Original Research Article

Big Data & Society
July–December 2019: 1–15
© The Author(s) 2019
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/2053951719858743
journals.sagepub.com/home/bds
SAGE

Occluded algorithms

Adam Burke

Abstract

Two definitions of algorithm, their uses, and their implied models of computing in society, are reviewed. The first, termed the structural programming definition, aligns more with usage in computer science, and as the name suggests, the intellectual project of structured programming. The second, termed the systemic definition, is more informal and emerges from ethnographic observations of discussions of software in both professional and everyday settings. Specific examples of locating algorithms within modern codebases are shared, as well as code directly impacting social and ethical concerns. The structural distinction between algorithms and social concerns is explained as mirroring the engineering construct of algorithms and data structures. It is proposed that, rather than this separation being an attempt to enforce a professional boundary and evade social responsibility, it is a crucial technical distinction within code which makes it clearer and more transparent. The power structures reinforced by the broader, cultural interpretations of algorithm are reconsidered, along with what it would mean for software to have an inclusive design culture.

Keywords

Algorithm, structured programming, critical algorithm studies, reuse, technical culture

Introduction

It is harder than you might at first expect to find algorithms in modern software. This may sound absurd. Software is nothing but sequences of predefined steps, executed mechanically. That definition makes any software an algorithm, in the sense of a recipe, the way it is often informally explained. Yet, we can also say that mergesort is an algorithm with $O(n \log n)$ worst case time complexity for an input of n items. The latter is a much narrower usage than the former, and between the two definitions lies a gulf of meaning in which many worthy concerns may drown.

When we analyze an axehead as an artifact of a material culture, we can consider the systems around it in multiple ways. We can study the way the axe handle slots into the axehead's socket in a specific technical machine. We can also consider the physics of how the components interact mechanically, the economy that the axe and its components are traded in, what is usually cut with the axehead, the people, animals, and social groups that participate in its construction, and other larger socio-technical systems. Using the structured programming definition, introduced below, algorithms are locatable objects in a material culture in this same sense. The maker of an axehead will need specific technical skills. Axes are tools for construction, harvesting, or war, and axe handles and axeheads may be designed with one of those specific uses in mind. An axehead, like an algorithm, is also a tool for making other tools.

The specific technical definition of algorithm used in this article is from Cormen et al. (2009: 5):

> Informally, an *algorithm* is any well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*. An algorithm is thus a sequence of computational steps that transform the input into the output. We can also view an algorithm as a tool for solving a well-specified *computational problem*. The statement of the problem specifies in general terms the desired

New Centre of Research and Practice, Grand Rapids, USA

Corresponding author:
Adam Burke, New Centre of Research and Practice, 4417 Broadmoor Avenue SE, Grand Rapids, MI 49512, USA.
Email: adam.burke@thenewcentre.org

input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

This is more specifically a software engineering definition, from the intellectual movement of structured programming. Transformations in how software is built over the last few decades have moved the location of algorithms within codebases, making them less obviously visible, while no less important. This is comparable to the historical transformation and componentization of the photographic camera, from room-sized *camera obscura*, to hand-held chemical reactors using film, to electronic components visible within other devices only by the small circle of the exposed lens.

Those impacted by modern software, including most humans on the planet, also need to describe decisions made in software platforms. This is a second use of the word algorithm. People talk of the Yelp moderation algorithm being broken (Diakopolous, 2013), or Google Search algorithms being anti-sex (Blue, 2019). This usage is both broader and looser than the technical sense of a particular formally analyzable protocol. Academics outside computer science or engineering analyze and criticize the way software and societies shape one another. Ride-share drivers may even talk of being ''blessed by the algorithm'' (Coleman, 2018); that they didn't cite a technical textbook correctly doesn't remove their dependence on software-conditioned whim to get paid, and put food on the table. This systemic, or popular, definition of algorithm describes an outsider perspective on decisions made by a complex, opaque, computationally mediated system. Although in the context of marking an undergraduate software engineering assignment, these examples would be treated as misuse, in practice, the horse has bolted: these broad definitions are in use in the world. Specific examples of this usage, including by users, technical professionals, and corporate representatives, are discussed in the ''System'' section below.

A number of authors, including Dourish (2016: 2–3), Mittelstadt et al. (2016: 2), and Yeung (2018: 3–4) have recognized that differing interpretations of algorithm exist, some more aligned with computer science, some with social science and public policy. These latter definitions tend to refer to larger sociotechnical systems in broad terms, much like the popular usage above. Seaver's ''Algorithms as culture: Some tactics for the ethnography of algorithmic systems'' (2017) represents perhaps the most expansive of these interpretations. After presenting a thoughtfully grounded understanding of possible problems with algorithm as a term, and insights based on ethnographic work at a software firm, it advocates a single multifarious definition as a solution (Seaver, 2017: 5):

> We can call this the *algorithms as culture* position: algorithms are not technical rocks in a cultural stream, but are rather just more water. Like other aspects of culture, algorithms are enacted by practices which do not heed a strong distinction between technical and non-technical concerns, but rather blend them together. In this view, algorithms are not singular technical objects that enter into many different cultural interactions, but are rather unstable objects, culturally enacted by the practices people use to engage with them.
> [. . .]
> Algorithms are multiple, like culture, because they *are* culture.

Seaver's *algorithms as culture* proposes to, if not quite to replace, then at least to subsume, narrower technical uses of the word by programmers. Seaver's framework serves as a useful contrast to technical definitions, and a conceptual limit for non-technical interpretations of algorithm.

Science and Technology scholars such as Winner (2010: 5–6) have described how technologies fade into a social background with use. Algorithms, though, have become objects of public discussion and concern just as they have become less visible in everyday programming practice. Rather than grouping these disparate usages under one broad cultural definition, this article expands on the model of two definitions, describing two distinct objects. In the intellectual superstructure of software engineering, ''algorithm'' is load-bearing jargon. Locatable, distinguished, transparent algorithms serve the goal of building transparent, understandable, and reliable software systems. This is often well aligned with the social concerns that programmers and non-programmers have with the design of software systems. Uses of broader cultural or systemic definitions, especially by technical professionals and organizations, do not do the same design work.

The structured programming definition of algorithm is here illustrated using three perspectives, in three sections, ''Structure'', ''Code'' and ''Reuse''. ''Structure'' shows the historical sharpening of the term algorithm by structured programming. ''Code'' shows distinctions between algorithms and other code in example programs. ''Reuse'' describes the encapsulation of algorithms in reusable libraries, which has made them less immediately visible. Two sections discuss the systemic definition, ''System'' and ''Culture''. In ''System'', examples of using algorithm in a broad systemic sense are analyzed. These cross technical, user and academic communities, as well as corporations and governments, and reflect both existing power structures and the opacity of large computational systems. In the section, "Culture", the algorithms as culture position is critiqued, alternative approaches are discussed, and structured programming is considered as a starting point for a shared technical culture.

Structure

Structured programming is a set of programming techniques, and a historical intellectual movement. It advocated for structure in two main ways. Firstly, for a much more limited vocabulary of instructions, organized in relatively small, reusable parts, to be used in writing programs. Secondly, for a structured, top-down planning and mathematical proof process to be used in building large software systems. The first stream, restructuring programming languages, was a spectacular success and is very much a foundation for all software engineering today, including more-or-less successor movements such as object-oriented or functional programming. The second stream, top-down planning and formal proof, suffered from its excesses of high-modernist planning and was finished off in the early 2000s by counter-movements promoting iterative development by small teams, notably Agile Software Development (Cockburn, 2002).

