GLOBAL EDITION

Computer Science

An Overview

THIRTEENTH EDITION

J. Glenn Brookshear
Dennis Brylow

Computer Science

AN OVERVIEW

13th Edition

Global Edition

J. Glenn Brookshear
(Author Emeritus)
and
Dennis Brylow
Marquette University

Pearson
330 Hudson Street, NY NY 10013

Senior Vice President, Courseware Portfolio Management: Engineering, Computer Science, Mathematics, Statistics, and Global Editions: Marcia Horton
Director, Portfolio Management: Engineering, Computer Science, and Global Editions: Julian Partridge
Executive Portfolio Manager: Tracy Johnson
Portfolio Management Assistant: Meghan Jacoby
Managing Producer, ECS and Mathematics: Scott Disanno
Senior Content Producer: Erin Ault
Acquisitions Editor, Global Edition: K.K. Neelakantan
Assistant Project Editor, Global Edition: Aurko Mitra
Media Production Manager, Global Edition: Vikram Kumar
Senior Manufacturing Controller, Global Edition: Angela Hawksbee
Manager, Rights and Permissions Manager: Ben Ferrini
Operations Specialist: Maura Zaldivar-Garcia
Inventory Manager: Bruce Bounty
Product Marketing Manager: Yvonne Vannatta
Field Marketing Manager: Demetrius Hall
Marketing Assistant: Jon Bryant
Full Service Project Management: Sasibalan Chidambaram, SPi Global
Cover Design: Lumina Datamatics, Inc.
Cover Photo: Amirul Syaidi/Shutterstock

Pearson Education Limited
KAO Two
KAO Park
Harlow
CM17 9NA
United Kingdom

and Associated Companies throughout the world

Visit us on theWorld Wide Web at: www.pearsonglobaleditions.com

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN 10: 1-292-26342-3
ISBN 13: 978-1-292-26342-7
eBook ISBN: 978-1-292-26344-1

eBook formatted by SPi Global

For Dexter,
who I know will eagerly read
this book cover to cover
before he turns eight.
Oh, the places you will go . . .

Contents

* Asterisks indicate suggestions for optional sections.

Preface

This book presents an introductory survey of computer science. It explores the breadth of the subject while including enough depth to convey an honest appreciation of the topics involved.

Audience

We wrote this text for students of computer science as well as students from other disciplines. As for computer science students, most begin their studies with the illusion that computer science is programming, web browsing, and Internet file sharing because that is essentially all they have seen. Yet computer science is much more than this. Beginning computer science students need exposure to the breadth of the subject in which they are planning to major. Providing this exposure is the theme of this book. It gives students an overview of computer science—a foundation from which they can appreciate the relevance and interrelationships of future courses in the field. This survey approach is, in fact, the model used for introductory courses in the natural sciences.

This broad background is also what students from other disciplines need if they are to relate to the technical society in which they live. A computer science course for this audience should provide a practical, realistic understanding of the entire field rather than merely an introduction to using the Internet or training in the use of some popular software packages. There is, of course, a proper place for that training, but this text is about educating.

While writing previous editions of this text, maintaining accessibility for nontechnical students was a major goal. The result was that the book has been used successfully in courses for students over a wide range of disciplines and educational levels, ranging from high school to graduate courses. This 13th edition is designed to continue that tradition.

New in the 13th Edition

Now in color! The move to a full color printing process in the 13th edition has allowed us to make many figures and diagrams more descriptive, and to use syntax coloring to better effect for clarifying code and pseudocode segments in the text. Most modern programming interfaces use color to aid the programmer's understanding of code; your computer science textbook should do no less.

A major theme during the development of this 13th edition has been highlighting the intersections with the new College Board Advanced Placement® Computer Science Principles ("CSP") exam. This "breadth-first" textbook for introducing computer science has included many of the big ideas and computational practices codified in the CSP framework since long before that exam came into existence; prior editions of the book have been used in pilot versions of CSP courses, and as a professional development resource for educators preparing to teach the high school version of the course. While the primary audience for this book remains college-level introductory courses, this edition explicitly calls out many points of intersection with CSP content to better assist students and instructors either preparing for the AP® CSP exam, or taking a college-level course that is intended to correspond with the credit from that exam.

The 13th edition continues the use of Python code examples and Python-like pseudocode adopted in the 12th edition. We made this change for several reasons. First, the text already contains quite a bit of code in various languages, including detailed pseudocode in several chapters. To the extent that readers are already absorbing a fair amount of syntax, it is appropriate to target that syntax toward a language they may actually see in a subsequent course. More importantly, a growing number of instructors who use this text have made the determination that even in a breadth-first introduction to computing, it is difficult for students to master many of the topics in the absence of programming tools for exploration and experimentation.

But why Python? Choosing a language is always a contentious matter, with any choice bound to upset at least as many as it pleases. Python is an excellent middle ground, with:

- a clean, easily learned syntax,
- simple I/O primitives,
- data types and control structures that correspond closely to the pseudocode primitives used in earlier editions, and
- support for multiple programming paradigms.

It is a mature language with a vibrant development community and copious online resources for further study. Python remains one of the top five most commonly used languages in the industry by some measures, and has seen a sharp increase in its usage for introductory computer science courses. It is particularly popular for introductory courses for non-majors, and has wide acceptance in other STEM fields, such as physics and biology, and as the language of choice for computational science applications.

Nevertheless, the focus of the text remains on broad computer science concepts; the Python supplements are intended to give readers a deeper taste of programming than previous editions, but not to serve as a full-fledged introduction to programming. The Python topics covered are driven by the existing structure of the text. Thus, Chapter 1 touches on Python syntax for representing data—integers, floats, ASCII, and Unicode strings. Chapter 2 touches on Python operations that closely mirror the machine primitives discussed throughout the rest of the chapter. Conditionals, loops, and functions are introduced in Chapter 5, at the time that those constructs are needed to devise a sufficiently complete pseudocode for describing algorithms. In short, Python constructs are used to reinforce computer science concepts rather than to hijack the conversation.

Every chapter has seen revisions, updates, and corrections from the previous editions.

