D is for Digital

WHAT A WELL-INFORMED PERSON SHOULD KNOW ABOUT COMPUTERS AND COMMUNICATIONS

Brian W. Kernighan

D is for Digital

Also by Brian Kernighan

The Elements of Programming Style (with P. J. Plauger)

Software Tools (with P. J. Plauger)

Software Tools in Pascal (with P. J. Plauger)

The C Programming Language (with D. M. Ritchie)

The AWK Programming Language (with A. V. Aho and P. J. Weinberger)

The Unix Programming Environment (with R. Pike)

AMPL: A Modeling Language for Mathematical Programming (with R. Fourer and D. M. Gay)

The Practice of Programming (with R. Pike)

D is for Digital

What a well-informed person should know
about computers and communications

Brian W. Kernighan

DisforDigital.net

Typeset in Times, Courier and Helvetica using groff, ghostscript, and other open source Unix tools.

Published by DisforDigital.net.
For information, mail `info@disfordigital.net`.

Printed by CreateSpace.

ISBN-13: 978-1463733896
ISBN-10: 1463733895

Copyright © 2011 by Brian W. Kernighan

ALL RIGHTS RESERVED.

No part of this book may be reproduced or transmitted in any form or by any means, electronic, optical or mechanical, including photocopying or recording, or by any information storage and retrieval system, without express written permission from the copyright owner.

10 9 8 7 6 5 4 3 2 1 1109221944

For Meg

Contents

Preface ix

Introduction 1

Part I: Hardware 7

1 What's in a Computer? 9
- 1.1 Logical Construction 10
  - 1.1.1 CPU 11
  - 1.1.2 RAM 12
  - 1.1.3 Disks and secondary storage 13
  - 1.1.4 Et cetera 14
- 1.2 Physical Construction 15
- 1.3 Moore's Law 18

2 Bits, Bytes, and Representation of Information 21
- 2.1 Analog versus Digital 21
- 2.2 Analog-Digital Conversion 23
- 2.3 Bits, Bytes, and Binary 28
  - 2.3.1 Bits 28
  - 2.3.2 Powers of two and powers of ten 30
  - 2.3.3 Binary numbers 31
  - 2.3.4 Bytes 32
- 2.4 Summary 34

3 Inside the CPU 35
- 3.1 The Toy Computer 36
- 3.2 Real CPUs 41
- 3.3 Caching 43

3.4 Other Kinds of Computers 44

Wrapup on Hardware 47

Part II: Software 51

4 Algorithms 53
4.1 Linear Algorithms 54
4.2 Binary Search 56
4.3 Sorting 57
4.4 Hard Problems and Complexity 60
4.5 Summary 62

5 Programming and Programming Languages 65
5.1 Assembly Language 66
5.2 High-Level Languages 67
5.3 Software Development 73
5.3.1 Libraries, interfaces, and development kits 74
5.3.2 Bugs 75
5.4 Software as Property 77
5.4.1 Intellectual property 78
5.4.2 Standards 81
5.4.3 Open source 82

6 Software Systems 85
6.1 Operating Systems 86
6.2 How an Operating System Works 90
6.2.1 System calls 91
6.2.2 Device drivers 91
6.3 Other Operating Systems 92
6.4 File Systems 93
6.4.1 Disk file systems 94
6.4.2 Removing files 95
6.4.3 Other file systems 97
6.5 Applications 98
6.6 Layers of Software 100

7 Learning to Program 103
7.1 Programming Language Concepts 104
7.2 A First JavaScript Example 105
7.3 A Second JavaScript Example 107
7.4 Loops 108
7.5 Conditionals 109
7.6 Libraries and Interfaces 111
7.7 How JavaScript Works 112

Wrapup on Software 115

Part III: Communications 117

8 Networking 123

8.1 Telephones and Modems 124
8.2 Cable and DSL 124
8.3 Local Area Networks and Ethernet 126
8.4 Wireless 129
8.5 Cell Phones 131
8.6 Summary 134

9 The Internet 135
9.1 An Internet Overview 136
9.2 Domain Names and Addresses 139
9.2.1 Domain Name System 139
9.2.2 IP addresses 140
9.2.3 Root servers 141
9.2.4 Registering your own domain 141
9.3 Routing 142
9.4 Protocols 144
9.4.1 IP, the Internet Protocol 144
9.4.2 TCP, the Transmission Control Protocol 145
9.5 Higher-level Protocols 147
9.5.1 FTP: File Transfer Protocol 148
9.5.2 Telnet: remote login 148
9.5.3 SMTP: Simple Mail Transfer Protocol 149
9.5.4 File sharing and peer-to-peer protocols 151
9.5.5 An aside on copyright 153
9.5.6 Summary of high-level protocols 154
9.6 Bandwidth 155
9.7 Compression 155
9.8 Error Detection and Correction 157
9.9 Summary 159

10 The World Wide Web 161
10.1 How the Web Works 162
10.2 HTML 164
10.3 Forms 165
10.4 Cookies 166
10.5 Active Content in Web Pages 167
10.6 Active Content Elsewhere 169
10.7 Viruses and Worms 170
10.8 Web Security 172
10.8.1 Attacks on clients 172
10.8.2 Attacks on servers 175
10.8.3 Attacks on information in transit 176
10.8.4 Defending yourself 177
10.9 Cryptography 178
10.9.1 Secret-key cryptography 180
10.9.2 Public-key cryptography 181
10.10 Summary 184

11 Data, Information, and Privacy 187
11.1 Search 188
11.2 Tracking 192
11.3 Databases, Information, and Aggregation 196
11.4 Giving It Away 198
11.5 Cloud Computing 200
11.6 Summary 204

12 Wrapping Up 205

Notes 209

Glossary 213

Index 219

Preface

Since the fall of 1999, I have been teaching a Princeton course called "Computers in Our World." The course title is embarrassingly vague, but I had to invent it in less than five minutes one day and then it became too hard to change. Teaching the course itself, however, has proven to be the most fun thing that I do, in a job that is almost all enjoyable.

The course is based on the observation that computers and computing are all around us. Some computing is highly visible: every student has a computer, each one of which is far more powerful than the single computer that cost several million dollars, occupied a very large room, and served the whole Princeton campus when I was a graduate student there in 1964. Indeed, a typical cell phone has much more computing power than that same computer. Every student has high speed Internet access, as does at least half the population of the United States in their homes. We search and shop online, and we use email, texting and social networks to keep in touch with friends and families.

But this is only part of a computing iceberg, much of which lies hidden below the surface. We don't see and usually don't think about the computers that lurk within appliances, cars, airplanes and the pervasive electronic gadgets that we take for granted—cameras, video recorders, DVD players, tablets, GPS navigators, games. Sometimes their pluses and minuses come to the surface, almost accidentally, as in a newspaper article that quoted an executive of Hewlett-Packard as saying "In essence, a digital camera is a computer with a lens." The same article also quoted an unhappy consumer: "This isn't a camera, it's a computer," surely a comment on how hard it sometimes is to use computers.

Nor do we think much about the degree to which infrastructure depends on computing: the telephone network, television and cable, air traffic control, the power grid,

and banking and financial services.

The pervasive nature of computing affects us in unexpected ways. Although we are from time to time reminded of the growth of surveillance systems, incursions into our privacy, and the perils of electronic voting, we perhaps do not realize the extent to which they are enabled by computing and communications.

Most people will not be directly involved in creating such systems, but everyone is strongly affected by them, and some will have to make important decisions about them. Wouldn't it be better if people had a better understanding of computers? An educated person ought to know at least the rudiments of computing: what computers can do and how they do it; what they can't do at all and what's merely extremely hard right now; how they talk to each other and what happens when they do; and the many ways that computing and communications influence the world around us.

The students in my course are not technical—no engineers, physicists or mathematicians. Instead they are English and politics majors, historians, classicists, economists, musicians and artists, a wonderful slice through the humanities and social sciences. By the end of the course these bright people should be able to read and understand a newspaper article about computing, to learn more from it, and perhaps to spot places where it might not be accurate. More broadly, I want my students to be intelligently skeptical about technology, to know that it is often a good thing but not a panacea; conversely, though it sometimes has bad effects, technology is not an unmitigated evil.

A fine book by Richard Muller called *Physics for Future Presidents* attempts to explain the scientific and technical background underlying major issues that leaders have to grapple with—nuclear threats, terrorists, energy, global warming, and the like. Well-informed citizens without aspirations to be president should know something of these topics as well. Although there are many differences between the scientific principles and reasoning that Muller is able to bring to bear on his topics, and what one should understand about computing, the approach is a good metaphor for what I would like to accomplish: "Computing for Future Presidents."

What should the president know about computing? What should a well-informed person know about computing? Everyone will have their own ideas; here are mine.

There are three core technical areas—hardware, software, and communications—and the book is organized around them.

Hardware is the tangible part, the computers we can see and touch, that sit in our homes and offices, and that we carry around in our phones. What's inside a computer, how does it work, how is it built? How does it store and process information? What are bits and bytes, and how do we use them to represent music, movies, and everything else?

Software, the instructions that tell computers what to do, is by contrast hardly tangible at all. What can we compute, and how fast can we compute it? How do we tell computers what to do? Why is it so hard to make them work right? Why are they so often hard to use?

Communications means computers, phones, and other devices talking to each other on our behalf and letting us talk to each other: the Internet, the Web, email and social networks. How do these work? The rewards are obvious, but what are the risks, especially to our privacy and security, and how can they be mitigated?

To this trio one might add *data*, which is all the information that hardware and software collect, store and process, and which communications systems send round the world. Some of this is data we contribute voluntarily, whether prudently or not, by uploading pictures and videos. Some is personal information about us, usually gathered and shared without our knowledge, let alone agreement.

President or not, you should know about the world of computing because it affects you personally. No matter how non-technical your life and work, you're going to have to interact with technology and technical people. Knowing something of how devices and systems operate is a big advantage, even something as simple as recognizing when a salesperson or a help line is not telling you the whole truth. Indeed, ignorance can be directly harmful. If you don't understand viruses, phishing and similar threats, you become more susceptible to them. If you don't know how social networks leak, or even broadcast, information that you thought was private, you're likely to reveal much more than you realize. If you're not aware of the headlong rush by commercial interests to exploit what they have learned about your life, you're giving up privacy for little benefit. If you don't know why it's risky to do your personal banking in a coffee shop or an airport, you're vulnerable to theft of money and identity.

I debated for quite a while whether this book should be a textbook, with exercises, problem sets and lists of things to think about at the ends of chapters. Or should it be what publishers call a "trade" book, something for general readers, with lots of anecdotes but nothing that looks too much like school? In the end, it's some of each. No matter what, it's certainly not "scholarly," with footnotes, references and long sentences. That's deadly for the writer, let alone the poor reader. There are no footnotes, and the references to other sources make no pretense of being complete. Any topic could be greatly expanded on, but to treat each topic in depth would make the book ten times longer and probably not much more useful. The notes at the end include some books that I particularly like, and other resources like web sites that might be helpful supplements.

The book is meant to be read from front to back but you might prefer to skip ahead to topics of personal interest and come back later. For example, you could begin by reading about networks, cell phones, the Internet, the web and privacy issues starting in Chapter 8; you might have to look back at earlier chapters to understand a few parts, but mostly it will be accessible. You can skip anything quantitative, for instance how binary numbers work in Chapter 2, and ignore details of programming languages in a couple of chapters. A glossary at the end gives brief definitions and explanations of important technical terms and acronyms.

Any book about computing can become dated quickly, so it's guaranteed that some details will be wrong or at least out of date by the time you read this. I've tried

to make sure that things of lasting value are clearly identified; for the rest, check out the book's web site at `kernighan.com` for updates, corrections, extra material, and the like.

My goal for this book is that you will come away with some appreciation for an amazing technology and a real understanding of how it works, where it came from, and where it might be going in the future. Along the way, perhaps you'll pick up a helpful way of thinking about the world. I hope so.

Acknowledgments

I am deeply indebted to friends and colleagues for generous help and advice. In particular, Jon Bentley provided detailed comments on almost every page of several drafts. Clay Bavor, Dan Bentley, Hildo Biersma, Stu Feldman, Gerard Holzmann, Joshua Katz, Mark Kernighan, Meg Kernighan, Paul Kernighan, David Malan, Tali Moreshet, Jon Riecke, Mike Shih, Bjarne Stroustrup, Howard Trickey, and John Wait read complete drafts with great care, made many helpful suggestions, and saved me from some major gaffes. I also thank Jennifer Chen, Doug Clark, Steve Elgersma, Avi Flamholz, Henry Leitner, Michael Li, Hugh Lynch, Patrick McCormick, Jacqueline Mislow, Jonathan Rochelle, Corey Thompson, and Chris Van Wyk for valuable comments. I hope that they will recognize the many places where I took their advice, but not notice the few where I did not.

David Brailsford offered a great deal of helpful advice on self-publishing and text formatting, based on his own hard-won experience. Greg Doench and Greg Wilson were generous with advice about publishing. I am indebted to Gerard Holzmann and John Wait for photographs.

Harry Lewis was my host at Harvard during the 2010-2011 academic year, when the first few drafts of the book were written. Harry's advice and his experience teaching an analogous course were of great value, as were his comments on multiple drafts. Harvard's School of Engineering and Applied Sciences and the Berkman Center for Internet and Society provided office space and facilities, a friendly and stimulating environment, and (yes, there is such a thing!) regular free lunches.

I am especially grateful to the many hundreds of students who have taken COS 109, "Computers in our World." Their interest, enthusiasm and friendship have been a continual source of inspiration. I hope that when they are running the world a few years from now, they will have profited in some way from the course.

Introduction

"Any sufficiently advanced technology is indistinguishable from magic."
Arthur C. Clarke, Report on Planet 3, *Technology and the Future*, 1972

Here are two stories about technology, one of worldwide interest, the other entirely personal and close to home.

In January 2011, popular revolutions in Tunisia and Egypt overthrew long-time dictators and led to a breath of freedom in countries that had suffered repression for many years. In both cases, huge numbers of ordinary citizens broadcast news of events and rallying places by Twitter, they took cell phone pictures and movies of protests and police brutality, and they posted them on Facebook and YouTube to inspire their fellow citizens and inform the world. In Tunisia, during the most dramatic periods citizens posted nearly 20,000 tweets per hour, which were posted to Facebook and blogs, and re-tweeted. For Egypt, Google quickly wrote an application called speak2tweet that let anyone send a tweet by making a phone call and speaking the message; the messages were automatically converted into text and posted. One could read the messages in their original languages (often French and Arabic) or have them spoken aloud, or translated into other languages through Google Translate. In both countries, government efforts to control communications during the events were unsuccessful, though whether the revolutions will lead to long-term change is unclear.

Shortly before Christmas in 2010, I wondered briefly about buying a bigger television set. I turned on my Android phone, and spoke the words "big screen TV" into a search engine. Back came a list of a dozen or more sets with 45-inch or larger screens. Most were way too expensive but there was one at $650 that looked promising. I clicked the link and was offered a choice of buying online or checking the

situation at local stores. That latter path led to a list of more than half a dozen stores, listed by increasing distance from my home, where I was sitting at the time. There were customer reviews of both the set and the stores themselves. I could order it right away to be picked up, and there were driving instructions to the selected store, with time, distance, and turn-by-turn directions for the fastest route; current traffic was superimposed on images that gave a perspective view of the terrain and street level views of the roads. In the end I decided to wait—my current television set is big enough for the modest amount that I watch it—but I was tempted.

Do these stories seem like magic, or do you just take the technical wizardry for granted? Either way, this is today's technology, not even advanced, and certainly not magic. If there's any magic at all, it's in the degree to which the technological world has changed so much and so quickly in the past ten or twenty years, and in its unimaginable effect on people's lives.

Let's look briefly at some of today's technologies as they were twenty years ago, say in the early 1990s. If you're a college student or recent graduate right now, this is what the world was like when you were born. Dinosaurs no longer walked the earth (your parents excepted) but most of what I described above didn't exist at all.

Phones. In 1990, telephones were bulky, had a rotary dial or 12 numeric buttons and no display, were connected by wires to phone outlets in homes and offices, and could only be used to talk to people, or, with the help of special equipment, to connect to fax machines and computers. By 2000, cell phones were becoming common but they could only be used for phone calls. Cell phones are now ubiquitous, outnumbering wired phones nearly everywhere (80 percent of Egypt's population have cell phones; in Tunisia, it's about 75 percent). Smartphones like the iPhone (released in mid 2007) have an amazing array of capabilities that can be extended by downloading applications from Apple's App Store, which opened for business in the middle of 2008, or the Android Market.

Pictures. In 1990, most cameras took pictures on film, a roll of plastic coated with sophisticated dyes that changed properties after the film was exposed to light (by taking a picture) and run through an elaborate sequence of chemical processes ("developed"). A roll of film held at most 36 pictures, and there was no way to see the results without having the roll developed, a process that took anywhere from an hour to a week. Once that was done, all you had was a piece of paper; if you wanted to share a picture with friends, you had to show it to them in person or mail it to them (in a paper envelope). You probably kept your pictures at home in envelopes or boxes scattered around the house. Extra copies cost another trip to the store and more money. Photography was expensive enough that most people didn't take all that many pictures. But digital cameras began to supplant film cameras, slowly at first and then rapidly, to the point where film has pretty much disappeared for most people. Today, photographs are sent by email, uploaded to web sites like Facebook and Flickr, and downloaded to digital picture frames. High quality photo printers are cheap if you do want paper copies, and inexpensive point and shoot cameras are

being superseded by cell phone cameras.

Music. In 1990 music was distributed as albums on CDs and audio tapes; long-playing vinyl records ("LPs") were on the way out, though still common. If you wanted to make a copy of a song to share with a friend or listen to in your car, you would likely use a cassette recorder, though you might be able to make a copy of a CD. There wasn't much if any downloading of music either. That changed dramatically in 1999 with Napster, which made it possible to easily share music online. Napster died, or, more accurately, was killed quickly, but online services like iTunes (which dates from 2003) rose out of the ashes, selling single songs. On the portable hardware side, we went from boomboxes and cassette players to CD players to MP3 players like the iPod before settling on the phone.

Email. Twenty years ago, most people didn't use email, or even know what it was. The movie "You've Got Mail," starring Meg Ryan and Tom Hanks, appeared in 1998, when email was just starting to catch on with the general public, for whom "mail" still meant paper envelopes brought by a letter carrier. Today most people use online services like Gmail for personal correspondence; their mail exists somewhere on the Internet, accessed from their phones.

Facebook. In 1990 there was no Facebook; it was born in 2004 (and Mark Zuckerberg, its founder, was born in 1984). If you wanted to keep in touch with friends, you had to call them on the phone or send them letters. Ten years ago you might use email. Today, many people prefer Facebook posts and texting.

YouTube. There was no YouTube either; it began life in 2005. If you wanted to share home movies with other people, it was like film, only worse; video recorders were bulky and awkward, and videotape was expensive. Today most cell phones take decent digital video, and storage for it is cheap and compact; in mid 2011, people were uploading nearly 50 hours of video to YouTube every minute.

Twitter. Twitter was launched in the middle of 2006. Blogging was somewhat popular at that time (the word "blog", a contraction of "web log", first appeared in 1999), but Twitter introduced the idea of microblogging, and a mechanism for immediate rebroadcasting to very large audiences.

Speech processing. In 1990, it was barely possible for a machine to understand spoken language, and then only for a restricted vocabulary and under laboratory conditions. Today if you call some company to get service or register a complaint, it's hard to avoid "talking" to a speech-recognition system; getting to a real person is nearly impossible. This is not necessarily progress. Machine translation was in much the same situation; today, useful, if not perfect, translations are freely available from any of nearly sixty languages into any of the others. Speech synthesis—speech from text—is routine, though readily distinguishable from a real voice.

Shopping. Twenty years ago there were no online stores; the closest parallel might be shopping from paper catalogs sent by retailers like Sears or L.L.Bean. If you wanted to compare features or look for the best price, you might be able to use the advertising flyers that arrived at your physical mailbox, but more likely you'd have to visit stores in person to do comparisons. Product reviews generally appeared

only in magazines like *Consumer Reports*. Today search engines provide links to physical and online stores, along with prices and reviews, all accessible from a phone, sitting at home, or standing in a store where scanning a barcode provides the information.

Maps. In 1990, if you didn't know how to get somewhere, you'd have to find the address from a phone book, look on a paper map to see where it was, then navigate while driving and holding the map. If you got lost, you'd have to figure out for yourself where you were and how to get back on track. You'd have to guess about traffic, or hope to find a report on the radio. If you wanted to see what the route looked like from the air, you'd need to rent a plane! GPS navigation devices for cars appeared about ten years ago, solving many of the navigation problems, and today they are being supplanted by similar systems running on phones. And with street view maps, you can see pictures of your destination before you arrive.

The purpose of this book is to remove any vestige of magic, so you understand how such systems operate. How can pictures, music, and movies be sent around the world in no time at all? How does email work? How private is your email? Why is spam so easy to send and so hard to get rid of? Do cell phones really know where you are? How do iPhones and Android phones differ and why are they fundamentally exactly the same? By the end, you should have a pretty decent idea of how computer and communications systems work and how they affect you.

A handful of important underlying ideas, which I will discuss in much more detail in the rest of the book, cut across the hardware, software and communications organization mentioned above.

First is the *universal digital representation of information*. Complex and sophisticated mechanical systems like those for storing pictures and music have been replaced by a single uniform mechanism. This is possible because information is represented in digital form (as numbers) rather than in specialized forms (colored dyes embedded in plastic film or magnetic patterns on audio tape, for example). Printed mail is replaced by digital mail. Paper maps are replaced by electronic ones. Different analog representations of information are replaced by a single digital representation.

Second is the *universal digital processor*. All of this information is being processed by a single device, the digital computer. Digital computers that process the uniform digital representation of information have replaced complicated physical machines that process specialized analog representations. I've mostly avoided the word "computer" so far, but in fact a cell phone is a computer of considerable sophistication and with as much computing power as a laptop that you might have used half a dozen years ago. As we'll see, computers are all equally capable in what they can compute, differing only in how fast they operate and how much data they can store. It's inevitable that more and more of what might have been limited to desktop or laptop computers will find its way onto cell phones. If anything, this convergence process is accelerating.

Third is the *universal digital network*. The Internet connects the digital computers that process the digital representation; it connects computers and cell phones to mail, search, social networks, shopping, banking, and everything else. You can exchange email with anyone, regardless of where they might be or how they choose to access their mail. You can search, comparison shop and purchase from your phone, laptop or tablet. Social networks keep you in touch with friends and family, again from phone or computer. Of course there is great deal of infrastructure that makes all these computing services work.

An immense amount of *digital data* is also continuously being collected and updated. Maps, aerial and street-level photographs of the whole world are freely available. Search engines tirelessly scan the entire Internet so they can answer queries efficiently. Millions of books are available in digital form. Social networks and sharing sites maintain enormous amounts of data for and about us. Online stores and services provide access to their wares while quietly recording everything we do when we visit them online, aided and abetted by search engines and social networks. For all of our online interactions, Internet service providers log the connections we make and perhaps more.

All of this is changing rapidly because digital systems are getting smaller, faster, and cheaper at an exponential rate: every year or two, things are twice as powerful for the same price, or cost half as much as they did. New phones with fancier features, better screens, and more interesting applications arrive continuously. New gadgets appear all the time; the most useful ones often find their functionality subsumed into phones over time. This is a natural by-product of digital technology, in which any technological development leads to improvement across the board for digital devices: if some change makes it possible to handle data cheaper, faster or in larger quantity, all devices will benefit. As a result, digital systems are pervasive, an integral part of our lives both visibly and behind the scenes.

This progress must surely seem wonderful, and indeed in most ways it is. But there are occasional clouds around the silver lining. One of the most obvious and perhaps the most worrying to individuals is the impact of technology on personal privacy. When I searched for a TV, then visited store web sites, all parties kept records of what I visited and what I clicked on. How did they know who I was? Well, I was using a cell phone, which identifies me uniquely. How did they know where I was so they could give me the list of stores? That's easy. Our cell phones report our locations *all the time*. My cell phone has GPS so the phone company had me located to within about five meters in my house; even without GPS, they know my location to within a hundred meters or so. The tracking that we permit without even thinking about it makes the monitoring in Orwell's *1984* look casual and superficial.

The records of what we do and where we do it are also likely to live forever. Digital storage is so cheap and the value of data is high enough that information is rarely discarded. If you post something embarrassing online or send mail that you subsequently regret, it's too late. Information about you can be combined from multiple

sources to create a detailed picture of your life, and is typically available to commercial and government interests without your knowledge or permission. It is likely to remain available and could surface to embarrass you at any time in the future.

The societal mechanisms that protect our privacy and our property have not kept up with the rapid advances in technology. Twenty years ago, I dealt with a local bank where I knew the tellers, they knew me and my banking patterns, and they could fix problems and would notice unusual events. That friendly local operation has long since been swallowed by a huge national bank where I often feel like merely a number on an account. A bank error, a blunder on my part, or a screwup by some third party could let someone on the far side of the world clean out my account, steal my identity, ruin my credit rating, and who knows what else; no one would notice right away. The universal network and its universal digital information have made us vulnerable to strangers to a degree never imagined a decade or two ago. At the same time, legitimate owners of digital data like music, movies and books are unable to protect their rights against widespread unauthorized and usually illegal copying.

This book is about understanding how all of these systems work and how they are changing our lives. Of necessity it's a snapshot, so you can be certain that five or ten years from now, today's systems will seem clunky and antiquated. Technological change is not an isolated event but an ongoing process—rapid, continuous, and accelerating. Fortunately, the basic ideas of digital systems will remain the same, so if you understand those, you'll understand tomorrow's systems too, and you'll be in a better position to deal with the issues and the opportunities that they present.

Part I

Hardware

Hardware is the solid, visible part of computing: devices and equipment that you can see and put your hands on. The history of computing devices is interesting, but I will only mention a little of it here. Some trends are worth noting, however, especially the exponential increase in how much circuitry and how many devices can be packed into a given amount of space, often for a fixed price. As digital equipment has become more powerful and cheaper, widely disparate mechanical systems have been superseded by much more uniform electronic ones.

Computing machinery has a long history, though most early computational devices were specialized, often for predicting astronomical events and positions. For example, there's a theory that Stonehenge was an astronomical observatory, though the theory remains unproven. The Antikythera mechanism, from about 100 BCE, is an astronomical computer of remarkable mechanical sophistication. Arithmetic devices like the abacus have been used for millennia, especially in Asia. The slide rule was invented in the early 1600s, not long after John Napier's description of logarithms. I used a slide rule as an undergraduate engineer in the 1960s but slide rules are now curiosities, and my painfully acquired slide rule expertise is useless.

The most relevant precursor to today's computers is the Jacquard loom, which was invented in France by Joseph Marie Jacquard around 1800. The Jacquard loom used rectangular cards with multiple rows of holes that specified weaving patterns. The Jacquard loom thus could be "programmed" to weave a wide variety of different patterns under the control of instructions that were provided on punched cards; changing the cards changed the pattern.

Arguably, computing in today's sense began in England in the mid 19th century with the work of Charles Babbage. Babbage was a scientist who was interested in navigation and astronomy, both of which required tables of numeric values from

which positions could be computed. Babbage spent much of his life trying to build computational devices that would mechanize the tedious and error-prone arithmetic calculations that were done by hand. For a variety of reasons, including alienating his financial backers, he never succeeded in his ambitions, but his designs were sound and modern implementations of his machines, using tools and materials from his time, can be seen in places like the Science Museum in London and the Computer History Museum in Mountain View, California.

Babbage encouraged a young woman, Augusta Ada Byron, the daughter of the poet George Byron and later Countess of Lovelace, in her interests in mathematics and his computational devices. Lovelace wrote detailed descriptions of how to use Babbage's Analytical Engine (the most advanced of his planned devices) for scientific computation and speculated that machines could do non-numeric computation as well, such as composing music. Ada Lovelace is often called the world's first programmer, and the Ada programming language is named in her honor.

Herman Hollerith, working with the US Census Bureau in the late 1800s, designed and built machines that could tabulate census information far more rapidly than could be done by hand. Using ideas from the Jacquard loom, Hollerith used holes punched in paper cards to encode census data in a form that could be processed by his machines; famously, the 1880 census had taken eight years to tabulate, but with Hollerith's punch cards and tabulating machines, the 1890 census took only one year. Hollerith founded a company that in 1924 became, through mergers and acquisitions, International Business Machines, which we know today as IBM.

Babbage's machines were complex mechanical assemblies of gears, wheels, levers and rods. The development of electronics in the 1900s made it possible to imagine computers that had no moving parts. The first significant one of these all-electronic machines was ENIAC (Electronic Numerical Integrator and Computer), which was built during the 1940s at the University of Pennsylvania in Philadelphia, by Presper Eckert and John Mauchly. ENIAC occupied a large room and required a large amount of electric power; it could do about 5,000 additions in a second. It was intended to be used for ballistics computations and the like, but it was not completed until 1946 when World War II was long over. (Parts of ENIAC are on display in the Moore School of Engineering at UPenn.)

Babbage saw clearly that a computing device could store its operating instructions and its data in the same form, but ENIAC did not store instructions in memory along with data; instead it was programmed by setting up connections through switches and re-cabling. The first truly stored-program computers were built in England, notably the EDSAC (Electronic Delay Storage Automatic Calculator) in 1949.

Early electronic computers used vacuum tubes as computing elements. Vacuum tubes are electronic devices roughly the size and shape of a cylindrical light bulb; they were expensive, fragile, bulky, and power hungry. With the invention of the transistor in 1947, and then of integrated circuits in 1958, the modern era of computing really began; devices based on this technology are one of the primary reasons why today's electronic systems continue to become smaller, cheaper and faster.

1

What's in a Computer?

Let's begin our discussion of hardware with an overview of what's inside a computer. This has two aspects: the logical or functional organization—what the pieces are, what they do and how they are connected—and the physical structure—what the pieces look like and how they are built. The goal of this chapter is to see what's inside, learn roughly what each piece does, and get some sense of what the myriad acronyms and numbers mean.

Think about your own computer. Many readers will have some kind of "PC," that is, a machine descended from the Personal Computer or PC that IBM first sold in 1981, running some version of the Windows operating system from Microsoft. Others will have an Apple Mac that runs a version of the Mac OS X operating system. More specialized devices like cell phones and tablets are also powerful computers. These computers look different, and when you use them they feel different as well, but underneath the skin, they are really the same. We'll talk about why.

There's a loose analogy to cars. Functionally, cars have been the same for over a hundred years. A car has an engine that uses some kind of fuel to make the engine run and the car move. It has a steering wheel that the driver (software?) uses to guide the car. There are places to store the fuel and places to store the passengers and their goods. Physically, however, cars have changed greatly over a century: they are made of different materials, and they are faster, safer, and much more reliable and comfortable. There's a world of difference between my first car, a well-used 1959 Volkswagen Beetle, and a Ferrari, but either one will carry me and my groceries home from the store or across the country, and in that sense they are functionally the same. (For the record, I have never even sat in a Ferrari, let alone owned one, so I'm speculating about whether there's room for the groceries.)

The same is true of computers. Logically, today's computers are pretty similar to those of the 1950s, but the physical differences are far beyond the kinds of changes that have occurred with the automobile. Today's computers are much smaller, cheaper, faster and more reliable than those of 50 years ago, and are literally a million times better in some properties. Such improvements are the fundamental reason why computers are so pervasive.

The distinction between the functional behavior of something and its physical properties—the difference between what it does and how it's built or works inside—is an important idea. For computers, the "how it's built" part changes at an amazing rate, as does how fast it runs, but the "what it does" part is quite stable. This distinction will come up repeatedly in what follows.

I sometimes do a survey in my class in the first lecture. How many have a PC? How many have a Mac? The ratio was fairly constant at 10 to 1 in favor of PCs in the first half of the 2000s, but changed rapidly over a few years, to the point where Macs now account for well over three quarters of the machines, though this is definitely not typical of the world at large, where PCs dominate by a wide margin.

Is the ratio unbalanced because one is superior to the other? If so, what changed so dramatically in such a short time? I ask which kind is better, and for objective criteria on which to base that opinion. What led you to your choice when you bought your computer?

Naturally, price is one answer. PCs tend to be cheaper, the result of fierce competition in a marketplace with myriad suppliers. A wider range of hardware add-ons, much more software, and more expertise are all readily available. (This is an example of what economists call a *network effect*: the more other people use something, the more useful it will be for you too, approximately in direct proportion to how many others there are.)

On the Mac side are perceived reliability, quality and esthetics, a sense that "things just work," and even a component of fear and loathing, which make many consumers willing to pay a premium.

The debate goes on, with neither side convincing the other, but it raises some good questions, and helps to get people thinking about what is different between the two kinds of computers, and what is really the same.

1.1 Logical Construction

If we were to draw an abstract picture of what's in a computer—its logical or functional architecture—it would look like the diagram below for both Mac and PC: a processor (the CPU), some primary memory (RAM), some secondary memory (a disk) and a variety of other components, all connected by a set of wires called a bus that transmits information between them.

The basic organization—with a processor, storage for instructions and data, and input and output devices—has been standard for over sixty years. It's often called the *von Neumann architecture*, after John von Neumann, who described it in a 1946 paper entitled "Preliminary discussion of the logical design of an electronic computing instrument" (co-authored with Arthur Burks and Herman Goldstine). Though there is still debate over whether von Neumann gets too much credit for work done by others, the paper is so clear and insightful that it is well worth reading even today. For example, the first sentence says "Inasmuch as the completed device will be a general-purpose computing machine it should contain certain main organs relating to arithmetic, memory-storage, control and connection with the human operator." Translated into today's terminology, the CPU provides arithmetic and control, the RAM and disk are memory storage, and the keyboard, mouse and display connect with the human operator.

1.1.1 CPU

The processor or *central processing unit* ("CPU") is the brains of the computer, if there could be said to be such a thing. The CPU does arithmetic, moves data around, and controls the operation of everything else. The CPU has a limited repertoire of basic operations that it can perform, but it performs them blazingly fast (billions per second). It can decide what operations to perform next based on the results of previous computations, so it is to a considerable degree in charge of its own destiny. We will spend more time on this in Chapter 3 because it's so important.

If you go to a store or shop online to buy a computer, you'll normally find most of these components mentioned, usually accompanied by mysterious acronyms and equally mysterious numbers. For example, you might see the CPU described as a "2.1 GHz Intel Core Duo." What's that? Intel makes the CPU and there are actually two CPUs in a single package; in this context, "core" has become a synonym for "processor."

"2.1 GHz" is the more interesting part. CPU speed is measured, at least approximately, in terms of the number of operations or instructions or parts thereof that it can do in a second. The CPU uses an internal clock, rather like a heartbeat or the ticking of a clock, to step through its basic operations, and one measure of speed is

the number of such ticks per second. One beat or tick per second is called one *hertz*, after the German engineer Heinrich Hertz, whose discovery of how to produce electromagnetic radiation in 1888 led directly to radio and other wireless systems; it's abbreviated Hz. Radio stations give their broadcast frequencies in megahertz (millions of hertz), like 102.3 MHz. Computers today typically run in the billions of hertz, or gigahertz, or GHz; my quite ordinary 2.1 GHz processor is zipping along at 2,100,000,000 ticks per second. The human heartbeat is about 1 Hz or almost 100,000 beats per day, which is around 30 million per year, so my CPU does in 1 second the number of beats my heart would do in about 70 years.

This is our first encounter with some of the numerical prefixes like mega and giga that are so common in computing. "Mega" is one million, or 10^6; "giga" is one billion, or 10^9 (and usually pronounced with a hard "g" as in "gig"). We'll see more units soon enough, and there is a complete table in the glossary.

1.1.2 RAM

The primary memory or *random access memory* ("RAM") stores information that is in active use by the processor and other parts of the computer; its contents can be changed by the CPU. The RAM stores not only the data that the CPU is currently working on, but also the instructions that tell the CPU what to do with the data. This is a crucially important point: by loading different instructions into memory, we can arrange that the CPU does a different computation. This makes the stored-program computer a general-purpose device: the same computer can run a word processor and a spreadsheet, surf the web, send and receive email, do my taxes, and show a movie, all by placing suitable instructions in the RAM. There is no way to overstate the importance of the stored-program idea.

RAM provides a place to store information while the computer is running. It stores the instructions of programs that are running, like Word, iTunes or a browser. It stores their data—the pictures on the screen, the documents being edited, the music that's currently playing. It also stores the instructions of the operating system—Windows, Mac OS X or something else—that operates behind the scenes to let you run multiple applications at the same time. We'll talk about operating systems in Chapter 6.

RAM is called "random access" because the CPU can access the information stored at any place within it as quickly as in any other; there's no speed penalty for accessing memory locations in any random order. Compare this to an old VCR tape, where to look at the final scenes of a movie, you have to fast forward (slowly!) over everything from the beginning; that's called "sequential access."

RAM is *volatile*, that is, its contents disappear if the power is turned off, and all this currently active information is lost. That's why it's prudent to save your work often, especially on a desktop machine, where tripping over the power cord can be a real disaster.

Your computer has a finite amount of RAM. Capacity is measured in *bytes*, where a byte is an amount of memory that's big enough to hold a single character

(like `w` or `@`), or a small number like 42, or a part of a larger value. Chapter 2 will show how information is represented in memory and other parts of a computer, since it's one of the fundamental issues in computing. But for now, you can think of the RAM as a large collection of identical little boxes, numbered from 1 up to a billion or two, each of which can hold a small amount of information.

What is the capacity? The laptop I'm using right now has 2 billion bytes or 2 gigabytes or GB of RAM, and many people would deem that a bit too small. The reason is that more RAM usually translates into faster computing, since there's never enough for all the programs that want to use it at the same time. If you want your computer to run faster, buying extra RAM is likely to be the best strategy.

1.1.3 Disks and secondary storage

The RAM has a large but limited capacity to store information, and its contents disappear when the power is turned off. Secondary storage holds information even when the power is turned off. The most common kind of secondary storage is the magnetic *disk*, sometimes called the hard disk or hard drive. The disk stores much more information than RAM and it's not volatile: information on the disk stays there indefinitely, power or not. So data, instructions, and everything else is stored on the disk for the long term and brought into RAM only transiently. Disk space is about 100 times cheaper per byte than RAM, but accessing information is much slower.

Disks store information by setting the direction of magnetization of tiny regions of magnetic material on rotating metallic surfaces. The whirring and clicking that you hear when a computer is doing something is the disk in action, moving a sensor to the right places on the surface. You can see the surface and sensor in this picture of a standard laptop disk; the platter is 2½ inches (6¼ cm) in diameter.

Some computers have "solid state disks," which use *flash memory* instead of rotating machinery. Flash memory is non-volatile; information is stored as electric charges in circuitry that needs no power to maintain the charge in individual circuit elements. Stored charges can be tested to see what their values are, and they can be erased and over-written with new values. Right now flash memory is more expensive per byte than conventional disk storage, but it's faster, lighter, more reliable and requires less power. It's used in cell phones, cameras, and the like, and it might well take over from mechanical disks.

A typical laptop disk today holds several hundred gigabytes, and external drives that can be plugged in to a USB socket have capacities in the terabyte (TB) range. ("Tera" is one trillion, or 10^{12}, another unit that you'll see more and more often.)

How big is a terabyte, or even a gigabyte for that matter? One byte holds one alphabetic character in the most common representation of text. *Pride and Prejudice*, about 250 pages on paper, has about 550,000 characters, so 1 GB could hold nearly 2,000 copies of it. More likely, I would store one copy and then include some music. MP3 music is about 1 MB per minute, so an MP3 version of one of my favorite CDs, *The Jane Austen Songbook*, is about 60 MB, and there would still be room for another 15 hours of music. The two-disk DVD of the 1995 BBC production of *Pride and Prejudice* with Jennifer Ehle and Colin Firth is less than 10 GB, so I could store it a hundred times over on a 1 TB disk.

A disk is a good example of the difference between logical structure and physical implementation. We see disk contents organized as a hierarchy of folders and files when we run a program like Explorer in Windows or Finder in Mac OS X. But the data is stored on rotating machinery, integrated circuits with no moving parts, or something else entirely. The particular kind of "disk" in a computer doesn't matter. Hardware in the disk itself and a significant part of the operating system called the file system create the organizational structure. We will return to this in Chapter 6.

The logical organization is so well matched to people (or, more likely, by now we're so completely used to it) that other devices provide the same organization even though they use completely different physical means to achieve it. For example, a CD-ROM or DVD stores information in a form that software makes look like a disk file system. So do USB devices, and digital cameras and other gadgets that use removable memory cards. Even the venerable floppy disk, now totally obsolete, looked the same at the logical level.

1.1.4 Et cetera

Myriad other devices serve special functions. Some, like mouse and keyboard or touch screen, allow users to provide input. Others, like the display, printers and speakers, are for output to users. Networking components like an Ethernet connection or wireless are for communicating with other computers.

The architecture drawing shows these as if they were all connected by a set of wires called a *bus*, a term borrowed from electrical engineering. In reality, there are

multiple buses inside a computer, with properties appropriate to their function (short, fast, but expensive between CPU and RAM; long, slow, but cheap to the headphone jack). Some of the buses make an appearance outside as well, like the ubiquitous Universal Serial Bus or USB for plugging devices into a computer.

I won't spend much time on other devices at the moment, though I'll come back once in a while to mention them in some specific context. For now, try to list the different devices that might accompany your computer or be attached to it: mice, keyboards, touch pads and touch screens, displays, printers, scanners, game controllers, music players, headphones, speakers, microphones, cameras, phones, connections to other computers. The list goes on and on. All of these have gone through the same evolution as processors, memory, and disks: the physical properties have changed rapidly, always towards more capabilities in a smaller package at a lower price.

It's also worth noting the degree to which some of these devices are converging into a single one. Cell phones now serve as watches, still and video cameras, music and movie players, game consoles, barcode readers, and navigators; they are even occasionally used to talk to other people. A cell phone has the same architecture as a laptop, though with differences due to size and power constraints. Phones don't have disks, but they do have non-volatile flash memory to store information—address books, apps, and the like—when the phone is turned off. They don't have many external devices either, though there's likely Bluetooth, a socket for headphones and an external microphone, and a USB connector.

Tablets like the iPad and its many competitors, which are taking off as I write, occupy another position in the space of possibilities; they too are computers with the same general architecture and similar components.

1.2 Physical Construction

In class, I pass around a variety of hardware devices (the legacy of several decades of dumpster diving), many with their innards exposed. So many things in computing are abstract that it's helpful to be able to see and touch disks, integrated circuit chips, the wafers on which they are manufactured, and so on. It's also interesting to see the evolution of some of these devices. For example, a laptop disk today is indistinguishable from one a decade old; the newer one is likely to have 10 or 100 times the capacity but the improvement is invisible.

On the other hand, there's a clear progression in the circuit boards that hold the components of a computer; there are fewer components today because more of the circuitry is inside them, the wiring is finer, and the connecting pins are more numerous and much closer together than they were twenty years ago. The picture below shows a desktop PC circuit board from the late 1990s. The components like CPU and RAM are mounted on or plugged into this board, and are connected by wires printed on the other side (and not visible here).

Electronic circuits in computers are built from large numbers of a handful of basic elements. The most important of these is the *logic gate*, which computes a single output value based on one or two input values; it uses input signals like voltage or current to control an output signal, also voltage or current. Given enough of such gates connected in the right way it's possible to perform any kind of computation. Charles Petzold's book *Code* is a nice introduction to this, and there are web sites with graphical animations that show how logic circuits can do arithmetic and other computations.

The fundamental circuit element today is the *transistor*, a device invented at Bell Labs in 1947 by John Bardeen, Walter Brattain and William Shockley, who shared the 1956 Nobel Prize in physics for their invention. In a computer, a transistor is basically a switch, a device that can turn a current on or off under the control of a voltage; with this simple foundation, arbitrarily complicated systems can be constructed.

Logic gates used to be built from discrete components—vacuum tubes the size of light bulbs in ENIAC and individual transistors about the size of a pencil eraser in the computers of the 1960s. This picture shows a replica of the first transistor (on the left), a vacuum tube, and a CPU chip in its package; the actual circuit part is at the center, and is about 1 cm square; the vacuum tube is about 4 inches (10 cm) long. A modern CPU chip of this size would contain hundred of millions of transistors.

Today logic gates are created on *integrated circuits* or ICs, often called *chips* or *microchips*. An IC has all the components and wiring of an electronic circuit on a single flat surface (a thin sheet of silicon) that is manufactured by a complex sequence of optical and chemical processes to produce a circuit that has no discrete pieces and no conventional wires. ICs are thus much smaller and far more reliable than discrete-component circuitry. Chips are fabricated en masse on circular *wafers* about 12 inches (30 cm) in diameter; the wafers are sliced into separate chips that are individually packaged. A typical chip is mounted in a larger package with dozens to hundreds of pins that connect it to the rest of the system. The picture below shows an integrated circuit in its package; the actual circuit part is at the center, and is about 1 cm square.

The fact that integrated circuits are based on silicon led to the nickname *Silicon Valley* for the region in California south of San Francisco where the integrated circuit

business first took off; it's now a shorthand for all the high-tech businesses in the area, and the inspiration for dozens of wannabes like Silicon Alley in New York and Silicon Fen in Cambridge, England.

ICs were independently invented around 1958 by Robert Noyce and Jack Kilby; Noyce died in 1990, but Kilby shared the 2000 Nobel Prize in physics for his role. Integrated circuits are central to digital electronics, but other technologies are used as well: magnetic storage for disks, lasers for CDs and DVDs, and lasers and optical fiber for networking. All of these have had dramatic improvements in size, capacity and cost over the past 50 years.

1.3 Moore's Law

In 1965, Gordon Moore, later the co-founder and long-time CEO of Intel, published a short article entitled "Cramming more components onto integrated circuits." He observed that as technology improved, the number of devices (transistors, basically) that could be manufactured on an integrated circuit of a particular size was doubling approximately every year, a rate that he later revised to every two years, and others have set at 18 months. Since the number of transistors is a rough surrogate for computing power, this meant that computing power was doubling every two years if not faster. In 20 years there would be ten doublings and the number of devices would have increased by a factor of 2^{10}, that is, about one thousand. In forty years, the factor is a million or more.

This exponential growth, known as *Moore's Law*, has been going on for fifty years, so integrated circuits now have well over a million times as many transistors as they did in 1965. This enormous quantitative growth is really a qualitative change. Graphs of Moore's Law in action, especially for processor chips, show the number of transistors rising from a couple of thousand in Intel's 8008 CPU in the early 1970s to hundreds of millions in the processors found in inexpensive consumer laptops today.

The basic number that characterizes the scale of circuitry is the size of individual features on an IC, for example, the width of a wire. This number has been shrinking steadily for many years. The first (and only) integrated circuit I designed in 1980 used 3.5 micron (3.5 micrometer) features. For many circuits today, that minimum feature size is 32 nanometers, that is, 32 billionths of a meter, and the next step will be 22 nanometers. ("Nano" is one billionth, or 10^{-9}, and nanometer is abbreviated nm.) For comparison, a sheet of paper or a human hair is about 100 micrometers (1/10th of a millimeter) thick. ("Micro" is one millionth, or 10^{-6}; "milli" is one thousandth, or 10^{-3}.)

The design and manufacture of integrated circuits is an extremely sophisticated business, and highly competitive. Manufacturing operations ("fabrication lines") are expensive as well; a new factory can easily cost several billion dollars. A company that can't keep up technically and financially is at a serious competitive disadvantage, and a country that doesn't have such resources must depend on others for its technology, potentially a serious strategic problem.

At some point Moore's Law will stop working. Its limits have often been predicted in the past, but ways around them have been found so far. We are getting to the point where there are only a handful of individual atoms in some circuits, however, and that's too small to control. CPU speeds no longer double every couple of years (in part because faster chips generate too much heat), but RAM capacity does; meanwhile, processors have more than one CPU on the chip.

It's striking to compare a personal computer of today to the original IBM PC, which appeared in 1981. The PC had a 4.77 MHz processor; the clock rate in a 2.3 GHz CPU is about 500 times faster. The PC had 64 kilobytes of RAM; today's 4 GB computers have about 60,000 times as much. ("Kilo" is one thousand, abbreviated "K".) It had at most 750 KB of floppy disk and no hard disk; today's machines are creeping up on a million times as much disk space. It had an 11-inch screen that could only display 24 rows of 80 green letters on a black background; I wrote a lot of this book on a 24-inch screen with 16 million colors. A PC with 64 KB of memory and a single 160 KB floppy disk cost $3,000 in 1981 dollars, which, thirty years later, might be more like $5,000 to $10,000; today a laptop with a 2 GHz processor, 4 GB of RAM, and 400 GB of disk costs only a few hundred dollars.

One of the great discoveries of 20th century computer science was that the logical or functional properties of today's digital computers, the original PC, and its physically much bigger but less powerful ancestors, are all the same—if we ignore issues like speed and storage capacity, they all can compute exactly the same things. I'll talk more about this in Chapter 3.

2

Bits, Bytes, and Representation of Information

In this chapter I'm going to discuss three fundamental ideas about how computers represent information.

First, *computers are digital processors*: they store and process information that comes in discrete chunks and takes on discrete values, basically just numbers, by contrast with analog information, which implies smoothly varying values.

Second, *computers represent information in bits*. A bit is a binary digit, that is, a number that is either 0 or 1. Everything inside the computer is represented with bits. The binary number system is used internally, not the familiar decimal numbers that people use.

Third, *groups of bits represent larger things*. Numbers, letters, words, names, sounds, pictures, movies, and the instructions that make up the programs that process them—all of these are represented as groups of bits.

You can safely skip the numeric details in this chapter, but the ideas are really important.

2.1 Analog versus Digital

First, let's draw the distinction between analog and digital. "Analog" comes from the same root as "analogous," and is meant to convey the idea of values that change smoothly as something else changes. Much of what we deal with in the real world is analog, like a water tap or the steering wheel of a car. If you want to turn the car a little bit, you turn the wheel a little bit; you can make as small an adjustment as you like. Compare this to the turn signal, which is on or off; there's no middle ground.

In an analog device, something (how much the car turns) varies smoothly and continuously in proportion to a change in something else (how much you turn the steering wheel). There are no discrete steps, and a small change in one thing implies a small change in another.

Digital systems deal with discrete values: there are only a limited number of possible values (the turn signal is either off or it's on in one direction or the other). A small change in something results either in no change or in a sudden change in something else, from one of its discrete values to another.

Think about a watch. "Analog" watches have an hour hand, a minute hand, and a second hand that goes around once a minute. Although modern watches are controlled by digital circuitry inside, the hour hand and the minute hand move smoothly as time passes, and all three hands pass through every possible position. A digital watch or a cell phone clock displays time only with numbers. Every second the display changes, every minute a new minute value appears, and there's never a fractional minute.

Think about a car speedometer. Most cars have analog speedometers, where a needle moves smoothly up and down in direct proportion to the car's speed. The transitions from one speed to another are smooth and there's no break. By contrast, car GPS navigators display speed numerically, to the nearest mile or kilometer per hour. Go a tiny bit faster and the speed goes from 65 to 66; go a tiny bit slower and it drops back to 65. There's never a display of 65.5.

Think about a thermometer. The kind with a column of red liquid (colored alcohol, usually) or mercury is analog: the liquid expands or contracts in direct proportion to temperature changes, so a small change in temperature will produce a similarly small change in the height of the column. But the sign that flashes `37` outside a building is digital: the display is numeric, and it shows 37 for all temperatures between 36.5 and 37.4.

This can lead to some odd situations. Some years ago, I was listening to my car radio on a US highway within reception distance of Canada, which uses the metric system. The announcer, trying to be helpful to everyone in his audience, said "the Fahrenheit temperature has gone up one degree in the last hour; the Celsius temperature is unchanged."

Why digital instead of analog? After all, our world is analog, and analog devices like watches and speedometers are easier to interpret at a glance. Nevertheless, much modern technology is digital; in many ways, that's the story in this book. Data from the external world—sound, images, movement, temperature, and everything else—is converted as soon as possible to a digital form on the input side, and is converted back to analog form as late as possible on the output side. The reason is that digital data is easy to work with. It can be stored, transported, and processed in many ways regardless of its original source, which is not true for analog information. As we'll see in Chapter 9, digital information can be compressed by squeezing out redundant or unimportant information. It can be encrypted for security and privacy, it can be merged with other data, it can be copied without error, it can be shipped anywhere

via the Internet, and it can be stored in an endless variety of devices. Most of this is simply not possible with analog information.

Digital systems have another advantage over analog: they are much more easily extended. For instance, in an analog planetarium, adding a newly-discovered star requires painstaking optical work by an expert; in a digital planetarium, it requires adding a line to a data file. My digital watch continuously displays elapsed times to a hundredth of a second; adding that capability to an analog watch would be challenging. On the other hand, analog systems sometimes have the advantage: old media like clay tablets, stone carvings, parchment, books and photographs have all stood the test of time in a way that digital forms may not.

2.2 Analog-Digital Conversion

How do we convert analog information into digital form? Let's look at some of the basic examples, beginning with pictures and music, which between them illustrate the important ideas.

Conversion of images to digital form is probably the easiest way to visualize the process. Suppose we take a picture of the family cat:

A film camera creates an image by exposing a light-sensitive area of film to light from the object being photographed; different areas of the film receive different amounts of light of different colors, and that affects dyes in the film. When the film is developed and printed, the colors are displayed as varying amounts of colored dyes.

In a digital camera, the lens focuses the image onto a rectangular array of tiny light-sensitive detectors that lie behind red, green and blue filters. Each detector stores an amount of electric charge that is proportional to the amount of light that falls on it. These charges are converted into numeric values and the digital representation of the picture is the sequence of resulting numbers that represent the light intensities. The smaller and more numerous the detectors and the more precisely the charges are measured, the more accurately the digitized image will capture the original.

Each element of the sensor array is a set of detectors that measure the amount of red, green and blue light; each group is called a *pixel*, for picture element. If the image is 3,000 by 2,000 pixels, that's six million picture elements, or six megapixels, small for today's digital cameras. The color of a pixel is usually represented by three values, the intensities of red, green and blue it contains, so a 6-megapixel image has 18 million color values altogether. Screens display images on arrays of triplets of tiny red, green and blue lights whose brightness levels are set by the corresponding levels in the pixel; if you look at the screen of a phone or computer with a magnifying glass, you can easily see the individual colored spots.

The second analog-to-digital conversion example is sound, especially music. Digital music is a nice example because it was one of the first areas where the properties of digital information began to have major social, economic and legal implications. Unlike records or audio cassettes, digital music can be copied perfectly on any home computer as many times as desired, for free, and the perfect copies can be sent anywhere in the world without error via the Internet, also for free. The recorded music industry saw this as a serious threat, and began a campaign of legal and political action to try to suppress copying. The war is far from over; new skirmishes happen every day and battles are regularly fought in courts and political arenas.

What is sound? A sound source creates fluctuations in air pressure by vibration or other rapid motion, and our ears convert the pressure changes into neural activity that our brains interpret as "sound." In the 1870s, Thomas Edison built a device that he called a "phonograph," which converted the fluctuations into an analogous pattern of grooves in a wax cylinder that could be used later to recreate the air pressure fluctuations. Converting a sound into a pattern of grooves was "recording"; converting from the pattern to fluctuations in air pressure was "playback." Edison's invention was rapidly refined, and by the 1940s had evolved into the long-playing record or LP, which is still in use today, though not in large quantities. A microphone converts changes in sound pressure over time into a changing value that is recorded and then used to press into a vinyl disk a long spiral groove with a pattern that matches the pressure changes. When the LP is played, a fine needle follows the fluctuations of the groove and its motion is converted into a fluctuating electrical current that is amplified and used to drive a loudspeaker or an earphone, devices that create sound by vibrating a surface.

It's easy to visualize sound by plotting how air pressure changes with time. The pressure can be represented in any number of physical ways; here we'll assume it's voltage in an electronic circuit, though it could be current, or brightness of a light, or a purely mechanical system as it was in Edison's original phonograph.

The height of the sound pressure wave is the sound intensity or loudness, and the horizontal dimension is time: the number of waves per second is the pitch or frequency.

Suppose we measure the height of the curve—the voltage, let's say—at regular intervals, as indicated by the vertical lines in the diagram below

The measurements provide a sequence of numeric values that approximate the curve; the more often and the more precisely we measure, the more accurate the approximation will be. The resulting sequence of numbers is a digital representation of the waveform that can be stored, copied, manipulated and shipped elsewhere. We can do playback with a device that converts the numeric values into a matching pattern of voltage or current to drive a speaker or an earphone. Converting from the waveform to numbers is analog to digital conversion and the device is called an A/D converter; the other direction is of course digital to analog conversion, or D/A. The conversion is not perfect; something is lost in each direction. But for most purposes the loss is imperceptible (though audiophiles claim that CD sound quality is not as good as LPs).

