# 2

## THE MANY BENEFITS OF COMPUTER SCIENCE EDUCATION

In this chapter we'll explore the cognitive, academic, and professional benefits of computer science education that can enhance your students' grades, personal development, and careers. Because you're reading this book, most likely you already value a computer science education. But as an educator, you'll encounter students, parents, and administrators who need convincing. The benefits covered in this chapter include talking points you can use to motivate students to succeed, encourage parents to support their children academically, and garner support from school administrators for your computer science program.

These benefits build upon one another. The cognitive benefits can make students sharper and improve some areas of academic performance. The academic benefits can improve academic outcomes and lead to more successful professional lives for students.

## Cognitive Benefits

The exercise of coding can provide cognitive benefits, such as enhancing problem solving, verbal acuity, working memory, and tenacity. Although programming is associated with improved cognitive performance on a variety of tests, there are caveats. To take advantage of these benefits, they must be explicitly written into lesson plans. For example, in Chapter 4, we'll explore the concept of *algorithms*, a set of rules specifying how to solve a problem, and how students are already processing many algorithms in their heads daily when they follow various instructions. Educators who want their students to transfer their computer science understanding of algorithms to troubleshooting algorithms in their daily lives must construct lesson plans intentionally to address this concept.

### *Problem-Solving*

Navigating code requires thinking abstractly and formally. Many problem-solving skills and techniques that we learn crafting code are transferable to other problem domains. For instance, working on an electronics project with some students for the first time, I was floored to discover how my software development background empowered me to mentor in this novel domain. The code logic and wiring logic were analogous to one another, and I was able to apply the same problem-solving skills I use to write and debug code to wire and troubleshoot the electronics.

For example, when programming, it's useful to *decompose* larger problems into smaller components that are easier to comprehend and solve. When applying decomposition to coding, we practice *coding a bit and testing a bit (CABTAB)*, making sure each piece of code works before moving on to the next piece. I applied this convention to test every light as the students and I wired it up and to retest the lights we had wired prior. When components stopped working, I used the *wolf fence* debugging technique, which is to find a point in the middle of the code or wiring and see whether the problem occurs before or after it. The technique is similar to a farmer building a fence down the middle of their farm to narrow down in which half of the land the wolf is still eating their sheep. The coding logic and electronics logic were *isomorphs* of one another, different representations of the same problem. Educators should explicitly highlight and reinforce the similarities in problem-solving strategies between domains to help students understand. Because so much of computer science emphasizes modeling problems from the real world into formal, symbolic logic, educators can use this opportunity to teach students how to apply formal reasoning to real-world problems.

Research on the transference of problem-solving skills involved in programming to other domains shows mixed results. In 1990, David Palumbo, Assistant Professor of Instructional Technology at the University of Houston—Clear Lake, analyzed the

research on programming language instruction and the transference of problem-solving skills to other domains. He found that students' ability to apply the skills they learned in programming to other areas often depended on how similar the problem domains were, the age-appropriateness and cognitive prerequisites of the instruction, how the language was taught to students, which language was taught, the degree to which the teacher mediated student learning, and what level of expertise a student could obtain within the instruction's time frame.

The more closely a domain resembles a programming environment, the better the skill transference. For example, one study found that learning the now-defunct BASIC programming language enhanced performance on specific algebra word problems. Additionally, the way programming is taught can impact the transference of problem-solving skills to other domains. Another study found that a mediated learning approach that specifically focused on problem-solving skills led to significant improvement in students' abilities to break down problems into smaller components, make analogies, engage in systematic trial and error, and make logical inferences from data. How coding is taught has a huge impact on the non-coding skills students might take away from the lessons.

### *Programming Is Communication*

Coding is an exercise in communication. In a 2007 study, 95 percent of 780 surveyed programmers considered understanding existing code a significant part of their job. When we code, we're composing a story in such a way that our collaborators can follow it. Those collaborators can also include our future selves: we might one day return to maintain the code and be reading it as if for the first time. Other collaborators can be quite obtuse, such as the computer, which reads our code literally, doing exactly what we ask it to even though it's not what we intended. When writing code, we must keep our audience in mind, making it literal for the computer and expressive enough for our peers.

