Has Our Curriculum Become Math-Phobic? (an American Perspective)

Charles Kelemen (cfk@cs.swarthmore.edu), **Allen Tucker** (allen@bowdoin.edu), **Peter Henderson** (phenders@butler.edu), **Kim Bruce** (kim@cs.williams.edu), **Owen Astrachan** (ola@cs.duke.edu)

Abstract

We are concerned about a view in undergraduate computer science education, especially in the early courses, that it's okay to be math-phobic and still prepare oneself to become a computer scientist. Our view is the contrary: that any serious study of computer science requires students to achieve mathematical maturity (especially in discrete mathematics) early in their undergraduate studies, thus becoming well-prepared to integrate mathematical ideas, notations, and methodologies throughout their study of computer science. A major curricular implication of this theme is that the prerequisite expectations and conceptual level of the first discrete mathematics course should be the same as it is for the first calculus course – secondary school pre-calculus and trigonometry. Ultimately, calculus, linear algebra, and statistics are also essential for computer science majors, but none should occur earlier than discrete mathematics. This paper explains our concerns and outlines our response as a series of examples and recommendations for future action.

1 The spread of math phobia

Much anecdotal evidence suggests that computer science undergraduate majors are largely averse to the use of mathematical notations, principles, and methods in their day-to-day computer science coursework. The extent to which this view is encouraged by computer science faculty may also be surprisingly high [Henderson 99]. Although some faculty may believe that computer science *does not* require a heavy investment in mathematics or mathematical ideas, theirs is probably not a prevailing belief among a majority of computer science educators (e.g., see [8]).

This leaves the uncomfortable feeling that, as computer science educators, we are not teaching what we profess to be at the heart of our discipline. Starting with the first course, the ubiquitous CS1 course, the great majority of instructors teach this course as "the programming course," most recently migrating toward object-oriented, interactive programming. While there are some exceptions, this course usually develops in students an understanding that computer science is about hacking code and solving problems by long hours of trial and error in the computer lab. The notion that computer science (including programming) has mathematical, experimental (scientific), and design principles and notations and themes running through it is usually not evident in these courses.

Second, while most CS major programs require a collection of mathematics courses, starting with calculus and then discrete math, they seldom find ways to convincingly integrate these mathematical topics within their own core courses. Oddly, calculus usually appears as a prerequisite for discrete mathematics, although its topics do not appear until much later in the CS curriculum. The argument that calculus should precede discrete mathematics is only justified on grounds that it provides the "mathematical maturity" needed to begin a discrete math course. Thus, the path of prerequisites to the data structures and algorithms courses, where discrete mathematics topics ought to be heavily used, is unnecessarily long.

Third, students who take the data structures and algorithms courses encounter a superficial level of rigor and integration for such mathematical concepts as induction, proof, and logic. In many cases, these concepts are not even included. At best, these mathematical topics play a minor role in comparison to the heavy levels of programming and trial-and-error debugging that these courses typically require. A cursory examination of the current popular data structures and algorithms texts reinforces this observation.

Finally, and not insignificantly, the course "theory of computation," which includes the mathematical principles underlying formal languages, automata, and Turing computability, is left as an elective (or not even offered) in increasing numbers of undergraduate programs. Since the theory course's topics are also not well integrated into the core of the undergraduate curriculum, we conclude that that core is fairly barren of mathematical content at most colleges and universities. The Liberal Arts Model Curriculum [11,16] is an exception to this conclusion.

2 Stopping the spread

To determine the mathematical needs of disciplines that are related to mathematics, the Committee on Undergraduate Preparation in Mathematics (CUPM) of the Mathematical Association of America is sponsoring workshops to help

inform a revision of its curriculum recommendations for undergraduate mathematics [6].

The first of these workshops addressed the mathematics needs of computer science and physics programs. In computer science, the workshop concluded that students "should be comfortable with abstract thinking, basic mathematical notation and its meaning. They should be able to generalize from examples and create examples of generalizations. In order to estimate the complexity of algorithms, they should have a feeling for functions that represent different rates of growth (e.g., logarithmic, polynomial, exponential). In order to reason effectively about the complexity and correctness of algorithms, they should have some facility with formal proofs, especially induction proofs. The same kind of clear and careful thinking and expression needed for a coherent mathematical argument is needed for the design and effective implementation of a computer program" [7]. This conclusion resonates with others' expressions of concern about the effective development and integration of the mathematical dimensions of computer science (e.g., [13]).