The audio compact disk or CD appeared around 1982, and was the first consumer example of digital sound; you've undoubtedly seen this familiar logo:

Rather than the analog groove of an LP record, a CD records *numbers* in a long spiral track on one side of the disk. The surface at each point along the track either is smooth or has a tiny pit. These pitted or smooth spots are used to encode the numeric values of the wave; each spot is a single bit, and a sequence of bits represents the numeric value in a binary encoding, as we will discuss in the next section. As the disk rotates, a laser shines on the track and a photoelectric sensor detects how much light is reflected from each spot. If there's not much light, there was a pit; if there's a lot of reflected light, there was no pit. The standard encoding for CDs uses 44,100 samples per second; each sample is two values (left side, right side, for stereo) of amplitudes measured to an accuracy of one part in 65,536 (which is 2^{16}, not coincidentally). The spots are so small that they can only be seen with a microscope; the surface of a CD has over 6 billion of them. (DVDs are similar; smaller spots and a higher frequency laser allow them to store nearly 5 GB, compared to

about 700 MB for a CD.)

The audio CD almost drove the LP out of existence because it was so much better in most ways—not much bothered by dirt, not subject to wear, not fragile, and definitely compact. As I write, LPs are enjoying a modest renaissance, while CDs of popular music are in decline because it's easier and cheaper to get music over the Internet. In time CDs will probably seem as antique as LPs, though I personally am happy that my music collection is entirely on CDs. I own them outright and they will outlast me. CDs also had a second career as a storage and distribution medium for software and data, but that has been superseded by DVDs, which in turn will likely be replaced by downloading.

Sound and images are often compressed, since they contain more detail than humans can actually perceive. For music, this is typically done with a compression technique called MP3, which reduces the size by a factor of about 10 without much if any perceptible degradation. For images, the most common compression technique is called JPEG (named after the organization Joint Photographic Experts Group that defined it); it also shrinks an image by a factor of 10 or more. Compression is an example of the kind of processing that can be done on digital information but would be extremely difficult if not impossible with analog. We'll discuss it further in Chapter 9.

What about movies? In the 1870s, the photographer Eadweard Muybridge showed how to create the illusion of motion by displaying a sequence of still images one after another in rapid succession. Today, a motion picture displays images at 24 frames per second and TV at 25 to 30 frames per second, which is fast enough that the human eye perceives the sequence as motion. We can create a digital representation of a movie by combining (and synchronizing) the sound and picture components. Compression can be used to reduce the amount of space required; that leads to standard movie representations like MPEG ("Moving Picture Experts Group"). In practice, video representation is more complicated than audio, in part because it's intrinsically harder, but also because much of it is biased by television, which for most of its lifetime has been analog. Analog television is being phased out in most parts of the world; in the USA, broadcast television switched to digital in 2009.

Some other kinds of information are easy to represent in digital form, since no transformation is needed beyond agreement on what the representation is. Consider ordinary text, like the letters, numbers and punctuation in this book. We could assign a unique number to each different letter—`A` is 1, `B` is 2, and so on—and that would be a fine digital representation. In fact, that's exactly what is done, except that in the standard representation, `A` through `Z` are 65 through 90, `a` through `z` are 97 through 122, the digits `0` through `9` are 48 through 57, and other characters like punctuation take other values. This representation is called ASCII, the American Standard Code for Information Interchange.

Here is part of ASCII; I've omitted the first four rows, which contain tab, backspace and other non-printing characters.

32	space	33	!	34	"	35	#	36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+	44	,	45	–	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	‘	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

There are multiple character-set standards in different regions, but the world is converging on a single standard called Unicode, which specifies a unique numeric value for every character in every language. This is a big collection, since humans have been endlessly inventive but rarely systematic in their creation of writing systems. Unicode has well over 100,000 characters at this point and the number rises steadily. As might be imagined, Asian character sets like Chinese account for a large part of Unicode but by no means all. The `unicode.org` site has charts of all the characters; it’s fascinating and well worth a detour.

The bottom line: a digital representation can represent all of these kinds of information and indeed anything that can be converted into numbers. Because it is just numbers, it can be processed by digital computers, and as we will see in Chapter 9 it can be copied to any other computer by the universal digital network, the Internet.

2.3 Bits, Bytes, and Binary

> “There are only 10 kinds of people in the world—those
> who understand binary numbers and those who don’t.”

Digital systems represent information of all types as numbers, but perhaps surprisingly at first glance, they do not use the familiar base ten (decimal) number system internally. Instead they use binary numbers, that is, numbers in base two.

Although everyone is more or less comfortable with arithmetic, in my experience understanding of what a number means is sometimes shaky, at least when it comes to drawing the analogy between base ten (totally familiar) and base two (not familiar to most). I’ll try to remedy this in this section, but if things seem confused or confusing, keep repeating to yourself, “It’s just like ordinary numbers, but with two instead of ten.”

2.3.1 Bits

The most elemental way to represent digital information is with bits. The word *bit* is a contraction of *binary digit* that was coined by the statistician John Tukey in the mid 1940s. (Tukey also coined the word *software* in 1958.) It is said that Edward Teller, best known as the father of the hydrogen bomb, preferred “bigit,” a term that

mercifully didn't catch on. The word "binary" suggests something with two values (the prefix "bi" means two, of course), and that is indeed the case: a bit is a digit that takes on either the value 0 or the value 1, with no other possibilities. This can be contrasted with the 10 possible values of the decimal digits 0 through 9.

With a single bit, we can encode or represent any choice that involves selecting one of two values. Such binary choices abound: on/off, true/false, yes/no, high/low, in/out, up/down, left/right, north/south, east/west, male/female, and so on. A single bit is sufficient to identify which one of the pair was selected. For example, we could assign 0 to off and 1 to on, or vice versa, so long as everyone agrees on which value represents which state. Here's a picture of a power switch on my printer; by universal convention 0 is off and 1 is on:

A single bit is enough to represent on/off, true/false and similar binary choices but we often need to deal with more choices or to represent more complicated things. For that, we use a group of bits, assigning meanings to the different possible combinations of 0's and 1's. For example we could use 2 bits to represent the four years of college—freshman (00), sophomore (01), junior (10) and senior (11). If there were one more category, say graduate students, two bits is not sufficient because there are now five different possibilities but only four different combinations of two bits are available. Three bits would be enough, however, and in fact could represent as many as eight different kinds of things, so we could also include faculty, staff and postdocs. The combinations would be 000, 001, 010, 011, 100, 101, 110, and 111.

There's a pattern that relates the number of bits to the number of items that can be labeled with that many bits. The relationship is simple: if there are N bits, the number of different bit patterns is 2^N, that is, $2 \times 2 \times \cdots \times 2$ (N times). Thus:

number of bits	number of values	number of bits	number of values
1	2	6	64
2	4	7	128
3	8	8	256
4	16	9	512
5	32	10	1,024

This is directly analogous to the situation with decimal digits: with N decimal digits, the number of different digit patterns (which we call "numbers") is 10^N:

number of digits	number of values	number of digits	number of values
1	10	6	1,000,000
2	100	7	10,000,000
3	1,000	8	100,000,000
4	10,000	9	1,000,000,000
5	100,000	10	10,000,000,000

2.3.2 Powers of two and powers of ten

Since everything in a computer is handled in binary, properties like sizes and capacities tend to be expressed in powers of two. If there are N bits, there are 2^N possible values so it's handy to know the powers of two up to some value, say 10. Once the numbers get larger, they are certainly not worth memorizing. Fortunately, there's a shortcut that gives a good approximation: some powers of two are close to powers of ten, in an orderly way that's easy to remember:

2^{10} = 1,024	10^3 = 1,000 (kilo)
2^{20} = 1,048,576	10^6 = 1,000,000 (mega)
2^{30} = 1,073,741,824	10^9 = 1,000,000,000 (giga)
2^{40} = 1,099,511,627,776	10^{12} = 1,000,000,000,000 (tera)
2^{50} = 1,125,899,906,842,624	10^{15} = 1,000,000,000,000,000 (peta)
...	

(The table includes one more size prefix, "peta" or 10^{15}; it's pronounced like "pet", not "Pete". The glossary at the end of the book has a larger table with still more units.) The approximation gets worse as the numbers get bigger, but it's only 12.6 percent too high at 10^{15}, so it's useful over a very wide range. You'll find that people often blur the distinction between the power of two and the power of ten (sometimes in a direction that favors some point that they are trying to make), so "kilo" or "1K" might mean one thousand, or it might mean 2^{10} or 1,024. This is usually a minor difference, so the powers of two and ten are a good way to do mental arithmetic on big numbers involving bits.

It probably seems picky, but the distinction sometimes matters. For instance, Seagate, a disk drive manufacturer, was involved in a California class action lawsuit based on this distinction. "Plaintiff claims that Seagate's use of the decimal definition of the storage capacity term 'gigabyte' (or GB), whereby 1 GB = 10^9 (1 billion) bytes, misleads consumers because computer operating systems report hard drive capacity using a binary definition of GB, whereby 1 GB = 2^{30} (1,073,741,824) bytes, a difference of approximately seven percent." I personally doubt that many consumers were seriously misled, but Seagate settled rather than fighting it out.

2.3.3 Binary numbers

A sequence of bits can represent a numeric value if the digits are interpreted in the usual place-value sense, but using base 2 instead of base 10. Ten digits, 0 through 9, are enough to count or assign labels to up to ten items. If we need to count beyond ten, we must use more digits; with two decimal digits, we can count or label up to 100 things, with labels 00 through 99. For more than 100 items, it's on to three digits, which gives a range of 1000, from 000 to 999. (Conventionally we don't write the leading zeroes for ordinary numbers, but they are there implicitly. In daily life we also start counting at 1, not zero.)

Decimal numbers are shorthands for sums of powers of ten; for example, 1867 is $1\times10^3 + 8\times10^2 + 6\times10^1 + 7\times10^0$, which is $1\times1000 + 8\times100 + 6\times10 + 7\times1$, which is 1000+800+60+7. In elementary school, you might have called these the 1's column, the 10's column, the 100's column and so on. This is so familiar that we don't think about it at all.

Binary numbers are the same except that the base is two instead of ten, and the only digits involved are 0 and 1. A binary number like 11101 is interpreted as $1\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$, which we express in base ten as 16+8+4+0+1 or 29.

The fact that sequences of bits can be interpreted as numbers means that there is a natural pattern for assigning binary labels to items: put them in numeric order. We saw that above with the labels for freshmen, sophomores, etc.: 00, 01, 10, 11, which in base ten have the numeric values 0, 1, 2 and 3. The next sequence would be 000, 001, 010, 011, 100, 101, 110, 111, with numeric values from 0 to 7.

Here's an exercise to confirm your understanding. We're all familiar with counting up to ten on our fingers, but how high can you count on your fingers if you use binary numbers, with each finger and thumb (a digit!) representing a binary digit? What's the range of values? If the binary representation of 132 reminds you of something, you've got the idea.

As you can see, it's easy to convert binary to decimal: just add up the powers of two for which the corresponding bit of the number is 1. Converting decimal to binary is harder but not much. Repeatedly divide the decimal number by two. Write down the remainder, which will be either zero or one, and use the quotient as the next value to divide. Keep going until the original number has been divided down to zero. The sequence of remainders is the binary number, except it's in reverse order, so flip it end for end.

As an example, convert 1867 to binary:

number	quotient	remainder
1867	933	**1**
933	466	**1**
466	233	**0**
233	116	**1**
116	58	**0**

58	29	**0**
29	14	**1**
14	7	**0**
7	3	**1**
3	1	**1**
1	0	**1**

Reading the bits off backwards, we have 111 0100 1011, which we can check by adding up the powers of two: 1024+512+256+64+8+2+1 = 1867.

Each step of this procedure produces the least significant (rightmost) bit of the number that remains. It's analogous to the process you would go through to convert a large number that represents a time in seconds into days, hours, minutes and seconds: divide by 60 to get minutes (and the remainder is the number of seconds); divide the result by 60 to get hours (and the remainder is the number of minutes); divide the result by 24 to get days (and the remainder is the number of hours). The difference is that time conversions do not use a single base but mix 60 and 24.

Binary arithmetic is really easy. Since there are only two digits to work with, the addition and multiplication tables only have two rows and two columns.

+	0	1
0	0	1
1	1	0 and carry 1

×	0	1
0	0	0
1	0	1

It's unlikely that you'll ever need to do binary arithmetic yourself, but the simplicity of these tables shows why computer circuitry for arithmetic is much simpler than it would be for decimal arithmetic.

2.3.4 Bytes

In all modern computers, the basic unit of processing and memory organization is 8 bits that are treated as a unit. Eight bits is called a *byte*, a word coined in 1956 by Werner Buchholz, a computer architect at IBM. A single byte can encode 256 distinct values (2^8, all the different combinations of 8 zeroes and ones), which could be an integer value between 0 and 255, or a single character in the ASCII character set, or something else. Often, a particular byte is part of a larger group that represents something bigger or more complicated. Two bytes together provides 16 bits, enough to represent a number between 0 and $2^{16} - 1$, or 65,535. They could also represent a character in the Unicode character set, perhaps one or the other of

東京

which are the two characters of "Tokyo"; each character is two bytes. Four bytes is 32 bits, which could be "Tokyo" or it could be a number up to $2^{32} - 1$, around 4.3 billion. There's no limit to what a set of bytes might represent, though the CPU itself has a modest set of specific groupings (like integers of various sizes) and instructions

for processing such groups.

If we want to write down the numeric value represented by one or more bytes, we could express it in decimal, which is convenient for a human reader if it's really numeric. We could write it in binary to see the individual bits, which matters if different bits encode different kinds of information. Binary is bulky, however, more than three times longer than the decimal form, so an alternative notation called *hexadecimal* is commonly used. Hexadecimal is base 16, so it has 16 digits (like decimal has 10 digits and binary has 2); the digits are `0`, `1`, ..., `9`, `A`, `B`, `C`, `D`, `E`, and `F`. Each hex digit represents 4 bits, with the obvious numeric values: hex `0` is binary `0000`, and so on up through hex `9` is binary `1001`. Then hex `A` is the next value, binary `1010` (decimal 10), hex `B` is binary `1011` (decimal 11), and so on through hex `F`, which is binary `1111` (decimal 15).

Unless you're a programmer, there are only a handful of places where you might see hexadecimal. One is colors on a web page. As mentioned earlier, the most common representation of colors in a computer uses three bytes for each pixel, one for the amount of red, one for the amount of green, and one for the amount of blue; this is called RGB encoding. Each of those components is stored in a single byte, so there are 256 possible amounts of red; for each of those there are 256 possible amounts of green, and then for each of those there are 256 possible blue values. Altogether that's $256 \times 256 \times 256$ possible colors, which sounds like a lot. We can use powers of two and ten to get a quick estimate of how many. It's $2^8 \times 2^8 \times 2^8$, which is 2^{24}, or $2^4 \times 2^{20}$, or about 16×10^6, or 16 million. You've probably seen this number used to describe computer displays ("More than 16 million colors!").

An intense red pixel would be represented as `FF0000`, that is, maximum red, no green, no blue, while a bright but not intense blue, like the color of links on many web pages, would be `0000CC`. Yellow is red plus green, so `FFFF00` would be the brightest possible yellow. Shades of gray have equal amounts of red, green and blue, so a medium gray pixel would be `808080`, that is, the same amount of red, green and blue. Black and white are `000000` and `FFFFFF` respectively.

The Unicode code tables use hex to identify characters:

東京

is `6771 4EAC`. You will also see hex in Ethernet addresses, which we'll talk about in Chapter 8, and representing special characters in URLs, one of the topics of Chapter 10.

You'll sometimes see the phrase "64-bit" in contexts like advertising for computers ("Windows 7 Home Premium 64-bit Edition"). What does this mean? Computers manipulate data internally in chunks of different sizes; the chunks include numbers (for which 32 bits and 64 bits are convenient) and addresses, that is, locations of information in RAM. It is this latter property that is being referred to. Twenty-five or more years ago there was a transition from 16-bit addresses to 32-bit addresses (which are big enough to access up to 4 GB of RAM), and now it's moving from 32 to 64. I won't try to predict when the transition from 64 to 128 will occur, but we

should be safe for quite a while.

The critical thing to remember in all of this discussion of bits and bytes is that the meaning of a group of bits depends on their context; you can't tell by looking at them what they mean. A single byte could be a single bit representing male or female and 7 unused bits, or it could be storing a small integer or an ASCII character like `#`, or it could be part of a character in another writing system, part of a larger number with 2 or 4 or 8 bytes, or part of a picture or a piece of music, or part of an instruction for the CPU to execute.

Indeed, one program's instructions are another program's data. When you download a new program from the net or install it from a CD-ROM or DVD, it's just data, bits to be copied blindly. But when you run the program, its bits are treated as instructions and data as they are processed by the CPU.

2.4 Summary

Why binary instead of decimal? It's much easier to make physical devices that only have two states, like on and off, than devices that have ten states. This comparative simplicity is exploited in a wide variety of technologies: current (flowing or not), voltage (high or low), electric charge (present or not), magnetism (north or south), light (bright or dim), reflectance (shiny or dull). John von Neumann clearly realized this when in 1946 he said "Our fundamental unit of memory is naturally adapted to the binary system since we do not attempt to measure gradations of charge."

Why should anyone know or care about binary numbers? That's a fair question. At least for my course, part of the reason is that working with numbers in an unfamiliar base is an example of quantitative reasoning that might even improve understanding of how numbers work in good old base ten. Beyond that, it's also important because the number of bits is usually related in some way to how much space, time or complexity is involved. And fundamentally, computers are worth understanding, and binary is central to their operation.

Binary shows up in real-world settings unrelated to computing as well, probably because doubling or halving sizes, lengths, and so on is a natural operation for people. For instance, Don Knuth's *The Art of Computer Programming* describes English wine container units in the 1300s that run over 13 binary orders of magnitude: 2 gills is one chopin, 2 chopins is one pint, 2 pints is one quart, and so on until 2 barrels is one hogshead, 2 hogsheads is one pipe, and 2 pipes is one tun. About half of those units are still in common use in the English system of liquid measures, though charming words like firkin and kilderkin (two firkins, or half a barrel) are now rarely seen.

3

Inside the CPU

In Chapter 1, I said that the central processing unit or CPU was "the brain" of a computer, though with a caveat that the term didn't really make sense. It's now time to take a more detailed look at the CPU, since it is the most important component of a computer and the one whose properties are the most significant for the rest of the book.

How does the central processor work? What does it process and how? To a first approximation, the CPU has a small repertoire of basic operations that it can perform. It can do arithmetic—add, subtract, multiply and divide numbers, like a calculator. It can fetch data from the RAM to operate on, and it can store results back into the RAM, like the memory operations on many calculators. The CPU also controls the rest of the computer, arranging that data from peripheral devices like the mouse and keyboard is acted on, that information is displayed on the screen, and that all the other gadgets connected to the computer are controlled and coordinated.

Most important, it can make decisions, albeit of a simple kind: it can compare numbers (is this number bigger than that number?) or other kinds of data (is this piece of information the same as that one?), and it can decide what to do next based on the outcome. This is the most important thing of all, because it means that although the CPU can't really do a lot more than a calculator can, it can do its job without human intervention. As von Neumann said, "It is intended that the machine be fully automatic in character, i.e., independent of the human operator after the computation starts."

Because the CPU can decide what to do next based on data that it is processing, it is able to run the whole system on its own. Although its repertoire is not large or complicated, the CPU can perform billions of operations every second so it can do some exceedingly complicated processing.

3.1 The Toy Computer

Let me explain how a CPU operates by describing a machine that doesn't exist. It's a made-up or "pretend" computer that uses the same ideas as a real computer but is much simpler. Since it exists only on paper, I can design it in any way that might help explain how real computers work. I can also create a program for a real computer that will simulate my paper design, so I can write programs for the pretend machine and see how they run.

I've taken to calling this made-up machine the "Toy" computer, since it's not real but it has many of the properties of the real thing; it's actually about the level of minicomputers of the late 1960s, and somewhat similar to the example in the von Neumann paper. The Toy has RAM for storing instructions and data, and it has one additional storage area called the *accumulator* that has enough capacity to hold a single number. The accumulator is analogous to the display in a calculator, which holds the number most recently entered by the user or the most recent computational result. The Toy has a repertoire of only about ten instructions, basic operations like those described above. Here are a few:

`GET`	get a number from the keyboard into the accumulator
`PRINT`	print the contents of the accumulator
`STORE M`	store a copy of the accumulator contents into memory location `M` (accumulator contents do not change)
`LOAD Val`	load the accumulator with value `Val`, which is a number or memory location (`Val` is unchanged)
`ADD Val`	add `Val` to contents of accumulator (`Val` is unchanged)
`STOP`	stop running

As an over-simplification, each RAM location holds one number or one instruction (I'm skipping over how those are represented), so a program consists of some mixture of instructions and data items stored in the RAM. In operation, the CPU starts at the first RAM location and repeats a simple cycle:

Fetch:	get the next instruction from RAM
Decode:	figure out what that instruction does
Execute:	do the instruction
	go back to *Fetch*

To create a program for the Toy, we have to write a sequence of instructions that will do the desired task, put them into the RAM, and tell the CPU to start executing them. As an example, suppose that the RAM contains exactly these instructions (which would actually be stored as their numerical equivalents):

```
GET
PRINT
STOP
```

When this sequence is executed, the first instruction will ask the user for a number, the second will print that number, and the third will tell the processor to stop. This is

pretty boring but it is enough to see what a program looks like. Given a real Toy machine, the program could even be run. Fortunately, there are working Toys; here's what one of them looks like in operation:

When you push RUN, this window pops up:

When you type a number and push OK the result is this:

As promised, the program asked for an input number, printed it, and stopped.

The next program is slightly more complicated and adds a new idea: storing a value in the RAM and then retrieving it. The program gets a number into the accumulator, stores it in memory, gets another number into the accumulator (which overwrites the first number), adds the first number to it (taken from RAM, where we carefully stored it), prints the sum of the two numbers, and then stops.

```
        GET          get first number into the accumulator
        STORE Mem    store it in a RAM location called Mem
        GET          get second number into the accumulator
        ADD Mem      add first number to it
        PRINT        print resulting sum
        STOP
    Mem -            a place in RAM to hold the first input number
```

The CPU starts at the beginning of the program and fetches instructions one at a time. It executes each instruction, then goes on to the next one.

The only tricky bit is that we need to set aside a place in the RAM to hold a data value, the first number that will be read. We can't leave it in the accumulator, since the second `GET` instruction will overwrite it. Since it's data, not an instruction, we have to put it someplace in the RAM where it won't be interpreted as an instruction. If we put it at the end of the program, after all of the instructions, the CPU will never try to interpret the data value as an instruction because it will `STOP` before it gets there.

We also need a way to refer to that location when it's needed by instructions of the program. One possibility would be to observe that the data will be in the 7th memory location (following the six instructions) but counting instructions is a nuisance and the location might require updating if the program were changed. The convention is to give the data item a name, and as we'll see in Chapter 5, a program can do the clerical task of keeping track of where the data is actually located in the RAM, replacing the name by the proper location. The name is arbitrary, though it's good practice to use a name that suggests the purpose or meaning of the data or instruction being named. Hence `Mem`, though perhaps `Firstnum` would have been better.

How could this program be extended so it will add three numbers? It would be easy enough to add another sequence of `STORE`, `GET` and `ADD` instructions (there are two places they could be inserted), but that certainly won't scale up to adding a thousand numbers, nor would it work if we don't know in advance how many numbers there are.

The solution is to add a new kind of instruction to the CPU's repertoire that lets it re-use sequences of instructions. The `GOTO` instruction, sometimes called "branch" or "jump," tells the CPU to take its next instruction not from the next one in sequence but from the location specified in the instruction itself.

With a `GOTO` instruction we can make the CPU return to an earlier part of the program and repeat instructions. One simple example is a program that prints each number as it is entered; that's the essence of a data-copying program and it shows what the `GOTO` instruction does. I labeled the first instruction of the program `Top` (an arbitrary name that suggests its role), and the last instruction causes the CPU to go back to that first instruction.

```
Top   GET          get a number into the accumulator
      PRINT        print it
      GOTO Top     go back to Top to get another number
```

This gets us part way: we can re-use instructions, but there's no way to stop the loop. For that we need another kind of instruction, something to test a condition and decide what to do next rather than pressing on blindly. That kind of instruction is called a "conditional branch" instruction or conditional jump or conditional goto. Details vary among real machines, but one possibility is an instruction that tests whether a value is zero, and jumps somewhere else if it is. Let's add an `IFZERO` to the Toy's repertoire. The `IFZERO` instruction will branch to a specified location if the accumulator value is zero; otherwise execution will continue at the next sequential instruction. `IFZERO` is used in this program, which prints input values until a value of zero is input:

```
Top   GET          get a number into the accumulator
      IFZERO Bot   if the accumulator value is zero, go to instruction labeled Bot
      PRINT        value was not zero, so print it
      GOTO Top     go back to Top to get another number
Bot   STOP
```

This keeps fetching data and printing it until the user gets tired and enters a zero, at which point the program jumps to the `STOP` instruction (labeled `Bot` for "bottom") and quits.

Note that the program does not print the zero that signaled the end of the input. How would you modify the program so that it does print the zero before stopping? This is not a trick question—the answer is clear—but it is a good illustration of how programs may embody subtle distinctions in what they are supposed to do, or how they might do something different than what was intended, all because of a simple transposition of two instructions.

The combination of `GOTO` and `IFZERO` lets us write programs that repeat instructions until some specified condition becomes true; the CPU can change the course of a computation according to the results of previous computations. With `IFZERO` in its repertoire, the Toy CPU can now be used to do literally *any* computation. It's probably not obvious, but this is all we need to compute anything that can be computed by any digital computer—any computation can be broken down into small steps using elementary instructions. We'll come back to this from time to time, since it's an important idea. (You might think about whether `GOTO` is strictly necessary if you have `IFZERO`—is there a way to simulate the former with the latter?)

For example, here's a program that will add up a bunch of numbers, stopping when zero is entered. Using a special value to terminate a sequence of inputs is a common practice. Zero works well here because we're adding numbers and adding a zero data value isn't necessary.

```
Top   GET          get a number
      IFZERO Bot   if it's zero, go to Bot
      ADD Sum      add running sum to the latest number
      STORE Sum    store result as new running sum
      GOTO Top     go back to Top to get another number
Bot   LOAD Sum     load running sum into accumulator
      PRINT        and print it
      STOP
Sum   0            memory location to hold running sum
                     (initialized to 0 when program starts)
```

Toy simulators interpret an "instruction" like the last line of this program to mean "assign a name to a memory location and put a value in that location before the program starts to run." We need that because we are adding up a bunch of numbers and the running sum has to start at zero.

How would you check this program to be sure it works? It looks right on the surface, but problems are easily overlooked, so it's important to test systematically. The operative word is "systematically"; it's not effective to just throw random inputs at a program.

What's the simplest test case? If there are no numbers at all except the zero that terminates the input, the sum should be zero, so that's a good first test case. The second thing to try is to input a single number; the sum should be that number. Next is to try two numbers whose sum you already know, like 1 and 2; the result should be 3. At this point, you can be reasonably confident that the program is working. If you're careful, you can test the code before it ever gets near a computer by stepping carefully through the instructions yourself. Good programmers do this kind of checking on everything they write.

So far I have ducked the question of how instructions and data are represented in the RAM. Here's one possibility. Suppose that each instruction requires one memory location to store its numeric code, and another location right afterwards if the instruction refers to memory or has a data value. That is, instructions like `IFZERO` and `ADD` that reference a memory location occupy two memory cells, the second of which is the location that they refer to. Suppose also that data values fit in a single location. This is a simplification, though not wildly far off what really happens. Finally, suppose that the numeric values of the instructions are `GET` = 1, `PRINT` = 2, `STORE` = 3, `LOAD` = 4, `ADD` = 5, `STOP` = 6, `GOTO` = 7, `IFZERO` = 8, following their order of appearance a few pages back.

Now consider the program above that adds up a sequence of numbers. When the program is about to begin, the RAM contents would be as shown below; the diagram also shows the names attached to three memory locations, the instructions and addresses that correspond to the RAM contents, and at the bottom the actual RAM location.

Top:									Bot:				Sum:
GET	IFZERO	Bot	ADD	Sum	STORE	Sum	GOTO	Top	LOAD	Sum	PRINT	STOP	
1	8	10	5	14	3	14	7	1	4	14	2	6	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14

The Toy simulator illustrated above is written in JavaScript, a programming language that we will talk about in Chapter 7. The simulator is easy to extend; for instance it's straightforward to add a multiplication instruction or a different kind of conditional branch even if you've never seen a computer program before.

3.2 Real CPUs

A CPU performs a simple cycle over and over. It fetches the next instruction from memory, which is normally the instruction stored in the next memory location, but could instead be one specified by a `GOTO` or `IFZERO`. The CPU decodes the instruction, that is, it figures out what the instruction does and makes whatever preparations are necessary to carry it out. It executes the instruction, by fetching information from memory, doing arithmetic or logic, and storing results, in whatever combination is appropriate for that instruction. Then it goes back to the fetch part of the cycle. The fetch-decode-execute cycle is what real processors do too, though with elaborate mechanisms for making the whole process run fast. It's just a loop, like the ones we showed above for adding up numbers.

Real computers have more instructions than our Toy does, though basically they are of the same kinds. They have more ways to move data around, more ways to do arithmetic and on different sizes and kinds of numbers, more ways to compare and branch, and more ways to control the rest of the machine. A typical CPU will have a few dozen to a few hundred different instructions; instructions and data typically occupy multiple memory locations, often 2 to 8 bytes. A real processor will have multiple accumulators, often 16 or 32, so that it can hold multiple intermediate results in what amounts to extremely fast memory. Real programs are enormous by comparison with our Toy examples, sometimes with millions of instructions. We'll come back to how such programs are written when we talk about software in later chapters.

Computer architecture is the discipline that deals with designing the CPU and its connections to the rest of the machine; in universities, it's usually a subfield on the border of computer science and electrical engineering.

One concern of computer architecture is the instruction set—the repertoire of instructions that the processor provides. Should there be a large number of instructions to handle a wide variety of different kinds of computation, or should there be fewer instructions that might be simpler to build and might run faster? Architecture involves complicated tradeoffs among functionality, speed, complexity, programmability (if it's too complicated, programmers won't be able to take advantage of the

features), power consumption, and other issues. Quoting von Neumann, "In general, the inner economy of the arithmetic unit is determined by a compromise between the desire for speed of operation [...] and the desire for simplicity, or cheapness, of the machine."

How is the CPU connected to the RAM and the rest of the computer? Processors are very fast, typically performing an instruction in a fraction of a nanosecond. (Recall that "nano" is one billionth, or 10^{-9}.) By comparison, RAM is excruciatingly slow—fetching data or an instruction from RAM might take 25 to 50 nanoseconds. That's fast in absolute terms, of course, but it's slow from the perspective of the CPU, which might have done a hundred instructions if it didn't have to wait for data to arrive.

Modern computers use a small number of high-speed memories called *caches* between the CPU and the RAM to hold recently-used instructions and data. If information can be found in a cache, that will be faster than waiting for it to come from RAM. I'll discuss caches and caching in the next section.

Designers also have a collection of architectural tricks that make processors run faster. A CPU can be designed to overlap fetch and execute so that there are several instructions in various stages of completion; this is called pipelining. (It's similar in spirit to cars moving along an assembly line.) The result is that although any given instruction still takes the same time to complete, others are in process at the same time and the overall completion rate is higher. Another option is to do multiple instructions in parallel if they do not interfere with or depend on each other; the automotive analogy would be to have parallel assembly lines. Sometimes it's possible to do instructions out of order if their operations do not interact.

Yet another option is to have multiple CPUs operating at the same time. This is the norm in laptops today and even in cell phones. The Intel Core Duo processor has two CPUs ("cores") on a single integrated circuit chip, and the trend is strongly towards more and more processors on single chips. As integrated-circuit feature sizes become smaller, it's possible to pack more transistors onto a chip, and those tend to be used either for more CPUs or for caches.

A different set of tradeoffs in processor design comes into play when one considers where the processor will be used. For a long time, the main target was desktop computers, where both electrical power and physical space were comparatively plentiful. That meant that designers could concentrate on making the processor run as fast as possible, knowing that there would be plenty of power and a way to dissipate heat with fans. Laptops significantly changed this set of tradeoffs, since space is tight and an unplugged laptop gets its power from a battery that is heavy and expensive. All else being equal, processors for laptops tend to be slower and use less power.

Cell phones and other highly portable devices push this tradeoff even further, since size, weight and power are even more constrained. This is an area where tweaking a design doesn't work. Although Intel is the dominant supplier of processors for desktops and laptops, almost all cell phones use a processor called an "ARM," which use much less power. ARM processors are based on a design licensed

from the English company ARM Holdings.

Speed comparisons between different CPUs are not terribly meaningful. Even basic operations like arithmetic can be handled in ways that are sufficiently different that it's hard to make a head to head comparison. For example, one processor might require three instructions to add two numbers and store the result in a third (as the Toy does). A second processor might require two instructions, while a third processor might have a single instruction for that operation. One CPU might be able to handle several instructions in parallel, or overlap several so that they are proceeding in stages. It's common for processors to sacrifice speedy execution for lower power consumption, even adjusting their speed dynamically depending on whether power is coming from batteries. You should be cautious about claims that one processor is "faster" than another; your mileage may vary.

3.3 Caching

It's worth a brief digression here on caching, an idea of broad applicability in computing. In the CPU, a *cache* is a small, very fast memory that is used to store recently used information to avoid accessing the RAM. A CPU will normally access groups of data and instructions multiple times in short succession. For instance, the half-dozen instructions of the loop in the program that adds up numbers will be executed once for each input number. If those instructions are stored in a cache, then there will be no need to fetch them from RAM each time through the loop, and the program will run perhaps fifty times faster because it does not have to wait for RAM. Similarly, keeping `Sum` in a data cache will speed access as well.

A typical CPU has two or three caches, successively larger but slower, and often called levels L1, L2 and L3; the largest might hold some megabytes of data (my Macbook has 3 MB in its L2 cache), and most CPUs have separate caches for instructions and data. Caching works because recently used information is more likely to be used again soon—having it in the cache means less waiting for RAM. The caching process usually loads blocks of information all at once, for instance a block of consecutive RAM locations when a single byte is requested, because adjacent information will probably be used as well and is thus likely to already be in the cache when needed; references to nearby information may not have to wait.

This kind of caching is invisible to users except insofar as it improves performance. But caching is a much more general idea that is helpful whenever we use something now and are likely to use it again soon or are likely to use something nearby soon. Multiple accumulators in CPUs are in effect a form of cache at the high speed end. RAM can be a cache for disks, and RAM and disks are both caches for data coming from a network. Networks often have caches to speed up the flow of information from faraway servers, and servers themselves have caches.

You may have seen the word in the context of "clearing the cache" for a web browser. The browser keeps a local copy of images and other comparatively bulky material that is part of some web page, since it will be faster to use the local copy

than to download it again if a page is revisited. The cache can't grow indefinitely, so the browser will quietly remove old items to make room for newer ones, and it offers you a way to remove everything as well.

You can often observe cache effects for yourself. Here are a couple of experiments to try. First, start a big program like Word or Firefox and measure how long before it's finished loading and ready to use. Then quit the program and restart it immediately. Usually the startup will be noticeably faster the second time, because the instructions of the program are still in RAM, which is serving as a cache for the disk. As you use other programs over a period of time, the RAM will fill with their instructions and data and the original program won't be cached any more.

Second, search for some unlikely word or phrase at Google and note the length of time that Google reports it spent answering the query. Now do the same search again. The reported time will usually be significantly shorter, because Google has cached the search result on its servers. Others who do the same search will profit from that as well, since the cache is at Google, not on your machine. You can verify that the cache is at Google by asking someone else to do the same search on their computer right after you do yours. It's not guaranteed but generally the second search will be faster.

3.4 Other Kinds of Computers

It's easy to think that all computers are PCs or Macs, because that's what we see most often. But there are many other kinds of computers, large and small, that share the core attributes of what they can logically compute, and that have similar architectures but make very different tradeoffs among cost, power, size, speed, and so on. Cell phones and tablets are computers too, running an operating system and providing an increasingly rich computing environment. Even smaller systems are embedded in almost all the digital gadgets that clutter our lives, including cameras, video recorders, GPSs, appliances, games, and on and on and on.

Bigger computers have for many years had multiple CPUs connected to the same memory. This provides a way to compute faster as long as the task can be broken down in such a way that different CPUs can cooperate on solving it, rather than waiting for or interfering with each other too much. Widely used on larger systems, this is now common on personal machines as well, with multiple processors on a single chip, and is likely to be the norm in the future.

Supercomputers tend to have a large number of processors and a lot of memory, and the processors themselves may have instructions that process certain kinds of data much faster than their more conventional siblings. Today's supercomputers are often based on clusters of speedy but basically ordinary CPUs rather than specialized hardware. The web site `top500.org` publishes a new list every six months of the world's 500 fastest computers. It's striking how quickly the top speed rises; a machine that might have been in the top handful a few years ago would now not even be on the list at all. The top machine in June 2011 has over half a million CPUs and

can execute more than 8×10^{15} arithmetic operations per second.

Distributed computing refers to computers that are more independent—they don't share memory, for example, and they may be physically more spread out, even located in different parts of the world. This makes communication even more of a potential bottleneck, but it enables people and computers to cooperate at large distances. Large-scale web services—search engines, online shopping, and social networks—are distributed computing systems, with many thousands of computers cooperating to provide results quickly for large numbers of users.

All of these kinds of computing share the same fundamental principles. They are based on a general-purpose processor that can be programmed to perform an endless variety of tasks. Each processor has a limited repertoire of simple instructions that do arithmetic, compare data values, and select the next instruction to perform based on the results of previous computations. The general architecture hasn't changed much since the late 1940s, though the physical construction has continued to evolve at an amazing rate.

Perhaps unexpectedly, all of these computers have the same logical capabilities and can compute exactly the same things (leaving aside practical considerations like speed and memory requirements). This result was proven independently in the 1930s by several people, including the English mathematician Alan Turing. Turing's approach is the easiest for a non-specialist to understand. He described a very simple computer, much simpler than our Toy, and showed that it could compute anything that was computable; that kind of computer is today called a *Turing machine*. He then showed how to create a Turing machine that would simulate any other Turing machine; that's now called a *universal Turing machine*. It's easy to write a program that will simulate a universal Turing machine, and it's also possible (though not easy) to write a program for a universal Turing machine that will simulate a real computer. Hence, all computers are equivalent in what they can compute, though clearly not in how fast they operate.

During World War II, Turing turned from theory to practice: he was central to the development of computers for decrypting German military communications. In 1950, he published a paper called "Computing machinery and intelligence", which proposed a test (today called the *Turing test*) that one might use to assess whether a computer was displaying human intelligence. Imagine a computer and a human communicating with a human interrogator via a keyboard and display. By having a conversation, can the interrogator determine which is the human and which is the computer? Turing's thought was that if they could not be reliably distinguished, the computer was displaying displaying intelligent behavior.

Turing's name is part of the (somewhat forced) acronym *CAPTCHA*, which stands for "Completely Automated Public Turing test to tell Computers and Humans Apart." CAPTCHAs are the distorted patterns of letters that are widely used to try to ensure that the user of a web site is a human, not a program:

A CAPTCHA is an example of a *reverse Turing test*, since it attempts to distinguish between human and computer by using the fact that people are better at identifying the words than computers are.

Turing is one of the most important figures in computing. He contributed greatly to our understanding of computation. The Turing Award, computer science's equivalent of the Nobel Prize, is named in his honor; later chapters will describe half a dozen important computing inventions that have received the Turing Award.

Wrapup on Hardware

We've come to the end of our discussion of hardware, though we'll return occasionally to talk about some gadget or device. Here are the fundamental ideas you should take away from this part.

A digital computer contains a processor and memory. The processor executes simple instructions very quickly. It can decide what to do next based on the results of earlier computations and inputs from the outside world. The memory contains both data and instructions that determine how to process the data.

A computer is a general-purpose machine. It takes its instructions from memory, and one can change the computation it performs by putting different instructions in the memory. Instructions and data are indistinguishable except by context; one person's instructions are another person's data.

The Turing result: all computers of this structure (which includes anything you're likely to see) have exactly the same computational power, in the sense that they can compute exactly the same things. Their performance can vary widely, of course, but all are equivalently capable except for issues of speed and memory capacity. The tiniest and simplest computer can compute anything that its larger siblings could. Indeed, any computer can be programmed to simulate any other, which is in effect how Turing proved his result.

The logical structure of computers has not changed much since von Neumann, but the physical construction has changed enormously. Moore's Law, in effect for about fifty years and by now almost a self-fulfilling prophecy, has described an exponential decrease in the size and price of individual components and thus an exponential increase in computational power for a given amount of space and money.

There's one last topic that should be mentioned. These are *digital* computers: everything is ultimately reduced to bits, which individually or in groups represent

information, whether instructions or data, as numbers. The interpretation of the bits depends on the context. Anything that we are able to reduce to bits can be represented and processed by a digital computer.

The huge amount of available data can make it possible to do a decent job on tasks normally associated only with humans. Translation from one language to another is one notable example; there is so much parallel text available that clever programs can do a reasonable job of translating from one language to another, and although they are far from perfect, they continue to get better. Here's the paragraph above rendered in French by Google Translate:

> "Ce sont des ordinateurs numériques: tout est finalement réduit en morceaux, qui, individuellement ou en groupes représentent des informations, si les instructions ou des données, comme des nombres. L'interprétation des bits dépend du contexte. Tout ce que nous sommes en mesure de réduire de bits peuvent être représentées et traitées par un ordinateur numérique."

then back into English:

> "These are digital computers, everything is finally reduced to pieces which, individually or in groups that represent such information, either instructions or data. The interpretation of bits depends on the context. All that we are able to cut in pieces may be represented and processed by a digital computer."

This is pretty remarkable from one perspective, even allowing for grammar problems, but it has missed the crucial fact that when talking about computers, "bits" does not mean "pieces," nor is "reduced to bits" the same as "cut in pieces". Similar results apply today to understanding speech, recognizing faces, and other kinds of image processing: computers can do remarkable things but no one would say their abilities were at a human level yet. By the way, the translation will be different if you try it; the algorithms and underlying data seem to change frequently.

Computers play chess at the grandmaster level but are poorer at face recognition than babies are. Computers used to play chess quite poorly, but as machines have become faster, the level of chess has improved markedly; it's almost entirely due to being able to look further ahead than human players can. Somewhat analogously, the most progress in tasks like language translation and speech understanding has come from having large amounts of data—parallel texts in different languages, for instance—that can be used to come closer to human performance.

In February 2011, the popular television program *Jeopardy* pitted the two best human players against an IBM computer system called Watson. *Jeopardy* requires a great deal of factual knowledge and an ability to handle puns, elliptical meanings, loose associations, and other verbal skills. It's fair to say that Watson roundly defeated the humans. The Turing test asks whether it's possible to determine which of two players is the human and which is the computer by asking questions of them, masking irrelevancies like physical appearance, response time, and the like. It is hard to distinguish the humans from the computer by reading a transcript of the *Jeopardy* show.

Nevertheless, there are many, many things that we do not know how to encode in bits, nor how to process in a computer. Most of these are important things in life: art, creativity, truth, beauty, love, honor, and values. I suspect those will remain beyond our computers for some while. You should be skeptical of anyone who claims to know how to deal with such matters "by computer."

Part II

Software

The good news is that a computer is a general-purpose machine, capable of performing any computation. Although it only has a few kinds of instructions to work with, it can do them blazingly fast, and it can largely control its own operation.

The bad news is that it doesn't do anything itself unless someone tells it what to do, in excruciating detail. A computer is the ultimate sorcerer's apprentice, able to follow instructions tirelessly and without error, but requiring painstaking accuracy in the specification of what to do.

Software is the general term for sequences of instructions that make a computer do something useful. It's "soft" in contrast with "hard" hardware, because it's intangible, not easy to put your hands on. Hardware is quite tangible: if you drop a computer on your foot, you'll notice. Not true for software.

In the next few chapters we're going to talk about software: how to tell a computer what to do. Chapter 4 is a discussion of software in the abstract, and focuses on algorithms, which are in effect idealized programs for focused tasks. Chapter 5 discusses programming and programming languages, which we use to express sequences of computational steps. Chapter 6 describes the major kinds of software systems that we all use, whether we know it or not. The final chapter of this section, Chapter 7, is about programming in JavaScript.

Something to keep in mind as we go: modern systems increasingly use general-purpose hardware—a processor, some memory, and connections to the environment—and create specific behaviors by software. The conventional wisdom is that software is cheaper, more flexible, and easier to change than hardware is (especially once some device has left the factory). For example, if a computer controls how power and brakes are applied in a car, apparently different functions like anti-lock braking and electronic stability control are software features. In a striking example,

the April 2010 issue of *Consumer Reports* rated the Toyota Lexus GX 460 "don't buy: safety risk" because its electronic stability control system allowed the rear end of the SUV to slide out excessively during a turning maneuver. Within a month, Toyota issued a *software* upgrade that fixed the problem. There was never a mechanical problem, just software that didn't work right originally.

As an aside, Toyota owners had to visit a dealer to have the upgrade installed, which was presumably done by plugging a connector into the proper socket in the engine compartment—maybe not a USB socket, but it would be the same idea. It's not hard to imagine a scenario where the upgrade could be done by driving past the dealer's location, or wirelessly from a smartphone. Of course, it's also not hard to imagine potential problems with drive-by upgrades.

This example also reminds us that computers are central to many critical systems, and that software controls those systems. Medical imaging systems like MRI and CT scans use computers to control signals and form images for doctors to interpret (and film has been replaced by digital images). A modern automobile has dozens of small computers that control features like braking and stability control; if something goes wrong, that can be serious. The same is true of trains, boats, and airplanes; a pilot friend recently told me how a software upgrade had completely changed the behavior of the autopilot in his plane. It's also true of infrastructure like the air traffic control system, navigational aids, the power grid, and phones. Computer-based voting machines have had serious flaws. Military systems are entirely dependent on computers as well, as are the world's financial systems. "Cyber warfare", referring to attacks on these and other computer systems, has become a popular term. Are these realistic threats? Some seem to be; for instance the Stuxnet worm of late 2010 attacked uranium-enrichment centrifuges in Iran, which certainly doesn't sound like a random target.

It's already the case that if our software is not reliable and robust, we're in trouble. The potential for real trouble is only going to get worse as our dependency grows. As we'll see, it's hard to write software that is completely reliable. Any error or oversight in logic or implementation can cause a program to behave incorrectly, and even if that doesn't happen in normal use, it might leave an opening for an adversary.

4

Algorithms

A popular metaphor for explaining software is to compare it to recipes for cooking. A recipe lists the ingredients needed to make some dish, the sequence of operations that the cook has to perform, and the expected outcome. By analogy, a program needs certain data to operate on, spells out what to do to the data, and tells what it's meant to produce. Real recipes are much more vague and ambiguous than programs could ever be, however, so the analogy isn't terribly good. For instance, our family copy of *Joy of Cooking* says that to poach an egg, "Put in a small bowl: 1 egg" but fails to specify that the egg should be broken and the shell removed first.

Tax forms are a better metaphor: they spell out in painful detail what to do ("Subtract line 30 from line 29. If zero or less, enter 0. Multiply line 31 by 25%, ..."). The analogy is still imperfect but tax forms capture the computational aspects much better than recipes do: arithmetic is required, data values are copied from one place to another, and subsequent computations depend on the results of earlier ones.

For taxes especially, the process should be complete—it should always produce a result, the amount of tax due, no matter what the situation. It should be unambiguous—anyone starting from the same initial data should arrive at the same final answer. And it should stop after a finite amount of time. Speaking from personal experience, these are all idealizations, because terminology is not always clear, instructions are more ambiguous than the tax authorities care to admit, and frequently it's uncertain what data values to use.

An *algorithm* is the computer science version of a careful, precise, unambiguous recipe or tax form, a sequence of steps that is guaranteed to perform a particular computation correctly. Each step is expressed in terms of basic operations whose meaning is completely specified, for example "add two numbers." There's no ambiguity about what anything means. The nature of the input data is given. All possible

situations are covered; the algorithm never encounters a situation where it doesn't know what to do next. (When in a pedantic mood, computer scientists usually add one more condition: the algorithm has to stop eventually. By that standard, the classic shampoo instruction to "Lather, Rinse, Repeat" is not an algorithm.)

The design, analysis, and implementation of efficient algorithms is a core part of academic computer science, and there are many real-world algorithms of great importance. I'm not going to try to explain or express algorithms precisely, but I do want to get across the idea of specifying a sequence of operations in sufficient detail that there is no doubt about what they mean and how to perform them, even if they are being performed by an entity with no intelligence or imagination. I also want to discuss algorithmic efficiency, that is, how computation time depends on the amount of data to be processed. I'll do this for a handful of basic algorithms that are familiar and readily understood.

You don't have to follow all the details nor the occasional formulas in this chapter, but the ideas are worth studying.

4.1 Linear Algorithms

Suppose we want to find out who is the tallest person in the room. We could just look around and make a guess, but an algorithm must spell out the steps so precisely that even a dumb computer can follow them. The basic approach is to ask each person in turn about their height, and keep track of the tallest person seen so far. Thus we might ask each person in turn: "John, how tall are you? Mary, how tall are you?" and so on. If John is the first person we ask, then so far he's the tallest. If Mary is taller, she's now the tallest; otherwise, John retains his position. Either way, we ask the third person. At the end of the process, after each person has been asked, we know who the tallest person is and his or her height. Simple variations would find the richest person, or the person whose name comes first in the alphabet, or whose birthday is closest to the end of the year, or who was born in May, or who is named Chris.

There are complications. How do we handle duplicates, for instance two or more people who are the same height? We have to choose whether to report the first or the last or a random one, or perhaps all of them. Notice that finding the largest group of people who are equally tall is a harder problem, since it means we have to remember the names of all people who are the same height: we won't know until the end who is on the list. This example involves data structures—how to represent the information needed during a computation—which are an important part of many algorithms, though we won't talk much about them here.

What if we want to compute the average height? We could ask each person for their height, add up the values as they are obtained (perhaps using the Toy program for adding up a sequence of numbers), and at the end divide the sum by the number of people. If there are N heights written on a piece of paper, we might express this example more "algorithmically" like this:

```
set sum to 0
for each height on the piece of paper
   add the height to sum
average is sum / N
```

We have to be more careful if we're asking a computer to do the job, however. What happens, for example, if there are no numbers on the piece of paper? This isn't a problem if a person is doing the work, since we realize that there's nothing to do. A computer, by contrast, has to be told to test for this possibility and how to act if it occurs. If the test isn't made, the result will be an attempt to divide the sum by zero, which is an undefined operation. Algorithms and computers have to handle all possible situations. If you've ever seen a check made out for "0 dollars and 00 cents" or received a bill telling you to pay a balance due of zero, you've seen an example of failing to test all cases properly.

What if we don't know ahead of time how many data items there will be, as is usually the case? Then we need a version that counts the items as it accumulates the sum:

```
set sum to 0
set N to 0
while there are heights left to process
   add the next height to sum
   add 1 to N
if N is greater than 0
   average is sum / N
otherwise
   report that no heights were given
```

This shows one way to handle the problem of potential division by zero, by testing for the awkward case explicitly.

One crucial property of algorithms is how efficiently they operate—are they fast or slow, and how long are they likely to take to handle a given amount of data? For the examples given above, the number of steps to perform, or how long it might take a computer to do the job, is directly proportional to the amount of data that must be processed: if there are twice as many people in the room, it will take twice as long to find the tallest or compute the average height, and if there are ten times as many, it will take ten times as long. When the computation time is directly or linearly proportional to the amount of data, the algorithm is called *linear-time* or just *linear*. If we were to plot running time against the number of data items, the graph would be a straight line pointing upward and to the right. Most of the algorithms we encounter in day-to-day life are linear, since they involve doing the same basic operation or operations on some data, and more data means more work in direct proportion.

Linear algorithms take the same basic form. There may be some initialization, like setting the running sum to zero or the largest height to a small value. Then each item is examined in turn, and a simple computation is done on it—counting it, comparing it to a previous value, transforming it in a simple way. At the end, something

may be needed to finish the job, like computing the average or printing the sum or the largest height. If the operations take the same amount of time for each item, then the total time is proportional to the number of items.

4.2 Binary Search

Can we ever do any better? Suppose we have a bunch of names and telephone numbers in a printed list or a stack of business cards. If the names are in no particular order and we want to find Mike Smith's number, we have to look at all the names until we find his (or fail to find it because it's not there at all). If the names are in alphabetic order, however, we can do better.

Think about how we look up a name in an old-fashioned paper phone book. We start approximately in the middle. If the name we're looking for is earlier in the alphabet than names on the middle page, we can ignore the last half of the book and look next at the middle of the first half (one quarter of the way into the original); otherwise, we ignore the first half and check the middle of the last half (three quarters). Because the names are in alphabetic order, at each step we know in which half to look next. Eventually we get down to the point where either the name we're looking for has been found, or we know for sure that it's not there at all.

This search algorithm is called *binary search*, because each check or comparison divides the items into two groups, one of which can be eliminated from further consideration. It's an example of a general strategy called *divide and conquer*. How fast is it? At each step, half the remaining items are eliminated, so the number of steps is the number of times we can divide the original size by 2 before we get down to a single item.

Suppose we start with 1,024 names (a number chosen to make the arithmetic easy). With one comparison, we can eliminate 512 names. Another comparison gets us down to 256, then 128, then 64, then 32, then 16, 8, 4, 2, and finally 1. That's 10 comparisons. It's clearly not a coincidence that 2^{10} is 1,024. The number of comparisons is the power of 2 that brings us to the original number; running the sequence up from 1 to 2 to 4 ... to 1,024, we're multiplying by 2 each time.

If you remember logarithms from school (not many people do—who would have thought?), you may recall that the logarithm of a number is the power to which one raises the base (2 in this case) to get the number. So the log (base 2) of 1,024 is 10, since 2^{10} is 1,024. For our purposes, the log is the number of times you have to divide a number by 2 to get down to 1, or the number of times you have to multiply 2 by itself to get up to the number. In this book, there's no need for precision or fractions; ballpark figures and integer values are good enough, a real simplification.

The important thing about binary search is that the amount of work to be done grows slowly with the amount of data. If there are 1,000 names in alphabetical order, we have to look at 10 names to find a specific one. If there are 2,000 names, we only have to look at 11 names because the first name we look at immediately eliminates 1,000 of the 2,000, bringing us back to where we have to look at 1,000 (10 tests). If

there are a million names, that's 1,000 times a thousand. The first 10 tests get us back down to 1,000, and another 10 gets us down to 1, which is 20 tests in all. A million is 10^6, which is close to 2^{20}, so the log (base 2) of a million is about 20.

From this, you should be able to see that looking up a name in a directory of a billion names (the whole earth phone book, nearly) would only take 30 name comparisons, since a billion is about 2^{30}. That's why we say that the amount of work grows slowly with the amount of data—a thousand times more data requires only 10 more steps.

As a quick validation, I decided to search for my friend Harry Lewis in an old Harvard paper telephone directory, which has about 20,000 names in 224 pages. I began at page 112, finding the name Lawrence. "Lewis" comes later than that; it's in the second half, so I next tried page 168, halfway between 112 and 224, to find Rivera. "Lewis" is before that, so I tried 140 (halfway between 112 and 168), leading to Morita. Next to 126 (halfway between 112 and 140) to find Mark. The next try was 119 (Little), then 115 (Leitner), then 117 (Li), to 116. There were about 90 names on this page, so another 7 comparisons, all on the same page, found Harry among a dozen other Lewises. This experiment took 14 tests in all, which is about what would be expected, since 20,000 lies between 2^{14} (16,384) and 2^{15} (32,768).

Divide and conquer shows up in real-world settings like the knockout tournaments used in many sports. The tournament starts with a large number of competitors, for example 128 for men's singles tennis at Wimbledon. Each round eliminates half the contenders, leaving a pair in the final round, from which a single winner emerges. Not by coincidence, 128 is a power of two (2^7), so Wimbledon has seven rounds. One could imagine a worldwide knockout tournament; even with seven billion participants, it would only take 33 rounds to determine the winner. (If you recall the discussion of powers of two and ten in Chapter 2, you can verify this with easy mental arithmetic.)

