Computer Science Report to the
CUPM Curriculum Foundations Workshop in Physics and Computer Science

at Bowdoin College
October 28-31, 1999

Charles F. Kelemen (ed.), Owen Astrachan, Doug Baldwin,
Kim Bruce, Peter Henderson, Dale Skrien, Allen Tucker, and Charles Van Loan

This report focuses on the needs of computer science from the first two years of college mathematics instruction. While the authors have all been involved in computer science curriculum design in the past, this report does not represent the position of any official ACM or IEEE sanctioned curriculum committee.

Our general conclusion is that undergraduate computer science majors need to acquire mathematical maturity and skills, especially in discrete mathematics, early in their college education. The following topics are likely to be used in the first three courses for CS majors: logical reasoning, functions, relations, sets, mathematical induction, combinatorics, finite probability, asymptotic notation, recurrence/difference equations, graphs, trees, and number systems. Ultimately, calculus, linear algebra, and statistics topics are also needed, but none earlier than discrete mathematics. Thus, such a discrete mathematics course should be offered in the first semester and the prerequisite expectations and conceptual level should be the same as for the Calculus I course offered to mathematics and science majors. Our detailed recommendations respond directly to the series of questions of direct relevance to the CUPM Initiative posed by the Workshop hosts.

Understanding and Content

What conceptual mathematical principles must students master in the first two years?

Students should be comfortable with abstract thinking, 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 [Ralston 84, Henderson 97].

What mathematical problem solving skills must students master in the first two years?

Students should be able to represent 'real-world' problem situations with discrete structures such as arrays, linked lists, trees, finite graphs, other multi-linked structures, and matrices. They should be able to develop and analyze algorithms that operate on these structures (e.g., [Cormen 90]). They should understand what a mathematical model is and be able to relate mathematical models to real problem domains (e.g., [Wolz 94, Woodcock 88]). General problem solving strategies such as divide-and-conquer and backtracking strategies are also essential.

What broad mathematical topics must students master in the first two years?

The first three courses for CS majors are typically an introduction to computer science (containing a large amount of programming), a course in data structures and algorithms, and a course in computer architecture/organization. Some schools put computer architecture before data structures and some do the opposite. A few schools cover discrete mathematics topics before they do much programming [e.g., Baldwin 92, Henderson 90]. The following topics are likely to be used in the first three courses for CS majors: logical reasoning (propositions, DeMorgan’s laws, including negation with quantifiers), functions, relations (equivalence relations and partitions), sets, notation ($f: A \dashrightarrow B$; $A \times B$; $A \cap B$) , 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.

Some examples:

Propositional logic and number systems

A student may have the following code in a program:

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

After some analysis, it is discovered that `thing1` is not necessary at all. The student would like to negate the condition of the if statement and do `thing2` if the negated condition is true; an application of deMorgan’s law in propositional logic. This kind of change comes up often in the first two CS courses. Many students have great difficulty negating a compound logical expression such as the one above.

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 realize adders, multiplexors, decoders, etc. are essential for this course. Fluency with the propositional calculus is thus an important prerequisite here too.

Beyond these two examples, a more extended discussion of the centrality of logic in computer science is provided in [Meyers 90, Gries 96].

Growth of functions

In analyzing nested loops, the sum $\sum_{k=1}^{n} k$ occurs often. That this sum evaluates to $n(n+1)/2$ and that as n gets large this sum is quite different from n is important. The sum $\sum_{k=1}^{n} 1/k$ also appears often. That this is approximately ln n is important. In fact, the notion that $O(\ln n) = O(\log_2 n)$ is also important.