"If structured programming can be thought of as a revolution, then surely Dijkstra's landmark paper, "Programming Considered as a Human Activity," published in 1965, marks its beginning," reads the introduction to "Classics in software engineering" (Yourdon, 1979: 1). From the title itself, it is apparent that this was not a purely technical intervention, but a thoughtful discussion on the way humans work with and understand these new machines. Dijkstra (1979), explains that the complexity of large-scale programs escalates so rapidly that reliability can only be achieved by assembling them from small, understandable parts. He goes on to advocate for specific design constraints, such as the elimination of the GOTO instruction, which jumps from one code point to another without an easily visible relation in program text-space; and the elimination of self-modifying code. In the more detailed article "Notes On Structured Programming" (Dijkstra, 1972), the technical notes on program proof and structure are deliberately prefaced by a reflective essay on the psychological relationship of programmers to code, and the social responsibility of building good software systems. Context of this kind is also weaved through the more technical discussion. Certain passages could serve almost verbatim as cultural criticism of technologists today (Dijkstra, 1972: 16):

> In my life I have seen many programming courses that were essentially like the usual kind of driving lessons, in which one is taught how to handle a car instead of how to use a car to reach one's destination.
>
> My point is that a program is never a goal in itself; the purpose of a program is to evoke computations and the purpose of the computations is to establish a desired effect.

Dijkstra had important predecessors, such as Hopper's advocacy and invention of programming language structures that follow human language patterns (Hopper and Mauchly, 1953), and the theoretical result of the Structured Programming Theorem (Böhm and Jacopini, 1966). He also had peers and students in what became a broad intellectual project. It is as part of this movement that programming language designer Nikolas Wirth wrote a textbook called $Algorithms + Data\ \ Structures = Programs$ (Dourish, 2016: 2; Wirth, 1976). Parallel advances in mathematical computer science also made it possible to show the relationship between time taken for an algorithm to complete and the size of the data input, called algorithmic complexity (Knuth, 1976; Cormen et al., 2009: 23–28). (Algorithmic complexity is discussed further in the "Code" section.)

Introna (2016: 5), in a discussion of software and governmentality, describes the line distinguishing algorithms from other elements of code as an "agential cut," one that divides up an assemblage according to perspectives of the agents expected to deal with it. Introna takes the term "agential cut" from Barad (2007: 139–140), describing any particular cut as both arbitrary and performative. That the cut is arbitrary is, in a way, running Dijkstra and the Structured Programming Theorem in reverse: the same mechanical function could be achieved by code in a significantly different textual shape. It could even be "spaghetti code," programming idiom for difficult to understand code where lines of causation are unclear. The structured programming cut is also performative, in that it is constitutive of the elements so constructed. For Barad (2007), the agential cut is pervasive and sustained by ongoing performance: "The line between subject and object is not fixed, but once a cut is made (i.e. a particular practice is being enacted), the identification is not arbitrary but in fact materially specified and determinate for a given practice." (155, 160). Cuts are made by humans and non-humans to constitute what a salmon is, or an axehead, or an axe handle, or an algorithm.

Discussions of algorithms including Introna (2016: 4) and Cormen et al. (2009) have a tendency to show the code left after the cut, or not show the code at all. This is understandable – one of the problems of discussing code is dealing with text at scale – but it does have the side effect of making the cut a decontextualized border around some apparently arbitrary code, or an abstract entity. Algorithms are identifiable, concrete media artifacts, and they are easier to identify when seen as part of a larger machine, as in the examples in the following section. The "structure" introduced by "structured programming" is introduced through cuts of this kind: separating algorithms from data structures, input data from programs, or named constants from arbitrary numeric values.

Code

In inspecting algorithms in code, the most visible aspect is "an algorithm is thus a sequence of computational steps that transform the input into the output" (Cormen et al., 2009: 5). This also suits formal reasoning about computational properties.

Sorting algorithms make excellent examples for this point of view, though any of the classes of problem in a standard textbook would do. Sorting algorithms are widely documented and analyzed, clearly relatable to real world examples, the simple ones are fairly easy to understand, and there are many substitutable implementations. For reasons of accessibility, we use bubble sort (Cormen et al., 2009: 40), presented in Figure 1, with the caveat that it is almost solely a teaching tool in practice.

Bubble sort has worst case time complexity of O(n^2). This is to say, the time taken to execute the algorithm is proportional to the square of the number of elements in the list, n. The worst case occurs when the list is in reverse sorted order: informally, every single member of the list gets "bubbled" n-x times down the remainder of the list, therefore n*(n-x) operations occur, and the smaller terms are dropped in O() notation. The worst case space complexity is O(n) because the size of the data structure used never exceeds the original array plus one or two spots for buffer, and again constants are abstracted out.

The cut effected by the technical term algorithm is seen in the textual boundary of the sort() function. The remaining code (assertEquals() and below) is a harness for the sort algorithm, establishing a larger machinic assemblage, in this case one that shows a programmer, tester, or continuous build infrastructure evidence of the sort working as expected.

To avoid the impression that bubble sort in an imperative language is the only algorithm that exists in practice, consider the code in Figure 2 for searching lists and trees.

In this case, two algorithms and data structures are compared. The two algorithms are represented in the inTree() and inList() functions and show a trade-off between space cost and time. As the tree is in order, and data spread in a balanced way among nodes, searching with the inTree() function is now more dependent on the height of the tree, and an item can be found in O(log n) time, with irrelevant parts of the tree skipped. By skipping some elements, the time taken is proportional to the logarithm of the number of elements, n. This is faster than with a simple list, where every item needs to be checked, taking O(n) operations. The space cost is higher: we now allocate O(n log n) memory to hold this data, rather than O(n) for a simple list. A more detailed introduction to tree data structures is in Cormen et al. (2009: 286–288).

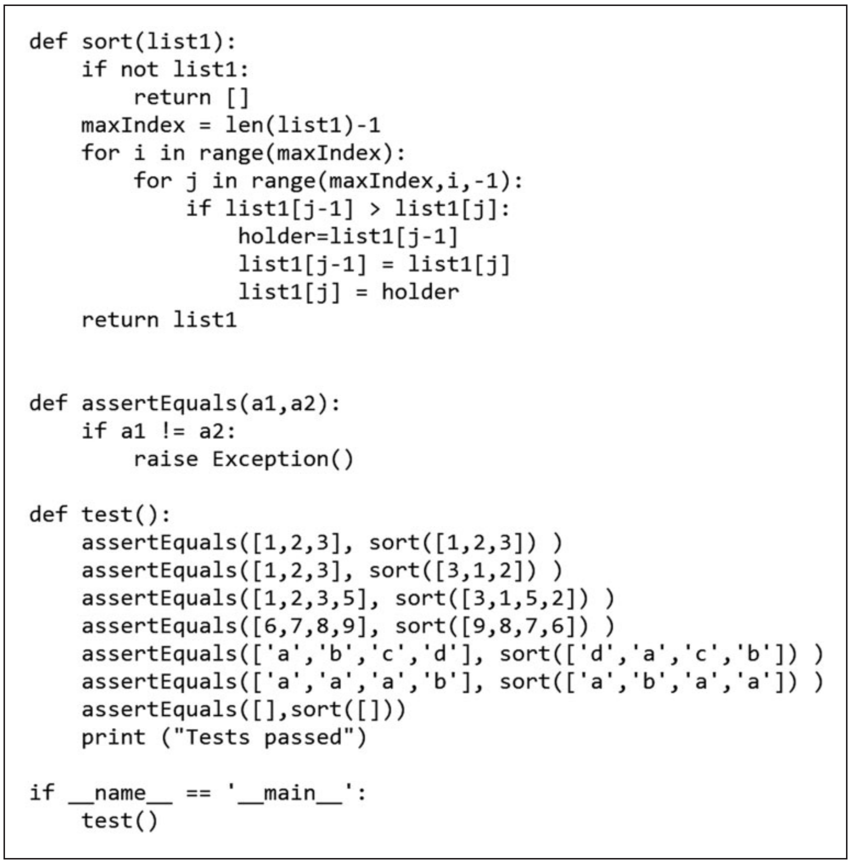
Little or no reference is made here to content: for sorting and searching, that is abstracted behind the comparators, "<," ">," and "==", with similar constructs existing in many modern languages. Usually, the comparator will be associated with the formal type of the items being sorted, or specifically nominated by the code invoking the sort() function. This function is already available for us on list objects in Python (Python Software Foundation, 2019), making our reimplementation in Figure 1 redundant in practice, and the code in Figure 3 more conventional.

