT.PDS.BACKUP
```

By the turn of the millennium, gurus were different. They listened to different music and had never touched a keypunch machine. But many of them did know that the following Java method (when compiled with the appropriate libraries) allows the user to select a file, which is read in and parsed using whitespace delimiters. From the parsed file, the program builds a frequency map, which shows how often each word occurs in the file:

```
public static TreeMap<String, Integer> create() throws IOException
    public static TreeMap<String, Integer> create() throws
                IOException
    {  Integer freq;
       String word;
       TreeMap<String, Integer> result = new TreeMap<String, Integer>();
       JFileChooser c = new JFileChooser();
       int retval = c.showOpenDialog(null);
       if (retval == JFileChooser.APPROVE_OPTION)
            {  Scanner s = new Scanner( c.getSelectedFile());
               while( s.hasNext() )
               {  word = s.next().toLowerCase();
                  freq = result.get(word);
                  result.put(word, (freq == null ? 1 : freq + 1));
               }
            }
        return result;
    }
}
```

[2]It safely reorganizes and compresses a partitioned dataset.

Along the way, other programming languages became popular, at least within some circles. There was a time when some people bragged that they could write code like:[3]

```
(Γ/V)>(+/V)-Γ/V
```

Today's programmers can't read code from 50 years ago. Programmers from the early days could never have imagined what a program of today would look like. In the face of that kind of change, what does it mean to learn the science of computing?

The answer is that there are mathematical properties, both of problems and of algorithms for solving problems, that depend on neither the details of today's technology nor the programming fashion *du jour*. The theory that we will present in this book addresses some of those properties. Most of what we will discuss was known by the early 1970s (barely the middle ages of computing history). But it is still useful in two key ways:

- It provides a set of abstract structures that are useful for solving certain classes of problems. These abstract structures can be implemented on whatever hardware/software platform is available.
- It defines provable limits to what can be computed, regardless of processor speed or memory size. An understanding of these limits helps us to focus our design effort in areas in which it can pay off, rather than on the computing equivalent of the search for a perpetual motion machine.

In this book our focus will be on analyzing problems, rather than on comparing solutions to problems. We will, of course, spend a lot of time solving problems. But our goal will be to discover fundamental properties of the problems themselves:

- Is there any computational solution to the problem? If not, is there a restricted but useful variation of the problem for which a solution does exist?
- If a solution exists, can it be implemented using some fixed amount of memory?
- If a solution exists, how efficient is it? More specifically, how do its time and space requirements grow as the size of the problem grows?
- Are there groups of problems that are equivalent in the sense that if there is an efficient solution to one member of the group there is an efficient solution to all the others?

[3]An expression in the programming language APL □. It returns 1 if the largest value in a three element vector is greater than the sum of the other two elements, and 0 otherwise [Gillman and Rose 1984, p. 326]. Although APL is not one of the major programming languages in use today, its inventor, Kenneth Iverson, received the 1979 Turing Award for its development.

1.2 Applications of the Theory Are Everywhere

Computers have revolutionized our world. They have changed the course of our daily lives, the way we do science, the way we entertain ourselves, the way that business is conducted, and the way we protect our security. The theory that we present in this book has applications in all of those areas. Throughout the main text, you will find notes that point to the more substantive application-focused discussions that appear in Appendices G–Q. Some of the applications that we'll consider are:

- Languages, the focus of this book, enable both machine/machine and person/machine communication. Without them, none of today's applications of computing could exist.

> Network communication protocols are languages. (I. 1) Most web pages are described using the Hypertext Markup Language, HTML. (Q.1.2) The Semantic Web, whose goal is to support intelligent agents working on the Web, exploits additional layers of languages, such as RDF and OWL, that can be used to describe the content of the Web. (I. 3) Music can be viewed as a language, and specialized languages enable composers to create new electronic music. (N.1) Even very unlanguage-like things, such as sets of pictures, can be viewed as languages by, for example, associating each picture with the program that drew it. (Q.1.3)

- Both the design and the implementation of modern programming languages rely heavily on the theory of context-free languages that we will present in Part III. Context-free grammars are used to document the languages' syntax and they form the basis for the parsing techniques that all compilers use.

