BY
Kim B. Bruce,
Robert L. Scot Drysdale,
Charles Kelemen,
and
Allen Tucker

Why Math?

The mathematical thinking, as well as the mathematics, in a computer science education prepares students for all stages of system development, from design to the correctness of the final implementation.

Math requirements! These words are enough to send chills down the spines of a good share of new computer science majors every year. Evidence that even some practitioners and educators question the value of mathematics for computer science is discussed in [2]. They might claim mathematics is used simply as a filter—weeding out students too weak or unprepared to survive—or just to pare down the hordes of potential computer science majors to a more manageable size. Others might argue it is just another sign that faculty in their ivory towers have no clue what practitioners really do or need. Each of these views surely has its adherents, but we argue here that learning the right kind of mathematics is essential to the understanding and practice of computer science.

What is the right kind of mathematics for preparing students for real-world responsibilities? In computer science, discrete mathematics is the core need. For applications of computer science, the appropriate mathematics is whatever is needed to model the application discipline. Software (and hardware) solutions to most problems, including those in banking, e-commerce, and airline reservations, involve constructing a (mathematical) model of the real (physical) domain and implementing it. Mathematics can be helpful in all stages of development, including design, specification, coding, and verifying the security and correctness of the final implementation. In many cases, specific topics in mathematics are not as important as having a high level of mathematical sophistication. Just as athletes might cross-train by running and lifting weights, computer science

Illustration by Jean-François Podevin

students improve their ability to abstract away from details and be more creative in their approaches to problems through exposure to challenging math and mathematically oriented computer science courses.

Discrete mathematics includes the following six topics, or discrete structures, the core in the ACM/IEEE computer science report *Computing Curricula 2001: Computer Science* [5]: Functions, relations, and sets; Basic logic; Proof techniques (including mathematical induction and proof by contradiction); Basics of counting; Graphs and trees; and Discrete probability.

We begin our exploration of the need for discrete mathematics in computer science with a simple problem whose solution involves its use. Vectors are supported in standard libraries of C++ and Java. From the programmer's point of view a vector looks like an extensible array. That is, while a vector is created with a given initial size, if something is added at an index beyond its extent, the vector automatically grows to be large enough to hold a value at that index.

A vector can be implemented in many ways (such as a linked list), but the most common implementation uses an array to hold the values. In such an implementation, if an element is inserted beyond its extent, the data structure creates a new array large enough to include the index, copies the elements from the old array to the new array, then adds the new element at the proper index. This vector implementation is straightforward, but how much should the array be extended each time it runs out of space?

Keeping things simple, suppose the array is being filled in increasing order, so each time it runs out of space, it needs to be extended by only one cell. There are two strategies for increasing the size of the array: always increase its size by the same fixed amount, F, and always increase its size by a fixed percentage, P%. A simple analysis using discrete mathematics (really just arithmetic and geometric series) shows that in a situation in which there are many additions, the first strategy results in a situation where the average cost of each addition is $O(n)$, where n is the number of additions (that is, the total of n additions costs some constant multiplied by n^2); the average cost for each addition with the second strategy results in a constant (that is, the total of n additions costs a constant multiplied by n).[1]

This simple but important example analyzes two different implementations of a common data structure. But we wouldn't know how to compare their quite significant differences in cost without being able to perform a mathematical analysis of the algorithms involved in the implementations.

Here, we aim to sketch out some other places where mathematics or the kind of thinking fostered by the study of mathematics is valuable in computing. Some of the applications involve computations, but more of them rely on the notion of formal specification and mathematical reasoning.

Determining Efficient Algorithms

Mathematics is central to designing and analyzing algorithms. We could discuss how to solve recurrence relationships, average-case analyses, and many other things everyone would agree are highly mathematical. But the argument could be made that only a handful of specialists need to do such things; everybody else can just look up the algorithms others have developed.

Still, evaluating and selecting algorithms is not simple. Consider a simple consulting job: Suppose the independent cab and limo operators in Salt Lake City had decided to contract with a consultant to write a program to help each of them schedule all the customers who wanted to ride with them during the 2002 Winter Olympics. Their first request might have been for the consultant to write a program into which customers could enter requests of the form: "I want a cab and driver from such and such a start date and time to such and such a finish date and time." As drivers are paid a flat rate per ride, the program would provide a driver the largest possible subset of requests that did not overlap in time.

Later, the drivers might have realized that instead of charging a fixed rate they could have customers bid for how much they were willing to pay for the requested period; the opening ceremonies and figure skating were, for example, more popular than the biathlon. The second version of the program scheduled the set of non-overlapping requests to maximize the amount of money the driver using the program would earn.

