Computer Science

CRAFTY Curriculum Foundations Project
Bowdoin College, October 28–31, 1999

Charles F. Kelemen, Report Editor
Allen Tucker and William Barker, Workshop Organizers

Summary

The general conclusion of the workshop participants 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 computer science 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.

While the authors of this report 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.

At the time of the Bowdoin meeting reported here, a joint task force of the Association for Computing Machinery (ACM) and Computer Society of the Institute for Electrical and Electronic Engineers (IEEE-CS) was beginning to formulate a new set of curriculum recommendations for undergraduate programs in computing. The final report (CC2001) of that joint task force (dated December 15, 2001) is available at:

`www.computer.org/education/cc2001/final/index.htm`

The mathematics recommendations to be found in CC2001 are consistent with those included here from the Bowdoin meeting. A few quotations concerning mathematics from CC2001 are included in Appendix 2.

Narrative

Introduction and Background

We are not aware of any concise, agreed upon, definition of Computer Science. A concise statement that appeals to the editor of this report was given by Jim Foley, Chair of the Computing Research Association Board. "Computer Science discovers and uses the laws of "how to" compute and "how to" organize information to create computational and information artifacts. Computer Science is also concerned with the organization—that is, the architecture—of the physical artifacts that perform computations and that store and transmit information."

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 have the mathematical background necessary to 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 computer science 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 computer science majors: logical reasoning (propositions, De Morgan's laws, including negation with quantifiers), functions, relations (equivalence relations and partitions), sets, notation ($f : A \rightarrow B$; $A \times B$; $A \cap B$) , 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 De Morgan's law in propositional logic. This kind of change comes up often in the first two computer science 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 logical 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 probability

One of the best sorting algorithms is called quicksort. Given an array to be sorted, say a[first..last], one array element is chosen as a "pivot". The array is then rearranged by the subprogram partition so that the pivot element is in its proper sorted location and all the elements with value less than or equal to the pivot value are moved to array locations with indices less than the pivot values new index (partdiv). All elements greater than the pivot value are moved to array locations with indices greater than partdiv. Thus the pivot element, now in a[partdiv], is in the correct position for the sorted array. Quicksort then "divides-and-conquers" by making recursive calls to itself to sort both the subarray of elements smaller than a[partdiv] and the subarray of elements larger than a[partdiv]. A version of quicksort is shown below that sorts the integer array a[first..last], with pre- and post-conditions as noted.

```
//Pre-condition: 0 <= first <= last < MAXARRSIZE
//Post-condition:     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);
     //The partition function returns the index for
     //the sorted position for the pivot element and
     //rearranges the array elements so that upon return
     // a[first..partdiv-1] <= a[partdiv]
     //        < a[partdiv+1..last]
    quicksort(a, first, partdiv-1);
     //sort left part
    quicksort(a, partdiv+1, last);
     //sort right part
  }  // end if
}    // end quicksort
```

In a separate argument, using loop invariants, one can prove that the function 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 $T(n) < T(n\text{-}1)+n$. 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 computer science 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 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 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 computer science curriculum

Many intermediate and advanced computer science courses use mathematical topics that we would like students to master.

- 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 computer science courses.

- Transforms are used in speech understanding and synthesis algorithms.
- Wavelets are used in compression algorithms.
- Number theory, 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 United States.The GRE in computer science [GRE 99] weights 25% on theory and 15% on mathematical background. Theory depends heavily on discrete mathematics.

Topics listed under mathematical background include:

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

What priorities exist among these topics?

For the early computer science 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 computer science courses themselves. This slows down the computer science course and probably leads to a more cursory treatment of the mathematics topics than might be appropriate in a mathematics course. Given the current difficulty in hiring computer science faculty, we suspect that most computer science departments would welcome a freshman-level discrete mathematics course covering the topics needed for computer science, but taught by the mathematics department. In fact, many computer science 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?