Use of recurrence, induction, and finite probabability

One of the best sorting algorithms is called quicksort. One variant follows.

```
//Pre:   0 <= first <= last < MAXARRSIZE
//Post:  a[first..last] is in ascending order
void quicksort(IntArr a, int first, int last)
{ int pivotind;   // pivot index before partitioning
  int partdiv;    //partition division point after partitioning
  if ((last - first) > 0)    //there is something here to sort
  { pivotind = (first+last)/2;  //pivot on middle element
    partdiv = partition(a,first, last, pivotind);
    quicksort(a, first, partdiv-1); //sort left part
    quicksort(a, partdiv+1, last);  //sort right part
  }  // end if
}  // end quicksort
```

`partition` is a function that returns an index `partdiv` and rearranges the elements of the array `a` so that after the return from `partition` we have
`a[first..partdiv-1] <= a[partdiv] < a[partdiv+1..last]`

In a separate argument, using loop invariants, one can prove that partition is correct. **Strong induction** is used to prove quicksort correct. Attempting to prove an incorrectly formulated algorithm correct is often the best way to find out what is wrong with it.

It can be shown that partition takes less than n 'element comparisons' to partition an array of n elements. Using this and assuming that partition always divides the array into equal portions, we get the **recurrence** $T(n) < 2T(n/2) + n$ where $T(n)$ represents the number of 'element comparisons' to quicksort 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 $T(n) < T(n-1) + n$. The first case yields $O(n \log n)$ complexity, while the second yields $O(n^2)$. Being able to derive recurrences of this sort and to solve them is important in early CS courses. 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 many other sorting algorithms whose worst-case performance is $O(n \log n)$. Thus, **Probabilistic analysis** is important.

Binary search trees are important data structures covered in a second CS 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 on binary search trees depend upon the height of the tree. Results relating the height of the tree to the number of nodes in the tree are most easily proved using **induction**.

Mathematics in the rest of the CS curriculum

Many intermediate and advanced CS courses use mathematical topics that students hopefully master in their first two years.

- Scientific computing and numerical analysis use differential and integral calculus, multidimensional calculus, and linear algebra.

- Computer Graphics uses linear algebra (matrix algebra, change of coordinates), 3-dimensional calculus, and topics from geometry.

- Theory of Computation and Algorithms courses use induction and diagonalization proofs. Counterexamples and proof by contradiction are important.

More advanced mathematical topics may also be used in select upper division CS courses.

- Transforms are used in speech understanding and synthesis algorithms.

- Wavelets are used in compression algorithms.

- Group and ring theory are used in encryption algorithms.

The Computing Sciences Accreditation Board [CSAB 99] recommends the following for undergraduate computer science majors: "The curriculum must include at least one-half year [4 or 5 courses] 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 [ACM/IEEE 91] and the Liberal Arts Model Curriculum [Walker 96, Gibbs 86], which are widely-used models for designing undergraduate computer science major programs in the US.

The GRE in computer science [GRE 99] weights 25% on Theory and 15% on mathematical background. The Theory topics depend heavily on discrete mathematics topics. Topics listed under mathematical background include:

- Discrete Structures (1. Mathematical Logic; 2. Elementary combinatorics, including graph theory and counting arguments; 3. Elementary discrete mathematics including number theory, discrete probability, recurrence relations);
- Numerical mathematics (1. Computer arithmetic including number representations, roundoff errors, overflow and underflow; 2. Classical numerical algorithms; 3. Linear algebra).

What priorities exist between these topics?

For the early CS courses, discrete mathematics topics take priority over calculus and linear algebra [Ralston 84]. If these discrete mathematics topics are not covered in a first- or second-semester mathematics course they must be introduced in the CS courses themselves. This slows down the CS course and probably leads to a more cursory treatment of the mathematics topics than might be possible in a mathematics course. Given the current difficulty in hiring CS faculty, we suspect that most CS departments would welcome a freshman level discrete mathematics course covering the topics needed for CS, but taught by the mathematics department. In fact, many CS departments consider these topics so important that they offer their own courses covering them. Some of these courses bear titles like "Discrete Structures" or "Computational Structures." (E.g., see [Epp 95, Gersting 99, Rosen 99] for a sampling of contemporary discrete mathematics texts.)