With increasing market demands for technical professionals, some administrators are suggesting schools award students foreign language credit for taking computer programming classes. But this is misguided. Learning a foreign language gives us deeper insights into our own language and builds bridges to other cultures to better understand our own. At the same time, programming languages share many characteristics with natural languages that might allow the two subjects to complement and reinforce one another. Programming languages, like natural languages, have syntax, grammar, and semantics. It's possible to write lines of code that are syntactically correct but are semantically incorrect, just as Noam Chomsky's "Colorless green ideas sleep furiously" is grammatically correct but semantical nonsense. Like natural languages, programming languages branch out and relate to one another.

Bilingualism is linked to numerous cognitive benefits, such as improved metalinguistic awareness, creativity, and problem-solving. To determine whether programmers experienced these bilingual benefits, Hannah Wright, a child development master's student at the University of London, did a study. She worked with 10 professional computer programmers (aged 22–25), 10 adolescent computer programmers (aged 14–17), and an equal number of controls for a total of 40 monolingual, English-speaking young adults and adolescents to see how they performed on cognitive tests where bilinguals had an advantage. The two groups of programmers performed significantly better on the Attention Network Task, which measured their ability to achieve and maintain an alert state, select information from sensory input, and monitor and resolve conflicts.

In addition to communicating with their fellow programmers, the literal-thinking computer, and their future selves, programmers must also communicate all their ideas in the formal and foreign language of code. Computer programming is a complex, challenging, and rewarding medium to converse in.

### *Coding Exercises Working Memory*

Students immersed in code exercise their *working memory* and focus. Working memory is the ability of individuals to remember and process information in their immediate consciousness—normally 10 to 15 seconds. How much information a person can hold in working memory and for how long varies depending on the task and from person to person. Psychologists often use n-back tasks, such as reciting increasingly long strings of numbers, to measure working memory. Writing code is a working memory–intensive task. It requires the coder to hold many variables and algorithms in focus to understand and work with the code.

Every software developer is familiar with this scenario: you're tracking down an especially irksome bug in the software. Maybe it's a bit of data that is getting corrupted. You've traced the value from the database table where it's stored, through the view sending it to the business layer, past the object model, watching function after function handle it. You track it coming out to the user interface where more functions interact with it and the user can manipulate it. Next, it gets sent back to form handlers, persistence functions, and then your phone vibrates and the whole mental construct comes crashing down.

Students who come into programming with a strong working memory are more successful at learning how to code. In 1991, Valerie Shute, at the Air Force Human Resources Laboratory, tested 260 people. They took a seven-day Pascal programming class from an automated tutoring system to assess their working memory, problem-solving skills, and learning styles. Using a wide variety of tests to measure different dimensions of the students' working memory, including quantitative, verbal, and spatial skills, she found a

strong correlation between students having a strong working memory and the ability to successfully learn programming.

Although this study found that students with better working memories are better programmers, it doesn't mean students with weaker working memories can't learn programming. It simply means that students with weaker working memories need to break down programs and problems into smaller components with fewer variables to keep in mind. Computer science education encourages students to practice this programming technique by decomposing larger problems into smaller problems that are easier to solve.

Also, these students might only need to break down problems when they first start to learn programming. Multiple research studies have found that working memory responds to exercise, just like our muscles or other components of our brain, due to its plasticity. In one study, researchers found that subjects could expand their working memory from one item to four in just 10 hours of practice spread across five days.

### *Coding Exercises Tenacity*

Coding demands students work through problems. Often, those problems require long-term engagement. When students decompose a complex coding problem into a bunch of smaller problems, they feel rewarded more often for each little problem they solve. For especially challenging problems, those in which they must research for hours or days gathering bits of evidence, they will ideally begin to grow eager to know what they'll eventually uncover. Coding is a self-reinforcing task that rewards students for tackling big projects.

In recent years, researchers have identified tenacity—or motivation, effort, perseverance, or "grit"—as a crucial "non-cognitive" skill in academic performance. The Common Core state standard "Make sense of problems and persevere in solving them" recognizes the importance of stick-to-itiveness in academic achievement. Coding projects provide an excellent opportunity for students to exercise their tenacity.