4.3 Sorting

But how do we get the names into alphabetical order in the first place? Without that preliminary step, we can't use binary search. This brings us to another fundamental algorithm, *sorting*, which puts things into order so that subsequent searches can run fast.

Suppose we want to sort some names into alphabetical order so we can later search them efficiently with binary search. One algorithm is called *selection sort*, since it keeps selecting the next name from among the ones not yet in order. It's based on the technique for finding the tallest person in the room that I showed earlier.

Let me illustrate by sorting these 16 familiar names alphabetically:

Intel Facebook Zillow Yahoo Picasa Twitter Verizon Bing
Apple Google Microsoft Sony PayPal Skype IBM Ebay

Intel is first, so it's alphabetically first so far. Facebook comes earlier in the alphabet, so it temporarily becomes the new first name. Zillow is not earlier, nor are any other names until Bing, which replaces Facebook; it is replaced by Apple. We examine the rest of the names but none of them precedes Apple, so Apple is truly first in the list, which now looks like

Apple
Intel Facebook Zillow Yahoo Picasa Twitter Verizon Bing
Google Microsoft Sony PayPal Skype IBM Ebay

Now we repeat the process to find the second name, starting with Intel, which is the first name in the unsorted group. Again Facebook replaces it, then Bing becomes the first element. After the second pass, the result is

Apple Bing
Intel Facebook Zillow Yahoo Picasa Twitter Verizon
Google Microsoft Sony PayPal Skype IBM Ebay

Eventually this algorithm produces a completely sorted list.

How much work does it do? It makes repeated passes through the remaining items, each time finding the next name in alphabetic order. With 16 names, the search for the first name looks at 16 names. Finding the second name takes 15 steps, finding the third takes 14 steps, and so on. In the end, we have to look at 16+15+14+...+3+2+1 names, 136 in all. Of course we might get lucky and discover that the names are already in order, but computer scientists who study algorithms are pessimists and assume the worst case (the names could be in reverse alphabetical order).

The number of passes through the names is directly proportional to the original number of items (16 for my example, or N in general). Each pass has to deal with one less item, however, so the amount of work in the general case is

$$N + (N-1) + (N-2) + (N-3) + \cdots + 2 + 1$$

This series adds up to $N \times (N+1)/2$ (most easily seen by adding the numbers in pairs from each end), which is $N^2/2 + N/2$. Ignoring the division by two, the work is proportional to $N^2 + N$. As N gets bigger, N^2 eventually swamps N (for instance, if N is a thousand, N^2 is a million), so the effect is that the amount of work is approximately proportional to N^2 or the square of N, a growth rate called *quadratic*. Quadratic is worse than linear; in fact, it's much worse. If there are twice as many items to sort, it will take four times as long; if there are ten times as many items, it will take one hundred times as long; and if there are a thousand times as many items, it will take a million times as long! This is not good.

Fortunately, it is possible to sort much more quickly. Let's take a quick look at one clever way, an algorithm called *Quicksort*, which was invented by the English computer scientist Tony Hoare around 1962. (Hoare won the Turing Award in 1980 for multiple contributions, including Quicksort.) It's a great example of divide and

conquer.

Here are the unsorted names again:

Intel Facebook Zillow Yahoo Picasa Twitter Verizon Bing
Apple Google Microsoft Sony PayPal Skype IBM Ebay

To sort the names with Quicksort, first go through the names once, putting the one that begin with A through M in one pile and N through Z in another. That produces two piles, each having about half the names (assuming that the distribution of names isn't too badly skewed). In our case, the two piles would each have eight names:

Intel Facebook Bing Apple Google Microsoft IBM Ebay
Zillow Yahoo Picasa Twitter Verizon Sony PayPal Skype

Now go through the A-M pile, putting A through F in one pile and G through M in another; go through the N-Z pile, putting N-S in one pile and T-Z in another. At this point, we have made two passes through the names, and have four piles, each with about one quarter of the names:

Facebook Bing Apple Ebay
Intel Google Microsoft IBM
Picasa Sony PayPal Skype
Zillow Yahoo Twitter Verizon

The next pass goes through each of those piles, splitting the A-F pile into ABC and DEF and G-M into GHIJ and KLM, and the same for N-S and T-Z; at this point we have eight piles with about 2 names in each:

Bing Apple
Facebook Ebay
Intel Google IBM
Microsoft
Picasa PayPal
Sony Skype
Twitter Verizon
Zillow Yahoo

Of course eventually we will have to look at more than just the first letter of the names, for example to put IBM ahead of Intel and Skype ahead of Sony. But after one or two more passes, we'll have 16 piles of one each, and the names will be in alphabetical order.

How much work was that? We looked at each of the 16 names in each pass. If the split were perfect each time, the piles would have 8 names, then 4, then 2, then 1. The number of passes is the number of times we divide 16 by 2 to get down to 1. That's the log base 2 of 16, which is 4. Thus the amount of work is $16 \log_2 16$ for 16 names. If we make four passes through the data, that's 64 operations, compared to 136 for selection sort.

This algorithm will always sort the data, but it is only efficient if each division divides the group into piles of about the same size. On real data, Quicksort has to guess the middle data value so it can split into two approximately equal-size groups each time, but this can be estimated well enough by sampling a few items. In general (and passing over some subtleties), Quicksort takes about $N \log N$ operations to sort N items: the amount of work is proportional to $N \times \log N$. That's worse than linear, but not terribly so, and it's enormously better than quadratic or N^2 when N is at all large.

As an experiment, I generated 10 million random 9-digit numbers, analogous to US Social Security numbers, and timed how long it took to sort groups of various sizes, with selection sort (N^2 or quadratic) and with Quicksort ($N \log N$). The dashes in the table are cases I did not run.

Number of numbers (N)	Selection sort time (sec)	Quicksort time (sec)
1,000	0.047	-
10,000	4.15	0.025
100,000	771	0.23
1,000,000	-	3.07
10,000,000	-	39.9

It's hard to accurately measure programs that run for very short times, so take these numbers with a large grain of salt. Nevertheless, you can see (roughly) the expected $N \log N$ growth of run time for Quicksort, and you can also see that selection sort might be feasible up to say 10,000 items, though far from competitive; at every stage, it's hopelessly outclassed by Quicksort.

You might also have noticed that the selection sort time for 100,000 items is nearly 200 times larger than for 10,000, rather than the expected 100 times. This is almost surely a caching effect—the numbers don't all fit in the cache and thus sorting is slower.

4.4 Hard Problems and Complexity

We've now studied several points along a spectrum of algorithm "complexity" or running time. At one end is $\log N$, as seen in binary search, where the amount of work grows quite slowly as the amount of data increases. The most common case is linear, or plain N, where the work is directly proportional to the amount of data. There's $N \log N$, as in Quicksort; that's worse (grows faster) than N, but still eminently practical for very large values of N. And there's N^2, or quadratic, which grows quickly enough that it's somewhere between painful and impractical.

There are plenty of other complexity possibilities, some easy to understand (like cubic, or N^3, which is worse than quadratic but the same idea) and others so esoteric that only specialists could care about them. One more is worth knowing about, since it occurs in practice and it's especially bad. It's called *exponential*, and it grows like

2^N (which is not the same as N^2). In an exponential algorithm, the amount of work grows exceptionally rapidly: adding one more item *doubles* the amount of work to be done. In a sense exponential is at the opposite end from a $\log N$ algorithm, where doubling the number of items adds only one more step.

Exponential algorithms arise in situations where there is no better way to compute something than to try all the possibilities one by one. Fortunately, there is a silver lining to the existence of problems that require exponential algorithms. Some algorithms, notably in cryptography, are based on the exponential difficulty of performing a specific computational task. For such algorithms, one chooses N large enough that it's not computationally feasible to solve the problem directly without knowing a secret shortcut—it would simply take far too long—and that provides the protection from adversaries. We'll take a look at cryptography in Chapter 10.

By now you should have an intuitive understanding that some problems are easy to deal with, while others seem harder. It's possible to make this distinction more precise. "Easy" problems are "polynomial" in their complexity, that is, their running time is expressed as some polynomial like N^2, though if the exponent is more than 2, they are likely to be challenging. (Don't worry if you've forgotten about polynomials—here, the word just means an expression with only integer powers of a variable, like N^2 or N^3.) Computer scientists call this class of problems "P", for "polynomial," because they can be solved in polynomial time.

A large number of problems that occur in practice or as the essence of practical problems seem to require exponential algorithms to solve, that is, we know of no polynomial algorithm. These problems are called "NP" problems. NP problems have the property that it's possible to quickly verify that a proposed solution is correct but we can't find a solution quickly. NP stands for "nondeterministic polynomial," a technical term that informally means that they can be solved in polynomial time by an algorithm that always guesses right when it has to make a choice. In real life, nothing is lucky enough to always choose correctly, so this is a theoretical idea only.

Many NP problems are so technical that it's not worth the trouble to try to state them here, but one is easy to explain and intuitively appealing, and its practical applications can be imagined.

This is called the *Traveling Salesman Problem*. A salesman has to start from his or her home city, visit a set of other cities and return home. The goal is to visit each city exactly once (no repeats) and to travel the minimum total distance. This captures the idea of efficiently routing school buses or garbage trucks; when I worked on it long ago, it was used for tasks as different as planning how to drill holes in circuit boards and sending boats to pick up water samples at specific places in the Gulf of Mexico.

The Traveling Salesman Problem has been the subject of intense study for over 50 years, and although we're better at solving larger instances, the solution techniques still amount to clever variations of trying all possible routes. The same is true for a wide variety of other problems of many different types: we don't have a good way to

solve them efficiently. For people who study algorithms, this is frustrating. We don't know whether these problems are intrinsically hard, or we're just dumb and haven't figured out how to deal with them yet, though the betting heavily favors the "intrinsically hard" side.

A remarkable mathematical result proved by Stephen Cook in 1970 showed that many of these problems are equivalent, in the sense that if we could find a polynomial-time algorithm (that is, something like N^2) for one of them, that would enable us to find polynomial-time algorithms for all of them. Cook won the 1982 Turing Award for this work.

The Clay Mathematics Institute offers a prize of one million dollars each for the solution of seven open problems, one of which is whether P equals NP, that is, whether the hard problems are really the same class as the easy problems. (The Poincaré Conjecture, another problem on the list, dates from the early 1900s. It was resolved by the Russian mathematician Grigoriy Perelman and the prize was awarded in 2010, so there are only six problems left—better hurry before someone beats you to it.)

There are a couple of things to keep in mind about this kind of complexity. Although the P=NP question is important, it is more a theoretical issue than a practical one. Complexity results, as stated by computer scientists, are "worst case" results, that is, some problem instances could require the full time to do the work, but not all instances need be that hard. They are also "asymptotic" measures; that is, they only apply for large values of N. In real life, there might be techniques that get the right answer easily for most problems of interest, or that get approximate answers that are good enough for practical purposes, or N might be small enough that the asymptotic behavior doesn't matter. For instance, if you're only sorting a few dozen or even a few hundred items, selection sort might be fast enough even though its complexity is quadratic and thus asymptotically much worse than Quicksort's $N \log N$. If you're only visiting half a dozen cities, it's no big deal to try all possible routes, but that would be infeasible for 60 cities and impossible for 600. Finally, in most real situations, an approximate solution is likely to be good enough; there's no need for an absolutely optimal solution. Algorithms that provide good approximate answers may be entirely adequate, and are much more likely to be practical.

On the flip side, some important applications, like cryptographic systems, are based on the belief that a particular problem truly is difficult, so the discovery of an attack, however impractical it might be in the short run, would likely be significant.

4.5 Summary

Computer science as a field has spent years refining the notion of "how fast can we compute"; the idea of expressing running time in terms of the amount of data (like N, or $\log N$, or N^2, or $N \log N$) is the distillation of that thinking. It ignores concerns like whether one computer is faster than another, or whether you are a better programmer than I am, but it does capture the complexity of the underlying program

or algorithm, and for that reason, it's a good way to make comparisons and to reason about whether some computation is likely to be feasible or not. (The intrinsic complexity of a problem and the complexity of an algorithm to solve it need not be the same. For example, sorting is an $N \log N$ problem, for which Quicksort is an $N \log N$ algorithm, but selection sort is an N^2 algorithm.)

The study of algorithms and complexity is a major part of computer science, as both theory and practice. We're interested in what can be computed and what can't, and how to compute fast and without using more memory than necessary (or perhaps by trading off speed against memory). We look for fundamentally new and better ways to compute; Quicksort is a nice example of that, though from long ago.

In real life, there are many important algorithms that are more specialized and complicated than the basic searching and sorting that we've talked about here. For example, compression algorithms attempt to reduce the amount of memory occupied by music (MP3), pictures (JPEG) and movies (MPEG). Error detection and correction algorithms are also important. Data is subject to potential damage as it is stored and transmitted (for example over noisy wireless channels); algorithms that add controlled redundancy to the data make it possible to detect and even correct some kinds of errors. We'll come back to those in Chapter 9 since they have implications when we talk about communications networks.

Cryptography, the art of sending secret messages so they can only be read by the good guys, not by the bad guys, depends strongly on algorithms. We'll discuss cryptography in Chapter 10, since it's highly relevant when computers exchange private information in a secure fashion.

Search engines like Bing and Google are yet another place where algorithms are crucial. In principle, much of what a search engine does is simple: collect some web pages, organize the information, and make it easy to search. The difference is one of scale. When there are billions of web pages and billions of queries every day, even $N \log N$ isn't good enough, and much algorithmic and programming cleverness goes into making search engines run fast enough to keep up with the growth of the web and our interest in finding things on it. We'll talk more about search engines in Chapter 11.

5

Programming and Programming Languages

So far we have talked about algorithms, which are abstract or idealized descriptions of processes that ignore practicalities. An algorithm is a precise and unambiguous recipe. It's expressed in terms of a fixed set of basic operations whose meanings are completely known and specified; it spells out a sequence of steps using those operations, with all possible situations covered; and it's guaranteed to stop eventually.

On the other hand, a *program* is the opposite of abstract—it's a concrete statement of the steps that a real computer must perform to accomplish a task. The distinction between an algorithm and a program like the difference between a blueprint and a building; one is an idealization and the other is the real thing.

One way to view a program is as one or more algorithms expressed in a form that a computer can process directly. A program has to worry about practical problems like inadequate memory, limited processor speed, invalid or even malicious input data, faulty hardware, broken network connections, and (in the background and often exacerbating the other problems) human frailty. So if an algorithm is an idealized recipe, a program is the instructions for a cooking robot preparing a month of meals for an army while under enemy attack.

We can only get so far with metaphors, of course, so I'm going to talk about real programming enough that you can understand what's going on, though not enough to make you into a professional programmer. Programming can be quite hard—there are many details to get right, and tiny slip-ups can lead to large errors.

Given the amount of programming needed to make computers do everything we need or want, there aren't enough programmers in the world. So one continuing theme in computing has been to enlist computers to handle more and more of the

details of programming. This leads to a discussion of programming languages: languages that let us express the computational steps to perform some task.

It's also hard to manage the resources of a computer, especially given the complexity of modern machines. So we also use the computer to control its own operations, which leads us to operating systems. Programming and programming languages are the topic of this chapter, and software systems, especially operating systems, will come in the next.

You can certainly skip the syntactic details in the programming examples in this chapter, but it's worth looking at similarities and differences in how computations are expressed.

5.1 Assembly Language

When the first truly programmable electronic computers came on the scene, beginning with EDSAC in 1949, they were programmed by writing the numbers for instructions and data that would be stored in memory on a medium like punch cards or paper tape, then loading those numbers into the memory to be executed. This was incredibly difficult even for tiny programs—hard to get right in the first place, and hard to change if a mistake was found or instructions and data had to be changed or added.

Early in the 1950s, programs were created to handle some of the straightforward clerical chores, so that programmers could use meaningful words for instructions (`ADD` instead of `5`, for example) and names for specific memory locations (`Sum` instead of `14`). This powerful idea—a program that will manipulate another program—has been at the heart of most significant advances in software. The program that does this specific manipulation is called an *assembler*, because originally it also assembled any necessary parts of a program that had been written earlier by other programmers. The language is called *assembly language* and programming at this level is called *assembly language programming*. The language that we used to describe and program the Toy processor in Chapter 3 is an assembly language. Assemblers make it much easier to change a program by adding or removing instructions because the assembler keeps track of where instructions and data are located in memory, rather than requiring the programmer to do it.

An assembly language for a particular processor is specific to that processor; it usually matches the instructions of the CPU one for one and it knows the specific way that instructions are encoded in binary, how information is placed in memory, and so on. This means that a program written in the assembly language of one particular kind of CPU, say an Intel processor in a Mac or PC, will be very different from an assembly language program for the same task for a different CPU, like the ARM processor in a cell phone. If one wants to convert an assembly language program from one of those processors to the other, the program must be totally rewritten.

To make this concrete, in the Toy computer it takes three instructions to add two numbers and store the result in a memory location:

```
LOAD X
ADD Y
STORE Z
```

This would be similar in a variety of current processors. In a CPU with a different instruction repertoire, however, this computation might be accomplished with a single instruction:

```
ADD X, Y, Z
```

To convert a Toy program to run on the second computer, a programmer has to be intimately familiar with both processors, and meticulous in converting from one instruction set to the other. It's hard work.

5.2 High-Level Languages

During the late 1950s and early 1960s, another step was taken towards getting the computer to do more for programmers, arguably the most important step in the history of programming. This was the development of "high-level" programming languages that were independent of any particular CPU architecture. High-level languages make it possible to express computations in terms that are closer to the way a person might express them.

Code written in the high-level language is converted by a translator program into instructions in the assembly language of a specific target processor, which in turn are converted to bits to be loaded into the memory and executed. The translator is usually called a *compiler*, another historical term that doesn't convey much insight or intuition.

For example, the computation above that adds two numbers `X` and `Y` and stores the result as a third, `Z`, would be expressed in most high-level languages as

```
Z = X + Y
```

This means "get the values from the memory locations called `X` and `Y`, add them together, and store the result in the memory location called `Z`."

A compiler for the Toy would convert this into the sequence of three instructions, while a compiler for the other computer would convert it into a single instruction. The respective assemblers would then be responsible for converting their assembly language instruction(s) into the actual bit patterns of real instructions, as well as setting aside memory locations for the quantities `X`, `Y` and `Z`. Naturally the bit patterns would be different for the two computers.

This process is illustrated in the diagram below, which shows the same input expression going through two different compilers and their respective assemblers, to produce different sequences of instructions.

As a practical matter, a compiler will likely be divided internally into a "front end" that processes programs in the high-level programming language into some intermediate form, and several "back ends" that convert the intermediate representation into assembly languages for various architectures. This organization is simpler than having multiple completely independent compilers.

High-level languages have great advantages over assembly language. First, they make it possible for many more people to program computers and to program them more quickly. Because high-level languages are closer to the way that people think, they are easier to learn and use; one doesn't need to know anything about the instruction repertoire of any CPU to program effectively in a high-level language. Second, a program in a high-level language is independent of any particular architecture, so the same program can be run on different architectures, usually without any changes at all, merely by compiling it with a different compiler, as in the diagram above. The program is written only once, and can be run anywhere. This makes it possible to amortize development costs over multiple kinds of computers. The compilation step also provides a preliminary check for some kinds of gross errors—misspellings, syntax errors like unbalanced parentheses, operations on undefined quantities, etc.—that the programmer must correct before an executable program can be produced. Such errors are hard to detect in assembly language programs, where any sequence of instructions must be presumed legal. (Of course a syntactically correct program can still be full of semantic errors undetectable by the compiler.) It's hard to overstate the importance of high-level languages.

I'm going to show the same program in five of the most important high-level programming languages so you can get a sense of their similarities and differences. Each program does the same thing as the program that we wrote for the Toy in Chapter 3: it adds up a sequence of numbers; when a zero value is read, it prints the sum and stops. They all have the same structure: name the quantities that the program uses, initialize the running sum to zero, read numbers until zero is encountered, then print the sum. Don't worry about syntactic details; this is mostly to give you an impression of what the languages look like.

The first high-level languages concentrated on specific application domains. One of the earliest languages was called FORTRAN, a name derived from "Formula Translation." FORTRAN was developed by an IBM team led by John Backus, and was very successful for expressing computations in science and engineering. Many scientists and engineers (including me) learned FORTRAN as their first programming language. FORTRAN is alive and well today; it has gone through several evolutionary stages since 1958, but is recognizably the same at its center. Backus received the Turing Award in 1977 in part for his work on FORTRAN.

Here's a FORTRAN program to add up a series of numbers.

```
      integer num, sum
      sum = 0
   10 read(5,*) num
      if (num .eq. 0) goto 20
      sum = sum + num
      goto 10
   20 write(6,*) sum
      stop
      end
```

This is written in FORTRAN 77; it would look a bit different in an earlier dialect or a later one like FORTRAN 2008, the most recent version. You can probably imagine how to translate it into Toy assembly language.

A second major high-level language of the late 1950s was COBOL (Common Business Oriented Language), which was strongly influenced by the work of Grace Hopper on high-level alternatives to assembly language. Hopper worked with Howard Aiken on the Harvard Mark I and II, early mechanical computers, and then at Univac. She was one of the first people to see the potential of high-level languages and compilers. COBOL was specifically aimed at business data processing, with language features to make it easy to express the kinds of computations that go into managing inventories, preparing invoices, doing payrolls, and the like. COBOL too lives on, much changed but still recognizable.

BASIC (Beginner's All-purpose Symbolic Instruction Code), which was developed at Dartmouth in 1964 by John Kemeny and Tom Kurtz, is another language of the time. BASIC was meant to be an easy language for teaching programming. It was especially simple and required very limited computing resources, so it was the first high-level language on the first personal computers. In fact, Bill Gates and Paul

Allen, the founders of Microsoft, got their start by writing a BASIC compiler for the Altair microcomputer in 1975; it was the company's first product. Today one major strain of BASIC is still actively supported as Microsoft Visual Basic.

In the early days when computers were expensive yet slow and limited, there was concern that programs written in high-level languages would be too inefficient, because compilers would not produce assembly code as good as a skilled assembly language programmer could write. Compiler writers worked hard to generate code that was as good as hand-written, which helped to establish the languages. Today, with computers millions of times faster and with plentiful memory, programmers rarely worry about efficiency at the level of individual instructions, though compilers and compiler writers certainly still do.

FORTRAN, COBOL and BASIC achieved part of their success by focusing on specific application areas and intentionally did not try to handle every possible programming task. During the 1970s, languages were created that were intended for "system programming," that is, for writing programmer tools like assemblers, compilers, programming tools, and even operating systems. By far the most successful of these languages was C, developed at Bell Labs by Dennis Ritchie in 1973 and still very widely used. C has changed only a small amount since then; a C program today looks much like one from 30 years earlier.

For comparison, here's the same "add up the numbers" program in C:

```
#include <stdio.h>
main() {
   int num, sum;
   sum = 0;
   while (scanf("%d", &num) != EOF && num != 0)
      sum += num;
   printf("%d\n", sum);
}
```

The 1980s saw the development of languages like C++ (by Bjarne Stroustrup, also at Bell Labs) that were meant to help manage the complexities of very large programs. C++ evolved from C, and looks similar; in most cases a C program is also a valid C++ program (the one above is), though definitely not vice versa. Here's the example of adding up numbers in C++ (one of many ways to write it):

```
#include <iostream>
using namespace std;
main() {
   int num, sum;
   sum = 0;
   while (cin >> num && num != 0)
      sum += num;
   cout << sum << endl;
}
```

Most of the major programs that we use on our own computers today are written in C or C++. I'm writing this book on Macs, where most software is written in C and Objective-C (a C dialect), originally using Word (a C and C++ program), and making backup copies on Unix and Linux operating systems (both C programs) while surfing with Firefox and Chrome (both C++).

During the 1990s, more languages were developed in response to the growth of the Internet and the World Wide Web. Computers continued to get faster processors and bigger memories, and programming speed and convenience became more important than machine efficiency; languages like Java and JavaScript make this tradeoff intentionally.

Java was developed in the early 1990s by James Gosling at Sun Microsystems. Its original target was small embedded systems like appliances and electronic gadgets, where speed didn't matter much but flexibility did. Java was repurposed to run on web pages, where it has not caught on, but it is widely used by web services: when you visit a site like eBay, your computer is running C++ and JavaScript but eBay might well be using Java to prepare the page it sends to your browser. Java is simpler than C++ (though evolving towards similar complexity) but more complicated than C. It's also safer than C since it eliminates some dangerous features and has built-in features to handle error-prone tasks like managing memory use. For that reason, it's also popular as the first language in programming classes.

Here's the add-up-the-numbers program in Java.

```
import java.util.*;
class Addup {
   public static void main (String [] args) {
      Scanner keyboard = new Scanner(System.in);
      int num, sum;
      sum = 0;
      num = keyboard.nextInt();
      while (num != 0) {
         sum += num;
         num = keyboard.nextInt();
      }
      System.out.println(sum);
   }
}
```

The program is a little wordier than in the other languages, which is not atypical of Java, but I could make it two or three lines shorter by combining a couple of computations.

This brings up an important general point about programs and programming: there are always many ways to write a program to do a specific task. In this sense, programming is like literary composition; indeed, concerns like style and effective use of language that matter when writing prose are also important in programming and help separate truly great programmers from the merely good. The rich variety of ways in which programmers can express a computation also means that it's not hard

to spot that a program has been copied from another program rather than being an original creation. This point is made strongly at the beginning of every programming class, yet students still think that changing names or the placement of lines will be sufficient to disguise plagiarism. Sorry—it doesn't work.

JavaScript is a language in the same broad family that began with C, though with plenty of differences. It was created at Netscape by Brendan Eich in 1995. It was designed from the beginning to be used in a browser to achieve dynamic effects on web pages; today almost all web pages include some JavaScript code. We will say more about JavaScript in Chapter 7, but to make the side by side comparisons easy, here is a version of adding up the numbers in JavaScript:

```
var num, sum;
sum = 0;
num = prompt("Enter new value, or 0 to end");
while (num != 0) {
   sum = sum + parseInt(num);
   num = prompt("Enter new value, or 0 to end");
}
alert("Sum = " + sum);
```

In some ways JavaScript is the easiest of all languages to experiment with. The language itself is simple. You don't need to find a compiler; there's one in every browser. The results of your computations are immediately visible. As we'll see shortly you could add a couple of additional lines and put this example on a web page for anyone in the world to use.

Where will languages go from here? My guess is that we will continue to make programming easier by using more computer resources to help us. We'll also continue to evolve towards languages that are safer for programmers. For example, C is an exceedingly sharp tool, and it's easy to write a program that has errors that aren't detected until too late, perhaps after they have been abused for nefarious purposes. Newer languages make it easier to prevent or at least detect some errors, though sometimes at the cost of running slower and using more memory. Most of the time, this is the right tradeoff to make, though there are certainly still plenty of applications—control systems in cars, planes, spacecraft and weapons, for instance—where tight, fast code matters a lot, and highly efficient languages like C will still be used.

Although all languages are formally equivalent (in that they can be used to simulate a Turing machine), they are by no means equally good for all programming tasks. There's a world of difference between writing a JavaScript program to control a complicated web page, and writing a C++ program that implements the JavaScript compiler. It would be unusual to find a programmer equally expert at both of these tasks. Experienced professional programmers might be comfortable and passably proficient in a dozen languages, but they are not going to be equally skilled in all of them.

Thousands if not tens of thousands of programming languages have been invented, though probably fewer than a hundred are in widespread use. Why so

many? As I've hinted, each language represents a set of tradeoffs among concerns like efficiency, expressiveness, safety, and complexity. Many languages are explicitly a reaction to the perceived weaknesses of earlier languages, taking advantage of hindsight and more computing power, and often strongly influenced by the personal taste of the designer. New application areas also give rise to new languages that focus on the new domain.

No matter what happens, programming languages are an important and fascinating part of computer science. As the American linguist Benjamin Whorf said, "Language shapes the way we think and determines what we can think about." There is still much debate about whether this is true for natural languages, but it does seem to apply to the artificial languages that we invent to tell our computers what to do.

5.3 Software Development

Programming in the real world tends to happen on a large scale. The strategy is similar to what one might use to write a book or undertake any other big project: figure out what to do, starting with a broad specification that is broken into smaller and smaller pieces, then work on the pieces separately, while making sure that they hang together. In programming, pieces tend to be of a size such that one person can write the precise computational steps in some programming language. Ensuring that the pieces written by different programmers work together is challenging, and failing to get this right is a major source of errors. For instance, NASA's Mars Climate Orbiter failed in 1999 because the flight system software used metric units for thrust, but course correction data was entered in English units, causing an erroneous trajectory that brought the Orbiter too close to the planet's surface.

The examples above that illustrate different languages are all less than ten lines long (Java excepted). Small programs of the kind that might be written in an introductory programming course will have a few dozen to a few hundred lines of code. The first "real" program I ever wrote—real in the sense that it was used by a significant number of other people—was about 1,000 lines of FORTRAN. It was a simple-minded word processor for formatting and printing my thesis, and it was taken over by a student agency and used for another five years or so after I graduated. The good old days! A more substantial program to do a useful task today might have thousands to tens of thousands of lines. Students in my project courses, working in small groups, routinely turn out two or three thousand lines in 8 to 10 weeks, including the time to design a system and learn a new language or two; the product is often a web service like a used-textbook marketplace or easy access to some university database.

A compiler or a web browser might have hundreds of thousands to a million lines. Big systems, however, will have multiple millions or even tens of millions of lines of code, with hundreds or thousands of people working on them at the same time, and lifetimes measured in years or even decades. Software on this scale requires teams of programmers, testers and documenters, with schedules, deadlines, layers of management, and endless meetings to keep it all going. (I once heard it claimed, by a

colleague who was in a position to know, that there was one meeting for every line of code in a major system he worked on. Since the system had several million lines, perhaps he exaggerated, but experienced programmers might say "not by much.")

5.3.1 Libraries, interfaces, and development kits

If you're going to build a house today, you don't start by cutting down trees to make lumber and digging clay to make your own bricks. Instead, you buy prefabricated pieces like doors, windows, plumbing fixtures, a furnace, and a water heater. House construction is still a big job, but it's manageable because you can build on the work of many others and rely on an infrastructure, indeed an entire industry, that will help.

The same is true of programming. Hardly any significant program is created from nothing. Many components written by others can be taken off the shelf and used. For instance, if you're writing a program for Windows or a Mac, there are libraries of prefabricated menus, buttons, text editors, graphics, network connections, database access, and so on. Much of the job is understanding the components and gluing them together in your own way. Of course many of these components in turn rest on other simpler and more basic ones, often for several layers. Below that, everything runs on the operating system, a program that manages the hardware and controls everything that happens. We'll talk about operating systems in the next chapter.

At the simplest level, programming languages provide a mechanism called functions that make it possible for one programmer to write code that performs a useful task, then package it in a form that other programmers can use in their programs without having to know how it works. For example, the C program a few pages ago includes these lines:

```
while (scanf("%d", &num) != EOF && num != 0)
   sum += num;
printf("%d\n", sum);
```

This code "calls" (that is, uses) two functions that come with C: `scanf` reads data from an input source (analogous to `GET` in the Toy) and `printf` prints output (like `PUT`). A function has a name and a set of input data values that it needs to do its job; it does a computation and returns a result to the part of the program that called it. The syntax and other details here are specific to C and would be different in another language, but the idea is universal. Functions make it possible to create a program by building on components that have been created separately and that can be used as necessary by all programmers. A collection of related functions is usually called a *library*. For instance, C has a standard library of functions for reading and writing data on disks and other places, and `scanf` and `printf` are part of that library.

The services that a function library provides are described to programmers in terms of an *Application Programming Interface*, or *API*, which lists the functions, what they do, how to use them in a program, what input data they require, and what values they produce. The API might also describe data structures—the organization

of data that is passed back and forth—and various other bits and pieces that all together define what a programmer has to do to request services and what will be computed as a result. This specification must be detailed and precise, since in the end the program will be interpreted by a dumb literal computer, not by a friendly and accommodating human.

An API includes not only the bare statement of syntactic requirements but also a great deal of supporting documentation to help programmers use the system effectively. Large systems today often involve a *Software Development Kit* or *SDK* so programmers can navigate increasingly complicated software libraries. For example, Apple provides an environment and supporting tools for developers writing iPhone and iPad code; Google provides an analogous SDK for Android phones; Microsoft provides a variety of development environments for writing Windows code in different languages for different devices.

5.3.2 Bugs

Sadly, no substantial program works the first time; life is too complicated and programs reflect that complexity. Programming requires perfect attention to detail, something that few people can achieve. Thus all programs of any size will have errors, that is, they will do the wrong thing or produce the wrong answer under some circumstances. Those flaws are called *bugs*, a term popularly attributed to Grace Hopper, who I mentioned above. In 1947, Hopper's colleagues found a literal bug (a dead moth) in the Harvard Mark II, a mechanical computer that they were working with, and she apparently said that they were "debugging" the machine. The bug was preserved and has passed into a kind of immortality; it can be seen in the Smithsonian's American History museum in Washington and in the photograph on the next page.

Hopper did not coin this use of the word "bug," however; it dates from 1889. As the Oxford English Dictionary (second edition) says,

> **bug.** A defect or fault in a machine, plan, or the like. orig. U.S.
> 1889 Pall Mall Gaz. 11 Mar. 1/1 Mr. Edison, I was informed, had been up the two previous nights discovering 'a bug' in his phonograph—an expression for solving a difficulty, and implying that some imaginary insect has secreted itself inside and is causing all the trouble.

Bugs arise in so many ways that it would take a large book to describe them (and there are such books). Among the endless possibilities are forgetting to handle a case that might occur, writing the wrong logical or arithmetic test to evaluate some condition, using the wrong formula, accessing memory outside the range allotted to the program or a part of it, using the wrong operation for the kind of data, and failing to validate user input.

As a contrived example, here are a pair of JavaScript functions that convert from Celsius to Fahrenheit and vice versa:

```
function ctof(c) {
   return 9/5 * c + 32
}
function ftoc(f) {
   return 5/9 * f - 32;
}
```

One of these functions has an error. Can you see it? I'll come back to it in a moment.

A big part of real-world programming is testing code as it's being written. Software companies often have more testers than programmers, in the hope of identifying as many bugs as possible before software is shipped to users. That's hard, but one can at least get to the state where bugs are encountered infrequently. How would you test the temperature conversion functions above? You would certainly want to try some simple test cases with known answers, for example Celsius temperatures of 0 and 100 where the corresponding Fahrenheit values should be 32 and 212. Those work fine.

But in the other direction, Fahrenheit to Celsius, things don't work so well: the function reports that 32F is −14.2C and 212F is 85.8C, both grossly wrong. The problem is that parentheses are necessary to subtract 32 from the Fahrenheit value before multiplying it by 5/9; the expression in `ftoc` should read

```
    return 5/9 * (f - 32);
```

Fortunately these are easy functions to test, but you can imagine how much work might be involved in testing and debugging a program with a million lines when the failures are not so obvious.

By the way, it's possible to test this pair of functions thoroughly, since they are inverses of each other. If you pass any value through each function in turn, the result should be the original number, except perhaps for a tiny discrepancy caused by the fact that computers don't represent non-integers with perfect precision.

Bugs in software that is accessible through the Internet can leave systems vulnerable to attack, often by permitting an adversary to overwrite memory with his own malicious code. The prevalence of such bugs explains the frequent updates of important programs like browsers, which are the focus of attention for many hackers. For instance, the version of Firefox that I'm using has gone through 19 revisions that have fixed nearly 100 security-related bugs in about 18 months. This is not atypical, and certainly does not mean that Firefox programmers are incompetent; it means that writing robust programs is very hard, and the bad guys are always watching for openings.

Another complexity in real-world software is that the environment changes all the time, and programs have to be adapted. New hardware is developed; it needs new software that may require changes in systems. New laws and other requirements change the logic of programs—think about what has to happen every time a tax law changes, for instance. Computers, tools and languages become obsolete and have to be replaced. Data formats become obsolete—for example, Word files from the early 1990s can't be read by today's version of Word. So do the physical devices that store and process the data. Expertise disappears too, as people retire, die, or get fired in a corporate downsizing. Student-created systems at universities suffer in the same way when the expertise graduates.

Keeping up with steady change is a major problem of software development and maintenance, but it has to be done; otherwise, programs suffer "bit rot," and after a while don't work any more or can't be updated because they can't be recompiled or some necessary library has changed too much. At the same time, the effort to fix or update often creates new bugs or changes to behaviors that users depended on.

5.4 Software as Property

Ownership of software raises difficult legal issues, I think more so than hardware does, though that may be my bias as a programmer. Software is a newer field than hardware; there was no software before about 1950, and it's only in the past 20 or 30 years that software has become a major independent economic force. As a result, there has been less time for mechanisms like laws, commercial practice and social norms to evolve. Software is valuable yet intangible. It takes a great deal of sustained hard work to create and maintain a significant body of code. At the same time, software can be copied in unlimited quantities and distributed worldwide at zero cost,

it's readily modified, and ultimately it's invisible. As a result, issues of ownership seem especially tricky. In this section, I'll discuss some of the problems though without offering any solutions; the hope is to give you enough technical background that you can at least appreciate the situation from multiple viewpoints. I'm also writing from the perspective of US law; other countries have analogous systems but differ in many respects.

5.4.1 Intellectual property

The term *intellectual property* refers to various kinds of intangible property that result from individual creative efforts like inventions or authorship. Several legal mechanisms for protecting intellectual property apply, with varying degrees of success, to software. These include trade secrets, copyrights, patents, and licenses.

Trade secret is the most obvious. The property is kept secret by its owner, or disclosed to others only under a legally binding contract like a non-disclosure agreement. This is simple and often effective, but provides little recourse if the secret is ever revealed. The classic example of a trade secret, in a different domain, is the formula for Coca-Cola. In theory, if the secret became public knowledge, anyone could make an identical product, though they could not call it Coca-Cola or Coke, because those are trademarks, another form of intellectual property.

Copyright protects creative expression. Copyright is familiar in the context of literature, art, music and movies—it protects creative work from being copied by others, at least in theory, and it gives creators the right to exploit their work for a period of time. That period used to be rather limited but is now the lifetime of the author plus 70 years in the US, and only modestly shorter in other countries. (In 2003, the US Supreme Court ruled that 70 years after the author's death is a "limited" term. This is technically correct but practically not very different from "forever.")

Enforcing copyright for digital material is difficult, and countries do not all have identical copyright laws. Physical copies of CDs and DVDs are easy to make, and any number of electronic copies can be made and distributed throughout the online world at no cost. Attempts to protect copyrighted material by encryption and other forms of digital rights management (DRM) have uniformly failed—the encryption usually proves breakable, and if all else fails, material can be re-recorded as it is being played (the "analog hole"). Legal recourse against copyright infringement is hard for individuals and even for large organizations to pursue effectively. I'll return to this topic in Chapter 9.

Copyright also applies to programs. If I write a program, I own it, just as if I had written a novel. No one else can use my copyrighted program without my permission. That sounds simple enough, but as always the devil is in the details. If you study my program's behavior and create your own version, how similar can it be without violating copyright? If you change the formatting and the names of all the variables in the program, that's still a violation, but beyond that, it's not obvious, and the issues can only be settled by an expensive legal process. If you study my program, understand it thoroughly, and then do a genuinely new implementation, that's

likely to be ok. In fact, in the technique called "clean room development," programmers explicitly have no access to or knowledge of the code whose properties they are trying to replicate. They write new code that behaves the same way as the original but which has demonstrably not been copied. Then the legal question becomes one of proving that the clean room really was clean and no one was tainted by exposure to the original code.

Patents provide legal protection for inventions. This contrasts with copyrights, which protect only expression—how the code is written—and not any original ideas it might contain. There are plenty of hardware patents like the cotton gin, the telephone, the transistor, the laser, and of course myriad processes, devices, and improvements on them.

Originally, software was not patentable, since it was thought to be "mathematics" and thus not within the scope of patent law. This thinking has evolved since the 1960s to a complicated mess today, where many observers (including me) think that the patent system for software and related areas like business methods is grievously broken. Whether algorithms and programs should be patentable is unclear and very much in flux. One position against such patents is that algorithms are merely mathematics, which the law says is not patentable (not "statutory" material). That seems wrong to me as a programmer with a modest background in mathematics—algorithms are not mathematics, though they might involve mathematics. (Think about Quicksort, which today might well be patentable.) Another position is that many software patents are obvious, no more than using a computer to do some straightforward or well-known process, and thus should not be patented because they lack originality. I'm much more in sympathy with that position, though again as a non-expert and certainly not a lawyer.

The poster child for software patents might be Amazon's "1-click" patent. In September 1999, US patent 5,960,411 was granted to four inventors at Amazon.com, including Jeff Bezos, the founder and CEO. The patent covers "A method and system for placing an order to purchase an item via the Internet"; the claimed innovation was allowing a registered customer to place an order with a single mouse click. The patent has been the subject of debate and legal struggle ever since. It's probably fair to say that most programmers think the idea is obvious, but the law requires that an invention be "unobvious" to "a person having ordinary skill in the art" at the time of invention, which was 1997, in the early days of web commerce. The US Patent Office has denied some claims of the patent; others have been sustained upon appeal. In the meantime, the patent has been licensed by other companies, including Apple for its iTunes online store, and Amazon has obtained injunctions against companies using 1-click without permission.

Licenses are legal agreements granting permission to use a product. Everyone is familiar with one step in the process of installing a new version of some software: the "End User License Agreement" or EULA. A dialog box shows a small window on an enormous amount of tiny type, a legal document whose terms you must agree to before you can go further. Most people just click to get past the roadblock, and thus

are in principle and probably in practice legally bound by the terms of the agreement.

If you do read those terms, it won't be a big surprise to discover that they are one-sided. The supplier disclaims all warranties and liability, and in fact doesn't even promise that the software will do anything. To quote from the EULA for the iTunes App Store,

> "APPLICATION PROVIDER DOES NOT WARRANT AGAINST INTERFERENCE WITH YOUR ENJOYMENT OF THE LICENSED APPLICATION, THAT THE FUNCTIONS CONTAINED IN, OR SERVICES PERFORMED OR PROVIDED BY, THE LICENSED APPLICATION WILL MEET YOUR REQUIREMENTS, THAT THE OPERATION OF THE LICENSED APPLICATION OR SERVICES WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN THE LICENSED APPLICATION OR SERVICES WILL BE CORRECTED."

You can't sue for damages if the software does you harm. There are conditions on what the software can be used for, and you agree that you won't try to reverse-engineer or disassemble it. You can't ship it to certain countries. My lawyer friends say that such licenses are generally valid and enforceable if the terms are not too unreasonable, which seems to beg the question.

Another clause may come as a bit of a surprise, especially if you have purchased the software in a physical or online store: "This software is licensed, not sold." For most purchases, a legal doctrine called "first sale" says that once you have bought something, you own it. If you buy a book, it's your copy and you can give it away or resell it to someone else, though you can't violate the author's copyright by making and distributing copies. But suppliers of digital goods almost always "sell" them under a license that lets the supplier retain rights and restrict what you can do with "your" copy.

A great example of this surfaced in July 2009. Amazon "sells" lots of books for its Kindle e-book readers, but in fact the books are licensed, not sold. At one point, Amazon realized that it was distributing some books that it did not have permission for, so it "unsold" them by disabling them on all Kindles. In a wonderful ironic coincidence, one of the recalled books was an edition of George Orwell's dystopian novel *1984*. I'm sure he would have loved the Kindle story.

APIs also raise some legal questions, mostly focused on copyrights. Suppose I'm the manufacturer of a programmable game system, analogous to the Xbox or PlayStation. I want people to buy my game machine, and that will be more likely if there are lots of good games for it. I can't possibly write all that software myself, so I carefully define an API—an application programming interface—so programmers can write games for my machine. I might also provide a software development kit, analogous to Microsoft's XDK for the Xbox, to help game developers. With luck, I will sell a bunch of machines, make a pile of money, and retire happy.

An API is in effect a contract between the service user and the service provider. It defines what happens on both sides of the interface—not the details of how it's implemented, but definitely what each function does when used in a program. That

means that someone else could also play the provider side, by building a competing game machine that provides the same API as mine. If that were done well—everything works the same—and if the competitor's machine were better in other ways, like a lower price and sexier physical design, it could drive me out of business. That's bad news for my hopes of becoming wealthy.

What are my legal rights? I defined the API, so I should be able to protect it by copyright, allowing others to use it only if they have licensed the rights from me; the same is likely to be the case if I provide an SDK. Is that sufficient protection? This legal question, and a variety of others like it, is not really settled, and I could argue either side of it depending on what point I want to make. In the meantime, real companies are engaged in real arguments (using real lawyers and spending enormous amounts of real money) to try to resolve it. I've been an occasional consultant trying to guide lawyers on one side or the other, and have found it enjoyable and instructive, an example of the kind of explanation of technical material about computing that makes up this book. But it must be frustrating and irritating for parties on both sides of such disputes, who have an immediate stake in the outcome.

5.4.2 Standards

A *standard* is a precise and detailed description of how some artifact is built or is supposed to work. Examples of software standards include programming languages (the definition of syntax and semantics), data formats (how information is represented), algorithmic processing (the steps necessary to do a computation), and the like.

Some standards, like the Word `.doc` file format, are *de facto* standards—they have no official standing but everyone uses them. The word "standard" is best reserved for formal descriptions, often developed and maintained by a quasi-neutral party like a government or a consortium, that define how something is built or operated. The definition is sufficiently complete and precise that separate entities can interact or provide independent implementations. We benefit from hardware standards all the time, though we may not notice how many there are. If I buy a new television set, I can plug it into the electrical outlets in my home thanks to standards for the size and shape of plugs and the voltage they provide. The set itself will receive signals and display pictures because of standards for broadcast and cable television. I can plug other devices into it through standard cables and connectors like HDMI, USB, S-video and so on. But every TV needs its own remote control and every cell phone needs a different charger because those have not been standardized.

Computing has plenty of standards as well, including character sets like ASCII and Unicode, programming languages like C and C++, algorithms for encryption and compression, and protocols for exchanging information over networks. There are sometimes even competing standards, which seems counter-productive. (As computer scientist Andy Tanenbaum once said, "The nice thing about standards is that you have so many to choose from.") Historical examples include Betamax versus VHS for videotape, and, more recently, HD-DVD versus Blu-ray for high-definition

video disks. In both of these cases, one standard eventually won out, but in other cases multiple standards may co-exist, like the two incompatible cell phone technologies in the USA.

Standards are important. They make it possible for independently created things to cooperate, and they open an area to competition from multiple suppliers, while proprietary systems tend to lock everyone in. (Naturally the owners of proprietary systems prefer lock-in.) Standards have disadvantages too—a standard can impede progress if it is inferior or outdated yet everyone is forced to use it. But these are modest drawbacks compared to the advantages.

5.4.3 Open source

The code that a programmer writes, whether in assembly language or (much more likely) in a high-level language, is called *source code*. The result of compiling it into a form suitable for a processor to execute is called *object code*. This distinction, like several others I've made, might seem pedantic but it's important. Source code is readable by programmers, though perhaps with some effort, so it can be studied and adapted, and any innovations or ideas it contains are visible. By contrast, object code has gone through so much transformation that it is usually impossible to recover anything remotely like the original source code or to extract any form that can be used for making variants or even understanding how it works. It is for this reason that most commercial software is distributed only in object-code form; the source code is a valuable secret and is kept metaphorically and perhaps literally under lock and key.

Open source refers to an alternative in which the source code is freely available for study and improvement.

In early times, most software was developed by companies and most source code was unavailable, a trade secret of whoever developed it. Richard Stallman, a programmer working at MIT, was frustrated that he could not fix or enhance the programs that he was using because their source code was proprietary and thus inaccessible to him. In 1983, Stallman started a project that he called *GNU* ("GNU's Not Unix", `gnu.org`) to create free and open versions of important software systems, like an operating system and compilers for programming languages. He also formed a non-profit organization called the Free Software Foundation to support open source. The goal was to produce software that was perpetually "free" in the sense of being non-proprietary and unencumbered by restrictive ownership. This was accomplished by distributing implementations under a clever copyright license called the GNU General Public License or GPL.

The Preamble to the GPL says "The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users." The GPL specifies that the licensed software can be freely used, but if it is distributed to anyone else the distribution must make the source code available with the same "free for any use" license. The GPL is strong enough that companies

that have violated its terms have been forced to stop their use of the code or to distribute source that they based on licensed code.

The GNU project, supported by many companies, organizations, and individuals, has produced a large collection of program development tools and applications, all covered by the GPL; other open source programs have analogous licenses. In many cases, the open source versions set the standard against which proprietary commercial versions are measured. The Firefox and Chrome browsers are open source; so is the Apache web server that operates the majority of web servers; so is the Android operating system for cell phones. There are open source compilers for all major programming languages and open source versions of most major programmer tools (including the ones I used to produce this book).

The Linux operating system is perhaps the most visible open source system (though it is not part of the GNU project); it is widely used by individuals and large commercial enterprises like Google, which runs its entire infrastructure on Linux. If you want the Linux kernel source code, you can download it for free from the site `kernel.org`; you can use it for your own purposes and make any modifications you like. But if you distribute it in any form, for example, in a new gadget that has an operating system, you must make your source code available under the same GPL.

Open source is intriguing. How can one make money by giving software away? Why do programmers voluntarily contribute to open source projects? Can open source written by volunteers be better than proprietary software developed by large teams of coordinated professionals? Is the availability of source code a threat to national security?

These questions continue to intrigue economists and sociologists, but some answers are becoming clearer. For example, Red Hat is a public company traded on the New York Stock Exchange; in 2011 it had annual revenue of nearly $1 billion and a market capitalization of over $8 billion. It distributes Linux source code that you can get for free on the web but makes money by charging for support, integration, and other services. Some open source programmers are regular employees of companies that use open source and contribute to it; IBM is a notable example but certainly not unique. The companies profit from being able to help guide the evolution of programs and from having others fix bugs and make enhancements.

Not all open source software is best of breed, and it's common for the open source versions of some software to lag behind the commercial systems that are being copied. But for core programmer tools and systems, open source is hard to beat.

6

Software Systems

In this chapter, we're going to look at two major kinds of software: operating systems and applications. As we'll see, an *operating system* is the software underpinning that manages the hardware of a computer and makes it possible to run other programs, which are called *applications*.

When you use a computer, say a personal computer at home, school or office, you have a wide variety of programs available, including browsers, word processors, a music player, something for watching DVDs, one for taxes (sigh), a virus scanner, plenty of games, and tools for mundane tasks like searching for files or looking at folders.

The jargon term for all of these programs is *application*, presumably from "this program is an application of a computer to some task." It's clumsy but standard terminology for programs that are more or less self-contained and focused on a single task. The word used to be the province of computer programmers, but with the runaway success of Apple's App Store, which sells applications to run on iPhones, the abbreviated form *app* has become part of popular culture.

When you buy a new computer, it usually comes with a large number of such programs already installed, and more are added over time, as you buy them or download them from the web. Apps in this sense are important to us as users, and they have interesting properties from multiple technical perspectives. I'm going to talk briefly about several examples, then focus on a specific one, the browser. A browser is a representative example that everyone is familiar with, but it still holds some surprises, including an unexpected parallel with operating systems.

Let's begin, however, with the program that makes it possible to use applications: the operating system.

6.1 Operating Systems

In the early 1950s, there was no distinction between application and operating system. Computers were so limited that they could only run one program at a time, and that program took over the whole machine; indeed, programmers had to sign up for a time slot to use the computer (in the middle of the night if one were a lowly student) to run their one program. As computers became more sophisticated, it was too inefficient to have amateurs running them; the job was turned over to professional operators, who fed applications in and distributed the results. Operating systems began as programs to help automate those tasks for the human operators.

Operating systems steadily became more elaborate, matching the evolution of the hardware that they controlled, and as hardware became more powerful and complicated, it made sense to devote more resources to controlling it. The first widely used operating systems appeared in the late 1950s and early 1960s, usually provided by the same company that made the hardware, so there were operating systems from IBM and Univac, and later from smaller computer companies like Digital Equipment and Data General.

Operating systems were also objects of research at universities and industrial labs. MIT was a pioneer, creating in 1961 a system called CTSS ("Compatible Time-Sharing System") that was especially advanced for its time, and, unlike its industrial competitors, a pleasure to use. The Unix operating system, which was created at Bell Labs by Ken Thompson and Dennis Ritchie starting in 1969, profited from their first-hand experience with CTSS and its more elaborate but less successful child Multics. Today most operating systems, except those from Microsoft, are either descended from the original Bell Labs Unix or the compatible but independent Linux. Ritchie and Thompson shared the 1983 Turing Award for their creation of Unix.

A modern computer is a complicated beast indeed. It has many parts—processor, memory, disk, display, network interfaces, and on and on and on. To use these components effectively, it needs to run multiple programs at the same time, some of which are waiting for something to happen (a web page to download), some of which are demanding an immediate response (tracking the mouse as it moves or updating the display as you play a game), and some of which are interfering with the others (starting up a new program that needs space in the already over-crowded RAM). It's a mess.

The only way to manage this elaborate juggling act is to use a program, another example of making the computer help with its own operation. That program is called an operating system. For computers at home or work, Microsoft Windows, in its various evolutionary stages, is the most common operating system; it runs perhaps 90 percent of the computers one sees in day-to-day life. Apple computers run Mac OS X, a Unix variant, which at the consumer level is next most popular. Many behind-the-scenes computers (and some foreground ones as well) run Unix or Linux. Cell phones run operating systems too, smaller versions of Windows, Unix, Linux or more specialized systems. For instance, iPhones and iPads run iOS, an operating system derived from Mac OS X, while my Android cell phone runs Linux, as does my

television, the TiVo, Amazon's Kindle, and Barnes & Noble's Nook. I can even log in to my Android phone and run standard Unix commands there.

An operating system controls and allocates the resources of a computer. First, it manages the CPU, scheduling and coordinating the programs that are currently in use. It switches the CPU's attention among the programs that are actually computing at any given moment, both applications and background processes like anti-virus software and checking for updates. It suspends programs that are waiting for an event, for example a dialog box waiting for a user to click on it. It keeps individual programs from hogging resources—if one program demands too much CPU time, the operating system throttles it back so other tasks get a reasonable share as well.

A typical operating system will have several dozen processes or tasks in simultaneous operation. Some are programs started by users, though most are system tasks that are invisible to the casual user. You can see what's going on with programs like Activity Monitor on Mac OS X and Task Manager on Windows. Here's a small part of what Activity Monitor displays for the Mac where I am currently typing:

Process Name	% CPU	Real Mem	Rcvd Msgs	Sent Msgs
Firefox	0.8	176.7 MB	45,123,699	90,909,408
Terminal	0.0	153.1 MB	3,425,882	7,252,889
Preview	0.0	136.8 MB	2,440,518	5,298,824
kernel_task	5.7	125.2 MB	257,493,122	268,772,970
Google Chrome Renderer	0.3	87.8 MB	4,187,136	7,709,353
Microsoft Word	0.6	84.6 MB	882,789	1,731,595
WindowServer	2.0	76.4 MB	4,233,362	14,035,897
mds	0.9	63.5 MB	110,776,448	200,043,018
Google Chrome	0.2	36.0 MB	1,108,402	2,321,157
Google Chrome Renderer	0.4	34.9 MB	4,315,431	7,998,707
Activity Monitor	0.4	32.9 MB	205,821	475,771
mdworker	6.3	32.6 MB	1,033,421	4,108,493
Finder	0.0	27.6 MB	2,062,267	4,020,002
Google Chrome Renderer	0.3	24.8 MB	3,264,641	5,974,244
devdraw	0.1	17.9 MB	235,064	855,888
SystemUIServer	0.1	17.3 MB	1,320,592	2,631,674
Shockwave Flash (Chro...	0.3	15.0 MB	21,744,080	48,381,952
mdworker	0.0	11.9 MB	209,238	814,472
Grab	6.7	11.7 MB	2,976	6,938

In this display, processes are sorted by the amount of RAM they are using; Firefox is at the top, but you can see that several Chrome processes together add up to about the same amount of memory. No process is using very much of the CPU.