What is the desired balance between theoretical understanding and computational skill?

We think both theoretical understanding and computational skill are important. Computational skill (in the sense of plug and chug) is less important than the ability to recognize when these topics may be used productively in algorithmic problem solving and computational modeling. On the other hand, we would really like students to be able to formulate and complete induction proofs. If this is considered computational, then computational skill is very important.

What are the mathematical needs of different student populations and how can they be fulfilled?

CS courses for humanities students do not require sophisticated mathematics. CS courses specifically designed for business majors are well served by the business mathematics courses. Some colleges and universities offer special CS courses for science and engineering majors. These students have such heavy mathematics and science requirements in the first two years that it is probably not possible to require them to take a discrete mathematics course early. Covering some discrete mathematics topics (say induction and propositional logic) in Calculus I would be helpful for these CS courses. Ideally, for computer science majors, discrete mathematics should be covered before Calculus.

Often the first two CS courses for CS majors are also taken by majors from mathematics, the natural sciences, economics, social sciences, and others who want to gain a deeper mastery of this important field. For many of these students, a first semester discrete mathematics course would be of value.

Technology

How does technology affect what mathematics should be learned in the first two years?

We support the goal of FITness (Fluency in Information Technology) promulgated in the NAS report, "Being Fluent with Information Technology" [NAS 99]. The key idea is that students of all disciplines should learn enough foundational material in their formal education that they can embark on "a process of lifelong learning in which individuals continually apply what they know to adapt to change and acquire more knowledge to be more effective at applying information technology to their work and personal lives." In other words, everyone needs more than a superficial acquaintance with technology as a tool in their own areas of interest. Computer technology should be incorporated deeply and thoroughly into all mathematics curricula.

Instructional Interconnections

What impact does mathematics education reform have on instruction in your discipline?

We feel that the migration of Calculus toward problem solving (from “plug and chug”) is good, though less relevant in impact on CS than similar reforms in “Discrete Math” might be. The mathematics community’s inattention to Discrete Math early has forced many CS departments to assimilate and teach these topics themselves.

How should education reform in your discipline affect mathematics instruction?

The use of labs, group work, and peer learning has proven very beneficial in computer science education [Parker 90]. We suspect that the use of these techniques would be productive in some mathematics courses, especially discrete mathematics courses (e.g. [Epp 95]).

How can dialogue on educational issues between your discipline and mathematics best be maintained?

A joint IEEE Computer Society/ACM Task Force on the "Year 2001 Model Curricula for Computing" [ACM/IEEE 01] has been formed "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 the CUPM committee will be able to interact with this CS curriculum planning group. Other forums might be MAA and SIGCSE conferences and articles and newletters of these organizations.

Instructional Techniques

What instructional methods best develop the mathematical comprehension needed for your discipline?

We have no easy answers here. The following methods seem to work well in CS and we would presume that they might work well in mathematics: interactive collaborative learning leading to team/group reports, peer learning/teaching, learning center/laboratories (staffed), encouraging high-quality public written and oral communications, and providing research opportunities.

What guidance does educational research provide concerning mathematical training in your discipline?

We know that some work has been done on this but we are not familiar with the results. A Web site with many links in the general area of computer science education is: http://www.cacs.usl.edu/~mccauley/edlinks/. For research in computer science education see: http://www.cs.utexas.edu/users/csed/academic/.

Thanks

We would like to thank CUPM, CRAFTY, and the local organizers of the Bowdoin workshop (William Barker, Guy Emery, Allen Tucker, Katharine Billings) for a very pleasant and enlightening experience.

