1

Introduction: What Are the Extraordinary Ideas Computers Use Every Day?

This is a gift that I have ... a foolish extravagant spirit, full of forms, figures, shapes, objects, ideas, apprehensions, motions, revolutions.

—William Shakespeare, *Love's Labour's Lost*

How were the great ideas of computer science born? Here's a selection:

- In the 1930s, before the first digital computer has even been built, a British genius founds the field of computer science, then goes on to prove that certain problems cannot be solved by any computer to be built in the future, no matter how fast, powerful, or cleverly designed.
- In 1948, a scientist working at a telephone company publishes a paper that founds the field of information theory. His work will allow computers to transmit a message with perfect accuracy even when most of the data is corrupted by interference.
- In 1956, a group of academics attend a conference at Dartmouth with the explicit and audacious goal of founding the field of artificial intelligence. After many spectacular successes and numerous great disappointments, we are still waiting for a truly intelligent computer program to emerge.
- In 1969, a researcher at IBM discovers an elegant new way to structure the information in a database. The technique is now used to store and retrieve the information underlying most online transactions.
- In 1974, researchers in the British government's lab for secret communications discover a way for computers to communicate securely even when another computer can observe everything that passes between them. The researchers are bound by government secrecy—but fortunately, three American professors

independently discover and extend this astonishing invention that underlies all secure communication on the internet.

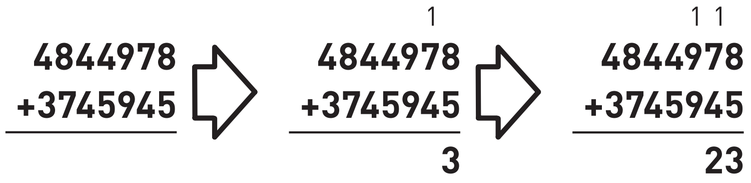
- In 1996, two Ph.D. students at Stanford University decide to collaborate on building a web search engine. A few years later, they have created Google, the first digital giant of the internet era.

As we enjoy the astonishing growth of technology in the 21st century, it has become impossible to use a computing device—whether it be a cluster of the most powerful machines available or the latest, most fashionable handheld device—without relying on the fundamental ideas of computer science, all born in the 20th century. Think about it: have *you* done anything impressive today? Well, the answer depends on your point of view. Have you, perhaps, searched a corpus of billions of documents, picking out the two or three that are most relevant to your needs? Have you stored or transmitted many millions of pieces of information, without making a single mistake—despite the electromagnetic interference that affects all electronic devices? Did you successfully complete an online transaction, even though many thousands of other customers were simultaneously hammering the same server? Did you communicate some confidential information (for example, your credit card number) securely over wires that can be snooped by dozens of other computers? Did you use the magic of compression to reduce a multimegabyte photo down to a more manageable size for sending in an e-mail? Or did you, without even thinking about it, exploit the artificial intelligence in a hand-held device that self-corrects your typing on its tiny keyboard?

Each of these impressive feats relies on the profound discoveries listed earlier. Thus, most computer users employ these ingenious ideas many times every day, often without even realizing it! It is the objective of this book to explain these concepts—the great ideas of computer science that we use every day—to the widest possible audience. Each concept is explained without assuming any knowledge of computer science.

ALGORITHMS: THE BUILDING BLOCKS OF THE GENIUS AT YOUR FINGERTIPS

So far, I've been talking about great "ideas" of computer science, but computer scientists describe many of their important ideas as "algorithms." So what's the difference between an idea and an algorithm? What, indeed, *is* an algorithm? The simplest answer to this



The first two steps in the algorithm for adding two numbers.

question is to say that an algorithm is a precise recipe that specifies the exact sequence of steps required to solve a problem. A great example of this is an algorithm we all learn as children in school: the algorithm for adding two large numbers together. An example is shown above. The algorithm involves a sequence of steps that starts off something like this: “First, add the final digits of the two numbers together, write down the final digit of the result, and carry any other digits into the next column on the left; second, add the digits in the next column together, add on any carried digits from the previous column…”—and so on.