As the educator and coach, you must assure your students that *there is a solution to every problem*. Finding that solution might involve some part of the code they never thought to investigate. It might involve having the student explain the problem to a peer, because articulating the problem in a way someone else can understand often helps to uncover deeper insights into it. The solution might come to student at three in the morning and leave them staring at the ceiling for several hours waiting to go to school. Coaching your students to stick with their coding problems encourages them to flex their tenacity muscles and develop coping strategies for seeing their way through difficult problems.

## Quantifiable You

We've touched on many different studies examining the potential cognitive benefits of coding and computational thinking. In each of these studies, researchers had to identify nebulous cognitive attributes, concretely define them, and establish standard quantifiable measures for them. Software developers must also wrestle with what are often vaguely defined requirements or processes and concretely define them to implement variables and logic the computer can work with.

Try this self-improvement exercise with your students: ask them, "What are the variables that define you?" For example, in physical fitness, we can track numerous metrics: resting heart rate, body mass index (BMI), number of pull-ups, minutes per mile, blood-glucose levels, maximum heart rate, number of steps in a day, and so forth. All of these are variables we can quantify about our athleticism and healthy lifestyle. You could make each metric a column in a spreadsheet. Then make each row a date when you measured it, and chart a graph of how you improve every one over time.

Your school tracks many academic performance variables in the form of grades, but the metrics defining a math grade are very different from those behind an English grade. For example, an English teacher evaluating an essay might count how many times the student cites evidence, measuring frequency. A math teacher might evaluate the number of correct answers on a test, measuring proportionality. Additionally, teachers evaluate behavior variables, such as the duration for which a student focuses or reads independently, the latency of how long it takes a student to answer a question, or the quantity of how many prompts a student needs on a task versus working independently. In academia, these different measurement points are called *rubrics*, and being aware of them can help students know what to focus on and track in their personal progress.

There are also many activities in students' personal lives that school doesn't formally measure but that contribute to their academic success. For example, time spent independently reading, focusing on difficult math problems, or making progress on a long-term project also improves their grades. Encouraging students to log these metrics in terms of duration, frequency, proportionality, and independence can help them think about their thinking, appreciate how much work they're doing, and make them aware of how they can best improve themselves while also practicing the art of defining and quantifying variables needed to evaluate them computationally.

## Educational Benefits

When we write code, we're working toward an outcome, a goal. Every application we write requires inquiry, deep thinking, self-direction, presentation, peer review, revision, and iteration. Each application we craft is a project, which is what makes it complement academia so well.

*Project-based learning (PBL)* is a teaching method in which students engage in deep sustained inquiry into a complex problem or question and produce a research-based artifact to present to their peers. The PBL approach takes advantage of a student's natural curiosity, challenging them with a problem and sending them off to find the answer on their own with minimal teacher guidance. While addressing the main project, students will encounter many related subproblems they must solve, just like solving larger real-world problems.

PBL is associated with positive educational outcomes. In one study, students in a school using PBL significantly outperformed those in a traditional school in mathematics and conceptual and applied knowledge, and three times as many students passed the British national exam. Additionally, PBL aligns well with the intentions of the Common Core state standards, such as "Research to Build and Present Knowledge," "Comprehension and Collaboration," and "Presentation of Knowledge and Ideas."

PBL is a highly rewarding way of learning. Producing artifacts deeply immerses students in the subject matter, making demands of their critical thinking and problem-solving skills. Through sustained inquiry, each question leads to more questions and each reference to more references. They share their product with their peers, who will offer critiques and enhancements. At the end of all their efforts, the product will be an artifact of which they're deeply proud.

