Computational Thinking: A Competency Whose Time Has Come

Chapter · December 2017

DOI: 10.5040/9781350057142.ch-003

2 authors:

Shuchi Grover

91 PUBLICATIONS **3,805** CITATIONS

SEE PROFILE

Roy D. Pea

Stanford University

264 PUBLICATIONS **18,589** CITATIONS

SEE PROFILE

3

Computational Thinking: A Competency Whose Time Has Come

Shuchi Grover and Roy Pea

Chapter outline

Chapter synopsis

Computational thinking encompasses a range of specific thinking skills for problem solving including abstraction, decomposition, evaluation, pattern recognition, logic and algorithm design. While what exactly is included in computational thinking has been the topic of some debate, this chapter will consider each of the elements of CT, how the learning of these concepts and practices can be facilitated within the school curriculum, and the role of CT skills in other domains.

3.1 Introduction

The United States presidential election of 2016 was one for the ages. Beyond the theatrics and intrigues, it also spotlighted the undeniably growing role computing is playing in momentous geopolitical events. Whether it was AI's coming of age in the form of chatbots to influence (for better or worse) critical news and information or the e-mail server issue that dogged one major political party, the election underscored the need for the citizenry to be better equipped to understand the fundamentals of computing and engage in computational thinking. When a federal agency announced the examination of 650,000 e-mails over nine days, individuals across mainstream and social media and on the election stump, pondered:

> A couple of questions are probably running through many Americans' minds right now: Is it possible to 'review' 650,000 emails in just eight days? Does the FBI recruit superhuman speed readers to process and internalize vast amounts of information? Is the FBI lying?
>
> Yasmeh, 2016

In contrast, observers familiar with computing and automation understood this as a problem readily solvable by computing. The minimal press that attempted to explain 'de-duping' and how enormous quantities of data, such as e-mails could, in fact, be examined efficiently and effectively using programs searching for specific strings and patterns, did little to alleviate the suspicions of sceptics unfamiliar with how computation works.

The twenty-first century is arguably the century of computing. Artificial intelligence has finally come of age as it becomes embedded in the transformation of work, commerce and everyday life. Big data, speech and facial recognition, robotics, internet of things, cloud computing, autonomous vehicles and 24×7 access to anyone, anywhere in the world via social media is changing how and where people work, collaborate, communicate, shop, eat, travel, get news and entertainment, and quite simply, live. Computing is also transforming industry and innovation in every discipline, becoming an integral tool that is spurring new ways of doing and thinking. In such a world saturated by computing, 'Computational Thinking' or CT (Wing, 2006, 2011) is now recognized as a foundational competency for being an informed citizen and being successful in STEM work, one that also bears the potential as a means for creative problem solving and innovating in all other disciplines. In this decade, systematic endeavours have gained momentum to take computer science (CS) education and CT to scale in K-12 classrooms in states across the US and internationally. In 2012, the UK National Curriculum programme began introducing CS to all students at all class levels. US efforts found heightened legitimacy with the 2016 Presidential mandate of 'Computer Science For All' initiated in collaboration with federal funding agencies, academic research institutions, professional associations, prominent industry partners and non-profit organizations. In his call supporting CS education, President Obama echoed the beliefs of many in the education community with his assertion that all children from kindergarten to high school need to learn CS and be equipped with computational thinking skills they need to participate in our technology-driven world (Smith, 2016).

3.2 What is computational thinking?

We have witnessed over the last two decades a concerted paradigm shift (Kuhn, 1970) in our beliefs of what is important to learn not only in STEM subjects but also in the humanities. This shift privileges teaching higher-order critical thinking abilities fundamental in each and every domain beyond rote learning and procedural skills, in what has been designated as 'deeper learning' (Pellegrino and Hilton, 2013). Nationwide US efforts around the Common Core standards for subjects such as mathematics and English language, and the Next Generation Science Standards (NRC, 2013) mirror similar shifts in other countries which emphasize disciplinary thinking and ways of knowing and being beyond rote learning. So, teaching mathematics has moved towards thinking like a mathematician; science learning now involves developing competencies for thinking like and enacting the authentic practices of, a scientist.

It seems only logical, then, that educators and policy makers keen to teach computer science are attempting to privilege computational thinking or thinking like a computer scientist, over other aspects of computing (such as learning binary arithmetic). What is 'computational thinking' anyway? How is it defined and understood?

Jeannette Wing's definition

Though somewhat opaque, Jeanette Wing's definition of computational thinking first articulated in a 2006 *Communications of the ACM* article (Wing, 2006, 2011) deserves mention for capturing the collective imagination of educators and researchers worldwide and spurring global efforts to create a generation of computational thinkers. She uses *'computational thinking' as shorthand for 'thinking like a computer scientist'*.

Key concept: Computational thinking

Computational thinking is the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer – human or machine – can effectively carry out.

Informally, computational thinking describes the mental activity in formulating a problem to admit a computational solution. The solution can be carried out by a human or machine. This latter point is important. First, humans compute. Second, people can learn computational thinking without a machine. Also, computational thinking is not just about problem solving, but also about problem formulation (Wing, 2014).

Computational thinking is fundamentally about problem solving using concepts and strategies most closely related to computer science (hence its name). Problem formulation should be considered a key part of this problem-solving process. Since formulating the solution for the problem using CT need not involve the computer, even though the execution of the solution usually

does, CT can be taught without the use of the computer. K-12 educators now aspire to teach these skills, with and without the computer, in ways that equip students to apply them in various contexts and domains, and more often than not, where a computer or computing device must carry out the solution. This is somewhat of a shift from early views of CT promulgated by Papert (1980), whose pioneering work in children and programming continues to inspire student-centred, constructionist CS curricula and pedagogies even today.

