Informatics in Education, 2021, Vol. 20, No. 1, 361–390
© 2021 *Vilnius University, ETH Zürich*
DOI: 10.15388/infedu.2021.21

Computational Thinking: A Disciplinary Perspective

Peter J. DENNING[1], Matti TEDRE[2]
[1]*Naval Postgraduate School, USA*
[2]*University of Eastern Finland, Finland*
e-mail: pjd@nps.edu, matti.tedre@uef.fi

Received: May 2021

Abstract. Over its short disciplinary history, computing has seen a stunning number of descriptions of the field's characteristic ways of thinking and practicing, under a large number of different labels. One of the more recent variants, notably in the context of K-12 education, is "computational thinking", which became popular in the early 2000s, and which has given rise to many competing views of the essential character of CT. This article analyzes CT from the perspective of computing's disciplinary ways of thinking and practicing, as expressed in writings of computing's pioneers. The article describes six windows into CT from a computing perspective: its intellectual origins and justification, its aims, and the central concepts, techniques, and ways of thinking in CT that arise from those different origins. The article also presents a way of analyzing CT over different dimensions, such as in terms of breadth vs. depth, specialization vs. generalization, and in terms of skill progression from beginner to expert. Those different views have different aims, theoretical references, conceptual frameworks, and origin stories, and they justify their intellectual essence in different ways.

Keywords: computational thinking, CT, computing as a discipline, history, professionals, advanced, perspectives.

1. Introduction

We are in a computer revolution. Nearly every device now has computers in it – phones, tablets, desktops, watches, navigators, thermometers, medical devices, clocks, televisions, DVD players, and even toothbrushes. Nearly every service is powered by software – bookstores, retail stores, banks, transportation, hotel reservations, filmmaking, entertainment, data storage, online courses, and even daily fitness. These changes have brought enormous benefits as well as worrying concerns. It looks like everything that can be digitized is being digitized and computers are everywhere storing and transforming that information.[1]

This presents an enormous challenge for educators (Guzdial, 2015). What do we need to understand about computers? What must we do to put a computer to work for us? How do computers shape the way we see the world? What are computers not good for? What should be taught to learners at different stages of their education?

Computational thinking (here abbreviated CT) has offered educators some answers to these questions (National Research Council, 2010, 2011). From the days of Charles Babbage, we have wanted computers to do for us jobs that we cannot do ourselves. Much of CT is specifically aimed at figuring out how to get a computer to those jobs for us, and algorithms are the procedures that specify how the computer should do them. Computers are much better at carrying out algorithms than humans are – modern computers can do a trillion steps in the time it takes a human to do one step. A major task for educators is to teach children how to think so that they can design algorithms and machines that reliably and safely do jobs no human can do.

But there is more. Information and computational processes have become a way of understanding natural and social phenomena (Kari and Rozenberg, 2008). Much CT today is oriented toward learning how the world works. Physical scientists, life scientists, social scientists, engineers, humanists, artists, and many others are looking at their subject matter through a computational lens (Rosenbloom, 2013; Meyer and Schroeder, 2015). Computer simulation enables previously impossible virtual experiments. The information interpretation of the world offers conceptual and empirical tools that no other approach does (Kari and Rozenberg, 2008; Fredkin, 2003). Another major task for educators is to teach children how to bring an information interpretation to the natural and virtual worlds without sacrificing wisdom in the process (Weintrop *et al.*, 2016).

Despite the enthusiasm for the power of computers, most jobs cannot be done by computers in any reasonable amount of time, if at all. Students who understand the limits of computing can avoid the trap of thinking that all problems are ultimately solvable by computers.

Most of the education discussion of CT has been formulated for K-12 schools (García-Peñalvo *et al.*, 2016; Guzdial, 2015; Lockwood and Mooney, 2017). It is oriented to helping beginners learn to think about computers. Some of the definitions are shallow and have sown confusion among teachers who do not understand how to teach basic computing and assess student progress (Mannila, 2014; Lockwood and Mooney, 2017). Some definitions lead to conclusions that seem to defy students' common sense about computers, and teachers are asking for clarifications.

[1] Shoshana Zuboff's Second Law, "Everything that can be informated will be informated" describes a techno-deterministic development of society and technology. Zuboff traced her three "laws" to the 1980s in *Be the friction – Our Response to the New Lords of the Ring* (Frankfurter Allgemeine, June 25, 2013) and they became popular through her mid-1990s lectures and were passed around through word of mouth. The original sources, a formulation of her laws, and a yet-unpublished book-length analysis of them, were lost in a tragic accident (Zuboff, December 8, 2018, personal communication).

Our objective in this essay is to describe computational thinking from the viewpoint of computing as a discipline. We will examine CT in three dimensions. First, we will show that CT has a long and distinguished genealogy that began over 4,000 years ago. Many of the concepts of modern CT existed well before digital computers were invented, and many key concepts of CT were painstakingly developed by large numbers of people through the formative years of computing as a discipline. Understanding the evolution of CT engenders a deep respect for CT as well as a wisdom about its applications based on human experience that came before.

Second, the essay will demonstrate that the practices of CT fall on a spectrum from beginner to professional. Much of the CT literature has focused on CT for beginners – a natural consequence of the desire to bring CT into K-12 schools. But there is also a considerable literature on advanced CT as used by professional designers, engineers, and scientists. Beginner CT, aimed at inspiring students' interest in computing, has little to say about a profession that relies on advanced CT. Because of all it leaves out, beginner CT does not describe ways of thinking and practicing of professional computer scientists, either. At the very least, the discussion of a spectrum can assist teachers in showing their students the path they must follow to become professionals in computing.

Third, the essay will show that much of what is today labeled as CT grew out of the computational science movement of the 1980s. That movement emphasized computing as a new way of doing science capable of cracking the visionary "grand challenge" problems. Its wide acceptance in science and engineering created a background of listening that left us open to the resurgence of CT in the 2000s. Scientists who found computing to be materially different from the traditional ways of theory and experiment used the term "computational thinking" to describe the mental disciplines needed for this new kind of science. They also discovered that many natural phenomena can be understood by modeling them as information processes and using computing to simulate them. Thus, CT in the sciences has had a tremendous shaping effect on CT in computing.

This essay is aimed at computing educators interested in situating CT ideas in the broader picture of computing as a discipline and in the sciences in general. It describes computational thinking as an extension of several centuries-long traditions in science, mathematics, and engineering, and it describes how many ideas today labeled "CT" have been presented, in different forms, in many other sciences much before the birth of the modern computer. Science and engineering through the ages are replete with basic "computational thinking" ideas like abstraction, modeling, generalization, data representation, and logic (see, e.g., Grover and Pea, 2018; Barr and Stephenson, 2011; Bundy, 2007; Yadav *et al.*, 2014; Hemmendinger, 2010). The essay describes how the most common descriptions of CT – "CT for beginners" – are just a small subset of computing's disciplinary ways of thinking and practicing. Finally, the essay also explains the ways in which computing really is a new and unique way of looking at problem-solving and automation, as well as of interpreting phenomena in the world.

2. Defining Computational Thinking

Computational thinking (Papert, 1980) has become a buzz word with a multitude of definitions. Much has been written and said about it since 2006 (Wing, 2006; Saqr *et al.*, 2021). Numerous books, journal articles, blog posts, and large educational initiatives have contributed to the development of the concept. High-level workshops have been organized to discuss and define it (National Research Council, 2010, 2011). Countless years of labor have been invested into descriptions of what exactly CT is, how it is different from other kinds of thinking, and, perhaps most visibly, how to teach it to schoolchildren (Lockwood and Mooney, 2017; Guzdial, 2015; García-Peñalvo *et al.*, 2016; Shute *et al.*, 2017; Grover and Pea, 2013; Mannila *et al.*, 2014; Apiola, 2019; Larsson *et al.*, 2019). The public face of CT is that of beginner, or basic, CT – the kind of computational insights and ways of thinking and practicing that can be taught to children in K-9 or K-12 education. That is a laudable goal and a noble continuation of the "computing for everyone" efforts that span over half a century (Guzdial, 2015).

There is a certain degree of consensus on CT basics. The most commonly mentioned skills and concepts include decomposition, abstraction, debugging, iteration, generalization, and algorithms and their design (Shute *et al.*, 2017). Other recommendations include representing, collecting, and analyzing data; automation; parallelization; problem decomposition; and simulation (Barr and Stephenson, 2011). Although touted as the foundations of computing, none of the basic CT descriptions shows students the path to becoming a computing professional. Other descriptions have aimed to show such a path, including computational design (Denning, 2017), computational participation (Kafai, 2016), computational making (Tenenberg, 2018), computational doing (Barr, 2016; Hemmendinger, 2010), computationalist thinking (Isbell *et al.*, 2010), computational literacy (diSessa, 2000), computational fluency (Resnick, 2017)[2] and computational practices (Lye and Koh, 2014), to mention a few. These names do not capture the full gamut of names used for CT – in previous generations, CT has been known as algorithmizing, procedural thinking, algorithmic thinking, procedural literacy, IT literacy, fluency with ICT, and proceduracy (Tedre and Denning, 2016).

Despite the success of CT in convincing many decision-makers, teachers, and curriculum designers to include and integrate computing in K-12 educational systems, much literature in CT is critical of aspects of the current wave of CT. Concerns have been raised about narrow views of CT, such as undue focus on programming, or even coding, at the cost of high-level CT strategies (Armoni, 2016; Mannila *et al.*, 2014). Concerns have been raised about attempts to separate computing from computers (cf. Connor *et al.*, 2017; Nardelli, 2019; Armoni, 2016; Kafai, 2016; Lye and Koh, 2014;

[2] While Resnick's book (Resnick, 2017) does not use the phrase computational fluency, Resnick links it to the book in `https://medium.com/@mres/computational-fluency-776143c8d725`

Lu and Fletcher, 2009; Shute *et al.*, 2017; Bers *et al.*, 2014; Repenning *et al.*, 2010). Concerns have been raised about the uniqueness claims for basic CT – there are many similarities between CT and other kinds of thinking in STEM fields (cf. Grover and Pea, 2018; Barr and Stephenson, 2011; Bundy, 2007; Yadav *et al.*, 2014; Hemmendinger, 2010; Pears, 2019; Sengupta *et al.*, 2013; Werner *et al.*, 2012). Concerns have been raised about the lack of clear demarcation between CT and computer science (Nardelli, 2019; Armoni, 2016; Barr and Stephenson, 2011). Concerns have been raised about the ahistoricity of the CT story – CT is often presented as a new phenomenon without any consideration of its co-evolution with science and mathematics – which has led to further critiques that CT ignores past lessons from computing education (cf. Guzdial, 2015; Voogt *et al.*, 2015; Denning, 2017; Tedre and Denning, 2016). One analyst of CT wrote that "flow optimisation in a cafeteria, the classic example offered by Wing, is a clear example of the application of techniques first used in time and motion studies for process optimisation" (Pears, 2019), and another argued that "considering CT as something new and different is misleading: in the long run it will do more harm than benefit" (Nardelli, 2019). And while there is little discord over the importance of CT to sciences, the relationship between CT and other fields is complicated (cf. Grover and Pea, 2018; Barr and Stephenson, 2011; Bundy, 2007; Yadav *et al.*, 2014; Tedre and Denning, 2017; Hemmendinger, 2010; Hambrusch *et al.*, 2009).