Coding projects are ideal for PBL because they require deep reading, experimentation, research, collaboration, peer review, and an end product they can showcase in a portfolio with other academic achievements. John McManus and Philip Costello at Randolph-Macon College published their positive experience applying PBL to coding projects in their classrooms in the paper *Project Based Learning in Computer Science: A Student and Research Advisor's Perspective.* The projects consisted of students programming autonomous drones to collect scientific data. Although their personal experience was anecdotal, they found students were driven to perform well due to the sense of project ownership. They also felt that the PBL approach allowed students to practice what they had previously learned while acquiring new hands-on skills that are difficult to teach in a classroom environment. These hands-on skills were seen as fostering discipline and professional growth that would benefit students beyond an academic setting. Chapter 8 covers in detail how to facilitate PBL projects in your classroom and how to align the project development process with software development practices.

### *Code Is a Complex Text Requiring Close Reading*

Close reading is an exercise that has students develop a deep, critical understanding of a text. A close reading of a complex text might have students first read the text for key ideas, then reread key passages to understand the style and structure, and then reread the text again to draw inferences and conclusions that the text supports. The Common Core standards recommend close reading by telling students, "Read closely to determine what the text says explicitly and to make logical inferences from it; cite specific textual evidence when writing or speaking to support conclusions drawn from the text."

Reading unfamiliar code requires close reading. From 1983 to 1984, Nancy Pennington studied 80 professional programmers to learn how they comprehend code and form mental representations of it in their minds (*Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs*). She found that when programmers read unfamiliar code, they break up that code into generalized chunks that describe what is going on in sections of the software solution without getting into the minutiae.

Programmers closely read the lines of code to construct a narrative they can later recall as a story. Just as in close reading, the programmer revisits functions or code blocks the same way a critic returns to key passages in a text. They scrutinize naming conventions, semantics, and code structure with a critical eye to see whether there are ways to make it clearer and more legible, which improves its maintainability. The critical analysis of code is very much like the critical analysis of a complex text and exercises a student's ability to engage in deep, sustained focus.

### *Coding Requires Research and Collaboration*

Software developers spend much of their time asking questions. What's the best way to verify that user input is numeric? How do I filter out spaces in a string? How can my program upload a file or send an email? Writing a literary essay means starting from the reading and developing hypotheses from it, whereas writing software begins with defining an end goal and then researching your way to achieving it.

In the early 2000s, programmers kept thick books on their desks, one for each language they used in their day-to-day operations. Each book was approximately 1,000 pages long, listing every function the language encapsulated in alphabetical order. Programmers would spend hours of their day flipping through these texts, looking up functions, what arguments to send them, and what outputs they would return. It was a very intensive and solitary process.

The barriers to coding are much lower today than in previous decades. Coders now spend much of their time searching the web for answers. If they don't find what they need on the search engines, they can post a question to a forum where participants score

reputation points for providing the best, most accurate answers. Online research becomes a *dialogue* among experts and novices. Today, kids have it much easier when it comes to research, and everyone should be happy and excited for them because it's a good thing: this ecosystem of experts openly debating every detail of the programming craft is why software advances so rapidly.

After writing code based on their research, students can then share what they've written with others in the classroom and learn from one another. One concern with student-to-student peer review is that it creates a situation in which novices guide novices. Students peer-reviewing text often express opinions without supporting them. But the formality of code tends to force students to defend why they think the code should be refactored.

Because the code produces a verifiable output, the first criteria for evaluating a peer's comments is to see whether the code still works with their change request. Another criterion is to ask whether the change makes the code easier to comprehend and maintain. For example, a student might recommend moving a block of code into its own function. To justify this move, they might argue, "This block of code validates an email address. If you encapsulate it in a function, you can abstract away many lines of code into a single `verifyEmail()`, which you can call anywhere else you need to validate an email address. Then, if you ever need to change the verification logic, you need only do it in one place."

But students can learn even from their peers' misunderstandings. Just as with critiquing a text, students peer-reviewing one another's code will ask questions that should prompt the coder to evaluate whether they're presenting their ideas clearly. If a peer is confused by a variable named `searchResult` and the function it serves in a contact list, the coder might rewrite it as `searchContacts` to indicate what search result is being returned. Ideally, students should practice presenting their ideas more clearly in their code with an eye to making their code clearer for their collaborators.