This workshop reaffirmed that computer science students should be able to model "real-world" problems precisely using mathematics and represent situations using structures such as arrays, linked lists, trees, finite graphs, and matrices. They should be able to design and analyze algorithms that transform these structures (e.g., [4]), understand the nature of a mathematical model, and relate mathematical models to real problem domains (e.g., [17, 18]). General problem solving strategies such as divide-and-conquer and backtracking are also essential.

This means that the first three courses for computer science majors – CS1, data structures, and computer organization – should integrate mathematical ideas and notations so that students become comfortable with them and see how they complement the main themes of computer science. Minimally, the following topics should be integrated into these first three courses: logical reasoning (propositions, DeMorgan's laws, including negation with quantifiers), functions, relations (equivalence relations and partitions), function and set notation (f: A −> B; A × B; A ∩ B) , mathematical induction (structural, strong and weak), combinatorics, finite probability, asymptotic notation (e.g., $O(n^2)$, $O(2^n)$), recurrence/difference equations, graphs and trees, and number systems. Examples below examples illustrate the integration of these topics into these courses.

Propositional logic and number systems A computer science student typically encounters the following code in a program in the CS1 course:

```
if ( (i > n)  &&  (a[i] != x) ) do thing1
       else do thing2
```

After some analysis, the student discovers that thing1 is not necessary, requiring that the condition of the if statement be negated and do thing2 be retained as the only alternative. The idea that negating this condition is an application of DeMorgan's law in logic should be helpful, allowing an avoidance of trial-and-error debugging to implement this change. While this situation occurs often in the early CS courses, many students have difficulty negating a compound logical expression such as this one.

Computer architecture is usually taught in the first two years of a computer science major. Decimal, binary, and hexadecimal number systems are used extensively. The use of logic expressions and their circuits to model the design of adders, multiplexors, and decoders are essential elements of this course. Fluency with logic is thus an important mathematical skill in this course.

Beyond these two examples, extended discussions of the centrality of logic in the computer science curriculum is provided in [15, 12].

Growth of functions During the analysis of nested loops, the sum $\Sigma_{k=1}^{n} k$ often occurs. The fact that this sum is equivalent to $n(n+1)/2$, and that as n grows this sum is different from the function n itself is important. The sum $\Sigma_{k=1}^{n} 1/k$ occurs often, as does its approximation $ln\ n$. Finally, the idea that $O(ln\ n) = O(log_2\ n)$ is also central.

Use of recurrence, induction, and finite probabability One of the best sorting algorithms is quicksort, which can be implemented as shown below.

```
//Pre:   0 <= first <= last
//Post:  a[first..last] is in ascending order
void quicksort(IntArr a, int first, int last)
{ int pivotind;  // pivot index
  int partdiv;   // partition division point
  if (first < last) // something's here to sort
  { pivotind = (first+last)/2; // pivot element
    partdiv = partition(a,first,last,pivotind);
    quicksort(a, first, partdiv-1); // sort left
    quicksort(a, partdiv+1, last); // and right
  }
}
```

Here, `partition` is a function that returns the index `partdiv` and rearranges the elements of the array `a` so that:

a[first..partdiv-1] <= a[partdiv] < a[partdiv+1..last]

In a separate argument, one can use a *loop invariant* to prove the (omitted) `partition` algorithm. *Strong induction* is used to prove `quicksort` correct. Attempting to prove an incorrectly formulated algorithm can sometimes be as effective as other debugging methods.

It can be shown that `partition` takes no more than n comparisons to partition an array of n elements. Using this fact and assuming that `partition` divides the array into equal portions, we get the recurrence $T(n) <= 2T(n/2) + n$

for `quicksort`, where $T(n)$ represents the number of comparisons to sort an array of n elements. If the initial ordering of the array is such that partition divides the array into parts containing 0 and n-1 elements, then the recurrence for `quicksort` is $T(n) < T(n\text{-}1) + n$. The first case yields O(n *log* n) complexity, while the second yields $O(n^2)$. An ability to derive and solve these recurrences is a key mathematical skill for computer science majors.