Second, the operating system manages RAM. It loads programs into memory so they can begin executing instructions. It moves them out to the disk temporarily if there isn't enough RAM for everything that's happening at the same time, then moves them back in when there's room again. It keeps separate programs from interfering with each other so one program can't access the memory allocated to another program or to the operating system itself. This is partly to maintain sanity but it's also a safety measure: one doesn't want a rogue or buggy program poking around where it shouldn't be. (The "blue screen of death" that used to be a common sight in Windows was sometimes caused by a failure to do this protection adequately.)

It takes good engineering to make effective use of RAM. One technique is to bring only part of a program into RAM when needed and copy it back out to disk when inactive, a process called *swapping*. Programs are written as if they had the entire computer to themselves and unlimited RAM. This makes programming significantly easier. The operating system then has to support the illusion by swapping chunks of program in and out, with help from the hardware to translate the memory addresses that the program thinks it is accessing into real addresses in real memory. This mechanism is called *virtual memory*. Like most uses of the word "virtual," it means giving the illusion of reality but not the real thing.

Third, the operating system manages information stored on disks. A major component of the operating system called the *file system* provides the familiar hierarchy of folders and files that we see when we use a computer. We'll come back to file systems later in this chapter, since they have enough interesting properties to warrant a more extended discussion.

Finally, the operating system manages and coordinates the activities of the devices connected to the computer. It maintains multiple windows on the display, making sure that the right information gets to the right window, and that it's properly restored when the window is moved, resized, or hidden and re-exposed. It directs input from the keyboard and mouse to the program that's expecting it. It handles traffic to and from network connections, whether wired or wireless. It sends data to printers and DVD burners and it fetches data from scanners.

Notice that I said that an operating system is a program. It's just another program, like the ones in the last chapter, written in the same kinds of languages, most often C and C++. Early operating systems were small, since the job was simpler. The earliest operating systems only ran one program at a time, so only limited swapping was needed. There wasn't a lot of memory to allocate (a few hundred kilobytes at most) so memory management was straightforward. They didn't have many external devices to deal with, certainly not the rich variety of today. Operating systems are now very large—millions of lines of code—and complicated because they are doing a complicated task.

For calibration, the 6th edition of the Unix operating system, which is the ancestor of many systems today (though not Windows), was 9,000 lines of C and assembler in 1975, and was written by two people, Ken Thompson and Dennis Ritchie. Today Windows 7 has perhaps 100 million lines of code and Linux has well over 10 million lines; each system is the work of thousands of people over decades. Of course these aren't directly comparable, since modern computers are much more complicated and deal with much more complicated environments and far more devices; there are also differences in what's included in the operating system.

Since an operating system is just a program, you can in principle write your own. Indeed, Linux began when Linus Torvalds, a Finnish college student, decided to write his own version of Unix from scratch in 1991. He posted an early draft on the Internet and invited others to try it and help out. Since then, Linux has become a major force in the software industry, used by many large companies and smaller players. As noted in the last chapter, Linux is open source, so anyone can use it and contribute. Today there are many thousands of contributors, with a core of full-time developers; Torvalds maintains overall control and is the ultimate arbiter.

You can run a different operating system on your hardware than might have been originally intended—running Linux on Windows computers is a good example. You can store several operating systems on disk and determine which one to run each time you turn the computer on. This "multiple boot" feature can be seen with Apple's Boot Camp, which makes it possible to start up a Mac running Windows instead of Mac OS X.

You can even run one operating system under the control of another, as a *virtual operating system*. Virtual operating system programs like VMware, Parallels and Xen (which is open source) make it possible to run one operating system, say Windows or Linux, as a guest operating system on a host, say Mac OS X. The host operating system intercepts requests made by the guest that would require operating system privileges, such as file system or network access. The host does the operation and then returns to the guest. When the host and guest are both compiled for the same hardware, the guest system runs at the full hardware speed for the most part, and feels as responsive as it would if it were on the bare machine.

This diagram shows schematically how a virtual operating system runs on a host operating system; the guest operating system is an ordinary application as far as the host operating system is concerned.

Virtual operating systems raise some interesting ownership questions. If a company runs a large number of virtual Windows instances on one physical computer,

how many Windows licenses does it need to buy from Microsoft? Technically, the answer is one, but Microsoft's licenses for Windows limit the total number of virtual instances that you can legitimately run without paying for more copies.

Another use of the word "virtual" should be mentioned here. A program that simulates a computer, whether a real one or a pretend one (like the Toy), is also often called a *virtual machine*. That is, the computer exists only as software, a program that mimics its behavior as if it were hardware. Such virtual machines are common. Browsers have a virtual machine to interpret JavaScript programs, and a separate virtual machine for Java programs as well. There is also a Java virtual machine in Android cell phones.

6.2 How an Operating System Works

The CPU is constructed so that when the computer is powered on, the CPU starts by executing a few instructions stored in a permanent memory. Those instructions in turn read instructions from a small flash memory that contains enough code to handle minimal devices, which in turn read more instructions from a known place on a disk, CD, USB memory, or network connection, which in turn read still more instructions, until finally enough code has been loaded to do useful work. This getting-started process is called *booting*, after the old image of pulling oneself up by one's bootstraps. The details vary, but the basic idea is the same—a few instructions are sufficient to find more, which in turn find still more.

Part of this process may involve querying the hardware to find out what devices are attached to the computer, for example whether there is a printer or a wireless device. Memory and other components are checked to verify that they are working correctly. The boot process may involve loading software components (drivers) for connected devices so the operating system can use them. All of this takes time, usually while we're waiting impatiently for the computer to start doing something useful. It's frustrating that although computers are far faster than they used to be, it still takes a minute or two to boot.

Once the operating system is running, it settles down to a fairly simple cycle, giving control in turn to each application that is ready to run or that needs attention. If I am typing this text in a word processor, checking my mail, surfing randomly, and playing music in the background, the operating system gives the CPU's attention to each of these processes in turn, switching the focus among them as necessary. Each program gets a short slice of time, which ends when the program requests a system service or its allocated time runs out.

The system responds to events like the end of the music, the arrival of mail or a web page, or a key-press; for each, it does whatever is necessary, often relaying the fact that something has happened to the application that must take care of it. If I decide to rearrange the windows on my screen, the operating system tells the display where to put the windows, and it tells each application what parts of that application's window are now visible so the application can redraw them. If I quit an

application with File / Exit or by hitting the little × in the corner of the window, the system notifies the application that it's about to die, so it has a chance to put its affairs in order (for instance by asking the user "Do you want to save the file?"). The operating system then reclaims any resources that the program was using, and tells the apps whose windows are now exposed that they have to redraw.

6.2.1 System calls

An operating system provides an interface between the hardware and other software. It makes the hardware appear to provide higher-level services than it really does, which makes programming easier. In the jargon of the field, the operating system provides a *platform* upon which applications can be built.

The operating system defines a set of operations or services that it offers to application programs, like storing data in a file or retrieving data from a file, making network connections, fetching whatever has been typed on the keyboard, reporting mouse movements and button clicks, drawing on the display, and so on.

The operating system makes these services available in a standardized or agreed-upon fashion, and an application program requests them by executing a particular instruction that transfers control to a specific place in the operating system. The system does whatever the request implies, and returns control and the results to the application. These entry points into the system are called *system calls*, and their detailed specification really defines what the operating system is. A system call could be implemented as a direct entry to code within the operating system or as a call to a library function that arranges for the service, but for the most part that distinction isn't relevant even for programmers. For this reason, it's also hard to say how many system calls there are, though it's typically a hundred or two.

6.2.2 Device drivers

One other operating system component is worth a brief section of its own. A *device driver* is a program that acts as a bridge between the operating system and a specific kind of hardware device like a printer or a mouse. A driver contains code that has detailed knowledge of how to make a particular device do whatever it does—how to access motion and button information from a specific mouse, make a disk read and write information on a spinning magnetic surface, make a printer put marks on paper, or make a specific wireless card send and receive radio signals.

Consider a printer. The operating system wants to make standard requests: print this text at this position on the page, draw this image, move to the next page, describe your capabilities, report your status, and so on, in a uniform way that would apply to any printer. Printers differ in their capabilities, however, for example whether they support color, two-sided printing, multiple paper sizes, and the like. The driver for a given printer is responsible for dealing with converting the operating system requests into whatever is needed to make the particular device do those tasks. In effect, the operating system makes generic requests, and the device driver implements them for its own hardware.

The driver insulates the rest of the system from the idiosyncrasies of particular devices—all devices of one kind, like keyboards, have basic properties and operations that the operating system cares about—and the driver interface lets the operating system access the device in a uniform way so it's easy to switch devices.

A general-purpose operating system will have many drivers; for example, Windows is shipped with drivers already installed for an enormous variety of devices that might be potentially used by consumers, and every device manufacturer has a web site from which new and updated drivers can be downloaded.

Part of the boot process is to load drivers for currently available devices into the running system; the more devices there are, the more time this will take. It's also normal for new devices to appear out of nowhere. When an external disk is plugged into a USB socket, Windows recognizes a new device, determines (a part of the device driver interface) that it is a disk, and loads a USB disk driver to communicate with the disk. Mac OS X does the same thing. Normally there's no need to find a new driver; the whole thing is so standardized that the operating system already has what it needs and the specifics of driving the device are buried in a processor within the device itself.

This diagram illustrates the relationships among the operating system, system calls, drivers, and applications.

6.3 Other Operating Systems

One of the results of ever cheaper and smaller electronics is that it becomes feasible to include more of that hardware in a device; as a result many devices have significant processing power and memory. Calling a digital camera "a computer with a lens" is not far off the mark. As processing power and memory have increased, cameras have ever more capabilities. Cell phones are another great example, and of course cameras and cell phones are converging; any random cell phone today has more megapixels than my first digital camera did, though the lens quality is another story.

The overall result is that many devices are taking on the trappings of mainstream general-purpose computers. They have a powerful processor, a lot of memory, and some peripheral devices (think lens and display on the camera). They have sophisticated user interfaces. They may have a network connection so they can talk to other systems—cell phones use the telephone network and Wi-Fi, game controllers use infrared and Bluetooth—and many use USB for occasional ad hoc connections.

As this trend continues, it increasingly makes more sense to run a commodity operating system than to write one's own. Unless the environment is quite unusual, it's likely to be cheaper to use a stripped-down version of Linux, which is robust, adaptable, portable and free, rather than trying to develop one's own specialized system or licensing a commercial product. A drawback is that one has to release the operating system parts of the resulting code under the GPL, which may raise issues of how to protect intellectual property in the device, but this has not proven insurmountable for the likes of Kindle and TiVo.

6.4 File Systems

The file system is the part of the operating system that makes physical storage media like disks, CDs and DVDs, removable memory devices, and other gadgets look like hierarchies of files and folders. The file system is a great example of the distinction between logical organization and physical implementation; file systems organize and store information on many different kinds of devices, but the operating system presents the same interface for all of them. The ways that file systems store information and for how long turn out to have legal ramifications, so another reason for studying file systems is to understand why "removing a file" does not mean that its contents are gone forever.

Everyone has used Windows Explorer or Finder on Mac OS X, which show the hierarchy starting from the top (the C: drive on Windows, for example). A *folder* contains the names of other folders and files; examining a folder will reveal more folders and files. (Unix systems traditionally use the word *directory* instead of folder.) The folders provide the organizational structure, while the files hold the actual contents of documents, pictures, music, spreadsheets, web pages, and so on. All the information that your computer holds is stored in the file system and is accessible through it if you poke around. This includes not only your data, but the executable forms of programs (a browser, for example), libraries, device drivers, and the files that make up the operating system itself. There's an astonishing amount; I was amazed that my modest Macbook has over 900,000 files.

The file system manages all this information, making it accessible for reading and writing by applications and the rest of the operating system. It coordinates accesses so they are performed efficiently and don't interfere with each other, it keeps track of where data is physically located, and it ensures that the pieces are kept separate so that parts of your email don't mysteriously wind up in your spreadsheets or tax returns. On systems that support multiple users, it enforces information privacy and

security, making it impossible for one user to access another user's files without permission, and it may impose quotas on the amount of space that each user can use.

File system services are available through system calls at the lowest level, usually supplemented by libraries to make common operations easy to program.

6.4.1 Disk file systems

The file system is a wonderful example of how a wide variety of physical systems can be made to present a uniform logical appearance, a hierarchy of folders and files. How does it work?

A 100 GB hard disk holds 100 billion bytes, but software on the disk itself is likely to present this as something like 100 million chunks or *blocks* of 1,000 bytes each. (In real life, these sizes would be powers of 2, and larger; I'm using decimal numbers to make it easier to see relationships.) A file of say 2,500 bytes, like a small mail message, would be stored in three of these blocks; it's too big for two blocks but three is enough. File systems don't store bytes of one file in the same block as bytes of a different file, so there's always some waste because the last block is not completely full (500 bytes are unused in the last block in our example). That's a modest price for a considerable simplification in bookkeeping effort, especially since disk memory is so cheap.

A folder entry for this file would contain its name, its size of 2,500 bytes, the date and time it was created or changed, and other miscellaneous facts about it (permissions, type, etc., depending on the operating system). All of that information is visible through a program like Explorer or Finder.

The folder entry also contains information about where the file is stored on disk—which of the 100 million blocks contain its bytes. There are many different ways to manage that location information. The folder entry could contain a list of block numbers; it could refer to a block that itself contains a list of block numbers; or it could contain the number of the first block, which in turn gives the second block, and so on. This diagram sketches an organization with blocks that refer to lists of blocks:

Blocks need not be physically adjacent on disk, and in fact they typically won't be, at least for large files. A megabyte file will occupy a thousand blocks, and those are likely to be scattered to some degree. The folders and the block lists are themselves stored in blocks, though that's not shown in this diagram.

A folder is a file that contains information about where folders and files are located. Because information about file contents and organization must be perfectly accurate and consistent, the file system reserves to itself the right to manage and maintain the contents of folders. Users and application programs can only change folder contents implicitly, by making requests of the file system.

In fact, folders *are* files; there's no difference in how they are stored except that the file system is totally responsible for folder contents, and application programs have no direct way to change them. But otherwise, it's just blocks on the disk, all managed by the same mechanisms.

When a program wants to access an existing file, the file system has to search for the file starting at the root of the file system hierarchy, looking for each component of the file path name in the corresponding folder. That is, if the file is `/Users/bwk/book/book.txt` on a Mac, the file system will search the root of the file system for `Users`, then search within that folder for `bwk`, then within that folder for `book`, then within that for `book.txt`. On Windows, the name might be `C:\My Documents\book\book.txt`, and the search would be analogous. This is a divide-and-conquer strategy, since each component of the path narrows the search to files and folders that lie within that folder; all others are eliminated. Thus multiple files can have the same name for some component; the only requirement is that the full path name be unique. In practice, programs and the operating system keep track of the folder that is currently in use so searches need not start from the root each time, and the system is likely to cache frequently-used folders to speed up operations.

When a program wants to create a new file, it makes a request of the file system, which puts a new entry in the appropriate folder, including name, date, and so on, and a size of zero (since no blocks are allocated to the brand new file yet). When the program follows up by asking to write some data into the file, say inserting the text of a mail message, the file system finds enough currently unused or "free" blocks to hold the requested information, copies the data into them, inserts them into the folder's list of blocks, and returns to the application.

This suggests that the file system maintains a list of all the blocks on the disk that are not currently in use, that is, not already part of some file. When a request for a new block arrives, it can be satisfied with a block taken from the list of free blocks. The free list is kept in file system blocks too, but is only accessible to the operating system, not to application programs.

6.4.2 Removing files

When a file is removed, the opposite happens: the blocks of the file are returned to the free list and the folder entry for the file can be cleared, so the file appears to have gone away.

In reality, it's not quite like that, and therein lie a number of interesting implications.

When a file is removed in Windows and Mac OS X it goes to the "Recycle Bin" or "Trash," which appears to be just another folder, albeit with slightly different properties. Indeed, that's exactly what the recycle bin is. When a file is to be removed, its folder entry is copied from the current folder to a folder called Recycle or Trash, and the original folder entry is cleared. The blocks of the file and thus its contents do not change at all! Recovering a file from the recycle bin reverses the process, restoring the entry to its original folder.

"Emptying the trash" is more like what we originally described, where the Recycle or Trash folder entry is cleared and the blocks are really added to the free list. This is true whether the emptying is done explicitly or happens quietly behind your back because the file system knows it's running low on free space.

Suppose you empty the trash explicitly. That clears the entries in the recycle folder itself and puts the blocks on the free list, but their contents haven't yet been deleted—all the bytes of each block of the original files are still sitting there untouched. They won't be overwritten with new content until that block is taken off the free list and given to some application.

This means that information that you thought was removed still exists and is readily accessible if one knows how to find it. Any program that can read the disk by blocks, that is, without going through the file system, can see what the old content was.

This has potential benefits. If something goes wrong with your disk, it might still be possible to recover information even if the file system is confused. But there's no guarantee that the data is really gone, which is bad if you truly want the information removed, perhaps because it's private or you are plotting some evil deed. A diligent bad guy or law enforcement agency would have no trouble recovering it, so if you're planning something nefarious, or if you're just paranoid, you have to use a program that will erase the information from freed blocks, like the "secure erase" option on a Mac that overwrites the contents with random bits before freeing the blocks.

In practice you may have to do better than this, since a truly dedicated adversary with lots of resources might be able to extract traces of information even when new information has over-written it. Military-grade file erasing overwrites blocks multiple times with random patterns of 1s and 0s. Even better is to demagnetize the whole disk by putting it near a strong magnet. Best of all is to physically destroy it; that's the only way to ensure that the contents are gone. Even that might not be enough, however, if your disk is being backed up automatically all the time (as mine is at work) or if your files are kept on a network file system rather than on your own drive.

A somewhat similar situation applies to the folder entry itself. When you remove a file, the file system will note that the folder entry no longer points to a valid file. It could do that simply by setting a bit in the folder that means "this entry is not in use." Then it would be possible to recover the original information about the file, including the contents of any blocks that had not been re-allocated, until the folder entry was

itself re-used. This mechanism was at the heart of some file recovery systems for Microsoft's MS-DOS system in the 1980s, which marked free entries by setting the first character of the filename to a special value. This made it easy to recover the whole file if the recovery was attempted soon enough.

The fact that the contents of files can survive long after their creator thought them deleted has some implications for legal procedures like discovery and document retention. It's remarkably common, for example, for old email messages to surface that are in some way incriminating or at least embarrassing. If such records exist only on paper, there's a decent chance that careful shredding will destroy all copies, but digital records can proliferate, are easily copied onto removable devices, and can be tucked away in many places. (The collection of classified diplomatic documents that Wikileaks obtained in 2010 was apparently on CDs.) It's wise to be cautious about what you say in mail and indeed in any information that you commit to a computer.

6.4.3 Other file systems

I've been discussing conventional file systems on hard drives, including removable ones, since that's where most of the information is and that's what we most often see on our own computers. But the file system abstraction applies to other media as well.

For example, floppy disks, a now-vanished technology, provided the same logical view, though the details are different. CD-ROMs and DVDs also provide access to their information as if it were a file system, again with a hierarchy of folders and files, but generally with read-only access. Solid state disks use flash memory to simulate conventional hard drives while weighing less and using less power.

Flash memory file systems on USB and SD ("Secure Digital") are ubiquitous. When plugged into a Windows computer, a flash drive appears as another disk drive. It can be explored with Explorer and files can be read and written exactly as if it were a hard disk. The only difference is that its capacity is smaller, and maybe access is somewhat slower.

If that same device is plugged into a Mac, it appears there as a file system as well, to be explored with Finder, and files can be transferred back and forth. It can also be plugged into Unix or Linux computers, and again appears in the file system on those systems. Software makes the physical device look like a file system, with the same abstractions of folders and files, on a variety of operating systems. Internally, the organization is probably a Microsoft FAT file system and everyone else emulates it,

but we don't know for sure and we don't need to. The abstraction is perfect. (By the way, "FAT" stands for File Allocation Table; it's not a commentary on implementation quality.)

My first digital camera stored pictures in an internal file system and I had to connect the camera to a computer to retrieve them. My current camera has a removable SD memory card (shown above) that plugs into most computers and I can upload pictures by moving the card from camera to computer. This is much faster than it was before, and as an unexpected fringe benefit, it has freed me completely from the camera manufacturer's appallingly awkward and flaky software. A familiar and uniform interface replaces a clumsy and unique one. One could imagine that the manufacturer is also happy that it's no longer necessary to provide specialized file transfer software.

It's worth mentioning one other version of the same idea: the network file system, which is common in schools and businesses. By software, one can access the file system on some other computer as if it were on one's own machine, again using Explorer, Finder or other programs to access the information. The file system at the far end may be of the same kind (both Windows computers, for example) or it might be something different, say a Mac or Linux, but as with flash memory devices, the software hides the differences and presents a uniform interface so it looks like a regular file system on the local machine.

Network file systems are often used as backup as well as primary file storage. Multiple older copies of files can be copied onto archival media for storage at a different site; this protects against a disaster like a fire that destroys the only copy of vital records. Some disk systems also rely on a technique called RAID ("redundant array of independent disks") that writes data onto multiple disks with an error-correction code that makes it possible to recover information even if one of the disks breaks. Naturally such systems also make it hard to ensure that all traces of information have been erased.

6.5 Applications

"Application" is a blanket term for all kinds of programs or software systems that do some task, using the operating system as a platform. An application might be tiny or large; it might focus on one specific task or handle a broad range; it might be sold or given away; its code could be highly proprietary, freely available open source, or unrestricted.

It might be useful to think in terms of a couple of kinds of applications: small, self-contained programs that do only one thing, and large programs that do complex sets of operations, like Word, iTunes or Photoshop.

As an example of a really simple application, consider the Unix program called `date`, which simply prints the current date and time:

```
$ date
Thu Sep 22 19:44:07 EDT 2011
```

The `date` program behaves the same way on all Unix-like systems; you can try it on a Mac by starting a Terminal window. The implementation of `date` is tiny, because it builds on a system call that provides the current date and time in an internal format, and on libraries of code for formatting and printing dates. Here's the code in C so you can see how short it is:

```
#include <stdio.h>
#include <time.h>
int main() {
  time_t t = time(0);
  printf("%s", ctime(&t));
  return 0;
}
```

Unix systems have a command that lists the files and folders in a directory, a bare-bones text-only version of programs like Windows Explorer and Mac OS X Finder. Other programs copy files, move them, etc., operations that have graphical equivalents in Finder and Explorer. Again, these programs use system calls to provide the basic information about what is in folders, and rely on libraries to read, write, format and display information.

An application like Word is much, much bigger than a program to explore the file system. It clearly has to include some of the same kind of file system code so that it can allow users to open files, read file contents and save documents in the file system. Word also includes sophisticated algorithms, for example to update the display continuously as the text changes. It also supports an elaborate user interface that displays information and provides ways to adjust sizes, fonts, colors, layout, and so on; this is likely to be a significant part of the program. Word and other large programs of substantial commercial value undergo continuous evolution as new features are added. I have no idea how big Word is, but would not be surprised if it were several million lines of C and C++.

A browser is an example of a large, free, and sometimes open source application of even greater complexity in some dimensions. You have surely used at least one of Firefox, Safari, Internet Explorer, Chrome or Opera, and if you're like me, you routinely use several. Chapter 10 will talk more about the web and how browsers fetch information; here I want to focus on the ideas in big, complicated programs.

Seen from the outside, a browser sends requests to web servers and retrieves information from them for display. Where's the complexity?

First, the browser has to deal with *asynchronous* events, that is, events that happen at unpredictable times and in no particular order. For instance, the browser has sent a request for a page because you clicked on a link, but it can't just wait for the reply; it has to remain responsive in case you scroll the current page, or abort the request if you hit the Back button or click on a different link, even while the requested page is coming in. It has to update the display if you reshape the window, perhaps

continuously as you reshape back and forth while data is arriving. If the page involves special functions like sound or movies, it has to manage those as well. Programming asynchronous systems is always hard, and browsers have lots of asynchrony.

The browser has to support many kinds of content, from static text to interactive programs that want to change what the page contains. Some of this can be delegated to helper programs—this is the norm for standard formats like PDF and movies—but the browser has to provide mechanisms for starting such helpers, sending and receiving data and requests for them, and controlling them.

The browser manages multiple tabs and/or multiple windows, each of which may be doing some of the foregoing operations. It maintains a history for each of these, along with other databases like bookmarks, favorites, and so on. It accesses the local file system for uploads and downloads.

It provides a platform for extensions at several levels: plug-ins like Flash and Silverlight, virtual machines for JavaScript and Java, and add-ons like those supported by Firefox, Safari and Chrome.

With all of this complicated code, a browser is vulnerable to attacks through bugs in its own implementation or in the programs it enables, and through the innocence, ignorance and stupidity of its users, most of whom (save the readers of this book) have almost no understanding of what's going on or what the risks might be. It's not easy.

If you look back over the description in this section, does it remind you of something? A current browser is very similar to an operating system. It manages resources, it controls simultaneous activities, it stores and retrieves information from multiple sources, and it provides a platform on which application programs can run.

For many years, it has seemed like it should be possible to use the browser as the operating system, and thus be independent of whatever operating system is controlling the underlying hardware. A decade or more ago, this was a fine idea but there were too many practical hurdles. Today it's a much more realistic possibility. Numerous services are already accessed exclusively through a browser interface—mail is the most obvious example—and this is likely to continue. Google distributes a browser-only operating system called Chrome OS that relies entirely on web-based services; a Chromebook is a computer that only runs Chrome OS. We'll come back to this topic when we talk about cloud computing in Chapter 11.

6.6 Layers of Software

Software, like many other things in computing, is organized into layers, analogous to geological strata, that separate one concern from another. Layering is one of the important ideas that help programmers to manage complexity.

At the bottom, at least for our purposes, is the hardware, which is more or less immutable except that buses make it possible to add and remove devices while the system is running.

The next level is the operating system proper, often called the *kernel* to suggest its central function. The operating system is a layer between the hardware and the applications. No matter what the hardware, the operating system can hide its specific properties and provide applications with an interface or facade that is independent of many of the details of the specific hardware. If the interface has been well designed, it's quite possible for the same operating system interface to be available on different kinds of CPUs and to be provided by different suppliers.

This is true of the Unix operating system interface—Unix runs on all kinds of processors, providing the same core services on each. In effect, the operating system is a commodity; the underlying hardware doesn't matter much except for price and performance, and the software on top doesn't depend on it. With care, all that's necessary to move a program to a new processor is to compile it with a suitable compiler. Of course the more tightly tied a program is to particular hardware, the harder this job will be, but it's eminently feasible for many programs. As a large-scale example, Apple converted its software from the IBM PowerPC processor to Intel processors in less than a year in 2005-2006.

This is less true of Windows, which has been fairly closely tied to the Intel architecture that began with the Intel 8086 CPU in 1978 and its many evolutionary steps since. (The family of processors is often called "x86" since for many years Intel processors had numbers ending in 86, including the 80286, 80386, and 80486.) The association is so tight that Windows running on Intel is sometimes called "Wintel."

The next layer above the operating system is a set of libraries that provide generally useful services so that individual programmers don't have to re-create them. Some libraries are at a low level, dealing with basic functionality (computing mathematical functions like square root or logarithm, for example, or date and time computations like those in the `date` command above); others are much more complicated (cryptography, graphics, compression). Components for graphical interfaces—menus, buttons, check boxes, scroll bars, tabbed panes, and the like—involve a lot of code, so once they are in a library, everyone can use them, which helps to ensure a uniform look and feel. That's why most Windows applications, or at least their basic graphical components, look so similar; the same is even more true on a Mac. It's too much work for most software vendors to reinvent and re-implement, and pointlessly different visual appearances are confusing to users.

As described above, a typical application program will use libraries as well as basic operating system services, gluing them all together into a set of operations that does the job. Indeed, the distinction between library function and system call is not well defined; a particular service could be implemented as a system call or as a library call that uses some system call.

Sometimes the distinction between kernel, library and application is not as clear as I have made it sound, since there are many ways to create software components and connect them. For instance, the kernel could provide fewer services and rely on libraries in a layer above to do most of the work. Or it could take on more of the task itself, relying less on libraries. The border between operating system and application

is not sharply defined.

What is the dividing line? A useful guideline, though not perfect, is that anything necessary to ensure that one application does not interfere with another is part of the operating system. Memory management—deciding where to put programs in memory as they run—is part of the operating system. Similarly, the file system—where to store information on the disk—is a central function. So is control of devices—two applications should not be able to run the printer at the same time, nor write to the display without coordination. At the center, control of the CPU is an operating system function, since that is necessary to ensure all the other properties.

A browser is not part of the operating system, since it's possible to run any browser, or multiple ones simultaneously, without interfering with shared resources or control.

This might sound like a technical fine point, but it has had major legal ramifications. The Department of Justice versus Microsoft antitrust lawsuit that began in 1994 (and ended in 2011) was in part about whether Microsoft's Internet Explorer browser was part of the operating system or merely an application. If it was part of the system (as Microsoft argued), then it could not reasonably be removed and Microsoft was within its rights to require the use of IE; if it was just an application, then Microsoft could be deemed to be illegally forcing others to use IE when they did not need to. The case was of course more complicated than this, but the dispute about where to draw this line was an important part. For the record, the court decided that a browser is an application, not part of the operating system; in the words of Judge Thomas Jackson, "Web browsers and operating systems are separate products."

7

Learning to Program

In my course, I teach a limited amount of programming, in a language called JavaScript. I think that it's important for a well-informed person to know something about programming, perhaps only that it can be surprisingly difficult to get very simple programs working properly. There is nothing like doing battle with a computer to teach this lesson, but also to give people a taste of the wonderful feeling of accomplishment when a program does work for the first time. It may also be valuable to have enough programming experience that you are cautious when someone says that programming is easy, or that there are no errors in a program. If you had trouble making 10 lines of code work after a day of struggle, you might be legitimately skeptical of someone who claims that a million-line program will be delivered on time and bug-free. On the other hand, there are times when it's helpful to know that not all programming tasks are difficult, for instance when hiring a consultant.

What programming language? There are zillions of languages, though none is ideal as the first language for you to learn or for me to describe in a book like this. Long ago I used Visual Basic, a language and development environment from Microsoft that made it easy to produce professional-looking Windows applications. A dialect of VB is included in Microsoft Office programs and a number of others, so one can write VB programs that augment or control Word and Excel (and incidentally provide a fine path for virus infections).

Sadly, Visual Basic is no longer a good option. Although Microsoft supplies an almost full-featured version for free, the language and system have become significantly more complicated over the past decade. Most important, VB only runs on Windows. I needed a universally available language.

That reasoning led me to JavaScript, which has a number of advantages. It is ubiquitous, available in every browser. Almost every web page includes some

JavaScript, and that code can be easily examined. If you do write a program, you can use it on your own web pages to show your friends and family how clever you are. The language itself is simple and one can do neat things with comparatively little experience; at the same time, it is powerful enough to do quite remarkable computations. Many web page effects are enabled by JavaScript, including Google Docs and equivalent programs from other sources. JavaScript is also the language for APIs provided by web services like Twitter, Facebook, Amazon, and so on.

JavaScript certainly has disadvantages too. The browser interface is not as standardized as one would like, so programs don't always behave the same way on different browsers. At the level we're talking about, this is not an issue, and even for professional programmers it's getting better all the time. Some constructs in the language are awkward and there are definitely some surprising behaviors. JavaScript programs generally only run as part of a web page; they have little independent existence, though some other systems, notably Adobe's PDF reader, do support JavaScript. Because of that dependency on a browser as host, one typically has to learn a small amount of HTML, the language for describing the layout of web pages. In spite of these drawbacks, however, it's well worth the effort of learning a little JavaScript.

If you follow the material here, you will learn how to program, at least at a basic level, and that's a skill worth having. Whatever you learn will carry over into other languages, making them easier to learn. If you want to dig deeper or get a different take on it, search for "javascript tutorial" on the web and you'll get a long list of helpful sites.

All that said, however, it's fine to skim this chapter and ignore syntactic details; not much else depends on it.

7.1 Programming Language Concepts

Programming languages share certain basic ideas, since they are all notations for spelling out a computation as a sequence of steps. Every programming language thus will provide ways to get input data upon which to compute; do arithmetic; store and retrieve intermediate values as computation proceeds; display results along the way; decide how to proceed on the basis of previous computations; and save results when the computation is finished.

Languages have *syntax*, that is, rules that define what is grammatically legal and what is not. Programming languages are picky on the grammatical side: you have to say it right or there will be a complaint. Languages also have *semantics*, that is, a defined meaning for every construction in the language.

In theory there should be no ambiguity about whether a particular program is syntactically correct and if so what its meaning is. This ideal is not always achieved, unfortunately. Languages are usually defined in words and, like any other document in a natural language, ambiguities and different interpretations can occur, implementers can make mistakes, and languages evolve over time. As a result, JavaScript

implementations differ somewhat from browser to browser and even from version to version of the same browser.

JavaScript really has three aspects. First is the language itself: statements that tell the computer to do arithmetic, test conditions, and repeat computations. Second, JavaScript includes libraries of code that others have written that you can use in your own programs; these are pre-fabricated pieces that you don't have to write for yourself. Typical examples include math functions, calendar computations, and functions for searching and manipulating text. Third, there is access to the browser and the web page that the JavaScript program is part of; the program can get input from a user, react to events like a user pushing a button or typing into a form, and cause the browser to display different content or switch to a different page.

7.2 A First JavaScript Example

The first JavaScript example is as small as they get: it just pops up a dialog box that says "Hello, world" when the web page is loaded. Here's the complete page in HTML (which we will meet when we talk about the World Wide Web in Chapter 10); for now focus on the single line of highlighted JavaScript code, which appears between `<script>` and `</script>`.

```
<html>
  <body>
    <script>
      alert("Hello, world");
    </script>
  </body>
</html>
```

If you put these seven lines into a file called `hello.html` and tell your browser to load that file, you'll see a result like one of these:

These images are from Firefox, Chrome, Safari and Opera, all on a Mac. They're clearly different, not in any way that really matters, but they do make the point about how different browsers behave differently. For comparison, here are the dialog boxes from Firefox and Internet Explorer on a Windows PC running XP:

The `alert` function is part of the JavaScript library for interacting with the browser; it shows a dialog box that displays whatever text appears between the quotes. By the way, when you create your own JavaScript programs, you must use the standard double quote character that looks like this `"`, not the so-called "smart quotes" that you see in the rest of the book—that's an example of a syntax rule. Don't use a word processor like Word to create HTML files; use a text editor like

Notepad or TextEdit and make sure that it saves files as plain text (that is, without formatting information) even when the filename extension is `.html`.

Once you have this example working, you can extend it to do more interesting computations. I won't show the HTML parts from now on, just the JavaScript, which goes between `<script>` tags.

7.3 A Second JavaScript Example

The second little program asks the user for a name, then displays a personalized greeting:

```
var username;
username = prompt("What's your name?");
alert("Hello, " + username);
```

This program has several new constructions and corresponding ideas. First, the word `var` introduces or *declares* a *variable*, which is a place in RAM where the program can store a value as the program runs. It's called a variable because its value can be changed as a result of what the program is doing. Declaring a variable is the high-level language analog of giving a name to a memory location as we did in the Toy assembly language. Metaphorically, declarations are like the list of characters in a play. I named the variable `username`. I could have called it anything, but `username` describes its role in this program.

Second, the program uses a JavaScript library function called `prompt`, which is similar to `alert` but asks the user for input. Whatever the user types is made available to the program as the value computed by the `prompt` function. That value is assigned to the variable `username` by the line

```
username = prompt("What's your name?");
```

The equals sign = means "do the operation on the right side and store the result in the variable named on the left side," and is an example of semantics. This operation is called *assignment*. Most programming languages use the equals sign for assignment, in spite of potential confusion with mathematical equality. This = does not mean equality; it means copying a value.

Finally, the plus sign + is used in the `alert` statement

```
alert("Hello, " + username);
```

to join the word `Hello` (with its comma and space) and the name that was provided by the user. This is potentially confusing too because + in this context does not mean addition of numbers but concatenation of two sequences of text characters.

When you run this program, `prompt` displays a dialog box where you can type something:

If you type "Joe" and push OK, the result is this:

It would be an easy extension to allow for a first name and a last name as separate inputs, and there are plenty of variants that you might try for practice. Notice that if you respond with "My name is Joe", the result will be "Hello, My name is Joe". If you want smarter behavior from a computer, you have to program it yourself.

7.4 Loops

Chapter 5 showed a JavaScript version of the program that adds up a sequence of numbers. Here it is again so you don't have to look back:

```
var num, sum;
sum = 0;
num = prompt("Enter new value, or 0 to end");
while (num != 0) {
   sum = sum + parseInt(num);
   num = prompt("Enter new value, or 0 to end");
}
alert("Sum = " + sum);
```

As a reminder, the program reads numbers until a zero is entered, then prints the sum. We've already seen several of the language features in this program, like declarations, assignments, and the `prompt` function. The first line is a variable declaration that names the two variables `num` and `sum` that the program will use. The second line is an assignment statement that sets `sum` to zero, and the third line sets `num` to the value that the user types into the dialog box.

The really important one is the `while` loop, which includes lines four through seven. Computers are wonderful devices for repeating sequences of instructions over and over again; the question is how to express that repetition in a programming language. The Toy language introduced the `GOTO` instruction, which causes transfer to another place in a program rather than the next instruction in sequence, and the `IFZERO` instruction, which tests a condition and either transfers or not depending on the accumulator value.

These ideas appear in most high-level languages in a statement called a *while loop*, which provides a more orderly and disciplined way to repeat a sequence of operations. The `while` tests a condition (written between parentheses) and if the condition is true, executes all the statements between the `{ ... }` braces in order. It then goes back and tests the condition again. This cycle continues until the condition is false, at which time execution resumes with the statement that follows the closing brace.

This is almost an exact match for what we wrote with `IFZERO` and `GOTO` in Toy programs in Chapter 3, except that we don't need to invent labels and the test condition can be anything. Here the test is whether the variable `num` has a non-zero value. The operator `!=` means "not equal to"; it's inherited from C, as is the `while` statement itself.

I've been casual about the type of data that these sample programs process, but internally computers make a strong distinction between numbers like `123` and arbitrary text like `Hello`. Some languages require programmers to express that distinction carefully; other languages try to guess what the programmer might have meant. JavaScript is closer to the latter position, so sometimes it's necessary to be explicit about the type of data you're dealing with and how values are to be interpreted. The function `parseInt` is an example; it converts text into an internal form that can be used for arithmetic, that is, its input data is to be treated as an integer (like `123`) instead of as three characters that happen to be decimal digits. Without the `parseInt`, the data returned by `prompt` would be interpreted as text and the `+` operator would append it to the end of the previous text. The result would be the concatenation of all the digits the user entered—interesting, perhaps, but definitely not what was intended.

7.5 Conditionals

This next example does a slightly different job, finding the largest of all the numbers that were provided. It's an excuse to introduce another control-flow statement, `if-else`, which appears in some form in all high-level languages as a way to make decisions; it's in effect a general-purpose version of `IFZERO`. JavaScript's version of `if-else` is the same as in C.

```
var max, num;
max = 0;
num = prompt("Enter new value, or 0 to end");
while (num != 0) {
   if (parseInt(num) > max) {
      max = num;
   }
   num = prompt("Enter new value, or 0 to end");
}
alert("Maximum is " + max);
```

The `if-else` comes in two forms. The one shown here has no `else` part: if the parenthesized condition is true, the statements in `{ ... }` braces that follow are executed. Regardless, execution resumes with the statement that follows the closing brace. The more general form has an `else` part for a set of statements to be executed if the condition is false. Either way, execution resumes with the next statement after the whole `if-else`.

You might have noticed that the example programs use indentation to indicate structure: the statements controlled by `while` and `if` are indented. This is standard (good) practice; it makes it possible to see at a glance what the scope is for statements like `while` and `if` that control other statements. Indeed, there are some programming languages that require consistent indentation.

It's easy to test this program if you put it on a web page, but professional programmers would test it even before that, simulating its behavior by carefully stepping through the statements of the program one at a time, doing what a real computer would do. If you do that (it's good practice for being sure you understand how it works), you will conclude that the program works for any sequence of input values.

Or does it? It works fine if the inputs include positive numbers, but what happens if all of them are negative? If you try that, you'll discover that the program always says that the maximum is zero.

Think about why for a moment. The program keeps track of the largest value seen so far in a variable called `max` (just like finding the tallest person in the room). The variable has to have some initial value before we compare subsequent numbers to it, so the program sets it to zero at the beginning, before any numbers have been provided by the user. That's fine if at least one input value is greater than zero, as would be the case for heights. But if all inputs are negative, which could happen for checkbook entries, the program doesn't print the largest of the negative values; instead it prints the end-of-input value.

This bug is easy to eliminate. I'll show one solution at the end of the chapter but it's a good exercise to discover a fix by yourself.

The other thing that this example shows is the importance of testing. Testing requires more than throwing random inputs at a program. Good testers think hard about what could go wrong, including weird or invalid input, and "edge" or "boundary" cases like no data at all or division by zero. A good tester would think of the possibility of all negative inputs. The problem is that as programs become larger, it's

harder and harder to think of all the test cases, especially when they involve humans who are likely to input random values at random times in random order. There's no perfect solution, but careful program design and implementation helps, as does including checks and balances in the program from the beginning, so that if something does go wrong, it's likely to be caught early by the program itself.

7.6 Libraries and Interfaces

JavaScript has an important role as the extension mechanism for sophisticated web applications. Google Maps is a nice example. It provides a library and an API so that map operations can be controlled by JavaScript programs, not just by mouse clicks. Thus anyone can write JavaScript programs that display information on a map provided by Google. The API is easy to use; for example this bit of code (with a few extra lines of HTML):

```
function initialize() {
  var latlong = new google.maps.LatLng(
                 38.89767967065576, -77.03656196594238);
  var opts = {
    zoom: 18,
    center: latlong,
    mapTypeId: google.maps.MapTypeId.HYBRID
  };
  var map = new google.maps.Map(document.getElementById("map"),opts);
  var marker = new google.maps.Marker({
    position: latlong,
    map: map,
    title: "You are here, more or less"
  });
}
```

displays the image at the top of the next page, where perhaps some reader of this book will live.

As we'll see in Chapter 11, the trend on the web is towards more and more use of JavaScript, including programmable interfaces like Maps. It's difficult to protect intellectual property when you are forced to reveal source code, which you necessarily must do if you're using JavaScript. Anyone can look at the source of a page with the View Source menu item in a browser. Some JavaScript is obfuscated, either intentionally or as a by-product of trying to make it compact so that it can be downloaded more quickly, and the result can be totally impenetrable unless one is really determined.

7.7 How JavaScript Works

Recall the discussion of compilers, assemblers and machine instructions in Chapter 3. A JavaScript program is converted into executable form in an analogous way, but there are significantly different details. When the browser encounters JavaScript in a web page (when it sees a `<script>` tag, for example) it passes the text of the program to a JavaScript compiler that is normally a separate program or a library included with the browser. The compiler examines the program, checks for errors, and compiles it into assembly language instructions for a made-up machine analogous to the Toy, though with a richer instruction repertoire: a virtual machine as described in the previous chapter. It then runs a simulator like the Toy simulator to perform whatever actions the JavaScript program is supposed to do. The simulator and the browser interact closely: when a user pushes a button, for example, the browser notifies the simulator that the button has been pushed. When the simulator wants to do something like pop up a dialog box, it asks the browser to do the work by calling `alert` or `prompt`.

This is all we'll say about JavaScript here, but if you're interested in more, there are good books and online tutorials, some of which let you edit JavaScript code in place and show you the results immediately. Programming can be frustrating, but it can also be great fun, and you can even make quite a decent living with it. Anyone can be a programmer, but it helps if you have an eye for detail and are able to zoom in to look at fine points and zoom out again to the big picture. It also helps to be a bit compulsive about getting details right, since programs won't work well or perhaps

not at all if you're not careful.

Here's one possible answer to the programming question a few pages back:

```
num = prompt("Enter new value, or 0 to end");
max = num;
while (num != 0) ...
```

Set `max` to the first number that the user supplies; that is the largest value so far, no matter whether it's positive or negative. Nothing else needs to change, and the program now handles all inputs (though it will quit early if one of those values is zero). It even produces something sensible if the user provides no values at all, though to handle that well in general requires learning more about the `prompt` function.

Wrapup on Software

We've covered a lot of material in the software chapters, so here's a brief summary of the most important points.

Algorithms. An algorithm is a sequence of precise, unambiguous steps that performs some task and then stops; it describes a computation independent of implementation details. The steps are based on well-defined basic or primitive operations. There are many algorithms; we concentrated on basic ones like searching and sorting.

Complexity. The complexity of an algorithm is an abstract measure of how much work it performs, measured in terms of basic operations like examining a data item or comparing one data item with another, and expressed in terms of how that number of operations depends on the number of items. This leads to a hierarchy of complexities, ranging in our case from logarithmic at one end (doubling the number of items adds only one operation) through linear (directly proportional to the number of items, the most common case and the easiest to relate to) to exponential (where adding one more item doubles the number of operations). Complexity is a worst-case measure—real problem instances might mostly be easy—and it's asymptotic—only relevant when the number of items is large.

Programming. Algorithms are abstract. A program is the concrete expression of all the steps necessary to make a real computer do a complete real task. A program has to cope with limited memory and time, finite size and precision of numbers, and perverse or malicious users.

Programming languages. A programming language is a notation for expressing all those steps, in a form that people can write comfortably but that can be translated into the binary representation that computers ultimately use. The translation can be accomplished in several ways, but in the most common case, a compiler, perhaps with an assembler, translates a program written in a language like C into binary to

run on a real computer. Each different kind of processor has a different repertoire and representation of instructions and thus needs at least partly a different compiler. An interpreter or virtual machine is a program that simulates a real or made-up computer for which code can be compiled and run; this is commonly how JavaScript programs operate.

Libraries. Writing a program to run on a real computer involves a lot of intricate detail for common operations. Libraries and related mechanisms provide prefabricated components that programmers can use as they create their own programs, so that new work can build on what has already been done. Programming today is often as much gluing existing components together as writing original code. Components may be library functions like those we've seen in JavaScript, or big systems like Google Maps and other web services. Underneath, however, they are all created by programmers writing detailed instructions in some of the languages we've talked about or others like them.

Interfaces. An interface or API is a contract between software that provides some service and software that uses the service. Libraries and components provide their services through application programming interfaces. Operating systems make hardware look more regular and programmable through their system call interfaces.

Abstraction and virtualization. Software can be used to hide the details of implementation or to pretend to be something else; examples include virtual memory, virtual machines, and interpreters.

Bugs. Computers are unforgiving, and programming requires a sustained level of error-free performance from all-too-fallible programmers. Thus all big programs have bugs, and simply don't do what they were meant to. Some bugs are mere nuisances, more like bad design than actual error. ("That's not a bug, it's a feature" is a common saying among programmers.) Some are triggered only in such rare or unusual circumstances that they aren't even reproducible, let alone fixable. But some bugs are truly serious, with potentially grave consequences that put safety and even lives at stake. Liability is likely to become more of an issue in computing devices than it has been, especially as more and more critical systems are based on software. The model of "take it or leave it, without warranty" that operates in personal computing will probably be replaced by more reasonable product warranties and consumer protections as in the hardware world.

As we learn from experience, as programs are created more from proven components, and as existing bugs are squashed, in principle programs ought to become more and more free of errors. Against these advances, however, are the inevitable failures arising from continual changes as computers and languages evolve, as systems undertake new requirements, and as marketing and consumer desires create relentless pressure for new features. Unfortunately, bugs are likely to always be with us.

Part III

Communications

Communications is the third major part of our tripartite organization, after hardware and software. In many ways, this is where things start to get really interesting (occasionally in the sense of "May you live in interesting times"), because communications involves computers talking to each other, usually on our behalf, but sometimes up to no good at all. Most systems now combine hardware, software, and communications, so the pieces that we have been talking about will all come together. Communications is also the place where most societal issues arise, presenting difficult problems of privacy, security, and the competing rights of individuals, businesses, and governments.

I'm going to give a bit of historical background, talk about network technology, and then get to the Internet, which is the collection of networks that carries much of the world's computer-to-computer traffic. After that comes the (World Wide) Web, which in the mid 1990s took the Internet from a small and mostly technical user population to a ubiquitous service for everyone. We'll then turn to some of the applications that use the Internet, like mail, online commerce, and social networks, along with threats and countermeasures.

People have communicated over long distances since as far back as there are records, using much ingenuity and a remarkable variety of physical mechanisms. Every example comes with a fascinating story that would be worth a book of its own.

Long-distance runners have carried messages for millennia. In 490 BCE, Pheidippides ran 26 miles (42 kilometers) from the battlefield at Marathon to Athens to bring news of the Athenians' great victory over the Persians. Unfortunately, at least in legend, he gasped "Rejoice, we conquer" and died.

Herodotus described the system of riders who carried messages throughout the Persian empire at about the same time; his description lives on in the 1914 inscription

on the main post office building on Eighth Avenue in New York: "Neither snow nor rain nor heat nor gloom of night stays these couriers from the swift completion of their appointed rounds." The Pony Express, whose horseback riders carried mail 1,900 miles (3,000 km) between St. Joseph, Missouri, and Sacramento, California, is an icon of the American West, though it lasted less than two years, from April 1860 to October 1861.

Signal lights and fires, mirrors, flags, drums, carrier pigeons, even human voices ("stentorian" comes from the Greek word "stentor," a person whose loud voice carried messages across narrow valleys) have all communicated over long distances.

One early mechanical system is not as well known as it deserves to be: the *optical telegraph* invented in France by Claude Chappe around 1792 and independently in Sweden by Abraham Edelcrantz. The optical telegraph used a signaling system based on mechanical shutters or arms mounted on towers. The telegraph operator read signals coming from the adjacent tower in one direction and passed them on to the next tower in the other direction. By the 1830s there was an extensive network of these towers over large parts of Europe and some parts of the United States. Towers were about 10 kilometers (6 miles) apart; transmission speeds were a few characters per minute, and according to one account, a single character could be sent from Lille to Paris (230 km or 140 miles) in about 10 minutes.

The issues that arise in modern communications systems appeared even in the 1790s. Protocols were needed to agree on how information was represented, how messages were exchanged, and how errors were detected and recovered from. Sending information quickly was always a problem, though it would only take a few hours to send a short message from one end of France to the other. Security and privacy problems cropped up too: Chapter 61 of *The Count of Monte Cristo* by Alexandre

Dumas, published in 1844, relates how the count bribed a telegraph operator to send a false message to Paris, causing the financial ruin of the evil Danglars.

The optical telegraph had one major operational problem: it could only be used when visibility was good, not at night or in bad weather. The electrical telegraph, invented by Samuel F. B. Morse in the 1830s, came to fruition in the 1840s, and killed the optical telegraph within a decade. Commercial telegraph service soon linked major US cities (the first was between Baltimore and Washington in 1844), and the first trans-Atlantic telegraph cable was laid in 1858. The electrical telegraph caused many of the same hopes, aspirations and disappointments that people experienced during the early days of the Internet boom and dot-com bust of the late 1990s. Fortunes were made and lost, frauds were perpetrated, optimists predicted the coming of world peace and understanding, and realists correctly perceived that although the details were different, most of it had been seen before. "This time it's different" is rarely if ever true.

The story is that in 1876 Alexander Graham Bell beat Elisha Gray to the US Patent Office by a few hours with his invention, the telephone, though there is still uncertainty about the exact sequence of events. As the telephone evolved over the next hundred years, it did revolutionize communications (though it did not lead to world peace and understanding either). It let people talk to each other directly, it required no expertise to use, and standards and agreements among telephone companies made it possible to connect almost any pair of telephones in the world.

The telephone system profited from a long period of comparative stability. It only carried the human voice. A typical conversation lasted three minutes, so it didn't matter if it took a few seconds to set up the connection. A telephone number was a unique identifier with a fairly clear geographical location. The user interface, a plain black telephone with a rotary dial (mostly now vanished, leaving only the linguistic echo of "dialing the phone"), was spartan; all the intelligence was in the network and a user could do nothing but answer the phone when it rang, dial numbers to place a call, or ask a human operator for complicated services.

In fact, though the rotary dial phone is now rarely seen, a friend sent me a picture of one that his mother still uses:

All of this meant that the telephone system could concentrate on core values: high reliability and guaranteed quality of service. For 50 years, if one picked up the phone, there would be a dial tone (another linguistic echo), a call would go through, the person at the other end could be heard clearly, and it would stay that way until both parties hung up. (I probably have an excessively rosy view of the phone system, since I worked for over thirty years at Bell Labs, part of AT&T, and saw many of the changes from the inside, though far from the center of the action. On the other hand, I do miss the almost perfect reliability and clarity of pre-cell phone days.)

For the phone system, the last 25 years of the 20th century were a period of rapid change in technology, society and politics. The traffic model changed as fax machines became common in the 1980s. Computer to computer communications became more common as well, using modems that convert bits into sound and vice versa; like fax machines, they used audio to pass digital data through the analog phone system. Technology made it possible to set up calls faster, to send more information (with fiber optic cables in particular, both domestically and across oceans), and to encode everything digitally. Cell phones changed usage patterns even more dramatically; today they dominate telephony. The results have included a downward spiral in revenue for the old-line telephone companies and the rise and often fall of a lot of new players.

Politically there was a worldwide revolution as telephone companies that were either highly regulated private companies like AT&T, which dominated the USA, or government agencies in much of the rest of the world became deregulated private operations. This unleashed wide-open competition. In the USA, AT&T was split, first into long distance plus manufacturing (AT&T) and local service (the Bell operating companies) in 1984, and then the residual AT&T split voluntarily into long distance and services (AT&T) and manufacturing (Lucent) in 1995. AT&T fell upon hard times, and was ultimately acquired by Southwest Bell, one of the operating companies it had spun off in 1984. Lucent merged with the French telephone equipment company Alcatel in 2006.

Today, telephone companies continue to wrestle with the threats raised by new communications systems, primarily those based on the Internet, often in the face of declining revenue and market share. The main threat comes from Internet telephony. It's not hard to send digital voice over the Internet. Services like Skype push this further, offering free computer-to-computer calling and a way to call from the Internet to conventional phones for a nominal price, usually much less than the incumbent phone companies charge, especially for international calling. Google's services are if anything even cheaper (free in the US and Canada right now), so the handwriting is on the wall. I recall a friend telling AT&T management sometime in the early 1990s that prices for domestic long distance phone calls would drop to a cent a minute—at a time when AT&T was charging more like 10 cents a minute—and he was laughed at.

Naturally, incumbent companies are not happy with this state of affairs, and they are fighting to retain their revenues and effective monopolies by technical, legal and

political means. One approach is to charge competitors for access to the wired phones in your house. This seems fair, since the wiring to your home was installed by the telephone company, which thus has a legitimate right to recover costs. Another approach is to put bandwidth limitations and other slowdowns in the path of competitors that provide telephone service using the Internet ("voice over IP" or "VoIP") or other competition. This seems less legitimate.

It's part of a general issue called *net neutrality*. Should service providers be allowed to interfere with, degrade, or block traffic for any reason other than purely technical ones related to efficient network management? Should telephone and cable companies be required to provide the same level of Internet service to all users, or should they be able to treat services and users differently? If so, on what grounds? For instance, should a telephone company be allowed to slow down or block traffic for a competitor, say a VoIP company like Vonage? Should a cable company be allowed to slow down traffic for an Internet movie service like Netflix? Should service providers be allowed to slow down or block traffic for sites that espouse social or political views that the owners disagree with? As usual, there are arguments on both sides.

The resolution of the net neutrality issue will have a significant effect on the future of the Internet, which so far has profited greatly (and benefited us) by providing a neutral platform that carries all traffic without interference or restrictions.

8

Networking

In this chapter, I'm going to talk about several networking technologies that one encounters in home, office or dorm. These local networks are the first step in our connection to the Internet.

All communications systems share some properties. At the source, they convert information into a physical representation that can be transmitted over some medium. At the destination, they convert that representation back into a usable form.

Bandwidth is the most fundamental property—how fast the system can transmit data. This ranges from a few bits per second for systems that operate under severe power or environmental constraints to terabits per second for fiber optic networks.

Latency or *delay* measures how long it takes for a particular chunk of information to go through the system; driving a truck full of disk drives across the country has enormous bandwidth but high delay.

Jitter—the variability of delay—also matters in some communications systems, especially those dealing with speech.

Range defines how geographically big a network can be with a given technology. Some networks are very local, a few meters at most; others literally span the world.