Given the computing-saturated direction in which the world is moving, CT is thus especially relevant as a widely applicable thinking competency along with other critical thinking needed to solve the challenges posed in this century throughout various domains. CT played a key role in changing the course of World War II when computing pioneer Alan Turing used it to break the code of The Enigma Machine underlying the Nazi war efforts (Hodges, 2012). More recently, CT made its most compelling case for deserving serious attention when, at the dawn of this century, it was credited with cracking the human genome, touted to be among the thorniest problems that the biomedical community had struggled with for decades. Futurists believe that CT will similarly play a role in tackling thorny issues with which current and coming generations will have to contend, such as climate change, shortages of critical resources such as water and social inequities that continue to plague societies around the world.

Key concept: What CT is not!

It is easy to fall into the trap that CT is thinking like a computer. Yet it is a trap conveniently avoided if one keeps in mind our framing of *'thinking like a <domain expert>'* for <domain-specific> thinking competencies. Thinking is an inherently human trait that involves reasoning. Computers do not think, so CT is NOT 'thinking like a computer', rather it is about *thinking like a computer scientist*. It's the problem-solving approaches commonly used by computer scientists that constitute computational thinking.

What are these problem-solving approaches and ideas that CS embraces? Let's look at these next.

3.3 Elements of computational thinking (Breaking it down)

What does CT mean? What thought processes does it involve? Obtaining answers to these queries will help teachers and designers to develop curricula to prepare children's computational thinking competencies. Jeanette Wing's article and subsequent efforts to define CT – especially for K-12 education – spawned a large body of articles breaking CT down into several elements that aimed to clarify and outline what 'thinking like a computer scientist' means, including our CT 'state-of-the-

field' review in AERA's *Educational Researcher* (Grover and Pea, 2013).[1] All these elements comprise some combination of a list of competencies most will agree are facets of the thinking processes that computer scientists engage in when they solve problems. Keeping in mind that there is not yet an unassailable list, the elements that we find to be most comprehensive and useful to describe CT to teachers, with a few of our own tweaks, is that outlined by the British Computing At School initiative.

By observing what kinds of thinking computer scientists activate when they engage in problem solving, we find that CT encompasses the following *concepts* and *practices*. The inclusion of the practices view of CT in addition to CT concepts, is in keeping with the *'thinking like a <domain expert>'* notion and describes the behaviors that domain experts engage in in the field.

Key concept: CT concepts and practices

CT concepts include:

1. Logic and logical thinking
2. Algorithms and algorithmic thinking
3. Patterns and pattern recognition
4. Abstraction and generalization
5. Evaluation
6. Automation

CT practices include:

1. Problem decomposition
2. Creating computational artefacts
3. Testing and debugging
4. Iterative refinement (incremental development)
5. Collaboration & Creativity (part of broader twenty-first century skills)

We now describe each of these along with examples set in everyday non-computing contexts as well as computing and programming contexts. Where possible, we also provide a simple example or two of how teachers might teach these concepts and practices in the classroom.

3.4 CT concepts

Logic and logical thinking

Logical thinking involves analysing situations to make a decision or reach a conclusion about a situation. An everyday example of logical and analytical thinking might involve analysing whether

[1]Among the other more influential articles breaking down CT include the 'working definition' by ISTE, Google's 'Exploring Computational Thinking' (available at: https://goo.gl/eUBk7v); the UK's 'Computing at School' initiative and the APCS Principles' six CT Practices and seven Big Ideas of Computing.

it is worthwhile going to *Shop A* to buy a dress for $30 or *Shop B* where it's available for $20, taking into account factors such as distance to the shops, the time of day and weather. It may not make logical sense to go to *Shop B* if it is farther away than *Shop A* and the cost of travelling to *Shop B* is greater than the $10 difference in price of the dress.

Computer scientists also often use more of a formal logic framework in their work. *Boolean logic* is at the heart of all computing from computational circuitry to its use in software and programming to make decisions in flow of algorithmic control. As part of CT competency development, students must build analytical thinking skills by working on logical puzzles and problem-solving scenarios as well as learning formal Boolean logic through an understanding of AND, OR, NOT (and other variants of Boolean operators) and how to construct Boolean expressions using combinations of these primitive logic elements.

Example activity: Boolean expressions set in real-world settings

A simple example of logic thinking might involve constructing a Boolean expression for an alarm that would ring for soccer practice on Mondays at 4pm and Wednesdays at 5pm, like so:

SoccerAlarm rings **IF** ((WeekDay is Monday **AND** Time is 4pm) **OR** (WeekDay is Wednesday **AND** Time is 5pm))

Being able to reason thus with Boolean logic also translates well to game programming when games require the use of control statements involving Boolean expressions, such as, '*Game over if the player has collected all the gold coins or has no more lives left*'.[2] Of course, the program will require an additional logical check to determine whether to announce *'You won!'* or *'You lost!'* before it ends.

Algorithms and algorithmic thinking

Algorithms are precise step-by-step plans or procedures to meet an end goal or to solve a problem; algorithmic thinking is the skill involved in developing an algorithm. Cooking recipes are a common everyday example of algorithms (albeit less precise than what would be considered algorithms in computer science). Other common examples are route maps suggested by applications such as Google Maps or instructions for assembling a piece of furniture, instructions for knitting or crocheting a scarf, and so on. In fact, the precise set of actions to get ready for school every morning could be construed as an algorithm.