Students should also be able to analyze the *expected performance* of `quicksort`. If all orderings of the initial array are equally likely, the expected performance is O(n log n) and the constant hidden in the big-oh is small enough that `quicksort` is preferable to other sorting algorithms whose worst-case performance is O(n *log* n). Thus**,** comfort with finite probability is important in early computer science courses.

For another example, binary search trees are important data structures covered in a second computer science course. They are most easily defined using recursive definitions and most easily processed using recursive algorithms. For example, an inorder traversal of a binary search tree is easily expressed recursively but extremely difficult to code without using recursion. Many algorithms that analyze the complexity of binary search trees depend upon the height of the tree. Mathematical expressions that relate the height of the tree to the number of nodes in the tree are most easily proved by induction.

3 Mathematics curricular changes

In order to achieve effective access by computer science students to mathematical topics, it is clear that the topics found in discrete mathematics should be learned early; surely no later than the topics found in the calculus. While the recent calculus reform efforts to migrate away from "plug and chug" toward a more problem solving approach is laudable, this change is less relevant in impact on CS than would be similar reforms in the discrete mathematics course. The mathematics community's inattention to discrete mathematics reform has forced many computer science departments to teach these topics themselves.

Thus, we applaud recent experiments in the early mathematics curriculum that integrate the use of labs, group work, and peer learning [3, 14]. The use of these techniques has proven very beneficial in the early computer science curriculum, and we suspect that their use would be equally productive in the first discrete mathematics course (e.g., [9]). In any case, it is essential that mathematical curricula consistently offer the discrete mathematics course at least as early as the first calculus course, and without the encumbrance of calculus as a prerequisite.

4 CS curricular impact

The effective integration of mathematical themes and ideas into the computer science curriculum itself, though often promoted, has had an uneven history. For the most part, official curriculum recommendations from our professional societies and organizations stress that computer science majors must take mathematics courses during their undergraduate careers, though they are not particularly clear about the primacy of discrete mathematics. To a certain extent, they also provide points in the computer science curriculum where mathematical topics occur either as prerequisites or as integrated subject matter.

For example, the Computing Sciences Accreditation Board [5] recommends the following for undergraduate computer science majors: "The curriculum must include at least one-half year of mathematics. This material must include discrete mathematics, differential and integral calculus, and probability and statistics, and may include additional areas such as linear algebra, numerical analysis, combinatorics, and differential equations." Similar recommendations appear in the ACM/IEEE Curriculum 91 Report [1] and the Liberal Arts Model Curriculum [16, 11], which are widely used models for computer science major programs.

The core curriculum in computer science recommended by the ACM/IEEE Curriculum 91 report has a core set of knowledge units that span nine major subject areas and prescribe enough subject matter in these areas to cover about 7 semester-long courses. Of this, about 1-1/2 courses carry mathematics prerequisites (mostly discrete mathematics, but also some calculus and linear algebra). These parts of the computer science curriculum are more mathematical in nature, and most of them fall within the "algorithms and data structures" subject area.

The graduate record examination (GRE) in computer science (ftp://etsis1.ets.org/pub/gre/275516.pdf) weights 25% on theory and 15% on mathematical background. The theory topics depend heavily on discrete mathematics. Topics listed under mathematical background include:

A. Discrete Structures (Mathematical Logic; elementary combinatorics, including graph theory and counting arguments; discrete mathematics, including number theory, discrete probability, and recurrence relations);
B. Numerical mathematics (Computer arithmetic, including number representations, roundoff errors, overflow, and underflow; classical numerical algorithms; linear algebra).

In all these models, discrete mathematics topics take priority over calculus and linear algebra. If these discrete mathematics topics are not covered in a first- or second-semester mathematics course they must be introduced in the computer science courses themselves. These courses bear titles like "Discrete Structures" or "Computational Structures." (E.g., see [9, 10].) However, this option can slow down the development of other computer science topics and may lead to a more cursory treatment of mathematics topics than might occur if they were taught in a mathematics course. Given the current difficulty in hiring computer science faculty, we suspect that most

departments would welcome a freshman level discrete mathematics course from the mathematics department.