In the examples in Figures 1–3, Python and Haskell provide default comparators that map to a commonly understood ordering of integers. For types defined by the programmer, rather than the language itself, choices have to be made on how to implement the comparator, as in Figure 4.

The formal properties of time and space complexity will remain in the equivalent implementations of bubble sort in Java, C++, the re-implementation of Python that uses Chinese character keywords (Glaze, n.d.), or the esoteric language that requires code to be written in a form of iambic pentameter (Hasselström and Åslund, 2001). Each of these technological artifacts might certainly have interesting revelations from a cultural "deep reading," but the underlying structure of bubble sort is more akin to a physical or mathematical law about how "things" can be arranged. To emphasize this, textbooks or papers on algorithms often use an abstract programming language not implemented by a real compiler (Cormen et al., 2009; Knuth, 1997). Whether in an abstract or concrete programming language, and as seen in Figures 1 and 2, the algorithm is defined independently of whether the things are people, debts in euros, or rice fields.

The choice to arrange things in an order at all could potentially be regarded as a "computationalist" value scheme, rather than value neutral. It is not clear how such a view would be applied to the technical, ethical, or cultural design concerns in working with software systems, however, so the zeroing of this particular scale is left for others to consider.

Going back to our comparator example, we move closer to a specific domain, and our choices as programmers become much more value laden in the code in Figure 5.



```python
def sort(list1):
    if not list1:
        return []
    maxIndex = len(list1)-1
    for i in range(maxIndex):
        for j in range(maxIndex,i,-1):
            if list1[j-1] > list1[j]:
                holder=list1[j-1]
                list1[j-1] = list1[j]
                list1[j] = holder
    return list1


def assertEquals(a1,a2):
    if a1 != a2:
        raise Exception()

def test():
    assertEquals([1,2,3], sort([1,2,3]) )
    assertEquals([1,2,3], sort([3,1,2]) )
    assertEquals([1,2,3,5], sort([3,1,5,2]) )
    assertEquals([6,7,8,9], sort([9,8,7,6]) )
    assertEquals(['a','b','c','d'], sort(['d','a','c','b']) )
    assertEquals(['a','a','a','b'], sort(['a','b','a','a']) )
    assertEquals([],sort([]))
    print ("Tests passed")

if __name__ == '__main__':
    test()
```

Figure 1. Bubble sort implemented in the Python programming language, with some test scaffolding.[1]

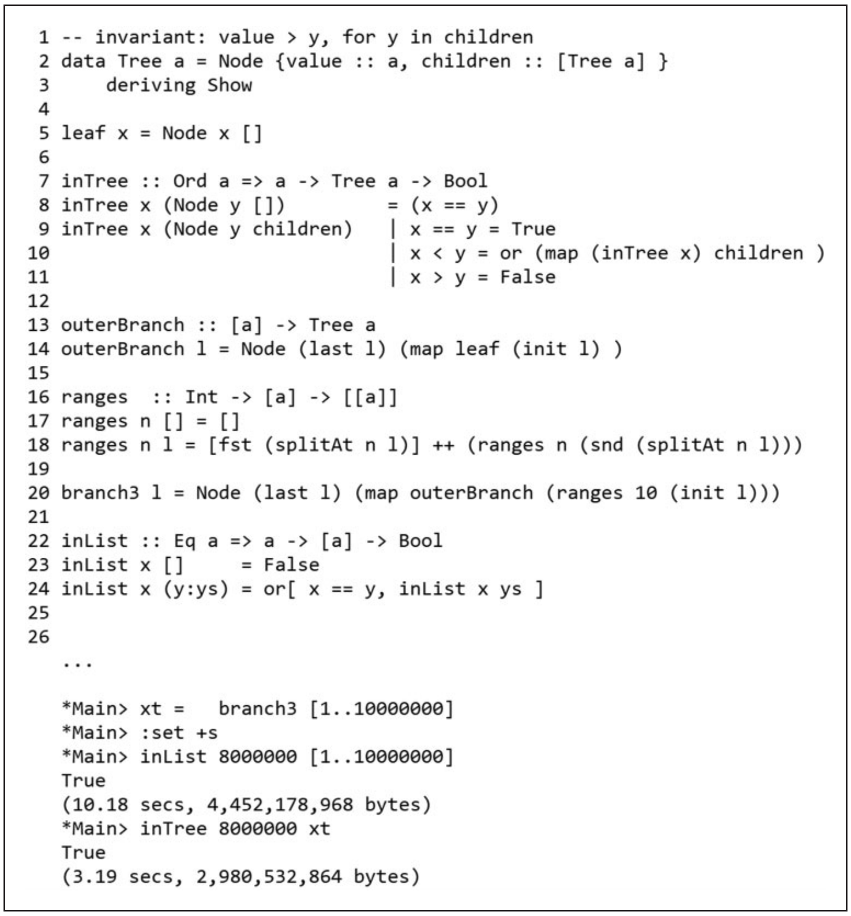
Is this discrimination ethically right? We'd have to know more about the system, but it is certainly laden with values. It might be sorting students into primary school classes, or redlining Buddhists into more expensive mortgage financing options – we don't know without that systemic context.

From a technical point of view, there's not much to analyze. There is a small number of constant time operations on primitive values. If this was identified as something that might often vary in the system, the choice about which comparator to use could be made configurable. Such settings would not be coded in a programming language at all, but found in a different location, such as a configuration file or database, making it even more decoupled from the code itself, even less algorithmic.

These more precisely located and contextual examples are intended to aid navigation and critique of software, and suggest why centering discussions of bias on the concept ''algorithm'' can be counter-productive. The value-laden nature of software and big data systems has been a topic of social science, policy, and computer science research in recent years, often from a perspective of discrimination or bias (Citron and Pasquale, 2014: 13–16; Crawford, 2013; US Executive Office of the President, 2016). Žliobaitė (2017) also reviews mathematical techniques for analyzing bias in programs given a range of inputs and outputs. The structured programming concept of an algorithm does not contradict these findings that software systems in general often reflect the values of the people, infrastructure, and society that created them, because algorithms are executable only with infrastructure around them. For example, viewing a page ranking algorithm as an encapsulated component of a larger platform is compatible with the argument by Noble (2018: 31–32 and throughout) that Google Search presents a media artifact whose representation of race or gender reinforces existing structural and historical discrimination in the United States. The title, *Algorithms of Oppression*, is harder to parse except with a systemic or cultural definition of an algorithm.

