Pathway: Computer Science

Pathway: Computer Science

Approved: 12.10.14

Preamble: See document on page 2.

Executive Summary:

1. Exposure to Discrete Structures
   a. Should include all Core Tier-1 learning outcomes identified in ACM/IEEE CS2013 curricula.
   b. Includes topics and learning outcomes normally included in a rigorous Discrete Structures course in a Computer Science major.
   c. See below for a list of learning outcomes. The full ACM/IEEE CS2013 curricula are available at http://www.acm.org/education/CS2013-final-report.pdf.
2. Exposure to Software Development Fundamentals
   a. Should include all Core Tier-1 learning outcomes identified in ACM/IEEE CS2013 curricula.
   b. This includes topics and learning outcomes normally included in the first three Computer Science courses in many Computer Science majors. Students master fundamental aspects of software development/programming and data structures and algorithms.
   c. See below for list of learning outcomes. The full ACM/IEEE CS2013 curricula are available at http://www.acm.org/education/CS2013-final-report.pdf.
3. Calculus I and II
   a. Rigorous treatment of differential and integral calculus, i.e., "for math majors" versions of the courses – titled simply "Calculus 1" and "Calculus 2" in the Core Transfer Library at www.transferIN.net/ctl.
4. A lab science (physics, biology, chemistry)
   a. Rigorous study of physics, biology, or chemistry, including laboratories, i.e., "for majors" versions of the courses – titled "Biology 1 and 2 with Lab" and similarly for Chemistry and Physics in the Core Transfer Library at www.transferin.net.ctl.

Competencies and Learning Outcomes:

1. Discrete Structures
   1.1 Explain with examples the basic terminology of functions, relations, and sets. [Familiarity]
   1.2 Perform the operations associated with sets, functions, and relations. [Usage]
   1.3 Relate practical examples to the appropriate set, function, or relation model, and interpret the associated operations and terminology in context. [Assessment]
   1.4 Convert logical statements from informal language to propositional and predicate logic expressions. [Usage]

1.5 Apply formal methods of symbolic propositional and predicate logic, such as calculating validity of formulae and computing normal forms. [Usage]
1.6 Use the rules of inference to construct proofs in propositional and predicate logic. [Usage]
1.7 Describe how symbolic logic can be used to model real-life situations or applications, including those arising in computing contexts such as software analysis (e.g., program correctness), database queries, and algorithms. [Usage]
1.8 Apply formal logic proofs and/or informal, but rigorous, logical reasoning to real problems, such as predicting the behavior of software or solving problems such as puzzles. [Usage]
1.9 Describe the strengths and limitations of propositional and predicate logic. [Familiarity]
1.10 Identify the proof techniques used in a given proof. [Familiarity]
1.11 Outline the basic structure of each proof technique (direct proof, proof by contradiction, and induction) described in this using. [Usage]
1.12 Apply each of the proof techniques (direct proof, proof by contradiction, and induction) correctly in the construction of a sound argument. [Usage]
1.13 Determine which type of proof is best for a given problem. [Assessment]
1.14 Explain the parallels between ideas of mathematical and/or structural induction to recursion and recursively defined structures. [Assessment]
1.15 Explain the relationship between weak and strong induction and give examples of the appropriate use of each. [Assessment]
1.16 Apply counting arguments, including sum and product rules, inclusion-exclusion principle and arithmetic/geometric progressions. [Usage]
1.17 Apply the pigeonhole principle in the context of a formal proof. [Usage]
1.18 Compute permutations and combinations of a set, and interpret the meaning in the context of the particular application. [Usage]
1.19 Map real-world applications to appropriate counting formalisms, such as determining the number of ways to arrange people around a table, subject to constraints on the seating arrangement, or the number of ways to determine certain hands in cards (e.g., a full house). [Usage]
1.20 Solve a variety of basic recurrence relations. [Usage]
1.21 Analyze a problem to determine underlying recurrence relations. [Usage]
1.22 Perform computations involving modular arithmetic. [Usage]
1.23 Illustrate by example the basic terminology of graph theory, as well as some of the properties and special cases of each type of graph/tree. [Familiarity]
1.24 Demonstrate different traversal methods for trees and graphs, including pre-, post-, and in-order traversal of trees. [Usage]