Most of the algorithms that students encounter in the context of K-12 CT learning involve three basic building blocks – *sequence, selection* and *repetition*. The steps in all algorithms follow a sequence; however, there could be conditional checks that make the algorithm select either one of another, or set(s) of actions that are repeated. In the context of programming, conditional checks

[2] Several examples drawn from Grover, Pea and Cooper (2015), Grover (2017), and Grover and Basu (2017).

involve the *if-then-else* constructs, and looping involves constructs such as *do-while*, *for*, *repeat*, or *repeat-until* to perform repetitive actions. Recursion is a related idea to repetition that is a technique unique to computer science although it is a more advanced idea and relevant mostly to curricula in high school and beyond. Grover, Pea and Cooper (2015) describe a structured middle school curriculum aimed at teaching algorithmic thinking skills.

Computer scientists use this concept of algorithms to devise precise solutions to problems. These solutions could be described in the form of flowcharts, pseudo-code or a bulleted list written in an abstract everyday language that could then be coded or programmed (by the same computer scientist who creates the algorithm or by other programmers) using a programming language to be interpreted and carried out (or 'executed') by a computer. As with logic, disciplinary learning in computer science at the undergraduate level also involves a more formal study of algorithms that students may not encounter in their K-12 CS learning, involving examining aspects of efficiency, resource optimization and complexity of algorithms.

Patterns and pattern recognition

We are all familiar with the concept of patterns and pattern recognition from our early learning of shapes or math topics such as multiplication and number series completion. Computational thinking includes these ideas of pattern recognition and extends the idea to problem-solving settings. Pattern recognition in CT could lead to the definition of a generalizable solution (which also has overlaps in maths) that can leverage automation in computing for dealing with a generic situation, for example any *Step n* of a series no matter how large *n* gets. Recognizing a repeating pattern also informs how to incorporate iteration or recursion in an algorithmic solution or a functional breakdown of a problem (that also serves the cause of creating manageable and modular solutions). CT also leverages pattern recognition by examining what parts of a problem are similar to something one has already solved (or programmed) before. This is the bedrock of the powerful idea of design patterns or programming paradigms in software development.

In addition to these basic ideas of pattern recognition, computer scientists have advanced more formal use and understandings of the idea of pattern recognition in topics such as machine learning and artificial intelligence that focus on recognizing patterns in data. Pattern recognition is used in computer vision algorithms for recognizing images and faces (recall how Facebook is able to automatically 'recognize' and tag a face?) or for recommending products on Amazon, your next article on a news site or your next song using iTunes 'Genius'.

Abstraction and generalization

There is broad consensus that abstraction is the keystone of computer science (and consequently, computational thinking). Abstraction, however, is a general property of symbolic systems that has been leveraged in arithmetic, algebra and other mathematical sub-domains for centuries.

Jeanette Wing refers to abstraction as the most important and high-level thought process in CT. It is related to several elements of CT described above. Simply put, abstraction is 'information

hiding'. The act of 'black-box'-ing details allows one to focus only on the input and output. In this sense, then, abstraction provides a way of simplifying and managing complexity. It is also the ability to generalize based on similarities and differences. CT involves knowing the right types of abstractions to create and use in a computation solution.

K-12 education should strive to provide children with a sense of how computers and programming languages are also abstractions. Though the computer is a complex, physical machine made up of circuits and wires, as users of computers we interact with it through sophisticated operating systems and applications. Even computer application software developers don't need to think in terms of the physical circuitry. Programming languages used by software developers represent an abstraction of the computer that understands the constructs and keywords used in that language. These higher-level languages hide the complexity of performing operations in the more primitive instructions that are used in lower-level programming languages and ultimately the lowest level 'machine language'. A developer or engineer at any stage typically needs to know only how to interact with one level below and what is to be seen by the next higher level. Every algorithm is also an abstraction as is every model or simulation that represents some real-world phenomenon. Every procedure defined within a program that stands for a set of instructions is also an abstraction. Data, stored in variables and data structures in programs, is the abstract 'stuff' procedures or programs act on and manipulate. They are abstract because they encapsulate and hide the details of the physical things they represent. In this important respect, computer programs are akin to more familiar algebraic equations, which also hide the details of the physical things represented in algebra equations by their variables and values.

Abstraction, admittedly, is among the more abstract ideas to teach and assess as part of a CS/CT curriculum. It is *so* pervasive, however, that it pops up nearly everywhere. As Wing puts it:

> Abstraction is used in defining patterns, generalizing from specific instances, and parameterization. It is used to let one object stand for many. It is used to capture essential properties common to a set of objects while hiding irrelevant distinctions among them. For example, an algorithm is an abstraction of a process that takes inputs, executes a sequence of steps, and produces outputs to satisfy a desired goal. An abstract data type defines an abstract set of values and operations for manipulating those values, hiding the actual representation of the values from the user of the abstract data type. Designing efficient algorithms inherently involves designing abstract data types.
>
> Wing, 2014

Evaluation

Evaluation goes hand-in-hand with several of the elements of CT described above. Solutions to problems in the form of algorithms or abstractions in the form of programs, models or simulations must be evaluated for correctness and appropriateness based on the goal as well as constraints. While it involves analysis and analytical thinking, the idea of evaluation is grander. Solutions to problems are *evaluated* for accuracy and correctness with respect to the desired result or goal. There are often other grounds for evaluation. Think of the algorithm that provides directions. It could be evaluated based on any of several criteria – shortest, fastest, most scenic or other constraint such as: *Does it take*

you past a grocery store or gas station where you may need to make a stop? Computer scientists dealing with complex problems and algorithms often evaluate their solutions based on efficiency constraints such as time to completion, resource usage and human factors or user experience considerations.