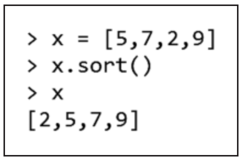
Reuse

But the principle behind all technology is to demonstrate that a technical element remains abstract, entirely underdetermined, as long as one does not relate it to an *assemblage* it presupposes. (Deleuze and Guattari, 1987: 397)



```
-- invariant: value > y, for y in children
data Tree a = Node {value :: a, children :: [Tree a] }
   deriving Show

leaf x = Node x []

inTree :: Ord a => a -> Tree a -> Bool
inTree x (Node y [])          = (x == y)
inTree x (Node y children)    | x == y = True
                              | x < y = or (map (inTree x) children )
                              | x > y = False

outerBranch :: [a] -> Tree a
outerBranch l = Node (last l) (map leaf (init l) )

ranges  :: Int -> [a] -> [[a]]
ranges n [] = []
ranges n l = [fst (splitAt n l)] ++ (ranges n (snd (splitAt n l)))

branch3 l = Node (last l) (map outerBranch (ranges 10 (init l)))

inList :: Eq a => a -> [a] -> Bool
inList x []     = False
inList x (y:ys) = or[ x == y, inList x ys ]


...

*Main> xt =   branch3 [1..10000000]
*Main> :set +s
*Main> inList 8000000 [1..10000000]
True
(10.18 secs, 4,452,178,968 bytes)
*Main> inTree 8000000 xt
True
(3.19 secs, 2,980,532,864 bytes)
```

Figure 2. Sorted tree search in the Haskell programming language, loosely based on example from the Haskell tutorial (Thomas, 2000), with example usage at console.[2]



```
> x = [5,7,2,9]
> x.sort()
> x
[2,5,7,9]
```

Figure 3. Conventional Python sort() usage.

Very few developers write sorting algorithms any more. Over the last few decades, an idea that much software should be assembled from existing, reusable, parts flowered into everyday reality for the working programmer. What was aspirational for the school of structured programming around Wirth and Dijkstra became more widespread with the provision of standard libraries with popular languages in the 1990s and early 2000s. For example, version one of Java, in 1995, shipped with an extensive set of standard libraries (Kramer, 1996). Later, package management tools and open source libraries made many capabilities more quickly and reliably available when reused, rather than written from scratch. Sort functions available with platform libraries might be variants of algorithms such as mergesort or quicksort, which have provably better performance over more naive approaches like bubble sort.

Today, a lot more of the work of a programmer is stitching together existing components in a way that makes sense for that particular domain and application, calling functions like sort() as needed. This makes algorithms less visible on the surface of code worked on in everyday professional settings. As Deleuze and Guattari (1987: 397) describe, sort() is a technology both formal and abstract (in viewing the code text, it is shorn of value references beyond moving information), and concrete and executable within the right assemblage. This incorporation of standard algorithms into standard libraries, then found in larger machines, is similar to componentization in industrial electronics. A camera has transformed from a room-sized object to something smaller than a thumbnail, accessed as a component by apps within a smartphone.

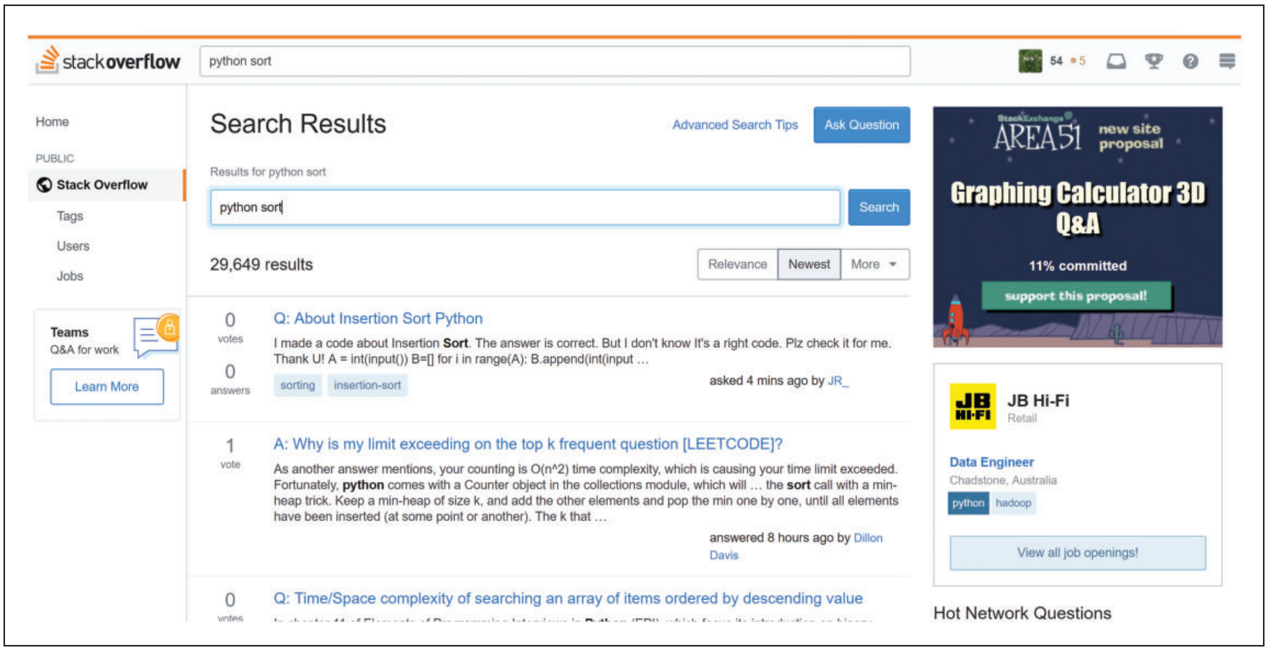
```
class Student:
    def __init__(self, name, grade, age):
        self.name = name
        self.grade = grade
        self.age = age
    def __repr__(self):
        return repr((self.name, self.grade, self.age))

student_objects = [
    Student('john', 'A', 15),
    Student('jane', 'B', 12),
    Student('dave', 'B', 10)]

>>> sorted(student_objects, key=lambda student: student.age)   # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

Figure 4. Python sorting, in lexical order of name, grade and age, and example use, from Python documentation (Dalke and Hettinger, 2018).

```
from functools import total_ordering

@total_ordering
class Person:
    def __init__(self, firstname, lastname, religion):
        self.firstname = firstname
        self.lastname = lastname
        self.religion = religion

    def __repr__(self):
        return repr((self.firstname, self.lastname, self.religion))

    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname") and
                hasattr(other, "religion"))

    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.religion.lower(), self.lastname.lower(),
                    self.firstname.lower()) ==
                (other.religion.lower(), other.lastname.lower(),
                    other.firstname.lower()))

