Preface

In the context of human history, computers are a fairly recent invention. But the idea of *computation*—of solving a complex problem by repeated, systematic execution of a series of simple and straightforward operations—is thousands of years old. Ancient Greek, Egyptian, and Chinese philosophers discovered many important facts about numbers and their relationships, and developed computational methods that are still used today. In post-Renaissance Europe, mathematicians and scientists used computational techniques to fill books of mathematical tables. Numeric integration, carried out painstakingly by hand, was used to calculate the future positions the Moon and planets, and the nautical almanacs produced by "human computers" were essential for navigation well into the twentieth century.

Computation is now an essential part of modern life. Every day we write mail, share photographs, play music, read the news, and pay bills using our personal computers. Engineers use computers to design cars and airplanes, meteorologists use computers to predict the weather, pharmaceutical companies use computers to design new drugs, and investment firms use computational models to predict whether complex transactions are likely to succeed. Modern astronomers also rely heavily on computation. Computers perform calculations that track the movement of planets, asteroids, and comets, keeping an eye out for bodies that pose a potential hazard to the Earth. Astrophysicists use computation to investigate theories on how black holes are formed and how planets coalesce from clouds of interstellar dust. Telescopes today gather massive amounts of data, requiring sophisticated new methods to sift through the information and catalog objects.

Computer science is the study of computation. Given the name "computer science" one might think the field could be characterized as "the study of computers," but as the discussion above showed, the idea of computing has been around a lot longer than there have been machines to do the computations. In the words of one influential computer scientist, Edsger Dijkstra (1930–2002), "computer science is no more about computers than astronomy is about telescopes." Computer hardware plays a huge role, of course, since much of the motivation for studying computer science comes from the fact that computations are run on machines that perform a wide variety of essential tasks. For many people, a large part of the satisfaction of working in computer science derives from the fact that abstract ideas can be turned into programs that run on real computer systems and address important real-world problems.

Another misconception is that computer science is the same as "programming" and computer science courses are all about teaching students to write programs. Computer science is much more than simply writing software. Computer science is a rich intellectual field where practitioners apply a computational approach to address a wide variety of interesting and challenging problems. Computer scientists are engaged in research in core areas of theoretical computer science, computer systems design, algorithms, and programming languages, as well as more application-oriented areas such as databases, networking, and informatics.

This book is a textbook, intended for courses that are an introduction to computer science. The emphasis is on how computation helps people solve problems. Computer science is a huge field, and entire books have been written about algorithms, theory, programming languages, databases, networks, and other areas. Rather than trying to survey the entire field and give a brief introduction to each important area, the goals in this book are to focus on the fundamental idea of computation itself and to give readers some insight into how computation can be used to solve a variety of interesting and important real-world problems.

Active Learning

The distinguishing feature of this book is its active learning approach. Each chapter includes a tutorial project that guides students as they use an interactive environment to explore important ideas in computing by running programs, modifying them, and trying them out on different inputs.

One of the inspirations for this approach was the active learning embodied in a role-playing game called *The Oregon Trail*. Students who played this game learned about the great westward migration of the nineteenth century by making decisions for their character as they traveled from Missouri to Oregon in 1848, trying to manage their resources and avoid hazards along the way. By actively engaging with the material in a virtual environment, and making decisions that would affect the outcome of the game, students gained a much deeper appreciation of what life was like for the people who set out on that journey. One of the factors often cited for the success of this game is that students were able to try many different variations. Students could play the game several times, assuming several different roles, and often seeing a different outcome, even when they made the same decisions.

The title of this book, *Explorations in Computing*, conveys the idea of how we will use a similar active learning approach to study computation. Each chapter is organized around a single project that introduces an important concept or application in computer science. To complete the project, students type commands in Ruby, an interactive programming language, following a detailed script set out in the text. The aim is for the students to immerse themselves in the interactive environment and experience first-hand what goes on inside a computer as it solves some interesting problems. Many parts of the projects are open-ended, and students are encouraged to continue exploring on their own, after using the exercises in the book as a starting point.

An example of this approach is the project in a chapter on the N-body problem, where students set up an experiment that simulates the motion of the planets in the solar system. After running the basic exercises, which lead up to a simulation that shows the planets moving in elliptical orbits around the Sun, students can explore on their own, to see what happens in a chaotic system when the initial conditions change just slightly, or when the mass of one of the planets is increased to the point where the system has two bodies the size of the Sun. Another example is a project on the traveling salesman problem, where students run experiments based on a genetic algorithm. After running a set of preliminary experiments to learn how the algorithm evolves an efficient tour through random mutations, students can run more simulations, varying the simulation parameters to see what effect each has on the outcome.

