Computer Science: Not about Computers, Not Science

Kurt D. Krebsbach
Department of Mathematics and Computer Science, Lawrence University, Appleton, Wisconsin 54911

Abstract— *This paper makes two claims about the fundamental nature of computer science. In particular, I claim that—despite its name—the field of computer science is neither the study of computers, nor is it science in the ordinary sense of the word. While there are technical exceptions to both claims, the nature, purpose, and ultimately the crucial contributions of the beautiful discipline of computer science is still widely misunderstood. Consequently, a clearer and more consistent understanding of its essential nature would have an important impact on the awareness of students interested in computing, and would communicate a more informed perspective of computer science both within academia and in the larger society.*

Keywords: computer, science, education, algorithms

1. Not about computers

Pick up almost any book on the history of computer science and Chapter 1 will be devoted to the evolution of the machines [1]: Pascal's adding machine, the brilliant designs of Babbage's Difference Engine and Analytical Engine, Turing's Bombe and Flowers' Colossus at Bletchley Park, and the ENIAC at the University of Pennsylvania, to name a few. It is no secret that every computer scientist is fascinated by—and enormously indebted to—these and many other magnificent achievements of computing machinery.

But is our discipline called "computer science" (CS) because we use the scientific method to empirically study these physical computing machines? I believe that few computer scientists would answer affirmatively, for it is not a computing machine that is the chief object of our study: it is the *algorithm*.

1.1 The centrality of algorithms

I am certainly not the first to claim that CS has been around long before modern electronic computers were invented; or in fact, before *any* computing machines were invented.[1] Let us instead consider that CS began as a discipline when an algorithm (i.e., a procedure for achieving a goal) was first discovered, expressed, or analyzed. Of course, we can't know precisely when this was, but there is general agreement that Euclid (mid-4th century BCE) was among the first computer scientists, and that his method for computing the greatest common divisor (GCD) of any two positive integers is regarded as the first documented algorithm. As Donald Knuth—author of the discipline's definitive multi-volume series of texts on algorithms—states: "We might call Euclid's method the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day" [3]. Knuth's version of Euclid's GCD algorithm [4] is shown in Figure 1.

E1.	[Find remainder.]	Divide m by n and let r be the remainder. (We will have $0 \leq r < n$.)
E2.	[Is it zero?]	If $r = 0$ the algorithm terminates; n is the answer.
E3.	[Interchange.]	Set $m \leftarrow n, n \leftarrow r$, and go back to step E1.

Fig. 1: Euclid's 2300-year-old algorithm for finding the greatest common divisor of two positive integers m and n.

1.2 Requirements of algorithms

Let us use this simple example to state the requirements of a valid algorithm (adapted from [5]):

1) It consists of a *well-ordered* collection of steps. From any point in the algorithm, there is exactly one step to do next (or zero when it halts).
2) The language of each step is *unambiguous*, and requires no further elaboration. The "computing agent" (CA) responsible for executing this algorithm (whether human or machine) must have exactly one meaning for each step. For example, in GCD, the CA must have a single definition of "remainder".
3) Each step is *effectively computable*, meaning that the CA can actually perform each step. For instance, the CA must be able to compute (via a separate algorithm) a remainder given two integer inputs.
4) The algorithm produces a *result* (here, an integer).
5) It *halts* in a finite amount of time. Intuitively, each time E1 is executed, r strictly decreases, and will eventually reach $r = 0$, causing it to halt at E2.

Finally, a mathematical *proof of correctness* exists to show that GCD is *correct* for all legal inputs m and n, although correctness itself is not a strict requirement for algorithms. (As educators we see more incorrect, but still legitimate, algorithms than correct ones!) The importance of an algorithm's close relationship to mathematical proof will become apparent shortly.

[1] Indeed, the term "computer" was originally used to refer to a person. The first known written reference dates from 1613 and meant "one who computes: a person performing mathematical calculations" [2]. As recently as the start of the Cold War, the term was used as an official job designation in both the military and the private sector.

1.3 Algorithms as abstract objects

Note that so far we have assumed no particular computing device. No arguments about PCs vs Macs, and therefore debates over the merits of Windows, OSx, and Linux are moot. In fact, we have not even been forced to choose a particular programming language to express an algorithm.