Organization

This text follows a bottom-up arrangement of subjects that progresses from the concrete to the abstract—an order that results in a sound pedagogical presentation in which each topic leads to the next. It begins with the fundamentals of information encoding, data storage, and computer architecture (Chapters 1 and 2); progresses to the study of operating systems (Chapter 3) and computer networks (Chapter 4); investigates the topics of algorithms, programming languages, and software development (Chapters 5 through 7); explores techniques for enhancing the accessibility of information (Chapters 8 and 9); considers some major applications of computer technology via graphics (Chapter 10) and artificial intelligence (Chapter 11); and closes with an introduction to the abstract theory of computation (Chapter 12).

Although the text follows this natural progression, the individual chapters and sections are surprisingly independent and can usually be read as isolated units or rearranged to form alternative sequences of study. Indeed, the book is often used as a text for courses that cover the material in a variety of orders. One of these alternatives begins with material from Chapters 5 and 6 (Algorithms and Programming Languages) and returns to the earlier chapters as desired. I also know of one course that starts with the material on computability from Chapter 12. In still other cases, the text has been used in "senior capstone" courses where it serves as merely a backbone from which to branch into projects in different areas. Courses for less technically oriented audiences may want to concentrate on Chapters 4 (Networking and the Internet), 9 (Database Systems), 10 (Computer Graphics), and 11 (Artificial Intelligence).

On the opening page of each chapter, we have used asterisks to mark some sections as optional. These are sections that cover topics of more specific interest, or perhaps explore traditional topics in more depth. Our intention is merely to provide suggestions for alternative paths through the text. There are, of course, other shortcuts. In particular, if you are looking for a quick read, we suggest the following sequence:

Section	**Topic**
1.1–1.4	Basics of data encoding and storage
2.1–2.3	Machine architecture and machine language
3.1–3.3	Operating systems
4.1–4.3	Networking and the Internet
5.1–5.4	Algorithms and algorithm design
6.1–6.4	Programming languages
7.1–7.2	Software engineering
8.1–8.3	Data abstractions
9.1–9.2	Database systems
10.1–10.2	Computer graphics
11.1–11.3	Artificial intelligence
12.1–12.2	Theory of computation

There are several themes woven throughout the text. One is that computer science is dynamic. The text repeatedly presents topics in a historical perspective, discusses the current state of affairs, and indicates directions of research. Another theme is the role of abstraction and the way in which abstract tools are used to control complexity. This theme is introduced in Chapter 0 and then echoed in the context of operating system architecture, networking, algorithm development, programming language design, software engineering, data organization, and computer graphics.

To Instructors

There is more material in this text than students can normally cover in a single semester, so do not hesitate to skip topics that do not fit your course objectives or to rearrange the order as you see fit. You will find that, although the text follows a plot, the topics are covered in a largely independent manner that allows you to pick and choose as you desire. The book is designed to be used as a course resource—not as a course definition. We suggest encouraging students to read the material not explicitly included in your course. We underrate students if we assume that we have to explain everything in class. We should be helping them learn to learn on their own.

We feel obliged to say a few words about the bottom-up, concrete-to-abstract organization of the text. As academics, we too often assume that students will appreciate our perspective of a subject—often one that we have developed over many years of working in a field. As teachers, we think we do better by presenting material from the student's perspective. This is why the text starts with data representation/storage, machine architecture, operating systems, and networking. These are topics to which students readily relate—they have most likely heard terms such as JPEG and MP3; they have probably recorded data on DVDs and flash drives; they have interacted with an operating system; and they use the Internet and smartphones daily. By starting the course with these topics, students discover answers to many of the "why" questions they have been carrying for years, and learn to view the course as practical rather than theoretical. From this beginning, it is natural to move on to the more abstract issues of algorithms, algorithmic structures, programming languages, software development methodologies, computability, and complexity, that those of us in the field view as the main topics in the science. As already stated, the topics are presented in a manner that does not force you to follow this bottom-up sequence, but we encourage you to give it a try.

We are all aware that students learn a lot more than we teach them directly, and the lessons they learn implicitly are often better absorbed than those that are studied explicitly. This is significant when it comes to "teaching" problem solving. Students do not become problem solvers by studying problem-solving methodologies. They become problem solvers by solving problems—and not just carefully posed "textbook problems." So this text contains numerous problems, a few of which are intentionally vague—meaning that there is not necessarily a single correct approach or a single correct answer. We encourage you to use these and to expand on them.

Other topics in the "implicit learning" category are those of professionalism, ethics, and social responsibility. We do not believe that this material should be presented as an isolated subject that is merely tacked on to the course. Instead, it should be an integral part of the coverage that surfaces when it is relevant. This is the approach followed in this text. You will find that Sections 3.5, 4.6, 7.9, 9.7, and 11.7 present such topics as security, privacy, liability, and social awareness in the context of operating systems, networking, software engineering, database systems, and artificial intelligence. You will also find that each chapter includes a collection of questions called Social Issues that challenge students to think about the relationship between the material in the text and the society in which they live.

Thank you for considering our text for your course. Whether you do or do not decide that it is right for your situation, I hope that you find it to be a contribution to the computer science education literature.

Pedagogical Features

This text is the product of many years of teaching. As a result, it is rich in pedagogical aids. Paramount is the abundance of problems to enhance the student's participation—over 1,000 in this 13th edition. These are classified as Questions and Exercises, Chapter Review Problems, and Social Issues. The Questions and Exercises appear at the end of each section (except for the introductory chapter). They review the material just discussed, extend the previous discussion, or hint at related topics to be covered later. These questions are answered in Appendix F.

The Chapter Review Problems appear at the end of each chapter (except for the introductory chapter). They are designed to serve as "homework" problems in that they cover the material from the entire chapter and are not answered in the text.

Also, at the end of each chapter are the questions in the Social Issues category. They are designed for thought and discussion. Many of them can be used to launch research assignments culminating in short written or oral reports.

Each chapter also ends with a list called Additional Reading that contains references to other material relating to the subject of the chapter. The websites identified in this preface, in the text, and in the sidebars of the text are also good places to look for related material.

Supplemental Resources

A variety of supplemental materials for this text are available at the book's companion website: www.pearsonglobaleditions.com. The following are accessible to all readers:

- Chapter-by-chapter activities that extend topics in the text and provide opportunities to explore related topics.
- Chapter-by-chapter "self-tests" that help readers to rethink the material covered in the text.
- Activities that teach the basics of Python in a pedagogical sequence compatible with the text.