Intended Audience

This book was written for students who want to learn what computer science is about. It was written with two different audiences in mind: students who intend to continue on to major in computer science, and would like a general overview of the field, and students who have chosen to major in a different field, but who would like to take a computer science course as a science elective.

Although the projects in this book are set up to run in an interactive programming environment, no prior experience with programming is necessary. To complete a project, students follow detailed instructions from the text, much like working on a tutorial to learn how to use a new software application. By working through the projects in each chapter, readers will build up a working knowledge of the concepts and terminology of Ruby programming, but the projects do not require students to write any of their own programs.

Although students do not need programming experience, they should be proficient computer users. Students who want to do the projects on their own computers will need to install some software (explained in more detail below), so they should be comfortable with the process of connecting to the Internet and downloading and installing applications. Several projects also require students to create folders and navigate through a file system, and to open, edit, and save text files.

The projects are based on a variety of different subjects, but none of the exercises assume any detailed knowledge of the subject area. The introduction to each section should provide the necessary background to work on the project in that chapter.

As might be expected for a science class, many projects do require a basic proficiency in math. Students should be comfortable with logarithms, exponents, square roots, and other basic mathematical functions. Projects do not require students to solve equations or to work through proofs, as they would in a math course, but students should understand what the functions are and how they are used. Students will gain a deeper appreciation for scalability and other important concepts in computer science when they have the necessary math background.

♦ Advanced Topics

Some chapters include more challenging ideas or exercises. In some cases there will be an entire section devoted to an advanced topic, and in other cases there may be recommended projects for students who want to learn more about a topic or do some more exploration on their own. These sections and projects are indicated by a ♦ symbol, to indicate material that requires more advanced skills, like the trails marked with a "black diamond" at a downhill ski area.

Notes for Students

You have no doubt heard the adage, "What you get out of a course depends on what you put into it." That saying is especially true for learning about computation with this book. Each chapter is built around a project that helps you explore a particular problem and ways of solving it computationally. If you work through the project and spend some time

thinking about what your computer is doing as it runs a computation, you will be rewarded by gaining a deeper insight into how computers can help solve a wide variety of important problems.

A useful analogy for these projects are the lab projects that go along with an introductory chemistry course. An instructor selects a set of concepts the students should learn and then develops a set of lab projects to help students gain some experience and reinforce their learning of the concepts. The materials and methods are all spelled out in great detail, and students follow a set of well-specified steps. Those who continue on in chemistry will later learn to design their own experiments, but for beginners everything is set up by the instructor.

That same approach is taken here in this book. The "computational experiments" in each chapter are tutorials that contain detailed instructions for how to start a piece of software and then what to type in order to run the experiment. As you interact with the software you will see how the computation unfolds. The tutorials are designed so that you should be able to complete them in about the same amount of time you would spend on a lab project in a chemistry class. You could run through the tutorials in less time—about as fast as you can type, or if you get examples from web pages, as fast as you can cut and paste—but you should take the time to make sure you understand what your computer is doing as you carry out each step in the tutorial.

At the end of each chapter you will find a set of exercises. These are similar to the questions you would find in a more traditional textbook and are designed to test your understanding of the material in that chapter. If you have completed the tutorial and understood what happened at each step along the way you should be able to answer these questions.

Notes for Instructors

This book is an introduction to computer science for premajors and nonmajors, a course commonly called CS0 in the computer science literature. As part of a program for premajors, the book would be a suitable text for a first course in an introductory computer science sequence, or as part of a "great ideas" course. The book would also be a good text for a stand-alone science elective, or for a course on computational thinking. When augmented with programming assignments, it could also be used in a programming-first or objects-first CS1 course.

As mentioned above, the book is organized around a set of projects that give students an opportunity to experiment with important ideas in computer science. In most cases, the important concepts are algorithms, and the projects are examples of how algorithms provide computational solutions to important problems. An interactive programming language provides a "computational workbench" where students can experiment with algorithms by typing expressions and seeing the results. The interactive language sets up an environment where students can run computations and explore the effects of changing parameters or modifying operations performed at key steps of the computation.