Mathematics at advanced levels in the computer science curriculum Many intermediate and advanced CS courses use mathematical topics that students hopefully master in their first two years. For example:

- Scientific computing courses use differential, integral, and multivariate calculus and linear algebra.
- Transforms are used in speech understanding and synthesis algorithms.
- Computer graphics courses use linear algebra, 3-dimensional calculus, and topics from in geometry.
- Wavelets, groups, and rings are used in compression and encryption algorithms.
- Operating systems and networking courses use probability and statistical methods.
- Theory of computation uses induction proofs. Proof by contradiction is also important here.

Often the first two computer science courses for majors are also required for majors from mathematics, the natural sciences, economics, social sciences, and other fields that require a deeper mastery of computer science fundamentals. For these students, a first semester discrete mathematics course would seem also to be of value. Moreover, these two courses should themselves have sufficient coverage of mathematical ideas that nonmajors' needs are well-served.

5 Conclusions

A joint IEEE/ACM Task Force on the "Year 2001 Model Curricula for Computing" [2] has been formed. Its charter is "to review the 1991 curricula and develop a revised and enhanced version for the Year 2001 that addresses developments in computing technologies in the past decade and will sustain through the next decade." We hope that this curriculum planning effort will interact effectively with the ongoing CUPM curriculum effort in mathematics [7] so that the interrelationships between mathematics and computer science can be more effectively expressed in their respective curriculum recommendations. This is an opportune moment in the history of computer science and mathematics curriculum development.

References

[1] Tucker, A., et al. "A Summary of the ACM/IEEE-CS Joint Task Force Report 'Computing Curricula 1991'". Communications of the ACM, June 1991. pp. 69-84. For the full report, including mathematics requirements for CS, see http://www.acm.org/education/curr91/homepage.html.

[2] "Year 2001 Model Curricula for Computing" http://computer.org/education/cc2001/index.htm.

[3] Calculus: The Dynamics of Change, MAA Notes 39, Mathematical Association of America, 1996.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms, MIT Press, Cambridge, Massachusetts, 1990, pp. 64-72.

[5] Computing Sciences Accreditation Board, Revised CSAC Evaluative Criteria, http://www.csab.org.

[6] Committee on the Undergraduate Program in Mathematics, The Undergraduate Major in the Mathematical Sciences. Washington, D.C.: Mathematical Association of America, 1991.

[7] Charles F. Kelemen (ed.), Owen Astrachan, Doug Baldwin, Kim Bruce, Peter Henderson, Dale Skrien, Allen Tucker, and Charles Van Loan, Computer Science Report to the CUPM Curriculum Foundations Workshop in Physics and Computer Science, Bowdoin College, October 28-31, 1999.

[8] Denning, P. et al., Computing as a Discipline, Communications of the ACM, April 1987.

[9] Susanna Epp. Discrete Mathematics with Applications, 2nd Edition, PWS Publishing, Boston, Massachusetts, 1995.

[10] Judith L. Gersting. Mathematical Structures for Computer Science, Fourth Edition, W.H. Freeman, 1999.

[11] Gibbs, N. and A. Tucker. "A Model Curriculum for a Liberal Arts Degree in Computer Science", Communications of the ACM, Mar. 1986. pp. 202-210.

[12] Gries, D. "The Need for Education in Useful Formal Logic", IEEE Computer, April 1996, pp. 29-30.

[13] Henderson, P.B., "Problem Solving, Discrete Mathematics and Computer Science," DIMACS Series on Discrete Mathematics and Theoretical Computer Science, Vol. 36, American Mathematical Society, 1997.

[14] Deborah Hughes-Hallett et al., Calculus, John Wiley and Sons, 1999.

[15]. J.P. Myers. "The Central Role of Mathematical Logic in Computer Science", ACM SIGCSE Bulletin. 22(1), pp 22-26, 1990.

[16]. Walker, H. and G. M. Schneider. "A Revised Model Curriculum for a Liberal Arts Degree in Computer Science", Communications of the ACM, Dec. 1996. pp. 85-95.

[17]. Wolz, U. and E. Conjura. "Integrating Mathematics and Programming into a Three Tiered Model for Computer Science", ACM SIGCSE Bulletin. 26(1), pp 223-227, 1994

[18] Woodcock, J. and M. Loomes. Software Engineering Mathematics, Addison-Wesley, 1988.