Other properties include whether the network broadcasts so that multiple receivers can hear one sender (as with radio), or is point to point, pairing a specific sender and receiver. Broadcast networks are intrinsically more vulnerable to eavesdropping, which may have security implications. One has to worry about what kinds of errors can occur and how they are handled. Naturally cost is always a factor.

8.1 Telephones and Modems

The telephone network is a large and successful worldwide network that began by carrying voice traffic and eventually evolved to carrying considerable data traffic as well; it served for nearly twenty years as the primary way that people connected home computers to the Internet.

At the residential level, the telephone system carries analog voice signals, not data. So it's necessary to have a device that converts digital data—bits—into analog sound and back again before one can use the telephone system as a data network. The process of imposing an information-carrying pattern on a signal is called *modulation*. At the other end, it's necessary to convert the pattern back into bits, which is called *demodulation*. The device that does modulation and demodulation is called a *modem*. A telephone modem used to be a large and expensive separate box of electronics, but it's gradually been reduced to a single chip, and it's practically free. In spite of that, the use of wired telephones to connect to the Internet is less common now, and fewer computers have modems.

Using the telephone for data connections has drawbacks. It requires a phone line, so if you only have one phone line in your home, you have to choose between connecting or leaving the phone available for voice calls. More important for most people, however, is that there is a limit on how quickly information can be sent by telephone. The maximum bandwidth is about 56 Kbps (56,000 *bits* per second—a lower case "b" conventionally stands for bits, in contrast to the upper case "B" that stands for bytes), or 6 or 7 KB per second. A 20 KB web page thus takes 3 seconds to download, a 400 KB image takes nearly 60 seconds, and a software update can easily take hours. It's just too slow. Finally, for much of the time during which modems were widely used, "long distance," that is, calling outside some narrow local calling area, was expensive. Companies like AOL that offered dial-up access had to provide banks of local telephone numbers so that their users did not have to pay long distance charges. Even then, phone companies were unhappy because data calls typically lasted much longer than voice calls but generated no extra revenue. The telephone network was designed for 3-minute voice calls, not multi-hour data calls, and local calls were billed at a flat rate.

8.2 Cable and DSL

The limitations on how fast a telephone line can carry signals are inherent in its design. Telephones were meant to carry human speech, which only requires about 3 KHz of bandwidth, so clever encoding, compression, and other tricks can't raise the speed much beyond 56 Kbps. Two other technologies provide an alternative for many people, with at least 10 times the bandwidth.

The first is to use the cable that carries cable television into many homes. That cable can carry hundreds of channels simultaneously. It has enough excess capacity that it can be used to carry data to and from homes as well; on a typical cable system a few Mbps is standard. The device that converts signals from the cable into bits for

a computer and back again is called a *cable modem*, since its task is directly analogous to a telephone modem, though it runs quite a bit faster.

The high speed is illusory in a way. The same TV signal goes to every house, regardless of whether it's being watched or not. On the other hand, although the cable is a shared medium, data going to my house is meant for me and won't be the same data at the same time as is going to your house, so there's no way to share the content. Data bandwidth has to be shared among data users of the cable, and if I'm using a lot of it, you won't get as much. More likely, we'll both get less. Fortunately, we're not likely to interfere with each other too much. It's rather like a communications version of deliberate over-booking by airlines and hotels. They know that not everyone will show up, so they can safely over-commit resources. It works in communications systems too.

Now you can see another problem. We all watch potentially the same TV signals, but I don't want my data going to your house any more than you want your data going to my house. After all, it's personal—it includes my email, my online shopping and banking information, and perhaps even personal entertainment tastes that I would rather not have others know about. This can be solved by encryption, which prevents anyone else from reading data to or from me; we'll talk more about this in Chapter 10.

There is yet another complication. The first cable networks were one-way: signals were broadcast to all homes, which was easy to build but provided no way to send information back from the customer to the cable company. Cable companies had to find a way to deal with this anyway, to enable pay-per-view and other services that require communication from the customer, so cable systems have become two-way, which makes them usable as communications systems for computer data. Sometimes the "two-way" aspect uses two different paths; for instance, some satellite TV systems use a telephone line for the upstream link. This can be painfully slow for accessing the Internet, though it's fine for ordering a movie. It is common for the upstream speed to be significantly lower than downstream, which can make it slow to upload images and video.

The other reasonably fast network technology for home use is based on the other system that's already in the home, the good old telephone. It's called Digital Subscriber Loop or *DSL* (sometimes *ADSL*, for "asymmetric," because the bandwidth to the home is higher than the bandwidth from the home), and it provides much the same service as cable, but with several significant differences underneath.

DSL sends data on the telephone wire with a technique that doesn't interfere with the voice signal, so you can talk on the phone while surfing and neither affects the other. This works well but only up to a certain distance. If you live within about 3 miles (5 km) of a local telephone company switching office, as many urban and suburban users do, you can have DSL. If you're farther away, you're out of luck.

The other nice thing about DSL is that it's not a shared medium. It uses your phone wires, but no one else does, so you don't share the capacity with your neighbors, nor do your bits go to their homes. A special box at your house—another

modem, with a matching one at the telephone company's building—converts signals into the right form to be sent along the wires. Otherwise, cable and DSL look and feel much the same. The prices tend to be about the same too, at least when there is competition.

The capacity of these systems is determined by the hardware, and up to some limit it doesn't take more equipment or anything else to provide more of that capacity to consumers. Thus suppliers can play a cute game where they can give you more capacity for more money without changing anything except an entry in a database.

All such systems have the advantage that they can be left on all the time; there's no notion of dialing a call or tying up a phone line.

Technology continues to improve, and in some places it is now possible to obtain home fiber optic service rather than the older coaxial cable or copper wires. Fiber optic systems are much faster than the alternatives. Signals are sent as pulses of light along a very thin and extremely pure glass fiber with low loss; signals can propagate for kilometers before needing regeneration. Starting in the early 1990s, I was part of a "fiber to the home" research experiment and for a decade had a 160 Mbps [sic] connection to my house. That gave me serious bragging rights but not much else since there wasn't any service that could take advantage of that much bandwidth.

8.3 Local Area Networks and Ethernet

Telephone, cable and DSL are all networking technologies that connect a computer to a larger system and usually at a considerable distance. Historically, there was another thread of network development that led to one of today's most common systems, Ethernet.

In the late 1960s and early 1970s, Xerox's Palo Alto Research Center (Xerox PARC) developed a personal computer called the Alto that was very innovative itself and was a vehicle for experiments that in turn led to other innovations. It had the first window system, the first bitmap display (that is, not restricted to displaying characters), and the first laser printers. Although Altos were too expensive to be personal computers in today's sense, every researcher at PARC had one.

One problem was how to connect Altos to each other or to a shared resource like a printer. The solution, invented by Bob Metcalfe and David Boggs in the early 1970s, was a networking technology that they called *Ethernet*. An Ethernet carried signals between computers that were all connected to a coaxial cable, physically similar to the one that carries cable TV into your house today. The signals were pulses of voltage whose strength or polarity encoded bit values; the simplest form might use a positive voltage for a 1-bit and a negative voltage for a 0-bit. Each computer was connected to the Ethernet by a device with a unique identification number. When one computer wanted to send a message to another, it listened to make sure no one else was already sending, then broadcast its message onto the cable along with the identification number of the intended recipient. Every computer on the cable could hear the message, but only the computer to which it was directed would read and process

it.

Every Ethernet device has a 48-bit identification number, different from all other devices, called its (Ethernet) address. This allows for 2^{48} (about 2.8×10^{14}) devices in all. You can find the Ethernet addresses for your computer, since they are sometimes printed on the bottom, and they can also be displayed by programs like `ipconfig` on Windows or `ifconfig` on Macs. There will often be one for a wired interface and one for wireless. Ethernet addresses are always given in hexadecimal, two digits per byte, so there are 12 hex digits in all. Look for something like `00096BD0E705`, though since that's from one of my laptops, you won't likely find exactly that one on your computer.

From the discussion of cable systems above, you can imagine that Ethernets will have a couple of the same problems: privacy and contention for a limited resource.

Contention is handled by a neat trick: if a network interface starts to send but detects that someone else is sending too, it stops, waits a brief period, and tries again. If the waiting time is random and gradually increases over a series of failures, then eventually everything goes through.

Privacy wasn't originally an issue since everyone was an employee of the same company and all were working in the same small building. Privacy is a major problem today, however. It's possible for software to set an Ethernet interface into "promiscuous mode," in which it reads the contents of all messages on the network, not just those intended for it specifically. That means it can easily look for interesting content like unencrypted passwords. Such "sniffing" used to be a common security problem on Ethernet networks in college dorms. Fortunately, encryption of everything on the cable is a solution, though sometimes it's hard to convince people to use encryption software.

You can experiment with sniffing with a free program called Wireshark, which displays information about Ethernet traffic, including wireless. I occasionally demonstrate Wireshark in class when it seems that students are paying more attention to their laptops than to me; the demo does catch their attention, albeit briefly. The Firefox add-on called Firesheep is even more compelling, since it not only reveals who is accessing what sites, but makes it trivial to masquerade as one of those users. Firesheep watches traffic on open (unencrypted) networks and collects authentication information from sites like Facebook, Twitter, and many others, which can be used to pretend to be the real user. If one sits in a popular location, it's quite possible to pick up a dozen or two connections within a few minutes, and undetectably become any of those people, just by pushing a button. (Warning: doing so is likely to get you into trouble in most settings.)

Information on an Ethernet is transmitted in packets. A *packet* is a sequence of bits or bytes that contains information in a precisely defined format so that it can be packed up for sending and cracked open when received. If you think of a packet as an envelope (or perhaps a postcard), with the sender's address, the recipient's address, the real content, and miscellaneous other information, in a standard format, that's a reasonably good metaphor, as are the standardized packages used by shipping

companies like FedEx.

The details of packet format and content vary greatly among networks, as we will see. An Ethernet packet is a minimum of 64 bytes long and a maximum of 1518 bytes. It includes six-byte source and destination addresses, some miscellaneous information, and up to about 1500 bytes of data.

source address	destination address	data length	data (48-1500 bytes)	error check

Ethernet has been an exceptionally successful technology. It was first made into a commercial product (not by Xerox, but by 3COM, a company founded by Metcalfe), and over the years billions of Ethernet devices have been sold by a large number of vendors. The first version ran at 3 Mbps, but today's versions run anywhere from 100 Mbps up to 10 Gbps. As with modems, the first devices were bulky and expensive, but today an Ethernet interface is a single inexpensive chip.

The original coaxial cable has been superseded by an 8-wire cable, with the familiar 8-pin RJ45 connector on each end, that connects each device to a "hub" or "switch", which broadcasts incoming data to the other connected cables. Your computer probably has a socket that accepts this standard connector, which also appears on devices like wireless base stations, cable modems and DSL modems that simulate Ethernet behavior.

Ethernets have a limited range, typically a few hundred meters. It is possible to extend the range with hardware devices that connect one segment to another and relay packets across the boundary; the smarter relays learn which hosts are on which computers and send packets only to places where they belong. All of this is hidden from ordinary users and is of concern only to system administrators who have to

figure out what's wrong when it doesn't work.

8.4 Wireless

Ethernet has one significant drawback: it needs wires, real physical equipment that snakes through walls, under floors, and sometimes (I speak from personal experience here) across the hall, down the stairs and through the dining room and kitchen on its way to the family room. A computer on an Ethernet can't be easily moved around, and if you like to lean back with your laptop on your lap, an Ethernet cable is a nuisance.

Fortunately, there's a way to have your cake and eat it too, with wireless. A wireless system (no wires, obviously) uses radio to carry data, so it can communicate from any place where there is enough signal. The normal range for these networks is tens to hundreds of meters. Unlike infrared, which is used for TV remotes, wireless doesn't have to be line of sight, but metal (like walls and partitions) and concrete (like floors) interfere, so in practice the range can be quite a bit less than in open air.

Wireless systems use electromagnetic radiation to carry signals. The radiation is a wave of a particular frequency measured in Hz (or more likely MHz or GHz for the systems we encounter, like the 103.7 MHz of a radio station). A modulation process imposes an information signal onto the carrier wave. For example, amplitude modulation ("AM") changes the amplitude or strength of the carrier to convey information, while frequency modulation ("FM") changes the frequency of the carrier around a central value. The received signal strength varies directly with power level at the transmitter, and inversely as the square of the distance from the transmitter to the receiver; because of this quadratic relationship, a receiver twice as far away as another will receive a signal only one quarter as strong. Radio waves are diminished by passing through various materials, some much more than others; for example metal does not allow any power through. Higher frequencies are generally absorbed more than lower frequencies, all else being equal.

Wireless systems operate under strict rules about what range of frequencies they can use—their *spectrum*—and how much power they can use for transmission. Spectrum allocation is a contentious process, since there are always many competing demands on it. Spectrum is allocated by government agencies like the Federal Communications Commission (FCC) in the United States, and international agreements are coordinated by the International Telecommunications Union or ITU, an agency of the United Nations.

The standard for wireless for computers today has the catchy name IEEE 802.11b/g/n, though you will often see the generic term Wi-Fi. IEEE is the Institute of Electrical and Electronics Engineers, a professional society that among other things establishes standards for a wide variety of electronic systems, including wireless. 802.11 is the number of the standard, which has several parts; 802.11b is the 11 Mbps version, 802.11g is 54 Mbps, and 802.11n is 600 Mbps. These nominal rates overstate the effective bandwidth that one might achieve in typical conditions, often

by a factor of two.

A wireless device encodes digital data into a form suitable for carrying on radio waves. A typical 802.11 system is packaged so it behaves like an Ethernet; you can replace an Ethernet by a wireless connection. The range is likely to be similar but there are no wires to contend with.

Wireless Ethernet devices operate at frequencies in the range 2.4-2.5 GHz and some forms of 802.11 operate at 5 GHz as well. When wireless devices all use the same narrow range of frequencies, contention is a distinct possibility. Worse, other devices use this same overcrowded frequency band, including some cordless telephones, medical equipment, and even microwave ovens. I have a tough time using one of my old laptops in the kitchen because 30 seconds of warming up a cup of coffee is enough to make the laptop drop its wireless connection.

I'm going to describe very briefly three wireless systems that are in widespread use. The first is Bluetooth, named after the Danish king Harald Bluetooth (c. 935-985 AD), which is intended for short-range ad hoc communications. It uses the same 2.4 GHz frequency band as 802.11 wireless. The range is from 1 to 100 meters, depending on power level, and the data rate is 1 to 3 Mbps. Bluetooth is used in wireless microphones, earphones, keyboards, mice, and game controllers, where low power consumption is critical.

RFID or radio-frequency identification is a low-power wireless technology that is used in electronic door locks, id tags for a variety of goods, automatic toll systems, implanted chips in pets, and even in documents like passports. The tag is basically a small radio receiver and transmitter that broadcasts identification information. Passive tags do not have batteries, but obtain power from an antenna that receives a signal broadcast by an RFID reader. RFID systems use a wide variety of frequencies, though 13.56 MHz is typical. RFID chips do make it possible to quietly monitor where things and people are. Chips in pets are popular—our cat has one—and as might be expected, there have been suggestions for implanting people as well, for both good and bad reasons.

The Global Positioning System (GPS) is an important one-way wireless system, commonly seen in car and phone navigation systems. GPS satellites broadcast precise time information, and a GPS receiver uses the time it takes for signals to arrive from three or four satellites to compute its position on the ground. But there is no return path. It's a common misconception that GPS somehow tracks its users. As a story in the New York Times had it a few years ago, "Some [cell phones] rely on a global positioning system, or GPS, which sends the signal to satellites that can pinpoint almost exactly where a user is." This is plain wrong. GPS-based tracking needs ground-based systems like cell phones to relay the location. A cell phone is in constant communication with base stations, as we will discuss next, so the phone can (and does) report your location continuously; with a GPS receiver it can do so more accurately.

8.5 Cell Phones

The most common wireless system for most people is the cell or mobile phone, now usually just "cell" or "mobile", a technology that barely existed in the 1980s, but which is now used by well over half of the world's population. Cell phones are a case study for the kinds of topics that are covered in this book—interesting hardware, software and of course communications, with plenty of social, economic, political and legal issues to go along with them.

The first commercial cell phone system was developed by AT&T in the early 1980s. The phones were heavy and bulky; advertisements of the time show users carrying a small suitcase for the batteries while standing next to a car that carries the antenna. Why "cell"? Because spectrum and radio range are both limited, a geographical area is divided into "cells," rather like imaginary hexagons, with a *base station* in the center of each cell; the base station is connected to the rest of the phone system. Phones talk to the closest base station, and when they move from one cell to another, a call in progress is handed off from the old base station to the new one; most of the time the user doesn't know this has happened.

Since received power falls off quadratically with distance, frequency bands within the allotted spectrum can be re-used in non-adjacent cells without interference; this was the insight that made it possible to use the limited spectrum effectively. In the diagram below, the base station at 1 will not share frequencies with base stations 2 through 7, but can share with 8 through 19 because they are far enough away to avoid interference. The details depend on factors like antenna patterns; this picture is an idealization.

Cell sizes vary, from a few hundred meters to a few tens of kilometers, depending on traffic, terrain, obstacles, and the like.

Cell phones are part of the regular telephone network, but are connected to it by radio (via the base stations) instead of wires. The essence of cell phones is mobility. Cell phones move long distances, often at high speed, and can appear in a new location with no warning at all, as they do when turned on again after a long flight.

Cell phones share a narrow radio frequency spectrum that has limited capacity for carrying information. Phones must operate with low radio power because they use

batteries, and by law their transmission power is limited to avoid interference with others. The bigger the battery, the longer the time between charges, but the bigger and heavier the phone; this is another of the tradeoffs that must be made.

Cell phone systems use different frequency bands in different parts of the world, but generally around 900 MHz. Each frequency band is divided into multiple channels, and a conversation uses one channel for each direction. Signaling channels are shared by all phones in the cell; they are also used for text messages and data in some systems.

Each phone has a unique identification number (which is not its phone number), analogous to an Ethernet address. When the phone is turned on, it broadcasts its id number. The nearest base station hears the phone and validates it with the home system. As the phone moves around, its location is kept up to date by base stations reporting to the home system; when someone calls the phone, the home system knows which base station is in contact with the phone.

Phones talk to the base station with the strongest signal. The phone continuously adjusts its power level, to use less power when it's closer to the base station; this preserves its own battery and creates less interference for other phones. Keeping in touch uses much less power than a phone call does, which is why standby times are measured in days while talk times are in hours. If the phone is in an area of weak or non-existent signal, it will use up its battery faster as it keeps looking for a base station.

Two different and completely incompatible techniques are used in the USA. AT&T and T-Mobile use GSM (Global System for Mobile Communications), a European system that divides the frequency band into narrow channels and then puts several conversations in successive time slots within each channel. GSM is the most widely used system in the rest of the world. Verizon and Sprint use CDMA (Code Division Multiple Access), a "spread spectrum" technique that spreads the signal out over the whole frequency band, but modulated with a different code pattern for each independent conversation; this means that most of the time there is no concentrated interference between conversations even though they are all sharing the same frequency band.

Both systems use data compression to squeeze the signal into as few bits as possible, then add error correction to cope with the inevitable errors of sending data over a noisy radio channel in the face of interference.

Cell phones raise difficult non-technical issues. Spectrum allocation is clearly one; in the US the government restricts the use of the allotted frequencies to at most two companies in each band. Thus spectrum is a very valuable resource.

Cell tower locations are another. Cell phone towers are not the most esthetic of outdoor structures, so many communities don't want towers within their borders, though of course they want high quality phone service. This picture shows a "Frankenpine," a cell tower imperfectly disguised as a tree.

Phone companies generally can get their way on tower placement and construction, but sometimes only after protracted legal wrangling. There are also intriguing questions when corporations rent cell phone antenna space from non-taxable entities like churches, whose steeples are often the highest existing structures in a neighborhood.

On the social level, cell phones have revolutionized many aspects of life. Teenagers send an average of 50 to 100 text messages a day (a high-profit revenue source for phone companies, since text messages are almost free to provide). Smartphones like iPhone and Android are transforming expectations—people use phones less for conversation than for all the other features—and are changing the software world as well. They have also become an important form of Internet access, since they provide browsing, mail, shopping, entertainment and social networking, albeit on a small screen. Indeed, there is some convergence between laptops and cell phones as the latter become more powerful while remaining highly portable. Cell phones are also absorbing the functions of other devices, from watches and address books to cameras, music and movie players, GPS navigators, and the like.

Some cell phone activities, like downloading movies, require a lot of bandwidth. Tablets are a part of this too, not yet often used for voice, but for all other cell phone activities. As cell phone and tablet use proliferates, the strain on existing facilities is only going to increase. In the USA, carriers have added usage-sensitive pricing and bandwidth caps to their data plans, ostensibly to restrain bandwidth hogs who download full-length movies, though the caps apply even when traffic is light. It's also

possible to "tether" a laptop to a cell phone, permitting the computer to use the cell phone as its connection to the Internet. Carriers don't like this either, and are moving towards limits and extra charges, since tethering can also use a lot of bandwidth.

8.6 Summary

Spectrum is a critical resource for wireless systems, and there is never enough for the demand. Many political and economic interests compete for spectrum space, with established interests like broadcasting and telephone companies resisting change. One way to cope is to use existing spectrum more efficiently. Cell phones used to use analog encoding but those systems were phased out long ago, in favor of digital systems that use much less bandwidth. Sometimes existing spectrum is repurposed; the switch to digital TV in the US in 2009 freed up a large block of spectrum space for other uses to fight over. Finally, it's possible to use higher frequencies, but that usually means shorter range.

Wireless is a broadcast medium, so anyone can listen in; encryption is the only way to control access and protect information in transit. The original standard for wireless encryption on 802.11 networks (WEP, or Wired Equivalent Privacy) proved to have significant weaknesses; newer encryption standards like WPA (Wi-Fi Protected Access) are better. Some people still run open networks, that is, with no encryption at all, in which case anyone nearby can not only listen in, but can use the wireless service for free. "War driving" refers to finding places where open networks are available. Anecdotally, the number of open networks seems lower today than a few years ago, as people have become more sensitive to the perils of eavesdropping and free riding.

Free Wi-Fi services in coffee shops, airports, and so on are exceptions; for example coffee shops want their customers to linger (and buy expensive coffee) while using their laptops. Information passing over those networks is open to all unless you use encryption, and not all servers will encrypt on demand. Certainly you should not do anything sensitive over a public network, given the existence of programs like Firesheep.

In spite of some problems, it seems likely that wireless will be the visible face of networking in the future, though wired connections will certainly remain a major component behind the scenes.

9

The Internet

We've talked about local network technologies like Ethernet and wireless. The phone system connects telephones worldwide. How do we do the same for computer networks? How do we scale up to connect one local network to another, perhaps to link all the Ethernets in a building, or to connect computers in my home to computers in your building in the next town, or to connect a corporate network in Canada to one in Europe? How do we make it work when the underlying networks use unrelated technologies? How do we do it in a way that expands gracefully as more networks and more users are connected, as distances grow, and as equipment and technology change over time?

The Internet is one answer to those questions, and it has been so successful that for most purposes, it has become *the* answer.

The Internet is neither a giant network nor a giant computer. It's a loose, unstructured, chaotic, ad hoc collection of networks, bound together by standards that define how networks and the computers on them communicate with each other.

How do we connect networks that have different physical properties—optical fiber, Ethernet, wireless—and that may be far away from each other? We need names and addresses so we can identify networks and computers, the equivalent of telephone numbers and a phone book. We need to be able to find routes between networks that are not directly connected. We need to agree on how information is formatted as it moves around, and on a great number of other less obvious matters like coping with errors, delays and overload; without such agreement, communication would be difficult or even impossible.

Agreements on data formats, who speaks first and what responses can follow, how errors are handled, and the like, are dealt with in all networks, and in the Internet especially, with *protocols*. "Protocol" has somewhat the same meaning as it does in

ordinary speech—a set of rules for interacting with another party—but network protocols are based on technical considerations, not social custom, and are much more precise than even the most rigid social structure.

It might not be entirely obvious, but the Internet strongly requires such rules: everything has to agree on protocols and standards for how information is formatted, how it is exchanged between computers, how computers are identified and authorized, and what to do when something fails. Agreeing on protocols and standards can be complicated, since there are many vested interests—companies that make equipment or sell services, entities that hold patents or secrets, and governments that might want to monitor and control what passes across their borders and between their citizens.

Some resources are in scarce supply; spectrum for wireless services is one obvious example. Names for web sites can't be handled by anarchy. Who allocates such resources and on what basis? Who pays what to whom for the use of limited resources? Who adjudicates the inevitable disputes? What legal system(s) will be used to resolve disputes? Indeed, who gets to make the rules? It might be governments, companies, industry consortia, and nominally disinterested or neutral bodies like the UN's International Telecommunications Union, but in the end everyone has to agree to abide by the rules.

It's clear that such issues can be resolved—after all, the telephone system works worldwide, connecting disparate equipment in different countries. The Internet is more of the same, though it's newer, larger, changing more rapidly and much more disorderly: it's a free-for-all compared to the controlled environments of traditional phone companies, most of which were either government monopolies or tightly regulated companies.

9.1 An Internet Overview

Before we dive into details, here's the big picture. The Internet began in the 1960s with attempts to build a network that could connect computers at widely separated geographical locations. The funding for much of this work came from the Advanced Research Projects Agency of the US Department of Defense, and the resulting network came to be called ARPANET. The first ARPANET message was sent from a computer at UCLA to one at Stanford, a distance of about 350 miles or 550 km, on October 29, 1969, which thus could be called the birthday of the Internet.

From the beginning the ARPANET was designed to be robust in the face of failure of any of its components and to route traffic around problems. The original ARPANET computers and technology were replaced over time. The network itself linked computer science departments and other research operations, and then spread into the commercial world in the 1990s, becoming "the Internet" somewhere along the way.

Today the Internet consists of many millions of loosely-connected, independent networks. Any given network may be connected to one or more other networks. Nearby computers are connected by local area networks, often Ethernets. Networks in turn are connected to other networks via *gateways* or *routers*—specialized computers that route packets of information from one network to the next. (Wikipedia says that a gateway is a more general device, and a router is a special case. I'll generally say "gateway" but the usage is not universal.) Gateways exchange routing information so they know, at least in a local way, what is connected and thus reachable.

Each network may connect many host computers, like PCs or Macs in homes, offices and dorm rooms. Individual computers within a home are likely to use wireless to connect to a router that in turn is connected to an *Internet Service Provider* or *ISP* by cable or DSL; office computers might use wired Ethernet connections.

As I mentioned in the previous chapter, information travels through networks in chunks called packets. A packet is a sequence of bytes with a specified format; different devices use different packet formats. Part of the packet will contain address information that tells where the packet comes from and where it is going; other parts of the packet will contain information about the packet itself like its length; and finally, usually the biggest part is the information being carried, the payload.

On the Internet, data is carried in packets called IP packets (for "Internet Protocol"). IP packets all have the same format. On a particular physical network, an IP packet may be transported in one or more physical packets. For instance, a large IP packet will be split into multiple smaller Ethernet packets because the largest possible Ethernet packet (1,500 bytes) is much smaller than the largest IP packet (65,000 bytes).

Each IP packet passes through multiple gateways; each gateway passes the packet on to a gateway that is closer to the packet's ultimate destination. As a packet travels from here to there, it might pass through 20 gateways, owned and operated by a dozen different companies or institutions, and quite possibly in different countries.

To make this work, we need several mechanisms.

Addresses: Each host computer must have an address that will identify it uniquely among all hosts on the Internet, rather like a telephone number. This identifying number, the *IP address*, is either 32 bits (4 bytes) or 128 bits (16 bytes). The shorter addresses are for version 4 of the Internet protocol ("IPv4"), while the longer are for version 6 ("IPv6"). IPv4 has been used for many years and is still dominant, but most IPv4 addresses have now been allocated, so the shift to IPv6 is accelerating.

IP addresses are analogous to Ethernet addresses. An IPv4 address is conventionally written as the values of its 4 bytes, each as a decimal number, separated by periods, as in 128.112.132.86 (which is `www.princeton.edu`). This odd notation is called *dotted decimal*; it's used because it's easier for humans to remember than pure decimal or hexadecimal would be. Here is that IP address in dotted decimal, binary, and hex:

decimal	128	.112	.132	.86
binary	`10000000`	`01110000`	`10000100`	`01010110`
hex	`80`	`70`	`84`	`56`

IPv6 addresses like `2620:0:1003:100c:9227:e4ff:fee9:05ec` are even less perspicuous, so I'll skip them for now.

A central authority assigns a range of consecutive IP addresses to the administrator of a network, which in turn assigns individual addresses to host computers on that network. Each host computer thus has a unique address that is assigned locally according to the network it is on. This address might be permanent for a desktop computer but for mobile devices, it's usually dynamic and changes at least every time the device reconnects to the Internet.

Names: A host that people will try to access directly must have a name that they can use, since few of us are good at remembering arbitrary 32-bit numbers, even in dotted decimal. Names are the ubiquitous forms like `www.stanford.edu` or `microsoft.com`, which are called *domain names*. A critical piece of Internet infrastructure, the *DNS* or *Domain Name System*, converts names to addresses.

Routing: There must be a mechanism for finding a path from source to destination for each packet. This is provided by the gateways mentioned above, which continuously exchange routing information among themselves about what is connected to what, and use that to forward each incoming packet onward to a gateway that is closer to its ultimate destination.

Protocols: Finally, there must be rules and procedures that spell out exactly how all of these and other components operate, so that information does get copied successfully from one computer to another.

The core protocol is called IP. It defines a uniform transport mechanism and a common format for information in transit. IP packets are carried by different kinds of network hardware using their own protocols.

Above IP, a protocol called TCP (Transmission Control Protocol) uses IP to provide a reliable mechanism for sending arbitrarily long sequences of bytes from a source to a destination. Above TCP, higher-level protocols use TCP to provide the services that we think of as "the Internet," like browsing, mail, file sharing, and so on. There are a wide variety of other protocols as well. For example, changing IP addresses dynamically is handled by a protocol called DHCP (Dynamic Host Configuration Protocol). All these protocols, taken together, define the Internet.

We will talk more about each of these topics shortly.

9.2 Domain Names and Addresses

Who makes the rules? Who controls the allocation of names and numbers? Who's in charge? For many years, the Internet was managed by informal cooperation among a small group of technical experts. Much of the Internet's core technology was developed by a loose coalition operating as the Internet Engineering Task Force (IETF) to produce designs and documents that describe how things should work. Technical specifications were (and still are) hammered out through regular meetings and frequent publications called "Requests for Comments," or RFCs, that eventually become standards. RFCs are available on the web (there are over 6,000 of them at the moment). Not all RFCs are deadly serious; check out RFC-1149, "A Standard for the Transmission of IP Datagrams on Avian Carriers," which was published on April 1, 1990.

Other aspects of the Internet are managed by a non-profit organization called ICANN, the Internet Corporation for Assigned Names and Numbers (`icann.org`), which provides technical coordination of the Internet. That includes assignment of names and numbers that have to be unique for the Internet to function, like domain names, IP addresses, and some protocol information. ICANN also accredits domain name *registrars*, who in turn assign domain names to individuals and organizations. ICANN began as an agency of the US Department of Commerce but is now an independent non-profit organization financed largely by fees from registrars and domain name registrations.

9.2.1 Domain Name System

The Domain Name System or DNS provides the familiar hierarchical naming scheme that brings us `berkeley.edu` or `cnn.com`. The names `.com`, `.edu` and so on, and two-letter country codes like `.us` and `.ca` are called *top-level domains*. Top-level domains delegate responsibility for administration and further names to lower levels. Thus, for example, Princeton University is responsible for `princeton.edu`, and can define sub-domain names within that range, such as `cs.princeton.edu` for the Computer Science department and `classics.princeton.edu` for the Classics department. Those in turn can define domain names like `www.cs.princeton.edu`; such hierarchical names can be extended indefinitely.

Domain names impose a logical structure, but need not have any geographical significance at all. For instance IBM operates in many countries but its computers are all included in `ibm.com`. It's possible for a single computer to serve multiple domains, which is common for companies that provide hosting services, and for a single domain to be served by many computers, as with large sites like Facebook and Amazon.

The lack of geographical constraints has some interesting implications. For example, the country of Tuvalu, a group of small islands in the South Pacific halfway between Hawaii and Australia, has the country code `.tv`. Tuvalu leases the rights to that country code to commercial interests, who will happily sell you a `.tv` domain. If the name you want has some potential commercial significance, say `news.tv`,

you'll likely have to pay handsomely for it. But `kernighan.tv` is under forty dollars a year. Other countries blessed by linguistic accidents include the Republic of Moldova, whose `.md` might appeal to doctors, and `.it` (Italy), which shows up in sites like `play.it`. Normally domain names are restricted to the 26 letters of English, digits and hyphens, but in 2009, ICANN approved some internationalized top-level domain names like

.中国

as an alternative to `.cn` for China.

9.2.2 IP addresses

Each network and each connected host computer must have an IP address so that it can communicate with others. An IPv4 address is a unique 32-bit quantity: only one host at a time can use that value. The addresses are assigned in blocks by ICANN, and the blocks are sub-allocated by the institutions that receive them. For example, Princeton has two blocks, 128.112.ddd.ddd and 140.180.ddd.ddd, where each ddd is a decimal number between 0 and 255. Each of these blocks allows up to 65,536 (2^{16}) hosts, or about 131,000 in all.

These two blocks of addresses have no numerical or geographical significance whatsoever. In the same way that area code 212 is New York City and 213 is Los Angeles, there's no reason to expect that adjacent blocks of IP addresses represent physically nearby computers, and there's no way to infer a geographical location from an IP address by itself, though it's often possible to deduce where an IP address is from other information. For example, DNS supports reverse lookups (IP address to name), and reports that 128.112.132.86 is `www.princeton.edu`, so it's reasonable to guess that's in Princeton, New Jersey, though the server could be somewhere else entirely.

A little more arithmetic shows that there are only about 4.3 billion IPv4 addresses. That's less than one per person on earth, so at the rate that people are using more and more communications services, something will run out. In fact, the situation is worse than it sounds, since IP addresses are handed out in blocks and thus are not used as efficiently as they might be (are there really 131,000 computers at Princeton?). The potential for running out is real and immediate: essentially all IPv4 addresses will be allocated by 2012.

Techniques for piggy-backing multiple hosts onto a single IP address provide a bit of breathing room. Home wireless routers generally provide a *network address translation* or *NAT* service, where a single external IP address can serve multiple internal IP addresses. If you have a NAT, all of your home devices appear externally to have the same IP address; hardware and software in the device handle the conversion in both directions.

Once the world shifts to IPv6, which uses 128-bit addresses, the pressure will be off—there are 2^{128} or about 10^{38} of those, so we won't run out in a hurry.

9.2.3 Root servers

The crucial DNS service is to convert names into IP addresses. The top-level domains are handled by a set of *root name servers* that know the IP addresses for all top-level domains, for example `mit.edu`. To determine the IP address for `www.cs.mit.edu`, one asks a root server for `mit.edu`'s IP address. That's sufficient to get to MIT, where one asks the MIT name server for `cs.mit.edu`, and that in turn leads to a name server that knows about `www.cs.mit.edu`.

DNS thus uses a divide-and-conquer approach to searching: an initial query at the top immediately eliminates most potential addresses from further consideration. The same is true at each level as the search works its way down the tree.

In practice, name servers maintain caches of names and addresses that have been looked up and passed through them recently, so a new request can often be answered from local information rather than going far away. If I access `kernighan.com`, the odds are strong that no one else has looked at that recently, and the local name servers may have to ask a root server for the IP address. If I use that name again soon, however, the IP address is cached nearby and the query runs faster. When I tried this, the first query took 1/3 of a second; a few seconds later the same query took one tenth of that, as did another query several minutes later.

You can do your own DNS experiments with commands like `nslookup`. Try the command

```
nslookup a.root-servers.net
```

In principle one could imagine having a single root server, but it would be a truly bad idea to have a single point of failure for such a critical system. So there are thirteen root servers, spread over the whole world, with about half in the USA. Some of these servers consist of multiple computers at widely separated locations that function like a single computer but use a protocol that routes requests to the nearest member of the group. Root servers run different software systems on different kinds of hardware, so they are less vulnerable to bugs and viruses than a monoculture would be. Nevertheless, from time to time root servers are subjected to coordinated attacks, and it's conceivable that some combination of circumstances could bring them all down together.

9.2.4 Registering your own domain

It's easy to register your own domain if the name you want isn't already taken. ICANN has accredited hundreds of registrars, so you can pick one, choose your domain name, pay for it, and it's yours. There are some restrictions—names are limited to 30 characters and until recently could only contain letters, numbers and hyphens—but there seem to be no rules against obscenities (easily verified by trying a few) or personal attacks, to the point where corporations and public figures are forced to preemptively acquire domains like `bigcorpsucks.com` in self-defense.

You need a *host* for your site, that is, a computer to hold and serve the content that your site will display to visitors. You also need a *name server* to respond with the IP address of your host when someone tries to find your domain's IP address. That's a separate component, though registrars usually provide the service or easy access to someone who does.

Competition has brought prices down; registration typically costs ten or twenty dollars initially and a similar amount per year to maintain, and a hosting service costs five or ten dollars a month for low-volume casual use; merely "parking" a domain with a generic page might well be free. Some hosting services are free, or have a nominal price if you're not doing much with it, or for a short time while you're testing the waters. For example, Google will set you up with registration and hosting for ten dollars per year.

Who owns a domain name? How are disputes resolved? What can I do if someone else has registered `kernighan.com`? That one is easy: not much except offer to buy it. For a name with commercial value, like `mcdonalds.com` or `apple.com`, courts and ICANN's dispute resolution policies have tended to favor the party with the clout. If your name is McDonald or Apple, you won't have much chance of wresting a domain away from one of them, and you might have trouble hanging on to it even if you got there first. (In 2003, a Canadian high school student named Mike Rowe set up a web site for his tiny software business at `mikerowesoft.com`. This brought a threat of legal action from a rather larger corporation with a similar-sounding name. Eventually the case was settled and Mr. Rowe chose a different domain name.)

9.3 Routing

Routing—finding a path from source to destination—is a central problem in any network.

Some networks use static routing tables that provide a next step in the path for all possible destinations. The problem with the Internet is that it's too large and too dynamic for static tables to work well. As a result, Internet gateways continuously refresh their routing information by exchanging information with adjacent gateways; this ensures that information about possible and desirable paths is always relatively up to date.

The sheer scale of the Internet requires a hierarchical organization to manage routing information. At the top level, some tens of thousands of *autonomous systems* provide routing information about the networks they contain; normally an autonomous system corresponds to a large Internet service provider (ISP). Within a single autonomous system, routing information is exchanged locally, but the system presents a unified single set of routing information to outside systems.

Physically, there is a sort of hierarchical organization as well, though not formally or rigorously. One accesses the Internet via an ISP, which is a company or other organization that in turn connects to other Internet providers. Some ISPs are tiny,

some are huge (those run by telephone and cable companies, for example); some are run by organizations like a company, university or government agency, while others offer access as a service for a fee—telephone and cable companies are typical examples. Individuals connect to their ISP by cable (common for residential service) or telephone; companies and schools offer Ethernet or wireless connections.

ISPs connect to each other through gateways, and for high volume between major carriers, there are "peering points" where network connections from several companies meet and physical connections are made between networks, so data from one network can be efficiently passed to another.

Some countries have comparatively few gateways that provide access into the country, and these can be used to monitor and filter traffic that the government considers undesirable.

You can explore routing with a program called `traceroute`, which is available on Unix systems (including Macs) and on Windows as `tracert`, and there are web-based versions as well. Here's a random example, the path from Princeton, New Jersey to a computer at the Vatican, edited for space.

```
$ traceroute michael.vatican.va
traceroute to michael.vatican.va (212.77.0.2),
               30 hops max, 40 byte packets
 1 ignition.CS.Princeton.EDU (128.112.155.129)  0.789 ms
 2 csgate.CS.Princeton.EDU (128.112.139.193)  0.756 ms
 3 gigagate1.Princeton.EDU (128.112.12.57)  0.771 ms
 4 vgate1.Princeton.EDU (128.112.12.22)  0.814 ms
 5 local1.princeton.magpi.net (216.27.98.113)  2.819 ms
 6 remote.internet2.magpi.net (216.27.100.54)  4.725 ms
 7 198.32.11.51 (198.32.11.51)  91.898 ms
 8 so-2-0-0.rt1.fra.de.geant2.net (62.40.112.9)  99.025 ms
 9 as1.rt1.gen.ch.geant2.net (62.40.112.21)  108.320 ms
10 as0.rt1.mil.it.geant2.net (62.40.112.34) 114.343 ms
11 garr-gw.rt1.mil.it.geant2.net (62.40.124.130) 114.505 ms
12 rt1-mi1-rt-mi2.mi2.garr.net (193.206.134.190) 114.691 ms
13 rt-mi2-rt-rm2.rm2.garr.net (193.206.134.230) 124.099 ms
14 83-103-94-182.ip.fastwebnet.it (83.103.94.182) 119.198 ms
15 michael.vatican.va (212.77.0.2) 119.088 ms
```

The round-trip times show the jump across the Atlantic and it's fun to try to figure out where the various gateways are from the cryptic encodings in their names. My guess is that this route goes through two or three European countries before it reaches Italy. This geographical reach might be surprising, and perhaps undesirable, depending on the nature of the traffic and the countries involved.

Unfortunately, security concerns have made `traceroute` less informative over time, as more and more sites choose not to provide the information necessary to make it work.

9.4 Protocols

A protocol defines the rules that govern how two parties interact with each other: whether one offers to shake hands, how deeply to bow, who goes through the door first, which side of the road to drive on, and so on. Most protocols in daily life are pretty informal, though the proper side of the road has the force of law.

The Internet has many protocols, two of which are absolutely fundamental. IP is the *Internet Protocol*; it defines how individual packets are formatted and transmitted. TCP, the *Transmission Control Protocol*, defines how IP packets can be combined into streams of data and connected to services. Together they are called TCP/IP.

Gateways route IP packets, though each physical network has its own format for carrying IP packets, and the gateway has to convert between the network format and IP as packets come in and go out.

Above the IP level, TCP provides reliable communication, so that users (programmers, really) don't have to think about packets, just streams of information. Most of the services that we think of as "the Internet" use TCP.

Above these are application-level protocols that provide those services, mostly built on TCP: the web, mail, file transfer, and the like. Thus there are several layers of protocols, each relying on the services of the one below and providing services to the one above. This is an excellent example of the layering of software described in Chapter 5, and one conventional diagram looks vaguely like a layered wedding cake:

UDP, the *User Datagram Protocol*, is another protocol at the same level as TCP. UDP is much simpler than TCP, and is used for data exchange that doesn't require two-way streams, just efficient packet delivery with a few extra features. DNS uses UDP, as do video streaming, VoIP, and some online games.

9.4.1 IP, the Internet Protocol

IP, the Internet Protocol, provides an unreliable connectionless packet delivery service. "Connectionless" means that each IP packet is self-contained and has no relationship to any other IP packet. IP has no state or memory: the protocol does not have to remember anything about a packet once it has been passed on to the next gateway.

"Unreliable" means both more and less than it appears to. IP is a "best effort" protocol that makes no guarantees about how well it delivers packets—if something

goes wrong, that's tough. Packets may be lost or damaged; they may be delivered out of order, and they may arrive too fast to be processed or too slowly to be useful. In actual use, IP is very reliable. When a packet does go astray or get damaged, however, there's no attempt at recovery. It's like dropping a postcard into a mailbox in some strange place: the postcard probably gets delivered though it might be damaged en route. Sometimes it doesn't arrive at all, and sometimes delivery takes much longer than you expected. (There is one IP failure mode that postcards don't share: an IP packet can be duplicated, so the recipient gets more than one copy.)

IP packets have a maximum size of about 65 KB. Thus a long message must be broken up into smaller chunks that are sent separately, then reassembled at the far end. An IP packet, like an Ethernet packet, has a specified format. This shows some of the IPv4 format; the IPv6 packet format is analogous, but the source and destination addresses are each 128 bits long.

version	type	hdr len	total len	TTL	source address	destination address	error check	data (up to 65KB)

One interesting part of the packet is the *time to live*, or TTL. The TTL is a one-byte field that is set to an initial value by the source of the packet, and is decreased by one by each gateway that handles the packet. If the count ever gets down to zero, the packet is discarded and an error packet is sent back to the originator. A typical trip through the Internet might involve 15 to 20 gateways, so a packet that takes 255 hops is clearly in trouble, probably in a loop. The TTL field doesn't eliminate loops but it does prevent individual packets from living forever.

9.4.2 TCP, the Transmission Control Protocol

Higher-level protocols synthesize reliable communications from this unreliable substrate. The most important of these is TCP, the transmission control protocol. To its users, TCP provides a reliable two-way stream: put data in at one end and it comes out the other end, with little delay and low probability of error, as if it were a direct wire from one end to the other.

I won't go into the details of how TCP works—there are a lot of them—but the basic idea is simple enough. A stream of bytes is chopped up into pieces and put into TCP packets or *segments*. A TCP segment contains not only the real data, but also a "header" with control information, including a sequence number, so the recipient knows which part of the stream each packet represents; in this way a lost segment will be noticed and can be resent. Error-detection information is included, so if a segment is damaged, that is likely to be detected as well. Each TCP segment is transmitted in an IP packet. This diagram shows the contents of a TCP segment header, which will be sent inside an IP packet along with the data:

source port	destination port	sequence number	acknowledgment	error check	other info

Each segment must be acknowledged, positively or negatively, by the recipient. For each segment that I send to you, you must send me an acknowledgment that you got it. If I don't get the acknowledgment after a decent interval, I must assume that the segment was lost and I will send it again. Similarly, if you are expecting a particular segment and haven't received it, you must send me a *negative acknowledgment* ("Segment 27 has not arrived") and I will know to send it again.

Of course if the acknowledgments themselves are lost, the situation is even more complicated. TCP has a number of timers that determine how long to wait before assuming that something is wrong. If an operation is taking too long, recovery can be attempted; ultimately, a connection will "time out" and be abandoned. (You've probably encountered this with unresponsive web sites.) All of this is part of the protocol.

The protocol also has mechanisms to make this work efficiently. For example, a sender can send packets without waiting for acknowledgments of previous packets, and a receiver can send a single acknowledgment for a group of packets; if traffic is flowing smoothly, this reduces the overhead due to acknowledgments. If congestion occurs and packets begin to get lost, however, the sender quickly backs off to a lower rate, and only slowly builds back up again.

When a TCP connection is set up between two host computers, the connection is not just to a specific computer but to a specific *port* on that computer. Each port represents a separate conversation. A port is represented by a two-byte (16-bit) number, so there are 65,536 possible ports, and thus in principle a host could be carrying on 65,536 separate TCP conversations at the same time.

A hundred or so "well known" ports are reserved for connections to standard services. For instance, web servers use port 80, and mail servers use port 25. If a browser wants to access `yahoo.com`, it will establish a TCP connection to port 80 at Yahoo, but a mail program will use port 25 to access a Yahoo mailbox. The source and destination ports are part of the TCP header that accompanies the data.

There are many more details but the basic ideas are no more complicated than this. TCP and IP were originally designed around 1973 by Vint Cerf and Bob Kahn, who shared the 2004 Turing Award for their work. Although the TCP/IP protocols have undergone refinements, they remain essentially the same even though network sizes and traffic speeds have grown by many orders of magnitude. The original design was remarkably well done, and today TCP/IP handles most of the traffic on the Internet.

9.5 Higher-level Protocols

TCP provides a reliable two-way stream that transfers data back and forth between two computers. Internet services and applications use TCP as the transport mechanism but have their own protocols specific to the task at hand. For example, HTTP, the HyperText Transfer Protocol, is a particularly simple protocol used by web browsers and servers. When I click on a link, my browser opens a TCP/IP connection to port 80 on the server, say `amazon.com`, and sends a short message that requests a specific page. In the following picture, the browser is the client application at the top left; the message goes down the chain of protocols, crosses the Internet (normally with many more steps), and then back up to the matching server application at the far end. The return path need not be the same as the original path.

At Amazon, the server prepares the page, then sends it to me along with a small amount of additional data, perhaps information about how the page is encoded. My browser reads this response and uses the information to display the contents. You can try this for yourself by using Telnet, a TCP service for establishing a remote login session on another machine. Normally Telnet uses port 23, but it can be aimed at other ports as well. Type these lines into a Mac Terminal or Windows command window:

```
telnet www.cnn.com 80
GET / HTTP/1.0
    [type an extra blank line here]
```

and back will come nearly 100,000 characters that a browser would use to display the page.

`GET` is one of a handful of HTTP requests, "`/`" asks for the default file at the server, and `HTTP/1.0` is the protocol name and version. We'll talk more about HTTP and the Web in the next chapter.

Given the Internet as a carrier of information, what can we do with it? In this section, we'll look at a handful of the earliest applications that used the nascent Internet; all of them date from the early 1970s, yet they are still used today, a tribute to their design and utility. All use TCP/IP to carry the bytes. All are command-line programs, and although mostly they are simple, they are aimed at comparative experts, not casual users.

9.5.1 FTP: File Transfer Protocol

One of the purposes of the early Internet was to provide a way for researchers to easily access information at other sites; this might be experimental data and test results or programs for creating and analyzing such data. The tool for doing this was called FTP, the *File Transfer Protocol*.

FTP illustrates many of the basic ideas. The operations are simple: open a connection to another site, get a file from that site or send a file, close the connection. A handful of other commands provide for navigating the remote file system. To show how simple FTP is, here's a dialog with an FTP server, edited for compactness. The commands I typed are in ***this font***.

```
$ ftp ftp.cs.princeton.edu
Connected to ftp.cs.princeton.edu.
220 ftp.CS.Princeton.EDU NcFTPd Server ready.
Name (ftp.cs.princeton.edu:bwk): anonymous
331 Guest login ok, send your e-mail address as password.
Password: [I typed xxx here]
230 Logged in anonymously.
Remote system type is UNIX.
ftp> ls     (dir also works)
150 Data connect accepted from 72.85.133.214:58230
-rw-r-r-  1 ftpuser  ftpusers  255749 May 14 15:07 875.pdf
-rw-r-r-  1 ftpuser  ftpusers  261940 Jun 18 11:35 876.pdf
-rw-r-r-  1 ftpuser  ftpusers  548998 Jun 21 09:31 877.pdf
226 Listing completed.
ftp> get 876.pdf
150 Data connection accepted from 72.85.133.214:58232;
       transfer starting for 876.pdf (261940 bytes).
226 Transfer completed.
261940 bytes received in 00:01 (132.92 KiB/s)
ftp> quit
```

Note that as configured here, FTP permits anonymous access and doesn't check the password, though the server does know the client's IP address, and it won't let me access non-public files without authenticating me as someone with permission.

FTP has a `help` command, as do most of the early protocols, so you can learn to use it by experiment.

9.5.2 Telnet: remote login

The second early tool is a program called Telnet, which provides a way to access a remote machine as if one were directly connected to it; I showed an example of its use earlier. Telnet accepts keystrokes from the client and passes them to the server as if they had been typed there directly; it intercepts the server's output and sends it back to the client. Telnet makes it possible to log in to any computer on the net and use it as if it were a local machine. Embellishments allow for file copying, directing keystrokes to the local machine instead of the remote, and so on, but the basic usage is simple. Although Telnet was originally meant for remote login, it can be used to

connect to any port and thus can be used for simple experiments with other protocols. For instance, here's how to use it to do a search:

```
telnet www.google.com 80
GET /search?q=whatever
    [type an extra blank line here]
```

This produces about 50,000 bytes of output, mostly JavaScript, but the search results are visible within if one looks carefully.

Telnet, like FTP, is open. If the remote system accepts logins without a password, none is requested. If the remote asks for a password, Telnet sends the client's password in the clear, so anyone watching the flow of data could see it. This total lack of security is one reason why Telnet is rarely used now except in special circumstances where security does not matter. Its descendant SSH (Secure Shell) is widely used, however, since it encrypts all traffic in both directions and thus can exchange information securely; it uses port 22.

9.5.3 SMTP: Simple Mail Transfer Protocol

The third example is SMTP, the *Simple Mail Transfer Protocol*. We usually send and receive mail with a browser or a standalone program like Outlook or Mac Mail. But like many other things on the Internet, several layers lie beneath this surface, each enabled by programs and protocols. Mail involves two basic kinds of protocols. SMTP is used to exchange mail with another system. It establishes a TCP/IP connection to port 25 at the recipient's mail computer, and uses the protocol to identify the sender and recipient, and to transfer messages. SMTP is text based; you can run it with Telnet on port 25 if you want to see how it works, though there are enough security restrictions that you might have trouble using it even locally on your own computer. Here's a sample dialog (edited for compactness) from an actual session with a local system, in which I sent myself mail as if from someone else (in effect, spam):

```
$ telnet localhost 25
Connected to localhost (127.0.0.1).
220 localhost ESMTP Sendmail 8.13.8/8.13.1
HELO localhost
250 localhost Hello localhost [127.0.0.1], pleased to meet you
mail from:liz@royal.gov.uk
250 2.1.0 liz@royal.gov.uk... Sender ok
rcpt to:bwk@princeton.edu
250 2.1.5 bwk@princeton.edu... Recipient ok
data
354 Enter mail, end with "." on a line by itself
Dear Brian --
Would you like to be knighted?  Please let me know.
ER
.
250 2.0.0 p4PCJfD4030324 Message accepted for delivery
```

This nonsensical (or at least improbable) message was duly delivered to my mailbox:

Security restrictions make it unlikely that you can run this on your own system, and if you can it's almost sure to get you into trouble if you send mail to anyone other than yourself.

Since SMTP requires that mail messages be ASCII text, a standard called MIME (Multipurpose Internet Mail Extensions, in effect another protocol) describes how to convert other kinds of data into text and how to combine multiple pieces into a single mail message. This is the mechanism used to include mail attachments like pictures and music, and it's also used by HTTP.

Although SMTP is an end-to-end protocol (as are FTP and Telnet), the TCP/IP packets typically pass through 15 to 20 gateways on their way from source to destination. It is totally possible for any gateway along the path to examine the packets and make copies for leisurely inspection. SMTP itself can make copies of the contents, and mail systems keep track of contents and headers. If the contents are to be kept private, they must be encrypted at the source. Bear in mind that encrypting the contents does not conceal the identities of sender and receiver.

As an aside, SMTP transfers mail from source to destination but it has nothing to do with access to the mail thereafter. Once the mail arrives at the target computer, it normally waits there until the recipient retrieves it. Two mail-retrieval protocols are in widespread use, POP and IMAP.

POP, the Post Office Protocol, handles the case where you want to move your mail from the system where it has been collected to your own computer. You retrieve your mail from a server, copy it to your computer to read, and delete the copy on the server.

By contrast, IMAP (Internet Message Access Protocol) handles the case where your mail remains on a server and you access it from multiple places. IMAP ensures that your mail is in a single place and that your mailbox is in a consistent state at all times. It permits multiple simultaneous readers and updaters of the mailbox. IMAP is the best choice when you may be reading mail from several places, for instance browsers and cell phones, since there's no need to make multiple copies of messages or to transfer them around among computers.

It's common for mail to be handled "in the cloud" by systems like Gmail or Yahoo mail. Underneath, these use SMTP for transfer and behave like IMAP for client access. I'll talk about cloud computing in Chapter 11.

9.5.4 File sharing and peer-to-peer protocols

In June 1999, Shawn Fanning, a freshman at Northeastern University, released Napster, a program that made it dead easy for people to share music compressed in MP3 format. Fanning's timing was great. Audio CDs were ubiquitous but expensive. Personal computers were fast enough to do MP3 encoding and decoding, and the algorithms were widely available. Bandwidths were high enough that songs could be transferred across a network reasonably quickly, especially for college students on dormitory Ethernets. Fanning's design and implementation were well done, and Napster spread like wildfire. A company was formed to provide the service in mid 1999, and claimed to have 80 million users at its peak. The first lawsuits were filed later in 1999, alleging theft of copyrighted music on a grand scale, and a court decision put Napster out of business by mid 2001. From nothing to 80 million users to nothing in barely two years was a vivid illustration of the then-popular phrase "Internet time." The original Napster is long dead, but the company has been reconstituted and the name lives on in a pay service.

To use Napster, one had to download a Napster client to run on one's own computer. The client established a local folder for the files that would be sharable. When the client subsequently logged in to a Napster server, it uploaded the *names* of shareable files and Napster updated a central directory of currently available filenames. The central directory was continually updated: as new clients connected, their filenames were added, and when a client failed to respond to a probe, its filenames were de-listed.