From our study of the genealogy, the science, and the beginner-professional continuum, we have distilled the spirit of the multitude into a definition used throughout this essay:

> Computational thinking is the mental skills and practices for *designing* computations that get computers to do jobs for us, and for *explaining* and *interpreting* the world in terms of information processes.

The design aspect reflects the engineering tradition in computing; in which people build methods and machines to help other people. The explanation aspect reflects the science tradition in computing; in which people seek to understand how computation works and how it shows up in the world. In principle, it is possible to design computations without explaining them, or explain computations without designing them. In practice, these two aspects go hand in hand.

Computations and jobs for computers are not the same. Computations are complex series of numerical calculations and symbol manipulations. Jobs are tasks that someone considers valuable. Today many people seek automation of jobs that previously have not been done by a machine. Computers are now getting good enough at some routine jobs that loss of employment to automation has become an important social concern. We do not equate "doing a job" with automation. Well-defined, routine jobs can be automated, but ill-defined jobs such as "meeting a concern" or "negotiating an agreement" cannot.

3. Algorithmic Genealogy of Computational Thinking

One of the more common descriptions of computing's disciplinary work is that it thinks in terms of algorithms, procedures, or well-defined processes (e.g. Knuth, 1981; Dijkstra, 1974; Harel, 1987). That perspective, which is central to today's descriptions of CT, is also one of the oldest characterizations of computer science[3]. Long before computer science existed, Ada Lovelace, who with Charles Babbage designed the first programs for a programmable computer in the 1840s, described computing as a new "science of operations" (Menabrea, 1842; Priestley, 2011). In the late 1950s, when computing was starting to emerge as a new field, Alan Perlis argued that *algorithmizing* would eventually become necessary for everyone (Katz, 1960). Algorithmizing was his name for computing's unique kind of reasoning for designing solutions to problems. The path from Lovelace to Perlis was created from a number of separate historical milestones in the first half of the 1900s.

This section has three parts: The first traces the historical roots of algorithm-oriented CT concepts, the second presents some central insights from theoretical computer science, and the third examines contemporary views of those concepts. The CT concepts that were born in the algorithmic tradition of computing range from beginner concepts, such as unambiguous computational steps, to advanced concepts, such as regular expressions and computational complexity.

Algorithmic Genealogy

The algorithmic view of CT has roots in computational methods of applied mathematics. Algorithm-like procedures have been found on ancient Babylonian clay tablets (Knuth, 1972), and the term "algorithm" comes from the 800 CE Persian mathematician Muhammad ibn Mūsā al-Khwārizmī whose procedures preceded the modern notion of the algorithm (Knuth, 1981). In the history of mathematics, computational methods helped traders, builders, and scientists to reliably perform important calculations (Grier, 2005; Cortada, 1993; Westfall, 1980). Famous examples abound: Euclid's method found the greatest common divisor of two numbers, the Sieve of Eratosthenes found prime numbers, and Gauss Elimination found solutions to systems of linear equations (Chabert, 1999). These methods were motivated by the pragmatic goal of enabling laypeople to perform mathematical procedures without deep knowledge of mathematics.

Over the centuries, algorithmists gradually developed a complex set of ideas for making algorithms effective. These included representing numbers and other data, specifying unambiguous steps, establishing a rigorous logical framework for a procedure, and dealing with round-off errors that result when continuous quantities are represented with finite numbers of bits. In the next paragraphs, we will comment on each of these elements of the algorithmic tradition of computing.

[3] For examples of operational definitions of CT in education, see Google's and ISTE/CSTA's notes at `https://id.iste.org/docs/ct-documents/computational-thinking-operational-definition-flyer.pdf` and `https://edu.google.com/resources/programs/exploring-computational-thinking/`

Start with representations. Through the long history of algorithms, every algorithm designer has had to think about how to represent numbers and other symbols (Grier, 2005; Cortada, 1993). The numbers computed by algorithms are actually codes standing for numbers, using a finite set of symbols. Binary coding systems, which use just two symbols, can be found as far back as Babbage's Analytic Engine, and before. The Hollerith machines built for the 1890 US Census used punched cards with patterns of holes representing a person's age, education, and marital status – again just two symbols, a hole or non-hole at a location on the card. Since the 1940s, digital computers used just two symbols, 1 and 0, represented as high or low voltages in the circuits (De Mol *et al.*, 2018). Today the process of encoding information into a binary representation is called "digitization." Designing good representations is fundamental issue in CT.

Next is the issue of specifying the computational steps of algorithms. The birth of calculus in the mid-seventeenth century gave scientists a much more reliable way to deal with problems requiring copious calculations of functions (Westfall, 1980; Grier, 2005). But that posed a problem: the procedures had to be composed of unambiguous operations, for otherwise they might not give the same results in the hands of different persons. Each step of a computation had to be so precisely defined that there would be no need for human interpretation, intuition, or judgment – or error (Grier, 2005). The use of procedures built from unambiguous steps has become a cornerstone of CT.

Next is the issue of devising a logical plan for computing a function. A procedure specifies individual operations, such as addition and subtraction, and also choices between different sets of operations. Mathematicians turned to formalization of logic to do this precisely and unambiguously. The usual story of the influence of logic on computing starts with the philosophers René Descartes and Gottfried Leibniz, who sought to formalize how humans reason (Dasgupta, 2014; Davis, 2012; Tedre, 2014). George Boole made a breakthrough when he presented an algebra of logic that represented logical formulas with expressions composed from connectives AND, OR, and NOT (Boole, 1854; Davis, 2012). In 1937 Claude Shannon showed how to represent the switching circuits of telephone systems and computers with Boolean formulas (Shannon, 1937). But Boole's work did not include formal means to deal with making choices and repeating operations. The basis for that was provided in 1879 by Gottlob Frege (1879). The combination of Boole's and Frege's insights became the basis for many programming languages and an important element of CT.

It is an irony that despite the great lengths those designing algorithms and machines went to avoid error, errors have been a plague for programmers in all ages. One study of 40 volumes of old mathematical tables found 3700 errors in them, and another found 40 errors on just one page (Williams, 1997). To reduce execution errors, many algorithms include elaborate acceptance checking for whether the computation is producing results that meet specifications. Sometimes different algorithms designed by different teams are run in parallel. And a great effort has been made to apply formal logic to prove that programs meet their specifications. A similar effort is done by hardware designers to increase trust that the machine implements the basic operations correctly. Minimizing or removing errors will continue to be an important element of CT from the beginning – *CS Unplugged*, for example, uses a fun and engaging "magic trick" for teaching children error checking[4].

Next is the problem of designing algorithms to cope with the limited precision of finite-string representations. For example, numbers are represented in many computers as 32-bit quantities, which are incapable of representing all possible numbers. Algorithms can be designed to ensure that round-off errors do not accumulate over long calculations. Mathematical pioneers such as Euler, Lagrange, and Jacobi worked out methods to minimize round-off errors in algorithms long before there were computing machines (Grier, 2005; Bullynck, 2016; Goldstine, 1977).

Finally, there is one more important aspect of the algorithmic tradition: characterizing the limits of computation. The late 1800s to early 1900s were a heyday of formalism in that quest: Not only did Frege's predicate logic fill in gaps where Boole's logic could not reach, but mathematics and logic merged in *Principia Mathematica*, the magnum opus of Russell and Whitehead. Logical empiricism came to rule the sciences. Mathematicians fervently believed that logic would finally allow them to realize Descartes's and Leibniz's dream of formalizing human thought. They sought an ultimate algorithm that could definitively solve the *Decision Problem*: whether a statement in predicate logic is true or false. The quest to find an algorithm for the Decision Problem was taken in 1928 as one of the major challenges in mathematics (Hilbert and Ackermann, 1928). That problem was resolved in the 1930s simultaneously by several people, among them a young Cambridge mathematics student named Alan Turing. Turing developed a mathematical model of a computing machine capable of hosting any algorithm for the Decision Problem (Hodges, 1983). He called his machines *a*-machines, with "a" for automatic (Turing, 1936). Turing's conclusion was negative: an algorithm for the Decision Problem is impossible.

Alonzo Church labeled Turing's mathematical model the "Turing machine". Turing presented a universal machine that could simulate any other machine, leading to a universal way of representing all computable activities (Cooper and van Leeuwen, 2013). Turing then showed that an algorithm for the Decision Problem was logically impossible on any machine. Turing's machine model of computing was a signal achievement in mathematical logic. It soon became a cornerstone of the theory of computing and a rallying point for a new kind of computational thinking (Daylight, 2014, 2016). It led to the theory of noncomputable functions and to algorithmic complexity theory.

Nearly all the CT concepts underlying algorithms existed before the dawn of the Information Age and were used in many fields including mathematics, logic, science, and engineering. The contribution of Computer Science was to unify them together into a framework for getting electronic computers to reliably use algorithms to solve problems.

Efficiency of Automation

Turing's model came to symbolize the question of what can be automated – later dubbed one of the most inspiring philosophical questions of contemporary civilization (For-

[4] `https://classic.csunplugged.org/error-detection/`

sythe, 1968; Arden, 1980). But another question vexed those who worked with human computing projects: how to minimize the hand-computing effort and keep the time spent on computing within bearable limits. In the 1800s, even before fully programmable computing machinery had been built, Babbage foresaw the issue of how can "results be arrived at by the machine in the shortest time" (Babbage, 1864, p.137). After the birth of the programmable, fully-electronic, digital computer, early programmers grappled with matters of efficiency, and with the issue that some problems were inherently more open to efficient algorithms than others. For the nascent computer science community, a formalization of that phenomenon was provided in 1965 (Hartmanis and Stearns, 1965), and the concept of computational complexity quickly became a central feature of computational thinking.

As another example of the diverse origins of central CT ideas, an early discussion of an "NP-complete" problem and its consequences was started by Gödel, on a question concerning linear vs. quadratic time for proofs in first-order logic (Fortnow and Homer, 2003). In 1971 Steve Cook gave a formal definition of a set of "NP-complete" problems – their known algorithms took impractically long much time to find solutions, but any solutions could be rapidly validated (Cook, 1971).