Computer science courses for humanities students do not require sophisticated mathematics. Computer science courses specifically designed for business majors are well served by the business mathematics courses. Some colleges and universities offer special computer science 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 computer science courses. Ideally, for computer science majors, discrete mathematics should be covered before Calculus.

Often the first two computer science courses for computer science 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 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 computer science 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.

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 computer science than similar reforms in Discrete Math might be. The mathematics community's inattention to Discrete Math early has forced many computer science 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 computer science curriculum planning group. Other forums might be MAA and SIGCSE conferences and articles and newsletters of these organizations.

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 computer science majors, see www.acm.org/education/curr91/homepage.html.

[ACM/IEEE 01] "Year 2001 Model Curricula for Computing" 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, 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), 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 99] 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.

WORKSHOP PARTICIPANTS

Computer Science Participants
(See the Appendix for detailed biographies.)

Owen Astrachan, Associate Professor of the Practice and Director of Undergraduate Studies for Teaching and Learning in Computer Science, Duke University.

Doug Baldwin, Associate Professor of Computer Science, State University of New York at Geneseo.

Kim B. Bruce, Frederick Lattimer Wells Professor of Computer Science, Williams College.

Peter B. Henderson, Professor and Head, Department of Computer Science and Software Engineering, Butler University.

Charles F. Kelemen, Professor of Computer Science, Swarthmore College.

Dale Skrien, Professor of Computer Science, Colby College.

Allen B. Tucker, Anne T. and Robert M. Bass Professor of Natural Sciences in the Department of Computer Science, Bowdoin College.

Charles F. Van Loan, Chair of the Department of Computer Science, Cornell University.

Mathematics Participants

Thomas Banchoff, Professor of Mathematics, Brown University.

William Barker, Professor of Mathematics, Bowdoin College.

Thomas Berger, Professor of Mathematics, Colby College.

Susan Ganter, Associate Professor of Mathematical Sciences, Clemson University.

Deborah Hughes Hallett, Professor of Mathematics, University of Arizona.

Harvey Keynes, Professor of Mathematics, University of Minnesota.

William McCallum, Professor of Mathematics, University of Arizona.

Donald Small, Professor of Mathematics, U.S. Military Academy, West Point.

ACKNOWLEDGEMENTS

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

APPENDIX 1 : Biographies of the Workshop Participants

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 the State University of New York at 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 and 1999 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 computer science 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.

Peter B. Henderson is professor and head of the Department of Computer Science and Software Engineering at Butler University in Indianapolis. For the previous 25 years he was 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 and Chair of Computer Science at Swarthmore College where he holds the Edward Hicks Magill Professorship of Mathematics and Natural Sciences. 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 `www.cs.cornell.edu/cv/`.

APPENDIX 2: Some Quotes from Curriculum 2001

This material is being added as this report goes to press in October, 2002. It was not available at the time of the Bowdoin meeting. The full ACM, IEEE-CS report is available at:

www.computer.org/education/cc2001/final/index.htm

From Section 7.4 Integrating discrete mathematics into the introductory curriculum.

As we discuss in , the CC2001 Task Force believes it is important for computer science students to study discrete mathematics early in their academic program, preferably in the first year. There are at least two workable strategies for accomplishing this goal:

From Appendix A: The CS body of knowledge.

DS. Discrete Structures (43 core hours)

DS1. Functions, relations and sets (6)
DS2. Basic Logic (10)
DS3. Proof Techniques (12)
DS4. Basics of Counting (5)
DS5. Graphs and trees (4)
DS6. Discrete Probability (6)

From Appendix B: CS Course Descriptions

CS115. *Discrete Structures for Computer Science*

Offers an intensive introduction to discrete mathematics as it is used in computer science. Topics include functions, relations, sets, propositional and predicate logic, simple circuit logic, proof techniques, elementary combinatorics, and discrete probability.

Prerequisites: Mathematical preparation sufficient to take calculus at the college level.

