Why Math?*

Kim B. Bruce†
Williams College

Robert L. Scot Drysdale
Dartmouth College

Charles Kelemen
Swarthmore College

Allen Tucker
Bowdoin College

April 29, 2002

Math requirements! Those words are enough to send chills down the spines of a good share of new Computer Science majors every year. Bring the same topic up with practitioners and a good share will rant about how much time and effort they wasted on college mathematics that they have never used. Why is it that we force this material on unwilling students, especially when so many practitioners claim it is unnecessary?

Some might claim that mathematics is simply used as a filter – to weed out those too weak to survive – or even just to cut down the hordes of students to a more manageable size. Others might argue that it is just another sign that faculty in their ivory towers have no clue what practitioners really do or what they need. While surely there are subscribers to each of these views, we argue here that the right kind of mathematics is essential to the understanding and practice of computer science.

What is the right kind of mathematics? For computer science qua computer science, the core need is discrete mathematics. For applications of computer science, the appropriate mathematics is whatever is needed to model the application discipline.

Software solutions to most problems (e.g. banking, on-line commerce, airline reservations, etc.) consist of constructing a (mathematical) model of the real (physical) domain and implementing it in software. Mathematics can be helpful in all stages of software development: design, specification, coding, verifying security and correctness of the final implementation. In many cases, particular topics in mathematics are not as important as a high level of mathematical sophistication. Just as athletes cross-train by running and lifting weights, exposure to challenging mathematics courses can help computer science students in their ability to abstract away from details and be more creative in their approaches to problems.

What is discrete mathematics? Here is the list of topics in discrete mathematics considered core in Curriculum 2001 [oCC02]:

DS1. Functions, relations, and sets.

DS2. Basic logic.

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

*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.

DS4. Basics of counting.

DS5. Graphs and trees.

DS6. Discrete probability.

As a warm up for our presentation of the use of discrete mathematics, let's start with a very simple problem. Vectors are supported in standard libraries of C++ and Java. From the outside a vector looks very much like an extendable 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 may 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. That's pretty straightforward, but how much should the array be extended each time it runs out of space?

For simplicity, let's suppose that 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. One is always to increase it in size by the same fixed amount, F. The other is to always increase it in 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 (or, in other words, the total of n additions costs some constant times n^2), while the second strategy results in a constant cost, on average, for each addition (or, in other words, the total of n additions costs a constant times n).[1]

This is a simple, yet very important example using a very common data structure. Yet 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 rest of this article, we 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 even more rely on the notion of formal specification and mathematical reasoning.

1 Determining efficient algorithms

In this section we discuss some deceptively simple-seeming problems that require mathematical analysis in order to choose and evaluate efficient algorithms. The example of vectors above is in this same category of problems.

1.1 Scheduling cabs and limos for the Olympics

Mathematics is central to designing and analyzing algorithms. We could discuss solving recurrence relationships, doing average-case analyses, and many other things that everyone agrees are highly

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

mathematical. However, 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.

However, it is not that simple. Consider a simple consulting job: suppose that the independent cab and limo operators in Salt Lake City wanted a program to help them schedule all the customers who want to hire them during the recent Winter Olympics. Their first request is that they want a program into which a driver can put n requests of the form "I want a cab and driver from this start date and time to this finish date and time." The program should select the largest possible subset of the requests that do not overlap in time.

They then realized that instead of charging a fixed rate they could let customers bid for how much they are willing to pay for the requested period. (Opening ceremonies and figure skating are more popular than the biathlon.) The second version of the program is to schedule the set of non-overlapping requests that maximizes the amount of money that the driver will receive.

However, some customers want to have the same driver for the whole time that they are in Salt Lake. To accommodate this, a third version of the program is desired. This version takes a set of time period requests, along with a single bid for the whole set. The driver must agree to drive for all of the requested intervals or refuse the request. The program should pick the sets of requests that maximizes the amount of money that the driver will receive without overlapping in time.

At first glance, it seems like the main difference between the three cases will be in the user interface. But that is not the case. The first problem 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.)

This greedy algorithm will not solve the second problem – but there is a nice $O(n \log n)$ dynamic programming algorithm that will solve it.

The third problem is NP-hard. For practical purposes, this means that we won't find a substantially better solution than trying all the 2^n possible subsets of requests, so we had better try to find a good but not optimal solution rather than promising to find the best solution.