This idea forever changed computational thinking: Tens of thousands of optimization problems from flight scheduling to protein folding were shown to be NP-complete (Vardi, 2013). This was gloomy news for those searching for fast algorithms for those hard problems – Moshe Vardi commented, "first-order logic is undecidable, the decidable fragments are either too weak or too intractable, even Boolean logic is intractable" (Vardi, 2013). Over time, algorithm experts found approximations, probabilistic methods, and other heuristics that do surprising well for problems in the harder complexity classes (cf. Fortnow and Homer, 2003; Vardi, 2013). Understanding the framework of computational complexity, its foundations, limitations, and its theoretical vs. practical consequences has become essential for intermediate to advanced CT.

Is the Algorithm The Spirit of Computing?

Today's notions of algorithms are rooted in the mathematical definitions of computability that emerged around the 1930s from the pioneering work of Church, Gödel, Kleene, Post, and Turing (Chabert, 1999, p.457). But with the birth of the digital computer, the concept of algorithm developed along a different, much less mathematical path (Bullynck, 2016; Chabert, 1999), shaped by the pragmatics of getting software to run reliably on real computers (Daylight, 2016). These pragmatics drove a consensus on the main features of algorithms: they are finite sequences of definite, stepwise machine-realizable operations that manipulate symbols; they may have inputs; they always have outputs; and they finish in a finite length of time (Knuth, 1997). Notice how much this definition of algorithm is tied to computers. Knuth saw algorithms as different in kind from nearly all types of human-executable plans: "An algorithm must be specified to a degree that even a computer can follow the directions" (Knuth, 1997, p.6).

In the 1950s, programming was regarded as a process to specify algorithms in a formal language that instructed a machine to carry out its steps. Originally, the formal language was assembly language, which was simply the instruction set of the machine.

Programming in assembly language was tedious and error prone. In the mid-1950s, higher level languages began to appear. These languages provided single statements corresponding to sequences of many machine instructions. They simplified the expression of algorithms and came with compilers that translated to machine code (Mahoney, 2011). They included Fortran, ALGOL, COBOL, and Lisp. The quest for efficient compilers was a strong driver of the research on automata.

By the 1960s, algorithms, programs, and compilers were seen as the heart of computing. A number of prominent people even proposed that the field be renamed to "algorithmics" (Traub, 1964; Knuth, 1981; Harel, 1987). In a series of papers, Knuth described how algorithmic thinking differed from classical mathematical thinking (Knuth, 1974a, 1981, 1985). He concluded that the main differences are the design of complex algorithms by composition of simpler algorithmic pieces, the emphasis on information structures, the attention to how actions alter the states of data, the use of symbolic representations of reality, and the skill of inventing new notations to expedite problem-solving.

Others joined Knuth in clarifying how computing differs from mathematics. One author highlighted computing's use of procedural (action-oriented) knowledge instead of mathematics' declarative knowledge (Tseytin, 1981). Another insisted that while mathematicians might be interested in syntactical relations between symbols and their semantics, computing is inherently pragmatic because it aims for software that works (Gorn, 1963). Another argued that the concerns of mathematicians and computing people are fundamentally different (Forsythe, 1968). Another wrote that computer scientists differ from mathematicians by their ability to express algorithms in both natural and formal languages, to devise their own notations to simplify computations, to master complexity and agilely switch between abstraction levels, and to design their own concepts, objects, theories, and notations when necessary (Dijkstra, 1974).

The pure algorithmic view of computing began to be challenged in the late 1960s from a new direction by a larger view of computing that included many people sharing information and machine resources via operating systems and networks (Denning, 2016). Operating systems had a pragmatic origin in the late 1950s. Computers were scarce and were housed in computing centers where engineers could keep them running. Computing centers had to process many programs submitted by many independent users. Their personnel queued jobs for execution, loaded them into the machine, allocated machine resources such as memory and input-output, and delivered results back to their users. Computing center engineers invented the first operating systems to automate this work. However, users detested those early operating systems for their long turnaround times, often 24 hours. In 1960, researchers began to experiment with time-sharing to eliminate turnaround times by enabling interactive programming. Time sharing operating systems were much more complex. By 1970, a set of operating systems principles had emerged to deal with the complexity – concurrent processes, virtual memory, locality of reference, globally naming digital objects, protection, sharing, levels of abstraction, virtual machines, and system programming (Denning, 2016). An operating system was seen as a "society of cooperating processes" rather than a set of algorithms. Control of concurrency to avoid nondeterministic behavior and coordinate signalling across networks became central concerns. The algorithmic view was insufficient to capture everything people wanted to do in their shared systems.

Probably because of its relative simplicity, the algorithms viewpoint has dominated the K-12 CT movement since 2006. A common description in the K-12 curriculum recommendations is that CT is the habits of mind involved in formulating problems in a way that allows them to be solved by computational steps and algorithms (Aho, 2011). These habits include designing abstractions that hide details behind simple interfaces; dissecting solutions into discrete, elementary computational steps; representing data with symbols; and knowing a library of common useful algorithms (Shute *et al.*, 2017). Little or nothing is said about operating systems, networks, concurrency, memory management, information sharing, and information protection – concepts often seen as more advanced forms of CT for which beginners are not ready.

4. A Professional Continuum of Computational Thinking

4.1. *CT: Automation and Machine Control*

Turn now to automation – how to get computing machines to do jobs for us (Forsythe, 1969; Arden, 1980; Denning and Tedre, 2019). Automation is a bigger issue than finding an algorithm that will solve a problem. Autopilots, for example, fly planes as well as pilots. They are complex mechanisms with gyroscopes, GPS sensors, algorithms, and feedback loops. In computing we have looked to automation to enable tasks that humans might be able to do at small scale but cannot do at large scale. An example is finding out if a particular person is in a video of a moving crowd. Humans can do this reliably only for small crowds. By combining neural networks that can recognize faces with algorithms that search images, we can now automate this task for large crowds. Many forms of computer automation aim to extend small human tasks to large scales (Connor *et al.*, 2017). Another example is drawing the next frame of a video on a computer screen; it would take a human calculating every second of every day for a year to do this job, but a graphics system can do it in 10 milliseconds.

Some proponents of CT have ignored the distinction between doing small and large versions of a task. They argue that algorithms are executed by "information agents" and that humans are information agents[5]. This misleading claim is embedded into some K-12 definitions of CT. While this claim applies to small tasks that can be completed in a short time, it does not apply to large tasks. A machine "agent" can, in a short time, complete large tasks that are completely beyond human capabilities.[6]

[5] Jeannette Wing, "Computational Thinking: What and Why?", Manuscript dated to November 17, 2010. Available at `https://www.cs.cmu.edu/~CompThink/resources/TheLinkWing.pdf`

[6] Indeed, before 1940, the term "computer" referred to a person doing computations. The first automatic computers were built because jobs such as calculating ballistic trajectories and cracking ciphers were well beyond the capabilities of human computers.

Today's CT discussions about the role of machinery in thinking about computing are strikingly similar to debates in the 1960s over the status of computing machinery in the nascent computing discipline (Tedre, 2014). At the time, many scientists in other fields claimed that computer science cannot be a science because it is about human-made computing machines, not processes of nature. The presence of machines meant that computer science was not really a science. The modern equivalent of this is the idea that an algorithm cannot be an algorithm if it depends on a machine. Even the most ardent supporters of the algorithmic view of computing do not endorse this view. They emphasize that algorithmic processes must be machine realizable. Donald Knuth, for example, in his monumental work *The Art of Computer Programming* (Knuth, 1997), uses a machine language, MMIX, that closely resembles a von Neumann machine instruction set (Knuth, 1997, p. *ix*). One Turing Award winner Richard Hamming quipped that without the computer almost everything in computing would be idle speculation, similar to medieval scholasticism (Hamming, 1969).

The Machine Counts

In practice algorithms and computing machines are strongly intertwined. On the one hand they seem separable because the history of science knew algorithms for centuries, if not millennia, with only a scattering of machines to implement them such as Pascal's arithmetic calculator (ca 1650) and slide rules inspired by Napier's logarithms (ca 1620). Early algorithms aided people to undertake complex computations. On the other hand, they seem inseparable today: Even the most shining examples of theoretical computer science are often investigated with machine-like terminology – such as the Turing Machine and Knuth's The Art of Computer Programming. Turing himself argued that manipulating symbols mechanically was essential for computing numbers.

In his 1936 paper, Alan Turing defined computability using an automaton that imitates a mathematician carrying out a proof. As observers, we would see the mathematician writing symbols on paper, then moving to adjacent symbols and possibly modifying them, all the while mentally keeping track of some sort of state. He modelled this behavior with an infinite tape and a finite state control unit. His simple machine, which he called an *a*-machine (*a* for automatic), was soon called a Turing Machine. Its moves were of the form, "possibly change the current symbol, move a square right (left), and enter a new state." From this he proved the existence of a universal machine (one that can simulate any other) and his remarkable proof that the Decision Problem could not be solved by any machine.

The Turing model of computation won out over competing models because its mechanical, machine-like form was the most intuitive (Kleene, 1981; Church, 1937). The modern definition of algorithm depends on the machine realizability of individual instructions as in the instruction cycle of a von Neumann CPU. Machines and algorithms intertwine.

Today's debates frame algorithms and automation as points of view that can be compared and contrasted. The algorithmic view sees computations as abstract entities one can reason about (cf. Smith, 1998, pp. 29–32). The automation view sees computations as the operations of physical machines that realize algorithmic tasks on physical media (Smith, 1998). Whereas the algorithmic view sees computations as abstract, the automation view sees them as physical processes (cf. Smith, 1998; Tseytin, 1981).

When historians analyze the progress of computing, they invariably cite progress with machines over progress with algorithms (Williams, 1997; Campbell-Kelly and Aspray, 2004). Many stories of computing reach back to roots with the Jacquard loom, which demonstrated that weaving machines could switch to new patterns by changing the cards (a sort of "program" for weaving). Electromechanical tabulating machinery enabled the 1890 US Census and eventually took over the jobs of thousands of people (Williams, 1997; Cortada, 1993). Analog computers such as mechanical integrators could trace the values of complex functions and solve differential equations (Williams, 1997; Tedre, 2014). Today's competition for world leadership in computing is measured by the speed of supercomputers or banks of Graphics Processing Units, not just the algorithms they run.

In the middle 1980s, John Rice, a pioneer of mathematical software, tried to achieve a more balanced view. He said that the mathematical software of the day had improved by 10^{12}, of which 10^6 was attributable to improvements in hardware and 10^6 to improvements in the design of algorithms. This is still true today. For example, machines to recognize faces were very slow and error prone in the 1980s whereas today they use the advanced algorithms of deep neural networks combined with the superior speed of GPU chips to do the job. Despite the desires of some advocates to simplify CT by ridding it of machines, it cannot be done: machines will continue to gather our attention in computing.