But using an interactive programming language raises a difficult issue: won't students have to learn to write programs? The approach taken in this book is to base the experiments on a set of scripted tutorials. Each project in the book has a detailed set of instructions for how to perform an experiment by loading software that has been written already. The

expressions students enter create and manipulate objects and call methods that implement the algorithm being studied.

The viability of the scripted tutorial approach is based on the fact that is is much easier to learn to read existing programs than to write new ones. Anyone who has tried to learn a foreign language knows how much easier it is to read phrases in the new language than it is to speak or write a sentence. A similar effect applies to programming languages as well. Beginning students can reach a surprising level of literacy by just learning a few fundamental concepts of object-oriented programming—objects, classes, methods, variables, and control flow—with the view that they are learning a language that is a notation for describing algorithms. Since students are only expected to understand programs, they do not need to learn how to design, implement, test, and debug their own code, and several messy details covered in introductory programming courses, like scope rules, call by reference, variable lifetimes, *etc.*, can safely be ignored.

I chose to use Ruby for these projects for several reasons, the foremost being the interactive programming environment that supports experimentation. Ruby has a very clean syntax, and for most operations it provides an intuitive notation. Ruby is open source and is easily downloaded and installed on a wide variety of systems.

An important question was whether to try to make the book a comprehensive introduction to the entire field of computer science, or whether to focus on fewer topics and go into them in more depth. I chose the latter. I think the projects will be much more interesting, and students will gain a better overall understanding of what computer science is about and how computer scientists think about problems, if the book has a few well-chosen examples, even if it means leaving out several important topics.

The topics presented in the book are outlined below. The general pattern for each chapter will be to first introduce the concept presented in that chapter; this introductory section will essentially be an essay that tries to make the case that the idea is interesting and worth understanding in more detail. The main part of the chapter will be the development, through a series of projects, of one or more algorithms that illustrate the idea and provide the student with a chance to experiment.

1 Introduction

The book starts with a general introduction to computation, expanding on the themes mentioned in the first section of this preface: computer science is not just about computers and is not just programming.

2 The Ruby Workbench

The second chapter is a practical introduction to Ruby and how it can be used as a "computational workbench" to set up experiments with computations. The tutorial takes the students through the construction of a simple program to convert temperature from Celsius to Fahrenheit, and introduces the ideas of variables, objects, and methods.

3 The Sieve of Eratosthenes

This chapter introduces the first real algorithm studied in the book. It also introduces a few more practical techniques used later in the book: making lists of numbers and iterating over a list. The tutorial starts with simple expressions involving integers, shows how to make a list of numbers, then how to selectively remove composite numbers, and leads finally to an algorithm that creates a complete list of prime numbers.

4 A Journey of a Thousand Miles

This chapter builds on the basic idea of iteration presented in the previous chapter. The project shows how iteration can be used to solve two common problems, searching and sorting, using linear search and insertion sort. An important idea in computing in this chapter is scalability, and students are introduced to $\mathcal{O}$ notation.

5 Divide and Conquer

The important idea in this chapter is that a more sophisticated strategy for solving a problem can lead to a more efficient computation. The tutorial shows how binary search takes up to $\log_2 n$ steps instead of n, and merge sort takes at most $n \log_2 n$ steps instead of n^2.

6 When Words Collide

The new concept in this chapter is that our ability to solve a problem computationally depends not only on the sequence of steps defined by an algorithm, but also on how the data is organized. The tutorial project is based on a data structure that implements an index for a large collection of data. Students learn about hash functions and eventually do experiments with a hash table that resolves collisions with buckets.

7 Bit by Bit

The projects in this chapter are related to encoding data: using patterns of binary digits to encode numbers and letters, the number of bits required to encode a set of items, text compression with Huffman trees, and error correction with parity bits.

8 The War of the Words

This chapter introduces the important ideas that functions can also be encoded as a string of bits and that instructions (bit patterns representing steps that implement functions) are stored in a computer's memory along with data. The tutorial uses the game of Corewar, which is a contest between two programs running in the same virtual machine; a program wins if it can write a halt instruction over the opponent's code. The tutorial leads the student through the phases of a processor's fetch-decode-execute cycle and emphasizes how a word that is a piece of data (the constant 0) for one program becomes an instruction (halt) for the other program.