However, some customers might have wanted the same driver the whole time they were at the games. To accommodate them, a third version of the program could have been developed to take a set of time-period requests, along with a single bid for the whole set. A driver would have had to agree to drive for all requested intervals or refuse the request. The program would pick the sets of requests that maximized the amount of money the driver would receive without overlapping in time.

At first glance, it seems like the main difference between the three program versions would have been in the user interface. But that was not the case. The version with the flat-rate pricing can be solved by a simple greedy algorithm in $O(n \log n)$ time: sort the requests by finish time and at each step schedule the first request that does not overlap the last job scheduled. However,

[1]The constants depend on the values of F and P. A very simple analysis is possible when the algorithm starts with an empty array and $F = 1$ (add one new element when the array runs out of space) and $P = 100\%$ (double the size of the array when it runs out of space).

this greedy algorithm would not solve the variable-rate version, but a particular $O(n \log n)$ dynamic programming algorithm would solve it.

The third problem—handling sets of requests—is NP-hard. For practical purposes, this means the consultant wouldn't have found a substantially better solution than trying all the 2^n possible subsets of requests and so should have tried to find a good but not optimal solution rather than promise to find the best solution.

How would the consultant have known that a simple greedy algorithm solves the first problem (but not the second) and that a dynamic programming algorithm solves the second problem? The consultant would have had to prove it. How would the consultant know that the third problem is NP-hard? The consultant would have had to prove it by reducing a known NP-hard problem—Set Packing—to this problem. There would have been no way to do a professional job on this consulting assignment without doing these proofs. (See [3] for more on algorithms.)

We could offer many more examples where similar problems must be solved or where some are easy and others intractable. Mathematical proofs are the only way to distinguish among the alternatives.

Formal Specifications in the Real World

The term "formal methods" in hardware and software design means that precise mathematical specifications are used to define a product and that the product's implementation (code) is verified using mathematical proof techniques. The extent to which formal methods are used to design a particular product depends on many factors, including the cost of development, efficiency of the resulting code, skills of the developers, and safety-critical nature of the application.

There has been a great deal of practitioner interest of late in formal specification and verification of hardware, as well as of software. The potential cost of a mistake in the design of, say, a chip can be enormous, thus it can be financially beneficial to commit the resources to verifying a hardware design. Also, when designing a protocol that could be widely used, it is crucial to verify it has the required performance and security properties.

Most software engineers tend to think of these formal proofs of correctness when they hear the words formal methods, but we consider formal methods more broadly to encompass a variety of situations where there are benefits to using a specification and mathematical tools by computer scientists.

XML, recursion, and mathematical induction. The syntax of a programming language is formally specified via context-free grammar or syntax diagrams. This specification makes it clear to both compiler writers and programmers what is legal syntax.

A promising development with the same flavor as the formal specification of programming language syntax is the introduction of XML as a structured way to transmit information between programs and systems [1]. Data is presented using tags similar to those in HTML, but the tags indicate the semantic structure of the data, rather than its layout in a browser. Data type definitions (DTDs) provide a formal specification of the constraints on the structure of data similar to the way a static type system indicates constraints on legal programs in a particular programming language.

XML data can be parsed like programming languages, resulting in structures like parse trees. The data itself can be verified against DTDs using techniques similar to the ones used in type checkers on programming languages. However, rather than being restricted to the inflexible structure of a fixed programming language, groups sharing data with similar meanings can agree on different sets of tags and DTDs for representing different kinds of data.

If sender and receiver agree on the DTD for data, the sender can generate XML-formatted data, while the receiver can parse, verify, and transform it into a format easier for the receiver to use. All this processing can use technology originally developed for compiling programming languages. The technology has been one of the great triumphs of theoretical computer science, providing provable algorithmic connections between the formal description of languages and programs for processing the languages.

However, even if programmers ignore this technology and simply process the data directly using the equivalent of recursive descent compilers, the mathematical understanding of XML as formally specified data provides tools for working with XML. The DTD provides a specification of the structure of data similar to that of a regular expression. Simple algorithms based on finite automata derived directly from such specifications can verify that incoming data satisfy the specifications, while other data-directed algorithms parse and transform the data into other formats.

XML documents can be understood in their parsed form as trees. Recursive algorithms for working with trees are significantly easier to understand than equivalent iterative algorithms using a stack. (Most programmers find it a real challenge to write an iterative algorithm to do an in-order traversal of a tree.) While many programmers have tried to avoid recursive algorithms—some because they didn't understand them, others because they felt they were too inefficient—processing recursively specified or tree-structured data is much easier with recursion.