Some proponents of basic CT mistakenly conflate the stored program idea with Turing's universal computer idea. Historians have showed that those two have developed on two parallel historical trajectories (Haigh, 2013). They are separate ideas.

The occasional attempts to separate algorithms from computers floundered in the past and will continue to flounder (MacKenzie, 2001). Even the staunchest advocates did not model the distinction. For example, Dijkstra showed great prowess writing efficient compilers and operating systems and yet said "the computing scientist could not care less about the specific technology that might be used to realize machines, be it electronics, optics, pneumatics, or magic" (Dijkstra, 1986). This was not a passing statement. He repeatedly said "computer science is not about machines, in the same way that astronomy is not about telescopes" (Fellows, 1993; Dijkstra, 2000; Daylight, 2012).

In the end, the marriage of the algorithm and automation views drove CT into central questions that shifted as new algorithms and machines were developed. For example, Babbage's idea that computers could eliminate human error was displaced a century later with the realization that machines were so complex that no one could be sure they did what the algorithms told them to do – and even less sure whether they met their designers' intentions (Smith, 1985, 1998). The early question of measuring "cost" of an algorithm as the CPU time it consumed gave way to network performance measures such as throughput and response time.

4.2. *Is Programming Essential in CT?*

The invention of the computer created a new concept – the program – that came to symbolize the computer age. Useful computer programs written in high-level languages can be transferred to different computers, where they can be compiled to different machine code and executed. Program libraries such as mathematical software became standard features of computing systems. Downloadable software "apps" are a standard feature of today's portable devices. Software libraries are universally available. CT aims to elucidate the thought processes behind the designs of all these programs (Bell and Roberts, 2016; Wing, 2006).

The preponderance of public discourse on CT is not the abstract algorithm but the executable computer program. Code.org, International Society for Technology in Education (ISTE), Computer Science Teachers Association (CSTA), and Google for Education all refer heavily to programming skills and concepts in their CT material, typically using Python (Google), Blockly (code.org), or Scratch (CSTA).[7]

The skilled practice of programming is seen widely as central to CT. Yet, many programming language concepts that are today regarded as self-evident – such as while-loops, data structures, and recursion – were not initially apparent, but were the result of much work by brilliant people over many years (Knuth and Trabb Pardo, 1980). A significant body of software was written before crucial programming language concepts started to emerge (Glass, 2005). At least one computing pioneer wondered aloud how all those non-trivial programs were made to work by people who had only "primitive" mental tools for programming (Dijkstra, 1980). Programming methodology, developed since 1970, aimed at improving dependability, reliability, usability, security, safety, and even elegance of programs, which are not always compatible goals (Daylight, 2012). Evolving programming methodology brought new programming language constructs and programming techniques such as structured programming and object-oriented programming (Liskov, 1996). Much CT terminology and concepts originate directly from developments in programming methodology and software engineering.

Origins of Tools for Computational Problem Solving

The five key programming aspects of basic CT – modularity, data structures, encapsulation, control structures, and recursion – are often held to be unique to computing. But this is not strictly true. Each has deep roots in many fields. This is good for CT because these ideas have stood the test of time.

Because programming is such a central aspect of computing, much effort has gone into the design of programming languages starting in the 1950s. High level languages simplified the programming job and reduced errors in programs. They came in many flavors – such as procedural programming, functional programming, symbolic programming, script programming, artificial-intelligence programming, object-oriented pro-

[7] See, e.g., `https://code.org/`, `https://hourofcode.com/`, `www.csteachers.org/resource/resmgr/CTExamplesTable.pdf`, and `https://edu.google.com/resources/programs/exploring-computational-thinking/`

gramming, and dataflow programming – each attuned to a particular style of problem-solving (Knuth and Trabb Pardo, 1980; Sammet, 1972; Wexelblat, 1981). The number of programming languages multiplied over the 1950s and 1960s (Sammet, 1972). Let us take a closer look at the origins of the five basic ideas of CT.

Modularization. The very first programming textbook from 1951 (Wilkes *et al.*, 1951) noted the need to divide programs into smaller, manageable pieces. Reducing complex systems to structures of many simple components is an old engineering practice. Modularization of programs was achieved by subroutines, functions, procedures, and classes in programming languages. The Atlas machine at the University of Manchester introduced hardware support for subroutine calls. The major languages of the late 1950s – Fortran, Algol, Lisp, and Cobol – all included subroutines. Structuring programs as callable modules fascinated educators, who called it procedural thinking (e.g., Solomon, 1976; Abelson *et al.*, 1976). The early development of software engineering after 1968 used metaphors from industrial and mechanical engineering (Mahoney, 2011, pp. 93–104), emphasizing parallels with the automobile industry, interchangeable parts, machine tools, and industrial mass production (Naur, 1969; Mahoney, 2011; Randell, 1979)

Data Structures and Encapsulation. In the early 1960s, experienced programmers advised their students to start the design of a program with the organization of the data. They had found that choosing the right data structure for the job at hand was key for finding a simple algorithm. By the late 1960s, this practice was called "data abstraction". It specified that a data structure would be hidden behind an interface of operations presented to users; users could not access the data directly. This approach allowed improvements to be made to a module without requiring changes to other modules that used it. For instance, Simula 67, a simulation language, incorporated this idea into the structure of a language (Holmevik, 1994). The idea evolved into object oriented programming by the early 1970s (Krajewski, 2011; Liskov, 1996; Hoare, 1972; Sammet, 1981).

Control Structures. The early ideas of von Neumann architecture conceived of instructions as "orders" that the machine obeyed. Programming was seen as a way to control machines. Thus a lot of attention was paid to the organization of control. In 1966 Böhm and Jacopini published a theorem that said three control structures (sequence, iteration, and selection) are sufficient for any program (Böhm and Jacopini, 1966). In 1968, Dijkstra introduced structured programming, which had specific statements for sequencing, iterating, and selecting; he emphasized that these are the three ways that we organize our proofs that a program works correctly. (Although structured programming was fundamentally about good abstraction practices, many well-known expert programs did not endorse it (Hoare, 1996). Unfortunately the debate derailed into one about whether it was wise to allow the GO TO statement (e.g., Dijkstra, 1968; Knuth, 1974b).) The Böhm-Jacopini minimalistic insight was taken as a loose programming analog of the basic logic gates for computer circuits. But it is clear from the notes of Babbage and Lovelace that they used the same programming and machine structures without giving them explicit names, and the same concepts have arisen in different contexts throughout history (Rapaport, 2018).

In the 1960s, the idea of control structures blossomed into many new ways to specify the order of operations in programs. They included new ways to control instruction flow between blocks of statements such as repeat-until, do-while, if-then-else, and case statements. They also included ideas to allow concurrent operations within a program, controlled by fork and join operations and synchronized with semaphore operations (Hoare, 1996; Knuth and Trabb Pardo, 1980; Glass, 2005).

Recursion. The technique of recursion was known to mathematicians in the 1800s as "definition by induction". It entered computing as a theoretical construct from mathematical logic in the 1930s (Soare, 1996). It was an integral part of Gödel's and Kleene's models of computation. It entered as a practical means of programming through the languages Algol and Lisp (Daylight, 2012, Ch.3). It entered as a means to specify elegant algorithms, such as Hoare's 1961 Quicksort. In the 1960s, the Burroughs Corporation built the B5000 and B6700 machines to provide highly efficient stack-oriented execution environments for recursive programs. These machines removed any doubts that recursive program execution could be efficient. Implementation of stacks in hardware and operating system software became permanent fixtures in computing.

These five ideas appropriated from early computer science and other fields of engineering, science, and mathematics formed the core of a new way of solving problems (Forsythe, 1959; Katz, 1960). Hundreds of articles and books described computational methods and CT concepts as tools for problem-solving in different programming languages. These ideas have become the core of the modern movement for beginner CT (Aho, 2011; Wing, 2006)[8].

4.3. *Software Development and Design*

Programming methodology promoted best programming practices for designing and writing programs. It helped programming evolve from a "black art of obscure codes" to a rigorous discipline (Wirth, 2008; Backus, 1980; Dijkstra, 1980). It provided the mental tools for analyzing problems in a way that permitted computational solutions. But, by the late 1960s, the software industry and its customers were painfully aware of how inadequate their programming methods were for large software systems, and just how difficult it is to write reliable program code for large systems (Ensmenger, 2010). Developers of large software systems faced chronic problems with missed deadlines, overrun budgets, poor reliability, usability, unmet specifications, managing software projects, and safety (Mahoney, 2011; Ensmenger, 2010; Friedman and Cornford, 1989). None of those problems could be addressed with improvements in programming methodology. In 1968 a NATO conference acknowledged the software crisis and agreed to launch a new field, software engineering, to do something about it (Friedman and Cornford, 1989). As software engineering gradually became a respected profession (Ensmenger, 2001), its new ideas gradually entered advanced computational thinking.

[8] For example, Google's CT course for educators states, "Computational Thinking (CT) is a problem solving process [. . .] used to support problem solving across all disciplines" see: `https://computationalthinkingcourse.withgoogle.com/unit`

Software engineering had broad appeal. It suggested that many traditional ideas from engineering could be brought to the development of large software systems. Soon the term software engineering turned into an umbrella term for a variety of practices to bring large, complex, safety-critical software systems into production (Ensmenger, 2010; Tedre, 2014). There was soon a debate among educators about whether software engineering is a branch of computer science or of engineering. Many doubted whether the mathematical mindset of computer science departments would be amenable to an engineering mindset for software. Many aspects of software engineering, such as design strategies, management of software projects, customer service issues, and safety issues did not seem to fit in computer science departments (Naur and Randell, 1969).

The terms *programming in the small* and *programming in the large* were used to distinguish between the design of single procedures, algorithms, or programs, and the design of large systems possibly consisting of many interacting programs. Computing pioneer David Parnas summed up programming in the large as managing "multi-person development of multi-version programs" (Parnas, 2011). He cited the issues of communicating with the intended users and elucidating their requirements, managing large teams of programmers, coordinating software development projects, dealing with complexities that arise from millions of lines of code and increasingly complex hardware, maintaining and improving software after its release, and training programmers to think like engineers (Ensmenger, 2010; Parnas, 2011; Mahoney, 2011). All the efforts for large systems opened a whole world of advanced CT concepts, practices, and professional skills.

Software Systems Thinking for Professionals

Systems engineering emerged when new sociotechnical systems grew so complex that single individuals could no longer design them. Grace Hopper pointed out the turning point in computing: "Life was simple before World War II. After that, we had systems" (Schieber, 1987).

Operating systems were among the first large complex software systems. There have been a few instances in history where the entire operating system was designed and implemented by one or two persons – for example, the THE multiprogramming system around 1968 (Dijkstra, 1968), the UNIX system around 1972 (Ritchie and Thompson, 1974), and the XINU system around 1980 (Comer, 2012). When large systems have been put together by large teams, they become too large for any one person to understand (Brooks, 1975).

