1

An Introduction to Computer Science

COMPUTER SCIENCE IS WITHIN YOU

Have you ever completed a Sudoku puzzle? Can you sing or play a tune from sheet music? Can you decipher a knitting pattern or follow steps to fold an origami figure? If you've ever attempted any of these things, then you have already experienced one of the fundamental building blocks in computer science (CS), the **algorithm**.

The word *algorithm* may sound technical and scary, but really it's just a list of steps that can be followed to carry out a task. We see algorithms every day when we engage in activities that have instructions. Games, recipes, and crafts all become physical representations of algorithms, unknowingly aligning us with our family computers—because in the end, algorithms are also at the heart of every piece of code.

Algorithm: A list of steps that can be followed to carry out a task.

In this book, we'll encourage you to explore your relationship with that digital diva, the personal computer. We'll act as guides, taking you from familiar territory into new terrain, where you'll interact with computers as a medium for invention. Start imagining how you'll equip your students for this journey, following President Obama's advice:

> *"Don't just buy a new video game, make one. Don't just download the latest app, help design it. Don't just play on your phone, program it!"*
>
> —President Barack Obama (quoted in Machaber, 2014)

As you prepare to dive into computing activities in the pages to come, pack up your enthusiasm for learning new things, arm yourself with a growth mindset, and start imagining limitless possibilities!

By the way, here's the first line of code ever written by a U.S. president:

```
moveForward(100);
```

If Obama can do it, you and your students can too!

Now, let's kick things off by investigating the underpinnings of CS—computational thinking. A brief scan here will ground you in the concept. We provide a more comprehensive look in Chapters 6 through 10 (with lesson plans!).

AN INTRODUCTION TO COMPUTATIONAL THINKING

The buzz around **computational thinking** began in the early 2000s when Jeannette M. Wing, a professor of computer science at Carnegie Mellon University, introduced the term in a series of academic papers (Wing, 2006). At the time, she presented computational thinking as a set of attitudes and skills a person would need to confidently persist toward identifying, posing, and solving problems. The term soon morphed to include capabilities that apply to preparation for CS. We'll explore these critical capabilities, as well as pay respect to the original concept of open-minded problem solving that sometimes goes missing in today's world of standardized testing and scripted lessons.

Computational thinking: Using special thinking patterns and processes to pose and solve problems or prepare programs for computation. Notably decomposition, pattern matching, abstraction, and automation.

Distilled down to its most fundamental elements, computational thinking is comprised of four pillars:

- Decomposition
- Pattern matching
- Abstraction, and
- Algorithms (sometimes referred to as *automation*)

With these four skills, one can prepare any problem for a mechanical solution. But what does that really mean? Let's unpack each element as we consider something you're likely familiar with, a sudoku puzzle (Figure 1.1).

These little grids seem unassuming enough, but once you start to play, you'll see they're packed with complexity. The point of a sudoku is to fill in all the blank squares with digits (typically 1 through 9) in such a way that there is exactly one of each digit in any given column, row, or block of cells.

Even with a ton of experience solving these little enigmas, it can still be quite challenging to describe how to go about it. If we want to prepare a sudoku *algorithm* (a procedure or formula) for *automation* (running on a machine), we're going to need to use some computational thinking.

7	8	1	2	4	6	9		
	3	9			1	4	7	
4	5	2			9	8	6	
	9	5		2			1	8
2			7	1		5	3	9
1	7		5			2		6
	2	3	4	6	7	1		5
8	1				5		2	7
	6			8	2	3		

FIGURE 1.1 Standard 9 × 9 Sudoku

First, let's use **decomposition** on the problem by consciously identifying the list of steps we might go through to figure out what should happen in each space. For ease of explanation, let's focus on one 4 × 4 square at a time in the mini-puzzle below, beginning with the upper left blank cell (row 1, column 1).

Decomposition: Breaking a problem down into smaller, more manageable parts.