1.25 Model a variety of real-world problems in computer science using appropriate forms of graphs and trees, such as representing a network topology or the organization of a hierarchical file system. [Usage]
1.26 Show how concepts from graphs and trees appear in data structures, algorithms, proof techniques (structural induction), and counting. [Usage]
1.27 Calculate probabilities of events and expectations of random variables for elementary problems such as games of chance. [Usage]
1.28 Differentiate between dependent and independent events. [Usage]
1.29 Identify a case of the binomial distribution and compute a probability using that distribution. [Usage]
1.30 Apply Bayes' theorem to determine conditional probabilities in a problem. [Usage]
1.31 Apply the tools of probability to solve problems such as the average case analysis of algorithms or analyzing hashing. [Usage]

2. Software Development Fundamentals
   2.1 Discuss the importance of algorithms in the problem-solving process. [Familiarity]
   2.2 Discuss how a problem may be solved by multiple algorithms, each with different properties. [Familiarity]
   2.3 Create algorithms for solving simple problems. [Usage]
   2.4 Use a programming language to implement, test, and debug algorithms for solving simple problems. [Usage]
   2.5 Implement, test, and debug simple recursive functions and procedures. [Usage]
   2.6 Determine whether a recursive or iterative solution is most appropriate for a problem. [Assessment]
   2.7 Implement a divide-and-conquer algorithm for solving a problem. [Usage]
   2.8 Apply the techniques of decomposition to break a program into smaller pieces. [Usage]
   2.9 Identify the data components and behaviors of multiple abstract data types. [Usage]
   2.10 Implement a coherent abstract data type, with loose coupling between components and behaviors. [Usage]
   2.11 Identify the relative strengths and weaknesses among multiple designs or implementations for a problem. [Assessment]
   2.12 Analyze and explain the behavior of simple programs involving the fundamental programming constructs variables, expressions, assignments, I/O, control constructs, functions, parameter passing, and recursion. [Assessment]
   2.13 Identify and describe uses of primitive data types. [Familiarity]
   2.14 Write programs that use primitive data types. [Usage]

2.15 Modify and expand short programs that use standard conditional and iterative control structures and functions. [Usage]
2.16 Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, the definition of functions, and parameter passing. [Usage]
2.17 Write a program that uses file I/O to provide persistence across multiple executions. [Usage]
2.18 Choose appropriate conditional and iteration constructs for a given programming task. [Assessment]
2.19 Describe the concept of recursion and give examples of its use. [Familiarity]
2.20 Identify the base case and the general case of a recursively-defined problem. [Assessment]
2.21 Discuss the appropriate use of built-in data structures. [Familiarity]
2.22 Describe common applications for each of the following data structures: stack, queue, priority queue, set, and map. [Familiarity]
2.23 Write programs that use each of the following data structures: arrays, records/structs, strings, linked lists, stacks, queues, sets, and maps. [Usage]
2.24 Compare alternative implementations of data structures with respect to performance. [Assessment]
2.25 Describe how references allow for objects to be assessed in multiple ways. [Familiarity]
2.26 Compare and contrast the costs and benefits of dynamic and static data structure implementations. [Assessment]
2.27 Choose the appropriate data structure for modeling a given problem. [Assessment]
2.28 Trace the execution of a variety of code segments and write summaries of their computations. [Assessment]
2.29 Explain why the creation of correct program components is important in the production of high-quality software. [Familiarity]
2.30 Identify common coding errors that lead to insecure programs (e.g., buffer overflows, memory leaks, malicious code) and apply strategies for avoiding such errors. [Usage]
2.31 Conduct a personal code review (focused on common coding errors) on a program component using a provided checklist. [Usage]
2.32 Contribute to a small-team code review focused on component correctness. [Usage]
2.33 Describe how a contract can be used to specify the behavior of a program component. [Familiarity]

2.34 Refractor a program by identifying opportunities to apply procedural abstraction. [Usage]

2.35 Apply a variety of strategies to the testing and debugging of simple programs. [Usage]

2.36 Construct, execute and debug programs using a modern IDE and associated tools such as unit testing tools and visual debuggers. [Usage]

2.37 Construct and debug programs using the standard libraries available with a chosen programming language. [Usage]

2.38 Analyze the extent to which another programmer's code meets documentation and programming style standards. [Assessment]

2.39 Apply consistent documentation and program style standards that contribute to the readability and maintainability of software. [Usage]