Similar to other engineering fields, as software systems grew too large for any single person to develop and maintain, there was a need to recognize new ways of planning, designing, and developing systems. The emergence of software engineering was a systems thinking-based response that superseded the older programming-in-the-small practices (Brooks, 1975, 1987). The systems responses typically arose to meet problems encountered in production. One computing pioneer reminisced, "I have never seen an engineer build a bridge of unprecedented span, with brand new materials, for a kind of traffic never seen before – but that's exactly what has happened on OS/360 and TSS/360" (Randell, 1979).

Software modules assumed a strong place in software engineering. Like hardware bolt-on modules familiar to engineers, software modules are system components that can be developed and maintained independently. Most modules are designed as black boxes that internally hold a hidden data structure, with an external interface that specifies the functions that can be performed on the internal data. Modules have external interfaces that provide all the functions implemented by the module. Modules developed for one system can be reused in another. Modularization facilitates decomposition of a large problem into small subproblems whose modules are easier to design. Modularization is the pragmatic approach of software developers to putting principles of abstraction to work.

For a while, development engineers believed that modules were the key to large systems, where the programming and testing had to be distributed among many programmers. Each programmer was given a detailed specification of the interface and asked to prove and validate through testing that the interface worked as intended and all the internal data were completely hidden. Yet, when independently developed modules were brought together in a system, they often failed. The failures arose from subtle differences in the ways that the development teams interpreted the interface operations. Somehow the overarching principles of the system must be communicated and understood by all the module development teams (Brooks, 1975)

Portability was an important side benefit of modularity. This meant that a module developed on one system could be transported into another system with possibly different operating systems and hardware. One approach is to gather a set of related modules into a library, such as mathematical software or Java language add-ons, and provide the library to users on many machines so they could link modules as needed. Another approach was to design the modules in high level languages and use the compilers to translate them into machine code for the specific machine. Still another was to design a family of machines (such as IBM OS/360) with the same instruction set, which allowed modules to be reused on other members of the family without recompilation (Brooks, 1975). And finally is the approached of the Java language, which defined a middle level virtual machine that can be compiled for each host machine. The compilers of the modules translate module operations to the virtual machine interface, which in turn the virtual machine translates into machine code.

In the 1990s, expert designers concluded that exchanging models is not necessarily the best way to share design expertise. Inspired by the work of Christopher Alexander, a famous architect (Alexander, 1979), they specified a number of important thought patterns that appear in software systems (Gamma *et al.*, 1994). They identified design patterns for a large number of common programming situations, such as where a program needs only one instance of a class, or where the program needs to sequentially access elements of a set. Another approach to sharing experts' computational thinking was design principles (e.g. Saltzer and Schroeder, 1975), which are holistic ways of thinking about rigorous designs for systems consisting of numerous interacting components. Another approach was design hints, which acknowledged the difference between designing algorithms and designing systems. Design hints were an attempt to crystallize the design choices and judgments skilled systems designers had learned to make:

They included maxims like "separate normal and worst cases," "make actions atomic," and "keep interface stable" (Lampson, 1983). Design patterns, principles, and hints are advice from experts to other experts and probably would make little sense to novice programmers.

All these aspects of system thinking for professionals are examples of advanced concepts of computational thinking. Whereas basic CT is typically more generic, more widely applicable, and less unique to computing as a discipline, advanced CT is typically more specialized, born and honed through experience in design, implementation, and maintenance of large computer and information systems.

5. Computational Thinking and Science

There is wide appreciation that computing has transformed science and engineering in fundamental ways. This appreciation is one of the most important reasons for the attractiveness of CT. Computing fundamentally improved the collection and analysis of data, the design of simulations and models, and the ability to model information processes found in nature. Computing has been called the "third pillar" of science (Oberkampf and Roy, 2010), the "fourth great scientific domain" (Rosenbloom, 2013), and the "most disruptive paradigm shift in the sciences since quantum mechanics" (Chazelle, 2006). Along with this shift, computing professionals deemphasized the idea that computing is a science of automation and embraced the idea that it is a science of natural and artificial information processes (Denning, 2007). Throughout computational science, computing does not just "enable" better research, but often drives productive new kinds of research (Meyer and Schroeder, 2015, p.207) – although many "new" ideas in computational science have clear counterparts in pre-computer science, too (Agar, 2006).

Despite early claims that basic computing ideas are easily transferred across domains, STEM educators have concluded that CT is not domain-independent; it looks different in different disciplines (Weintrop *et al.*, 2016; Yadav *et al.*, 2017; Barr and Stephenson, 2011). Critics have called the over-zealous push of a standardized notion CT into other domains of science "arrogant", "imperialistic", and "chauvinistic" or just plain "ill considered" (Hemmendinger, 2010; Denning *et al.*, 2017). What is more, the info-computational (Dodig-Crnkovic and Müller, 2011) or algorithmic revolution in science has not been a monolithic single revolution that overthrows an old regime (Tedre and Denning, 2017). The transformation has been gradual. Four distinctions emerged that were emblematic of computational thinking in science. They are discussed next.

1. A new instrument of science. Massive increases in computer speed and memory allowed scientists to run simulations and evaluate mathematical models that were previously untouchable (Grier, 2005). For example, scientists in computational fluid dynamics knew how to build models for complete aircraft simulation, but did not have access to supercomputers capable of running them until the late 1980s. Experimental scientists embraced data science as a new set of analytic methods to analyze very large data sets. Theoretical scientists got tools for numerically solving equations that had no closed-form solutions (Tedre and Denning, 2016).

These tools allowed complex models of dynamic systems to be evaluated in near real-time. Models for weather forecasting (Grier, 2005, pp.142–144,169) and nuclear reactions (Haigh *et al.*, 2016, p.5) pushed the state of the art since the 1940s. In the 1980s, scientists from all fields compiled a list of "grand challenge" problems that would be solved with sufficient computing power and, with help from Moore's law, they predicted when these solutions would be feasible (Executive Office of the President: Office of Science and Technology Policy, 1987). These problems included fusion energy, design of hypersonic aircraft, full simulation of aircraft in flight, cosmology, and natural language understanding.

2. New scientific methods. Since 1950, scientists were able to bring to their investigations new methods enabled by the electronic digital computer. Early computers during World War II allowed rapid calculation of ballistic trajectories of new ordnance and cracked the German Enigma cipher. Monte Carlo simulation became fashionable in the mid-1940s to find probabilistic approximations for thermonuclear and fission devices, cosmic rays, high-temperature plasma, and many other phenomena (Eckhardt, 1987). In the 1980s, supercomputers led to a rapid proliferation of simulations in the sciences, leading to discoveries that earned Nobel Prizes on topics such as phase transitions in materials and interactions between tumor viruses and cells (Tedre and Denning, 2017).

Computer modeling and simulation evolved into a new way of doing science. The new way, was investigations using the computer as the instrument and experimental apparatus. Physicists studied phase changes of materials, chemists design of new molecules, economists simulation of national and world economies, cosmologists the evolution of the universe, biologists the structures of DNA, and much more. None of these investigations could be done with the traditional methods of experimental or theoretical science. Computation was seen as a new way of doing science. All the fields using computational methods established new branches of computational science. The term "computational thinking" came into vogue to characterize the new kind of thinking required for this new way of science. The computational sciences movement received political support in the US on the passage of the High Performance Computing and Communications Act (1991), which opened new streams of funding for computational science research and development. Simulation has become so important that it is today inconceivable that major infrastructure investments could be built without exhaustive simulations in advance.

3. New lens for interpreting results. Simulations enabled "virtual experiments" in which natural processes could be modeled as information processes. The good agreement between these models and the real processes led many scientists to change their views and interpret their fields as study of "natural information processes". Biology was the first field to fully embrace this in its study of DNA sequencing and genome editing – in 2001, David Baltimore, who won a Nobel Prize in Biology, claimed "Biology is an information science" (Baltimore, 2002). Leonard Adleman declared himself to be a scientist studying information processes in DNA transcription and cell metabolism (Adleman, 1998). Many other fields soon followed. For example, cognitive science said it studies natural information processes in the brain, physics said that quantum processes are fundamentally information processes and can be used to power quantum computers, and economics said it is an information science. In short, computing changed the epistemology of science.

Since the computational science revolutions of the 1980s, many scientific fields established computational branches that interpret natural processes as information processes and study them with computers (Kari and Rozenberg, 2008). The term "natural computing" is often used for them. It is now common in many fields to explore natural phenomena with computational models such as cellular automata, neural networks, and quantum computing.

The computational methods of science are subject to the same limitations as any computations. Computational methods do not help with problems for which computational solutions are intractable. Computing's major question "**P=NP**?" has become a fundamental question in science, too. When science becomes more computational, the limits of computation draw new boundaries for knowledge.

An interesting side effect of this transformation of science is that the early controversy about whether computer science is science has disappeared.

4. New Speculations on the Structure of the World. The enthusiasm for natural information processes has led some prominent scientists to claim that the universe itself is an information process. For example, some theoretical physicists believe that the quantum wave functions that govern all the basic particles are information processes; since all matter is build from quantum particles, they speculate that the whole world an information processing system (Dodig-Crnkovic and Müller, 2011; Dodig-Crnkovic, 2013; Fredkin, 2003). Others argue that the unreasonable effectiveness of computational models in sciences demonstrates that everything in nature computes. Molecules compute their bonds and interactions (Hillis, 1998, pp.72–73), living organisms compute life (Mitchell, 2011), the universe computes its own time-evolution (Chaitin, 2006), the universe is a cellular automaton (Zuse, 1970;Wolfram, 2002), the universe is a quantum computer (Lloyd, 2007), and everything physical is information-theoretic by nature (Wheeler, 1990; Davies, 2010). In the "it from a bit" interpretation (Wheeler, 1990), information in the form of bits – or recently qubits – is the fundamental building block of the world. There are many forms of computational accounts of the world (Piccinini, 2017). But many of those views are controversial and are not widely accepted.

All these information views of the universe give the potential of CT being a fundamental thinking tool for understanding the mechanisms of the universe.

6. Reflections

We have taken a deep dive into several fundamental aspects of computational thinking: its definition, its genealogy, its continuum from beginner to professional, and its inheritance from computational science. Our findings are based on extensive literature on computing's disciplinary history and computational thinking. Now the question is, what is important for us educators to focus on as we continue our journey with computational thinking? Here are our reflections on this.