    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.religion.lower(), self.lastname.lower(),
                     self.firstname.lower()) <
                (other.religion.lower(), other.lastname.lower(),
                     other.firstname.lower()))
```

Figure 5. Comparator from the Python documentation (functools, n.d.), with religion added.

The process of building an application of significant size involves decomposing it into identifiable parts at multiple scales. The compiler or interpreter provides a layer of translation from human-comprehensible code to machine instructions. Small-scale structures built with the language are functions and types (/classes), these assemble into libraries, APIs and layers within services, and then stacks or networks of services consist an overall computational system. Choosing where and how to separate concerns into different structures, or to



29,649 results

0 votes 0 answers **Q: About Insertion Sort Python**
I made a code about Insertion **Sort**. The answer is correct. But I don't know It's a right code. Plz check it for me. Thank U! A = int(input()) B=[] for i in range(A): B.append(int(input …
sorting insertion-sort
asked 4 mins ago by JR_

1 vote **A: Why is my limit exceeding on the top k frequent question [LEETCODE]?**
As another answer mentions, your counting is O(n^2) time complexity, which is causing your time limit exceeded. Fortunately, **python** comes with a Counter object in the collections module, which will … the **sort** call with a min-heap trick. Keep a min-heap of size k, and add the other elements and pop the min one by one, until all elements have been inserted (at some point or another). The k that …
answered 8 hours ago by Dillon Davis

0 votes **Q: Time/Space complexity of searching an array of items ordered by descending value**

Figure 6. Search on Stack Overflow for "python sort," 5 March 2019.

assemble them into a related structure, is a core, perhaps the core, design activity of programming.

Simondon (1980: 28, 2016: 35) conceived technical objects as evolving by going through a process of differentiation and concretization. For Barad (2007), also "[c]hanging patterns of difference [. . .] are that which effects, or rather enacts, a causal structure" (137). Wirth's separation of *Programs* = *Algorithms* + *Data Structures* is an evolutionary step from previous programs where they were mixed together, differentiating two parts, making each more concrete. Feathers (2004: 29–33), in the context of improving existing code, describes places such as classes, functions, or preprocessor inclusions as "seams" where tests can be introduced and concepts confined. Each of these small-scale design choices, of which hundreds may happen during a programming day, are effectively agential cuts. A key agent in this case is "myself, or some other programmer, in one hour's time." Managing scope through structure is a way of working within the constraints of the limited human mind. As Dijkstra (1970: 3) explains, "as a slow-witted human being I have a very small head and I had better learn to live with it and to respect my limitations and give them full credit."

When it is well organized in this way, components of a system are more available for reuse, easier to test, more analyzable, and more maintainable: more under control.

Protocols like TCP/IP, which perform the low-level data communications underlying the Internet (Socolofsky and Kale, 1991) are also algorithmic, but distributed across multiple machines and processes, often combining multiple algorithms into a larger assemblage, requiring more careful cuts to identify.

Calling established library functions separates concerns of moving information from other concerns. For instance, the detail of the TCP/IP protocol is handled in network libraries that almost never need to be altered while developing a website.[3] Similar analysis can apply to moving information across services. Although the division of responsibilities across services is often political (Bailey et al., 2013), this architectural information flow can also be analyzed from a rather algorithmic point of view, when the names of the services are ignored, much like separating the comparator from the sort function. Tool sets for large-scale, distributed software systems using streaming data are still maturing, and the technical jargon and understanding around them has a corresponding instability. Systems like Apache Kafka (Kreps et al., 2011) are adapting the broad intellectual project of structured programming in this space. Processing a stream of events may be better related to functional programming, as in the Haskell example above, than imperative programming, as in the Python examples. The emergence of new structures, readily decoupled from more value-laden context, and formally analyzable for their computational properties, matches the historic experience of the software profession that algorithms are not missing from this new domain, but found in different shapes.

Machine learning practice establishes a new extreme of separation: it is a computational statistical technique for partitioning large sets of data. This makes possible rapid classification of new inputs from the learning domain. Although important algorithmic innovations happen in the space, most machine learning work uses toolkits like TensorFlow (Abadi et al., 2016). The design and development work is in the selection of the type of algorithm, the choice of training data set, the parameters used to train the model, and how to release the result for use in other systems; there are even tools which present this as a networked service (Golovin et al., 2017).

Lots of less controlled code exists, and programmers deal with it all the time. Structuring code is a learned skill, and confused structure rapidly and cruelly compounds, so it is particularly easy for an inexpert programmer to create a mess. Code which rewrites a readily available algorithm without good reason would be reviewed by another programmer as redundant. That which mixes domain-specific names or constants with more general code could be reviewed as "hardcoding," or not following a separation of concerns.

The structured programming definition of algorithm is powerful because it allows developers to locate and manage risks. The locations of those risks are the code choosing which algorithm to invoke, and the data feeding it. The ethical risks of embedded bias and social influence are not to be found in the sorting algorithms but in the comparator that sorts Anglicans above Buddhists, not in deep learning tensor arithmetic but in the training dataset sourced from prisoners from Shanxi province, not in the graphical rasterizing algorithm but in the palette choices that exclude color-blind users.

The most algorithmic code, where formal algorithmic analysis is most valuable, is also that least value laden because it is organized in a way that separates the management of cultural values into a separate scope. The values dictating the overall system will of course shape it, but the carefully managed scope ensures an understandable, reliable implementation. This parallels the conclusion, in "Is There An Ethics of Algorithms?," that the system should "as far as possible remain open to the user's ethical preferences" (Kraemer et al., 2011: 259). Analyzing medical imaging software, they point to parameterization of a threshold as a key point of ethical intervention – configuration, not the algorithm proper. Separating algorithm and configuration in turn makes transparency, and changing the code that expresses those cultural values, easier. The algorithm is an axe head, the wrapping code the axe socket, and the configuration the handle. Likewise a camera component (the algorithm) can be embedded in many hardware devices (the wrapping code), and its behavior is changed by a user's preference to never use the flash (configuration).

System

In a world where software is a pervasive tool, people rightly want to analyze and discuss cultural and ethical aspects of software's design and use. "Critical algorithm studies" is one term for the corresponding academic field (Gillespie and Seaver, 2016); terms like "algorithmic ethics" can also apply. A much broader definition of "algorithm" can be seen in this milieu, across popular usage, the press, policy discussions, and critical algorithm studies. Technologists, spokespeople for software companies, and governments can also be found using algorithm broadly and imprecisely, when compared to the structured programming definition. These examples all relate the perspective of a disempowered user outside an opaque decision-making system.

Using the term algorithm in popular discourse is a recent historical phenomenon, dating to the advent of cloud platforms and social networks. Sandvig (2014) notes that in 1997 the term was barely in popular use, but was later used in corporate marketing by Google, Facebook, and others. In 2007, Ask.com even put slogans such as "THE ALGORITHM CONSTANTLY FINDS JESUS" on advertising billboards (Sandvig, 2014). Bogost (2015) shares examples from that year of "the algorithm" being used in imprecise, systemic terms. Similar recent usage includes Australian Competition and Consumer Commission (2018: 9) and Chatterjee and Ravindran (2019).

A case where corporate and state opacity combine is the analysis performed by software on the likelihood of prisoner recidivism. In Wisconsin, a package called Compas is used as part of parole board review. The code is proprietary, and not available for review or rebuttal by a representative of the prisoner, a matter considered in the Supreme Court of Wisconsin case State v. Loomis (2016). As the company explains (Liptak, 2017):