In addition, the following supplements are available to qualified instructors at Pearson's Instructor Resource Center. Please visit `www.pearsonglobaleditions.com` or contact your Pearson sales representative for information on how to access them.

- Instructor's Guide with answers to the Chapter Review Problems
- PowerPoint lecture slides
- Test bank

To Students

Glenn Brookshear is a bit of a nonconformist (some of his friends would say more than a bit), so when he set out to write this text he didn't always follow the advice he received. In particular, many argued that certain material was too advanced for beginning students. But, we believe that if a topic is relevant, then it is relevant even if the academic community considers it to be an "advanced topic." You deserve a text that presents a complete picture of computer science—not a watered-down version containing artificially simplified presentations of only those topics that have been deemed appropriate for introductory students. Thus, we have not avoided topics. Instead, we've sought better explanations. We've tried to provide enough depth to give you an honest picture of what computer science is all about. As in the case of spices in a recipe, you may choose to skip some of the topics in the following pages, but they are there for you to taste if you wish—and we encourage you to do so.

We should also point out that in any course dealing with technology, the details you learn today may not be the details you will need to know tomorrow. The field is dynamic—that's part of the excitement. This book will give you a current picture of the subject as well as a historical perspective. With this background, you will be prepared to grow along with technology. We encourage you to start the growing process now by exploring beyond this text. Learn to learn.

Thank you for the trust you have placed in us by choosing to read our book. As authors we have an obligation to produce a manuscript that is worth your time. We hope you find that we have lived up to this obligation.

Acknowledgments

First and foremost, I thank Glenn Brookshear, who has shepherded this book, "his baby," through 11 previous editions, spanning more than a quarter century of rapid growth and tumultuous change in the field of computer science. While this is the second edition in which he has allowed a co-author to oversee all of the revisions, the pages of this 13th edition remain largely in Glenn's voice and, I hope, guided by his vision. Any new blemishes are mine; the elegant underlying framework are all his.

I join Glenn in thanking those of you who have supported this book by reading and using it in previous editions. We are honored. *Thirteen* editions for a computer science textbook? We must be nearing some kind of record.

Andrew Kuemmel (Madison West) has been an invaluable sounding board as we worked to identify the overlaps between the 13th edition and the CS Principles framework. He is the only person I know who has successfully taught many instances of the CSP course at both the high school and university levels, and his tireless advocacy for computer science educators in my home state of Wisconsin has been truly inspirational.

David T. Smith (Indiana University of Pennsylvania) played a significant role in co-authoring revisions to the 11th edition with me, many of which are still visible in this 13th edition. David's close reading of previous editions and careful attention to the supplemental materials have been essential. Andrew Kuemmel (Madison West), George Corliss (Marquette), and Chris Mayfield (James Madison) all provided valuable feedback, insight, and/or encouragement on drafts for this or previous editions, while James E. Ames (Virginia Commonwealth), Stephanie E. August (Loyola), Yoonsuck Choe (Texas A&M), Melanie Feinberg (UT-Austin), Eric D. Hanley (Drake), Sudharsan R. Iyengar (Winona State), Ravi Mukkamala (Old Dominion), and Edward Pryor (Wake Forest) all offered valuable reviews of the Python-specific revisions for the 12th edition.

Others who have contributed in this or previous editions include J. M. Adams, C. M. Allen, D. C. S. Allison, E. Angel, R. Ashmore, B. Auernheimer, P. Bankston, M. Barnard, P. Bender, K. Bowyer, P. W. Brashear, C. M. Brown, H. M. Brown, B. Calloni, J. Carpinelli, M. Clancy, R. T. Close, D. H. Cooley, L. D. Cornell, M. J. Crowley, F. Deek, M. Dickerson, M. J. Duncan, S. Ezekiel, C. Fox, S. Fox, N. E. Gibbs, J. D. Harris, D. Hascom, L. Heath, P. B. Henderson, L. Hunt, M. Hutchenreuther, L. A. Jehn, K. K. Kolberg, K. Korb, G. Krenz, J. Kurose, J. Liu, T. J. Long, C. May, J. J. McConnell, W. McCown, S. J. Merrill, K. Messersmith, J. C. Moyer, M. Murphy, J. P. Myers, Jr., D. S. Noonan, G. Nutt, W. W. Oblitey, S. Olariu, G. Riccardi, G. Rice, N. Rickert, C. Riedesel, J. B. Rogers, G. Saito, W. Savitch, R. Schlafly, J. C. Schlimmer, S. Sells, Z. Shen, G. Sheppard, J. C. Simms, M. C. Slattery, J. Slimick, J. A. Slomka, J. Solderitsch, R. Steigerwald, L. Steinberg, C. A. Struble, C. L. Struble, W. J. Taffe, J. Talburt, P. Tonellato, P. Tromovitch, P. H. Winston, E. D. Winter, E. Wright, M. Ziegler, and one anonymous. To these individuals we give our sincere thanks.

Diane Christie designed Java and C++ manuals for the companion website in a previous edition, from which our new Python resources are descended. Thank you, Diane. Another thank you goes to Roger Eastman, who was the creative force behind the chapter-by-chapter activities that accompanied prior editions of the text, the DNA of which can still be found in the current edition's companion website activities.

My thanks to the good people at Pearson who have supported this project. Tracy Johnson, Erin Ault, Carole Snyder, and Scott Disanno in particular have brought their wisdom and many improvements to the book throughout the process.

Finally, my thanks to my wife, Petra, who distracted our three children for many long afternoons and evenings while I worked on this edition. She is my rock.

D.W.B.
Marquette University
January 01, 2018

Acknowledgments for the Global Edition

Pearson would like to thank and acknowledge the following people for their contributions to this Global Edition.

Contributor

Manasa S. (NMAM Institute of Technology)

Reviewers

Ajay Mittal (University Institute of Engineering and Technology)

Lindsay Ward (James Cook University)

Chapter 0

In this preliminary chapter, we consider the scope of computer science, develop a historical perspective, and establish a foundation from which to launch our study.

Introduction

ENDURING UNDERSTANDINGS AND LEARNING OUTCOMES

The study of algorithms is at the core of computer science.

LO. Explain the importance of algorithms in the field of computer science.

The recent history of computer science is characterized by rapid advancements in computing power, miniaturization, and connectivity.

LO. Identify major milestones in computer science history that have paved the way to our modern day technological society.