Figure 1.2 below is a simple version of a sudoku puzzle. How would you determine what goes into the first empty corner? As a human, I might follow an algorithm like this:

1. Look at the numbers that are missing in row 1. (That would be 1 and 2.)
2. Look at the numbers that are missing in column 1. (That would be 2 and 3.)
3. Look at the numbers missing in quadrant 1. (That would be 1 and 2.)
4. If there is only one number missing from all three sets, that is the number that goes in the upper left cell. (And that would be 2!)
5. If there is a second number missing from all three sets, continue to the next cell and come back when you have more information.

	3	**4**	
4			**2**
1			**3**
	2	**1**	

FIGURE 1.2 Simplified 4 × 4 Sudoku

Sure, we've made this a bit straightforward, but the strategy should hold when it comes to solving for any individual cell. Now, to illustrate, let's look at the steps for the cell in row 2, column 3. (No answers this time!)

1. Look at the numbers that are missing in row 2.
2. Look at the numbers that are missing in column 3.
3. Look at the numbers missing in quadrant 2.
4. If there is only one number missing from all three sets, that is the number that goes in the square.
5. If there is a second number missing from all three sets, continue to the next square and come back when you have more information.

At this point, we can apply more computational thinking to try to get an algorithm that will work for automating the discovery of a solution for any sudoku.

Pattern matching: Finding similarities between items as a way of gaining extra information.

Here, we will use **pattern matching**. Do you see any patterns between our first set of steps and our second? Let's compare the first instruction in both.

> Look at the numbers that are missing in row 1.
>
> Look at the numbers that are missing in row 2.

The instructions are nearly identical. As a matter of fact, if you were to list out steps for each and every cell, you would find that the only thing that changed was the number of the row that you were working with. That's a pattern! What can we do with it?

Abstraction: Ignoring certain details in order to come up with a solution that works for a more general problem.

This is where **abstraction** comes in. Abstraction is simply the act of removing details that are too specific, so that one instruction can work for multiple problems.

To complete the abstraction on the instructions above, we might turn the sentence into something like this:

> Look at the numbers that are missing in row _____.

The blank now becomes a spot where you enter the row number of the empty square that you are currently working with.

Can you take the rest of the instructions and abstract them out so that you wind up with a final algorithm for automation of a sudoku of this size? Did you come up with a different method altogether?

Mental Agility

How do you feel while working at puzzles like sudoku, deciphering crochet patterns, learning new sheet music, or following diagrams to put together Ikea furniture? Do you feel the burn as you stretch your mental muscles? Do you enter a state of "flow" when deeply immersed? Do you find yourself surprised at how time flies when you're so engaged that every other concern seems to vanish? Are you deeply satisfied when troubleshooting helps you overcome obstacles and move forward? At the very least, aren't you pleased to have applied cognitive effort to a new challenge?

The kind of mental workout we've been talking about is similar to what one might feel as a genuine, bona fide, creative problem solver taking the first steps toward becoming a computer scientist. Get ready for more opportunities to apply your computational thinking to exercises and web activities in the next chapters, and prepare for a deeper dive into computational thinking in Chapters 6 through 10.

WHAT COMPUTER SCIENCE IS

You can think of **computer science** as the study of how to use computers and computational thinking to solve problems, not merely the act of using technology. It's like the difference between watching a movie and producing and directing one (Figure 1.3). We already have a generation of filmgoers; now we need more producers and directors (software designers, developers, and programmers) who can create the vast array of products and services the world needs.

Computer science: The study and use of computers and computational thinking to solve problems.