Syllabus:

- *Fundamental structures:* Functions (surjections, injections, inverses, composition); relations (reflexivity, symmetry, transitivity, equivalence relations); sets (Venn diagrams, complements, Cartesian products, power sets); pigeonhole principle; cardinality and countability
- *Basic logic:* Propositional logic; logical connectives; truth tables; normal forms (conjunctive and disjunctive); validity; predicate logic; limitations of predicate logic; universal and existential quantification; modus ponens and modus tollens
- *Digital logic:* Logic gates, flip-flops, counters; circuit minimization
- *Proof techniques:* Notions of implication, converse, inverse, contrapositive, negation, and contradiction; the structure of formal proofs; direct proofs; proof by counterexample; proof by contraposition; proof by contradiction; mathematical induction; strong induction; recursive mathematical definitions; well orderings
- *Basics of counting:* Counting arguments; pigeonhole principle; permutations and combinations; recurrence relations
- *Discrete probability:* Finite probability spaces; conditional probability, independence, Bayes' rule; random events; random integer variables; mathematical expectation.

Notes: This implementation of the Discrete Structures area (DS) compresses the core material into a single course. Although such a strategy is workable, many institutions will prefer to use two courses to cover this material in greater depth. For an implementation that uses the two-course model, see the descriptions of CS105 and CS106.

CS105. *Discrete Structures I*

Introduces the foundations of discrete mathematics as they apply to computer science, focusing on providing a solid theoretical foundation for further work. Topics include functions, relations, sets, simple proof techniques, Boolean algebra, propositional logic, digital logic, elementary number theory, and the fundamentals of counting.

Prerequisites: Mathematical preparation sufficient to take calculus at the college level.

Syllabus:

- *Introduction to logic and proofs:* Direct proofs; proof by contradiction; mathematical induction
- *Fundamental structures:* Functions (surjections, injections, inverses, composition); relations (reflexivity, symmetry, transitivity, equivalence relations); sets (Venn diagrams, complements, Cartesian products, power sets); pigeonhole principle; cardinality and countability
- *Boolean algebra:* Boolean values; standard operations on Boolean values; de Morgan's laws
- *Propositional logic:* Logical connectives; truth tables; normal forms (conjunctive and disjunctive); validity
- *Digital logic:* Logic gates, flip-flops, counters; circuit minimization
- *Elementary number theory:* Factorability; properties of primes; greatest common divisors and least common multiples; Euclid's algorithm; modular arithmetic; the Chinese Remainder Theorem
- *Basics of counting:* Counting arguments; pigeonhole principle; permutations and combinations; binomial coefficients

CS106. *Discrete Structures II*

Continues the discussion of discrete mathematics introduced in CS105. Topics in the second course include predicate logic, recurrence relations, graphs, trees, matrices, computational complexity, elementary computability, and discrete probability.

Prerequisites: CS105

Syllabus:

- *Review* of previous course
- *Predicate logic:* Universal and existential quantification; modus ponens and modus tollens; limitations of predicate logic
- *Recurrence relations:* Basic formulae; elementary solution techniques
- *Graphs and trees:* Fundamental definitions; simple algorithms ; traversal strategies; proof techniques; spanning trees; applications
- *Matrices:* Basic properties; applications
- *Computational complexity:* Order analysis; standard complexity classes
- *Elementary computability:* Countability and uncountability; diagonalization proof to show uncountability of the reals; definition of the P and NP classes; simple demonstration of the halting problem
- *Discrete probability:* Finite probability spaces; conditional probability, independence, Bayes' rule; random events; random integer variables; mathematical expectation

Notes: This implementation of the Discrete Structures area (DS) divides the material into two courses: CS105 and CS106. For programs that wish to accelerate the presentation of this material, there is also CS115, which covers the core topics in a single course. The two-course sequence, however, covers some additional material that is not in the compressed version, primarily in the Algorithms and Complexity area (AL). As a result, the introductory course in algorithmic analysis (CS210) can devote more time to advanced topics if an institution adopts the two-course implementation.

Like CS105, this course introduces mathematical topics in the context of applications that require those concepts as tools. For this course, likely applications include transportation network problems (such as the traveling salesperson problem) and resource allocation.