> The use of context-free grammars to define programming languages and to build their compilers is described in Appendix G.

- People use natural languages, such as English, to communicate with each other. Since the advent of word processing, and then the Internet, we now type or speak our words to computers. So we would like to build programs to manage our words, check our grammar, search the World Wide Web, and translate from one language to another. Programs to do that also rely on the theory of context-free languages that we present in Part III.

> A sketch of some of the main techniques used in natural language processing can be found in Appendix L.

- Systems as diverse as parity checkers, vending machines, communication protocols, and building security devices can be straightforwardly described as finite state machines, which we'll describe in Chapter 5.

> A vending machine is described in Example 5.1. A family of network communication protocols is modeled as finite state machines in I.1. An example of a simple building security system, modeled as a finite state machine, can be found in J.1. An example of a finite state controller for a soccer-playing robot can be found in P.4.

- Many interactive video games are (large, often nondeterministic) finite state machines.

> An example of the use of a finite state machine to describe a role playing game can be found in N.3.1.

- DNA is the language of life. DNA molecules, as well as the proteins that they describe, are strings that are made up of symbols drawn from small alphabets (nucleotides and amino acids, respectively). So computational biologists exploit many of the same tools that computational linguists use. For example, they rely on techniques that are based on both finite state machines and context-free grammars.

> For a very brief introduction to computational biology see Appendix K.

- Security is perhaps the most important property of many computer systems. The undecidability results that we present in Part IV show that there cannot exist a general purpose method for automatically verifying arbitrary security properties of programs. The complexity results that we present in Part V serve as the basis for powerful encryption techniques.

> For a proof of the undecidability of the correctness of a very simple security model, see J.2. For a short introduction to cryptography, see J.3.

- Artificial intelligence programs solve problems in task domains ranging from medical diagnosis to factory scheduling. Various logical frameworks have been proposed for representing and reasoning with the knowledge that such programs exploit. The undecidability results that we present in Part IV show that there cannot exist a general theorem prover that can decide, given an arbitrary statement in first order logic, whether or not that statement follows from the system's axioms. The complexity results that we present in Part V show that, if we back off to the far less expressive system of Boolean (propositional) logic, while it becomes possible to decide the validity of a given statement, it is not possible to do so, in general, in a reasonable amount of time.

> For a discussion of the role of undecidability and complexity results in artificial intelligence, see Appendix M. The same issues plague the development of the Semantic Web. (I.3)

- Clearly documented and widely accepted standards play a pivotal role in modern computing systems. Getting a diverse group of users to agree on a single standard is never easy. But the undecidability and complexity results that we present in Parts IV and V mean that, for some important problems, there is no single right answer for all uses. Expressively weak standard languages may be tractable and decidable, but they may simply be inadequate for some tasks. For those tasks, expressively powerful languages. that give up some degree of tractability and possibly decidability, may be required. The provable lack of a one-size-fits-all language makes the standards process even more difficult and may require standards that allow alternatives.

> We'll see one example of this aspect of the standards process when we consider, in I.3, the design of a description language for the Semantic Web.

- Many natural structures, including ones as different as organic molecules and computer networks, can be modeled as graphs. The theory of complexity that we present in Part V tells us that, while there exist efficient algorithms for answering some important questions about graphs, other questions are "hard", in the sense that no efficient algorithm for them is known nor is one likely to be developed.

> We'll discuss the role of graph algorithms in network analysis in I.2.

- The complexity results that we present in Part V contain a lot of bad news. There are problems that matter yet for which no efficient algorithm is likely ever to be found. But practical solutions to some of these problems exist. They rely on a variety of approximation techniques that work pretty well most of the time.

> An almost optimal solution to an instance of the traveling salesman problem with 1,904,711 cities has been found, as we'll see in Section 27.1. Randomized algorithms can find prime numbers efficiently, as we'll see in Section 30.2.4. Heuristic search algorithms find paths in computer games (N.3.2) and move sequences for champion chess-playing programs. (N.2.5)