Biographical sketches of the computer science working group

Owen Astrachan is Associate Professor of the Practice and Director of Undergraduate Studies for Teaching and Learning in Computer Science at Duke University. He has an AB in mathematics from Dartmouth College (1978) and MAT (1979), MS (1989), and PhD (1992) degrees in computer science from Duke University. He has published in the areas of automated reasoning, parallel programming, and computer science education. He has served as Chief Reader for Advanced Placement Computer Science and chaired College Board/SIGCSE committees making recommendations to the AP program about language changes. He is the PI or co-PI on more than one million dollars of NSF-funded sponsored research including a CAREER award bridging software engineering and computer science education. He has won teaching awards at both Duke and the University of British Columbia.

Doug Baldwin is associate professor of computer science at SUNY Geneseo, where he has led the development of a mathematically rigorous introductory computer science course sequence. He is the author of a number of papers, and recipient of a number of grants, in computer science education. He holds BS, MS, and Ph. D. degrees in computer science from Yale University. He is a member of the ACM, and of the IEEE Computer Society, and served on the 1998 and1999 program committees for the annual symposium of the ACM's Special Interest Group on Computer Science Education.

Kim B. Bruce is Frederick Lattimer Wells Professor of Computer Science at Williams College, and has served as department chair several times. He received his BA at Pomona College and MA and Ph.D. in Mathematical Logic at the University of Wisconsin. He has been a visiting professor or visiting scientist at Princeton University, Stanford University, MIT, the Ecole Normale Superieure in Paris, the University of Pisa, and the Newton Institute for Mathematical Sciences at Cambridge University. He has published widely in the area of the semantics and design of programming languages as well as computer science education. He was a contributor to the ACM/IEEE-CS Joint Curriculum Task Force that developed Computing Curricula 1991, and participated in the design of the original 1986 Liberal Arts Model Curriculum in Computer

Science and its revision in 1996. He currently chairs the advisory committee on programming languages for the ACM/IEEE CS Curriculum 2001 effort, and is workshop chair for the series of workshops on Foundations of Object-Oriented Languages. He is a "Golden Core" member of IEEE CS and has received ACM Meritorious Service and Recognition of Service awards.

Professor Peter B. Henderson will be heading a new Computer Science/Software Engineering department at Butler University in Indianapolis. For the past 25 years he has been at SUNY Stony Brook working in the areas of software engineering, programming environments and computer science education. He received his B.S. and M.S. degrees in Electrical Engineering from Clarkson College and his Ph.D. from Princeton University. Professor Henderson has chaired three SIGSOFT/SIGPLAN Symposiums on Software Development Environments and has numerous publications in computer science and mathematics education.

Charles F. Kelemen is Professor of Computer Science at Swarthmore College. He earned a BA in Mathematics from Valparaiso University and an MA and PhD from the Pennsylvania State University. Before coming to Swarthmore College, he held regular faculty positions at Ithaca College and LeMoyne College and visiting positions in Computer Science at Cornell University. In 1985, Kelemen founded the Computer Science Program at Swarthmore College and chaired it from 1985 until 1999. He participated in the design of the original 1986 Liberal Arts Model Curriculum in Computer Science and its revision in 1996 and was a reviewer of the ACM/IEEE-CS Joint Computing Curricula 1991. He has published books, research, and educational articles in both mathematics and computer science. He is a member of ACM, IEEE-CS, CPSR, MAA, and the Liberal Arts Computer Science Consortium (LACS).

Dale Skrien is a Professor of Computer Science at Colby College. He has a BA in mathematics from St. Olaf College, an MS and PhD in mathematics from the University of Washington (1980), and an MS in computer science from the University of Illinois (1985). He has taught mathematics and computer science at Colby College since 1980. He has also been a Fulbright lecturer at the University of Malawi and a software engineer contractor for Digital Equipment Corporation. His research interests include graph theory, computer music, and computer science education.