Automation

'Computing is the automation of our abstractions' (Wing, 2008, p. 3718). A key part of computational thinking, for computer science as well as computing in other domains, is working towards a solution that will be executed by a machine. Automation as a rationale to address a need that cannot be solved otherwise is often the motivation for using CT for problem solving in the real world. In such instances, recognizing when automation is needed and what abstractions and data representations will best help develop an automated solution is a key part of CT.

At the K-12 level, even though the end goal of applying CT is not always a computational solution implemented on a machine, it is important for learners to develop an understanding of when automation is the answer to the problem – what aspects of problems are better solved by humans and which are better solved by the machines.

3.5 CT practices

The CT practices described below outline *approaches* that computer scientists often use when they engage in computational problem solving.

Problem decomposition

This approach is not unique to computer scientists. It is suggested in Polya's (1957) seminal work on problem solving in the context of mathematics. Such a method for problem solving was enumerated as one of the rules for right thinking by Rene Descartes (1637) in his Discourse on Method: 'divide each of the difficulties under examination into as many parts as possible, and as might be necessary for its adequate solution'. It is, however, a key approach in computational problem solving. Breaking a problem down into smaller sub-problems makes the problem more tractable and the problem-solving process more manageable. Examples abound in everyday life. Getting ready for school or work usually involves getting cleaned up, getting dressed, having breakfast, packing lunch or a snack, and ensuring you have the right contents in your bag as you leave. Each of these sub-tasks contains its own set of actions, is independent of the other and often happens in the same sequence every day. Going back to the algorithmic process of cooking and the recipe as an algorithm, one can easily see problem decomposition at play when the recipe separates the pre-preparation process of marinating or getting the ingredients together, from preparing some portion of the dish (e.g. the dressing or gravy) and the preparation of the main dish and *then* combining it with some post-processing steps (cooling, garnishing and such) to get the dish ready for serving.

In the context of programming, the task of breaking down a problem often leads to pieces of code being written separately. These component parts of the program need to 'come together' when the whole solution is composed. This process is simple when the different sub-problems are independent of each other. Take, for example, the task of calculating the average score of an exam. The first sub-problem could involve asking for user input and creating a list or array of scores; the second could address traversing the list and adding up all the scores in some aggregator variable; and the final step could simply involve calculating the average by dividing the total of all scores by the number of student scores (after appropriately performing a divide-by-0 check!).

But the process of composing a solution or 'plan composition' is not always as straightforward for novice programmers when the sub-problems are intertwined or connected in some way. This was famously demonstrated by Elliot Soloway's 'Rainfall problem' (Soloway et al., 1983) where students were tasked with the following problem: 'Write a program that repeatedly reads in positive integers, until it reads the integer 99999. After seeing 99999, it should print out the average.' Here, the tasks of reading an input value, checking the input for the sentinel value and whether it is negative are intertwined with aggregating the values. This problem has repeatedly been shown to trip up novice programmers.

This example suggests that computational thinking requires not only the skill to decompose a problem but also to compose the solution after the sub-problems have been addressed.

Creating computational artefacts

Wing's definition suggests that the goal of computational thinking is to solve problems that can be executed by humans or computational devices. While several examples of computational thinking described above are situated in the real world and do not involve a computer, creating solutions to be executed by a computer is often a natural end goal of computational thinking and problem solving. Sometimes, the computational artefact is merely a simulation or model or interactive prototype of something that will eventually be a physical artefact; at other times the computational artefact is itself the end goal – a game or story or artefact of creativity and personal expression or software that could be used by others.

Programming is therefore seen as an especially useful platform for teaching CT since it brings together several of the elements – both concepts and practices – that are central to CT. In Grover and Pea (2013), we asserted, 'Programming is not only a fundamental skill of CS and a key tool for supporting the cognitive tasks involved in CT but a demonstration of computational competencies as well.'

Even so, it is important to observe that CT involves problem solving and thinking competencies that can be invoked in settings outside of programming. Programming, although important, cool, interesting and fun, is but one of the possible vehicles for developing CT competencies. The current rush to focus on coding often attracts attention towards the features of the programming environment and away from the important aspects of computational thinking that must be involved. Often these 'low-floor' programming environments allow for tinkering without the mindfulness and meta-cognition called for by deeper learning. This is akin to learning the syntax

of *a specific programming* language (what the constructs mean and accomplish) without a deeper appreciation for the deeper CT concepts and practices that equip learners with the competencies to be used in *any programming* context, whatever its specific features.

Testing and debugging

Testing and debugging are integral to any kind of problem solving (Miller, Galanter and Pribram, 1960). Evaluating one's solution for accuracy, detecting flaws in a faulty solution and fixing them, is part and parcel of any problem-solving process, such as hammering in a nail to be flush with the surface of the wood. We're all constantly 'debugging' all kinds of problems at hand, from tasting the amount of salt and spice in a dish (some problems are harder to fix even if you've identified the bug!) to proofreading our essays and e-mails to fix typos and grammar errors.