When a user searched the central directory for song titles or performers, Napster supplied a list of other users who were currently online and willing to share those files. When a user selected a supplier, Napster arranged the contact (sort of like a dating service) by providing an IP address and a port number, and the client program on the user's computer made the contact and retrieved the file. The supplier and the consumer reported status to Napster, but the central server was otherwise "not involved" because it never touched any of the music itself.

We are used to the client-server model, where a web browser (the client) requests something from a web site (the server). Napster was an example of a different model. It provided a central directory that listed the music that was currently available for sharing, but the music itself was only stored on user machines, and when a file was transferred, it went directly from one Napster user to another, rather than through the central system. Hence the organization was called *peer-to-peer*, with the sharers being the peers. Because the music itself was only stored on the peer computers, never on the central server, Napster hoped to sidestep copyright issues, but the court was not persuaded by that legal fine point.

The Napster protocol used TCP/IP, so it was in effect a protocol at the same level as HTTP and SMTP. Without in any way deprecating Fanning's work, which was very neat indeed, Napster is a simple system when the infrastructure of the Internet, TCP/IP, MP3, and tools for building graphical user interfaces are all in place already.

File sharing was enormously popular, and a host of alternatives arose well before Napster was put out of business; a few were even used to share non-copyrighted material. Some differed in a couple of crucial ways. First, Napster had been vulnerable because it used a central server for the directory of currently available files, so it could be shut down by shutting down that server. Second, it was based in California and thus was subject to US law. The newer file-sharing programs did not have those limitations. Most operated from outside the US. More important, they used a distributed directory scheme in which the information about sharable files was spread over a number of computers, and shutting down a subset was not sufficient to shut down the service. At the same time, these services began to share other kinds of material besides music, including movies, software, and pornography, which increased their appeal in some circles.

Some sharing programs like Kazaa included *adware*, programs that displayed advertising on client machines. This somewhat dampened the enthusiasm for file sharing, but not by much. (In a charming bit of irony, when someone copied Kazaa's code, stripped out all the adware, and made that available as Kazaa Lite, the distributors of Kazaa complained about copyright violation.)

For some years, a company called Grokster.com provided a major file-sharing client similar to Kazaa. Grokster claimed that it was only providing a program with legitimate uses, and that any copyright infringement was someone else's responsibility. Grokster was sued by media companies, and the case was ultimately decided by the US Supreme Court in 2005 in favor of copyright holders; the court ruled that Grokster was liable for "vicarious" copyright infringement, and the company went out of business immediately. The site `grokster.com` still exists; it's worth a visit to see the ominous warning about copyright violations.

LimeWire, another popular sharing site, was shut down by a court order late in 2010, and displays a similar warning.

Most current file sharing, whether legal or not, uses a peer-to-peer protocol called BitTorrent, which was developed by Bram Cohen in 2001. BitTorrent is particularly good for sharing very large popular files like movies and TV programs, because each site that starts to download a file with BitTorrent must also begin to upload pieces of the file to others who want to download. Files are found by searching distributed directories analogous to those used by Kazaa, and a small "torrent file" is used to identify a tracker that maintains a record of who has sent and received which blocks. BitTorrent users are vulnerable to detection, since the protocol requires that downloaders also upload, and thus they are easily identified in the act of making allegedly copyrighted material available.

File sharing is a good case study for the kinds of issues that this book is about, combining almost every major topic. On the hardware side, it deals with peripherals for sound and network connections, devices that were non-existent or at least expensive in the early 1990s. Representation of analog information in digital form is at the heart of audio CDs and MP3. The software includes applications and operating systems, client-server and peer-to-peer models, directories and files on local and remote

machines, user interfaces, and of course algorithms for searching, sorting and compression. The communications part uses the Internet and web browsers; it has to worry about bandwidth, compression, and error detection. Security and privacy enter whenever you let someone else run their program on your computer.

9.5.5 An aside on copyright

The entertainment industry, through trade groups like the Recording Industry Association of America (RIAA) and the Motion Picture Association of America (MPAA), has been active in pursuing legal redress against sharing of copyrighted material. This has included lawsuits and threats of legal action against large numbers of alleged copyright infringers, and intense lobbying efforts in support of legislation to make such activities illegal. It's not clear what's going to happen here, though it appears that businesses that charge a reasonable price for guaranteed quality copies of music that people want can make money; Apple's iTunes Store is a highly successful example.

In the USA, the primary law for digital copyright issues is the 1998 Digital Millennium Copyright Act or DMCA, which made it illegal to circumvent copyright protection techniques on digital media, including distribution of copyrighted material on the Internet; some other countries have analogous laws. The DMCA is the legal mechanism used by the entertainment industry to go after copyright violators. The DMCA provides a "safe harbor" provision for Internet service providers: if an ISP is notified by a copyright holder that a user of the ISP is supplying copyrighted material, the ISP itself is not liable for copyright violation if it requires the infringer to remove the copyrighted material. This safe harbor provision is important at universities, where the university is the ISP for its students and faculty. Thus every university has an office that deals with allegations of infringement and posts prominent links to it, like this:

> In accordance with the Digital Millennium Copyright Act ("DMCA"), Pub. L. 105-304, Harvard has designated an agent to receive notification of alleged copyright infringement occurring in the harvard.edu domain. If you believe that your copyrighted work is being infringed, notify our designated agent specified below.

The DMCA is also invoked (on both sides) in legal tussles between more evenly matched opponents. In 2007 Viacom, a major film and TV company, sued Google for $1 billion over copyrighted material available on YouTube, a Google service. Viacom said that the DMCA was not meant to enable wholesale theft of copyrighted material. Part of Google's defense was that it responds appropriately to DMCA takedown notices when they are properly presented, but that Viacom had not done so. A judge ruled in favor of Google in June 2010.

A relatively recent development is the enlistment of the US government in enforcing copyright issues. In a controversial series of actions, the US Immigration and Customs Enforcement, or ICE, a branch of the Department of Homeland Security, seized over a hundred domains because they were allegedly hosting copyrighted

material or offering counterfeit goods for sale. Domain owners can appeal a seizure, but if they really are on the shady side, it's easier to move to a new name and a registrar outside the USA.

9.5.6 Summary of high-level protocols

What these protocols have in common is that they move information between computer programs, using the Internet as a dumb network that copies bytes efficiently from one computer to another without trying to interpret or process them. This is an important property of the Internet: it's "dumb" in the sense that it leaves the data alone. Less pejoratively, this is known as the *end-to-end principle*: the intelligence lies at the end points, that is, with the programs that send and receive the data. This contrasts with traditional telephone networks, where all the intelligence has been in the network, and the end points, like old-fashioned telephones, were truly dumb, providing little more than a way to connect to the network and relay voice.

The "dumb network" model has been highly productive, since it means that anyone with a good idea can create smart end points and rely on the network to carry the bytes; waiting for a telephone or cable company to implement or support the good idea would not work. As might be expected, carriers would be happier with more control, especially in the evolving mobile area, where carriers want control but most of the innovation comes from elsewhere.

Smartphones like iPhone and Android are merely computers that usually communicate over the telephone network instead of the Internet. Carriers would love to make money from services running on those phones, but basically can only derive revenue from carrying data. In the early days, most cell phones had a flat rate per month for data service, but at least in the USA that is changing to a structure that charges more for more usage. For high-volume services like downloading movies, this might be reasonable, but it seems less defensible for services like texting, which costs the carriers almost nothing since the bandwidth is so tiny.

Finally, notice how the early protocols and programs trust their users. Telnet and FTP send passwords in the clear. FTP supports, indeed encourages, anonymous transfer. For a long time, SMTP would relay mail from anyone to anyone without restricting sender or receiver in any way. This "open relay" service was great for spammers—if you don't need a direct reply, you can lie about your source address, which makes fraud and denial of service attacks easy.

In short, the Internet protocols and the programs built upon them were designed for an honest, cooperative and well-meaning community of trusted parties. That is not remotely what today's Internet is like, so on a variety of fronts, we are playing catch-up on security of information as it flows and authentication of the parties involved.

9.6 Bandwidth

Data on a network flows no faster than the slowest link permits. Traffic can slow down at many places, so bottlenecks are common at the links themselves and in the processing at computers along the way. The speed of light is also a limiting factor. Signals propagate at 300 million meters per second in vacuum (about one foot per nanosecond, a useful factoid) and somewhat slower in electronic circuits, so it takes time to get a signal from one place to another even if there are no other delays. At the speed of light in a vacuum, the time to travel from the east coast of the US to the west coast (about 2,500 miles, or 4,000 km) is about 13 msec. For comparison, a typical Internet delay for the same trip, passing through a dozen routers, is about 40 milliseconds; western Europe is about 50 msec, Beijing is 140 msec, and Sydney is 110 msec.

We encounter a wide range of bandwidths in day to day life. My first modem ran at 110 bits/second, which was fast enough to keep up with a mechanical typewriter-like device; today's modems run at about 56 Kbps, as mentioned earlier. Home wireless systems running 802.11 can in theory operate up to 600 Mbps, though in practice the rates will be much lower. Wired Ethernets are commonly 1 Gbps. A DSL or cable connection between your home and your ISP might be a few megabits per second; both services will be slower upstream than downstream. Your ISP is likely connected to the rest of the Internet through a fiber optic cable that could in principle provide 100 Gbps or more, though it's likely to be significantly lower.

Cell phones are slow by comparison with almost all of these. "3G" systems for smartphones are supposed to provide 2 Mbps if the phone is stationary and 384 Kbps if it is moving, but the term 3G is imprecisely specified, leaving room for optimistic advertising.

The IP protocol itself makes no guarantees about bandwidth: as a best-effort service, it doesn't even promise that information will arrive, let alone how fast. The Internet makes extensive use of caching to try to keep things moving; we've already seen this in the discussion of name servers. Web browsers also cache information, so if you try to access a page or an image that you've looked at recently, it may come from a local cache rather than from the network. Major Internet servers also use caching to speed up responses. Companies like Akamai provide content distribution services for other companies like Yahoo; this amounts to caching content closer to the recipients. (Search engines also maintain large caches of pages they have found during their crawl of the web; that's a topic for Chapter 10.)

9.7 Compression

One way to make better use of available memory and bandwidth is to compress data. The basic idea of compression is to avoid storing or sending redundant information, that is, information that can be recreated or inferred when retrieved or at the other end of a communications link. The goal is to encode the same information in fewer bits. Some bits carry no information and can be removed entirely; some bits

can be computed from others; some don't matter to the recipient and can be safely discarded.

Consider English text like this book. In English, letters do not occur with equal frequency; "e" is most common, followed by "t", "a", "o", "i", and "n" in approximately that order; at the other end, "z", "x" and "q" are much less common. In the ASCII representation of text, each letter occupies one byte or 8 bits. One way to save a bit (so to speak) is to use only 7 bits; the 8th bit (that is, the leftmost) is always zero in US ASCII anyway and thus carries no information. But we can do even better by using fewer bits to represent the most common letters and if necessary more bits to represent infrequent letters, reducing the total bit count significantly. This is analogous to the approach taken by Morse code, which encodes "e" as a single dot, "t" as a single dash, but "q" as dash-dash-dot-dash.

Let's make this more concrete. *Pride and Prejudice* is somewhat over 97,000 words, or 550,000 bytes. The most common character is actually the space between words: there are over 91,000 spaces between words. The next most common characters are `e` (55,100), `t` (36,900), and `a` (33,200); at the other end, `X` occurs only once, and `Z` three times. The least common lower-case letters are `j` 469 times, `q` 509 times, and `z` and `x` around 700 each. Clearly if we used two bits each for space, `e`, `t`, and `a`, we'd save a great deal and it wouldn't matter if we had to use more than 8 bits for `q`, `x`, `z`, `j` and other infrequent letters. A technique called Huffman coding does this systematically, finding the best possible compression that encodes individual letters; it compresses *Pride and Prejudice* by 44 percent, to 310,000 bytes, so in fact the average letter occupies only about four bits.

It's possible to do better by compressing larger chunks than single letters, for example entire words or phrases, and by adapting to the properties of the source document. Several algorithms do this well; the widely used Zip program squeezes the book by 64 percent, down to 202,000 bytes. The best algorithm that I know of, in a Unix program called `bzip2`, gets it down to 145,000 bytes, barely one quarter of its original size.

All of these techniques do *lossless compression*—the compression loses no information so uncompressing restores the original source exactly. Although it might seem counter-intuitive, there are situations where it's not necessary to reproduce the original input exactly—an approximate version is good enough—and in those situations, a *lossy* compression technique can give even better results. Lossy compression is most often used for content that is going to be seen or heard by people. Consider compressing an image from a digital camera. The human eye can't distinguish colors that are very close to each other, so it's not necessary to preserve the exact colors of the input; fewer colors will do and those can be encoded in fewer bits. Similarly it's possible to discard some fine detail; the resulting image will not be quite as sharp as the original but the eye won't notice. The same is true for fine gradations of brightness. The JPEG algorithm that produces the ubiquitous `.jpg` images uses this kind of *perceptual coding* to compress a typical image by a factor of 10 or more without significant degradation as seen by our eyes. Most programs that produce JPEG allow

some control over the amount of compression; "higher quality" implies less compression.

Similar reasoning leads to the MPEG family of algorithms for compressing movies and TV. Individual frames can be compressed as in JPEG but in addition, it's possible to compress a sequence of blocks that do not change much from one frame to the next. It's also possible to predict the result of motion and encode only the change, and even to separate a moving foreground from a static background, using fewer bits for the latter.

MP3 is the audio part of MPEG; it's a perceptual coding scheme for compressing sound. Among other things, it takes advantage of the fact that the human ear can't hear frequencies higher than about 20 KHz (a number that falls off as one gets older), and that louder sounds mask softer sounds. MP3 encoding generally compresses standard CD audio by a factor of about 10.

Cell phones use a great deal of compression, focusing on the human voice. Voice can be squeezed significantly more than arbitrary sound because it has a narrow range of frequencies and it's produced by a vocal tract that can be modeled for individual speakers; knowing the characteristics of an individual speaker makes better compression possible.

The idea in all forms of compression is to reduce or eliminate bits that do not convey their full potential information content, by encoding more frequently occurring elements in fewer bits, by building dictionaries of frequent sequences, or by encoding repetitions with counts. Lossless compression allows the original to be reconstructed perfectly; lossy compression discards some information that is not needed by the recipient, and is a tradeoff between quality and compression factor.

It's possible to make other tradeoffs as well, for example between compression speed and complexity versus decompression speed and complexity. When a digital television picture breaks up into blocks or the sound starts to become garbled, that's a result of the decompression algorithm failing to recover from some input error, probably because the data didn't arrive fast enough. And no matter what the algorithm, some inputs will not shrink, as you can see by imagining applying the algorithm to its own output.

9.8 Error Detection and Correction

If compression is the process of removing redundant information, error detection and correction is the process of adding carefully controlled redundancy that makes it possible to detect and even correct errors.

Some common numbers have no redundancy and thus it's not possible to detect when an error might have occurred. For example, US Social Security numbers have 9 digits and almost any 9 digit sequence could be a legitimate number. (This is helpful when someone asks for yours when they don't need it; just make one up.) But if some extra digits were added, or if some possible values were excluded, it would be possible to detect errors.

Credit card and ATM numbers are 16 digits long but not every 16-digit number is a valid card number. They use a checksum algorithm invented by Peter Luhn at IBM in 1954 that detects single-digit errors and most transposition errors in which two digits are interchanged; those are the most common kinds of errors that occur in practice.

The algorithm is easy: starting at the rightmost digit, multiply each successive digit alternately by 1 or by 2. If the result is greater than 9, subtract 9. Add the resulting digits; the sum must be divisible by 10. Test it on your own cards, and on 4417 1234 5678 9112, a number that some banks use in advertisements. The result on the latter is 69, so this is not a valid number, but changing the final digit to 3 makes it valid.

The 10- or 13-digit ISBNs on books also have a checksum that defends against the same kinds of errors, using a similar algorithm. Barcodes and postal codes also use checksums.

Those algorithms are special-purpose and aimed at decimal numbers. A *parity code* is the simplest example of error detection that is applied to bits. A single additional *parity bit* is attached to each group of bits; the parity bit's value is chosen so that the total number of (say) 1-bits in the group is even. That way if a single bit error occurs, the receiver will see an odd number of 1-bits and know that something has been damaged. Of course this doesn't identify which bit was in error, nor does it detect the occurrence of two errors.

For example, here are the first half-dozen upper case letters in ASCII, written in binary. The even-parity column replaces the (unused) leftmost bit by a parity bit that makes the parity even (each byte has an even number of 1-bits), while in the odd-parity column each byte has an odd number of 1-bits. If any bit in any of these is flipped, the resulting byte does not have the correct parity, and thus the error can be detected. If a few more bits were used, a code could correct single bit errors.

letter	original	even parity	odd parity
A	01000001	01000001	11000001
B	01000010	01000010	11000010
C	01000011	11000011	01000011
D	01000100	01000100	11000100
E	01000101	11000101	01000101
F	01000110	11000110	01000110
...			

Error detection and correction are used extensively in computing and communications. Error correction codes can be used on arbitrary binary data, though different algorithms are used for different kinds of likely errors. For example, memories defend against single-bit errors at random places; CDs and DVDs use codes that protect against long runs of damaged bits; cell phones defend against short bursts of noise. As with compression, error detection can't solve all problems, and there will always be cases where an error converts one legal pattern into another legal pattern.

9.9 Summary

There are only a handful of basic ideas behind the Internet; it is remarkable how much can be accomplished with so little mechanism (though with a great deal of engineering).

The Internet is a packet network: information is sent in standardized individual packets that are routed dynamically through a large and changing collection of networks. This is a different model from the telephone system's circuit network, where each conversation has a dedicated circuit, conceptually a private wire between the two talking parties.

The Internet assigns a unique IP address to each host that is currently connected, and hosts on the same network share a common IP address prefix. Mobile hosts like laptops will likely have a different IP address each time they are connected, and the IP address might change as the laptop moves around. The Domain Name System is a large distributed database that converts names to IP addresses and vice versa.

Networks are connected by gateways, specialized computers that route packets from one network to the next as the packets make their way to their destination. Gateways exchange routing information, using routing protocols, so they always know how to forward a packet to get it closer to where it is bound, even as network topology changes and connections come and go.

The Internet lives and breathes by protocols and standards. IP is the common mechanism, the lingua franca for exchange of information. Specific hardware technologies like Ethernet and wireless systems encapsulate IP packets, but the details of how any particular piece of hardware works or even that it's involved are not visible at the IP level. TCP uses IP to create reliable streams to specific ports at hosts. Higher-level protocols use TCP/IP to create services.

Protocols divide the system into layers, each of which provides services to the next higher level while calling on the services of the next lower level. This layering of protocols is fundamental to the operation of the Internet, a way to organize and control complexity while hiding irrelevant details of implementation. Each layer sticks to what it knows how to do—hardware networks move bytes from one computer on a network to another, IP moves individual packets across the Internet, TCP synthesizes a reliable stream from IP, and application protocols send data back and forth on the stream. Each layer uses the services provided by the layer immediately below it, and provides services to the layer directly above it; no layer tries to do everything. The programming interfaces that each layer presents are of course fine examples of the APIs we talked about in Chapter 5.

Internet privacy and security are hard, as we'll discuss further in the next couple of chapters. It often feels like an arms race between attackers and defenders, with the attackers more often on the winning side. Data passes through shared, unregulated, and diverse media and sites scattered over the whole world, and it can be logged, inspected and impeded at any point along the path for commercial and governmental purposes. It's hard to control access and to protect information along the way. Many networking technologies use broadcast, which is vulnerable to eavesdropping.

Attacks on Ethernets require finding a cable and making a physical connection, but attacks on wireless don't need physical access to snoop.

The Internet has only been in existence since about 1969. The core TCP/IP protocols date from around 1973, and have remained largely the same since then, in spite of explosive growth of network size, traffic, and innovative applications. The whole thing is a remarkable achievement.

10

The World Wide Web

The most visible face of the Internet is the World Wide Web, or, by now, just "the web." There's a tendency to conflate the Internet and the web, but they're different. The web connects computers that provide information—servers—with computers that ask for it—clients like you and me. The web uses the Internet to make the connections and carry the information and it provides an interface for other Internet-enabled services.

Like many great ideas, the web is fundamentally simple. Only four things matter, given the existence of a pervasive, efficient and open underlying network, which is a big proviso.

First is the *URL*, or Uniform Resource Locator, like `http://www.amazon.com`, which provides a way to specify a name for information and where to find it.

Second is *HTTP*, the HyperText Transfer Protocol, which was mentioned briefly in the last chapter as an example of a higher-level protocol. HTTP is simple: a client makes a request for a specific URL, and the server returns the information that was asked for.

Third is *HTML*, the HyperText Markup Language, a language for describing the format or presentation of information returned by a server. Again, it's simple, and you need know very little to make basic use of it.

Finally, there's a *browser*, a program like Firefox or Internet Explorer that runs on your computer, makes requests of servers (using URLs and HTTP), retrieves the HTML sent by a server, and displays it.

The web began life around 1989, when Tim Berners-Lee, an English physicist working at CERN, the European science research center near Geneva, created a system to make scientific literature and research results more easily available over the Internet. His design included URLs, HTTP and HTML, and there was a text-only

client program for viewing what was available.

This program was in use by 1990. I saw it in action in October 1992; it's embarrassing to admit, but at the time I didn't find it all that impressive, and I certainly didn't know that in less than six months, the creation of the first browser would change the world. So much for being able to see the future.

That first graphical browser, Mosaic, was created by a group of students at the University of Illinois. It took off rapidly after its initial release in February 1993, and the first commercial browser, Netscape Navigator, became available a year later. Netscape Navigator was an early success, and the surge of interest in the Internet caught Microsoft unaware. The company did wake up and quickly produced a competitor, Internet Explorer (IE), which became the most widely used browser by a very large margin. Microsoft's marketplace dominance raised antitrust concerns in several areas, and the company was sued by the US Department of Justice. IE was a part of that legal proceeding because it was alleged that Microsoft was using its dominant position to drive Netscape out of business, as we discussed in Chapter 5. IE's commanding position has fallen with the arrival of strong competitors like Firefox, Safari and Chrome.

The technical evolution of the web is managed, or at least guided, by the World Wide Web Consortium, or W3C (`w3.org`), a non-profit enterprise. Berners-Lee, the founder and current director of W3C, made no attempt to profit from his contributions, generously preferring to make it free for everyone, though many who jumped on the Internet and web bandwagon enabled by his work became very rich. He was knighted by Queen Elizabeth II in 2004.

10.1 How the Web Works

Let's take a more careful look at the technical components of the web, starting with URLs and HTTP.

Imagine viewing a simple web page with your favorite browser. Some of the text on that page is likely to be in blue and underlined, and if you click on that text, the current page will be replaced by the new page that the blue text links to. Linking pages like this is called *hypertext* ("more than text"); it's an old idea but the browser has made it part of everyone's experience.

Suppose the link says something like "W3C home page." When you move the mouse over the link, the status bar at the bottom of your browser window is likely to display the URL that the link points to, something like `http://w3.org`, perhaps with further information after the domain name.

When you click on the link, the browser opens a TCP/IP connection to port 80 at the domain `w3.org` and sends an HTTP request for the information given by the rest of the URL. If the link was `http://w3.org/index.html`, the request is for the file `index.html`.

When it receives this request, the server at `w3.org` decides what to do. If the request is for an existing file on the server, the server sends that file back, and the

client, your browser, displays it. The text that comes back from the server is almost always in HTML, a form that combines the real content with information about how to format or display it.

In real life, it can be this simple, but there is usually more. The protocol includes provision for the browser to send a few lines of additional information along with the request from the client, and the reply from the server will generally include extra lines that indicate how much data follows and what kind it is.

The URL itself encodes information. The first part, `http`, is one of several possibilities that tell which specific protocol to use. HTTP is most common by a wide margin, but if you watch carefully, you will see others as well, including `file` for information from the local machine (not from the web), `ftp` for file transfer with FTP, and `https` for a secure (encrypted) version of HTTP, which we'll talk about shortly.

After `://` comes the domain name, which names the server. After the domain name there can be a slash (`/`) and an arbitrary string of characters. That string is passed verbatim to the server, which can do whatever it likes with it. In the simplest case, there's nothing at all, not even the slash, and the server returns a default page like `index.html`. If there's a file name, its contents are returned as is. A question mark after the initial part of the file name usually means that the server is supposed to run a program whose name is the part before the question mark and pass the rest of the text to that program. That's one way that information from forms on a web page is processed; for example, a Bing search is

```
http://www.bing.com/search?q=funny+cat+pictures
```

which you can verify by typing it into the address bar of a browser.

URLs must be written in a restricted character set that excludes spaces and most non-alphanumeric characters, so these are encoded. A + sign encodes a space, as in this example, and other characters are encoded in hexadecimal, each one prefixed by a percent sign (%). For example, the URL fragment

```
8%22%20to%202%27
```

means

```
8" to 2'
```

since hex `22` is a double quote character, hex `27` is a single quote character, and hex `20` is a space.

In late 2009 ICANN approved the use of "internationalized domain names," that is, domain names in Unicode, though only for top-level domain names and so far only for a handful of countries such as Egypt, where `.eg` is now also available in Arabic:

مصر .

This is clearly appealing in places where the Latin alphabet is not the norm.

10.2 HTML

The text that comes back from the server is almost always in HTML, a combination of content and formatting information. HTML is so simple that it's easy to create web pages with your favorite text editor. (If you use a word processor like Word, be sure to save web pages in a text format like `.txt`, not the default format.) The formatting information is given by *tags* that describe the content, and mark the beginning and often the end of regions of the page. The HTML for a simple web page might look like this:

```
<html>
  <title> My Page </title>
  <body>
    <h2>A heading</h2>
    <p> A paragraph...
    <p> Another paragraph ...
    <img src="news.gif">
    <a href="http://news.google.com">link to Google News</a>
    <h3>A sub-heading</h3>
    <p> Yet another paragraph
  </body>
</html>
```

and will be displayed like this by a browser:

A heading

A paragraph...

Another paragraph ... Google news link to Google News

A sub-heading

Yet another paragraph

If the file named in the `<img>` image tag is not accessible, the browser is likely to display some "broken" image in its place.

Some tags are self-contained, like `<img>`; some have a beginning and end, like `<body>` and `</body>`; others, like `<p>`, do not need a closing tag in practice, though a strict definition does require `</p>`. Indentation and line breaks are not necessary but can make the text easier to read.

The point of this section is to show just enough of HTML to demystify how web pages work. Considerable skill is required to create the polished web pages that you see on commercial sites, but the basics are so easy that with a few minutes of study you can make your own more modest pages. A dozen tags will see you through the

creation of most text-only web pages, and another dozen are enough to do pretty much anything that a casual user might care about. It's easy to create pages by hand, word processors have a "create HTML" option, and there are programs specifically aimed at creating professional-looking web pages. You only need these if you're going to do serious web design, but it's good to understand how things work underneath as well.

10.3 Forms

The original design of HTML returned plain text for the browser to display. It wasn't long, however, before browsers added the capability to display images, both simple artwork like logos and smiley faces in GIF format, and pictures in JPEG format. Web pages offered forms to be filled in, buttons to push, and new windows that popped up or replaced the current one. Sound, animations, and movies followed soon after, generally once there was enough bandwidth to download them quickly and enough processing power to display them.

The HTTP protocol provides a simple mechanism, the unintuitively named CGI or Common Gateway Interface, for passing information from the client (your browser) to a server, for example a name and password, or a search query, or the choices made with radio buttons and dropdown menus. This mechanism is controlled by the HTML tag `<form>` ... `</form>`. Within the `<form>` tag, you can include common user interface elements like text entry areas, buttons, and so on. If there is a Submit button, pushing that causes the data within the form to be sent to the server, along with a request to run a specific program using that data.

Here's a simple example.

```
<form action="http://kernighan.com/cgi-bin/echoname">
     Type your name here:
     <input name="username" type="text">
     <input type="submit">
</form>
```

If you put this in a web page, it will look like this:

Type your name here: John Q Public Submit Query

When you push the Submit button, the browser will send the contents of the username field to the server, where the program `echoname` will be executed with the username as its input. The `echoname` program doesn't do much; it just sends back a page with the name on it:

Hello, John Q Public

Forms have limitations. They support only a few kinds of interface elements like buttons and dropdown menus. Form data can't be validated except by writing JavaScript code or sending it to the server for processing. The password input field

provides no security whatsoever, since the password is sent and stored in logs in the clear. Nevertheless, forms are a crucial part of the web.

10.4 Cookies

The HTTP protocol is "stateless," a bit of jargon which means that an HTTP server is not required to remember any information from one request by a client to the next—it can discard all records of each exchange after it returns the requested page.

But that raises a question: suppose that the server really does need to remember something, perhaps the fact that the user has already provided a name and password so there's no need to keep asking for them on every subsequent interaction. How might that be made to work? The problem is that the interval between the first visit and the second might be hours or weeks or might never happen, and that's a long time for a server to retain information on speculation.

In 1994, Netscape invented a solution that it called a *cookie*, a cutesy but well-established programmer word for a small piece of information passed between programs. When a server sends a web page to a browser, it can include additional chunks of text (up to about 4,000 bytes each) that the browser can store; each chunk is called a cookie. When the browser makes a subsequent visit to the same server, it sends the cookies back. So in effect, the server uses memory on the client to remember something about the client's previous visit. Often the server will assign a unique identification number to a client and include that in a cookie; any permanent information associated with that id number is maintained in a database on the server. This might be login status, shopping cart contents, user preferences, and the like. Each time the user revisits the site, the server can use the cookie to identify him or her as someone seen before and set up or restore information.

For instance, I normally disallow all cookies, so when I visit Amazon, the initial page greets me with "Hello." But if I allow cookies, which I must do to add to my shopping cart, subsequent visits will say "Hello, Brian." Some people find this ostensibly personal touch endearing, but in my case, it reminds me to remove cookies and forbid them again to reduce tracking.

Each cookie has a name, and a single server may store multiple cookies with each visit. Cookies have an expiration date after which they are supposed to be deleted by the browser. They are only sent back to the domain from which they originated, but there's no requirement that the browser actually accept them or return them. Note that a cookie is entirely passive: it's just a string of characters that is stored and subsequently sent back; nothing goes back to the server that didn't originate with the server. A cookie is not a program and it has no active content.

It's easy to view the cookies on your computer; the browser itself will report the ones that it is managing and you can usually find them in the file system as well. For example, a visit to the New York Times web site deposited 15 cookies before I even logged in; this one, called WT_FPC, contains some meaningful information:

```
id=140.247.62.34-237799664.30112241:
      lv=1288631855059:ss=1288631855059
```

The id value looks like an IP address and some identification; the other two values are almost surely times, likely the most recent visit and when the session started.

In principle, this all sounds quite benign and it certainly was intended as such, but no good deed goes unpunished, and cookies have been co-opted into less desirable uses as well. The most common is to track people as they browse, to create a record of the sites they have visited, and then (at the least offensive level) to provide targeted advertisements. We'll talk about how this works in the next chapter, along with other techniques that track you as you travel around the web.

10.5 Active Content in Web Pages

The original design of the web took no particular advantage of the fact that the client is programmable. The first browsers could make requests on behalf of the user, send information from forms, and with the aid of helper programs display content like pictures and sound that required special processing. But browsers soon began to support execution of code that had been downloaded from the web; this is sometimes called *active content.* How this is done depends on the nature of the code and the specific browser, but the idea is to execute code written in some language after downloading it to the client's computer. As might be expected, this has consequences, some good and some definitely not.

As we saw in Chapter 6, browsers are like specialized operating systems, which can be extended to handle richer and ever more complicated content "to enhance your browsing experience." Some of that content requires your browser to execute programs that someone else wrote, and whose properties you almost surely don't understand.

The good news is that it's possible to do a lot with a program running in the browser, and interactions often run faster if computation is performed locally. The downside is that there are real risks in running code from unknown sources on your computer. "I have always depended on the kindness of strangers" is not a prudent security policy. In a Microsoft article titled *10 Immutable Laws of Security*, the first law is "If a bad guy can persuade you to run his program on your computer, it's not your computer any more."

Early versions of Netscape Navigator included a way to run Java programs. At the time Java was a comparatively new language, and it had been designed to be easily installed in environments with modest computing capabilities (like appliances), so it was technically feasible to include a Java interpreter along with the browser. This offered the prospect of being able to do significant computation in the browser, perhaps replacing conventional programs like word processors and spreadsheets, and even the operating system itself. The idea was sufficiently worrisome to Microsoft that it took a series of actions to undermine the use of Java; these were the subject of legal actions by Sun Microsystems, the creator of Java.

For a variety of reasons, Java never took off as the way to extend browsers. Java itself is a rich language, but its integration with the browser is more limited, and today it is used infrequently in that role.

Netscape also created a new language specifically for use within its browsers: JavaScript, which appeared in 1995. In spite of its name, which was chosen for marketing reasons, JavaScript is unrelated to Java except that both have a surface similarity to the C programming language. Both use a virtual machine implementation, but with significant technical differences. Java source code is compiled where it is created and the resulting object code is sent to the browser for interpretation; you can't see what the original Java source code looks like. By contrast, JavaScript *source code* is sent to the browser, and the entire compilation process takes place in the browser. This makes it possible for the recipient to see the code that is being executed: it's impossible to protect the JavaScript source code, since the recipient can study and adapt it as well as run it.

Most web pages today include some JavaScript, to provide graphical effects, to validate information in forms, to pop up both useful and irritating windows, and so on. Its use for advertising popups has been somewhat mitigated by popup blockers, now included with browsers, but its use for sophisticated tracking and monitoring is increasing rapidly. JavaScript is so ubiquitous that it's hard to use the web without it, though browser add-ons like NoScript and Ghostery give some control over what JavaScript code will be run. On balance, JavaScript is more good than evil, though there are days when I might come down on the other side, especially considering how much it's used for tracking (which we will discuss in the next chapter). I routinely disable JavaScript entirely, but then have to restore it for many of the sites that I care about.

Other languages and contents are also handled by the browser, either with code in the browser itself, or by *plug-ins* like Apple Quicktime and Adobe Flash. Plug-ins are programs, usually written by third parties, that are dynamically loaded into the browser as needed. If you visit a page that has content in a format that your browser can't already handle, you may be offered the chance to "get the plug-in." What that means is that you will be downloading a new program to run on your computer in cooperation with the browser. What can a plug-in do? Essentially anything it wants to, so you are pretty much forced to trust the supplier, or do without the content.

A plug-in is compiled code that executes as part of the browser using an API presented by the browser and in effect becomes part of the browser when it runs. Flash animation is ubiquitous; other common plug-ins include Microsoft Silverlight, an alternative to Flash for video and other services. The short story on plug-ins is that if you trust their source, you can use them with no more than the normal perils of code that will have bugs and can monitor your behavior. A new version of HTML called HTML5 provides browser facilities that will reduce the need for plug-ins, especially for video and graphics, but plug-ins are likely to be important for a long time.

The least controlled or controllable augmentation of a browser is Microsoft's ActiveX technology, which provides a way to load code into Internet Explorer and

run it. This opens a Pandora's box of features and threats. ActiveX code itself is unrestricted; it can execute any Windows functionality, so in effect it has total control of your computer. Naturally, this is a mixed blessing: ActiveX components can provide all kinds of neat features, but with increased risk of something going wrong, or, if code comes from a bad guy, the chance of real harm. You have no way of knowing what an ActiveX component does, so you have to trust the supplier.

Microsoft provides a digital signing mechanism called Authenticode that attempts to ensure that at least the code comes from the supplier it claims to, and one of the settings in Internet Explorer's security model is to permit the download and execution only of signed ActiveX code. This is safer than blindly executing code of unknown provenance, though of course it's still not a guarantee of correct operation, nor even of a well-intentioned source.

10.6 Active Content Elsewhere

Active content can appear in other places than web pages, and the growth of web-based services (described in the next chapter) is exacerbating this potential problem. Consider email. Once mail has arrived, it will be displayed by a mail-reading program. Obviously a mail reader has to display text; the interesting question is how far it goes in interpreting other kinds of content that might be included, since that has a significant effect on privacy and security.

What about HTML in mail messages? Interpreting size and font tags is benign: there's no risk in displaying part of a message in big red letters, though it might irritate the recipient. Should the mail reader display images automatically? That makes it easy to view pictures, but opens up the prospect of still more cookies. We could legislate those out of existence, but what's to prevent a mail sender from including an image consisting of a 1 by 1 transparent pixel whose URL encodes something about the message or the recipient. (These invisible images are sometimes called *web beacons*.) When your HTML-enabled mail reader requests the image, the site where the image is stored knows that you are reading that specific mail message at a particular time, which provides an easy way to track when mail has been read, and potentially reveals information that you might prefer to keep private.

If the mail reader handles HTML, does it also interpret JavaScript? I was a bit surprised when exploring a new web-based mail reader recently to learn that not only could it interpret JavaScript in mail messages but the default behavior was to do so without even telling me.

What happens if a mail message includes a Word document, or Excel or Powerpoint or PDF or a Flash movie? Should the mail reader run those programs automatically? Should it make it easy for you to do so by clicking somewhere in the message? Come to think of it, should it let you click directly on links in messages? That's a favorite way to entice victims. PDF documents can include JavaScript (which also surprised me when I first saw it); should that code be executed automatically by the PDF viewer that is invoked automatically by the mail reader?

It's convenient to be able to attach documents, spreadsheets and slide presentations to email messages, and it's standard operating procedure in business environments, but such documents can carry viruses, as we shall see shortly, and blindly clicking on them is one way to propagate viruses.

It's even worse if mail messages include executable files, like Windows `.exe` files or equivalents. Clicking on those launches the program, which with high probability is something that will damage you or your system. The bad guys use a variety of tricks to get you to run such programs. I once got mail that claimed to include a picture of Anna Kournikova, the Russian tennis player, and encouraged me to click on it. The file name was `kournikova.jpg.vbs`, but the `.vbs` extension was hidden (a misguided feature of Windows), concealing the fact that it was not a picture but a Visual Basic program. Fortunately, since I was using Pine, an antiquated text-only mail program on a Unix system, clicking was not an option, so I saved the "picture" in a file for later inspection.

10.7 Viruses and Worms

The Anna Kournikova "picture" was in fact a virus. Let's talk a bit about viruses and worms. Both of these terms refer to (often malicious) code that propagates from one system to another. The technical distinction, not very important, is that a virus requires help with propagation—it can only get to another system if you do something to enable it—while a worm can propagate without your assistance.

Although the potential for such programs had been known for a long time, the first example that made the evening news was the "Internet worm" launched by Robert T. Morris in November 1988, well before what we would call the modern Internet era. Morris's worm used two different mechanisms to copy itself from one system to another, relying on bugs in widely used programs, together with a dictionary attack (trying common words as potential passwords) so it could log itself in. Morris had no malicious intent—he was a grad student at Cornell and had been planning an experiment to measure the size of the Internet—but a programming error made the worm propagate much faster than he expected, with the result that many machines were infected multiple times, couldn't cope with the traffic, and had to be disconnected from the Internet. Morris was convicted under the then new Computer Fraud and Abuse Act, and was required to pay a fine and perform public service.

For some years, it was common for viruses to propagate via infected floppy disks, which before the widespread use of the Internet were a standard medium for exchanging programs and data between PCs. An infected floppy typically included a program that would run automatically when the floppy was loaded; it would copy itself to the local machine and thus assure that the virus would be passed on whenever a subsequent floppy disk was written.

Virus propagation became much easier with the arrival of Visual Basic in Microsoft Office programs, especially Word. Word included (and still does) a VB interpreter, and Word documents (`.doc` files) can contain VB programs, as can Excel and

Powerpoint files. It is dead simple to write a VB program that will take control when a document is opened, and because VB provides access to the entire power of the Windows operating system, the program can do anything it wants. The usual sequence is to install itself locally if it's not already present, and then arrange to propagate itself to other systems. In one common mode of propagation, the virus mails a copy of itself, along with some innocuous or appealing message, to every entry in the current victim's email address book. (The Anna Kournikova virus used this method.) If the recipient opens the document, the virus installs itself on the new system and the process repeats.

In the mid to late 1990s there were many such VB viruses. Because the default setting for Word was to blindly run VB programs without asking for permission, infections spread rapidly, and large corporations found themselves having to turn off all their computers and clean them up individually to stamp out the virus. VB viruses are still around but merely changing the default behavior of Word and similar programs has greatly reduced their impact; in addition, many mail systems now strip out VB programs and other executable content from incoming mail before the mail reaches a reader.

VB viruses are so easy to create that people who wrote them were called "script kiddies." Making a virus or worm that does its job without getting caught is harder. In late 2010, a sophisticated worm called Stuxnet was discovered in a number of process control computers. Its main target appears to have been uranium enrichment equipment in Iran. Its approach was subtle: it caused speed fluctuations in centrifuge motors that would lead to damage or even destruction that might seem like normal wear and tear; at the same time, it reported normal operation to monitoring systems so nobody would notice problems. No one has yet stepped forward to take credit for this program.

A *Trojan horse* (often shortened to Trojan in this context) is a program that masquerades as something beneficial or harmless but in fact does something harmful. The victim is induced to download or install the Trojan because it appears to do something helpful. One typical example offers to perform a security analysis on a system, but in fact installs malware instead.

I mentioned floppy disks as a vector for propagating viruses. The equivalent today is the infected USB flash drive. One might think that a flash drive is a passive device, since it's just memory. However, some systems, notably Windows, provide an "autorun" service that automatically runs a program *from the drive* when a CD, DVD or flash drive is plugged in. If this feature is enabled, malevolent software can be installed and damage done without any warning or chance to intervene. It's fairly common for systems to be infected in this fashion, even though most companies have strict policies against plugging USB drives into company computers. On occasion, brand new drives have been shipped with viruses already on them.

10.8 Web Security

The web raises many difficult security issues. Broadly speaking, one might divide the threats into three categories: attacks on clients (that's you), attacks on servers (online stores or your bank, for example), and attacks on information in transit (like snooping on your wireless). Let's talk about each in turn, to identify some of what might go wrong and what can be done to alleviate the situation.

10.8.1 Attacks on clients

Attacks on you include nuisances like spam and tracking, and more serious concerns, especially the release of private information like your credit card and bank account numbers, and the potential for someone else to masquerade as you to spend your money.

In the next chapter, we will discuss in detail how cookies and other tracking mechanisms can monitor your web activities, ostensibly to provide you with more appealing and thus less irritating advertising. This can be kept to manageable amounts by forbidding third-party cookies (that is, cookies that come from a different web site than the one you went to), which is how much of the tracking is done, and using other browser add-ons that disable trackers, turn off JavaScript, block Flash, and so on. It's a nuisance to maintain your defenses, since some sites are unusable when you have your shields up; you have to lower them temporarily, then remember to reset.

Spam—unsolicited mail that offers get-rich schemes, body-part enhancements, performance improvers, and a host of other unwanted goods and services—has become so voluminous as to endanger the very use of email. I generally get fifty to a hundred spam messages per day, many more than the number of real messages. It is sometimes claimed, believably, that 90 percent of all email traffic is spam; an article I recently read says that there are about 250 billion spam messages a day. Spam is so common because sending it is almost free (the same article said the price was about $80 per million messages). If even the tiniest fraction of the millions of recipients reply, that's enough to keep things profitable.

Spam filters try to separate the wheat from the chaff by analyzing the text for known patterns ("How would you like eBay to send you money every day?", asks a printable one from today's batch), or unlikely names, or bizarre spellings (\/|/-\GR/-\), or addresses that spammers favor. No single criterion could be sufficient so a combination of filters is used. Spam filtering is another example of an arms race, since as the defenders learn how to cope with one kind, the offenders find a new way around.

Spam is hard to stop at the source, because the source is well hidden. Much spam is sent by compromised personal computers, often running Windows. Security holes and lax management by users leave the computers vulnerable to the installation of *malware*, that is, malicious software that intends to do damage or otherwise interfere with the system. One kind of malware will broadcast spam mail upon command from an upstream control, which might in turn be controlled from another computer;

each step makes it that much harder to find the originator.

Phishing attacks try to convince the recipient to voluntarily hand over information that can be used for theft. You've almost surely gotten a "Nigerian" scam letter. It's hard to believe that anyone ever responded to something so implausible, though people still do. Phishing attacks are more subtle, since they appear to come from legitimate financial institutions or even from friends. Plausible mail arrives, ostensibly from a company that you know, asking you to verify your credentials. If you do, the bad guy now has information that he can use to steal your money or your identity. This one is typical of the state of the art a few years ago; grammar and spelling mistakes give it away and the fact that my name is not Bill and I have no Citibank account is also a clue.

> From: Citibank <cash@citibank.com>
> To: bill@research.bell-labs.com
> Subject: Please update your account details with Citibank
> Dear Citibank Customer,
> We recently noticed one or more attempts to log in to your Citibank account from a foreign IP address and we have reasons to believe that there was attempts to compromise it with brute forcing your PIN number. No successful login was detected and you have full protection by now. If you recently accessed your account while traveling, the unusual login attempts may have been initiated by you.
> The login attempt was made from IP address: 173.27.187.24, ISP Host: cache-82.proxyserver.cis.com
> By now, we used many techniques to verify the accuracy of the information our users provide us when they register on the Site. However, because user verification on the Internet is difficult, Citibank cannot and does not confirm each user's purported identity. Thus, we have established an offline verification system to help you evaluate with whom you are dealing with. The system is called CitiSafe and it's the most secure Citibank wallet so far. If you are the rightful holder of the account, click the link bellow, fill the form and then submit as we will verify your identity and register you to CitiSafe free of charge. This way you are fully protected from fraudulent activity on all the accounts that you have with us.
> Click to protect yourself from fraudulent activity!
> To make Citibank.com the most secure site, every user will be registered to CitiSafe.
> NOTE! If you choose to ignore our request, you leave us no choice but to temporally suspend your account.
> * Please do not respond to this e-mail, as your reply will not be received.
> Regards, Citibank Customer Support

It's easy to create a message that looks official—the format and images can be copied from the real site. It's easy to use an invalid return address because there's no check that the sender is legitimate. The IP address behind the link led to a site in China, but I had to examine the HTML source and run `traceroute` to deduce that. Like spam, phishing costs almost nothing, so even a tiny success rate can be profitable.

Newer attacks are harder to spot. Late in 2010, I received this mail message from a friend, sent to a very small private mailing list:

> Date: Wed, 1 Dec 2010 04:55:37 -0800 (PST)
> From: Charles [...] <[...]@gmail.com>
> Apologies for having to reach out to you like this, but I made a quick trip to the UK and had my bag stolen from me with my passport and credit cards in it. The embassy has cooperated by issuing a temporary passport, I just have to pay for a ticket and settle Hotel bills.
> To be honest,i don't have money with me,I've made contact with my bank but the best they could do was to send me a new card in the mail which will take 2-4 working days to arrive here.i was thinking of asking you to lend me some quick funds that i can give back as soon as i get in,i really need to be on a last minute flight that leaves in a few hours.
> I can furnish you with info on how you will get me the money. You can reach me via ([...]@aol.com) as am logged on via a smart device or hotel's desk phone, the number is, +447045749898.
> Thanks
> Charlie

It was plausible. Charlie is a frequent traveler and could have made a trip; I hadn't seen him for a couple of weeks. I called the number on speculation, but there was no ring, which was a bit odd. I replied to the mail, and got back a personal response:

> Hi Brian
> I am in a bit of trouble,please lend me $1,800. Western union will be OK, let me know if you need my details for the transfer. I will pay back once I return to the States.
> Charlie

I asked for instructions but I also asked a question that only the real Charlie could possibly have answered. Back came detailed steps for using Western Union, but there was no answer to my question. I was way beyond suspicious at this point, and a quick search for the phone number revealed that this was a well-known scam that had been going on for months; the perpetrators hadn't bothered to change the script or the number. It later became clear that Charlie's Gmail account had been compromised, which provided the attackers with focused lists of potential victims and relatively private information; it's for this reason that such precisely targeted attacks are sometimes called *spear phishing*. Spear phishing is a kind of *social engineering*: induce the victim to do something foolish by pretending to have a personal relationship like a mutual friend, or claiming to work for the same company.

Spyware refers to programs that run on your computer, sending information about you to somewhere else. Some of this is clearly malevolent, and sometimes it's merely commercial snooping. For instance, several of my laptops came with a program called "pinger" that starts automatically when the computer is rebooted and makes a regular call to the manufacturer thereafter. Or at least it used to until I noticed it one day and disabled it. I'm sure that the manufacturer would tell me that pinger is merely reporting on the continued good health of the computer and checking for software updates, but I was never told about it or offered the option of turning it on (or even of turning it off).

In a particularly flagrant example, in 2005 Sony sold audio CDs that used autorun to install spyware on Windows systems; when the disk was inserted it would automatically and silently install monitoring programs. The spyware tried to hide its existence, which suggests that someone knew the behavior was sketchy at best. It had implementation flaws that opened up the computer so anyone could install software. Presumably the intent was to prevent illegal copying, but the program reported on any music being played on the computer, whether from Sony or not. When all of this was discovered, Sony offered an uninstaller program that made the security problems worse.

On personal machines, it's common for an attacker to install a *zombie*, that is, a program that waits until it is awakened via the Internet and told to perform some hostile act like sending spam. Such programs are often called *bots* and networks of them with a common control are called *botnets*. At any point, there are thousands of known botnets and probably hundreds of thousands to millions of bots that can be called into service.

One can compromise a client computer and steal information at the source, either by looking for it in the file system or by using a surreptitiously installed *key logger* to capture passwords and other data as they are entered. A key logger is a program that monitors all keystrokes on the client and thus is able to capture passwords as they are typed; encryption cannot help here. Such a program could also turn on the microphone and camera in the computer.

The risks are exacerbated if your browser or other software has bugs that make it possible for malcreants to install their software on your machine. Browsers are large and complicated programs, and they have had numerous bugs that permit attacks on users. Keeping your browser up to date is a partial defense, as is configuring it so it does not release unnecessary information or permit random downloading. Being cautious about what you download is another; don't just click when a web page or program asks you to do so. We'll discuss more defenses in a few pages.

10.8.2 Attacks on servers

Attacks on servers aren't your problem directly, in that there isn't much you can do about them, but that doesn't mean that you can't be victimized.

Servers have to be carefully programmed and configured so that client requests, no matter how cleverly crafted, cannot cause the server to release unauthorized information or permit unauthorized access. Servers run big complicated programs, so bugs and configuration errors are common; both can be exploited. One frequent attack is called *SQL injection*. Servers are usually supported by databases that are accessed through a standard interface called SQL ("structured query language"). If the access is not carefully circumscribed, a clever attacker can submit queries for information about the database structure, extract unauthorized information, and even run the attacker's code on the server, code that might be able to gain control of the whole system. Such attacks are well understood, as are defenses against them, but they still occur surprisingly often.

Once a system has been compromised, there are few limits on the harm that can be done, especially if the attacker has managed to obtain "root" access, that is, access at the highest level of administrative privilege. This is true whether the target is a server or an individual home computer. At that point, an attacker can vandalize or deface a web site (posting embarrassing material like hate speech), download destructive programs, or store and distribute illegal content like child pornography and pirated software. Data can be stolen in bulk from servers, or in modest quantities from individual machines. For instance, in April 2011, hackers were able to steal names, addresses, email addresses and other information for 70 million users of Sony's PlayStation network. In another breach at about the same time, names and email addresses of millions of consumers were stolen from Epsilon, an email marketer that serves many large corporations. (I got two notifications about Epsilon from companies that I do business with; one said that only email addresses were taken, while the other said that it was both names and email. This discrepancy did not increase my confidence that either one was telling me the whole story.) Such breaches are frequent events, though not always on such a grand scale.

Servers can also be subject to *denial of service* (DoS) attacks, in which an adversary directs a large amount of traffic at a site to overwhelm it with sheer volume. This is often orchestrated with botnets; the compromised machines are told to send requests to a particular site at a particular time, leading to a coordinated flood of traffic. An attack that comes from many sources simultaneously is often referred to as a *distributed denial of service* (DDoS) attack.

One fundamental problem is asymmetry: defenders need to defend against all possible attacks, but attackers only need to find one weakness. The advantage lies with the attackers.

10.8.3 Attacks on information in transit

Attacks on information in transit have perhaps been the least of the concerns, though they are certainly still serious and common enough; with the spread of wireless systems, this may change, and not for the better. A bad guy could eavesdrop on your conversation with your bank to steal information like your account number and password. If traffic between you and the bank is encrypted, however, it will be difficult for an eavesdropper to understand what the traffic says. HTTPS is a version of HTTP that encrypts traffic in both directions, which generally makes it impossible for an eavesdropper to read information or to masquerade as one of the parties. The use of HTTPS is not universal, however, and programs like Firesheep can snoop on connections in coffee shops, airports and other public spaces that offer open wireless access, and allow an attacker to pretend to be you, quite undetectably. One large theft of credit card data involved listening to unencrypted communications between terminals in stores; the thieves parked outside stores and captured credit card information as it went by.

It's also possible to mount a "man in the middle" attack, in which the attacker intercepts messages, modifies them, and sends them on as if they had come directly

from their original source. (The *Count of Monte Cristo* story mentioned in Chapter 8 is an instance.)

A *virtual private network* (VPN) establishes an encrypted path between two computers, and thus generally secures information flow in both directions. Corporations often use VPNs to enable employees to work from home or in countries where the security of communication networks can't be trusted. Access requires both a password and a physical device in the possession of the user; this kind of "two-factor" authentication is safer than a password alone since it requires the user to know something (password) and have something (a device). The device can be a cell phone app that generates a number which has to match a number generated at the server end by the same algorithm. It may instead be a special-purpose device that displays a number that has to be provided along with the password. The two-factor technique is not restricted to VPNs; it's used for banking and access to online accounts as well.

Ironically, RSA, a company that makes a widely-used two-factor authentication device called SecurID, was hacked in March 2011. Security information was stolen, rendering some SecurID devices vulnerable; a VPN at defense contractor Lockheed Martin was subsequently attacked, apparently using this information.

10.8.4 Defending yourself

What can you do in your own defense? When someone asks me for advice, this is what I tell them. I divide defenses into three categories: ones that are very important, ones that are prudent and cautious, and ones that depend on how paranoid you are. (Just so you know, I'm well out on the paranoid end; most people won't want to go so far.)

Important:

Choose strong passwords so no one can guess them and trying a bunch of possibilities isn't likely to reveal them quickly. A sequence of random upper and lower case letters, numbers and special characters is more likely to be strong than single dictionary words, your birthday, names of family or pets, and especially "password" itself, which is chosen amazingly often.

Don't use the same password for critical sites like your bank and email as you do for throwaways like online newspapers. Don't use the same passwords at work as you do for personal accounts. Change your passwords from time to time, but not in an obviously formulaic way like adding 1 to a digit at the end.

Don't do anything important at places that offer open wireless. Make sure that connections use HTTPS, but don't forget that HTTPS only encrypts the contents.

Don't view attachments from strangers, don't view unexpected attachments from friends, don't automatically accept, click or install when requested. Don't download programs of dubious provenance; be wary about downloading and installing any software unless it comes from a trusted source.

If you use Windows, use anti-virus software and keep it up to date; don't click on the sites that offer to run a security check on your computer. Turn off macros in Microsoft Office programs; disable ActiveX as much as you can. If you use a Mac, anti-virus software does not seem to be necessary at the moment, but you still need to be careful. Keep other software, like browsers and the operating system itself, up to date as well, since security fixes are frequent.

Cautious:

Turn off pop-ups and third-party cookies. (It's a nuisance that cookies are stored on a per-browser basis, so you have to set up defenses for each browser you use.) Use add-ons like Adblock and Flashblock to reject advertising images.

Use a spam filter on your mail.

Turn off JavaScript in Adobe Reader.

Turn off services that you don't use. For example, my Mac offers to let me share printers, files, and devices, and to allow other computers to log in and manage my computer remotely. Windows has an analogous set. I turn them all off except printer sharing.

A *firewall* is a program that monitors incoming and outgoing network connections and blocks those that violate access rules. Turn on the firewall on your computer.

Use two-factor authentication on your important accounts if it's available.

Paranoid:

Disable HTML and JavaScript in your mail reader.

Use NoScript to limit JavaScript, and Ghostery to disable trackers.

Turn off all cookies except for sites that you explicitly want to permit.

Use less-frequently targeted systems, like Linux or Mac OS X instead of Windows; Chrome, Firefox, Safari or Opera instead of Internet Explorer; and so on.