> The key to our product is the algorithms, and they're proprietary," one of its executives said last year. "We've created them, and we don't release them because it's certainly a core piece of our business.

For the individual, the sense of urgent everyday need to describe a decision by a computational platform is neatly captured in a twitter discussion on ride sharing (Coleman, 2018):

> OH (from an awesome Lyft driver): "Today has been great. I've been blessed by the algorithm."
> Immediately had an eerie feeling that this could become an increasingly common way to describe a day.

Users may create a shorthand, inventing a singular decision-making agent with the name "algorithm."

Corporations and governments, even technically sophisticated ones, may twist definitions to serve the needs of power. All technical jargon is subject to these pressures and interpretations. Such a simple explanation does not account for empirical work that shows technologists also use "algorithm" in imprecise ways. Although bubble sort, for example, is clearly recognized as an algorithm, and that aligns with the structured programming definition, in other contexts, use even by software professionals is much wider. Devendorf and Goodman (2014) report the difficulties in locating "the algorithm" at a dating website, and how many data sources and components were ultimately involved in making matching recommendations. Seaver (2017) shares specific fieldwork done at a streaming music company, where it was routine to refer to "the algorithm" for music selection. When asked if they could show this object – say a particular code block corresponding to it – developers would respond in imprecise systemic terms. Even specialist developers with more academic computer science backgrounds asked to identify the algorithm in the codebase, would not, or would refer on to components owned by someone else. "It's very much black magic that goes on in there; even if you code a lot of it up, a lot of that stuff is lost on you," one commented (Seaver, 2017: 3).

This is well observed, and does match some of my own experience. I recall, for instance, a skilled algorithmic trading engine developer trying to explain their work to another developer outside the field, and saying "the algorithms aren't really algorithms ... they are much simpler than those from a textbook" (programmer, age 26, male, Singapore, 2008). This conveyed the complexity in such systems being around low latency reaction to market events, rather than code that looked like quicksort.

Seaver also notes that those with a deep technical understanding of software are often happy enough to use "algorithm" in this broad systemic way when it suits them. An additional example might be Evans (2018), partner at Andreesen Horowitz, discussing the "algorithmic feed" design choices made by Twitter and Facebook:

> All social apps grow until you need a newsfeed
> All newsfeeds grow until you need an algorithmic feed
> All algorithmic feeds grow until you get fed up of not seeing stuff/seeing the wrong stuff & leave for new apps with less overload
> All those new apps grow until...

The two solutions discussed by Evans are a strictly time sorted feed, and one where items of interest are filtered by a preference function. The preference function can have parameters of historical use, demographic profiling, and other statistical weightings we have to guess at. Describing one as "algorithmic" and one not is, of course, ridiculous (which Evans acknowledges elsewhere in the article). Either both are algorithmic or neither is, and singling out the time-sorted one as "unalgorithmic" is particularly perverse, given sort algorithms are one of the better characterized ways of rearranging information. He seems to be describing that the process is opaque. Our lives are being impacted by the black box calculation of a system we may benefit from, but cannot clearly explain.

Culture

> Computers are extremely flexible and powerful tools and many feel that their application is changing the face of the earth. I would venture the opinion that as long as we regard them primarily as tools, we might grossly underestimate their significance. Their influence as tools might turn out to be but a ripple on the surface of our culture, whereas I expect them to have a much more profound influence in their capacity of intellectual challenge! (Dijkstra, 1972: 6)

The observations that different definitions of algorithm are in use by technologists, and by others, call for explanation. Yeung (2018: 3–4) notes both usages. Dourish (2016: 2–3) warns against breakdowns of communication between scholarly communities when using two definitions. Mittelstadt et al. (2016: 2) recognizes a distinction, then largely dismisses it: "Any attempt to map an 'ethics of algorithms' must address this conflation between formal definitions and popular usage of 'algorithm' [...]. Our aim here is to map the ethics of algorithms, with 'algorithm' interpreted along public discourse lines." They go on to identify automated decisions as the nexus of these ethical problems, and traceability as one of six types of ethical concern (Mittelstadt et al., 2016: 5).

In 2013, Seaver proposed that "the algorithms we are interested in are not really the same as the ones in *Introduction to Algorithms [...]*. Outside of textbooks, "algorithms" are almost always "algorithmic systems"" (Seaver, 2019: 418, 419). This closely corresponds to what is here called the "algorithms as systems" definition, though bolder assertions that "there are no theoretical algorithms in actually existing software systems" do not reconcile with artifacts like Python list sort() described earlier. Later, after more fieldwork, Seaver (2017: 5) concludes that algorithm does not have a stable, well-understood textbook definition at all, but one so general and diffuse that "algorithms [...] are culture."

Seaver (2017: 2) also suggests that technologists insist on a narrow technical definition of "algorithm" to police a professional boundary and avoid responsibility for broader social impacts of their decisions:

> [T]erminological anxieties are first and foremost anxieties about the boundaries of disciplinary jurisdiction, and critical algorithm studies is, essentially, founded in a disciplinary transgression. The boundaries of expert communities are maintained by governing the circulation and proper usage of professional argot, demarcating those who have the right to speak from those who do not [. . .], and algorithms are no different.

Treating algorithms as culture is one solution to these problems. Certainly computing is not a field free of disciplinary jurisdiction. There are, however, reasons to be cautious about embracing this framework, which suggest slicing it into two definitions.

Ethnographic study is situated in a particular time and place. A streaming music startup (Seaver, 2017), or a dating website firm (Devendorf and Goodman, 2014), are both village-sized commercial communities focused on selling a single recommendation-focused product. They have a particularly strong motivation to invent a shorthand for decisions made by a simplifying, imagined, computational agent. They also have to take the position of the user in testing that singular recommendation product; they have to "listen to the algorithm" (Seaver, 2018). It is not clear how general this usage is, or how much longevity it has. The examples reviewed in this article, including those from Seaver, remain broadly compatible with the "algorithm as system" definition, and do not require the generality of "algorithms as culture."

Seaver relates the experience of anthropology in moving away from "culture" as a useful concept, then re-engaging with it, due to its use in communities anthropologists worked with (Seaver, 2017: 4). There are limits to this analogy. Computer science and software engineering have not had a multi-decade debate about the usefulness and existence of algorithms; they have had a multi-decade movement of encapsulating algorithms as reusable technical objects. Even beyond ongoing academic education and publication, and the use as components in code, they are still active objects of technical discussion. At time of writing, on the programming website Stack Overflow, there are 26,649 results for "python sort" (Figure 6) and 89,829 for "algorithm." Declaring it a dead concept only useful when repurposed for interdisciplinary communication is a little premature.

An alternative is seen in work on super-Turing computation, which considers problems of modern software that are computable, but cannot be solved algorithmically, such as "Driving A Car To Work" (Eberbach et al., 2004: 13–15). Eberbach introduces new terminology such as "Interactive problem" to contrast with "Algorithmic problem," and an idea of "interactive computation" which may include algorithmic components. "Mixed initiative" or "humans-in-the-loop" systems are another way to conceive of computation that is not strictly algorithmic (Allen et al., 1999; Sheridan, 2006: 1025–1052).

"Algorithms as systems" can be considered a different agential cut, between the user and a software-mediated sociotechnical system. In doing so, language shows a boundary, but not an exclusively disciplinary one. "Algorithm" is used in the systemic sense when not just programmers, but people in general, don't want, or don't have, access to the machinic detail required for a design decision. This cut is at a coarser granularity than the structured programming cut, and serves as a pragmatic everyday theory for users navigating software systems. This makes the implied apparatus defining the sub-components of "user" and "algorithmic system" at the scale of a community. In many of these examples, the cut also sketches the outline of an existing organization. Tools like computational complexity cannot be applied to the markings left by such an apparatus.