Like other CT concepts and practices, testing and debugging are related to many of the other elements described here. They are part of the process of evaluating a computational solution – whether it satisfies relevant rules and assumptions, whether the solution works for boundary conditions and all relevant inputs and situations, and whether it acts as expected for illogical or erroneous inputs. This also involves logical and 'if-then' analytical thinking to isolate the problem and zero in on the error. It is also integral to the incremental development and problem decomposition strategies described above.

Rigorous, systematic testing and debugging is an art and science in computing, and especially in software development. Developing test cases and taking the software through its paces is a significant part of the software development process, and it is a process that itself can become automated. In the context of programming, systematic testing the solution for correctness for the range of valid and invalid inputs is an integral competency in learning to program. It is worth noting that, despite the surface similarity with the scientific process, testing and debugging is far more pronounced in CT than, say, scientific thinking where one is evaluating the results and evidence from an experiment and revising the hypothesis as necessary.

Incremental development (or iterative refinement)

This is a very common strategy used in the context of programming. Though similar to the process of problem decomposition, it focuses not so much on the idea of decomposing the problem into sub-problems, as it does on 'growing the solution or program' iteratively with frequent testing and debugging in between to develop improvements. This is contrasted with – and preferable to – writing large chunks of code that make it difficult to isolate the bug(s) if the solution does not work as intended. The most frequently used avatar of this approach in professional software development circles goes by the moniker 'agile development' (Martin, 2003).

Consider a simple example from robotics. Imagine a roving bot that needs to turn around when it hits an obstacle. The student could first simply code and test the movement – have the robot go forward in a straight line. Then s/he adds the obstacle collision test by making the motors stop when the touch sensor indicates a collision. Once this is tested and found to be working, the student

would then add the reverse-and-turn-upon-collision code instead of simply stopping. This kind of incremental growth of the solution can be contrasted to coding all the pieces all at once and then testing. Imagine how unmanageable it can all become if the problem is more complex!

Collaboration and creativity

A couple of other elements, though not considered part of CT in earlier definitions of CT, are often described as common practices in computational problem solving. These include collaboration and creativity. Both are acknowledged as critical competencies for a new century, but they do also have a special meaning as CT practices and in the world of computer science. Collaboration is often fostered in K-12 computing classrooms through 'pair programming' (Williams and Kesseler, 2002) a practice that is increasingly popular in industry. The norms of collaboration in pair programming require programmers to alternate between taking the lead on typing or reviewing code and have been shown to be beneficial to problem-solving processes. There are other forms of collaboration that are unique to CS. The division of development tasks in software engineering is necessitated by CT practices such as problem decomposition and modularization. Parallel computing has also led to the division of computing tasks and is at the heart of globally important programming paradigms such as MapReduce used for many years at Google. Other interesting forms of collaboration in the world of CS include the use of Github to build on one another's work in projects, crowdsourcing computer games to advance a scientific agenda (as in FoldIt and Xylem) and collaborative software development as part of the free and open source movement that led to the creation of Linux and other systems.

Creativity as a CT practice acts on two levels – it aims to encourage out-of-the-box thinking and alternative approaches to solving problems; and it aims to encourage the creation of computational artefacts as a form of creative expression. Block-based 'open-ended' introductory programming environments such as Scratch, Alice and App Inventor have been developed with the goal of teaching creative coding and motivating learners as a conduit for teaching CT, especially in K-12 settings.

Fostering CT in the classroom and CT across subjects

Key concept: Programming and CT

Is programming the only way to foster CT? The answer to that question is a resounding 'No!' While we have many articles and discussions about the relationship between programming and computational thinking, there is little debate that computational thinking is about more than codifying a solution for execution by a computer through programming and that it is the loftier, more worthy goal of CT that we must strive to achieve through appropriate pedagogies even when students are engaging in programming. In fact, some believe that the typical design challenges and

design tools of modern computing involve little coding (Tedre and Denning, 2016). At the same time, few would challenge the proclamation that programming is an important – and engaging – vehicle for learning and applying (and hence teaching and assessing) CT. Few other activities involve as many concepts and practices of CT as does programming.

Nonetheless, it is abundantly clear from the description of CT concepts and practices that there *are* other ways of fostering CT. Analytical and logical thinking can be fostered through puzzles and word problems that require learners to engage in this crucial aspect of CT.

Problems involving such logical argumentation and thinking can be tackled in language arts, mathematics classrooms or CS classrooms.

Example activity: Logical thinking in the language arts classroom

Here is a simple problem involving logical thinking described in Grover (2009) an early ISTE article on CT ideas for teachers –

- If the Giants beat the Dodgers, then the Giants win the pennant.
- If PlayerX is out, then the Giants beat the Dodgers.
- PlayerX is out.

What is the conclusion?

Example activity: Logical thinking in the maths classroom

In a game, exactly six inverted cups stand side-by-side in a straight line. Each has exactly one ball hidden under it. The cups are numbered consecutively 1 to 6. Each of the balls is painted a single solid colour. The colours of the balls are green, magenta, orange, purple, red and yellow. The balls have been hidden under the cups in a manner that conforms to the following conditions:

The purple ball must be hidden under a lower-numbered cup than the orange ball.

The red ball must be hidden under a cup immediately adjacent to the cup under which the magenta ball is hidden.

The green ball must be hidden under cup 5.

Which of the following could be the colours of the balls under the cups, in order from 1 to 6?