Allen B. Tucker is the Anne T. and Robert M. Bass Professor of Natural Sciences in the Department of Computer Science at Bowdoin College. He has held similar positions at Geogetown and Colgate Universities. He has a BA in mathematics from Wesleyan University (1963) and an MS and PhD in computer science from Northwestern University (1970). He has various publications in the areas of programming languages, natural language processing, and computer science education. Professor Tucker co-chaired the ACM/IEEE-CS Joint Curriculum Task Force that developed Computing Curricula 1991 and is co-author of the 1986 Liberal Arts Model Curriculum in Computer Science. He is a Fellow of the ACM, and has been a member of the IEEE Computer Society, Computer Professionals for Social Responsibility, and the Liberal Arts Computer Science (LACS) Consortium.

Charles F. Van Loan is Chair of the Department of Computer Science at Cornell University where he is the Joseph C. Ford Professor of Engineering. He received his BS (1969), MS (1970), and PhD (1973) in Mathematics from the University of Michigan. He works in the area of matrix computations and has written several textbooks including Matrix Computations (with G.H. Golub), Computational Frameworks for the Fast Fourier Transform, Introduction to Scientific Computing, and Introduction to Computational Science and Mathematics. His papers "Computer Science and the Liberal Arts Student" and "Building Freshman Intuition for Computational Science" reflect his interest in undergraduate education. See http://www.cs.cornell.edu/cv/

References

[ACM/IEEE 91] 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 the mathematics requirements for CS majors, see http://www.acm.org/education/curr91/homepage.html.

[ACM/IEEE 01] "Year 2001 Model Curricula for Computing" http://computer.org/education/cc2001/index.htm.

[Baldwin 92] D. Baldwin and J. A. G. M. Koomen. "Using Scientific Experiments in Early Computer Science Laboratories". Proceedings of the Twenty-Third SIGCSE Technical Symposium on Computer Science Education, March 1992 (SIGCSE Bulletin, Mar. 1992). pp. 102-106.

[Cormen 90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms, MIT Press, Cambridge, Massachusetts, 1990, pp. 64-72.

[CSAB 99] Computing Sciences Accreditation Board, Revised CSAC Evaluative Criteria, http://www.csab.org.

[Epp 95] Susanna Epp. Discrete Mathematics with Applications, 2nd Edition, PWS Publishing, Boston, Massachusetts, 1995.

[Gersting 99] Judith L. Gersting. Mathematical Structures for Computer Science, Fourth Edition, W.H. Freeman, 1999.

[Gibbs 86] 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.

[GRE 99] Graduate Record Exams free publications, ftp://etsis1.ets.org/pub/gre/275516.pdf

[Gries 96] Gries, D. "The Need for Education in Useful Formal Logic", IEEE Computer, April 1996, pp. 29-30.

[Henderson 90] Henderson, P.B., "Discrete Mathematics as a Precursor to Computer Programming," Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education, Washington, DC, February 1990.

[Henderson 97] 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.

[Meyers 90]. J.P. Meyers. "The Central Role of Mathematical Logic in Computer Science", ACM SIGCSE Bulletin. 22(1), pp 22-26, 1990.

[NAS 99] National Academy of Sciences, "Being Fluent with Information Technology", National Academy Press (1999), http://books.nap.edu/catalog/6482.html

[Parker 90]. J. Parker, R. Cupper, C. Kelemen, R. Molnar, G. Scragg. "Laboratories in the Computer Science Curriculum", Computer Science Education 1, pp. 205-221, 1990.

[Ralston 84]. Anthony Ralston. "The First Course in Computer Science Needs a Mathematics Corequisite", Communications of the ACM, 27(10), pp 1002-1005, 1984.

[Rosen 99] Kenneth Rosen. Discrete Mathematics and its Applications, Fourth Edition, McGraw-Hill, 1999

[Stavely 99a] Allan M. Stavely. "High-Quality Software Through Semiformal Specification and Verification." Proc. 12th Conference on Software Engineering Education and Training (CSEE&T'99), March 22--24, 1999, New Orleans.

[Walker 96]. 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.

[Wolz 94]. 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

[Woodcock 88] Woodcock, J. and M. Loomes. Software Engineering Mathematics, Addison-Wesley, 1988.