How do we know that a simple greedy algorithm solves the first problem and that a dynamic programming algorithm solves the second problem? We prove it. How do we know that the third problem is NP-hard? We prove it by reducing a known NP-hard problem, Set Cover, to it. We know of no way to do a professional job on this consulting assignment without doing these proofs. (See [CLRS01], for example, for further information on algorithms.)

We can come up with dozens of examples where problems that seem very similar must be solved using different techniques, or one is easy and the other is intractable. Mathematical proofs are the only way to distinguish among the alternatives.

1.2 Akamai

Theory has applications with real consequences in the real world. One example is Akamai, a highly successful startup company that has survived the ".com" crash. Companies hire Akamai to provide faster and more reliable access to their web servers. The company was founded by Tom Leighton, a Professor of Applied Mathematics at M.I.T., and Danny Lewin, a Ph.D. student in the Algorithms group at M.I.T.'s Laboratory for Computer Science. Their patented algorithms for providing faster and more reliable web service formed the basis of the company and has resulted in their ubiquitous presence on the web.

2 Formal specifications in the real world

The term "formal methods" when used 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 the development, the efficiency of the resulting code, the capabilities and skills of the developers, and the safety-critical nature of the application.

There has been a great deal of interest of late in formal specification and verification of hardware, as well as software. The cost of a mistake in the design of a chip can be enormous, so that it can be financially very beneficial to expend the resources to verify a hardware design. Similarly, when designing a protocol that may be widely used, it is crucial to verify that it has the required properties in areas such as performance and security.

Most people tend to think of these formal proofs of correctness when they hear the word formal methods, but we are also thinking more broadly to encompass a variety of situations where there are benefits to formal specification and the use of mathematical tools by computer scientists in more general circumstances.

2.1 XML, recursion, and mathematical induction

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

An interesting recent development that has 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 similarly to the way programming languages are parsed, resulting in structures similar to 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 it, verify it, and then transform it into a format that is easier for the receiver to use.

All of this processing can use technology that has been developed for compiling programming languages. This technology was 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 we ignore this technology and simply process the data directly using the equivalent of recursive descent compilers, we immediately enter an area where 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 those 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. As anyone who has worked with trees knows, recursive algorithms for working with trees are significantly easier to understand than equivalent iterative algorithms that must use a stack. (If you've never tried to write an iterative algorithm to do an inorder traversal of a tree, you've missed a true challenge!) While many programmers 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 we best understand recursion and ensure that our recursive programs work properly? The answer, of course, is mathematical induction. This is one of many reasons that proof by induction is one of the most important topics in a discrete mathematics course. A programmer with a good understanding of mathematical induction finds it much easier to write and, even more importantly, provide convincing arguments for the correctness of recursive algorithms.

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

If a programmer can specify exactly what an algorithm does in terms of pre- and post-conditions, it is generally relatively easy to provide an informal argument of correctness: Does the base case satisfy the specification? Do complex cases eventually get down to a base case? If we presume that all embedded recursive calls do the right thing, does this case satisfy the specification? Moreover, rather than just using a process like this to verify an existing program, this 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 secure and safety-critical systems. Problems have included inadvertently downloading viruses and other malicious programs designed by hackers to gain access or destroy private data.

While most computer scientists will 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.

More recently there has been interesting research on "proof-carrying code" [Nec97]. In this case, the programmer provides a machine-assisted proof that the program will satisfy a given security policy (e.g., won't write to memory outside of fixed set of locations, won't write to files, etc.). This proof is typically much easier than a proof of correctness of the program. The proof may then be downloaded with the code, and the proof may be checked (automatically, of course) against the downloaded code to make sure that the proof is correct and that the downloaded code is secure.

While this may be deemed too expensive for code 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 (e.g., compiling to assembly language with proof annotations) that use mathematical proof techniques for the same purpose.

It has been apparent for some time now that security is a very 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

The above arguments and examples give a sense of why mathematics and mathematical thinking are important in computer science. There are many, many more examples that could have been cited if there were more space. We believe that these examples are interesting and different enough from the usual ones to suggest that the tools and reasoning taught in mathematics courses, especially discrete mathematics courses, are of great value in practice.

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

Instead, one of the most important goals for a college or university education is to provide the foundations for further learning. One way we have heard this put is that 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 don't know exactly when or what form it will take.

References

[BB99] Jon Bosak and Tim Bray. XML and the second-generation web. *Scientific American*, May 1999.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press/McGraw-Hill, second edition, 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. ACM Press, 1997.

[oCC02] The Joint Task Force on Computing Curricula. *Computing Curricula 2001: Computer Science*. IEEE Computer Society, 2002.