9 Now for Something Completely Different

The big idea in this chapter is randomness, and how random numbers can be used in a variety of algorithms, from games to scientific applications. There is an interesting paradox here: can we really generate random outputs from an algorithm? Isn't a method in Ruby supposed to carry out exactly the same calculations and produce the same result each time it is called? The answer is that random numbers generated by an algorithm are pseudorandom, and the project takes students through the steps in the development and testing of a pseudorandom number generator.

10 Ask Dr. Ruby

The tutorial project in this chapter is based on a Ruby implementation of Joseph Weizenbaum's ELIZA program, and shows how very simple pattern matching rules can be used to transform input sentences, giving the illusion that the computer is carrying on a conversation. By the end of the chapter students will see how difficult natural language processing is, and how semantics and real-world knowledge are required for effective natural language understanding.

11 The Music of the Spheres

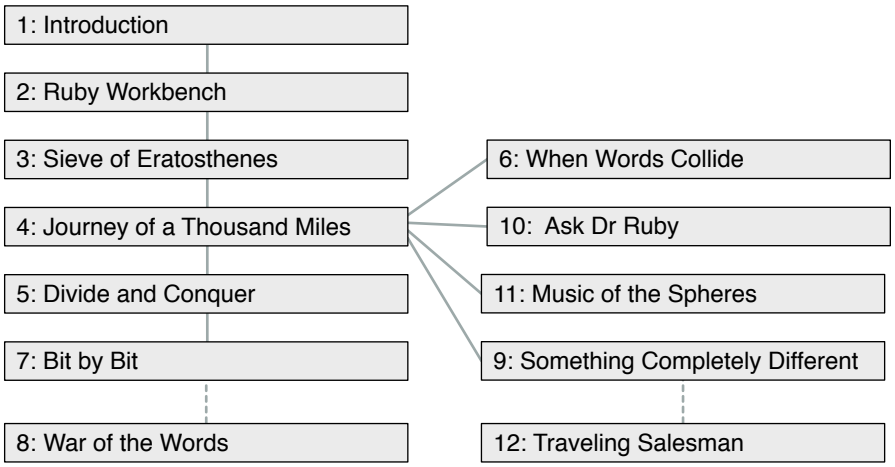
The big idea in this chapter is computer simulation. The project leads to an *ab initio* simulation of the motion of planets in the solar system. The chapter introduces issues related to verification and other topics in computer simulation.

12 The Traveling Salesman

The last chapter introduces the idea of intractable problems, building on ideas of scalability from earlier chapters. The project is based on a genetic algorithm, and gives students the opportunity to explore probabilistic solutions. The tutorial has students use predefined code for Map and Tour classes to create random tours, so they can see how tours can be mutated and how collections of tours evolve until an optimal or near-optimal solution is obtained.

Pedagogical Considerations

The chapters and projects described above have been used in a course at the University of Oregon (*CIS 170: The Science of Computing*). We cover the first two chapters during the first week, but after that we spend between one and two weeks on the remaining topics chosen for that term. Lectures emphasize material from the first sections of a chapter, describing the problem and how it might be solved computationally, and explaining how that week's lab project gives some experience with the computation. Students have an option of attending a lab session, where an instructor is available to help them work through the material, but many students do the tutorials on their own. Live demonstrations of the tutorial projects, both in lecture and in lab sessions, have proved to be very effective.



At the end of each chapter there is a set of exercises that ask questions about issues raised in the chapter. After the students have completed the tutorial, they are asked to answer a selected set of questions and submit them as a "lab report" that gives them a chance to explain what they learned. Similar questions are given on exams.

When selecting topics to use in a course, Chapters 1 through 4 should be used every term, since they introduce key concepts (algorithms, scalability) and practical lab skills (instantiating objects, calling methods, creating and iterating over containers) used in other projects. The remaining chapters are mostly independent, and can be selected according to the interests of the students. The chapters on data representations (Bit by Bit) and machine language programming (War of the Words) are both based on the idea of encoding information, but students will have no trouble completing the Corewar project without having done the data representation projects. Similarly, The Traveling Salesman uses random numbers, but students will get a lot out of this project even if they haven't seen how random numbers are generated.

It is also possible to organize a course that includes additional topics and activities beyond those described in this book. In Spring 2008, a few months before the national elections in the U.S., we used electronic voting as a theme for CIS 170. There were additional units on the history of elections and the need for privacy and security in voting, and the computer science topics included networking, encryption, and software engineering, all of which play a role in the design of electronic voting machines.