The Importance of Our History

A number of analysts have raised warnings about the tendency to present CT devoid of its historical context (e.g., Nardelli, 2019; Guzdial, 2015; Voogt *et al.*, 2015; Denning, 2017). Why is it important to know the history of those ideas? There are many reasons. None of the ideas about CT have formed in the emptiness of a new mind-space opened in the early 2000s. The disciplinary history of computing includes many attempts to describe the unique intellectual core of computing (Tedre, 2014). There is no shortage of literature to support investigations of the origins of ideas. These histories enable us to trace CT concepts back through the beginnings of computer science in the 1950s, and in some cases back much further, hundreds or even thousands of years. In short, the ideas we work with today are distillations of the work of many people before us. Our predecessors have sharpened and honed them, adopting ideas that work and steering clear of ideas that do not work. For example, our idea of information-hiding software structures originated as the engineering idea of modules – components with a definite interface whose inner workings are hidden. Software modules can be treated like hardware modules; they can be replaced by new versions without disrupting the rest of the system. When we make ourselves familiar with the history of our ideas, we become wiser and can tell our students why things have evolved as they are.

Awareness of history can also reveal blindnesses we have in the current day. Consider the Turing machine model for computation. When he introduced his model in 1936, Turing entered a competition to provide an answer to a problem in mathematical logic. Other proposals to represent computing included the string substitution systems of Post, the lambda-calculus of Church, and the recursions of Gödel. Within a few years, it was established that each model could be simulated by any of the others, showing that they are all equivalent in their power to represent computations. The Turing model won out as the standard because its mechanical, machine-like form was the most intuitive (Kleene, 1981; Church, 1937). Later, pioneers of computing learned much about what real computers could do (or not do) by studying the capabilities and limits of Turing machines. In our thinking today, we have inherited the Turing machine notion that algorithms are step-by-step procedures carried out by machines. Few of us think of computations as string substitutions, function evaluations, or recursive functions. The Turing model may be too narrow to allow us to understand new forms of computation such as deep learning networks and quantum computers.

Programming and Machines Are Essential

Some CT proponents have tried to distance CT from the computer and a few also from programming (see discussions in, e.g., Connor *et al.*, 2017; Nardelli, 2019; Armoni, 2016; Lye and Koh, 2014; Lu and Fletcher, 2009; Shute *et al.*, 2017). We have argued that it is difficult to understand many CT concepts without understanding the machine in the background. We repeat our prior warning that attempting to define and study algorithms without reference to a computing machine creates an unrealistic image of algorithms that is disconnected with how algorithms are understood in today's broader scientific and engineering discourse. Without a machine to execute it, an algorithm is an abstract mathematical construct that cannot produce real results in the world. If there were no computers, programming would be limited to narrow theoretical uses. In today's computing, algorithm is the connection from our mental idea of what we want done to a machine that carries out our intent. Since the birth of the field, computing as a discipline has been driven by the union of algorithms and machinery.

Another casualty of treating algorithms independently of machines is an understanding of differences between what a machine can do and what a human can do. A machine can carry an enormous number of calculations in the time a human can do just one calculation. Human agents are limited by their biology: they can carry out small algorithms that complete in a few minutes or hours. It is utterly impossible for a human agent to carry out the operations of most software programs. For example, a single frame refresh for a graphics display would take a human years to calculate. Similarly, most things that are routine for humans turn out to be computationally intractable for machines. The way humans reason about problems is fundamentally different from the way computers calculate solutions to problems.

We are able to get machines to go fast because the individual calculations are completely independent of context. They are executed by circuits that respond to their inputs by well-defined local rules. Humans bring great wisdom and understanding to their jobs and decisions because through their biology they can sense the context. Humans and machines are not equivalent. Computational thinking is, to all intents and purposes, not about how to design and reason about algorithms, but about how to make machines do algorithmic tasks for people. Without the machine there would no computational thinking today. It important to keep the machine in view, even if from a distance.

Basic CT Is Not Unique to Computing

We have expressed our concern that beginner CT, which is the public face of CT, leaves out many aspects computing's rich body of knowledge. We have stressed the importance of recognizing that there is a range of CT skills from the beginner to the professional. The beginner skills emphasize basic programming and algorithm design. It is entirely appropriate for K-12 curriculum recommendations to emphasize beginner skills – because the students are beginners.

However, therein lies a dilemma: basic "computational" thinking for beginners consists of skills and concepts that are not unique to computing: most of its central concepts are found in many disciplines. The ideas that make computing unique ideas are found much further along the spectrum from basic to advanced CT. The advanced skills of professionals include designing and building large, reliable, and safe software, simulations, and artificial intelligence, as well as performance evaluation of systems, distributed networks and operating systems, and interfaces for complex systems. Our teachers need an appreciation for what professionals do because many students will ask what comes next.

Domain Knowledge is Essential to Computational Thinking

One of the conceits of CT has been a claim that CT enriches the mind and enables problem-solving in many domains. This notion, which dates back to the 1950s (Forsythe, 1959; Katz, 1960), appears to have been reinforced by the 1980s computational science movement, when scientists from many fields claimed that computing is a new way to do science. It seemed that every field of science defined a computational branch to apply computing. This gives the appearance that computing concepts entered science and transformed how science is done.

But all experience in computational sciences tells us that the participating scientists need deep knowledge of the domain in addition to their computing knowledge. Take aircraft design as an example. In the 1980s, the largest aircraft were too big for any wind tunnel facility. In collaboration with the aircraft industry, NASA embarked on “aerodynamic simulation,” meaning the simulation of air flows around the wings and bodies of planes. The objective was a full simulation of an aircraft in flight – designed by supercomputer without wind tunnel testing. This required supercomputers doing computations of advanced fluid-flow equations around the aircraft. The scientists who programmed the simulations needed deep knowledge of fluid dynamics to understand how to design grids and prevent round-off errors from accumulating. No computer science curriculum teaches computational fluid dynamics.

This is true of every domain using computers. Algorithms and systems are designed with deep knowledge of the domain. They are not simply straightforward, uncomplicated applications of computing techniques.

Basic CT is not Computer Science

It is important to avoid the trap of equating CT with the academic discipline of computing. Basic CT does not teach how professional computer scientists see the world; it consists of a set of basic ideas that are the foundation for learning many skills and concepts central to computing (and other fields). For example, basic CT does not discuss operating systems. Operating systems, which have contributed a number of fundamental ideas to computing such as autonomous processes, concurrency control, and virtual memory are a core course in a CS curriculum. They are not discussed in basic CT. The basic CT skills come nowhere near describing what an Apple Genius knows.

A less obvious but more important reason is that basic CT is a practice of the computing discipline, along with advanced CT practices including large-scale programming, design, and modeling. The discipline of computing includes all these practices and has become one of their best teachers. Basic CT is not aimed at teaching the advanced practices.

Conclusion

The latest CT wave has done a remarkable job bringing the need for K-12 computing education into the global limelight. The arguments for integrating CT in the classroom have persuaded national decision-makers, and resources flow in. The concerted effort of educators in schools has resulted in impressive advances in methods for teaching computing in schools, both with computers and without. But the CT community continues to struggle with what seems an impenetrable fog of interrelated concepts. We have argued that much of the fog would disperse if we broaden CT’s perspective to include advanced (professional) CT. Some of this broader perspective can be integrated into the upper ends of a K-12 curriculum. We have also argued that a historically grounded view of computing practices increases understanding of what works and what does not, and reveals why certain ideas have stood the test of time. With these expansions, many will come to see the full richness of the computing field.

References

Abelson, H., Bamberger, J., Goldstein, I., Papert, S. (1976). *Logo progress report 1973–1975*. Memo 356, Massachusetts Institute of Technology, AI Laboratory.

Adleman, L.M. (1998). Computing with DNA. *Scientific American*, 279(2), 54–61.

Agar, J. (2006). What difference did computers make? *Social Studies of Science*, 6(36), 869–907.

Aho, A.V. (2011). Ubiquity symposium: Computation and computational thinking. *Ubiquity*, 2011(January).

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press, Oxford, UK.

Apiola, M.-V. (2019). Towards a creator mindset for computational thinking: Reflections on task-cards. *Constructivist Foundations*, 14(3), 404–406.

Arden, B.W. (Ed.) (1980). *What Can Be Automated? Computer Science and Engineering Research Study*. The MIT Press, Cambridge, MA, USA.

Armoni, M. (2016). Computer science, computational thinking, programming, coding: The anomalies of transitivity in K-12 computer science education. *ACM Inroads*, 7(4), 24–27.

Babbage, C. (1864). *The Life of a Philosopher*. Longman, Green, Longman, Roberts & Green, London, UK.

Backus, J. (1980). Programming in America in the 1950s – some personal impressions. In: Metropolis, N., Howlett, J., Rota, G.-C. (Eds), *A History of Computing in the Twentieth Century*, pages 125–135. Academic Press, New York, NY, USA.

Baltimore, D. (2002). How biology became an information science. In: Denning, P.J. (Ed.), *The Invisible Future*, pages 43–55. McGraw-Hill, New York, NY, USA.

Barr, V. (2016). Disciplinary thinking, computational doing: Promoting interdisciplinary computing while transforming computer science enrollments. *ACM Inroads*, 7(2), 48–57.

Barr, V., Stephenson, C. (2011). Bringing computational thinking to K-12: What is involved and what is the role of the computer science education community? *ACM Inroads*, 2(1), 48–54.

Bell, T., Roberts, J. (2016). Computational thinking is more about humans than computers. *Set*, (1), 3–7.

Bers, M.U., Flannery, L., Kazakoff, E.R., Sullivan, A. (2014). Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers & Education*, 72:145–157.

Böhm, C., Jacopini, G. (1966). Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5), 366–371.

Boole, G. (1854). *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. Walton and Maberly, London, UK.

Brooks, Jr.,F.P. (1975). *The Mythical Man-Month*. Addison-Wesley, New York, NY, USA.

Brooks, Jr.,F.P. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4), 10–19.

Bullynck, M. (2016). Histories of algorithms: Past, present and future. *Historia Mathematica*, 43(3), 332–341.

Bundy, A. (2007). Computational thinking is pervasive. *Journal of Scientific and Practical Computing*, 1(2), 67–69.

Campbell-Kelly, M., Aspray, W. (2004). *Computer: A History of the Information Machine*. Westview Press, Oxford, UK, 2nd edition.

Chabert, J.-L., editor (1999). *A History of Algorithms: From the Pebble to the Microchip*. Springer-Verlag, Berlin / Heidelberg, Germany.

Chaitin, G. (2006). Epistemology as information theory: From Leibniz to . In: Stuart, S. A. J., Dodig-Crnkovic, G., editors, *Computation, Information, Cognition: The Nexus and the Liminal*, pages 2–17, Newcastle, UK. Cambridge Scholars Publishing.

Chazelle, B. (2006). Could your iPod be holding the greatest mystery in modern science? *Math Horizons*, 13(4), 14–15, 30–31.

Church, A. (1937). Review of on computable numbers, with an application to the entscheidungsproblem by a. m. turing. *The Journal of Symbolic Logic*, 2(1), 42–43.