Note the almost mechanical feel of the algorithm’s steps. This is, in fact, one of the key features of an algorithm: each of the steps must be absolutely precise, requiring no human intuition or guesswork. That way, each of the purely mechanical steps can be programmed into a computer. Another important feature of an algorithm is that it always works, no matter what the inputs. The addition algorithm we learned in school does indeed have this property: no matter what two numbers you try to add together, the algorithm will eventually yield the correct answer. For example, although it would take a rather long time, you could certainly use this algorithm to add two 1000-digit numbers together.

You may be a little curious about this definition of an algorithm as a precise, mechanical recipe. Exactly how precise does the recipe need to be? What fundamental operations are permitted? For example, in the addition algorithm above, is it okay to simply say “add the two digits together,” or do we have to somehow specify the entire set of addition tables for single-digit numbers? These details might seem innocuous or perhaps even pedantic, but it turns out that nothing could be further from the truth: the real answers to these questions lie right at the heart of computer science and also have connections to philosophy, physics, neuroscience, and genetics. The deep questions about what an algorithm really is all boil down to a proposition known as the *Church–Turing thesis*. We will revisit these issues in chapter 10, which discusses the theoretical limits of computation and some aspects of the Church–Turing thesis. Meanwhile, the informal notion of an algorithm as a very precise recipe will serve us perfectly well.

Now we know what an algorithm is, but what is the connection to computers? The key point is that computers need to be programmed with very precise instructions. Therefore, before we can get a computer to solve a particular problem for us, we need to develop an algorithm for that problem. In other scientific disciplines, such as mathematics and physics, important results are often captured by a single formula. (Famous examples include the Pythagorean theorem, $a^2 + b^2 = c^2$, or Einstein's $E = mc^2$.) In contrast, the great ideas of computer science generally describe *how* to solve a problem—using an algorithm, of course. So, the main purpose of this book is to explain what makes your computer into your own personal genius: the great algorithms your computer uses every day.

WHAT MAKES A GREAT ALGORITHM?

This brings us to the tricky question of which algorithms are truly "great." The list of potential candidates is rather large, but I've used a few essential criteria to whittle down that list for this book. The first and most important criterion is that the algorithms are used by ordinary computer users every day. The second important criterion is that the algorithms should address concrete, real-world problems—problems like compressing a particular file or transmitting it accurately over a noisy link. For readers who already know some computer science, the box on the next page explains some of the consequences of these first two criteria.

The third criterion is that the algorithms relate primarily to the *theory* of computer science. This eliminates techniques that focus on computer hardware, such as CPUs, monitors, and networks. It also reduces emphasis on design of infrastructure such as the internet. Why do I choose to focus on computer science theory? Part of my motivation is the imbalance in the public's perception of computer science: there is a widespread belief that computer science is mostly about programming (i.e., "software") and the design of gadgets (i.e., "hardware"). In fact, many of the most beautiful ideas in computer science are completely abstract and don't fall in either of these categories. By emphasizing these theoretical ideas, it is my hope that more people will begin to understand the nature of computer science as an intellectual discipline.

You may have noticed that I've been listing criteria to eliminate potential great algorithms, while avoiding the much more difficult issue of defining greatness in the first place. For this, I've relied on

The first criterion—everyday use by ordinary computer users—eliminates algorithms used primarily by computer professionals, such as compilers and program verification techniques. The second criterion—concrete application to a specific problem—eliminates many of the great algorithms that are central to the undergraduate computer science curriculum. This includes sorting algorithms like quicksort, graph algorithms such as Dijkstra's shortest-path algorithm, and data structures such as hash tables. These algorithms are indisputably great and they easily meet the first criterion, since most application programs run by ordinary users employ them repeatedly. But these algorithms are generic: they can be applied to a vast array of different problems. In this book, I have chosen to focus on algorithms for specific problems, since they have a clearer motivation for ordinary computer users.