Advancements in computer science are profoundly impacting human culture and society.

LO. Discuss some of the social, ethical, or legal dilemmas that have resulted from advancements in computer science.

Computer science is the discipline that seeks to build a scientific foundation for such topics as computer design, computer programming, information processing, algorithmic solutions of problems, and the algorithmic process itself. It provides the underpinnings for today's computer applications as well as the foundations for tomorrow's computing infrastructure.

This book provides a comprehensive introduction to this science. We will investigate a wide range of topics including most of those that constitute a typical university computer science curriculum. We want to appreciate the full scope and dynamics of the field. Thus, in addition to the topics themselves, we will be interested in their historical development, the current state of research, and prospects for the future. Our goal is to establish a functional understanding of computer science—one that will support those who wish to pursue more specialized studies in the science as well as one that will enable those in other fields to flourish in an increasingly technical society.

0.1 The Role of Algorithms

We begin with the most fundamental concept of computer science—that of an **algorithm**. Informally, an algorithm is a set of steps that defines how a task is performed. (We will be more precise later, in Chapter 5.) For example, there are algorithms for cooking (called recipes), for finding your way through a strange city (more commonly called directions), for operating washing machines (usually displayed on the inside of the washer's lid or perhaps on the wall of a laundromat), for playing music (expressed in the form of sheet music), and for performing magic tricks (Figure 0.1).

Before a machine such as a computer can perform a task, an algorithm for performing that task must be discovered and represented in a form that is compatible with the machine. A representation of an algorithm is called a **program**. For the convenience of humans, computer programs are usually printed on paper or displayed on computer screens. For the convenience of machines, programs are encoded in a manner compatible with the technology of the machine. The process of developing a program, encoding it in machine-compatible form, and inserting it into a machine is called **programming**, or sometimes **coding**. Programs, and the algorithms they represent, are collectively referred to as **software**, in contrast to the machinery itself, which is known as **hardware**.

The study of algorithms began as a subject in mathematics. Indeed, the search for algorithms was a significant activity of mathematicians long before the development of today's computers. The goal was to find a single set of directions that described how all problems of a particular type could be solved. One of the best known examples of this early research is the long division algorithm for finding the quotient of two multiple-digit numbers. Another example is the Euclidean algorithm, discovered by the Ancient

Effect: The performer places some cards from a normal deck of playing cards face down on a table and mixes them thoroughly while spreading them out on the table. Then, as the audience requests either red or black cards, the performer turns over cards of the requested color.

Secret and Patter:

Step 1. From a normal deck of cards, select ten red cards and ten black cards. Deal these cards face up in two piles on the table according to color.

Step 2. Announce that you have selected some red cards and some black cards.

Step 3. Pick up the red cards. Under the pretense of aligning them into a small deck, hold them face down in your left hand and, with the thumb and first finger of your right hand, pull back on each end of the deck so that each card is given a slightly *backward* curve. Then place the deck of red cards face down on the table as you say, "Here are the red cards in this stack."

Step 4. Pick up the black cards. In a manner similar to that in step 3, give these cards a slight *forward* curve. Then return these cards to the table in a face-down deck as you say, "And here are the black cards in this stack."

Step 5. Immediately after returning the black cards to the table, use both hands to mix the red and black cards (still face down) as you spread them out on the tabletop. Explain that you are thoroughly mixing the cards.

Step 6. As long as there are face-down cards on the table, repeatedly execute the following steps:

6.1. Ask the audience to request either a red or a black card.

6.2. If the color requested is red and there is a face-down card with a concave appearance, turn over such a card while saying, "Here is a red card."

6.3. If the color requested is black and there is a face-down card with a convex appearance, turn over such a card while saying, "Here is a black card."

6.4. Otherwise, state that there are no more cards of the requested color and turn over the remaining cards to prove your claim.

Figure 0.1 An algorithm for a magic trick

Greek mathematician Euclid, for finding the greatest common divisor of two positive integers (Figure 0.2).

Once an algorithm for performing a task has been found, the performance of that task no longer requires an understanding of the principles on which the algorithm is based. Instead, the performance of the task is reduced to the process of merely following directions. (We can follow the long division algorithm to find a quotient or the Euclidean algorithm to find a greatest common

Description: This algorithm assumes that its input consists of two positive integers and proceeds to compute the greatest common divisor of these two values.

Procedure:

Step 1. Assign M and N the value of the larger and smaller of the two input values, respectively.

Step 2. Divide M by N, and call the remainder R.

Step 3. If R is not 0, then assign M the value of N, assign N the value of R, and return to step 2; otherwise, the greatest common divisor is the value currently assigned to N.

Figure 0.2 The Euclidean algorithm for finding the greatest common divisor of two positive integers

divisor without understanding why the algorithm works.) In a sense, the intelligence required to solve the problem at hand is encoded in the algorithm.

Capturing and conveying intelligence (or at least intelligent behavior) by means of algorithms allows us to build machines that perform useful tasks. Consequently, the level of intelligence displayed by machines is limited by the intelligence that can be conveyed through algorithms. We can construct a machine to perform a task only if an algorithm exists for performing that task. In turn, if no algorithm exists for solving a problem, then the solution of that problem lies beyond the capabilities of machines.

Identifying the limitations of algorithmic capabilities was solidified as a subject in mathematics in the 1930s with the publication of Kurt Gödel's incompleteness theorem. This theorem essentially states that in any mathematical theory encompassing our traditional arithmetic system, there are statements whose truth or falseness cannot be established by algorithmic means. In short, any complete study of our arithmetic system lies beyond the capabilities of algorithmic activities. This realization shook the foundations of mathematics, and the study of algorithmic capabilities that ensued was the beginning of the field known today as computer science. Indeed, it is the study of algorithms that forms the core of computer science.