(A) Green, yellow, magenta, red, purple, orange
(B) Magenta, green, purple, red, orange, yellow
(C) Magenta, red, purple, yellow, green, orange
(D) Orange, yellow, red, magenta, green, purple
(E) Red, purple, magenta, yellow, green, orange.

Currently, computational and algorithmic thinking problems such as these are found mostly in competitions such the Enigma Computational and Algorithmic Thinking contest run by Edfinity along with the Australian Math Trust[3] and Bebras,[4] but there is no doubt that students would be well-served by tackling such non-programming puzzles and problems as part of CT competency building.

There are several ways of encouraging algorithmic thinking practices in learners that involve articulating precise step-by-step procedures – storyboards, an ordered set of sentences, pseudo-code, flowcharts and the like. Even in the context of programming, expressing an algorithm in such ways *before* coding it into a programming language to be executed by a computer, is a well-established and recommended practice. One fun exercise used in the context of robotics involves writing a set of detailed steps in plain English to verbally guide a blindfolded student partner to perform a certain task. Ideas of exception handling, iterations and conditional actions could be woven into this fun exercise.

To learners, practicing these computational thinking concepts and approaches in contexts outside programming signals the importance of the CT and problem-solving process rather than simply codifying the solution in the syntax of a programming language. Grover, Pea and Cooper (2015) describe an example of a curriculum focusing on deeper, transferable learning of algorithmic thinking skills using a pedagogy that incorporates various pedagogical ideas from the learning sciences, in addition to assessments that cover cognitive as well as affective dimensions of deeper learning.

3.6 CT within and across subjects

It is reasonable to argue that it is in all the contexts outside of CS classrooms that CT truly shines with its generativity. From music, maths, social studies, history, language arts and throughout the sciences and engineering, curricular ideas can come alive with CT. Just as in disciplinary research in each of these fields, where computational thinking advances both everyday practice and its innovations, there is a role for creativity in curriculum design and teaching of other subjects through the integration of CT in those classrooms, while also providing rich and varied contexts for developing CT competencies.

[3] See https://edfinity.com/competitions
[4] See www.bebraschallenge.org/

Key concept

Computing and STEM share a deeply symbiotic relationship, and as such, mathematics and science classrooms provide perhaps the most intuitive and easy non-CS contexts for CT learning and use. The use of computational tools to enable deeper STEM learning has been shown in numerous prior research studies, and the reverse has also been shown to be true. STEM can enrich computational learning while also providing valuable opportunities to embed CT in established and accessible (as well as required) STEM courses. This latter point is especially relevant as many states and countries worldwide still struggle with providing adequate computing experiences to all students as part of K-12 education. Based on successful investigations in prior research, computational modelling and simulation are concrete mechanisms for integrating computing and STEM, and can benefit the learning of both the STEM content as well as the development of CT skills in such an emphasis (e.g. Honey and Hilton, 2011).

CT through STEM can also address the issue of teacher shortages, as preparing STEM teachers to use CT within the context of the disciplinary content areas with which they are familiar already may be more practical and amenable to scaling implementation of CT in school education.

Bringing CT into STEM classrooms will also better prepare students for the modern landscape of the STEM disciplines. This view is endorsed by the Next Generation Science Standards, calling for the use of mathematics and CT in science. We can call this the 'CT-in-STEM-Learning' research agenda, and it would need to seek answers to several key questions, including, 'What CT competencies are most important for various STEM disciplines?', 'How can learners best develop these competencies through their learning experiences with STEM coursework curricular units?' and 'How can we collaborate with STEM teachers towards their effective appropriation and uses of computational thinking methods and approaches in their curriculum?' Efforts to bridge CT and STEM in K-12 science, have centred mostly on building computational models and simulations to understand and study phenomena in science (e.g. Hansen et al., 2015; Sengupta et al., 2013; Sherin, 2001; Wilensky, Brady and Horn, 2014) and have shown much promise. These efforts can also be informed by substantial work from the mathematics education research community in their curricular studies of the functional mathematical thinking involved in real-world problem-solving processes and mathematical modelling by students and teachers (e.g. Burkhardt, 2006; Lesh and Lehrer, 2003; Lesh and Zawojewski, 2007). Growing the knowledge base on how best to effect the integration of CT and STEM has been called out as one of the imperatives for computing education research (Cooper et al., 2014).

The role of CT in non-STEM subjects such as music, social sciences, visual arts, language arts, history, is manifold. Barr and Stephenson (2011) and Barr, Harrison and Conery (2011) outline examples of what this integration might look like, as does Google's Exploring Computational Thinking. Examples of documented efforts for meaningfully integrating CT and non-STEM subjects can be seen in Repenning, Webb and Ioannidou (2010), Settle et al. (2012), and Wolz et al. (2010), among others.

Fablabs, making and computational crafts also open a whole world of possibilities of CT development in the context of art and craft that involves creating tangible computational artefacts. Michael Eisenberg's Craft Tech lab[5] and Leah Buechley's [6] innovative tool designs including the Lilypad Arduino[7] for e-textiles and 'sketch' electronics using microcontrollers and conductive ink have been found to reach audiences beyond those that robotics clubs and competitions typically attract.

3.7 Summary

In a world infused with computing, computational thinking is now being recognized as a foundational competency for being an informed citizen and being successful in all STEM work, and one that also bears the potential as a means for creative problem solving and innovating in all other disciplines. The roots of CT in education date back to Papert's work in the 1980s that centred on children developing thinking skills through programming computers. Recent efforts on bringing CT to school education, while still inspired by that early work, is informed by Wing's 2006 definition and call to action. Definitions and elements of CT have been broadened in this last decade to include aspects of collaboration and creativity.

