# Why Math?[*]

Kim B. Bruce[†]
Williams College

Robert L. Scot Drysdale
Dartmouth College

Charles Kelemen
Swarthmore College

Allen Tucker
Bowdoin College

June 28, 2003

Math requirements! Those words are enough to send chills down the spines of a good share of new Computer Science majors every year. Evidence that at least some practitioners and educators question the value of mathematics for computer science is discussed in "Our Curriculum Has Become Mathphobic" [BKT01].

Some practitioners and educators might claim that mathematics is used simply as a filter – weeding out those students too weak or unprepared to survive – or even 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? For the central topics 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 domain.

Software (and hardware) solutions to most problems (such as banking, on-line 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 verification of the security and correctness of the final implementation. In many cases, particular topics in mathematics are not as important as having a high level of mathematical sophistication. Just as athletes cross-train by running and lifting weights, computer science students improve their ability to abstract away from details and be more creative in their approaches to problems through exposure to challenging mathematics and mathematically-oriented computer science courses.

Discrete mathematics includes the following six topics considered core in the ACM / IEEE CS report, Computing Curricula 2001: Computer Science [Cur01]:

**DS1.** Functions, relations, and sets.

**DS2.** Basic logic.

[*]Bruce's research was partially supported by NSF grant CCR-9988210.

[†]Corresponding author: Kim B. Bruce, Department of Computer Science, Williams College, Williamstown, MA 01267. kim@cs.williams.edu,(413) 597-2273, FAX: (413) 597-4250.

**DS3.** Proof techniques (including mathematical induction and proof by contradiction).

**DS4.** Basics of counting.

**DS5.** Graphs and trees.

**DS6.** Discrete probability.

Let's start our exploration of the need for discrete mathematics with a simple problem. Vectors are supported in standard libraries of C++ and Java. From a programmer's point of view, a vector looks very much 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 – for example as a linked list, but the most common implementation uses an array to hold the values. With this implementation, if an element is inserted beyond its extent, the data structure creates a new array that is large enough to include that index, copies the elements from the old array to the new array, and then adds the new element at the proper index. This vector implementation is pretty straightforward, but how much should the array be extended each time it runs out of space?

To keep things simple, suppose the array is being filled in increasing order, so each time it runs out of space, it only actually needs to be extended by 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 average cost for each addition with the first strategy is $O(n)$, where $n$ is the number of additions (that is, the total of $n$ additions costs a constant multiplied by $n^2$); the average cost for each addition with the second strategy is a constant (or, in other words, the total of $n$ additions costs a constant multiplied by $n$).[1]

This is a simple, yet very important, example analyzing two different implementations of a very common data structure, the vector. However, we wouldn't know how to compare the quite significant differences in costs 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.

## 1 Determining efficient algorithms

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

Still, evaluating and selecting algorithms is not that simple. Consider a simple consulting job: suppose that the independent cab and limo operators in Salt Lake City had decided to contract

[1]The constants involved 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 1 new element when the array runs out of space) and $P = 100\%$ (double the size of the array when it run out of space).

with a consultant to write a program to help each of them schedule the customers who wanted to hire them during the recent Winter Olympics. Their first request of the consultant might have been 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 should provide a driver using the program with the largest possible subset of the requests that did not overlap in time.

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

However, some customers might want the same driver for the whole time that they were at the games. To accommodate them, a third version of the program could be developed that takes a set of time period requests from each customer, along with a bid for the whole set. A driver would have 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 that 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 in fact not the case. The version with 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, this greedy algorithm would not solve the variable rate version, though there is a nice $O(n \log n)$ dynamic programming algorithm that would solve it.

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

How can the consultant know 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 must prove it. How does the consultant know that the third problem is NP-hard? The consultant proves it by reducing a known NP-hard problem, Set Packing, to it. There is no way to do a professional job on this consulting assignment without doing these proofs. (See [CLRS01], for example, for more on algorithms.)