0.2 The History of Computing

Today's computers have an extensive genealogy. One of the earlier computing devices was the abacus. History tells us that it probably had its roots in ancient China and was used in the early Greek and Roman civilizations. The machine is quite simple, consisting of beads strung on rods that are in turn mounted in a rectangular frame (Figure 0.3). As the beads are moved back and forth on the rods, their positions represent stored values. It is in the positions of

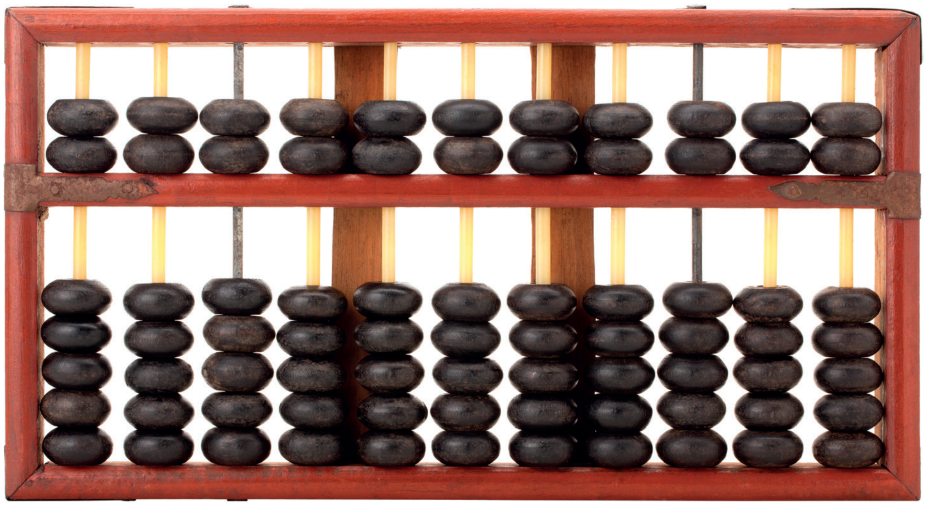


Figure 0.3 Chinese wooden abacus (Ekkapon/Shutterstock)

the beads that this “computer” represents and stores data. For control of an algorithm’s execution, the machine relies on the human operator. Thus, the abacus alone is merely a data storage system; it must be combined with a human to create a complete computational machine.

In the time period after the Middle Ages and before the Modern Era, the quest for more sophisticated computing machines was seeded. A few inventors began to experiment with the technology of gears. Among these were Blaise Pascal (1623–1662) of France, Gottfried Wilhelm Leibniz (1646–1716) of Germany, and Charles Babbage (1792–1871) of England. These machines represented data through gear positioning, with data being entered mechanically by establishing initial gear positions. Output from Pascal’s and Leibniz’s machines was achieved by observing the final gear positions. Babbage, on the other hand, envisioned machines that would print results of computations on paper so that the possibility of transcription errors would be eliminated.

As for the ability to follow an algorithm, we can see a progression of flexibility in these machines. Pascal’s machine was built to perform only addition. Consequently, the appropriate sequence of steps was embedded into the structure of the machine itself. In a similar manner, Leibniz’s machine had its algorithms firmly embedded in its architecture, although the operator could select from a variety of arithmetic operations it offered. Babbage’s Difference Engine (of which only a demonstration model was constructed) could be modified to perform a variety of calculations, but his Analytical Engine (never funded for construction) was designed to read instructions in the form of holes in paper cards. Thus Babbage’s Analytical Engine was programmable. In fact, Augusta Ada Byron (Ada Lovelace), who published a paper in which she demonstrated how Babbage’s Analytical Engine could be programmed to perform various computations, is often identified today as the world’s first programmer.

The idea of communicating an algorithm via holes in paper was not originated by Babbage. He got the idea from Joseph Jacquard (1752–1834), who, in 1801, had developed a weaving loom in which the steps to be performed during the weaving process were determined by patterns of holes in large thick cards made of wood (or cardboard). In this manner, the algorithm followed by the loom could be changed easily to produce different woven designs. Another beneficiary of Jacquard’s idea was Herman Hollerith (1860–1929), who applied the concept of representing information as holes in paper cards to speed up the tabulation process in the 1890 U.S. census. (It was this work by Hollerith that led to the creation of IBM.) Such cards ultimately came to be known as punched cards and survived as a popular means of communicating with computers well into the 1970s.

Nineteenth century technology was unable to cost-effectively produce the complex gear-driven machines of Pascal, Leibniz, and Babbage. But with the advances in electronics in the early 1900s, this barrier was overcome. Examples of this progress include the electromechanical machine of George Stibitz, completed in 1940 at Bell Laboratories, and the Mark I, completed in 1944 at Harvard University by Howard Aiken and a group of IBM engineers. These

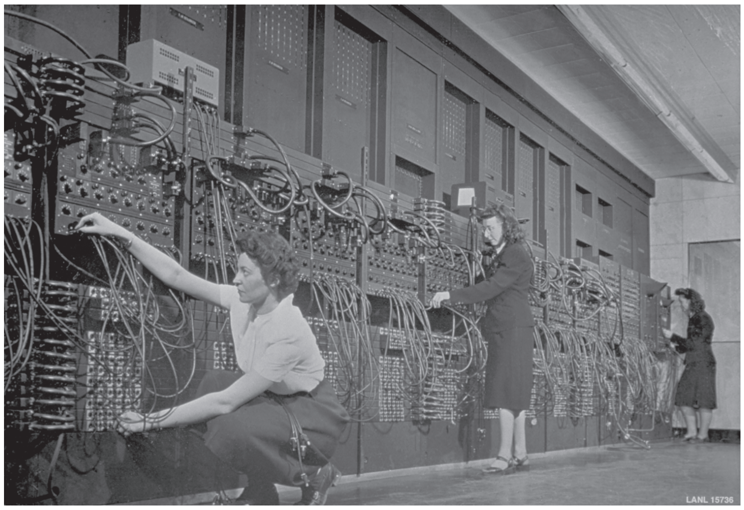


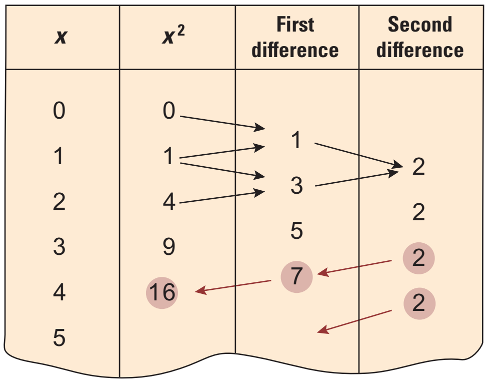
Figure 0.4 Three women operating the ENIAC's (Electronic Numerical Integrator and Computer) main control panel while the machine was at the Moore School. The machine was later moved to the U.S. Army's Ballistics Research Laboratory. (Courtesy of U.S. Army.)

machines made heavy use of electronically controlled mechanical relays. In this sense, they were obsolete almost as soon as they were built, because other researchers were applying the technology of vacuum tubes to construct totally electronic computers. The first of these vacuum tube machines was apparently the Atanasoff-Berry machine, constructed during the period from 1937 to 1941 at Iowa State College (now Iowa State University) by John Atanasoff and his assistant, Clifford Berry. Another was a machine called Colossus, built under the direction of Tommy Flowers in England to decode German messages during the latter part of World War II. (Actually, as many as ten of these machines were apparently built, but military secrecy and issues of national security kept their existence from becoming part of the "computer family tree.") Other, more flexible machines, such as the ENIAC (Electronic Numerical Integrator and Computer) developed by John Mauchly and J. Presper Eckert at the Moore School of Electrical Engineering, (Figure 0.4), University of Pennsylvania, soon followed.