## *Code Provides Playgrounds for Experimentation*

Today's classrooms are increasingly taking advantage of *manipulatives*: toys or objects that each teach a single skill or concept. For example, a teacher might use binary coins with printed values of 1, 2, 4, 8, 16, 32, 64, and so on, and have students use them as pretend currency to explore the base-2 number system. Manipulatives provide students with the opportunity to engage in self-directed learning through playful exploration and experimentation.

The act of coding requires perpetual experimentation and exploration. People who don't code often envision a very formal software writing process. They imagine project planners, engineers, and architects scoping and designing every detail of the system before a developer commits a single line of code. In practice, the process is much different. A significant portion of coding time is spent on *experimentation*. Often, code can

become so complex that even the coder doesn't know exactly how it works. So they play with it. They change the order of operations, tweak variables, or tell the computer to recite variables at different points to see whether they're transforming as intended. They might flip a value from "true" to "false" to see what happens. Or they might ask the application for a million apples. What about "tqewfsdfve" apples?  Code is itself a manipulative.

## Iterative You

Ask your students how the world has improved since they were younger. What are some technologies that have come out in the last year or two? These can be very small improvements, such as a new kind of headphone, smartphone, car dashboard, or other minor development we might otherwise not notice. How do these technologies build upon the technologies that existed before them? In other words, what technologies had to exist before these new technologies could emerge?

Consider a longer period of time. How has the world improved over a 90-year-old person's lifetime? What technologies didn't exist for them as children that enhance their lives today? Their lives were filled with small technological improvements that they might have hardly noticed at the time, but over a century, they amassed into a tidal wave of change.

In software development, these tiny regular enhancements are known as *iterations*. When developing software, we release a base working version of a product, usually version 1.0, and then iteratively add improvements to it in the form of versions 1.1, 1.2, 1.21, and so on. Not every iteration is wholly an improvement; some versions introduce new bugs or poorly thought-out changes, but additional iterations address these issues. All the major popular websites we use today started out as much smaller, less feature-rich products and grew into the expansive, complex applications we currently use.

Education works in an iterative fashion as well. We had to learn our letters and phonemes before we could read simple words. We had to read simple words before we could read compound words. We had to learn cardinality before we could learn addition and subtraction. We must be sufficiently proficient in math and reading to write software code.

One way to quantify how knowledge builds on knowledge is to draw a tree with prerequisites in the trunk and branches leading to more advanced subjects in the leaves. Where do your students feel they are in the tree? What skills do they need to climb onto the limbs of what they want to master? Up the tree of knowledge they

grow, as illustrated in Figure 2-1, learning building on learning they level up with each new grade until version 1.0 is released to the world on graduation day.