As cell phones become more of a target, similar precautions will become necessary for them as well.

10.9 Cryptography

Cryptography, the art of "secret writing," has been used for thousands of years to exchange private information with other parties. Julius Caesar used a simple scheme (coincidentally called the Caesar cipher) of shifting the letters in his secret messages over by three positions, so A became D, B became E, and so on. Thus the message "HI JULIUS" would be encoded as "KL MXOLXV". This algorithm lives on in a program called `rot13` that shifts by 13 places; it's used on newsgroups to hide offensive material and spoilers from accidental viewing, not for any cryptographic

purpose.

Cryptography has a long history, often colorful and sometimes dangerous to those who thought encryption kept their secrets safe. Mary, Queen of Scots lost her head in 1587 because of bad crypto. She was exchanging messages with conspirators who wanted to depose Elizabeth I and put Mary on the throne. The cryptographic system was cracked, and a man in the middle attack exposed the plot and the names of the collaborators; their fates make beheading seem like a humane alternative. Admiral Isoroku Yamamoto, commander-in-chief of the Japanese Combined Fleet, was killed in 1943 because the Japanese encryption system was not secure. American intelligence learned Yamamoto's flight plans, and American pilots were able to shoot down his plane. And it is claimed, though perhaps overstated, that World War II was significantly shortened because the British were able to decode German military communications encrypted with the Enigma machine, using computing techniques and expertise from Alan Turing.

The basic idea of cryptography is that Alice and Bob want to exchange messages, keeping the contents private, though not the fact that they are communicating. To do this, they need some kind of shared secret that can be used to scramble and unscramble messages, making them unintelligible to others but intelligible to Alice and Bob. This secret is called the *key*. In the Caesar cipher, for example, the key would be the distance that the alphabet is shifted, that is, three for replacing A by D, etc. For a complicated mechanical encryption device like the Enigma, the key is a combination of settings of several code wheels and the wiring of a set of plugs. For modern computer-based encryption systems, the key is a large secret number that is one input to an intricate algorithm that transforms the bits of a message so that it is infeasible to unscramble without knowing the secret number.

Cryptographic algorithms can be attacked in a variety of ways. Frequency analysis—counting the occurrences of each symbol—works well against the Caesar cipher and the simple substitution ciphers of newspaper puzzles. To defend against frequency analysis, algorithms must arrange that all symbols are equally likely and there are no patterns to analyze in the encrypted form. An attacker might be able to exploit known plaintext, that is, a message that is known to have been encrypted with the target key, and he might also be able to exploit a chosen plaintext, running a message of his own choosing through the algorithm with the target key. A good algorithm has to be immune to all such attacks.

It must be assumed that the cryptographic system is known and perfectly understood by adversaries, so that all of the security resides in the key. The alternative, assuming that adversaries don't know what scheme is being used or how it works, is called *security by obscurity,* and it never works for very long if at all. Mary, Queen of Scots, Admiral Yamamoto, and the Wehrmacht all failed to appreciate this crucial fact, and suffered for it. Indeed, if someone tells you that their cryptosystem is perfectly secure but they won't tell you how it works, you can be certain that it's not secure.

Modern cryptographic failures are usually not so physically dangerous, but they are embarrassing to those who trusted the failed systems. One of the nicest examples is the Content Scrambling System that encrypts movie DVDs. It was cracked in 1999 by a 15-year-old Norwegian student, Jon Lech Johansen ("DVD Jon"), who published a DeCSS program that is now widely disseminated. Johansen has gone on to reverse-engineer a number of other cryptographic systems.

Two fundamentally different kinds of cryptographic systems are in use today. The older is usually called *secret-key cryptography* or symmetric-key cryptography. "Symmetric" is more descriptive because the same key is used for encryption and decryption, but "secret-key" contrasts better with the name of the newer kind, *public-key cryptography*.

10.9.1 Secret-key cryptography

In secret-key cryptography, a message is encrypted and decrypted with the same secret key, which must be shared by all the parties who want to exchange messages. Assuming that the algorithm is entirely understood and has no flaws or weaknesses, the only way to decrypt a message is a *brute force attack*: try all possible secret keys, looking for the one that was used for encryption. This can take a long time; if the key has N bits, the effort is roughly proportional to 2^N. Brute force does not mean dumb, however. An attacker will try short keys before long ones and likely keys before unlikely ones, for example a *dictionary attack* based on common words and number patterns like "password" and "12345". If people are lazy or careless about choosing keys, such attacks can be very successful.

From about 1976 to the early 2000s, the most widely used secret-key cryptographic algorithm was the Data Encryption Standard or DES, which was developed by IBM and the National Security Agency (NSA). Although there was some suspicion that NSA had arranged a secret trap-door mechanism so they could easily decode DES-encoded messages, this was never proven. In any case, DES had a 56-bit key, and as computers became faster, this proved to be too short; by 1999, a DES key could be cracked by brute force in a day of computing. This led to the creation of new algorithms with longer keys.

The most widely used of these is the Advanced Encryption Standard, or AES, which was developed as part of a worldwide open competition sponsored by the US National Institute of Standards and Technology (NIST). Several dozen algorithms were submitted from all over the world and were subjected to intense public testing and critique. Rijndael, created by the Belgian cryptographers Joan Daemen and Vincent Rijmen, was the winner, and became an official US government standard in 2002. The algorithm is in the public domain, and anyone can use it without licenses or fees. AES supports three key lengths—128, 192, and 256 bits—so there are a lot of potential keys and a brute force attack will not likely be feasible for many years unless some weakness is discovered.

The big problem with AES and other secret key systems is *key distribution*: each communicating party must know the key, so there must be a secure way to get the

key to each of them. That could be as easy as having everyone over to your house for dinner, but if some members are spies or dissidents in hostile environments, there might not be any safe and secure channel for sending a secret key. Another problem is key proliferation: to have independent secret conversations with unrelated parties, one needs a separate key for each group, making the distribution issue even harder. Considerations like these led to the development of public-key cryptography, our next topic.

10.9.2 Public-key cryptography

Public-key cryptography is a completely different idea, invented in the mid 1970s by Whitfield Diffie and Martin Hellman at Stanford. The idea was independently discovered at about the same time by James Ellis and Clifford Cocks, cryptographers in a British intelligence agency, but since their work was secret, they were unable to publish and have missed out on most of the credit.

In a public-key cryptosystem, each person has a *key pair*, consisting of a public key and a private key. The keys of the pair are integers that are mathematically related and have the property that a message encrypted with one key of the pair can only be decrypted with the other key, and vice versa. With sufficiently long keys, brute force attacks are infeasible, and so is deducing one key from the other.

In use, the public key is truly public: it is available to everyone, often posted on a web page. The private key is strictly private, a secret known only to the owner of this pair of keys.

Suppose that Alice wants to send a message to Bob, encrypted so that only Bob can read it. She goes to Bob's web page and gets his public key, which she uses to encrypt her message to him. When she sends the encrypted message, an eavesdropper Eve might be able to see that Alice is sending a message to Bob but since it's encrypted, Eve can't tell what the message says.

Bob decrypts Alice's message with his private key, which only he knows, and which is the only way to decrypt a message that was encrypted with his public key. If he wants to send an encrypted reply to Alice, he encrypts it with *Alice's* public key. Again, Eve can see the reply but only in its encrypted form, which she can't understand. Alice decrypts Bob's reply with her private key, which only she knows.

This scheme solves the key distribution problem, since there are no shared secrets to distribute. Alice and Bob have their respective public keys on their web pages, and anyone can carry on a private conversation with either one of them without prearrangement or the exchange of any secrets. Indeed, there is no need for the parties to have ever met.

Public-key cryptography is a critical component of secure communication on the Internet. Suppose that I want to buy a book online. I have to tell Amazon my credit card number, but I don't want to send that in the clear so we need an encrypted communication channel. We can't use AES directly because we don't have a shared key. To arrange a shared key, my browser generates a random temporary password. It then uses Amazon's public key to encrypt the temporary password and sends the

encrypted password to Amazon. Amazon uses its private key to decrypt the temporary password. Amazon and my browser now use that password to encrypt information like my credit card number with AES.

The main drawback of public-key crypto is that its algorithms tend to be slow, perhaps several orders of magnitude slower than a secret-key algorithm like AES. Thus rather than encrypting everything with public-key, there's a two-step sequence: use public-key to agree on a secret key and then use AES for transferring data in quantity.

The communication is secure at each stage, initially with public-key to set up the temporary password, and then with AES for exchange of bulk data. If you visit a web store, an online mail service, and many other sites, you will wind up using this technique. You can see it in action, because your browser will show that you are connecting with the HTTPS protocol (HTTP with security) and will display some icon reminiscent of a closed lock to indicate that the link is encrypted:

Public-key cryptography has other interesting properties. For instance, it can be used as a *digital signature* scheme. Suppose Alice wants to sign a message so the recipient can be sure that it came from her and not an imposter. If she encrypts the message with her private key and sends the result, then anyone can decrypt it with her public key. Assuming that Alice is the only one who knows her private key, the message must have been encrypted by Alice. Clearly this only works if Alice's key has not been compromised. You can also see how Alice could sign a private message to Bob, so that no one else can read it and Bob can be sure it came from Alice. Alice signs the message by encrypting it with her private key, then encrypts the result with Bob's public key. Eve can see that Alice sent something to Bob, but only Bob can decrypt it with his private key; he then decrypts it with Alice's public key to verify that it came from her.

Public-key cryptography is not a panacea, of course. If Alice's private key is revealed, all past messages to her can be read and all of her past signatures are suspect. It's hard to revoke a key, that is, to say that a particular key is no longer valid, though most key-creation schemes include information about when the key was created and when it is meant to expire.

The most frequently used public-key algorithm is called *RSA*, after Ron Rivest, Adi Shamir and Len Adleman, the MIT computer scientists who invented it in 1978. The RSA algorithm is based on the difficulty of factoring very large composite numbers. RSA works by generating a large integer, say 300 digits long, with two large prime factors (each about half as long as the product) and uses these as the basis of the public and private keys. Someone who knows the factors (the holder of the private key) can quickly decrypt an encrypted message, while everyone else has to in effect factor the large integer, and this is computationally infeasible, or at least so it is believed. Rivest, Shamir and Adleman won the 2002 Turing Award for the invention of the RSA algorithm.

The length of the key is important. So far as we know, the computational effort required to factor a big integer grows exponentially with its length. I used to demonstrate RSA by generating a 512-bit key, which was the default length for the standard generator program. Around 2008, however, the generator refused to let me use 512, on the grounds that it wasn't long enough, so now I use 1,024 bits. Presumably 1,024 will also be deemed too short in a few years. RSA Laboratories, the company that held the rights to the RSA patent, ran a factoring challenge from 1991 until 2007. It published a list of composite numbers of increasing lengths and offered cash prizes for the first person to factor each one. The smallest numbers were about 100 digits long and were factored fairly quickly. When the challenge ended in 2007, the largest number factored had 193 digits (640 bits), and a prize of $20,000. (The list is still online if you want to try your hand.)

Although modern cryptography has amazing properties, in practice it still requires some degree of trust and the existence of trusted third parties. For example when I order a book, how can I be sure that I am talking to Amazon and not a sophisticated imposter? When I visit, Amazon verifies its identity by sending me a *certificate*, a digitally signed collection of information from an independent certificate authority that can be used to verify Amazon's identity. My browser checks this with a public key supplied by the certificate authority to verify that it belongs to Amazon and not someone else. In theory I can be sure that if the authority says it's Amazon's certificate, it really is. But I have to trust the certificate authority; if that's a fraud, then I can't trust anyone who uses it. A typical browser has an astonishing number of certificates—over 80 in my version of Firefox—and the majority are from organizations that I've never heard of and located in faraway places, like Chunghwa Telecom in Taiwan and Disig a.s. in Slovakia. In August 2011 it was discovered that a hacker had compromised DigiNotar, a certificate authority in Holland, and was able to create fraudulent certificates for a number of sites, including Google.

Because public-key algorithms are slow, documents are often signed indirectly, using a much smaller document derived from the original in a way that can't be forged nor can another document be found that has the same value. This short document is called a *message digest* or *cryptographic hash*. It is created by an algorithm that scrambles the bits of any input into a fixed-length sequence of bits—the digest or hash—that have the property that it's computationally infeasible to find some other input that produces the same digest; furthermore, the slightest change to the input changes approximately half the bits in the digest. Thus tampering with a document can be efficiently detected by comparing its digest or hash with the original digest. As an illustration, here are cryptographic hashes (in hex) of the letters x and X, which in ASCII differ by a single bit:

```
x    9DD4E461268C8034F5C8564E155C67A6
X    02129BB861061D1A052C592E2DC6B383
```

It is computationally infeasible to find another input that has the same hash value as either of these, and there's no way to go back from the hash to the original input.

There are two message digest algorithms in widespread use. MD5 was developed by Ron Rivest; it produces a 128-bit result. The other is SHA-1, which comes from NIST; it has 160 bits. Both MD5 and SHA-1 have been shown to have weaknesses that make them unsuitable for situations where a strong security guarantee is required. NIST is running a competition for a new algorithm, SHA-3, to address the problems with SHA-1.

In any security system, the weakest link is the people involved, who will subvert systems that are too complicated or hard to use. Think about what you do when you are forced to change your password, especially when the new one has to be invented right now and must satisfy weird restrictions like containing mixed case, digits, and some special characters but not others. If you're like me, you sometimes resort to formulas and writing it down, both of which potentially compromise security. (Ask yourself: if an enemy saw two of your passwords, could he or she guess others?) Even if everyone tries hard to be secure, a determined adversary can always use one of the 4 B's (bribery, blackmail, burglary, brutality) to gain access, and governments are increasingly using the threat of jail for people who refuse to divulge their passwords when asked.

10.10 Summary

The web has come from nothing in 1990 to a pervasive force today. The web has changed the face of business, especially at the consumer level, with search, online shopping, ratings systems, price comparison and product evaluation sites. It has changed a lot of our behavior as well.

Along with myriad opportunities and benefits, it has brought problems and risks because it enables action at a distance—we are visible and vulnerable to people that we have never met.

The web raises unresolved jurisdictional issues. For instance, in the US, many states charge sales tax on purchases within their borders, but online stores like Amazon do not collect sales tax from purchasers in most states, on the theory that if they have no physical presence in a state, they are not required to act as an agent of that state's tax authorities. Purchasers are supposed to report out-of-state purchases and pay tax on them, but no one does.

Libel is another area where jurisdiction is uncertain. In some countries it is possible to win a libel suit even though the person allegedly doing the libel has never been in the country where the suit is filed, merely because a web site (hosted elsewhere) is visible in that country.

Some activities are legal in one country, not in another; pornography and online gambling are two common examples. How does a government enforce its rules on its citizens when they are doing things on the Internet that are illegal within its borders? Some countries provide only a limited number of Internet pathways into and out of the country, and can use those to block traffic that they do not approve of; the "Great Firewall of China" is one example, but certainly not the only one. Requiring people

to identify themselves on the Internet is another approach. It sounds like a good way to prevent anonymous abuse and harassment but it also has a chilling effect on activism and dissent.

There will always be tension between the legitimate interests of individuals, governments (whether they have the support of their citizens or not) and corporations (whose interests often transcend national boundaries). The Internet makes all of these concerns more pressing.

11

Data, Information, and Privacy

Digital technology has brought us an enormous number of benefits, and our lives would be much poorer without it. At the same time, it has had a large effect, almost entirely negative, on our personal privacy, and (in my editorial opinion) this is likely to get worse. Some of this erosion of privacy is related to the Internet and the applications it supports, and some is merely the by-product of digital devices getting smaller, cheaper, and faster. Increased processing power, storage capacity and communications bandwidth combine to make it easy to capture and preserve information from many sources, analyze it efficiently, and disseminate it widely, all at minimal expense.

Security is often lumped with privacy, at least in the minds of individuals, since it's hard to feel safe and secure if most aspects of your personal life are widely known or easy to find out. The Internet in particular has had a significant effect on personal security—more financial than physical—since it has made it easy for information to be gathered from many places and collated, and it has opened up our lives to electronic intruders.

In this chapter, we will look at some aspects of privacy and security, and at counter-measures that individuals can take to slow down the invasion, though it might well be inevitable in the long run. It's a big subject, however, so this is a sample, not the whole story. I will mostly concentrate on topics related to the rest of the book, and won't say much about the social, economic, political and legal mechanisms by which we try to cope with technological change.

A recent newspaper advertisement says that mankind will generate about 2.5 zettabytes of data in the next few years. In case you don't recall, zetta is 10^{21}, so that's a lot of bytes by any measure. Where does all this data come from, and what's being done with it? The answers are interesting and in many ways sobering, since

some of that data is not for us but about us. The more data there is about us, the more that strangers can learn about us.

I'll start with web search, since a significant amount of data collection begins when we use search engines to find sites to visit. That leads to a discussion of tracking—what sites have you visited and what did you do while visiting. That in turn leads naturally to databases—collections of data that have been gathered by all kinds of parties—and to data aggregation and mining, because much of the value of data comes when it is combined with other data to give new insights. That's also where major privacy problems arise, since combining data about us from multiple sources can make it all too easy to learn things that are no one else's business. Next, I'll talk about the personal information people willingly give away or trade for entertainment or a convenient service. That has the side effect of making data aggregation even easier and thus more commercially valuable, though even more deleterious to our privacy. Finally, I'll discuss cloud computing, where we put our information in the hands of companies that provide storage and processing on their servers.

11.1 Search

Web search began around 1994 when the web was tiny by today's measures. The number of pages and the number of queries both rose rapidly over the next few years. The original paper on Google ("The anatomy of a large-scale hypertextual web search engine" by Sergey Brin and Larry Page, published in early 1998) said that Altavista, one of the most successful of the early engines, was handling 20 million queries per day late in 1997, and accurately predicted a web of a billion pages and hundreds of millions of queries a day by 2000. An article in the *Economist* at the end of 2010 says that Google alone handles two billion searches a day.

Search is big business, going from nowhere to a major industry in barely a decade. For example, Google was founded in 1998, went public in 2004, and had a market capitalization of $200 billion in mid 2011. That's more than many large companies with long-established businesses; for instance, it's four times what Ford Motor Company is worth and about the same as General Electric. Google is highly profitable and growing briskly, but there's plenty of competition, so who knows what might happen. (A disclosure is in order here: I am an occasional consultant and part-time Google employee, and I have many friends at the company. Naturally, nothing in this book should be taken as Google's position on anything.)

How does a search engine work? From a user's perspective, a query is typed into a form on a web page and sent to a server, which almost instantly returns a list of links and snippets of text. From the server side, it's more complicated. The server creates a list of pages that contain the word or words of the query, sorts them into order of relevance, wraps snippets of the page in HTML, and sends them back. The web is far too big for each user query to trigger a search over the web, however, so a major part of a search engine's task is to be prepared for queries by already having copies of pages stored and organized locally. This is done by a *crawl* of the web,

which scans pages and stores their contents in a database so that subsequent queries can be answered quickly. (The fact that search results are based on a pre-computed index of cached pages is not always understood; only a few years ago a national TV news program reported that "Type in 'Paris Hilton', and Google's computers search the entire Internet for every mention of her and every picture in half a second.")

This diagram shows approximately what the structure is, including the insertion of advertising into the result page:

The problem is scale. There are billions of users and many billions of web pages. Google used to report how many pages it was crawling to build its index, but stopped doing that sometime after the number of pages went past 10 billion. If a typical web page is 10 to 20 KB, storing fifty or a hundred billion pages requires a petabyte of disk space. Although some of the web is static pages that don't change for months or even years, a significant fraction changes rapidly (stock prices, news sites, blogs, Twitter feeds), so the crawl has to be ongoing; there's no chance to rest, lest the indexed information become dated. Search engines handle billions of queries a day; each of those requires scanning a database, finding the relevant pages, and sorting them into the right order. It also includes choosing advertisements to accompany the search results, and in the background, logging everything so there's data to improve search quality, keep ahead of competitors, and sell more advertising.

From our standpoint, a search engine is a nice example of algorithms in action, though the volume of traffic means that no simple algorithm is going to be fast enough.

One family of algorithms deals with the crawl: deciding what page to look at next, extracting the indexable information from it (words, links, images, etc.), and delivering those to the index builder. URLs are extracted, duplicates and irrelevancies are eliminated, and the survivors are added to a list to be examined in their turn. Complicating this is the fact that a crawler can't visit a site too often, because that could add a significant load and thus become a nuisance (and the crawler might be denied access). Since the rate of change of pages varies widely, there is also a payoff from

algorithms that can accurately assess the frequency with which pages change, and visit the rapid changers more often than the slow changers.

Index-building is the next component. This takes pages from the crawler, extracts the parts of the page to be indexed, and records the part, its location in the page, and its URL in the index. The specifics of this process depend on the content to be indexed: text, images, documents in standard formats like PDF, and so on. In effect, indexing creates a list of pages and locations for each word or other indexable item that has occurred in some web page, stored in a form that makes it possible to quickly retrieve the list for any specific item.

The final task is formulating a response to a query. The basic idea is to take all the words of the query, use the indexing lists to quickly find the URLs that match, then (also quickly) select the URLs that match best. The details of this process are the crown jewels for search engine operators, and you won't find out much about specific techniques by looking on the web. Again, scale dominates: any given word might appear in millions of pages, a pair of words might still be in a million pages, and those potential responses have to be rapidly winnowed down to the best ten or so. The better that the search engine brings accurate hits to the top and the more rapidly it responds, the more it will be used in preference to its competitors.

The first search engines merely displayed a list of pages containing the search terms, but as the web grew, search results were a jumble of largely irrelevant pages. Google's *PageRank* algorithm assigns a quality measure to each page. PageRank gives more weight to pages that other pages refer to or that are referred to by pages that themselves are highly ranked, on the theory that those are the most likely to be relevant to the query. As Brin and Page say, "Intuitively, pages that are well cited from many places around the web are worth looking at." Naturally there's far more than this to producing high-quality search results, and search engine companies are continuously trying to improve their results in comparison to their competitors.

It takes enormous computing resources to provide a search service at full scale—millions of processors, terabytes of RAM, petabytes of disk, gigabits per second of bandwidth, gigawatts of electrical power, and of course lots of people. These all have to be paid for somehow, usually by advertising revenue. At the most basic level, advertisers pay to display an advertisement on a web page, with the price determined by some measure of how many people and of what kind view the page. The price might be in terms of page views ("impressions," which count only that the ad appeared on the page) or in clicks (the viewer clicked on the ad). Viewers who might be interested in whatever is being advertised are clearly valuable, so in the most common model, the search engine company runs an auction for search terms. Advertisers bid for the right to display advertisements alongside results for specific search terms, and the company makes money when a viewer clicks on an advertisement.

For instance, someone searching for "car accident" might be interested in contacting a lawyer who specializes in liability. It's even better if the geographical location of the searcher can be used; lawyers are more likely to want to advertise to people in their own geographical area. If something is known about the demographics of the

searcher, like gender, marital status and age, that might make the search term more valuable as well. Here's the result of a search that I made from Cambridge, Massachusetts; notice that my approximate geographic location is known, though I did not volunteer that information.

The top search results (as opposed to the advertisements) were about preventing accidents, so search results were not skewed in favor of advertisers, all of which are lawyers. (The advertisement to the right doesn't seem as well targeted but it won't cost the advertiser anything unless I click on it.) If I were to click on one of the advertisements, the advertiser would pay Google the price that was set by auction. Search engine companies have elaborate mechanisms to prevent fraud from fake clicks; that's another set of implementation details that you won't see spelled out in public, especially since generating fake clicks is a favorite use of botnets, which were described in the previous chapter.

Google's AdWords makes it easy to experiment with a proposed advertising campaign. For instance, their estimation tool says that the likely cost for the search term "kernighan" will be about 65 cents per click, that is, every time someone searches for `kernighan` and then clicks on my advertisement (which they expect will be one of the top two results), I will have to pay Google 65 cents. They also estimate that there will be about 600 searches per day, though of course neither of us knows how many people would click on my advertisement and thus cost me money. I've never done the experiment to see what would really happen.

Could advertisers pay to bias the search results in favor of themselves and against competitors? This was a concern for Brin and Page, who wrote "we expect that advertising funded search engines will be inherently biased towards the advertisers and away from the needs of the consumers." Google derives most of its revenue from advertising, though it maintains a very clear separation between search results and advertisements, as do the other major search engines.

This discussion is phrased in terms of search, but of course the same kinds of considerations apply for any kind of advertising: the more accurately it can be targeted, the more likely it is to cause the viewer to respond favorably.

11.2 Tracking

It's inevitable that information about us is collected as we use the Internet; it's hard to do anything without leaving tracks. The same is true as we use other systems, especially cell phones, which know our physical location at all times. A GPS-enabled phone, which includes all smartphones, generally knows where you are to within about 10 meters when you're outside and can report your position at any time. (My phone locates me within about 20 meters even when I'm in my office in a medium-sized building.) Some digital cameras include GPS as well, which enables them to encode the geographic location in each picture they take; this is called *geo-tagging*.

When tracks like these are collected from multiple sources they can paint a detailed picture of our activities, our interests, our finances, and many other aspects of our lives. In the most benign setting, that information is used to help advertisers target us more accurately, so that we will see advertisements that we are likely to respond favorably to. But the tracking need not stop there, and its results can be used for much less innocent purposes, including redlining on the basis of income or worse, financial theft, identify theft, government surveillance, and even physical harm.

How is browsing information collected? Some information is sent with every request from your browser, including your IP address, the page that you were viewing (the "referrer"), the type and version of the browser, the operating system, and your language preference.

In addition, if there are cookies from the server's domain, those are sent as well. The cookie specification says that cookies only go back to the domain from which they originated, so how could one site use cookies to track my visits to other sites?

The answer is implicit in how links work. Web pages contain links to other pages—that's the essence of hyperlinking—and we're familiar with links that must be explicitly clicked before they're followed. But images do not need to be clicked; they are loaded automatically from their sources as the page is being loaded. If a web page contains a reference to an image, that image is loaded from whatever domain is specified. Thus when the browser fetches the image, the domain that provides the image knows which page I have accessed and it too can deposit cookies on my computer and retrieve cookies from previous visits.

This is the heart of tracking, so let's make it more specific. Suppose I'm thinking about buying a new car, and I visit `toyota.com`. This retrieves over 60 KB of HTML, along with some JavaScript and 40 images, among which is this one:

```
<img src="http://ad.doubleclick.net/ad/
            N2724.deduped_spotlight/B1009212;
               sz=1x1;tag=total_traffic;ord=1?"
     width=1 height=1 border=0>
```

This `<img>` tag fetches an image from the site `ad.doubleclick.net`. The image is 1 pixel wide, 1 pixel high, has no border and is probably transparent, so it's invisible; it's definitely not meant to be seen. When my browser requests it, however, DoubleClick knows that I was looking at a particular Toyota page and (if I allow it) can store a cookie on my computer. As I visit other sites that also use DoubleClick, it begins to build up a picture of what I'm looking at. If it's mostly about cars, DoubleClick can tell prospective advertisers and I'll start to see advertisements for car dealers, loans, repair services, and accessories. If it's about accidents and pain relief, I might see more from lawyers and therapists.

DoubleClick (which is part of Google) collects information about sites that people have visited and uses that to sell advertising space to customers like Toyota, who can in turn use it for targeted advertising and (perhaps) correlation with other information about me beyond my IP address. (DoubleClick does not sell the information itself to anyone.) As I visit more web pages, DoubleClick builds up an ever more detailed picture of my presumed characteristics and interests, and may eventually deduce that I'm over 60 years old, male, have a reasonably good income, live in central New Jersey and work at Princeton. The more they know about me, the more accurately DoubleClick's customers can target their advertising. At some point DoubleClick might be able to identify me personally, though most such companies say that they don't do that. But if I do give my name or email address on some web page, there's no guarantee that it won't be passed around.

Internet advertising is a sophisticated marketplace. When you request a web page, the web page publisher notifies an advertising exchange like Yahoo's Right Media or Google's Ad Exchange that space on that page is available, and provides information about the likely viewer (for instance, a 25-40 year old single woman in San Francisco who likes technology and good restaurants). Advertisers bid for the ad space, and the winner's advertisement is inserted into the page, all in a couple of hundred milliseconds.

Quite a few companies collect tracking information, though the industry is consolidating into a handful of large ones; DoubleClick used to be independent but was acquired by Google in 2008 for $3.1 billion. A recent paper shows that well over half of the most frequently visited web sites have tracking mechanisms turned on, some much more subtle and harder to defeat than so-called "third-party" cookies (that is, cookies like those from DoubleClick that did not come from the original domain). Many web sites have trackers from multiple operators; a page that I recently visited had *twelve* different trackers. An add-on called Ghostery disables tracking JavaScript but reports what it blocked; it's astonishing how much tracking goes on.

If being tracked does not appeal to you, it can be defeated or at least slowed down, though not without some cost. Browsers allow you to turn cookies off entirely or to disable third-party cookies, or to have them removed automatically when the browser is closed. Long-term cookies are stored in the file system on your computer, and you can remove them either with the browser or by removing the file(s) where they are stored. Most tracking companies provide an opt-out mechanism: if they encounter a specific cookie on your computer, they will not track your interactions for targeted advertising, though it's likely that they will still record your IP address. I use a Firefox add-on called TACO (Target Advertising Cookie Opt-out) that maintains a list of cookie-tracking sites (currently with about 150 names) and stores their opt-out cookies in the browser, though I also keep cookies turned off for most sites anyway.

None of these mechanisms is standardized across browsers or even across different versions of the same browser, and the default is usually to permit all cookies.

Blocking third-party cookies also doesn't help when the page content appears to the browser as if it all came from a single source but in fact is being relayed from somewhere else. A hypothetical URL like

```
http://www.bigcorp.com/relay?doubleclick.net/whatever
```

can be interpreted by a server at BigCorp and passed on to DoubleClick, which will provide the same kind of cookie information, which the BigCorp server will load in your browser. In effect this trick lets a third-party cookie masquerade as a first-party cookie.

Unfortunately, many sites don't work without cookies. Sometimes this is legitimate—perhaps they need to know that you have logged in already—but sometimes it's specifically so they can track you. This irritates me enough that I prefer not to patronize such places online or in person.

Single-pixel images or web beacons like the one described above can also be used for tracking. The URL used to fetch the pixel can include an identifying number that tells what page is being viewed and an identifier for the particular user; this is enough to keep track of your browsing.

Adobe's Flash player, which is used for a large fraction of the video on the web, including animation in advertisements, also stores "Flash cookies" on your computer. These are unrelated to the regular cookie mechanism, and are stored in a different place in the file system. They are meant to cache information that will help videos start up faster, but they are used for tracking as well. Adobe provides a mechanism for disabling them, though it requires visiting `adobe.com` and working through a series of pages. Anecdotally, I've seen no difference in performance after I turn off Flash cookies (which I have to do for each new computer), so the caching doesn't appear to have much effect.

JavaScript is used for tracking as well, since the browser will run any JavaScript code found in the original HTML file or loaded from a URL with `src="name.js"` in a `<script>` tag. This is heavily used for "analytics" that try to measure how particular pages are viewed. For example, this script loads and runs JavaScript code

from Google when the page that contains it is viewed:

```
<script type="text/javascript"
  src="http://pagead2.googlesyndication.com/pagead/show_ads.js">
</script>
```

JavaScript code can access cookies, and sometimes other information like the browser's history of pages visited. It can also monitor the position of your mouse continuously and report that back to a server, which can infer what parts of the web page might be interesting or not; it can also monitor places where you clicked even if they weren't sensitive areas like links. Surreptitious use of JavaScript appears to be increasing. If there's JavaScript on the page, it's also a potential vector for malicious attacks.

An article published late in 2010 described the "evercookie," listing over a dozen places where tracking information could be stored on a client computer, some outside the scope of the browser. It's hard for even an expert to remove all of them, and if only one instance survives, that's enough to regenerate all of them. Defenses against this specific technique were soon published, but the basic point is valid: there are many places to store information beyond the officially recognized ones. For instance, some tracking schemes are based on individual characteristics that have nothing to do with cookies, like the combination of operating system, browser, version, language preference, and installed software. Given enough of these, it's possible to distinguish and recognize individual users. Naturally advertisers and other organizations would love to have pinpoint identification of individuals.

Tracking mechanisms are not restricted to browsers, but can be used by systems like mail readers, video players, and so on that use HTTP and HTML. If your mail reader interprets HTML, then it's going to "display" those single-pixel images that permit someone to track you.

One especially pernicious tracking and monitoring scheme resurfaces from time to time: *deep packet inspection.* As we noted earlier, every IP packet travels from your computer through typically 15 to 20 gateways on its way to its destination, and the same is true of packets coming back. Each gateway along that path can inspect each packet to see what it contains and even modify the packet in some way. Usually this invasion happens at your ISP, since that's the place where you are most easily identified. Advertisers and government agencies, among others, really like this kind of information since it's not limited to web browsing, but includes all traffic between you and the Internet.

The rules governing what personally identifiable information can be collected and how it can be used vary from country to country. As a considerable over-simplification, in the US, anything goes—any company or organization can collect and distribute information about you, without notice or even offering you a chance to opt out. In the European Union (again, over-simplified), privacy is taken more seriously: companies cannot collect or use data about individuals without explicit permission from the individual.

One visible manifestation of this legal and cultural divide is shown by the different treatments of Google's Street View, which provides detailed panoramic photographic images of streets in many parts of the world. In all places, Google algorithmically blurs people's faces so individuals can't be identified, and it also blurs car license plates. In spite of this, privacy concerns have been raised in a variety of countries, which take different positions in trying to balance the value of the service against its potential for revealing private information.

11.3 Databases, Information, and Aggregation

The Internet and the web have caused a revolution in how people collect, store and present information; search engines and databases are of enormous value to everyone, to the point where it's hard to remember how we managed before the Internet. As always, there are some downsides to the proliferation of online data as well, especially for information that might reveal more about ourselves than is comfortable.

Some information is clearly public, and some collections of information are meant to be searched and indexed. If I create a web page with the hope that you will visit it, say the web page for this book, then it's definitely to my advantage to have it easily found by search engines.

What about public records? Legally, certain kinds of information are "public records," in that they are available to any member of the public upon application. In the United States, this typically includes court proceedings unless sealed, mortgage documents, house prices, local property taxes, birth and death records, marriage licenses, political contributions, and the like. (Note that birth records are likely to reveal the "mother's maiden name" that was often used as part of verifying one's identity.) In olden times it required a trip to a local government office to get this information, so although the records were technically "public," they were not accessible without some effort; the person seeking the data had to show up in person, perhaps identify himself or herself, and maybe pay for each physical copy. Today, if the data is available online, I can examine public records anonymously from the comfort of my home. I might even be able to run a business by collecting them in volume and combining them with other information. The popular site `zillow.com` combines maps, real estate advertisements and public data on properties and transactions to display house prices on a map. This is a valuable service if one is looking to buy or sell but otherwise might be thought intrusive. The Federal Election Commission database of election contributions (`fec.gov`) shows which candidates have been supported by friends and notables, and can reveal information like home addresses; `fundrace.huffingtonpost.com` organizes the FEC information, including name, address and occupation, and superimposes it on a map. This strikes an uneasy balance between the public's right to know and an individual's right to privacy.

Questions about what information should be so easily accessible are hard to answer. Political contributions should be public but street addresses should probably

be redacted. Public documents that include personal identifiers like US Social Security numbers should not be put on the web since they are too readily used for identity theft. But existing laws don't always prevent the release of such information and the horse has left the barn in many cases—once on the web, information is always on the web.

Concerns about freely available information become more serious when data is combined from multiple sources that might have seemed independent of each other. For instance, companies that provide web services record a great deal of information about their users. Search engines log all queries, with as much information as possible about the person who made the query. At a minimum, this will include the IP address from which the query came and any cookies stored on the user's computer from a previous visit.

In August 2006, with the best of intentions, AOL released a large sample of query logs for research. The logs of 20 million queries by 650,000 users over three months had been anonymized, so that in theory anything that could have identified individual users was completely removed. Good intentions notwithstanding, it rapidly became clear that in practice the logs were not nearly as anonymized as AOL had thought. Each user had been assigned a random but unique identification number, and this made it possible to find sequences of queries made by the same person, which in turn made it possible to identify at least a few individuals uniquely. People had searched for data like their own names and addresses, Social Security numbers, and other personal information where correlation of searches revealed more than AOL had believed and certainly much more than the original users would have wanted. AOL quickly removed the query logs from its web site, but of course that was far too late; the data had spread all over the world, and in fact it's still available, along with tools to help analyze it.

Query logs contain valuable information for running a business and improving the service but clearly they also contain potentially sensitive personal information. How long should Google, Yahoo and Microsoft retain such information? There are conflicting external pressures here: a short period for privacy reasons versus a long period for law enforcement purposes. How much should they process the data internally to make it more anonymous? All claim to remove part of the IP address for each query, typically the rightmost byte, but that may not be enough to de-identify users. What access should government agencies have to this information? How much would be discoverable in a civil action? The answers to these questions are far from clear. Some of the queries in the AOL logs were scary—questions about how to kill one's spouse, for example—so it might be desirable to make logs available to law enforcement in restricted situations, but where to draw the lines is most uncertain.

The AOL story illustrates a general problem: it's hard to truly anonymize data. Attempts to remove identifying information tend to take a narrow view: there's nothing in this particular data that could identify a specific person so it must be harmless. But in the real world, there are other sources of information, and it is often possible to learn a great deal by combining facts from multiple sources that might have been

completely unknown to the original providers and perhaps did not even exist until later. For example, suppose that a search engine does remove the rightmost byte of the IP address for each query. That would still leave my queries clearly traceable to the computer science department at Princeton, and if that information were combined with Princeton's log of when I was connected to the Internet with that IP address, it might be possible to link me to specific queries.

A famous example brought this re-identification problem to the surface vividly. In 1997, Latanya Sweeney, at the time a PhD student at MIT, analyzed ostensibly de-identified medical records for 135,000 Massachusetts state employees. The data had been released by the state's insurance commission for research and was even sold to a private company. Among many other things, each record included birth date, gender and ZIP code. Sweeney found six people born on July 31, 1945; three were male, and only one lived in Cambridge. Combining this information with public voter registration lists, she was able to identify that person as William Weld, the governor of Massachusetts at the time.

Perhaps there is a very loose analogy between anonymized data and security by obscurity, which we discussed in the previous chapter. In each case, it's tempting to believe that no one can figure out the secrets because they don't know enough. But in each case, adversaries may already know more than you think, and even if they don't today, more information will become available over time.

11.4 Giving It Away

Not long ago, I came across a web post that went something like this: "In a job interview, they asked me questions about things that weren't mentioned in my resume. They had been looking at my Facebook page, which is outrageous because Facebook is about my private life and has nothing to do with them." This is touchingly naive and innocent, but one suspects that many Facebook users would feel similarly violated, even though it's well known that employers and college admissions offices routinely use search engines, social networking and similar sources to learn more about applicants. In the US, it's illegal to ask a job applicant about race, religion, sexual orientation, and a wide variety of other personal information. But those can be readily, and quietly, determined from social networks and a bit of searching.

It's important to realize that tracking the web sites we visit is not the only way that information about us can be gathered. Indeed, social networks, broadly taken, are where we *voluntarily* give up a remarkable amount of personal privacy in return for entertainment and keeping in touch with other people.

Almost by definition, social networking sites are going to have privacy problems because they collect a great deal of information about their members, and they make their money by selling it to advertisers. They have existed only for a short time and have grown in size dramatically; Facebook was founded in 2004 and now claims more than 750 million users, that is, over ten percent of everyone on the planet. That growth rate doesn't allow much time for careful consideration of policies, nor does it

allow for leisurely development of robust computer programs. As a result, every social networking site has had problems with divulging private information through ill-considered features, user confusion over how to manage privacy settings (which change frequently), software errors, and data disclosures that are inherent in the overall system.

As the largest and most successful social network, Facebook's issues have been most visible. Some have come about because Facebook provides an API for third parties to write applications that run in the context of a Facebook user, and those have sometimes compromised privacy and revealed private information contrary to the company's privacy policies. Of course none of this is unique to Facebook.

Geolocation services like Foursquare display their users' locations on cell phones; this makes it easy to meet friends in person or to play location-based games. Targeted advertising can be especially effective when a potential customer's physical location is known; you're more likely to respond to an offer from a restaurant when you're standing outside its door than reading about it in a newspaper. Although it's fine to tell your friends where you are, broadcasting that information to the whole world is foolish. For instance, a demonstration called Please Rob Me scraped information from Twitter posts made by Foursquare users that could be used to deduce that they were away from home and thus vulnerable to burglary.

"Location privacy"—the right to have your location remain private—is compromised by a large number of systems that we use all the time, including credit cards, toll payment systems on highways and public transport, and of course cell phones. It's harder and harder to avoid leaving a trail of every place you've been. Cell phone apps often request access to essentially everything that the phone knows about you, including call data, information stored locally, physical location, and so on; it seems to me that they ask for much broader access than is necessary.

Personal information leaks from social networking and other sites even for non-users. For instance, if I get an electronic invitation (an "e-vite") to a party from a well-meaning friend, the company that runs the invitation service now has a guaranteed email address for me, even though I do not respond to the invitation or in any way give permission to use my address. If a friend sends me mail from his Gmail or Yahoo account, my email address has been given away without my permission. If a friend tags me in a picture that is posted on Facebook or Flickr (or both), my privacy has been compromised without my consent. Facebook offers image recognition so that friends can tag more easily, and the default is to allow tagging without the taggee's permission. It's easy for any system with a large user population to create a "social graph" of interactions among its direct users and to include those who are brought in without their consent or even knowledge. In all of these cases, there's really no way for an individual to avoid the problem ahead of time and no way to remove the information once it has been created.

Intelligence agencies have known for a long time that they can learn a great deal from traffic analysis—who talks to who—without knowing what the parties said. The same is true of the explicit and implicit connections in social networks. For

example, in 2009 a pair of MIT students reported that they could predict the sexual orientation of men by data-mining the stated orientations of their Facebook friends. Whether this specific instance is accurate or not, it indicates the kind of inferences that could be feasible. It is definitely the case that governments have mined the information on the Facebook pages of dissidents to learn more about who they associate with.

11.5 Cloud Computing

Think back to the model of computation described in Chapter 6. You have a personal computer, or several. You run separate application programs for different tasks, for example a word processor for creating documents, Quicken or a spreadsheet for personal finance, and iPhoto or Picasa for managing your pictures. The programs run on your own computer, though they may contact the Internet for some services. Every so often you get a chance to download a new version that fixes bugs, and occasionally you might have to buy an upgrade to get new features.

The essence of this model is that the programs and their data reside on your own computer. If you need to access a file that's stored on a computer at home while you're in your office or away on a trip, you're out of luck. If you need Excel or Powerpoint on both a PC and a Mac, you have to buy one for each. If you change a file on one computer and then need it on another, you have to transfer it yourself.

A different model is becoming common: using a browser to access and manipulate information that is stored on Internet servers. Mail services like Gmail and Yahoo mail are the most widespread example. Mail can be accessed from any computer, and from many cell phones. It's possible to upload a mail message that was composed locally or to download messages to a local file system, but this isn't the norm; more often you just leave the information wherever it is. There's no notion of software update, though from time to time new features appear. Almost without exception, the service is free; the only "cost" is that you might see advertisements as you read your mail, though if you're like me you don't even notice them. (Every once in a while I do, and am often puzzled. For example, I got mail from a friend thanking me for reading a manuscript; it was accompanied by ads for psychic services, apparently triggered by the word "reading".)

This model is often called *cloud computing*, because of the metaphor that the Internet is a "cloud" with no particular physical location, and information is stored somewhere "in the cloud." Mail is the most common cloud service, but there are plenty of others, like Flickr, Dropbox, online calendars, and social networks. Data is not stored locally but in the cloud, that is, by the service provider—your Gmail is stored on Google servers, your pictures are on Flickr servers, and so on. Programs may be implemented as some combination of local (running in the client) and in the cloud (in the server) but they come from the cloud to your browser.

Cloud computing is made possible by the alignment of a number of factors. As personal computers have become more powerful, so have browsers; browsers can

now efficiently run large programs with intensive display requirements, even though the programming language is interpreted JavaScript. Bandwidth and latency between client and server are much better now for most people than they were say ten years ago, and this makes it possible to send and receive data quickly, even responding to individual keystrokes. As a result a browser can do most of the user interface operations that would have required a standalone program in the past, while using a server to hold the bulk of the data.

A browser-based system can be nearly as responsive as a desktop system, while providing access to data from anywhere. One graphic illustration is the evolution of cloud-based "office" tools like Google Docs, which provides a word processor, a spreadsheet, and a presentation program that compete, at least in principle, with Microsoft Office, but include features like access from anywhere and simultaneous update by multiple users. One of the interesting issues is whether these cloud versions will ultimately run well enough to cut the ground out from under desktop versions. As might be imagined, Microsoft is concerned because Office provides a significant fraction of the company's revenue, and because Office runs primarily on Windows, which provides much of the rest of the revenue. Browser-based word processors and spreadsheets do not need anything from Microsoft and thus threaten both core businesses. At the moment, Google Docs and similar systems do not provide all the features of Word and Excel, but the history of technological advance is replete with examples where a clearly inferior system appeared, picked up new users for whom it was good enough, and gradually ate away at the incumbent systems. (This is well described in *The Innovator's Dilemma*, by Clayton Christensen.) Microsoft is obviously well aware of cloud versions of such programs, and in fact provides online versions of Office software.

Cloud computing relies on fast processing and lots of memory on the client, and high bandwidth to the server. The client side code is written in JavaScript and is usually quite intricate. The JavaScript code makes heavy demands on the browser to update and display graphical material quickly, responding to user actions like dragging and server actions like updated content. This is hard enough, but it's made worse by incompatibilities among browsers and versions of JavaScript, which require the supplier to figure out the best way to send the right code to the client. Both of these will improve, however, as computers get faster and standards are more closely adhered to.

Cloud computing can trade off where computation is performed and where information resides during processing. For example, one way to make JavaScript code independent of the particular browser is to include tests in the code itself, like "if the browser is Internet Explorer version 8 do this, else if it's Safari 5 do this, else do that." Such code is bulky, which means that it requires more bandwidth to send the JavaScript program to the client, and the extra tests might make the browser run more slowly. As an alternative, the server can ask the client what browser is being used and then send code that has been tailored to that specific browser. That code is likely to be more compact and run more quickly, though for a small program it might not

make much difference.

Web page contents can be sent uncompressed, which requires less processing at both ends, but more bandwidth; the alternative is to compress, which takes less bandwidth but requires processing on each end. Sometimes the compression is on one end only; large JavaScript programs are often squeezed by removing all unnecessary spaces and by using one and two-letter names for variables and functions; the results are inscrutable to human readers but the client computer doesn't care.

In spite of technical challenges, cloud computing has much to offer, assuming you always have access to the Internet. Software is always up to date, information is stored on professionally managed servers with plenty of capacity, and client data is backed up all the time so there's little chance of losing information. There's only one copy of a document, not multiple potentially inconsistent copies on different computers. It's easy to share documents and even to collaborate on them in real time. The price is hard to beat.

On the other hand, cloud computing raises difficult privacy and security questions. Who owns data stored in the cloud? Who has access to it and under what circumstances? Is there any liability if information is accidentally released? Who could force the release of the data? For instance, in what situations would your mail provider voluntarily or under threat of legal action release your correspondence to a government agency or as part of a lawsuit? Would you find out if it did? How does the answer depend on what country's government it is, and where you live? What if you're a resident of the European Union, with its relatively strict rules about the privacy of personal data, but your cloud data is stored on servers in the US and subject to laws like the Patriot Act?

These are not hypothetical questions. As a university professor, I necessarily have access to private information about students—grades, of course, but occasionally sensitive personal and family information—that arrives by email and is stored on university computers. Would it be legal for me to use Gmail for the mail and Google Docs for storing and processing my grade files and correspondence? If some accident on my part caused this information to be shared with the world, what could happen? What if Google were subpoenaed by a government agency seeking information about a student or a bunch of them? I'm not a lawyer and I don't know the answers, but I worry about this, so I don't use cloud services for student records and communications. If I maintain all such material on computers provided by my employer, then at least if something private leaks out through their negligence or error, I should be somewhat protected from liability claims. Of course, if I screw up personally, it probably doesn't matter where the data was kept.

Who can read your mail and under what circumstances? This is partly a technical question and partly a legal one, and the answer to the legal part depends on what jurisdiction you live in. In the United States, my understanding is that if you are an employee of a company, your employer can read your mail on company accounts at will and without notice to you, period. That's true whether the mail is business related or not, on the theory that since employers are providing the facilities, they

have the right to make sure that their facilities are being used for business purposes and in accordance with company and legal requirements.

My mail is usually not very interesting, but it would bother me a lot if my employers were to read it without a serious reason even if they have the legal right to do so. If you're a student, most universities take the position that student email is private in the same way that student paper mail is. In my experience, students don't use their university accounts much anyway except as relays; they forward everything to Gmail, thus removing any protection they might have had.

If you use an ISP or a cloud service for your personal mail, as many people do (e.g., Gmail, Yahoo, Hotmail, Verizon, AOL, and lots of smaller operations), privacy is between you and them. In general, such services take the public position that their customers' email is private, and they won't look at it or reveal it short of receiving a subpoena, but they don't discuss whether they will resist subpoenas that seem too broad or informal requests that come cloaked as "national security." You're relying on the willingness of that provider to stand up to powerful pressures. Trial balloons have been floated by the US government to obtain easier access to email, in the interests of fighting organized crime before 9/11 and terrorism after. The pressure for this kind of access will surely increase.

A related use of the term "cloud computing" applies to services provided by companies like Amazon that offer virtual machines upon which anyone can use Amazon's computers, storage and bandwidth. Amazon's EC2, or "Elastic Compute Cloud," is a cloud computing service whose capacity for any user can grow or shrink as load changes; pricing is usage sensitive, but Amazon has the resources to let individual users scale up or down instantaneously.

Suppose that I want to use EC2 to provide a service, "Space in the Cloud," that will let my users upload any information they want and have it synchronized automatically among all devices that use it; I will offer to encrypt my users' data so that no prying eyes can see it. I write the programs for my cloud service, upload them to Amazon's servers, and go public. Now who has what responsibility and liability for maintaining privacy? What happens if through a bug in Amazon's service it's possible for someone else to access the files, or credit card information from my paying customers, or their tax returns? What happens if my venture fails—who has access to the data that I've collected? What should I do when the government comes calling to ask about suspicious user behavior? After all, I have the encryption keys, since my service does the encryption.

Leaving aside privacy and security concerns, what liability does Amazon or another provider have? For instance, in April 2011, part of Amazon's cloud service was down for more than a day, taking out some major web sites. What recourse do such sites have if a configuration error causes their traffic to be slowed unacceptably? Service-level agreements are the standard way of spelling this out in contracts, but a contract doesn't ensure good service; it only provides a basis for legal action when something goes seriously wrong.

Late in 2010, Amazon terminated the service it was providing to Wikileaks, a site that had been posting secret US diplomatic cables that were causing considerable embarrassment, if not real harm. Amazon said that Wikileaks was violating the terms of service by posting material that it did not own or have the rights to, and that it could also cause injury to others; both of these activities are against the terms. Amazon took this action without an official government request, though it appears there may have been informal pressure. At the same time, a number of financial institutions, including PayPal, MasterCard, Visa and Bank of America, stopped providing payment service for Wikileaks, and one of Wikileaks' DNS providers dropped them. Meanwhile, Twitter received government requests for all its records of several people involved with Wikileaks; Twitter resisted, especially the requirement of keeping the request secret, but one can't count on pushback from every organization every time. And of course there are "good" dissidents (as in Tunisia and Egypt) and "bad" dissidents (as in Wikileaks). It's hard to support one kind while cracking down on the other kind, at least while maintaining a consistent philosophical position.

What responsibilities does a service provider have to its customers? When should it stand up and fight and when is it likely to yield to legal threats or quiet requests from "the authorities"? There's no limit to the number of such questions, and very few clear answers.

11.6 Summary

A very effective AT&T advertising slogan of the early 1980s said "Reach out and touch someone." The web, email, social networks, cloud computing, and peer-to-peer systems all make that easy. Sometimes this is just fine; you can make friends and share interests in a far larger community than you could ever meet in person. At the same time, the act of reaching out makes you visible and accessible throughout the world, where not everyone has your best interests at heart. That opens the door to spam, scams, spyware, viruses, tracking, surveillance, identify theft, and loss of your privacy and even your money. It's wise to be wary.

12

Wrapping Up

We've covered a lot of ground. What should you have learned along the way? What's likely to matter in the future? What computing issues will we still be wrestling with in five or ten years? What will be passé or irrelevant?

Superficial details change all the time, and many of the technical minutiae that I've talked about aren't terribly important except as a concrete way to help you understand how things work—most people learn better from specific instances than from abstractions, and computing has altogether too many abstract ideas.

On the hardware side, it's helpful to understand how computers are organized, how they represent and process information, what some of the jargon and numbers mean, and how they have changed over time.

For software, it's important to know how one defines tasks precisely, both as abstract algorithms (with some sense of how their computation time grows with the amount of data) and as concrete computer programs. Knowing how software systems are organized, how they are created with programs in a variety of languages, and often built from components, helps you to understand what is behind the major pieces of software that we all use. With luck, the bit of programming in a couple of chapters is enough that you could reasonably think about writing more code yourself.

Communications systems operate both locally and across the world. It's important to understand how information flows, who has access to it, and how it's all controlled. Protocols—rules for how systems interact—are very important as well, and their properties can have profound effects, as seen in the problems of authentication in the Internet today.

Some computing ideas are useful ways of thinking about the world. For example, I have frequently made the distinction between logical structure and physical implementation. This central idea shows up in myriad guises. Computers are a fine

example: how computers are constructed changes rapidly but the architecture has remained much the same for a long time. More generally, all computers have the same logical properties—they can compute the same things. In software, code provides an abstraction that hides implementation; implementations can be changed without changing things that use them. Virtual machines, virtual operating systems, and indeed even real operating systems are all examples of the use of interfaces to separate logical structure from actual implementation. Arguably, programming languages also serve this function, since they make it possible for us to talk to computers as if they all spoke the same language and it was one that we understood too.

Computer systems are good examples of engineering tradeoffs, a reminder that one never gets something for nothing—there is no free lunch. As we saw, a desktop, a laptop, a tablet and a cell phone are equivalent computing devices, but they differ markedly in how they deal with constraints on size, weight, power, and cost.

Computer systems are also good examples of how to divide large and complicated systems into small and more manageable pieces that can be created independently. Layering of software, APIs, protocols and standards are all illustrations.

The "universals" that I mentioned in the preface will remain important for understanding digital technology. To recapitulate:

First is the *universal digital representation of information*. Chemistry has more than 100 elements. Physics has well over a dozen elementary particles. Digital computers have two elements, zero and one, and everything else is built of those. Bits represent arbitrary kinds of information, from the simplest binary choices like true and false or yes and no, through numbers and letters, to anything at all. Large entities—say the record of your life derived from your shopping, browsing and cell phone history—are collections of simpler data items, and so on, down to the level of individual bits.

Second is the *universal digital processor*. A computer is a digital device that manipulates bits. The instructions that tell the processor what to do are encoded as bits and generally stored in the same memory as data; changing the instructions causes the computer to do something different, and is the reason why computers are general-purpose machines. The meaning of bits depends on context—one person's instructions are another person's data. Processes like copying, encryption, compression, and error detection can all be performed on bits independent of what they represent, though specific techniques may work better on known kinds of data. The trend to replacing specialized devices with general-purpose computers running general-purpose operating systems is likely to continue. The future may well bring other kinds of processors based on biological computing elements or perhaps quantum computers, but digital computers will be with us for a long while.

Third is the *universal digital network*, which carries the bits, both data and instructions, from one processor to another, anywhere in the world. It's likely that the Internet and the telephone network will blend together into a more truly universal network, mimicking the convergence of computing and communications that we seen

in cell phones today. The Internet is sure to evolve, though it remains an open question whether it will retain its free-wheeling wild-west character or become more constrained and controlled by business and government. (My bet is the latter, unfortunately.)

Finally, the *universal availability of digital systems*. Digital devices are likely to continue to get smaller, cheaper and faster as they incorporate technological improvements in multiple areas. Improvements in a single technology like storage density often have an effect on essentially all digital devices.