Computational thinking is defined as the set of the thinking skills used by computer scientists to address a broad range of problems in computing and other domains. Learning CT, much like learning scientific and mathematical thinking, is more about developing a set of problem-solving heuristics, approaches and 'habits of mind' than simply learning how to use a programming tool to create computational artefacts. That said, programming is a key vehicle for teaching, learning, expressing and assessing CT that is unarguably also deeply engaging for students in K-12 classrooms.

Much like recent movements in science and maths that have adopted a practices-view to STEM learning, the key elements of CT are broken down into concepts and practices. CT concepts are commonly believed to include logic and logical thinking; algorithms and algorithmic thinking; patterns and pattern recognition; abstraction and generalization, evaluation and automation; whereas CT practices include problem decomposition, creating computational artefacts, testing and debugging, iterative refinement (or incremental development). Collaboration and creativity, now seen as cross-cutting skills for the twenty-first-century learner, are also viewed as CT practices that often acquire a unique flavour in the context of CT.

It is reasonable to argue that it is in contexts outside of CS classrooms that CT truly shines with its generativity. As Denning (2017a) points out, CT emerged from within the scientific fields – it was not imported from computer science. Computing and STEM share a deeply symbiotic relationship, and as such, mathematics, science and engineering classrooms provide perhaps the most intuitive contexts for CT learning and use. Computational modelling and simulation are

[5] See http://l3d.cs.colorado.edu/~ctg/Craft_Tech.html
[6] See http://leahbuechley.com
[7] See www.arduino.cc/en/Main/arduinoBoardLilyPad

concrete mechanisms for integrating computing and STEM, and can benefit the learning of both the STEM content as well as the development of CT skills in such an emphasis (e.g. Honey & Hilton, 2011). The role of CT in non-STEM subjects such as music, social sciences, visual arts, language arts and history, is promising but as yet underdeveloped.

The thoughts and ideas reflected in this chapter present the current dominant framing of CT in the K-12 school education context. We adopt the disciplinary view of thinking lens that is driving new ways of teaching and learning across all subjects. However, it is by no means the last word on this evolving topic in education and fertile field of inquiry in education research.

Key points

- Computational Thinking (CT) is a key twenty-first century skill that helps students both to understand and take advantage of computing in various domains.
- Learning CT is about learning to think like a computer scientist – developing a specific set of problem-solving skills that can be applied in any domain to creating solutions that can be executed by a 'computer' (machine or human).
- Elements of CT include concepts such as logic, algorithms, abstraction, pattern recognition, evaluation and automation. It also includes practices such as problem decomposition, creating computational artefacts (usually through programming), testing and debugging, and iterative refinement. Collaboration and creativity are broader twenty-first century competencies that take on a special flavour in the context of CT.
- Although programming is a key vehicle to teach and learn CT, it can be taught in the classroom with or without a computer or programming.
- Bringing CT into STEM classrooms will also better prepare students for the modern landscape of the STEM disciplines; computational modelling and creating simulations are concrete mechanisms for integrating computing and STEM.
- The role of CT in non-STEM subjects such as music, social sciences, visual arts, language arts and history, is promising but still underdeveloped.

Further reflection

- Critics of the current movement to introduce CT warn against falling into the trap of assuming that CT will help learners build thinking skills that can be transferred to other domains. Both, the critics as well as the those who make the claim against and in support of transfer, ignore something we learning scientists know well. *Transfer of learning across contexts does not happen automatically*.

Pea's own research in the 1980s showed that students who were programming in LOGO did not automatically do well in problem-solving situations in maths or in planning route scheduling. The learning sciences advocate that transfer needs to be mediated through empirically established techniques that call for, among others things, making explicit connections between the original and transfer learning contexts. For example, in past work our classroom intervention included explicit mechanisms to mediate for, and assessing transfer from block-based to text based programming (Grover, Pea and Cooper, 2014).

- Denning (2017a, 2017b) urges the CS/CT movement in K-12 education not to lose sight of the fact that CT emerged from within the scientific fields – it was not imported from computer science. Indeed, computer scientists were slow to join the movement. He goes on to argue that to use CT productively in science domains one also needs the ability to design computations. *Computational design* is a better term to design the skill set than computational thinking. This view closely aligns CT to the domain of computational science. Does CT have much of a role besides computational science? We tend to believe so, however it is hard to ignore that evidence for good examples outside science are less abundant.
- Lastly, there are some who believe that CT, if broken down into elements as described and taught through unplugged or non-programming means, will be reduced to learning thinking skills that will not necessarily translate into the abilities necessary to create computational solutions and apply CT, in various domains as per the promise. In order to learn and apply CT, students need to be working with abstractions and thinking about general solutions along with other concepts such as patterns, logical and algorithmic thinking. Writing a cooking recipe alone, although an example of algorithmic thinking is not going to translate into providing learners the ability to develop computational solutions at the level of rigour that K-12 educators of CT and CS aim for their students.

References

Barr, V and C Stephenson (2011). 'Bringing Computational Thinking to K-12: What is Involved and what is the Role of the Computer Science Education Community?' 2(1) *ACM Inroads* 48–54.

Barr, D, J Harrison and L Conery (2011). 'Computational Thinking: A Digital Age Skill for Everyone' 38(6) *Learning & Leading with Technology* 20–23.

Burkhardt, H (2006). 'Modelling in Mathematics Classrooms' 38(2) *Zentralblatt für Didaktik der Mathematik* 178–195.