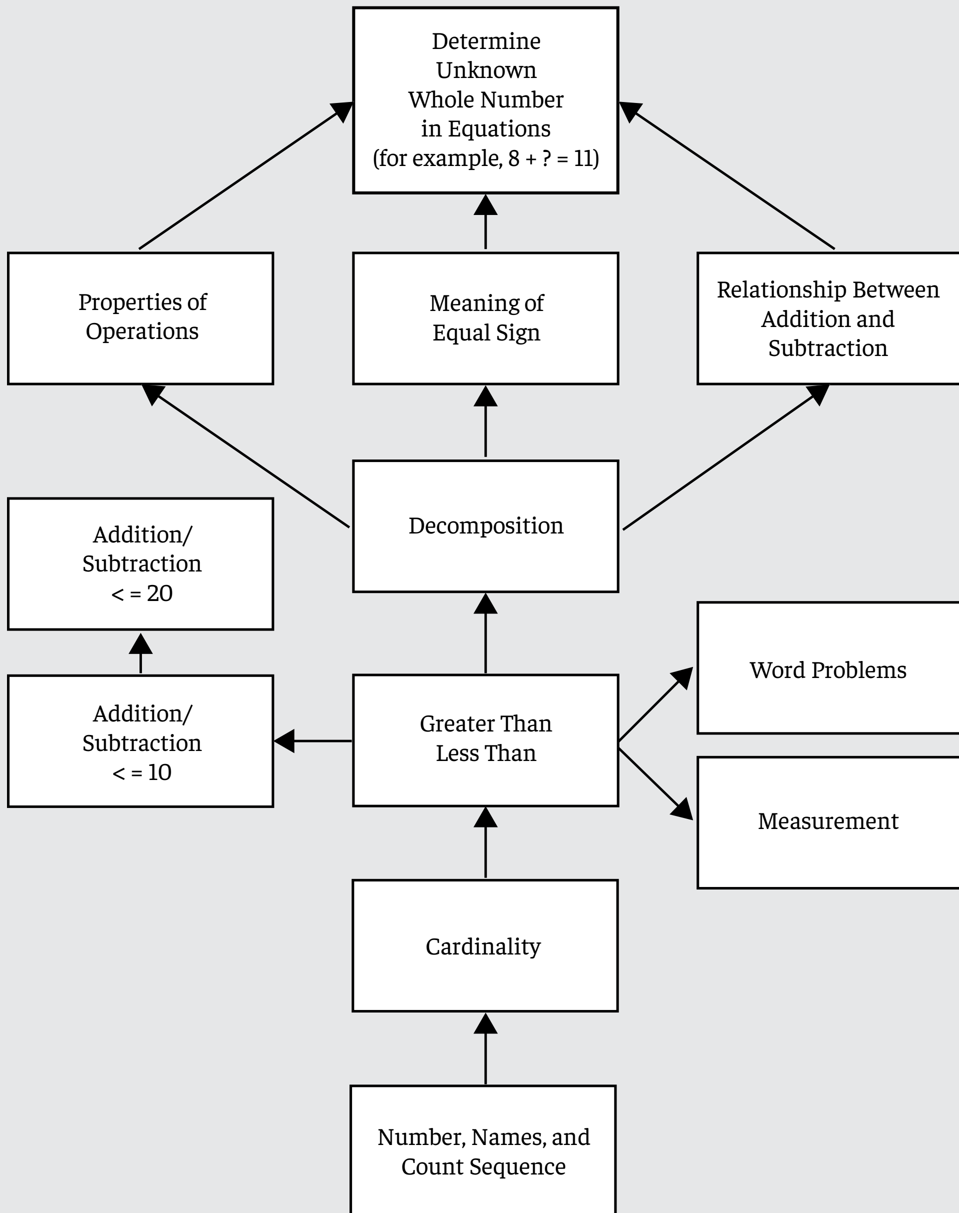


*Figure 2-1: How knowledge grows on knowledge in Kindergarten through second grade math*

Computers allow children to create their own learning environments through programming. Kids teach the computer how to think and behave, and they engage in metacognition in the process. Computers offer students access to an infinite number of manipulatives.

## Professional Benefits

It's difficult to think of anyone in today's workforce who doesn't work with information and doesn't augment their job performance with information systems. In the 21st century, many professions are tightly coupled to computational tools and information systems, and can therefore benefit from computer science education. In this section, we will learn a few of the many ways a computer science education will benefit your students' future selves when they enter the professional world.

### *Computer Science Makes You More Employable*

Software developers are in high demand and will remain so into the foreseeable future. In 2016, the average software developer's salary was $100,080, and the employment of software developers is projected to grow 22 percent from 2012 to 2022. In 2013, software engineers had a 3.6 percent unemployment rate, which was half the overall unemployment rate of 7.3 percent at that time.

But it's not just computer programmers who can improve their job security through computer science education. Everyone in an office building, from the managers to the financial overseers to the building supervisors, uses information systems to optimize their workflows. Outside of the office, construction workers must navigate an ocean of local, state, and federal building codes hosted in numerous online applications to properly do their jobs. Fast-food workers must rapidly monitor and manipulate data to ensure quality food gets to customers quickly. Freelance drivers must work with service provider apps and maximize their use of navigation applications to get customers to their destinations quickly. Nearly everyone works with software in today's economy, and those who understand what software can do will be more proficient than those learning on the job.