Both working definitions of algorithm are valid and interesting topics of scholarly study. Scholars and technologists may still want to take care in equating "algorithm" with a popular usage which places disempowered users outside an implicitly opaque platform. It also places scholars and users outside a design process of trade-offs driven by machinic and social detail. Interesting design insights can come from that systems point of view, and part of the contribution of humanities and social sciences is in illuminating alternative and disempowered perspectives. Yet in this case, the black box perspective is not just that of the disempowered user subject, but also that of empowered corporations and states. It is the same language, for example, used by the corporate leadership of Compas, when labeling decisions by a systemic governance platform using computers as simply "algorithmic." Likewise corporations and states may want to take care displacing agency onto a deliberately abstract mechanism when there is clearly much about its harnessing and use that is social and concrete.

Once the problem space has been linguistically framed this way, it constrains further analysis even by intelligent people who are being careful and sincere. It becomes a way of treating software platforms as a generic other, even if it was once intended to be a way of putting us all inside the box together. It does this by interrupting the design process of differentiation and concretization. The separation of elements into those easily analyzed for algorithmic complexity, those user-controlled, and so on, is of merit on engineering grounds alone – reasons of developer productivity, runtime performance, economic efficiency, and system safety, for instance. This goal of making systems easier to engineer and maintain is, however, largely complementary with making systems more transparent and controllable for goals more laden with other values. Definitions in matters of public concern are also likely to settle into static institutional and legislative forms. Having legal definitions push against the names and structure of well-designed software undermines both good engineering and good public policy.

The mislabeling of algorithms has a particular irony, because choosing good names is one of the craft skills of a great programmer, and one where aesthetic and sociolinguistic insights are crucial (Martin, 2009: 17–30). Code is overflowing with names, littered with them, hundreds or thousands of characters in a dense theatre of causation. Choosing great names, and using them to structure the evolving design of the system, could be a rich field of collaboration between technical and cultural concerns.

Attempts to synthesize the cultural and the technical aspects of software recall Simondon's *On The Mode of Existence of Technical Objects*. That book starts with a careful description of how evolving machines translate into distinct interlocking components and ends by discussing human relationships to technical objects (Simondon, 1980: 98):

> The technical object must be known in itself if the relationship between man and machine is to be steady and valid. Hence the need for a technical culture.

In a society where the technology of software is pervasive, a technical culture would be one where most people can talk with mutual intelligibility about software with software developers. A popular software technical culture would have an easy familiarity with fundamental technical terms, and an understanding that programmers, too, can be overwhelmed by software complexity.

Conclusion

Conceiving of algorithms separately from data structures is a scoping technique that makes thinking about software simpler for the limited capabilities of human minds. This technique cuts domain-specific details, including ones with ethical weight, into separate artifacts from abstracted algorithmic code focused on the movement of information. Structured programming and its successors are, amongst other things, social projects of legibility and control over computational systems. On those terms, structured programming was one of the great intellectual triumphs of the 20th century. It was so successful in establishing reusable libraries it removed one of its primary subjects – algorithms – from immediate view. It was so successful that even technologists can find themselves mangling clear technical terms to find ways to explain the operation of systems operating at a global scale, across millions of users, and atop millions of lines of code.

The separation of algorithms into reusable components has made direct work on algorithmic code less common. Programmers everywhere remain users of algorithms as components, and a sophisticated understanding of their performance and other formal properties remains a key software engineering skill. It also connects to techniques of scoping, decoupling and differentiation needed to build large software systems.

Corporations, states, journalists, technologists, and users of all kinds may also use the term "algorithm" less precisely to refer to phenomena around computational systems. This important observation can be explained by a secondary, popular definition of algorithm, when viewing an opaque computational system as a disempowered user. There is much about the complex, changing, difficult to observe properties of large software systems that can engender this perspective. The evidence reviewed here supports this theory as well as Seaver's interpretation that the usage is so diffuse that algorithms are culture. Given this, the precision of the secondary, algorithm as system, definition, and its characterization as a popular definition, is preferred.

This should not discourage cultural scholars and social scientists from engaging with the sociotechnical assemblages where algorithms are found. Cultural values may be found in code and other software artifacts, but they will mostly be adjacent to the abstract information-centric components called algorithms. Likewise, taking the perspective of a disempowered user outside an opaque system can be a productive critical stance. New names for these assemblages, and critiques of their power and interactions, can build a shared language and culture around their construction and use, and shape systems and societies to come.

Software development needs an inclusive extension of the structured programming project which explains large-scale systems, intertwined through social, software and physical structures. The precise use of technical terms, like algorithm, by everyone involved in design, is worthwhile because it supports that larger intellectual project. A shared design language is needed for a broad technical culture. Software development is too important to be left just to software developers.

Acknowledgements

The author thanks Bronwyn Sharman, Dominic Fox, Mohammed Salemy, and Reza Negarastani, who reviewed drafts of this work and provided general support.

The author also thanks the editors of *Big Data and Society*, and the anonymous peer reviewers, for providing further incisive comments which improved the article.

Declaration of conflict interests

The author(s) was employed as a software developer in electronic trading technology at Bank of America Merrill Lynch until September 2018, during which early drafts of this article were written. The article was written independently and there is minimal commentary on the business of BAML in this work.

Funding

The author(s) received no financial support for the research, authorship, and/or publication of this article.

Notes

1. A natural language gloss of bubble sort is as follows. Start with a row of items in no particular order. You wish to place them in order from lightest to heaviest. Take the rightmost item, and compare it to the item to its left. If it is heavier, leave it where it is, and move on to the next item to the left instead. If it is lighter, swap the two items, then repeat the comparison with the next item to the left. Repeat this, systematically working through the list as many times as there are items in the row, starting one item further left each time. This "bubbles" each item through to the correct place by successive swaps. A detailed and accessible account of bubble sort is the technical starting point for Introna (2016) when considering algorithms and governmentality. A visualization pairing an animated list with highlighted lines of code can be found at: https://visualgo.net/bn/sorting (Halim, 2019).
2. Collections of items stored in sorted order allow for quicker searching. A ready physical example is the collection of books in a public library. Each aisle is labeled with the range of items found there, according to some well-advertised classification scheme, such as alphabetic by author's name for fiction books. By checking the label, you exploit the sorted nature of the collection and avoid the time cost of walking down each aisle. The labels also take some extra physical space, a direct analogue to the additional memory usage of indexed data structures.

   Unlike a library shelf, in a tree data structure, the "slot" for each item holds both the item, and another shelf which can hold further items. The name refers to the way this structure "branches," like tree branches spreading out from a trunk.

   In Figure 2, lines are numbered for navigation. Key lines of code are:

   2: Defines the tree data structure.
   7–11: The search function inTree.
   9: If inTree finds the item, it returns True.
   10: If inTree is before the item it was looking for, it opens up the shelf and recursively checks each item/shelf within it.
   11: If inTree is past where the item should be, the item is not present in the collection, and the function returns False.
   13–21: Functions for creating test data.
   24: The inList function does not assume the list is sorted, but simply looks at every item in the list in turn.