We can offer many more examples where similar sounding problems must be solved using different techniques, or where one is easy and the other is intractable. Mathematical proofs are the only way to distinguish among the alternatives.

## 2 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 interest in industry of late in formal specification and verification of hardware, as well as software. The potential cost of a mistake in the design of a chip can be enormous, thus it can be financially very beneficial to commit the resources to verify a hardware design. Similarly, when designing a protocol that may 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 word formal methods, but that authors also consider formal methods more broadly to encompass a variety of situations where there are benefits to formal specification and the use of mathematical tools by computer scientists.

## 2.1 XML, recursion, and mathematical induction

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

A promising recent development with the same flavor as the formal specification of programming language syntax has been the introduction of XML as a structured way of transmitting information between programs and systems [BB99]. Data is presented using tags similar to those used in HTML, but the tags indicate the semantic structure of the data, rather than its layout in a browser. Data type definitions (DTD's) 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 programming language.

XML data can be parsed like programming languages, resulting in structures like parse trees. The data itself can be verified against DTD's using techniques similar to those 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 DTD's for representing different kinds of data.

If the sender and receiver agree on the DTD for data, then the sender can generate XML-formatted data, while the receiver can parse, verify, and then transform it into a format that is easier for the receiver to use. All of this processing can use technology originally developed for compiling programming languages. This 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. (It is a real challenge for most programmers to design an iterative algorithm to do an inorder traversal of a tree!) While many programmers have attempted to avoid recursive algorithms in the past (some because they didn't understand it, and others because they believed it was too inefficient), processing recursively-specified or tree-structured data is much, much easier with recursion.

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

We were careful to say "provide convincing arguments" rather than "prove" in the previous 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 can write down the specifications of the parts of an algorithm, it is generally relatively easy for them to provide an informal argument of correctness by asking and answering the following questions: Does the base case satisfy the specification? 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 be used to develop and verify a program at the same time.

## 2.2 Secure and safety-critical systems

Events over the last several years have highlighted the importance of and often critical need for secure and safety-critical systems. Problems have included inadvertently downloading viruses and other malicious programs designed by hackers to access or destroy private data. While most programmers do not write secure or safety-critical systems, they do need to understand the existing and potential threats to their computing systems. Interesting work has been done recently on ways of verifying that downloaded software from untrusted sources will not behave in ways that place a system at risk. Downloaded applets in Java (at least with the proper security policy set in the browser) are guaranteed to run in a "sandbox" that excludes reading from or writing to the local file system.

Even more recently there has been interesting research on "proof-carrying code" [Nec97]; 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 can be checked automatically against the downloaded code to make sure it is correct and the downloaded code is secure.

While developing and sending a proof may be deemed too expensive for code intended to run only once (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, that use mathematical proof techniques for the same purpose.

It has been apparent to all for some time now that security is an important issue when computers are attached to the internet. Solutions to security problems are very likely to involve provably secure protocols for guaranteeing certain kinds of safety.

## 3 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 of model checkers for verifying hardware. The examples we selected are interesting in their own right, and different enough from the ones regularly cited to suggest that the tools and reasoning taught in mathematics courses, especially those covering discrete mathematics, are of great value in practice.

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

Instead, 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: traditional university education provides "just-in-case" learning rather than the "just-in-time" learning provided by on-the-job training. Mathematical thinking will be of use – we just can't always predict exactly when or what form it will take.

## References

[BB99] Jon Bosak and Tim Bray. XML and the second-generation web. *Scientific American*, (5):243–247, May 1999.

[BKT01] Kim B. Bruce, Charles Kelemen, and Allen Tucker. Our curriculum has become math-phobic! In *SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, volume 33, pages 243–247, New York, NY, 2001. ACM Press.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press/McGraw-Hill, New York, NY, second edition, 2001.

[Cur01] Computing curricula 2001: Computer science. *The Journal on Educational Resources in Computing*, 1(3es), Article No. 1, fall 2001.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, 1997. ACM Press.