Along with this growing use of software comes new digital citizenship challenges for many professions. For example, as information technology grows more prevalent in hospitals, doctors and nurses will need to know how networked information systems comply with patient privacy laws. Marketers taking their messages to social media will need to be mindful of how those platforms are tracking the activities of their audience and monetizing that data. Writers, graphics designers, and editors working in Content Management Systems (CMS) will need to carefully ensure they don't violate local and global intellectual property and privacy laws.

As data grows more valuable and more sensitive as users entrust it to organizations, professionals must also be vigilant in protecting that data. A professional working in the organization's payroll department must be aware of the dangers of phishing attacks. They must proactively ask their organization's engineers about the software measures guarding their servers. They must be educated as to how having the organization's data compromised would be disastrous for their co-workers, and be familiar with cybersecurity concepts that secure that data.

### *Coding Makes You a Star*

Imagine you're at a meeting with a project manager, IT director, graphics designer, copy editor, marketing manager, sales manager, and various subject-matter experts discussing an online application. Each role has its own area of expertise to contribute. But the only person technically capable of translating these visions into the code needed to bring it to life is the programmer.

In many organizations, the software developer is the person who brings a project together. That new marketing write-up? The programmer is the one who knows how to typeset it online. The new banner image? The programmer is the one who knows how to upload it to the server and reference it in the code. A designer wants to animate a menu? You'll need someone code-literate to create that interactivity.

Programmers must learn how entire organizations operate. This is especially true when you're coding business logic. For example, when I coded for the Coast Guard, the pilots, mechanics, and officers shared everything they knew about aviation logistics with me. While coding for food safety laboratories, I learned about pathogens, chemistry, and quality testing from PhDs. Working at an educational organization, I learned the ins and outs of social networking, marketing, and publishing.

Over time, organizations grow more dependent on a coder's wisdom as other employees leave, and co-workers must reach out to the software developer to understand how their jobs were performed. Having such a deep understanding of an organization's business processes provides the software developer with a high degree of job stability.

### *Coding and Computational Thinking Make You Efficient*

It's not just the employees with the "software developer" job title who can benefit from knowing programming. Being capable of computational thinking and knowing some programming can open professionals to opportunities to improve efficiency and automate tasks.

For example, an office assistant managing a corporate website might need to reformat the contacts page from listing first name, last name, phone, and email to instead displaying last name, first name, email, and phone. The most straightforward way to do

this is to copy and paste to rearrange the fields, but this would be time-consuming and prone to errors. An assistant who knows about string manipulation functions could simply paste the list into a spreadsheet and apply a string concatenation function to output the fields in the new format.

By doing so, this professional improves their efficiency and accuracy. They've automated a repetitive task. The act of automating this process was more technically challenging than the repetitive task, but the time saved was the reward for tackling that complexity. Automation lets us work *smarter*, not *harder*. Figure 2-2 illustrates how this works in practice.