Some additional details about the selection of algorithms for this book. Readers of this book are not expected to know any computer science. But if you do have a background in computer science, this box explains why many of your old favorites aren't covered in the book.

my own intuition. At the heart of every algorithm explained in the book is an ingenious trick that makes the whole thing work. The presence of an "aha" moment, when this trick is revealed, is what makes the explanation of these algorithms an exhilarating experience for me and hopefully also for you. Since I'll be using the word "trick" a great deal, I should point out that I'm not talking about the kind of tricks that are mean or deceitful—the kind of trick a child might play on a younger brother or sister. Instead, the tricks in this book resemble tricks of the trade or even magic tricks: clever techniques for accomplishing goals that would otherwise be difficult or impossible.

Thus, I've used my own intuition to pick out what I believe are the most ingenious, magical tricks out there in the world of computer science. The British mathematician G. H. Hardy famously put it this way in his book *A Mathematician's Apology*, in which he tried to explain to the public why mathematicians do what they do: "Beauty is the first test: there is no permanent place in the world for ugly mathematics." This same test of beauty applies to the theoretical ideas underlying computer science. So the final criterion for the algorithms presented in this book is what we might call Hardy's beauty test: I hope I have succeeded in conveying to the reader at least some portion of the beauty that I personally feel is present in each of the algorithms.

Let's move on to the specific algorithms I chose to present. The profound impact of search engines is perhaps the most obvious example of an algorithmic technology that affects all computer users, so it's not surprising that I included some of the core algorithms of web search. Chapter 2 describes how search engines use *indexing* to find documents that match a query, and chapter 3 explains *PageRank*—the original version of the algorithm used by Google to ensure that the most relevant matching documents are at the top of the results list.

Even if we don't stop to think about it very often, most of us are at least *aware* that search engines are using some deep computer science ideas to provide their incredibly powerful results. In contrast, some of the other great algorithms are frequently invoked without the computer user even realizing it. Public key cryptography, described in chapter 4, is one such algorithm. Every time you visit a secure website (with `https` instead of `http` at the start of its address), you use the aspect of public key cryptography known as *key exchange* to set up a secure session. Chapter 4 explains how this key exchange is achieved.

The topic of chapter 5, error correcting codes, is another class of algorithms that we use constantly without realizing it. In fact, error correcting codes are probably the single most frequently used great idea of all time. They allow a computer to recognize *and correct* errors in stored or transmitted data, without having to resort to a backup copy or a retransmission. These codes are everywhere: they are used in all hard disk drives, many network transmissions, on CDs and DVDs, and even in some computer memories—but they do their job so well that we are never even aware of them.

Chapter 6 is a little exceptional. It covers pattern recognition algorithms, which sneak into the list of great computer science ideas despite violating the very first criterion: that ordinary computer users must use them every day. Pattern recognition is the class of techniques whereby computers recognize highly variable information, such as handwriting, speech, and faces. In fact, in the first decade of the 21st century, most everyday computing did not use these techniques. But as I write these words in 2011, the importance of pattern recognition is increasing rapidly: mobile devices with small on-screen keyboards need automatic correction, tablet devices must recognize handwritten input, and all these devices (especially smartphones) are becoming increasingly voice-activated. Some websites even use pattern recognition to determine what kind of advertisements to display to their users. In addition, I have a personal bias toward pattern recognition, which is my own area of research. So chapter 6 describes three of the most interesting and successful pattern recognition techniques: nearest-neighbor classifiers, decision trees, and neural networks.

Compression algorithms, discussed in chapter 7, form another set of great ideas that help transform a computer into a genius at our fingertips. Computer users do sometimes apply compression directly, perhaps to save space on a disk or to reduce the size of a photo before e-mailing it. But compression is used even more often under the covers: without us being aware of it, our downloads or uploads may be compressed to save bandwidth, and data centers often compress customers' data to reduce costs. That 5 GB of space that your e-mail provider allows you probably occupies significantly less than 5 GB of the provider's storage!