So algorithms—our fundamental objects of inquiry—do not depend on hardware. Our interest is not so much in specific *computers*, but in the nature of *computing*, which asks a different, deeper question: What is it possible to compute? For this, Alan Turing provides the answer.

1.3.1 Turing machines

Alan Turing, rightfully considered the father of modern computing, provided us with this insight before the invention of the first general purpose programmable computer. In 1936, Turing presented a thought experiment in which he described a theoretical "automated machine" that we now call a *Turing machine*. A Turing machine (TM) is almost comically simple, consisting of an infinitely-long tape of cells upon which a single read/write head can scan left and right one cell at a time. At each step, the head can read a symbol, write a symbol, move left or right one cell, and change state. An example TM instruction is shown in Figure 2. Turing further showed that a TM, given instructions

IF	state = 3 and symbol = 0
THEN	write 1, set state to 0, move right

Fig. 2: A sample Turing machine instruction.

no more complicated than in Figure 2, can calculate anything that is computable, regardless of its complexity. Several corollaries of this result provide insight into the nature of the abstract objects we call algorithms:

- A TM can simulate any procedure that any mechanistic device (natural or artificial) can carry out.
- A TM is at least as powerful as any analog, serial or parallel digital computer. They are equivalent in power, and can implement the same set of algorithms.
- Any general-purpose programming language that includes a jump instruction (to transfer control) is sufficient to express any algorithm (Turing-equivalence).
- Not every problem has an algorithmic solution.

So anything a physical computer can compute, a TM can too. The most advanced program in the world can be simulated by a simple TM - a machine that exists only in the abstract realm…no computer required! This is an important reason why CS is not fundamentally about physical computers, but is rather based on the abstract algorithms and abstract machines (or humans) to execute them.[2]

[2]However, see Section 1.5 for more on the obvious *usefulness* of implementing and executing programs on modern computers!

1.4 Evaluating algorithms

Computer science is not only about *discovering* algorithms. Much research is also devoted to *evaluating algorithms*. Here are some questions we ask when we compare two algorithms that achieve the same goal:

1) **Correctness:** Is the result guaranteed to be a solution?
2) **Completeness:** Does it always find a solution if one exists?
3) **Optimality:** Is the first solution found always a least-cost (best) solution?
4) **Time Complexity:** How many units of work (e.g., comparisons) are required to find a solution?
5) **Space Complexity:** How much memory is needed to find a solution?

A core area of CS involves evaluating algorithms independently of computer, operating system, or implemented program. The first three questions pertain to the type of answer an algorithm yields. Note that none of the answers to these questions depend on particular computer hardware or software.

Questions 4 and 5 pertain to an algorithm's *computational complexity*, which measures the amount of some resource (often time or space) required by a particular algorithm to compute its result. An algorithm is more *efficient* if it can do the same job as another algorithm with fewer resources.

But there is a counterintuitive insight lurking here. Even concerning issues of efficiency, it is of little use to know how many resources are required of a specific *implementation* of that algorithm (i.e., *program*), written in a specific *version* of a specific *programming language*, running under a specific *operating system*, on specific computer *hardware*, possibly hosting other processes on a specific *network*. Too many specifics! Instead, we want our hard-earned analysis to transcend all of these soon-to-be-obsolete specifics. Although space constraints preclude a sufficient explanation here, the sub-discipline of computational complexity provides a well-defined way to classify algorithms into classes according to the "order of magnitude" of the resources required as a function of the input size of the problem, and provides another illustration of the centrality and abstract nature of algorithms.

1.5 Implementing and executing algorithms

One of the most important and profound differences between modern CS (as a branch of mathematics) and pure mathematics is the ability to write computer programs to *implement* algorithms and then *execute* these programs on different sets of inputs without modifying the program.

This ability—to actually *automate* computation rather than simply describe it—is by far the most practical benefit of modern computing. For this reason, it is also the most visible aspect to the general public, and even, I would claim, to other academic disciplines (who tend to equate

"computer scientists" and "coders"). But although everyone agrees on the important role of programming, programs and algorithms are not equivalent. Abelson and Sussman express this distinction eloquently in their classic text: "First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology" [6]. In other words, programming is important not only for the machine-executable code produced but also to provide an appropriate set of abstractions with which to contemplate and express algorithms.