The core limitations and likely problems of digital technology will remain operative, and you should be aware of them. Technology contributes an amazing number of positive things, but it raises new forms of difficult issues and exacerbates existing ones. Here are some of the most important.

Personal privacy is under continuous threat from attempts to subvert it for commercial and governmental purposes. Extensive collection of our personal data will continue apace; personal privacy is likely to diminish even further than it already has. The Internet originally made it too easy to be anonymous, especially for bad practices, but today it is often too difficult to remain anonymous even with good intentions. Government proposals in several countries to require mandatory id's to use the Internet are not going to help the good guys much, though they will provide aid and comfort (and a single point of failure) for the bad guys. Oversimplifying, but not a lot, governments want their own citizens to be easily identifiable and monitored, but support the anonymity of dissidents in other countries. Businesses are eager to know as much as possible about current and potential customers. Once information is on the web, it's there forever; there's no practical way to call it back.

Surveillance technology, from tiny cameras and microphones to web tracking to recording where our cell phones are, continues to improve, and the exponentially decreasing cost of storage and processing makes it more and more feasible to keep an almost complete digital record of our entire lives. How much disk space would it would take to record everything you have heard in your life, and how much would that storage cost? If you're 20 years old, the answer is about 10 TB; today's price is under $500, and will surely be less when you read this. A complete video record would not be more than a factor of 10 or 20 larger.

Security for individuals, corporations and governments is also an ongoing problem. I'm not sure that terms like cyber-warfare, or indeed cyber-anything, are helpful, but it is certain that all individuals and larger groups are potentially and often actually under some kind of cyber-attack.

Copyright is exceptionally difficult in a world where it is possible to make unlimited copies of digital material at zero cost and distribute them throughout the world, also at zero cost. Traditional copyright functioned at least acceptably well for creative works before the digital era, when manufacturing and distribution of music, movies, and books cost money and required expertise and specialized equipment. Those days are gone. Copyright and fair use are being replaced by licensing and

digital rights management, which don't impede true pirates, though they do inconvenience ordinary people. How do we protect the interests of authors, composers, performers, film makers, and programmers, while ensuring that their works are not restricted forever?

Patents are a serious and growing problem. As more and more devices contain general-purpose computers controlled by software, how do we protect the legitimate interests of innovators while preventing extortion by the holders of too broad or inadequately researched patents?

Resource allocation, particularly scarce but valuable resources like spectrum, is always going to be contentious. Incumbents—those who already have the allocation, like big telecom companies—have a great advantage here, and they can use their position to maintain it through money, lobbying and natural network effects.

Jurisdiction is also difficult in a world where information can travel everywhere. Business and social practices that are legal in one jurisdiction may be illegal in others. Legal systems have not caught up with this at all. The problem is seen in issues like taxes across state borders in the US, in conflicting data privacy rules in the EU and the US, and in forum-shopping where plaintiffs start legal actions like patent or libel suits in jurisdictions where they expect more favorable outcomes, regardless of where the offenses or the defendants might be.

Control is perhaps the largest issue of all. Governments want to control what their citizens can say and do on the Internet, which of course is increasingly a synonym for all media. Companies want to control what their customers and their competitors can do. Individuals would like to limit the reach of both governments and corporations, but the playing fields are far from level.

Finally, one must always remember that although technology is changing very rapidly today, people are not; in most respects we're much the same as we were thousands of years ago, with similar proportions of good and bad people operating from good and bad motives. Social, legal and political mechanisms do adapt to technological changes, but it's a slow process and it moves at different speeds and comes to different solutions in different parts of the world. It's going to be fun to see how things evolve over the next few years. I hope that this book will help you anticipate, cope with, and positively influence some of the inevitable changes.

Notes

This section collects notes on sources (though by no means complete), including books that I have enjoyed and think you might as well. As always, Wikipedia is an excellent source for a quick survey and basic facts for almost any topic, and search engines do a fine job of locating related material; I have not tried to provide direct links for information that is readily found online.

ix: The IBM 7094 had about 150KB of RAM, a clock speed of 500 KHz, and cost nearly $3 million: http://en.wikipedia.org/wiki/IBM_7090.
x: Richard Muller, *Physics for Future Presidents*, Norton, 2008. An excellent book, and one of the inspirations for this one.
x: Hal Abelson, Ken Ledeen, Harry Lewis, *Blown to Bits: Your Life, Liberty, and Happiness After the Digital Explosion*, Addison-Wesley, 2008. Touches on many important social and political topics, especially about the Internet. Bits and pieces, so to speak, would make good material for my Princeton course, and it derives from an analogous course at Harvard.
4: James Gleick, *The Information: A History, A Theory, A Flood*, Pantheon, 2011. Interesting material on communications systems, focusing on Claude Shannon, the father of information theory. The historical parts are especially intriguing.
7: James Essinger, *Jacquard's Web: How a Hand-loom Led to the Birth of the Information Age*, Oxford University Press, 2004. Follows Jacquard's loom through Babbage, Hollerith and Aiken.
8: Doron Swade, *The Difference Engine: Charles Babbage and the Quest to Build the First Computer*, Penguin, 2002. Swade also describes the construction of one of Babbage's machines in 1991, now housed in London's Science Museum; a 2008 clone is in the Computer History Museum in Mountain View, California. See also http://www.computerhistory.org/babbage.
8: "Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the

engine might compose elaborate and scientific pieces of music of any degree of complexity or extent." From Ada Lovelace's translation and notes on Menabrea's "Sketch of the Analytical Engine", 1843.
8: Scott McCartney, *ENIAC: The Triumphs and Tragedies of the World's First Computer*, Walker & Company, 1999.
11: Burks, Goldstine and von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument", http://www.cs.unc.edu/~adyilie/comp265/vonNeumann.html.
14: Online copy of *Pride and Prejudice*: http://www.gutenberg.org/ebooks/1342.
16: Charles Petzold, *Code: The Hidden Language of Computer Hardware and Software*, Microsoft Press, 2000. How computers are built from logic gates; it covers a level or two below what this book does.
18: Gordon Moore, "Cramming more components onto integrated circuits," ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf.
28: "If the base 2 is used the resulting units may be called binary digits, or more briefly *bits*, a word suggested by J. W. Tukey." Claude Shannon, in *A Mathematical Theory of Information*, 1948.
30: Seagate settlement: http://www.harddrive-settlement.com.
34: Donald Knuth, *The Art of Computer Programming*, *Vol 2: Seminumerical Algorithms*, Addison-Wesley, 1997.
45: Alan Turing, "Computing machinery and intelligence," http://loebner.net/Prizef/TuringArticle.html. *The Atlantic* has an informative and entertaining article on the Turing test at http://www.theatlantic.com/magazine/archive/2011/03/mind-vs-machine/8386.
46: The CAPTCHA is a public domain image from http://en.wikipedia.org/wiki/File:Modern-captcha.jpg.
46: Turing home page maintained by Andrew Hodges: http://www.turing.org.uk/turing. Hodges is the author of the definitive biography *Alan Turing: The Enigma*, Burnett Books, 1983.
46: ACM Turing Award: http://awards.acm.org/homepage.cfm?awd=140.
67: Steve Lohr, *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts—the Programmers Who Created the Software Revolution*, Basic Books, 2001.
69: Kurt Beyer, *Grace Hopper and the Invention of the Information Age*, MIT Press, 2009. Hopper was a remarkable figure, a computing pioneer of great influence, and on her retirement at 79, the oldest commissioned officer in the US Navy. One of her set pieces in speeches was to hold out her hands a foot apart and say "That's a nanosecond".
73: NASA Mars Climate Orbiter report: ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf.
75: The bug picture is a public domain image from http://en.wikipedia.org/wiki/File:H96566k.jpg.
78: Supreme Court decision confirming the constitutionality of the 1998 Sonny Bono Copyright Term Extension Act (sarcastically known as the Mickey Mouse Protection Act): http://en.wikipedia.org/wiki/Eldred_v._Ashcroft.
79: Larry Lessig, *Code 2.0*, Basic Books, 2005. The book is also available online at http://codev2.cc under a Creative Commons license.

79: Amazon 1-click patent: http://www.google.com/patents?id=O2YXAAAAEBAJ.
102: Court's Findings of Fact, paragraph 154, 1999, at http://www.justice.gov/atr/cases/f3800/msjudgex.htm. The case finally ended in 2011 when the last oversight of Microsoft's compliance ended.
112: One useful JavaScript site is http://w3schools.com.
118: Gerard Holzmann and Bjorn Pehrson, *The Early History of Data Networks*, IEEE Press, 1994. Detailed and very interesting history of the optical telegraph.
118: The optical telegraph drawing is a public domain image from http://en.wikipedia.org/wiki/File:Chappe_telegraf.jpg.
119: Tom Standage, *The Victorian Internet: The Remarkable Story of the Telegraph and the Nineteenth Century's On-Line Pioneers*, Walker, 1998. Fascinating and entertaining reading.
119: Rotary dial telephone photograph courtesy of John Wait.
121: Edward Felten, "Nuts and Bolts of Network Neutrality", http://citp.princeton.edu/pub/neutrality.pdf.
128: RJ45 connector picture courtesy of Gerard Holzmann.
131: Guy Klemens, *Cellphone: The History and Technology of the Gadget that Changed the World*, McFarland, 2010. Detailed history and technical facts on the evolution of cell phones. Some parts are heavy going, but much is quite accessible; it gives a good picture of the remarkable complexity of a system we take for granted.
139: RFC on Avian carriers: http://tools.ietf.org/html/rfc1149.
149: SMTP session at http://technet.microsoft.com/en-us/library/bb123686.aspx.
154: Department of Justice domain seizures: http://viewdns.info/research/inside-the-dojs-domain-name-graveyards.
167: Microsoft's *10 Immutable Laws of Security*: http://technet.microsoft.com/en-us/library/cc722487.aspx.
177: RSA's public statement on the RSA hack: http://www.rsa.com/node.aspx?id=3891.
179: Simon Singh, *The Code Book*, Anchor, 2000. A delightful history of cryptography for the general reader. The Babington Plot (the attempt to put Mary, Queen of Scots on the throne) is fascinating.
180: David Touretzky, a professor of computer science at Carnegie-Mellon, maintains a rich collection of material on DeCSS at http://www.cs.cmu.edu/~dst/DeCSS/index.html.
181: Alice, Bob and Eve at http://xkcd.com/177.
183: RSA factoring challenge: http://www.rsa.com/rsalabs/node.asp?id=2092.
187: Bruce Schneier, *Secrets and Lies: Digital Security in a Networked World*, Wiley, 2000. Security and privacy threats and breaches, quite accessible to the non-specialist. Schneier's security blog is at http://www.schneier.com.
188: The original Google paper: http://infolab.stanford.edu/~backrub/google.html. The system really was called "BackRub" in its first incarnation.
188: Ken Auletta, *Googled: The End of the World As We Know It*, Penguin, 2009.
188: Steven Levy, *In the Plex: How Google Thinks, Works, and Shapes our Lives*, Simon & Schuster, 2011.
191: Siva Vaidhyanathan, *The Googlization of Everything*, University of California Press, 2011.
193: Bala Krishnamurthy and Craig Wills on tracking: http://www.research.att.com/~bala/papers/www09.pdf. Krishnamurthy's web site has numerous empirical studies of privacy.

199: Locational privacy: http://www.eff.org/wp/locational-privacy. The Electronic Frontier Foundation at eff.org is a good source for privacy and security policy information.
200: Carter Jernigan and Behram Mistree, "Gaydar: Facebook friendships expose sexual orientation," http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/2611/2302.
200: Evgeny Morozov, *The Net Delusion: The Dark Side of Internet Freedom*, Public Affairs, 2011. The same technology and systems that make it easy for us to keep in contact with each other make it easy for the authorities to keep track of us.
201: Clayton Christiansen, *The Innovator's Dilemma*, Harper, 1997. Introduced the idea of disruptive technologies that chip away at existing technologies and companies, starting at the low-price end, and eventually take over.
207: Jonathan Zittrain, *The Future of the Internet—And How to Stop It*, Penguin, 2008. The Internet is moving from a wide-open environment that anyone with a good idea can add to, to a closed "appliancized" system controlled by vendors.

Glossary

The glossary provides very brief definitions or explanations of important terms that appear in the book, focusing on ones that use ordinary words but with special meanings, and that you are likely to see frequently.

Computers and communications systems like the Internet deal with very large numbers, often expressed in terms of unfamiliar units. The table below defines all the units that appear in the book, along with the others of the International System of Units (SI). As technology advances, you'll see more of the ones that represent big numbers. The table also shows the nearest powers of two. The error is only 21 percent at 10^{24}; that is, 2^{80} is about 1.21×10^{24}.

SI name	power of 10	common name	nearest power of 2
yocto	10^{-24}		2^{-80}
zepto	10^{-21}		2^{-70}
atto	10^{-18}		2^{-60}
femto	10^{-15}		2^{-50}
pico	10^{-12}	trillionth	2^{-40}
nano	10^{-9}	billionth	2^{-30}
micro	10^{-6}	millionth	2^{-20}
milli	10^{-3}	thousandth	2^{-10}
-	10^{0}		2^{0}
kilo	10^{3}	thousand	2^{10}
mega	10^{6}	million	2^{20}
giga	10^{9}	billion	2^{30}
tera	10^{12}	trillion	2^{40}
peta	10^{15}	quadrillion	2^{50}
exa	10^{18}	quintillion	2^{60}
zetta	10^{21}		2^{70}
yotta	10^{24}		2^{80}

3G Third generation, a not very precise term characterizing the bandwidth of smartphones like iPhone and Android.
802.11 The standard for wireless systems like those used in laptops and home routers; also Wi-Fi.
add-on A small program added to a browser for extra features or convenience; privacy add-ons like Adblock and NoScript are examples.
AES Advanced Encryption Standard, a very widely used secret-key encryption algorithm.
algorithm A precise and complete specification of a computational process, but abstract and not directly executable by a computer, by contrast to a program.
AM Amplitude modulation, a mechanism for adding information like voice or data to a signal; usually seen in the context of AM radio.
analog Generic term for representation of information that uses a physical property that varies smoothly in proportion, such as the level of liquid in a thermometer; contrast with digital.
API Application Programming Interface, a description for programmers of services provided by a library or other collection of software; for instance the Google Maps API describes how to control map displays with JavaScript.
app, application A program or family of programs that perform some task, for example Word or iPhoto; *app* is most often used for cell phone applications like calendars and games.
architecture An imprecise word for the organization or structure of a computer program or system.
ASCII American Standard Code for Information Interchange, an 8-bit encoding of letters, digits, and punctuation.
assembler A program that translates instructions in the CPU's repertoire into bits for loading directly into the memory of a computer; *assembly language* is the corresponding level of programming language.
bandwidth How fast a communications path can carry information, for instance 56 kilobits per second for a telephone modem or 100 Mbps for Ethernet.
base station Radio equipment that connects wireless devices (cell phones, computers) to a network (telephone network, computer network).
binary Having only two states or possible values; also *binary number* for numbers in base 2.
binary search An algorithm that searches a sorted list, repeatedly dividing the part to be searched into two equal halves.
bit A binary digit (0 or 1) that represents the amount of information in a binary choice like on or off.
BitTorrent Peer-to-peer protocol for distributing large popular files efficiently; downloaders also upload.
Bluetooth Short-range low-power radio for hands-free cell phones, games, etc.
bot, botnet A computer running a malicious program under the control of a bad guy; a botnet is a collection of bots under a common control.
browser A program like Internet Explorer, Firefox, or Safari that provides the primary interface to web services for most people.
bug An error in a program.
bus A set of wires used to connect electronic devices; also see USB.
byte Eight bits, enough to store a letter, a small number, or part of a larger quantity; treated as a unit in modern computers.
cable modem A device for sending digital data over a cable television network.
cache Local storage that provides fast access to information that has been used recently.
certificate A piece of cryptographic data that can be used to verify the authenticity of a web site.
chip Small electronic circuit, manufactured on a flat silicon surface and mounted in a ceramic package; also integrated circuit, microchip.
Chrome A browser from Google; Chrome OS is an operating system that supports only a browser.

client A program, often a browser, that makes requests of a server, as in client-server.
cloud computing Computing performed on a server, with data stored on a server, replacing a desktop application; mail, calendar and photo sharing sites are examples.
code Text of a program in a programming language, as in source code; an encoding, as in ASCII.
compiler A program that translates programs written in a high-level language like C or Fortran into a lower-level form like assembly language.
complexity A measure of the difficulty of a computational task or algorithm, expressed in terms of how long it takes to process N data items, like N or log N.
compression Squeezing a digital representation into fewer bits, as in MP3 compression for digital music.
cookie Text sent by a server, stored by the browser on your computer, and then returned by your browser on your next access to that server; widely used for tracking visits to web sites.
CPU Central Processing Unit; see processor.
declaration A programming language construct that states the name and properties of some part of a computer program, for example a variable that will store information during a computation.
DES Data Encryption Standard, the first widely-used digital encryption algorithm; superseded by AES.
digital Representation of information that takes on only discrete numeric values; contrast to analog.
directory Same as folder.
DMCA Digital Millennium Copyright Act (1998), the US law that protects copyrighted digital material.
DNS Domain Name System, the Internet service that translates domain names into IP addresses.
domain name A hierarchical naming scheme for computers connected to the Internet, as in www.msnbc.com.
driver Software that controls a particular hardware device like a printer; usually loaded into the operating system as necessary.
DSL Digital Subscriber Loop, a technique of sending digital data over telephone lines. Comparable to cable.
Ethernet The most common local area network technology, used in most home and office wireless networks.
EULA End User License Agreement, the long legal document in tiny print that restricts what you can do with software and other digital information.
exponential Growing by a fixed proportion each time period, for example, doubling every year or (equivalently) growing by 6 percent a month; often used carelessly for "growing quickly."
fiber, optical fiber Fine strand of extremely pure glass used to carry light signals over long distances; the signals encode digital information. Most long-distance digital traffic is carried on fiber optic cables.
file system The part of an operating system that organizes and accesses information on disks and other storage media.
firewall A program that controls or blocks incoming and outgoing network connections from a computer.
Flash Adobe software system for displaying video and animation on web pages.
flash memory Integrated-circuit memory technology that preserves data without consuming power; widely used in cameras, cell phones, USB memory sticks and as a replacement for disk drives.
FM Frequency modulation, a technique for sending information by changing the frequency of a radio signal; usually seen in the context of FM radio.
folder A file that holds information about files and folders, including size, date, permissions, and location.
function Component of a program that performs a specific focused computational task, for instance computing a square root or popping up a dialog box, like the `prompt` function in JavaScript.

gateway A computer that connects one network to another; often called a router.
GNU GPL GNU General Public License, a copyright license that protects open-source code by requiring free access to source code, thus preventing it from being taken private.
GPS Global Positioning System; uses time signals from satellites to compute position on the surface of the earth. It's one-way; GPS devices like car navigators do not broadcast to the satellites.
GSM Global System for Mobile Communications, a cell phone system used in about 80 percent of the world; one of two systems used in the USA.
hard disk Device that stores data on rotating disks of magnetic material; also hard drive.
hexadecimal Base 16 notation, most often seen in Unicode tables, URLs, color specifications.
HTML Hypertext Markup Language; used to describe the contents and format of a web page.
IC Integrated Circuit; an electronic circuit component fabricated on a flat surface, mounted in a sealed package, and connected to other devices in a circuit. Most digital devices are made up mostly of IC's.
ICANN Internet Corporation for Assigned Names and Numbers, the organization that allocates Internet resources that must be unique, like domain names and protocol numbers.
intellectual property The product of creative or inventive acts, protectable by copyright and patents; it includes software and digital media.
interface Vague general term for the boundary between two independent entities. See API for programming interfaces. Another use is *user interface*, the part of a computer program that a human interacts with directly.
interpreter Program that interprets instructions for a computer, whether real or not, thus simulating its behavior; JavaScript programs in a browser are processed by an interpreter.
IP Internet Protocol, the fundamental protocol for sending packets through the Internet.
IP address Internet Protocol address, the unique numeric address currently associated with a computer on the Internet; loosely analogous to a telephone number.
ISP Internet Service Provider, an entity that provides connections to the Internet; examples include universities, and cable and telephone companies.
JavaScript A programming language used on web pages for visual effects; widely used for tracking as well.
JPEG Joint Photographic Experts Group, a standard compression algorithm and representation for digital images.
kernel The central part of an operating system, responsible for controlling operation and resources.
library A collection of related software components in a form that can be used as parts of a program, for example the standard functions that JavaScript provides for accessing the browser.
Linux An open-source Unix-like operating system, widely used on servers.
logarithm Given a number N, the power to which the base is raised to produce N. In this book, the base is 2 and the logs are integers.
loop A part of a program that repeats a sequence of instructions; an infinite loop repeats them a lot of times.
malware Software with malicious properties and intent.
microchip Another word for chip or integrated circuit.
modem Modulator / demodulator, a device that converts bits into an analog representation like sound and back.
MP3 A compression algorithm and representation for digital audio, part of the MPEG standard for video.
MPEG Moving Picture Experts Group, a standard compression algorithm and representation for digital video.
net neutrality The general principle that Internet service providers should treat all traffic the same way (except perhaps in cases of overload), rather than biasing treatment for economic or other non-technical reasons.

object code Instructions and data in binary form that can be loaded into RAM; the result of compilation and assembly. Contrast with source code.
open source Source code (written in a high-level language) that is freely available, usually under a license like the GNU GPL that keeps it available.
operating system Program that controls the resources of a computer, including CPU, file system, devices, and external connections; examples include Windows, Mac OS X, Unix, Linux.
packet A collection of information in a specified format, such as an IP packet; loosely analogous to a standard shipping container.
PDF Portable Document Format, a standard representation for documents, originally created by Adobe.
peer-to-peer Exchange of information between peers, that is, a symmetric relationship, in contrast to client-server. Most often used for file-sharing networks.
peripheral Hardware device connected to a computer, like a disk, printer, scanner, etc.
phishing, spear phishing Attempt, usually by email, to obtain personal information by pretending to have some kind of relationship with the victim; *spear phishing* is more precisely targeted.
pixel Picture element; a single point in a digital image.
platform Vague term for a software system, like an operating system, that provides services that can be built upon.
plug-in A program that runs in the context of a browser; Flash, Quicktime and Silverlight are common examples.
processor The part of the computer that does arithmetic and logic and controls the rest of the computer; also CPU. Intel and AMD processors are widely used in laptops; ARM processors are used in most cell phones.
program A set of instructions that causes a computer to do a task; written in a programming language.
programming language Notation for expressing sequences of operations for a computer, translated ultimately into bits to be loaded into RAM; examples include assembler, C, C++, Java, JavaScript.
protocol An agreement on how systems interact; most often seen in the Internet, which has a large number of protocols for exchanging information over networks.
quadratic Numeric growth proportional to the square of something, for instance how the running time of selection sort varies with the number of items to be sorted.
RAM Random Access Memory; the primary memory in a computer.
registrar A company that has the authority (from ICANN) to sell domain names to individuals and companies; GoDaddy is one of the most visible.
representation General word for how information is expressed in digital form.
RFID Radio-Frequency Identification, a very low power wireless system used in electronic door locks, pet identification chips, and the like.
RGB Red, Green, Blue, the standard way that colors are represented in computer displays as the combination of three basic colors.
router Another word for gateway: a computer that passes information from one network to another.
RSA The most widely-used public-key encryption algorithm.
SDK Software Development Kit, a collection of tools to help programmers write programs for some device or environment, for example cell phones and game consoles.
search engine Server like Bing or Google that collects web pages and answers queries about them.
server Computer or computers that provide access to data upon request from a client; search engines, shopping sites, and social networks are examples.
simulator Program that simulates (acts like) a device or other system.
smartphone Cell phone like iPhone and Android with the capability of downloading programs (apps).
social engineering Technique of deceiving a victim into releasing information by pretending to have a relationship like a mutual friend or the same employer.

solid state disk Storage that uses flash memory; a replacement for disk drives.
source code Program text written in a language comprehensible by programmers, to be compiled into object code.
spectrum Frequency range for a system or device, for example cell phone service or a radio station.
spyware Software that reports back to home on what happens on the computer where it is installed.
system call Mechanism by which an operating system makes its services available to programmers; a system call looks much like a function call.
standard Formal specification or description of how something works or is built or is controlled, precise enough to permit interoperability and independent implementations. Examples include character sets like ASCII and Unicode, plugs and sockets like USB, and programming language definitions.
TCP Transmission Control Protocol, protocol that uses IP to create two-way streams. TCP/IP is the combination of TCP and IP.
tracking Recording the sites that a web user visits and what he or she does there.
Trojan horse A program that promises to do one thing but in fact does something different and usually malicious.
Turing machine Abstract computer, conceived by Alan Turing, that is capable of performing any digital computation; a universal Turing machine can simulate any other Turing machine and thus any digital computer.
Unicode A standard encoding for all of the characters in all of the world's writing systems.
Unix An operating system developed at Bell Labs that forms the base of many of today's operating systems; Linux is a lookalike that provides the same services but with a different implementation.
URL Uniform Resource Locator, the standard form of a web address, like http://www.amazon.com.
USB Universal Serial Bus, a standard connector for plugging devices like external disk drives, cameras, displays and phones into computers.
variable A RAM location that stores information; a variable declaration names the variable and may provide other information about it, like an initial value or the type of data it holds.
virtual machine A program that simulates a computer; also interpreter.
virtual memory Software and hardware that give the illusion of unlimited RAM.
virus A program, usually malicious, that infects a computer; a virus needs help propagating from one system to another, in contrast to a worm.
VoIP Voice over IP, a method of using the Internet for voice conversations, often with a way to access the regular telephone system. Skype is a widely used example.
web beacon A small and usually invisible image used for tracking the fact that a particular web page has been downloaded.
web server A server focused on web applications.
Wi-Fi Wireless Fidelity, a marketing name for 802.11 wireless.
wireless router Radio device that connects wireless computers to a wired network, most often Ethernet.
worm A program, usually malicious, that infects a computer; a worm can propagate from one system to another without help, in contrast to a virus.

Index

1-click patent, Amazon, 79
3COM, 128
3G cell phone, 155, 214
64-bit memory address, 33
802.11 wireless standard, 129, 134, 155, 214

accumulator, 36, 41, 43
acknowledgment, TCP, 146
active content, 167
ActiveX, 169, 178
Activity Monitor, Mac OS X, 87
Ad Exchange, Google, 193
Ada Lovelace, 8
Ada programming language, 8
add-on, Adblock, 178, 214
add-on, Firefox, 100
add-on, Firesheep, 127, 134, 176
add-on, Flashblock, 178
add-on, Ghostery, 168, 178, 193
add-on, NoScript, 168, 178, 214
add-on, TACO, 194
address, Ethernet, 127
address, IP, 137, 140, 216
address, IPv4, 140
address, IPv6, 138, 140
address translation (NAT), network, 140
Adleman, Len, 182
Adobe, 104, 168, 178, 194
ADSL, *see* DSL
adware, 152
AdWords, 191
AES encryption standard, 180, 214
Aiken, Howard, 69
airplanes, computers in, 52
Akamai, 155
Alcatel, 120
`alert` function, 105–106, 112
algorithm, 115, 214
algorithm, approximate, 62
algorithm, ATM checksum, 158
algorithm, binary search, 56, 214
algorithm complexity, 60, 115, 215
algorithm, definition of, 53
algorithm, divide and conquer, 56, 59, 95, 141
algorithm, exponential, 60, 182, 215
algorithm, ISBN checksum, 158
algorithm, linear, 55
algorithm, MD5 message digest, 184
algorithm, N log N, 60
algorithm, PageRank, 190
algorithm, quadratic, 58, 217
algorithm, Quicksort, 58, 79
algorithm, RSA public-key, 182, 217
algorithm, selection sort, 57
algorithm, SHA-1, SHA-3 message digest, 184
algorithm, tax form as, 53
algorithm vs program, 65
Allen, Paul, 70
allocation, spectrum, 129, 134
Altair computer, 70
Alto computer, 126
Amazon, 80, 87, 104, 139, 147, 166, 181, 184, 203
Amazon 1-click patent, 79
Amazon EC2, 203
amplitude modulation (AM), 129, 214
analog hole, 78
analog vs digital, 21
analog watch, 22
analog-digital conversion, 23
Android Market, 2
Android operating system, 83
Android phone, 4, 86
Android SDK, 75
Anna Kournikova virus, 170–171
anonymizing data, 197
Antikythera mechanism, 7
anti-virus software, 178
AOL, 124, 203
AOL query logs, 197
Apache web server, 83
app, 85
App Store, *see* iTunes Store
Apple, 2, 9, 75, 79, 85–86, 101, 153, 168
Apple Boot Camp, 89
application, app, 85, 214
application, `date`, 98
Application Programming Interface (API), 74, 80, 104, 111, 199, 214
application-level protocol, 144
approximate algorithm, 62
architecture, von Neumann, 11
arithmetic, binary, 32
ARM processor, 42, 66
ARPANET, 136
ASCII characters, table of, 28
ASCII standard, 27, 34, 81, 150, 156, 214
ASCII with parity, table of, 158
assembler, 66, 214
assembly programming language, 66
assignment statement, 107–108
ATM checksum algorithm, 158
AT&T, 120, 131–132, 204
attack, brute force, 177, 180
attack, dictionary, 170, 180
attack (DoS), denial of service, 176
attack, man in the middle, 176, 179
attack, social engineering, 174, 217
attack, SQL injection, 175
authentication, password, 148–149, 154, 165, 175–177, 181, 184
authentication, two-factor, 177–178
Authenticode, 169
autonomous system, 142
autorun, 171, 175

Babbage, Charles, 7
Backus, John, 69
bandwidth, 123–124, 155, 214
barcode, 4, 15
Bardeen, John, 16
base 2 logarithm, 56
base station, cell phone, 131
base station, wireless, 128, 131, 214
BASIC programming language, 69
Bell, Alexander Graham, 119
Bell Labs, 70, 86, 120
Berners-Lee, Tim, 161–162
best effort, 144, 155
Betamax vs VHS, 81
Bezos, Jeff, 79
bigit, 28
binary addition table, 32
binary arithmetic, 32
binary digit, 28
binary multiplication table, 32
binary numbers, 31, 214
binary search algorithm, 56, 214
binary to decimal conversion, 31
Bing search engine, 63, 163
bit, origin of, 28
bit rot, 77
bits, meaning of, 29, 31, 214
BitTorrent protocol, 152, 214

block, disk, 94
blog, 3
blue screen of death, 88
Bluetooth, 15, 93, 130, 214
Blu-ray vs HD-DVD, 81
Boggs, David, 126
Boot Camp, Apple, 89
booting, 90
bot, botnet, 175, 214
branch instruction, 38
branch instruction, conditional, 39
Brattain, Walter, 16
break-in, Epsilon, 176
break-in, RSA, 177
break-in, SecurID, 177
break-in, Sony PlayStation, 176
Brin, Sergey, 188
broadcast network, 123, 159
browser, Chrome, 71, 83, 99, 106, 162, 214
browser extension, *see* add-on, plug-in
browser, Firefox, 71, 77, 83, 99, 106, 162
browser, Internet Explorer, 99, 102, 106, 162, 168–169
browser, Mosaic, 162
browser, Netscape Navigator, 102, 162, 167
browser, Opera, 99, 106
browser, Safari, 99, 106, 162
brute force attack, 177, 180
Buchholz, Werner, 32
bug, 75, 110, 113, 116, 170, 214
Burks, Arthur, 11
bus, 14, 214
byte, 12, 214
byte, origin of, 32
`bzip2` compression, 156

C programming language, 70–72, 81, 99, 168
C++ programming language, 70–72, 81, 99
cable modem, 125, 128, 214
cable television, 124
cache, 42–43, 141, 214
Caesar cipher, 178
CAPTCHA, 45
cars, computers in, 51
CDMA, 132
CD-ROM, 27, 34, 97
cell phone, 3G, 155, 214
cell phone base station, 131
cell phone cells, diagram of, 131
cell phone frequency, 132
central processing unit (CPU), 11, 215, 217
Cerf, Vint, 146
CERN, 161
certificate, 183, 214
certificate authority, 183
CGI, 165
Chappe, Claude, 118
checksum algorithm, ATM, 158
checksum algorithm, ISBN, 158
chip, 17, 214
Chrome browser, 71, 83, 99, 106, 162, 214
Chrome OS operating system, 100
Chromebook, 100
circuit board, 15
Clarke, Arthur, 1
Clay Mathematics Institute, 62
clean room development, 79
client-server, 151, 161, 165, 215, 217
cloud computing, 200, 215
coaxial cable, 128
COBOL programming language, 69
Cocks, Clifford, 181
Cohen, Bram, 152
color representation, 24, 33, 217
compact disk, 26
compilation process, diagram of, 68
compiler, 67, 112, 215
complexity, algorithm, 60, 115, 215
components, software, 74
compression, `bzip2`, 156
compression, data, 132, 155, 202, 215
compression, GIF, 165
compression, Huffman code, 156
compression, lossless, 156
compression, lossy, 156
compression, Zip, 156
computer, Altair, 70
computer, Alto, 126
computer architecture, 41
computer architecture, diagram of, 11
computer chess, 48
computer, EDSAC, 8, 66
computer, ENIAC, 8, 16
Computer Fraud and Abuse Act, 170
computer, IBM PC, 9, 19
computer, stored-program, 12, 45
computer, Toy, 36
computers in airplanes, 52
computers in cars, 51
computers in medical imaging, 52
computers in voting machines, 52
concatenation, text, 107
conditional branch instruction, 39
connector, RJ45, 128
Consumer Reports, 4, 52
Content Scrambling System (CSS), 180
conversion, analog-digital, 23
conversion, binary to decimal, 31
conversion, decimal to binary, 31
Cook, Steve, 62
cookie, 166, 215
cookie, Flash, 194
cookie, third-party, 172, 178, 193
cookies, tracking, 192
cooking robot, 65
copyright, 78, 151–152
copyright lifetime, 78, 208
Core Duo processor, 11, 42
Count of Monte Cristo, 118
crawl, crawler, 188
cryptographic hash, *see* message digest
cryptographic key, 179
cryptography, 61, 63, 178
`ctof` function, 76
CTSS operating system, 86
cyber warfare, 52, 207

data compression, 132, 155, 202, 215
Data General Corp, 86
`date` application, 98
DCHP protocol, 138
DDoS, *see* denial of service attack
debugging, 75, 77, 110
decimal to binary conversion, 31
declaration, variable, 107–108, 215, 218
deep packet inspection, 195
definition of algorithm, 53
deidentification, 197
delay, 123
denial of service attack (DoS), 176
Dept of Justice vs Microsoft, 102, 162
DES encryption standard, 180, 215
device driver, 91, 215
diagram of cell phone cells, 131
diagram of compilation process, 68
diagram of computer architecture, 11
diagram of dotted decimal notation, 138
diagram of Ethernet packet, 128
diagram of file system organization, 94
diagram of IPv4 packet, 145
diagram of operating system and libraries, 92
diagram of program in memory, 41
diagram of protocol layers, 144
diagram of search engine, 189
diagram of sound wave, 25
diagram of TCP segment, 146
diagram of TCP/IP connection, 147
diagram of virtual operating system, 89
dictionary attack, 170, 180
Diffie, Whitfield, 181
digit, binary, 28
Digital Equipment Corp (DEC), 86
digital rights management (DRM), 78
digital signature, 169, 182
digital television, 134
digital vs analog, 21
digital watch, 22
directory, *see* folder
disk block, 94
disk, compact, 26
disk, floppy, 14, 19, 97
disk, solid state, 14, 97, 218
distributed computing, 45
divide and conquer algorithm, 56, 59, 95, 141
division by zero, 55, 110
DMCA, 153, 215
DMCA safe harbor, 153
domain name, 138, 163, 215
domain name, internationalized, 140, 163
domain name registrar, 139, 141, 217
Domain Name System (DNS), 138–139, 204, 215
domain name, Unicode, 140, 163
domain seizure, 153
domain, top-level, 139
DoS, *see* denial of service attack
dotted decimal notation, 137
dotted decimal notation, diagram of, 138
DoubleClick, 193
DSL, 125, 137, 155, 215
DSL modem, 128
Dumas, Alexandre, 119
dumb network, 154
DVD, 26, 34, 97
DVD Jon, 180
Dynamic Host Configuration Protocol (DHCP), 138

eavesdropping, 159, 181
e-book reader, Kindle, 80, 87, 93
e-book reader, Nook, 87
EC2, Amazon, 203
Eckert, Presper, 8
Edelcrantz, Abraham, 118
Edison, Thomas, 24, 75
EDSAC computer, 8, 66
Eich, Brendan, 72
electrical telegraph, 119
Ellis, James, 181
email privacy, 202
embedded system, 51, 71
encryption, 78, 125, 127
encryption standard, AES, 180, 214
encryption standard, DES, 180, 215
encryption, WEP wireless, 134
encryption, WPA wireless, 134
end-to-end principle, 154
English wine measures, 34
ENIAC computer, 8, 16
Enigma, 45, 179
Epsilon break-in, 176
error correction, 63, 98, 132, 157
error detection, 157
error, semantic, 68
error, syntax, 68
Ethernet, 126, 215
Ethernet address, 127
Ethernet packet, 137
Ethernet packet, diagram of, 128
Ethernet range, 128
EULA, 79, 215
European Union, 195, 202

evercookie, 195
exponential algorithm, 60, 182, 215

Facebook, 1–3, 104, 127, 139, 198–199
factoring, 182
factoring challenge, RSA, 183
Fanning, Shawn, 151
FAT file system, 97
Federal Communications Commission (FCC), 129
Federal Election Commission (FEC), 196
fetch-decode-execute cycle, 36, 41
fiber, optical, 18, 120, 123, 126, 135, 155, 215
file sharing, 151
file system, 14, 93, 215
file system, FAT, 97
file system, network, 98
file system organization, diagram of, 94
file system, RAID, 98
file system, SD card, 97
file system, USB, 97
Firefox add-on, 100
Firefox browser, 71, 77, 83, 99, 106, 162
Firesheep add-on, 127, 134, 176
firewall, 178, 184, 215
firkin, 34
first sale doctrine, 80
Flash cookie, 194
flash memory, 14, 97, 215
Flash plug-in, 100, 168, 194, 215, 217
Flashblock add-on, 178
Flickr, 2, 199–200
floppy disk, 14, 19, 97
floppy disk virus, 170
folder, 93, 215
folder contents, 94
folder, trash, 96
form, HTML, 165
FORTRAN programming language, 69
Foursquare, 199
Frankenpine, 132
Free Software Foundation, 82
frequency, cell phone, 132
frequency modulation (FM), 129, 215
`ftoc` function, 76
FTP protocol, 148, 163
function, `alert`, 105–106, 112
function call, 74, 215
function, `ctof`, 76
function, `ftoc`, 76
function, `parseInt`, 109
function, programming language, 74
function, `prompt`, 107, 112–113

Gates, Bill, 69
gateway, 137–138, 142–145, 147, 159, 195, 216
geo-tagging, 192
Ghostery add-on, 168, 178, 193
GIF compression, 165
giga, 12, 30, 213
Gmail, 150, 199–200, 202
GNU General Public License (GPL), 82, 216
GNU project, 82
Goldstine, Herman, 11
Google, 44, 75, 83, 100, 120, 142, 153, 188–189, 191, 195, 200
Google Ad Exchange, 193
Google Docs, 104, 201–202
Google Maps, 111, 116
Google search engine, 63
Google speak2tweet, 1
Google Street View, 196
Google Translate, 48
Gosling, James, 71
`GOTO` instruction, 38, 109
GPS, 4, 130, 192, 216
gray scale, 33
Great Firewall of China, 184
Grokster, 152
GSM, 132, 216

hard disk, hard drive, 13, 216
Harvard Mark I, II, 69, 75
HD-DVD vs Blu-ray, 81
Hellman, Martin, 181
Herodotus, 117
hertz, 12, 129
hexadecimal, 33, 127, 137, 163, 216
high-level programming language, 67
Hoare, Tony, 58
Hollerith, Herman, 8
Hopper, Grace, 69, 75
host, 142
HTML, 105, 161, 164, 216
HTML form, 165
HTML tag, 164
HTML5, 168
HTTP protocol, 147, 161
HTTPS protocol, 163, 176–177, 182
Huffman code compression, 156
hypertext, 162

IBM, 32, 48, 69, 83, 86, 101, 139, 158, 180
IBM PC computer, 9, 19
ICANN, 139, 142, 163, 216
IEEE, 129
`ifconfig`, `ipconfig`, 127
`if-else` statement, 109
`IFZERO` instruction, 39, 109
IMAP protocol, 150
`<img>` tag, 164, 193
infrared, 129
instruction, branch, 38
instruction, conditional branch, 39
instruction, `GOTO`, 38, 109
instruction, `IFZERO`, 39, 109
instruction, jump, 38
instruction repertoire, 36, 41
instructions, numeric values of, 40
integrated circuit (IC), 8, 17, 216
Intel, 11, 18, 42, 66, 101
intellectual property, 78, 216
interface, 116, 216
International System of Units, 213
International Telecommunications Union (ITU), 129, 136
internationalized domain name, 140, 163
Internet Engineering Task Force (IETF), 139
Internet Explorer browser, 99, 102, 106, 162, 168–169
Internet Protocol (IP), 144, 216
Internet Service Provider (ISP), 137, 142, 153, 155, 195, 203, 216
Internet time, 151
interpreter, 116, 216
iOS operating system, 86
IP address, 137, 140, 216
IP packet, 137
iPad, 15
iPad SDK, 75
iPhone, 4, 85–86, 133
iPhone SDK, 75
IPv4 address, 140
IPv4 packet, diagram of, 145
IPv4 protocol, 137
IPv6 address, 138, 140
IPv6 protocol, 137
ISBN checksum algorithm, 158
Italy, `.it`, 140
iTunes Store, 2–3, 12, 79–80, 85, 153

Jacquard loom, 7
Java programming language, 71, 167
JavaScript programming language, 72, 103, 168, 201, 216
JavaScript tracking, 194
Jeopardy, 48
jitter, 123
Johansen, Jon Lech, 180
Joy of Cooking, 53
JPEG, 27, 63, 156–157, 165, 216
jump instruction, 38
jurisdiction, 184, 208

Kahn, Robert, 146
Kazaa, 152
kernel, 83, 101, 216
key, cryptographic, 179
key distribution, 180–181
key logger, 175
Kilby, Jack, 18
kilderkin, 34
kilo, 19, 213
Kindle e-book reader, 80, 87, 93
Knuth, Donald, 34

language translation, natural, 1, 48
laser, 18, 26, 79, 126
latency, 123
layers, software, 144
Leitner, Henry, 57
Lewis, Harry, 57
Lexus software, 52
library, program, 74, 111, 116, 216
license, virtual machine, 90
licenses, 79
LimeWire, 152
linear algorithm, 55
Linux operating system, 83, 89, 216
location privacy, 199
logarithm, 7, 56
logarithm, base 2, 56
logic gate, 16
long-playing record (LP), 3, 24, 27
loop, 39, 41, 43, 145, 216
loop, `while`, 109
lossless compression, 156
lossy compression, 156
Lovelace, Ada, 8
Lucent, 120
Luhn, Peter, 158

Mac OS X Activity Monitor, 87
Mac OS X operating system, 9, 86
Mac, Windows on a, 89
malware, 171–172, 216
man in the middle attack, 176, 179
Mars Climate Orbiter, 73
Mary, Queen of Scots, 179
Mauchly, John, 8
MD5 message digest algorithm, 184
meaning of bits, 29, 31, 214
medical imaging, computers in, 52
mega, 12, 213
megapixel, 24
memory address, 64-bit, 33
memory, flash, 14, 97, 215
memory (RAM), random access, 12, 217
memory, virtual, 88, 116, 218
message digest algorithm, MD5, 184
message digest algorithm, SHA-1, SHA-3, 184
Metcalfe, Robert, 126, 128
micro, 18, 213
microchip, 17, 216
Microsoft, 9, 70, 75, 86, 90, 97, 103, 162, 197, 201
Microsoft antitrust case, 102, 162
Microsoft security laws, 167
milli, 18, 213
MIME, 150
modem, 124, 155, 216
modem, cable, 125, 128, 214
modem, DSL, 128
modulation, 124, 126, 129
Moldova, `.md`, 140
Moore, Gordon, 18

Moore's Law, 18, 47
Morris, Robert T., 170
Morris worm, 170
Morse code, 156
Morse, Samuel, 119
Mosaic browser, 162
MP3, 27, 63, 151, 157, 216
MPAA, 153
MPEG, 27, 63, 157, 216
MS-DOS operating system, 97
Multics operating system, 86
Muybridge, Eadweard, 27

N log N algorithm, 60
name server, 142
name server, root, 141
nano, 18, 42, 213
Napier, John, 7
Napster, 3, 151
National Institute of Standards and Technology (NIST), 180, 184
National Security Agency (NSA), 180
natural language translation, 1, 48
negative acknowledgment, 146
net neutrality, 121, 216
Netscape, 72, 166
Netscape Navigator browser, 102, 162, 167
network address translation (NAT), 140
network effect, 10, 208
network file system, 98
New York Times, 130, 166
Nigerian scam letter, 173
Nobel Prize, 16, 18, 46
Nook e-book reader, 87
NoScript add-on, 168, 178, 214
Notepad, 107
Noyce, Robert, 18
`nslookup`, 141
numbers, binary, 31, 214
numeric values of instructions, 40

object code, 82, 168, 217
Objective-C programming language, 71
open relay, 154
open source, 82, 217
Opera browser, 99, 106
operating system and libraries, diagram of, 92
operating system, Android, 83
operating system, Chrome OS, 100
operating system, CTSS, 86
operating system, diagram of virtual, 89
operating system, iOS, 86
operating system, Linux, 83, 89, 216
operating system, Mac OS X, 9, 86
operating system, MS-DOS, 97
operating system, Multics, 86
operating system, Unix, 86–87, 89, 99, 218
operating system, virtual, 89
operating system, Windows, 9, 86, 89–90, 92, 101, 169–172, 178, 201
optical fiber, 18, 120, 123, 126, 135, 155, 215
optical telegraph, 118
origin of *bit*, 28
origin of *byte*, 32
Orwell, George, *1984*, 5, 80

packet, 127, 137, 217
packet, Ethernet, 137
packet, IP, 137
Page, Larry, 188
PageRank algorithm, 190
Parallels virtual operating system, 89
PARC, Xerox, 126
parity code, 158
`parseInt` function, 109
password authentication, 148–149, 154, 165, 175–177, 181, 184
patent, 79, 136, 208
patent, Amazon 1-click, 79
Patriot Act, 202
PayPal, 204
PC, *see* IBM PC computer
peering point, 143
peer-to-peer, 151, 217
perceptual coding, 156
personally identifiable information, 195, 199
peta, 30, 213
Petzold, Charles, 16
phishing, 173, 217
phishing, spear, 174, 217
phonograph, 24–25
pipelining, 42
pixel, 24, 169, 217
plain text, 107, 164
platform, 91, 100, 217
PlayStation break-in, Sony, 176
Please Rob Me, 199
plug-in, Flash, 100, 168, 194, 215, 217
plug-in, Quicktime, 168
plug-in, Silverlight, 100, 168
P=NP problem, 62
polynomial, 61
Pony Express, 118
POP protocol, 150
popup blocker, 168
port, 146
Portable Document Format (PDF), 100, 104, 169, 190, 217
PowerPC processor, 101
powers of 10, table of, 30
powers of 2 and 10, table of, 30, 213
powers of 2, table of, 29
Pride and Prejudice, 14, 156
privacy, email, 202
processor, ARM, 42, 66
processor, Core Duo, 11, 42
processor family, x86, 101
processor, PowerPC, 101
program in memory, diagram of, 41
program library, 74, 111, 116, 216
program testing, 40, 55, 76, 110
program vs algorithm, 65
programming language, Ada, 8
programming language, assembly, 66
programming language, BASIC, 69
programming language, C, 70–72, 81, 99, 168
programming language, C++, 70–72, 81, 99
programming language, COBOL, 69
programming language, FORTRAN, 69
programming language function, 74
programming language, high-level, 67
programming language, Java, 71, 167
programming language, JavaScript, 72, 103, 168, 201, 216
programming language, Objective-C, 71
programming language, Visual Basic, 70, 103
programming style, 71
promiscuous mode, 127
`prompt` function, 107, 112–113
property, intellectual, 78, 216
protocol, 118, 135, 144, 217
protocol, application-level, 144
protocol, BitTorrent, 152, 214
protocol, DCHP, 138
protocol, FTP, 148, 163
protocol, HTTP, 147, 161
protocol, HTTPS, 163, 176–177, 182
protocol, IMAP, 150
protocol, IPv4, 137
protocol, IPv6, 137
protocol layers, diagram of, 144
protocol, POP, 150
protocol, SMTP, 149–150
public records, 196
public-key algorithm, RSA, 182, 217
public-key cryptography, 181

quadratic, 129
quadratic algorithm, 58, 217
query logs, AOL, 197
Quicksort algorithm, 58, 79
Quicktime plug-in, 168

radio-frequency identification (RFID), 130, 217
RAID file system, 98
random access memory (RAM), 12, 217
random vs sequential access, 12
range, Ethernet, 128
range, wireless, 129
recycle bin, 96
Red Hat, 83
registrar, domain name, 139, 141, 217
repertoire, instruction, 36, 41
Request for Comments (RFC), 139
reverse DNS lookup, 140
reverse Turing test, 46
RIAA, 153
Right Media, Yahoo, 193
Ritchie, Dennis, 70, 86, 89
Rivest, Ron, 182, 184
RJ45 connector, 128
root name server, 141
`rot13`, 178
rotary dial phone, 119
router, *see* gateway
router, wireless, 140, 218
routing, 138, 142
Rowe, Mike, 142
RSA break-in, 177
RSA factoring challenge, 183
RSA public-key algorithm, 182, 217

Safari browser, 99, 106, 162
safe harbor, DMCA, 153
script kiddy, 171
`<script>` tag, 105, 112
SD card file system, 97
search algorithm, binary, 56, 214
search engine, Bing, 63, 163
search engine, diagram of, 189
search engine, Google, 63
secondary storage, 13
secret-key cryptography, 180
Secure Digital, *see* SD
secure erase, 96
SecurID break-in, 177
security by obscurity, 179
security laws, Microsoft, 167
selection sort algorithm, 57
semantic error, 68
semantics, 104
sequential vs random access, 12
SHA-1, SHA-3 message digest algorithm, 184
Shamir, Adi, 182
Shockley, William, 16
Silicon Valley, 17
Silverlight plug-in, 100, 168
simulator, Toy computer, 37, 41, 112
Skype, 120
smart quotes, 106
smartphone, 192, 217
SMTP protocol, 149–150
sniffing, 127, 176
social engineering attack, 174, 217
social graph, 199
software components, 74
Software Development Kit (SDK), 75, 80, 217
software layers, 144
software, Lexus, 52
solid state disk, 14, 97, 218

Sony PlayStation break-in, 176
Sony spyware, 175
sort times, table of, 60
sound wave, diagram of, 25
source code, 82, 168, 215, 218
Southwest Bell, 120
spam, 172
speak2tweet, Google, 1
spear phishing, 174, 217
spectrum allocation, 129, 134
speed of light, 155
speedometer, 22
Sprint, 132
spyware, Sony, 175
SQL injection attack, 175
Stallman, Richard, 82
standard, 802.11 wireless, 129, 134, 155, 214
standard, AES encryption, 180, 214
standard, ASCII, 27, 34, 81, 150, 156, 214
standard, DES encryption, 180, 215
standard, Unicode, 28, 32, 81, 218
standards, 218
statement, assignment, 107–108
statement, `if-else`, 109
Stonehenge, 7
stored-program computer, 12, 45
Street View, Google, 196
Stroustrup, Bjarne, 70
Stuxnet worm, 52, 171
Sun Microsystems, 71, 167
supercomputer, 44
swapping, 88
Sweeney, Latanya, 198
syntax, 104
syntax error, 68
system call, 91, 218
system, embedded, 51, 71

table, binary addition, 32
table, binary multiplication, 32
table of ASCII characters, 28
table of ASCII with parity, 158
table of powers of 10, 30
table of powers of 2, 29
table of powers of 2 and 10, 30, 213
table of sort times, 60
table of units, 213
tablet, 9, 15
TACO add-on, 194
tag, HTML, 164
tag, `<img>`, 164, 193
tag, `<script>`, 105, 112
Tanenbaum, Andrew, 81
Task Manager, Windows, 87
tax form as algorithm, 53
TCP acknowledgment, 146
TCP port, 146, 162
TCP segment, diagram of, 146
TCP, Transmission Control Protocol, 144, 218
TCP/IP, 144
TCP/IP connection, diagram of, 147
television, digital, 134
Teller, Edward, 28
Telnet, 147–148
tera, 14, 213
testing, program, 40, 55, 76, 110
text concatenation, 107
TextEdit, 107
texting, 133
third-party cookie, 172, 178, 193
Thompson, Ken, 86, 89
time to live (TTL), 145
TiVo, 87, 93
T-Mobile, 132
Tokyo in Unicode, 32–33
`top500.org`, 44
top-level domain, 139
Torvalds, Linus, 89
Toy computer, 36
Toy computer simulator, 37, 41, 112
Toyota Lexus software, 52
`traceroute`, 143, 173
tracking, 192, 218
tracking cookies, 192
tracking, JavaScript, 194
trade secret, 78
tradeoff, 41–42, 72, 132, 157, 201, 206
transistor, 8, 16, 79
translation, natural language, 1, 48
Transmission Control Protocol TCP, 144, 218
trash folder, 96
Traveling Salesman Problem, 61
Trojan, 171, 218
Tukey, John, 28
Turing, Alan, 45, 179
Turing Award, 46, 58, 62, 69, 86, 146, 182
Turing machine, 45, 47, 218
Turing test, 45, 48
Turing test, reverse, 46
Tuvalu, `.tv`, 139
Twitter, 1, 3, 104, 127, 189, 199, 204
two-factor authentication, 177–178

Unicode domain name, 140, 163
Unicode standard, 28, 32, 81, 218
Unicode, Tokyo in, 32–33
units, table of, 213
Univac, 69, 86
universal computer, 39
Universal Serial Bus (USB), 15, 218
universal Turing machine, 45
Unix operating system, 86–87, 89, 99, 218
URL, 161, 163, 218
URL encoding, 163
USB file system, 97
USB virus, 171

vacuum tube, 8, 16
variable declaration, 107–108, 215, 218
Verizon, 132, 203
VHS vs Betamax, 81
Viacom vs Google, YouTube, 153
virtual machine, 90, 100, 112, 116, 168, 218
virtual machine license, 90
virtual memory, 88, 116, 218
virtual operating system, 89
virtual operating system, diagram of, 89
virtual operating system, Parallels, 89
virtual operating system, VMware, 89
virtual operating system, Xen, 89
virtual private network (VPN), 177
virus, Anna Kournikova, 170–171
virus, floppy disk, 170
virus, USB, 171
virus, Visual Basic, 103, 170
virus vs worm, 170, 218
Visual Basic programming language, 70, 103
Visual Basic virus, 103, 170
VMware virtual operating system, 89
voice over IP (VoIP), 121, 144, 218
von Neumann architecture, 11
von Neumann, John, 11, 34–36, 42
voting machines, computers in, 52

wafer, 17
war driving, 134
Watson, 48
web beacon, 169, 193–194, 218
web server, Apache, 83
well known port, 146
WEP wireless encryption, 134
`while` loop, 109
White House, 111
Whorf, Benjamin, 73
Wi-Fi, 93, 129, 134, 218
Wikileaks, 97, 204
Windows on a Mac, 89
Windows operating system, 9, 86, 89–90, 92, 101, 169–172, 178, 201
Windows Task Manager, 87
wine measures, English, 34
Wintel, 101
wireless base station, 128, 131, 214
wireless encryption, WEP, 134
wireless encryption, WPA, 134
wireless range, 129
wireless router, 140, 218
wireless standard, 802.11, 129, 134, 155, 214
Wireshark, 127
World Wide Web Consortium (W3C), 162
worm, Morris, 170
worm, Stuxnet, 52, 171
worm vs virus, 170, 218
WPA wireless encryption, 134

x86 processor family, 101
Xen virtual operating system, 89
Xerox PARC, 126

Yahoo, 150, 155, 197, 199–200, 203
Yahoo Right Media, 193
Yamamoto, Isoroku, 179
YouTube, 1, 3, 153

zetta, 187, 213
Zillow, 196
Zip compression, 156
zombie, 175
Zuckerberg, Mark, 3