Chapter 8 covers some of the fundamental algorithms underlying databases. The chapter emphasizes the clever techniques employed to achieve *consistency*—meaning that the relationships in a database never contradict each other. Without these ingenious techniques, most of our online life (including online shopping and interacting with social networks like Facebook) would collapse in a jumble of computer errors. This chapter explains what the problem of consistency really is and how computer scientists solve it without sacrificing the formidable efficiency we expect from online systems.

In chapter 9, we learn about one of the indisputable gems of theoretical computer science: digital signatures. The ability to "sign" an electronic document digitally seems impossible at first glance. Surely, you might think, any such signature must consist of digital information, which can be copied effortlessly by anyone wishing to forge the signature. The resolution of this paradox is one of the most remarkable achievements of computer science.

We take a completely different tack in chapter 10: instead of describing a great algorithm that already exists, we will learn about an algorithm that *would* be great if it existed. Astonishingly, we will discover that this particular great algorithm is impossible. This establishes some absolute limits on the power of computers to solve problems, and we will briefly discuss the implications of this result for philosophy and biology.

In the conclusion, we will draw together some common threads from the great algorithms and spend a little time speculating about what the future might hold. Are there more great algorithms out there or have we already found them all?

This is a good time to mention a caveat about the book's style. It's essential for any scientific writing to acknowledge sources clearly, but citations break up the flow of the text and give it an academic flavor. As readability and accessibility are top priorities for this book, there are no citations in the main body of the text. All sources are, however, clearly identified—often with amplifying comments—in the "Sources and Further Reading" section at the end of the book. This section also points to additional material that interested readers can use to find out more about the great algorithms of computer science.

While I'm dealing with caveats, I should also mention that a small amount of poetic license was taken with the book's title. Our *Nine Algorithms That Changed the Future* are—without a doubt—revolutionary, but are there exactly nine of them? This is debatable, and depends on exactly what gets counted as a separate algorithm. So let's see where the "nine" comes from. Excluding the introduction and conclusion, there are nine chapters in the book, each covering algorithms that have revolutionized a different type of computational task, such as cryptography, compression, or pattern recognition. Thus, the "Nine Algorithms" of the book's title really refer to nine classes of algorithms for tackling these nine computational tasks.

WHY SHOULD WE CARE ABOUT THE GREAT ALGORITHMS?

Hopefully, this quick summary of the fascinating ideas to come has left you eager to dive in and find out how they really work. But you may still be wondering: what is the ultimate goal here? So let me make some brief remarks about the true purpose of this book. It is definitely not a how-to manual. After reading the book, you won't be an expert on computer security or artificial intelligence or anything else. It's true that you may pick up some useful skills. For example: you'll be more aware of how to check the credentials of "secure" websites and "signed" software packages; you'll be able to choose judiciously between lossy and lossless compression for different tasks; and you may be able to use search engines more efficiently by understanding some aspects of their indexing and ranking techniques.

These, however, are relatively minor bonuses compared to the book's true objective. After reading the book, you *won't* be a vastly more skilled computer user. But you *will* have a much deeper appreciation of the beauty of the ideas you are constantly using, day in and day out, on all your computing devices.

Why is this a good thing? Let me argue by analogy. I am definitely not an expert on astronomy—in fact, I'm rather ignorant on the topic and wish I knew more. But every time I glance at the night sky, the small amount of astronomy that I do know enhances my enjoyment of this experience. Somehow, my understanding of what I am looking at leads to a feeling of contentment and wonder. It is my fervent hope that after reading this book, you will occasionally achieve this same sense of contentment and wonder while using a computer. You'll have a true appreciation of the most ubiquitous, inscrutable black box of our times: your personal computer, the genius at your fingertips.