2. Not science

The authoritative Oxford English Dictionary defines CS as "the branch of knowledge concerned with the construction, programming, operation, and use of computers." This paper takes the opposite view. How can a "science" be *primarily* about construction, operation, or use of any machine?

Fellows and Parberry argue that: "Computer Science is no more about computers than astronomy is about telescopes, biology is about microscopes or chemistry is about beakers and test tubes" [7]. And while I have always liked the accuracy and clarity of the quotation, it is true because neither science *nor mathematics* is about tools. Computers are tools, and the general-purpose computer is about as useful as a tool can get. Engineering is about tools, and computer engineering is about designing and building incredibly useful tools for CS and really for every other discipline as well.

2.1 Mathematics, science, and engineering

Many would acknowledge that what we mean by the term "computer science" is an odd blend of mathematics, science, and engineering. Paul Graham even argues for the decomposition of the field into its component parts when he admits: "I've never liked the term 'computer science'... [it] is a grab bag of tenuously related areas thrown together by an accident of history, like Yugoslavia" [8]. Similarly, John McCarthy, coiner of the term "artificial intelligence" and inventor of the second-oldest programming language (LISP), implored practitioners to keep the distinction clear: "Science is concerned with finding out about phenomena, and engineering is concerned with making useful artifacts. While science and engineering are closer together in computer science than in other fields, the distinction is important" [9].

2.2 Procedural epistemology

In emphasizing the central role of *computational process* and *abstraction* in CS, Abelson and Sussman state: "Underlying our approach to this subject is our conviction that 'computer science' is not a science and that its significance has little to do with computers" [6]. They go on to associate CS with mathematics in describing the computer revolution as "the emergence of what might best be called *procedural epistemology*", but distinguish the two by saying, "Mathematics provides a framework for dealing precisely with notions of 'what is.' Computation provides a framework for dealing precisely with notions of 'how to'" [6].

I think the brilliance of this statement lies in the term *procedural epistemology*, i.e., that fundamentally we are in search of which procedures (algorithms) *exist*, and which can not exist (as Turing and others have proven). One of the central claims of this paper is that CS is fundamentally a branch of mathematics - but not because the most important algorithms compute results that are of interest to mathematicians. It is because *algorithms themselves* are abstract objects in the same way that *proofs* are abstract objects. They are not physical, but nonetheless exist, waiting to be discovered.

2.3 Empiricism

The primary definition of "science" from Princeton's Wordnet is: "The study of the physical and natural world using theoretical models and data from experiments or observation" [10].

While I have already discussed the first part, I would argue that the second part of the definition involving the empirical aspect of science is likewise not the chief method of investigation in CS (although there are recent exceptions, as discussed in Section 3.2). Science follows the scientific method, which involves forming falsifiable hypotheses, devising experiments to test them, and observing the results with the hope of building converging evidence for or against them. This model of investigation is appropriate for observing and predicting natural (and evolving) processes, but not for eternal objects such as proofs and algorithms, which are not directly observable. Surely, as our knowledge grows, we develop a richer set of abstractions with which to express and investigate them, but the objects themselves have always been there and always will be, waiting to be "discovered" by those interested in the "how to."

Ultimately, because algorithms are abstract and exist in neither the natural (or even physical) world, and because we therefore do not use the scientific method as the primary tool to investigate them, CS is not fundamentally science.

3. Caveats

My intent in this paper is to confine my claims to what is "primary" or "fundamental" to the discipline of CS, noting in several places when claims should not be interpreted as absolute or exclusionary. I now briefly discuss several of these important exceptions.

3.1 Computer hardware

While the design and manufacture of computer hardware is more closely identified with electrical engineering, students of CS can benefit greatly from instruction in hardware design, especially in the interface between the native instruction set of the computer (software), and the various levels

of hardware implementation of those instructions; in fact, the best computer scientists are those who can cross over to related disciplines and see the same objects from multiple perspectives. Learning to think like an electrical engineer makes one a better computer scientist, and *vice versa*, but that does not make one a subset of the other.