How might programmers best understand recursion and ensure their recursive programs satisfy the given specification? The answer is mathematical induction—one of many reasons that proof by induction is such an important topic in courses on discrete mathematics. Programmers with a good understanding of mathematical induction find it easier to write and, more important, provide convincing arguments for the correctness of recursive algorithms.

We were careful to say "provide convincing arguments" rather than "prove" in the preceding paragraph. While there are circumstances where a careful formal proof of correctness is called for, most of the time it is sufficient to provide an informal argument for the correctness of an algorithm.

If programmers are able to write the specifications of the parts of an algorithm, it is generally relatively easy for them to also provide an informal argument of correctness by asking, and answering: Does the base case satisfy the specification? and Do complex cases eventually get down to a base case? If the programmer presumes that all embedded recursive calls do the right thing, does this case satisfy the specification? Moreover, rather than just using such a process to verify an existing program, the process can also be used to develop and verify a program at the same time.

Secure and safety-critical systems. Recent virus and other security attacks highlight the importance of and often critical need for secure and safety-critical systems. While most computer scientists do not write secure or safety-critical systems, they must still understand the existing and potential threats to their systems. Interesting work has been done on ways to verify that downloaded software from untrusted sources will not behave in ways that put a system at risk. Downloaded applets in Java (at least with the proper security policy included in the browser) are guaranteed to run in a "sandbox," which excludes reading from or writing to the local file system.

Other interesting research has focused on "proof-carrying code" [4]; programmers provide a machine-assisted proof that the program satisfies a given security policy (such as it won't write to memory outside a fixed set of locations or won't write to files). This proof is typically much easier to develop than a proof of correctness of the program. The proof may be downloaded with the code and checked (automatically) against the downloaded code to ensure it is correct and the downloaded code is secure.

While developing and sending a proof might be deemed too expensive for code intended to run only one time (for which restricting execution to a sandbox may be sufficient), it can provide great assurance against accidentally downloading viruses or other damaging code as part of major programs that will be used repeatedly on a system. Other techniques are also being developed, including compiling to assembly language with proof annotations, using mathematical proof techniques for the same purpose.

Conclusion

These arguments and examples give a sense of why mathematics and mathematical thinking are important in computer science. We could have cited many more, including the remarkable success of relational databases and model checkers for verifying hardware. The examples we selected are interesting in their own right and different enough from the ones usually cited to suggest that the tools and reasoning taught in mathematics courses, especially those covering discrete mathematics, are of great value later in practice.

A computer science education is not intended to teach what students need to know for their first job. Nor is it to teach what they will need to know for all the jobs they will ever have. On-the-job learning, reading, and short- and semester-long courses (whether online or in person) provide much of what is needed over the course of one's career.

One of the most important goals for a university education is to provide the foundations for further learning. We have heard it described this way: A traditional university education provides just-in-case learning rather than the just-in-time learning provided by on-the-job training. We know that mathematical thinking will be of use; we just can't always predict exactly when or what form it will take. C

References

1. Bosak, J. and Bray, T. XML and the second-generation Web. *Sci. Am.* (May 1999).
2. Bruce, K., Kelemen, C., and Tucker, A. Our curriculum has become math-phobic! *SIGCSE Bulletin 33 (2001)*, 243–247.
3. Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction to Algorithms, 2nd Ed.* MIT Press/McGraw-Hill, New York, 2001.
4. Necula, G. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1997), 106–119.
5. The Joint Task Force on Computing Curricula. Computing Curricula 2001. *J. Educat. Res. Comput. 1,* 3 (2001).

Kim B. Bruce (kim@cs.williams.edu) is the Frederick Latimer Wells Professor of Computer Science in the Department of Computer Science at Williams College, Williamstown, MA
Robert L. Scot Drysdale (scot@cs.dartmouth.edu) is a professor in and chair of the Department of Computer Science at Dartmouth College, Hanover, NH.
Charles Kelemen (cfk@cs.swarthmore.edu) is the Edward Hicks Magill Professor of Computer Science and chair of the Computer Science Department at Swarthmore College, Swarthmore, PA.
Allen Tucker (allen@bowdoin.edu) is the Anne T. and Robert M. Bass Professor and chair of the Department of Computer Science at Bowdoin College, Brunswick, ME.

Bruce's research was supported in part by National Science Foundation grant CCR-9988210.