From that point on, the history of computing machines has been closely linked to advancing technology, including the invention of transistors (for which physicists William Shockley, John Bardeen, and Walter Brattain were awarded a Nobel Prize) and the subsequent development of complete circuits constructed as single units, called integrated circuits (for which Jack Kilby also won a Nobel Prize in physics). With these developments, the room-sized machines of the 1940s were reduced over the decades to the size of single cabinets. At the same time, the processing power of computing machines began to double every two years (a trend that has continued to this day). As work on integrated circuitry progressed, many of the components within a computer became readily available on the open market as integrated circuits encased in toy-sized blocks of plastic called chips.

Babbage's Difference Engine

The machines designed by Charles Babbage were truly the forerunners of modern computer design. If technology had been able to produce his machines in an economically feasible manner and if the data processing demands of commerce and government had been on the scale of today's requirements, Babbage's ideas could have led to a computer revolution in the 1800s. As it was, only a demonstration model of his Difference Engine was constructed in his lifetime. This machine determined numerical values by computing "successive differences." We can gain an insight to this technique by considering the problem of computing the squares of the integers. We begin with the knowledge that the square of 0 is 0, the square of 1 is 1, the square of 2 is 4, and the square of 3 is 9. With this, we can determine the square of 4 in the following manner (see the following diagram). We first compute the differences of the squares we already know: $1^2 - 0^2 = 1$, $2^2 - 1^2 = 3$, and $3^2 - 2^2 = 5$. Then we compute the differences of these results: $3 - 1 = 2$, and $5 - 3 = 2$. Note that these differences are both 2. Assuming that this consistency continues (mathematics can show that it does), we conclude that the difference between the value $(4^2 - 3^2)$ and the value $(3^2 - 2^2)$ must also be 2. Hence $(4^2 - 3^2)$ must be 2 greater than $(3^2 - 2^2)$, so $4^2 - 3^2 = 7$ and thus $4^2 = 3^2 + 7 = 16$. Now that we know the square of 4, we could continue our procedure to compute the square of 5 based on the values of 1^2, 2^2, 3^2, and 4^2. (Although a more in–depth discussion of successive differences is beyond the scope of our current study, students of calculus may wish to observe that the preceding example is based on the fact that the derivative of $y = x^2$ is a straight line with a slope of 2.)



A major step toward popularizing computing was the development of desktop computers. The origins of these machines can be traced to the computer hobbyists who built homemade computers from combinations of chips. It was within this "underground" of hobby activity that Steve Jobs and Stephen Wozniak built a commercially viable home computer and, in 1976, established Apple Computer, Inc. (now Apple Inc.) to manufacture and market their products. Other companies that marketed similar products were Commodore, Heathkit, and Radio Shack. Although these products were popular among computer hobbyists, they were not widely accepted by the business community, which continued to look to the well-established IBM and its large mainframe computers for the majority of its computing needs.

Augusta Ada Byron

Augusta Ada Byron, Countess of Lovelace, has been the subject of much commentary in the computing community. She lived a somewhat tragic life of less than 37 years (1815–1852) that was complicated by poor health and the fact that she was a nonconformist in a society that limited the professional role of women. Although she was interested in a wide range of science, she concentrated her studies in mathematics. Her interest in "compute science" began when she became fascinated by the machines of Charles Babbage at a demonstration of a prototype of his Difference Engine in 1833. Her contribution to computer science stems from her translation from French into English of a paper discussing Babbage's designs for the Analytical Engine. To this translation, Babbage encouraged her to attach an addendum describing applications of the engine and containing examples of how the engine could be programmed to perform various tasks. Babbage's enthusiasm for Ada Byron's work was apparently motivated by his hope that its publication would lead to financial backing for the construction of his Analytical Engine. (As the daughter of Lord Byron, Ada Byron held celebrity status with potentially significant financial connections.) This backing never materialized, but Ada Byron's addendum has survived and is considered to contain the first examples of computer programs. The degree to which Babbage influenced Ada Byron's work is debated by historians. Some argue that Babbage made major contributions, whereas others contend that he was more of an obstacle than an aid. Nonetheless, Augusta Ada Byron is recognized today as the world's first programmer, a status that was certified by the U.S. Department of Defense when it named a prominent programming language (Ada) in her honor.

In 1981, IBM introduced its first desktop computer, called the personal computer, or PC, whose underlying software was developed by a newly-formed company known as Microsoft. The PC was an instant success and legitimized the desktop computer as an established commodity in the minds of the business community. Today, the term PC is widely used to refer to all those machines (from various manufacturers) whose design has evolved from IBM's initial desktop computer, most of which continue to be marketed with software from Microsoft. At times, however, the term PC is used interchangeably with the generic terms *desktop* or *laptop*.

As the twentieth century drew to a close, the ability to connect individual computers in a world-wide system called the **Internet** was revolutionizing communication. In this context, Tim Berners-Lee (a British scientist) proposed a system by which documents stored on computers throughout the Internet could be linked together producing a maze of linked information called the **World Wide Web** (often shortened to "Web"). To make the information on the Web accessible, software systems, called **search engines**, were developed to "sift through" the Web, "categorize" their findings, and then use the results to assist users researching particular topics. Major players in this field are Google, Yahoo, and Microsoft. These companies continue to expand their Web-related activities, often in directions that challenge our traditional way of thinking.