3.2 Empirical science

While I have argued that computer scientists do not primarily employ the scientific method, the relatively new sub-discipline of *experimental algorithmics* is a promising exception to this rule. In her recent text, Catherine McGeoch explains: "Computational experiments on algorithms can supplement theoretical analysis by showing what algorithms, implementations, and speed-up methods work best for specific machines or problems" [11]. While the idea of empirically measuring the performance of implemented programs has long been in the toolbox of practicing software engineers, the idea of a *principled* study of strategies for tuning and combinatorially testing algorithms and data structures is newer, and promises to be of great practical value. However, even McGeoch explicitly suggests that experimental algorithmics fill a *supplemental* need in the discipline, and that the student can only benefit from these new methods by first having a solid working knowledge of the principles of algorithm analysis and data structures.

4. Societal impact

Despite serving as an umbrella term for a group of variously related efforts involving mathematics, science, and engineering, the central questions of CS revolve around abstract objects we call *algorithms*: identifying them, discovering them, evaluating them, and often implementing and executing them on physical computers. Thoughtfully unpacking the confusion between computers and computing can have important benefits for how the larger society—including future computer scientists—perceives of and ultimately considers our useful and fascinating discipline.

4.1 Public perception

At one end of the public spectrum are those who see computer scientists as *super users*; humans who have formed a special bond with computers (if not other humans?) and are therefore capable of inferring the correct sequence of menus, settings, and other bewildering incantations without ever having read the manual. Fortunately, as society has become more technically savvy this misunderstanding is fading, and perhaps the majority instead think that computer scientist is another name for *super programmer*.

Of course, this misconception is less problematic, as programming *is*, in fact, an important aspect of CS, as described in Section 1.5. And, in fact, many computer scientists *are* expert programmers. But, useful as they are, programs are not the core of CS. Algorithms exist regardless of whether someone has coded them, but not *vice versa*. Algorithms come first. There are excellent computer scientists whose research does not involve much programming (e.g., analysis of algorithms), just as there are excellent programmers who might not do much interesting CS beyond coding to detailed specs. But the conflation of these terms can lead people to mistakenly believe that an undergraduate curriculum in CS should consist of writing one big program after another, or learning a new programming language in every new course. These are simply not accurate reflections of the work computer scientists actually do.

4.2 Undergraduate expectations

My experience as an academic adviser has helped to motivate this paper. With the proliferation of mobile computing and omnipresent Internet, there *appears* to be a lot of enthusiasm for our field, but I often find that eager undergraduates are either excited about something that is *not* CS (gaming comes to mind), or stems from an advanced topic that definitely *is* CS (e.g., machine learning), but requires a strong foundation in algorithms, data structures, and programming abstractions to place the advanced concepts in context. Understanding the state of the art gives promising students the best opportunity to contribute something genuinely new.

Unfortunately, students have been led to believe that there are massive shortcuts to these fundamentals... online tutorials, cut-and-paste code, developer boot camps, and so on. And while each of these methods can have their uses, they are no substitute for the foundation that will serve a student throughout an entire career. In rare cases, shortcuts result in short-term success, but fundamental abstractions provide the foundation upon which to continue building and adapting throughout a lifetime of learning and innovating.

References

[1] P. E. Ceruzzi, *Computing: A Concise History*. The MIT Press, 2012.

[2] D. A. Grier, *When Computers Were Human*. Princeton, NJ, USA: Princeton University Press, 2007.

[3] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[4] ——, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[5] G. M. Schneider, J. Gersting, and S. Baase, *Invitation to Computer Science: Java Version*, 1st ed. Pacific Grove, CA, USA: Brooks/Cole Publishing Co., 2000.

[6] H. Abelson, G. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[7] M. R. Fellows and I. Parberry, "SIGACT trying to get children excited about CS," *Computing Research News*, vol. 5, no. 1, p. 7, Jan. 1993.

[8] P. Graham, *Hackers and Painters: Essays on the Art of Programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2004.

[9] J. McCarthy, "Merging CS and CE disciplines is not a good idea," *Computing Research News*, vol. 5, no. 1, pp. 2–3, Jan. 1993.

[10] Princeton University. (2010) About wordnet. [Online]. Available: http://wordnet.princeton.edu

[11] C. C. McGeoch, *A Guide to Experimental Algorithmics*, 1st ed. New York, NY, USA: Cambridge University Press, 2012.