   26–end: The output of running these tests at the console.

   A visualization of tree data structures, including searches, can be found at: https://visualgo.net/bn/bst (Halim, 2019).
3. The talk "Embracing algorithms" (Abrahams, 2018) is a good introduction to the way algorithms can be hidden by intermediate layers, organized around a fictional vignette on generational changes in software development.

ORCID iD

Adam Burke https://orcid.org/0000-0003-4407-2199

References

Abadi M, Barham P, Chen J, et al. (2016) Tensorflow: A system for large-scale machine learning. *OSDI* 16: 265–283.

Abrahams D (2018) Embracing algorithms. In: *2018 apple worldwide developer conference (WWDC)*, San Jose, USA, 4–8 June 2018. Available at: https://developer.apple.com/videos/play/wwdc2018/223/ (accessed 7 March 2019).

Allen J, Guinn CI and Horvitz E (1999) Mixed initiative interaction. *IEEE Intelligent Systems and their Applications* 14(5): 14–23.

Australian Competition and Consumer Commission (2018). *Digital Platforms Inquiry: Preliminary Report*. Australia: Australian Competition and Consumer Commission.

Bailey SE, Godbole SS, Knutson CD, et al. (2013) A decade of Conway's law: A literature review from 2003–2012. In: *2013 3rd international workshop on replication in empirical software engineering research*, October, pp.1–14. USA: IEEE.

Barad K (2007) *Meeting the Universe Halfway: Quantum Physics and the Entanglement of Matter and Meaning*. Durham: Duke University Press.

Blue V (2019) How sex censorship killed the internet we love (newspaper article). *Engadget*, 31 January. Available at: www.engadget.com/2019/01/31/sex-censorship-killed-internet-fosta-sesta/ (accessed 2 March 2019).

Bogost I (2015) The cathedral of computation. The Atlantic, January 15th, 2015. Available at http://www.theatlantic.com/technology/archive/2015/01/the-cathedral-of-computation/384300/ (accessed 1 March 2019).

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

Chatterjee S and Ravindran S (2019) There is need for a legal, organisational framework to regulate bias in algorithms. *The Indian Express*, 28 February. Available at: https://indianexpress.com/article/opinion/columns/rules-for-the-machine-algorithm-artificial-intelligence-5603795/ (accessed 1 March 2019).

Citron DK and Pasquale F (2014) The scored society: Due process for automated predictions. *Washington Law Review* 89: 1.

Cockburn A (2002) *Agile Software Development*. (Vol. 177). Boston: Addison-Wesley.

Coleman K (2018) OH (from an awesome Lyft driver): "Today has been great. I've been blessed by the algorithm." Immediately had an eerie feeling that this could become an increasingly common way to describe a day. *Twitter Post*, 16 March. Available at: https://twitter.com/kcoleman/status/974495158841499648 (accessed 24 October 2018).

Cormen TH, Leiserson C, Rivest R, et al. (2009) *Introduction to Algorithms*. third ed. Massachusetts London, England: MIT press.

Crawford K (2013) The hidden biases of big data. Harvard Business Review Blog. 1 April. Available at: http://blogs.hbr.org/2013/04/the-hidden-biases-in-big-data/ (accessed 7 March 2019).

Dalke A and Hettinger R (2018) Sorting HOW TO. Available at: https://docs.python.org/3/howto/sorting.html (accessed 7 March 2019).

Deleuze G and Guattari F (1987) *A Thousand Plateaus* (Trans. Massumi B). Minneapolis: University of Minnesota Press (Original work published 1980).

Devendorf L and Goodman E (2014) *The Algorithm Multiple, the Algorithm Material. Contours of Algorithmic Life*. California: UC Davis.

Diakopolous N (2013) Rage against the algorithms (newspaper article). *The Atlantic*, 3 October. Available at: www.theatlantic.com/technology/archive/2013/10/rage-against-the-algorithms/280255/ (accessed 2 March 2019).

Dijkstra E (1979) Programming considered as a human activity. In: Yourdon (ed) *Classics in Software Engineering*. New York: Yourdon Press, pp.1–9.

Dijkstra EW (1972) Notes on structured programming. In: O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (eds) Structured Programming. New York: Academic Press, pp.1–82.

Dourish P (2016) Algorithms and their others: Algorithmic culture in context. *Big Data & Society* 3(2): 1–11.

Evans B (2018). The death of the newsfeed. *Benedict Evans*, 2 April. Available at: https://www.ben-evans.com/benedictevans/2018/4/2/the-death-of-the-newsfeed (accessed 24 October 2018).

Eberbach E, Goldin D and Wegner P (2004) Turing's ideas and models of computation. In: *Alan Turing: Life and Legacy of a Great Thinker*. Berlin: Springer, Berlin, Heidelberg, pp.159–194.

Feathers M (2004) *Working Effectively With Legacy Code*. Upper Saddle River, NJ: Prentice Hall.

Gillespie T and Seaver N (2016) Critical algorithm studies: A reading list. *Social Media Collective*. Available at: https://socialmediacollective.org/reading-lists/critical-algorithm-studies/ (accessed 1 March 2019).

Glaze (n.d.) Finally, Python brings something interesting to the programming language. In: Zhongmang ██. Available at: www.chinesepython.org/english/english.html (accessed 24 October 2018).

Golovin D, Solnik B, Moitra S, et al. (2017) Google vizier: A service for black-box optimization. In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pp.1487–1495. New York:ACM.

Hasselström K and Åslund J (2001) The Shakespeare programming language. Available at: http://shakespearelang.sourceforge.net/report/shakespeare/shakespeare.html (accessed 24 October 2018).

Halim S (2019) VisuAlgo. Available at: https://visualgo.net/bn/sorting (accessed 22 February 2019).

Hopper GM and Mauchly JW (1953) Influence of programming techniques on the design of computers. *Proceedings of the IRE* 41(10): 1250–1254.

Introna LD (2016) Algorithms, governance, and governmentality: On governing academic writing. *Science, Technology, & Human Values* 41(1): 17–49.

Knuth DE (1976) Big omicron and big omega and big theta. *ACM Sigact News* 8(2): 18–24.

Knuth DE (1997) *The Art of Computer Programming: Fundamental Algorithms. Fundamental Algorithms*. Reading, MA: Addison-Wesley.

Kraemer F, Van Overveld K and Peterson M (2011) Is there an ethics of algorithms? *Ethics and Information Technology* 13(3): 251–260.

Kramer D (1996) Java API documentation 1.0.2. Sun Microsystems, Inc. Available at: http://web.mit.edu/java_v1.0.2/www/apibook/index.htm (accessed 24 October 2018).

Kreps J, Narkhede N and Rao J (2011) Kafka: A distributed messaging system for log processing. In: *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB 2011)*, Athens, Greece, Jun 12, 2011. Association of Computing Machinery, pp.1–7.

Liptak A (2017) Sent to prison by a software program's secret algorithms (newspaper article). *New York Times*, 1 May. Available at: https://www.nytimes.com/2017/05/01/us/politics/sent-to-prison-by-a-software-programs-secret-algorithms.html (accessed 24 October 2018).

Martin RC (2009) *Clean Code: A Handbook of Agile Software Craftsmanship*. London: Pearson Education.

Mittelstadt BD, Allo P, Taddeo M, et al. (2016) The ethics of algorithms: Mapping the debate. *Big Data & Society* 3(2): 1–17.

Noble SU (2018) *Algorithms of Oppression: How Search Engines Reinforce Racism*. New York: NYU Press.

Python Software Foundation (2019) Higher-order functions and operations on callable objects. Available at: https://docs.python.org/3/library/functools.html (accessed 5 March 2019).

Sandvig C (2014) Seeing the sort: The aesthetic and industrial defense of "the algorithm". *Journal of the New Media Caucus* 10(3) Available at: http://median.newmediacaucus.org/ art-infrastructures-information/seeing-the-sort-the-aesthetic-and-industrial-defense-of-the-algorithm/ (accessed 7 March 2019).

Seaver N (2017) Algorithms as culture: Some tactics for the ethnography of algorithmic systems. *Big Data & Society* 4(2): 1–12.

Seaver N (2018) What should an anthropology of algorithms do? *Cultural Anthropology* 33(3): 375–385.

Seaver N (2019) Knowing algorithms. In: Vertesi J and Ribes D (eds) *Digitalsts: A Field Guide for Science & Technology Studies*. Princeton: Princeton University Press.

Socolofsky TJ and Kale CJ (1991) TCP/IP Tutorial (RFC 1180). Report, RFC Editor. Available at: http://www.rfc-editor.org/rfc/rfc1.txt (accessed 7 March 2019).

Sheridan TB (2006) Supervisory control. In: Salvendy G (ed) *Handbook of Human Factors and Ergonomics*, 3rd ed. Hoboken, NJ: Wiley, pp.1025–1052.

Simondon G (1980) In: Hart J (ed.) *On the Mode of Existence of Technical Objects* (Trans. Mellamphy N). Unpublished manuscript, University of Western Ontario (translation of Du Mode d'existence des Objects Techniques. Paris: Aubier Montaigne, 1958).

Simondon G (2016) *On the Mode of Existence of Technical Objects* (Trans. Malaspina C and Rogove J). USA: Univocal Publishing (translation of Du Mode d'existence des Objects Techniques. Paris: Aubier Montaigne, 1958).

State v. Loomis, 881 N.W.2d 749 (Wis. 2016), No. 2015AP157-CR (13 July 2016).

Thomas R (2000) Values, types, and other goodies. Available at: https://www.haskell.org/tutorial/goodies.html (accessed 24 October 2018).

US Executive Office of the President (2016) *Big Data: A Report on Algorithmic Systems, Opportunity, and Civil Rights*. USA: US Executive Office of the President.

Winner L (2010) The Whale and the Reactor: A Search for Limits in an Age of High Technology. Chicago: University of Chicago Press.

Wirth N (1976) *Algorithms + Data Structures = Programs Prentice-Hall Series in Automatic Computation*. Upper Saddle River, NJ: Prentice Hall.

Yeung K (2018) Algorithmic regulation: A critical interrogation. *Regulation & Governance* 12(4): 505–523.

Yourdon EN (1979) *Classics in Software Engineering*. New York: Yourdon Press.

Žliobaitė I (2017) Measuring discrimination in algorithmic decision making. *Data Mining and Knowledge Discovery*, 31(4): 1060–1089.