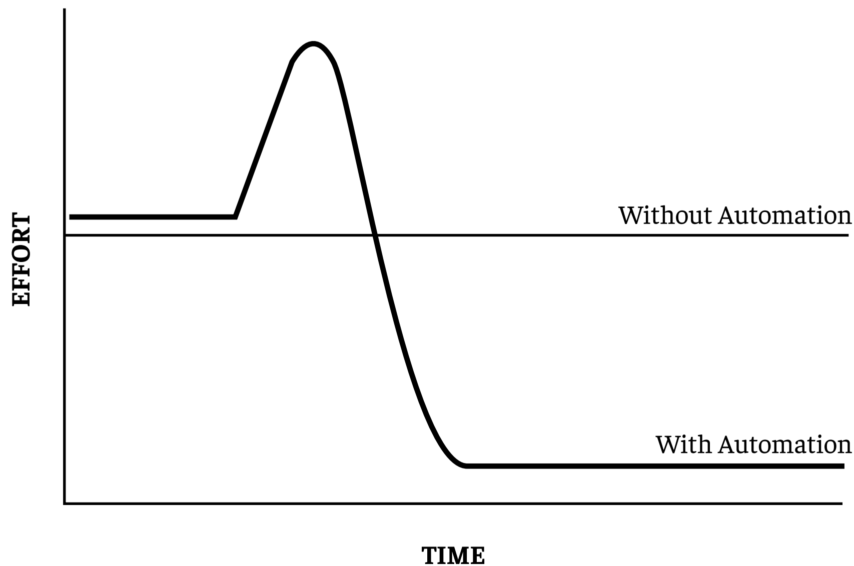


*Figure 2-2: Effort over time with and without automation*

The figure shows how the level of effort changes over time when we automate tasks. Without automation, the level of effort for a repetitive task remains the same forever. But when we decide to automate a task, we may initially be challenged with the complexity of how to automate it. There will be a learning curve and the increased stress that comes with problem-solving. Once the problem is solved, the effort and stress of performing the task drop to a minimum or even vanish.

Consider the simple example of scheduling a meeting for the fourth Tuesday of each month. The meeting organizer can look at each month of the calendar year and schedule 12 meetings. Alternatively, because they understand computer systems and can think computationally, they can use the extra effort to figure out how to schedule a recurring meeting one time and never think about it again.

## Opportunity Cost

Lots of students understand the powerful influence of digital distractions in their lives. You can use video games, social media, or binge-watching online shows as an example to illustrate the concept of opportunity cost by asking students to start thinking about their time as a finite resource.

For example, 40 hours spent watching the entirety of a show on a streaming service is 40 hours not spent on exercising, learning new skills, socializing, or earning income. This illustrates an *opportunity cost*, which means that the resources you spend on one activity are resources you can't spend on another. We must choose how to spend our resources wisely.

You can also expand on this further to show a few minutes spent more wisely each day can compound into huge benefits over time. For example, a student who reads 13 minutes a day will read 722,000 words in a year on average. A student who reads 17 minutes a day will consume about 1,168,000 words in that same time. And a student who reads 33 minutes a day will be exposed to approximately 2,357,000 words in a year. Trading just 33 minutes a day that would otherwise be spent on video games, social networking, or streaming videos to read a book for fun puts a student in the 90th percentile for amount of reading and the many academic benefits that come with it.

Have students quantify time from the obverse to see how they might optimize their time for efficiency. For instance, what if they could shave off just 30 seconds from one of their daily routines? That would add up to three hours of time saved each year. A student who gains 6.6 minutes each day on getting ahead on homework will have accumulated more than 40 hours of free time at the end of the year to spend on another season of online streaming.

Professionals who can automate their repetitive tasks and streamline their workflows using IT save time and maximize their efficiency. This increases their value to their organizations and enhances their professional lives.

## Summary

In this chapter, we explored the many ways learning computer science will benefit your students. Cognitively, students who learn computer science will develop a toolkit of problem-solving skills they can transfer to other knowledge domains. When writing code, students practice communicating in the very precise and literal way computers demand while also making their code understandable to peers and their future selves.

Writing code also exercises their working memory as they hold stacks of variables and logic in their brains to debug and enhance their programs. The act of coding exercises the student's grit or stick-to-itiveness to see their projects through to that moment of satisfaction when it finally executes successfully.

Because producing computational artifacts is PBL, computer science instruction can enhance a students' academic success as they engage in close reading, sustained inquiry, collaboration, peer review, and iterative development. Through online forums and in-class code reviews, students engage in dialogue with peers to collaborate on problem-solving and best practices. When tinkering with their code, students engage in exploration through experimentation and experience the joy of flow and discovery.

Professionally, computer science education makes your students more employable and higher salary earners. Knowing how to automate business logic makes them valuable to their employers, and their professional lives easier. As the solutions provider for their organization, they'll gain a deep understanding of the organization's business processes, making them indispensable to their employer.

Making students aware of computer science education's cognitive benefits encourages metacognition, awareness of the educational benefits makes them engaged partners in their schooling, and awareness of the subject's professional benefits provides them with incentives to succeed. Parents and administrators who are aware of these many benefits will have incentive to provide the support structures crucial to students' success in their projects.

Although all of these reasons for computer science education are pragmatic and rational, it's also important to stress the subject's humanistic side. In the next chapter, we'll take a deep dive through the history of computer science, starting with the icons on your desktop and descending through the code. We'll form a connection between technologies and people by learning about the individuals behind these innovations.