Comer, D. (2012). *Operating System Design: The Xinu Approach*. CRC Press / Taylor & Francis, Boca Raton, FL, USA.

Connor, R., Cutts, Q., Robertson, J. (2017). Keeping the machinery in computing education. *Communications of the ACM*, 60(11), 26–28.

Cook, S.A. (1971). The complexity of theorem-proving procedures. In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, STOC ‘71, pages 151–158, New York, NY, USA. ACM.

Cooper, S. B., van Leeuwen, J., editors (2013). *Alan Turing: His Work and Impact*. Elsevier, Waltham, MA, USA.

Cortada, J.W. (1993). *Before the Computer: IBM, NCR, Burroughs, and Remington Rand and the Industry They Created*. Princeton University Press, Princeton, NJ, USA.

Dasgupta, S. (2014). *It Began with Babbage: The Genesis of Computer Science*. Oxford University Press, New York, NY, USA.

Davies, P. (2010). Universe from bit. In: Davies, P., Gregersen, N.H., editors, *Information and the Nature of Reality: From Physics to Metaphysics*, pages 65–91. Cambridge University Press.

Davis, M. (2012). *The Universal Computer: The Road from Leibniz to Turing*. CRC Press, Boca Raton, FL, USA.

Daylight, E.G. (2012). *The Dawn of Software Engineering: From Turing to Dijkstra*. Lonely Scholar, Belgium.

Daylight, E.G. (2014). A Turing tale. *Communications of the ACM*, 57(10), 36– 38.

Daylight, E.G. (2016). *Turing Tales*. Lonely Scholar, Belgium.

De Mol, L., Bullynck, M., Daylight, E. G. (2018). Less is more in the fifties: Encounters between logical minimalism and computer design during the 1950s. *IEEE Annals of the History of Computing*, 40(1), 19–45.

Denning, P.J. (2007). Computing is a natural science. *Communications of the ACM*, 50(7), 13–18.

Denning, P.J. (2016). Fifty years of operating systems. *Communications of the ACM*, 59(3), 30–32.

Denning, P.J. (2017). Remaining trouble spots with computational thinking. *Communications of the ACM*, 60(6), 33–39.

Denning, P.J., Tedre, M. (2019). *Computational Thinking*. The MIT Press, Cambridge, MA, USA.

Denning, P.J., Tedre, M., Yongpradit, P. (2017). Misconceptions about computer science. *Communications of the ACM*, 60(3), 31–33.

Dijkstra, E. W. (1968). Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, 11(3), 147–148.

Dijkstra, E.W. (1974). Programming as a discipline of mathematical nature. *American Mathematical Monthly*, 81(6), 608–612.

Dijkstra, E.W. (1980). A programmer's early memories. In: Metropolis, N., Howlett, J., Rota, G.-C., editors, *A History of Computing in the Twentieth Century*, pages 563–573. Academic Press, New York, NY, USA.

Dijkstra, E.W. (1986). On a cultural gap. *The Mathematical Intelligencer*, 8(1), 48–52.

Dijkstra, E.W. (2000). Answers to questions from students of software engineering. Circulated privately.

diSessa, A.A. (2000). *Changing Minds: Computers, Learning, and Literacy*. The MIT Press, Cambridge, MA, USA.

Dodig-Crnkovic, G. (2013). Alan Turing's legacy: Info-computational philosophy of nature. In: Dodig-Crnkovic, G., Giovagnoli, R., editors, *Computing Nature*, volume 7 of *Studies in Applied Philosophy, Epistemology and Rational Ethics*, pages 115–123. Springer, Berlin / Heidelberg, Germany.

Dodig-Crnkovic, G., Müller, V.C. (2011). A dialogue concerning two world systems: Info-computational vs. mechanistic. In: Dodig-Crnkovic, G., Burgin, M., editors, *Information and Computation: Essays on Scientific and Philosophical Understanding of Foundations of Information and Computation*, volume 2 of *World Scientific Series in Information Studies*. World Scientific, Singapore.

Eckhardt, R. (1987). Stan Ulam, John von Neumann, and the Monte Carlo method. *Los Alamos Science*, Special Issue(15), 131–137.

Ensmenger, N.L. (2001). The 'question of professionalism' in the computer fields. *IEEE Annals of the History of Computing*, 23(4), 56–74.

Ensmenger, N.L. (2010). *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. The MIT Press, Cambridge, MA, USA.

Executive Office of the President: Office of Science and Technology Policy (1987). A research and development strategy for high performance computing. Policy Document, Appendix C.

Fellows, M.R. (1993). Computer science and mathematics in the elementary schools. In: Fisher, N. D., Keynes, H.B., Wagreich, P.D., editors, *Mathematicians and Education Reform 1990–1991*, volume 3 of *Issues in Mathematics Education*. American Mathematical Society, Providence, RI, USA.

Forsythe, G.E. (1959). The role of numerical analysis in an undergraduate program. *The American Mathematical Monthly*, 66(8), 651–662.

Forsythe, G.E. (1968). What to do till the computer scientist comes. *American Mathematical Monthly*, 75(May 1968), 454–461.

Forsythe, G.E. (1969). Computer science and education. In: *Proceedings of IFIP Congress 1968*, volume 2, pages 92–106, Edinburgh, UK. IFIP.

Fortnow, L., Homer, S. (2003). A short history of computational complexity. *Bulletin of the European Association for Theoretical Computer Science*, 80.

Fredkin, E. (2003). An introduction to digital philosophy. *International Journal of Theoretical Physics*, 42(2), 189–247.

Frege, G. (1879). *Begriffsschrift: eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Verlag von Louis Nebert, Halle an der Saale.

Friedman, A.L., Cornford, D.S. (1989). *Computer Systems Development: History, Organization and Implementation*. John Wiley & Sons, Inc., Toronto, Canada.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, USA.

García-Peñalvo, F. J., Reimann, D., Tuul, M., Rees, A., Jormanainen, I. (2016). An overview of the most relevant literature on coding and computational thinking with emphasis on the relevant issues for teachers. Technical report, TACCLE3 Consortium, Belgium.

Glass, R. L. (2005). "Silver bullet" milestones in software history. *Communications of the ACM*, 48(8), 15–18.

Goldstine, H.H. (1977). *A History of Numerical Analysis from the 16th Through the 19th Century*. Springer-Verlag, New York, NY, USA.

Gorn, S. (1963). The computer and information sciences: A new basic discipline. *SIAM Review*, 5(2), 150–155.

Grier, D.A. (2005). *When Computers Were Human*. Princeton University Press, Princeton, NJ, USA.

Grover, S., Pea, R. (2018). Computational thinking: A competency whose time has come. In: Sentance, S., Barendsen, E., Schulte, C., editors, *Computer Science Education: Perspectives on Teaching and Learning in School*, pages 19–37. Bloomsbury Academic, London, UK.

Grover, S., Pea, R.D. (2013). Computational thinking in K-12: A review of the state of the field. *Educational Researcher*, 42(1), 38–43.

Guzdial, M. (2015). *Learner-Centered Design of Computing Education: Research on Computing for Everyone*. Synthesis Lectures on Human-Centered Informatics. Morgan & Claypool, San Rafael, CA, USA.

Haigh, T. (2013). `Stored program concept' considered harmful: History and historiography. In: Bonizzoni, P., Brattka, V., Löwe, B., editors, *The Nature of Computation. Logic, Algorithms, Applications*, volume 7921 of *Lecture Notes in Computer Science*, pages 241–251. Springer, Berlin / Heidelberg, Germany.

Haigh, T., Priestley, M., Rope, C. (2016). *ENIAC in Action: Making and Remaking the Modern Computer*. The MIT Press, Cambridge, MA, USA.

Hambrusch, S., Hoffmann, C., Korb, J.T., Haugan, M., Hosking, A.L. (2009). A multidisciplinary approach towards computational thinking for science majors. In: *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 183–187, New York, NY, USA. ACM.

Hamming, R.W. (1969). One man's view of computer science. *Journal of the ACM*, 16(1), 3–12.

Harel, D. (1987). *Algorithmics: The Spirit of Computing*. Addison-Wesley, Reading, MA, USA.

Hartmanis, J., Stearns, R. (1965). On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306.

Hemmendinger, D. (2010). A plea for modesty. *ACM Inroads*, 1(2), 4–7.

Hilbert, D., Ackermann, W. (1928). *Grundzüge der theoretischen Logik*. Julius Springer, Berlin, Germany.

Hillis, W.D. (1998). *The Pattern on the Stone: The Simple Ideas That Make Computers Work*. Basic Books, New York, NY, USA.

Hoare, C.A.R. (1972). Proof of correctness of data representations. *Acta Informatica*, 1(4), 271–281.

Hoare, C.A.R. (1996). How did software get so reliable without proof? In Gaudel, M.-C., Woodcock, J., editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 1–17. Springer, Heidelberg, Germany.

Hodges, A. (1983). *Alan Turing: The Enigma*. Vintage Books, London, UK.

Holmevik, J.R. (1994). Compiling SIMULA: A historical study of technological genesis. *IEEE Annals of the History of Computing*, 16(4), 25–37.

Isbell, C.L., Stein, L.A., Cutler, R., Forbes, J., Fraser, L., Impagliazzo, J., Proulx, V., Russ, S., Thomas, R., Xu, Y. (2010). (re)defining computing curricula by (re)defining computing. *SIGCSE Bulletin*, 41(4), 195–207.

Kafai, Y. B. (2016). From computational thinking to computational participation in K-12 education. *Communications of the ACM*, 59(8), 26–27.

Kari, L., Rozenberg, G. (2008). The many facets of natural computing. *Communications of the ACM*, 51(10), 72–83.

Katz, D. L. (1960). Conference report on the use of computers in engineering classroom instruction. *Communications of the ACM*, 3(10), 522–527.

Kleene, S.C. (1981). Origins of recursive function theory. *Annals of the History of Computing*, 3(1), 52–67.

Knuth, D.E. (1972). Ancient Babylonian algorithms. *Communications of the ACM*, 15(7), 671–677.

Knuth, D.E. (1974a). Computer science and its relation to mathematics. *American Mathematical Monthly*, 81(Apr.1974), 323–343.

Knuth, D.E. (1974b). Structured programming with GO TO statements. *ACM Computing Surveys*, 6(4), 261–301.

Knuth, D.E. (1981). Algorithms in modern mathematics and computer science. In: Ershov, A. P., Knuth, D. E., editors, *Algorithms in Modern Mathematics and Computer Science*, volume 122 of *Lecture Notes in Computer Science*, pages 82–99. Springer, Berlin / Heidelberg, Germany.

Knuth, D.E. (1985). Algorithmic thinking and mathematical thinking. *American Mathematical Monthly*, 92(March), 170–181.