Google

Founded in 1998, Google LLC (formerly Google Inc.) has become one of the world's most recognized technology companies. Its core service, the Google search engine, is used by millions of people to find documents on the World Wide Web. In addition, Google provides electronic mail service (called Gmail), an Internet-based video-sharing service (called YouTube), and a host of other Internet services (including Google Maps, Google Calendar, Google Earth, Google Books, and Google Translate).

However, in addition to being a prime example of the entrepreneurial spirit, Google also provides examples of how expanding technology is challenging society. For example, Google's search engine has led to questions regarding the extent to which an international company should comply with the wishes of individual governments; YouTube has raised questions regarding the extent to which a company should be liable for information that others distribute through its services as well as the degree to which the company can claim ownership of that information; Google Books has generated concerns regarding the scope and limitations of intellectual property rights; and Google Maps has been accused of violating privacy rights.

At the same time that desktop and laptop computers were being accepted and used in homes, the miniaturization of computing machines continued. Today, tiny computers are embedded within a wide variety of electronic appliances and devices. Automobiles may now contain dozens of small computers running Global Positioning Systems (GPS), monitoring the function of the engine, and providing voice command services for controlling the car's audio and phone communication systems.

Perhaps the most revolutionary application of computer miniaturization is found in the expanding capabilities of **smartphones**, hand-held general-purpose computers on which telephony is only one of many applications. More powerful than the supercomputers of prior decades, these pocket-sized devices are equipped with a rich array of sensors and interfaces including cameras, microphones, compasses, touch screens, accelerometers (to detect the phone's orientation and motion), and a number of wireless technologies to communicate with other smartphones and computers. Many argue that the smartphone is having a greater effect on global society than the PC revolution.

0.3 An Outline of Our Study

This text follows a bottom-up approach to the study of computer science, beginning with such hands-on topics as computer hardware and leading to the more abstract topics such as algorithm complexity and computability. The result is that our study follows a pattern of building larger and larger abstract tools as our understanding of the subject expands.

We begin by considering topics dealing with the design and construction of machines for executing algorithms. In Chapter 1 (Data Storage), we look at how information is encoded and stored within modern computers, and in Chapter 2 (Data Manipulation), we investigate the basic internal operation of a simple computer. Although part of this study involves technology, the general theme is technology independent. That is, such topics as digital circuit design, data encoding and compression systems, and computer architecture are relevant over a wide range of technology and promise to remain relevant regardless of the direction of future technology.

In Chapter 3 (Operating Systems), we study the software that controls the overall operation of a computer. This software is called an operating system. It is a computer's operating system that controls the interface between the machine and its outside world, protecting the machine and the data stored within from unauthorized access, allowing a computer user to request the execution of various programs, and coordinating the internal activities required to fulfill the user's requests.

In Chapter 4 (Networking and the Internet), we study how computers are connected to each other to form computer networks and how networks are connected to form internets. This study leads to topics such as network protocols, the Internet's structure and internal operation, the World Wide Web, and numerous issues of security.

Chapter 5 (Algorithms) introduces the study of algorithms from a more formal perspective. We investigate how algorithms are discovered, identify several fundamental algorithmic structures, develop elementary techniques for representing algorithms, and introduce the subjects of algorithm efficiency and correctness.

In Chapter 6 (Programming Languages), we consider the subject of algorithm representation and the program development process. Here we find that the search for better programming techniques has led to a variety of programming methodologies or paradigms, each with its own set of programming languages. We investigate these paradigms and languages as well as consider issues of grammar and language translation.

Chapter 7 (Software Engineering) introduces the branch of computer science known as software engineering, which deals with the problems encountered when developing large software systems. The underlying theme is that the design of large software systems is a complex task that embraces problems beyond those of traditional engineering. Thus, the subject of software engineering has become an important field of research within computer science, drawing from such diverse fields as engineering, project management, personnel management, programming language design, and even architecture.

In the next two chapters, we look at ways data can be organized within a computer system. In Chapter 8 (Data Abstractions), we introduce techniques traditionally used for organizing data in a computer's main memory and then trace the evolution of data abstraction from the concept of primitives to today's object-oriented techniques. In Chapter 9 (Database Systems), we consider methods traditionally used for organizing data in a computer's mass storage and investigate how extremely large and complex database systems are implemented.

In Chapter 10 (Computer Graphics), we explore the subject of graphics and animation, a field that deals with creating and photographing virtual worlds. Based on advancements in the more traditional areas of computer science such as machine architecture, algorithm design, data structures, and software engineering, the discipline of graphics and animation has seen significant progress and has now blossomed into an exciting, dynamic subject. Moreover, the field exemplifies how various components of computer science combine with other disciplines such as physics, art, and photography to produce striking results.

In Chapter 11 (Artificial Intelligence), we learn that to develop more useful machines, computer science has turned to the study of human intelligence for insight. The hope is that by understanding how our own minds reason and perceive, researchers will be able to design algorithms that mimic these processes and thus transfer comparable capabilities to machines. The result is the area of computer science known as artificial intelligence, which leans heavily on research in such areas as psychology, biology, and linguistics.

We close our study with Chapter 12 (Theory of Computation) by investigating the theoretical foundations of computer science—a subject that allows us to understand the limitations of algorithms (and thus machines). Here we identify some problems that cannot be solved algorithmically (and therefore lie beyond the capabilities of machines) as well as learn that the solutions to many other problems require such enormous time or space that they are also unsolvable from a practical perspective. Thus, it is through this study that we are able to grasp the scope and limitations of algorithmic systems.

In each chapter, our goal is to explore the subject deeply enough to enable true understanding. We want to develop a working knowledge of computer science—a knowledge that will allow you to understand the technical society in which you live and to provide a foundation from which you can learn on your own as science and technology advance.

0.4 The Overarching Themes of Computer Science

In addition to the main topics of each chapter as listed above, we also hope to broaden your understanding of computer science by incorporating several overarching themes. The miniaturization of computers and their expanding capabilities have brought computer technology to the forefront of today's society, and computer technology is so prevalent that familiarity with it is fundamental to being a member of the modern world. Computing technology has altered the ability of governments to exert control; had enormous impact on global economics; led to startling advances in scientific research; revolutionized the role of data collection, storage, and applications; provided new means for people to communicate and interact; and has repeatedly challenged society's status quo. The result is a proliferation of subjects surrounding computer science, each of which is now a significant field of study in its own right. Moreover, as with mechanical engineering and physics, it is often difficult to draw a line between these fields and computer science itself. Thus, to gain a proper perspective, our study will not only cover topics central to the core of computer science but also will explore a variety of disciplines dealing with both applications and consequences of the science. Indeed, an introduction to computer science is an interdisciplinary undertaking.