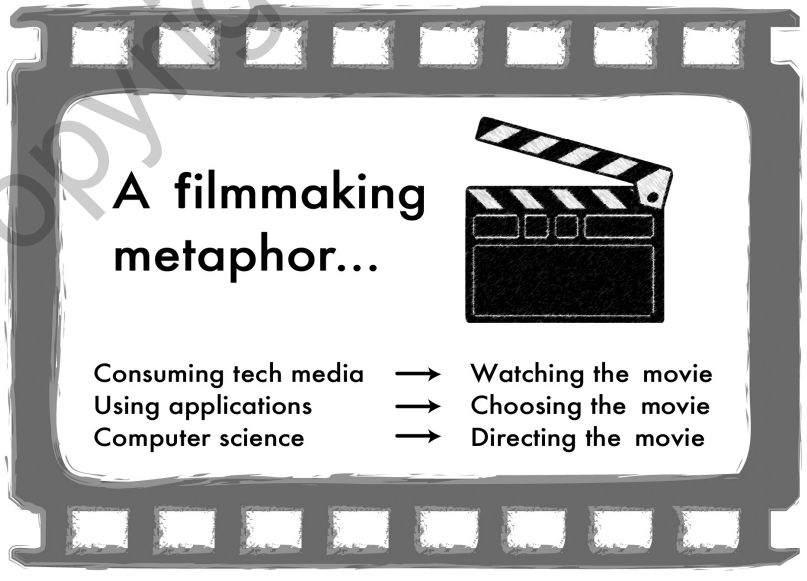

FIGURE 1.3 A Filmmaking Metaphor

In the Beginning

You may be surprised to find out that CS as we know it developed a full thirty years before the first reprogrammable computer (Rabin, 2012). It all came about when Alan Turing and Alonzo Church theorized that there were definite limits on what could be computed using methods of **automation**, formalizing the idea of algorithms in the process.

Automation: Controlling a process by automatic means, reducing human intervention to a minimum.

Since then, CS has evolved from the study of what can be automated to the practice of automating with finesse! Computer science and computational thinking are not the same thing, but computational thinking is a vital component when it comes to translating real-world situations or solutions into algorithms. In this book, we will often mention one without the other, but we do so with the understanding that the two are strongest together.

WHAT COMPUTER SCIENCE IS NOT

Next, and almost equally as important as knowing what computer science *is,* is knowing what computer science *is not.* Often, families and educators believe their students are learning CS because they are in a class that meets in the computer lab several times a week. More often than not, these sessions deal with learning to use specific software, such as a word processor or graphic design program. These are great reasons to work with computers, but they aren't CS.

Computer science is an excellent tool for developing students' digital skills. In the process of learning CS, even the youngest kids learn to use a keyboard, copy and paste, save files, and access the internet effectively and responsibly. CS encompasses these frequently taught skills, but the converse isn't true.

Let's address some other aspects of what CS *is not.* CS is not boring. It is not a solitary act, and it is not an advanced subject that's practiced only by the most brilliant or privileged. CS is a set of beautiful, digital art forms that allow you to express your thoughts and feelings while you innovate and provide solutions for humanity. CS is a growing subject, full of unexplored potential and room for deviation. It is not static or depleted.

CS is also not *just* programming. While you might be hard-pressed to describe the difference between the two terms, give us a few paragraphs and you won't be anymore.

Programming is just one specific area of CS. It is most often thought of as writing code for a machine, but programming also encompasses the thought processes, design structures, and **debugging** that occur while coding. Talking about the programming process of CS is a lot like talking about the scriptwriting process. It is a supremely important element that provides the road map for the finished product, but there are many other pieces to consider. In filmmaking, those pieces might include acting, directing, and editing; in CS, they include software engineering, user interfaces, and hardware design.

Debugging: To track down and correct errors.

People are also confused about the difference between *programming* and *coding*. This distinction is a little more nuanced. For the most part, the two terms can be used interchangeably, but to people who care, there is a difference.

It used to be that just about everyone who wrote code was a *programmer*. They were educated specialists who took pride in their craft and put great thought into the programs they wrote. In the 1980s and 1990s, more and more people began to teach themselves to develop computer code. The details of logical and beautiful design were not always embedded into the work of this new generation of self-made technologists, often referred to as *hackers* or *coders* by their professional counterparts.

Today, the term *hacker* has taken on a more sinister connotation, whereas the label *coder* continues to describe someone who can piece together a program but may not have the chops to design code with finesse. As such, a professional programmer may not take kindly to being called a *coder*.

That said, we frequently use the term *coding* in this book because it is appropriate when describing programming at an entry level. We will also frequently refer to *programming* and *computer science* when talking about practices, classes, or curricula that encompass more than an introductory glimpse of code.

As you read the rest of this book, we will show you not only how you can test the CS waters on your own but also how you can bring this capability to your students, beginning even with prereaders. No matter what your level of experience, this is a journey worth taking. So, if you're still concerned about this digital trek . . . don't worry, we've got you!