Knuth, D.E. (1997). *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, Reading, MA, USA, 3rd edition.

Knuth, D.E., Trabb Pardo, L. (1980). The early development of programming languages. In Metropolis, N., Howlett, J., Rota, G.-C., editors, *A History of Computing in the Twentieth Century*, pages 197–273. Academic Press, New York, NY, USA.

Krajewski, M. (2011). *Paper Machines: About Cards & Catalogs, 1548–1929*. The MIT Press, Cambridge, MA, USA.

Lampson, B.W. (1983). Hints for computer system design. *SIGOPS Operating Systems Review*, 17(5), 33–48.

Larsson, P., Apiola, M.-V., Laakso, M.-J. (2019). The uniqueness of computational thinking. In: Skala, K., editor, *Proceedings of 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO 2019)*, pages 795–800, Rijeka, Croatia. Croatian Society for Information and Communication Technology, Electronics and Microelectronics.

Liskov, B.H. (1996). A history of CLU. In: Bergin, Jr., T.J., Gibson, Jr., R.G., editors, *History of Programming Languages*, volume 2, pages 471–510. ACM Press, New York, NY, USA.

Lloyd, S. (2007). *Programming the Universe: A Quantum Computer Scientist Takes on the Cosmos*. Vintage Books, London, UK.

Lockwood, J., Mooney, A. (2017). Computational thinking in education: Where does it fit? A systematic literary review. Technical report, National University of Ireland Maynooth.

Lu, J.J., Fletcher, G.H. (2009). Thinking about computational thinking. *SIGCSE Bulletin*, 41(1), 260–264.

Lye, S.Y., Koh, J.H.L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior*, 41:51–61.

MacKenzie, D. (2001). *Mechanizing Proof: Computing, Risk, and Trust*. The MIT Press, Cambridge, MA, USA.

Mahoney, M.S. (2011). *Histories of Computing*. Harvard University Press, Cambridge, MA, USA.

Mannila, L., Dagienė, V., Demo, B., Grgurina, N., Mirolo, C., Rolandsson, L., Settle, A. (2014). Computational thinking in K-9 education. In: *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference*, ITiCSE-WGR '14, pages 1–29, New York, NY, USA. ACM.

Menabrea, L.F. (1842). *Sketch of the Analytical Engine Invented by Charles Babbage*. Number 82. Bibliothèque Universelle de Génève.

Meyer, E.T., Schroeder, R. (2015). *Knowledge Machines: Digital Transformations of the Sciences and Humanities*. The MIT Press, Cambridge, MA, USA.

Mitchell, M. (2011). Ubiquity symposium: Biological computation. *Ubiquity*, 2011(February).

Nardelli, E. (2019). Do we really need computational thinking? *Communications of the ACM*, 62(2), 32–35.

National Research Council (2010). *Report of a Workshop on the Scope and Nature of Computational Thinking*. The National Academies Press, Washington, DC, USA.

National Research Council (2011). *Report of a Workshop on the Pedagogical Aspects of Computational Thinking*. The National Academies Press, Washington, DC, USA.

Naur, P. (1969). Programming by action clusters. *BIT Numerical Mathematics*, 9(3), 250–258.

Naur, P., Randell, B., editors (1969). *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, Garmisch, Germany. NATO Scientific Affairs Division: Brussels, Belgium.

Oberkampf, W.L., Roy, C.J. (2010). *Verification and Validation in Scientific Computing*. Cambridge University Press, New York, NY, USA.

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, NY, USA.

Parnas, D.L. (2011). Software engineering: Multi-person development of multiversion programs. In: Jones, C. B., Lloyd, J.L., editors, *Dependable and Historic Computing*, volume 6875 of *Lecture Notes in Computer Science*. Springer, Berlin / Heidelberg, Germany.

Pears, A.N. (2019). Developing computational thinking, "fad" or "fundamental"? *Constructivist Foundations*, 14(3), 410–412.

Piccinini, G. (2017). Computation in physical systems. In: Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Stanford, CA, USA, summer 2017 edition.

Priestley, M. (2011). *A Science of Operations: Machines, Logic and the Invention of Programming*. Springer-Verlag, London, UK.

Randell, B. (1979). Software engineering in 1968. In: *Proceedings of the 4th International Conference on Software Engineering*, ICSE '79, pages 1–10, Piscataway, NJ, USA. IEEE Press.

Rapaport, W.J. (2018). Philosophy of computer science. Unpublished Manuscript.

Repenning, A., Webb, D., Ioannidou, A. (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*, SIGCSE '10, pages 265–269, New York, NY, USA. ACM.

Resnick, M. (2017). *Lifelong Kindergarten: Cultivating Creativity through Projects, Passion, Peers, and Play*. The MIT Press, Cambridge, MA, USA.

Ritchie, D.M., Thompson, K.L. (1974). The UNIX time-sharing system. *Communications of the ACM*, 17(7), 365–375.

Rosenbloom, P.S. (2013). *On Computing: The Fourth Great Scientific Domain*. The MIT Press, Cambridge, MA, USA.

Saltzer, J.H., Schroeder, M.D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1278–1308.

Sammet, J.E. (1972). Programming languages: History and future. *Communications of the ACM*, 15(7), 601–610.

Sammet, J.E. (1981). The early history of COBOL. In: Wexelblat, R. L., editor, *History of Programming Languages*, pages 199–277. Academic Press, New York, NY, USA.

Saqr, M., Ng, K., Oyelere, S.S., Tedre, M. (2021). People, ideas, milestones: A scientometric study of computational thinking. *ACM Transactions on Computing Education*, 21(3).

Schieber, P. (1987). The wit and wisdom of grace hopper. *The OCLC Newsletter*, (167).

Sengupta, P., Kinnebrew, J.S., Basu, S., Biswas, G., Clark, D. (2013). Integrating computational thinking with K-12 science education using agentbased computation: A theoretical framework. *Education and Information Technologies*, 18(2), 351–380.

Shannon, C.E. (1937). A symbolic analysis of relay and switching circuits. Master's thesis, Massachusetts Institute of Technology.

Shute, V.J., Sun, C., Asbell-Clarke, J. (2017). Demystifying computational thinking. *Educational Research Review*, 22:142–158.

Smith, B.C. (1985). Limits of correctness in computers. Technical Report CSLI-85-36, Center for the Study of Language and Information, Stanford University, Stanford, CA, USA.

Smith, B.C. (1998). *On the Origin of Objects*. The MIT Press, Cambridge, MA, USA.

Soare, R.I. (1996). Computability and recursion. *The Bulletin of Symbolic Logic*, 2(3), 284–321.

Solomon, C.J. (1976). Teaching the computer to add: An example of problem-solving in an anthropomorphic computer culture. Memo 396, Massachusetts Institute of Technology, AI Laboratory.

Tedre, M. (2014). *The Science of Computing: Shaping a Discipline*. CRC Press / Taylor & Francis, New York, NY, USA.

Tedre, M., Denning, P.J. (2016). The long quest for computational thinking. In: *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, pages 120–129, New York, NY, USA. ACM.

Tedre, M., Denning, P.J. (2017). Shifting identities in computing: From a useful tool to a new method and theory of science. In: Werthner, H., van Harmelen, F., editors, *Informatics In The Future – In The Year 2025*, pages 1–16. Springer, Berlin / Heidelberg, Germany.

Tenenberg, J. (2018). Computational making. *ACM Inroads*, 9(1), 22–23.

Traub, J.F. (1964). *Iterative Methods for the Solution of Equations*. Bell Telephone Labs, Inc., Murray Hill, NJ, USA.

Tseytin, G.S. (1981). From logicism to proceduralism (an autobiographical account). In: Ershov, A. P., Knuth, D.E., editors, *Algorithms in Modern Mathematics and Computer Science*, volume 122 of *Lecture Notes in Computer Science*, pages 392–396. Springer, Berlin / Heidelberg, Germany.

Turing, A.M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42(1936), 230–265.

Vardi, M.Y. (2013). A logical revolution (slides for keynote). In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 1, New York, NY, USA. ACM.

Voogt, J., Fisser, P., Good, J., Mishra, P., Yadav, A. (2015). Computational thinking in compulsory education: Towards an agenda for research and practice. *Education and Information Technologies*, 20(4), 715–728.

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., Wilensky, U. (2016). Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology*, 25(1), 127–147.

Werner, L., Denner, J., Campe, S., Kawamoto, D. C. (2012). The fairy performance assessment: Measuring computational thinking in middle school. In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 215–220, New York, NY, USA. ACM.

Westfall, R.S. (1980). *Never at Rest: A Biography of Isaac Newton*. Cambridge University Press, New York, NY, USA.

Wexelblat, R.L., editor (1981). *History of Programming Languages*. Academic Press, New York, NY, USA.

Wheeler, J. A. (1990). Information, physics, quantum: The search for links. In: Zurek, W.H., editor, *Complexity, Entropy, and the Physics of Information*, pages 309–336. Addison-Wesley, Redwood City, CA, USA.

Wilkes, M.V., Wheeler, D.J., Gill, S. (1951). *The Preparation of Programs for an Electronic Digital Computer*. Addison-Wesley, Cambridge, MA, USA.

Williams, M.R. (1997). *A History of Computing Technology*. IEEE Computer Society Press, Los Alamitos, CA, USA, 2nd edition.

Wing, J.M. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33–35.

Wirth, N. (2008). A brief history of software engineering. *IEEE Annals of the History of Computing*, 30(3), 32–39.

Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media, Champaign, IL.

Yadav, A., Mayfield, C., Zhou, N., Hambrusch, S., Korb, J.T. (2014). Computational thinking in elementary and secondary teacher education. *ACM Transactions on Computing Education*, 14(1), 5:1–5:16.

Yadav, A., Stephenson, C., Hong, H. (2017). Computational thinking for teacher education. *Communications of the ACM*, 60(4), 55–62.

Zuse, K. (1970). Calculating space. Technical Translation AZT-70-164-GEMIT, Massachusetts Institute of Technology, Cambridge, MA, USA.

P.J. Denning is a Distinguished Professor at the Naval Postgraduate School in Monterey, California. In his early career he was a pioneer in virtual memory and in the development of principles for operating systems. He is also known for his work on operational analysis of queuing network systems, the design and implementation of the Computer Science Network, CSNET, ACM digital library, and codifying the principles of computing.

M. Tedre is a Professor at School of Computing, University of Eastern Finland and Associate Professor at Department of Computer and Systems Sciences, DSV, Stockholm University, Sweden. Also, Adjunct professor at the Faculty of Informatics and Design, Cape Peninsula University of Technology, Cape Town, South Africa. Currently working on the philosophy of computer science, computer science/IT education, social studies of computer science, and methodology education in computer science. More than a decade of experience in international development co-operation projects: program design, operation, and evaluation.