Cooper, S, S Grover, M Guzdial and B Simon (2014). 'A Future for Computing Education Research' 57(11) *Communications of the ACM* 34–36.

Denning, PJ (2017a). 'Remaining Trouble Spots with Computational Thinking' 60(6) *Communications of the ACM* 33–39.

Denning, PJ (2017b). 'Computational Thinking in Science' 105(1) *American Scientist* 13.

Descartes, R (2006). A Discourse on Method: 1637. Pomona Books.
Grover, S (2009). 'Computer Science: Not Just For Big Kids' 37(3) *Learning and Leading with Technology* 27–29. International Society for Technology in Education.
Grover, S (2017). 'Assessing Algorithmic and Computational Thinking in K-12: Lessons from a Middle School Classroom' in P Rich and CB Hodges (eds), *Emerging Research, Practice, and Policy on Computational Thinking* (Boston, Springer International Publishing) 269–288.
Grover, S and S Basu. (2017). 'Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic' in *Proceedings of the 48th ACM Technical Symposium on Computer Science Education (SIGCSE '17)* Seattle, WA. ACM.
Grover, S and RD Pea (2013). 'Computational Thinking in K-12: A Review of the State of the Field' 42(1) *Educational Researcher* 38–43.
Grover, S, R Pea and S Cooper (2014). 'Expansive Framing and Preparation for Future Learning in Middle-School Computer Science' in *Proceedings of the 11th International Conference of the Learning Sciences,* Boulder, Colorado. ACM.
Grover, S, R Pea and S Cooper (2015). 'Designing for Deeper Learning in a Blended Computer Science Course for Middle School Students' 25(2) *Computer Science Education* 199–237.
Hansen, AK, A Iveland, H Dwyer, DB Harlow and D Franklin (2015). 'Programming Science Digital Stories: Computer Science and Engineering Design in the Science Classroom' 53(3) *Science and Children* 60–64.
Honey, MA and M Hilton (eds), (2011). *Learning Science through Computer Games and Simulations* (Washington DC, National Academies Press).
Lesh, R and R Lehrer (2003). 'Models and Modeling Perspectives on the Development of Students and Teachers' 5(2–3) *Mathematical Thinking and Learning* 109–129.
Lesh, R and J Zawojewski (2007). 'Problem Solving and Modeling' in FK Lester Jr (ed), *Second Handbook of Research on Mathematics Teaching and Learning* (Charlotte NC, Information Age Publishing) 763–804.
Martin, RC (2003). *Agile Software Development: Principles, Patterns, and Practices* (Harlow, Pearson).
Miller, G, E Galanter and K Pribram (1960). *Plans and the Structure of Behavior* (New York, Holt, Rinehart and Winston).
National Research Council (2013). Next generation science standards: For states, by states. Washington, DC: National Academy Press.
Pellegrino, JW and ML Hilton (eds), (2013). *Education for Life and Work: Developing Transferable Knowledge and Skills in the 21st century* (Washington DC, National Academies Press).
Papert, S (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. (New York, NY, Basic Books).
Polya, G. (1957). *How to Solve It: A New Aspect of Mathematical Method* 2nd edn (Garden City, NJ, Doubleday).
Repenning, A, D Webb and A Ioannidou (2010). 'Scalable Game Design and the Development of a Checklist for Getting Computational Thinking into Public Schools'. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* 265–269. ACM.
Sengupta, P, JS Kinnebrew, S Basu, G Biswas and D Clark (2013). 'Integrating Computational Thinking with K-12 Science Education using Agent-based Computation: A Theoretical Framework' 18(2) *Education and Information Technologies* 351–380.
Sherin, BL (2001). 'How Students Understand Physics Equations' 19(4) *Cognition and Instruction* 479–541.

Smith, M (2016). *Computer Science for All*. Online blog post. Available at: www.whitehouse.gov/blog/2016/01/30/computer-science-all

Soloway, E, J Bonar and K Ehrlich (1983). 'Cognitive Strategies and Looping Constructs: An Empirical Study' 26(11) *Communications of the ACM* 853–860.

Tedre, M and PJ Denning (2016). 'The Long Quest for Computational Thinking' in Proceedings of the 16th *Koli Calling* Conference on Computing Education Research, 24–27 November 2016, Finland, 120–129. ACM.

Williams, L and R Kessler (2002). *Pair Programming Illuminated* (Harlow, Addison-Wesley).

Wilensky, U, CE Brady and MS Horn (2014). 'Fostering Computational Literacy in Science Classrooms' 57(8) *Communications of the ACM* 24–28.

Wing, JM (2006). 'Computational Thinking' 49(3) *Communications of the ACM* 33–35.

Wing, JM (2008). Computational thinking and thinking about computing. Philosophical transactions of the royal society of London A: mathematical, physical and engineering sciences, 366(1881), 3717–3725.

Wing, J (2014). 'Computational Thinking Benefits Society'. *Social Issues in Computing.* Available at: http://socialissues.cs.toronto.edu/2014/01/computational-thinking/

Wolz, U, M Stone, SM Pulimood and K Pearson (2010). 'Computational Thinking via Interactive Journalism in Middle School' in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* 239–243. ACM.

Yasmeh, J. (2016). 'Is The FBI Lying About Reading 650,000 Emails In 8 Days?' Available at: www.dailywire.com/news/10561/fbi-lying-about-reading–650000-emails–8-days-joshua-yasmeh