Software, Documentation, and Lab Manuals

All of the software used for the tutorial projects is written in Ruby. Students can do the projects on computers in an instructional lab, or they can install Ruby on their own computers. Ruby is open-source, and it is a straightforward process to install the Ruby interpreter and associated applications:

- Users of Microsoft Windows XP can download a "one-click installer" that automates all the installation steps.

- A single command typed in a terminal window will install Ruby on a Linux system.
- Ruby is already installed on Mac OS X 10.4 and later (although users running Mac OS X 10.6 may have to reinstall Ruby to work on labs that have interactive visualizations).

The software students will use for the projects is named RubyLabs. RubyLabs is written exclusively in Ruby, using only libraries and modules that are part of the standard Ruby distribution. There is one Ruby module for each lab project. All of the modules have been collected into a single "Ruby gem," which makes it easy to install all the lab software in one step at the beginning of the term. The RubyLabs gem also includes data files and sample Ruby code that students can copy and modify.

A Lab Manual with step-by-step instructions for installing Ruby and the RubyLabs gem is available from the book web site at `http://www.cs.uoregon.edu/eic`. There is a separate version of the manual for Windows XP, Mac OS X, and Linux. The manual also includes tips for editing programs and running commands in a terminal emulator.

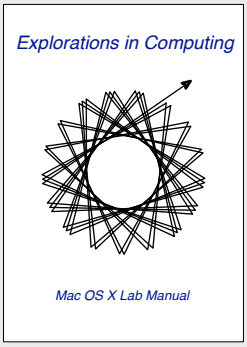
The web site also has on-line documentation of all the modules in the RubyLabs gem. After the gem has been installed, this documentation can be read locally by a web browser, without having to connect to the Internet.

Web Site

The web site for this book is

`http://www.cs.uoregon.edu/eic`

The web site will have
- copies of the lab manual (PDF documents that can be downloaded for free)
- links to the latest versions of the RubyLabs software and documentation
- errata and other news



Acknowledgments

The most important group of people who influenced the development of this book are, without a doubt, the students at the University of Oregon who have been involved in one way or another with CIS 170, the *Science of Computing* course I have taught each year since the 2005-06 academic year. Students who took the course and provided invaluable suggestions (often without waiting for me to ask) include: Joyce Corrao-Clanon, Alex Forbes, Emily Hayes, Isla Globus-Harris, Peri Moritz, Paul Russ, Charles Sheinin, Richard Suhr, and Ace Taylor. Jeff Blakeslee developed an early version of the binary tree visualizations used in Chapter 7, and Michael Maag made a key contribution to the RubyLabs canvas that is used in all the visualizations.

Anyone who has taught a college or university level course knows how important it is to have motivated and engaged teaching assistants, and I have been lucky to work with the best: Megen Brittell, Tom Bulatewicz, Victor Hanson-Smith, and Shad Stafford. Shad also used an early draft of Chapters 2 and 3 in a course he taught at Pacific University in Forest Grove, Oregon, and I received many helpful comments from Shad and his students.

I was thrilled when Phil Foglio responded to my e-mail and said he would be willing to make a set of illustrations for the book. Many thanks to Phil, Kaja Foglio (self-proclaimed "scanning bot"), and the other folks at Studio Foglio, LLC for making it happen.

John Impagliazzo and Andrew McGettrick, the series editors for Chapman & Hall/CRC Press, provided several constructive suggestions during the early phase of the development of the book. From the rough draft I initially submitted, they were able to help me focus on a better choice of topics and more readable presentation. I am also indebted to the external reviewers, especially Jessen Havill, of Denison University, and Andrew Neel, from the University of Memphis.

While the students, reviewers, and editors all had a tremendous influence on the contents of the book, it would have remained just another interesting idea instead of a real book if not for the efforts of Randi Cohen, my editor at CRC Press. Randi somehow knew exactly when I needed encouragement and positive feedback, to keep me going when it seemed like there was no end in sight, and when to set a firm deadline, when it looked like I was going to keep exploring forever.

Finally, I am grateful for the support of my wife Leslie and my daughter Kathleen. Kathleen looked over my lecture notes, read early versions of some of the chapters, and, thankfully, let me know how some of my attempts at humor would have been received by others of her generation. I love you both.

John Conery
Eugene, Oregon