As we set out to explore the breadth of the field of computing, it is helpful to keep in mind the main themes that unite computer science. While the codification of the "Seven Big Ideas of Computer Science"[1] post-dates the first ten editions of this book, they closely parallel the themes of the chapters to come. The "Seven Big Ideas" are, briefly: Algorithms, Abstraction, Creativity, Data, Programming, Internet, and Impact. In the chapters that follow, we include a variety of topics, in each case introducing central ideas of the topic, current areas of research, and some of the techniques being applied to advance knowledge in that realm. Watch for the "Big Ideas" as we return to them again and again.

Algorithms

Limited data storage capabilities and intricate, time-consuming programming procedures restricted the complexity of the algorithms used in the earliest computing machines. However, as these limitations began to disappear, machines were applied to increasingly larger and more complex tasks. As attempts to express the composition of these tasks in algorithmic form began to tax the abilities of the human mind, more and more research efforts were directed toward the study of algorithms and the programming process.

It was in this context that the theoretical work of mathematicians began to pay dividends. As a consequence of Gödel's incompleteness theorem, mathematicians had already been investigating those questions regarding algorithmic processes that advancing technology was now raising. With that, the stage was set for the emergence of a new discipline known as *computer science*.

Today, computer science has established itself as the science of algorithms. The scope of this science is broad, drawing from such diverse subjects as mathematics, engineering, psychology, biology, business administration, and linguistics. Indeed, researchers in different branches of computer science may have very distinct definitions of the science. For example, a researcher in the

[1]https://apstudent.collegeboard.org/apcourse/ap-computer-science-principles

field of computer architecture may focus on the task of miniaturizing circuitry and thus view computer science as the advancement and application of technology. But a researcher in the field of database systems may see computer science as seeking ways to make information systems more useful. And a researcher in the field of artificial intelligence may regard computer science as the study of intelligence and intelligent behavior.

Nevertheless, all of these researchers are involved in aspects of the science of algorithms. Given the central role that algorithms play in computer science (see Figure 0.5), it is instructive to identify some questions that will provide focus for our study of this big idea.

- Which problems can be solved by algorithmic processes?
- How can the discovery of algorithms be made easier?
- How can the techniques of representing and communicating algorithms be improved?
- How can the characteristics of different algorithms be analyzed and compared?
- How can algorithms be used to manipulate information?
- How can algorithms be applied to produce intelligent behavior?
- How does the application of algorithms affect society?

Abstraction

The term **abstraction**, as we are using it here, refers to the distinction between the external properties of an entity and the details of the entity's internal composition. It is abstraction that allows us to ignore the internal details of a complex device such as a computer, automobile, or microwave oven and use it as a single, comprehensible unit. Moreover, it is by means of abstraction that such complex systems are designed and manufactured in the first place. Computers, automobiles, and microwave ovens are constructed from

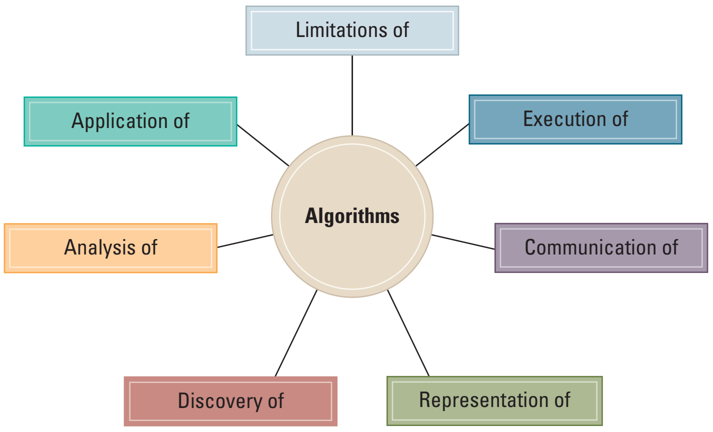


Figure 0.5 The central role of algorithms in computer science

components, each of which represents a level of abstraction at which the use of the component is isolated from the details of the component's internal composition.

It is by applying abstraction that we are able to construct, analyze, and manage large, complex computer systems, which would be overwhelming if viewed in their entirety at a detailed level. At each level of abstraction, we view the system in terms of components, called **abstract tools**, whose internal composition we ignore. This allows us to concentrate on how each component interacts with other components at the same level and how the collection as a whole forms a higher-level component. Thus, we are able to comprehend the part of the system that is relevant to the task at hand rather than being lost in a sea of details.

We emphasize that abstraction is not limited to science and technology. It is an important simplification technique with which our society has created a lifestyle that would otherwise be impossible. Few of us understand how the various conveniences of daily life are actually implemented. We eat food and wear clothes that we cannot produce by ourselves. We use electrical devices and communication systems without understanding the underlying technology. We use the services of others without knowing the details of their professions. With each new advancement, a small part of society chooses to specialize in its implementation, while the rest of us learn to use the results as abstract tools. In this manner, society's warehouse of abstract tools expands, and society's ability to progress increases.

Abstraction is a recurring pillar of our study. We will learn that computing equipment is constructed in levels of abstract tools. We will also see that the development of large software systems is accomplished in a modular fashion in which each module is used as an abstract tool in larger modules. Moreover, abstraction plays an important role in the task of advancing computer science itself, allowing researchers to focus attention on particular areas within a complex field. In fact, the organization of this text reflects this characteristic of the science. Each chapter, which focuses on a particular area within the science, is often surprisingly independent of the others, yet together the chapters form a comprehensive overview of a vast field of study.

Creativity

While computers may merely be complex machines mechanically executing rote algorithmic instructions, we shall see that the field of computer science is an inherently creative one. Discovering and applying new algorithms is a human activity that depends on our innate desire to apply our tools to solve problems in the world around us. Computer science not only extends forms of expression spanning the visual, language, and musical arts, but also enables new modes of digital expression that pervade the modern world.

Creating large software systems is much less like following a cookbook recipe than it is like conceiving of a grand new sculpture. Envisioning its form