Organization of the Body of Knowledge

The CS2013 Body of Knowledge is presented as a set of Knowledge Areas (KAs), organized on topical themes rather than by course boundaries. Each KA is further organized into a set of Knowledge Units (KUs), which are summarized in a table at the head of each KA section. We expect that the topics within the KAs will be organized into courses in different ways at different institutions.

Curricular Hours

Continuing in the tradition of CC2001/CS2008, we define the unit of coverage in the Body of Knowledge in terms of **lecture hours**, as being the sole unit that is understandable in (and transferable to) cross-cultural contexts. An "hour" corresponds to the time required to present the material in a traditional lecture-oriented format; the hour count does not include any additional work that is associated with a lecture (e.g., in self-study, laboratory sessions, and assessments). Indeed, we expect students to spend a significant amount of additional time outside of class developing facility with the material presented in class. As with previous reports, we maintain the principle that the use of a lecture-hour as the unit of measurement does not require or endorse the use of traditional lectures for the presentation of material.

The specification of topic hours represents the minimum amount of time we expect such coverage to take. Any institution may opt to cover the same material in a longer period of time as warranted by the individual needs of that institution.

Courses

Throughout the Body of Knowledge, when we refer to a "course" we mean an institutionally-recognized unit of study. Depending on local circumstance, full-time students will take several "courses" at any one time, typically several per academic year. While "course" is a common term at some institutions, others will use other names, for example "module" or "paper."

Guidance on Learning Outcomes

Each KU within a KA lists both a set of topics and the learning outcomes students are expected to achieve with respect to the topics specified. Learning outcomes are not of equal size and do not have a uniform mapping to curriculum hours; topics with the same number of hours may have quite different numbers of associated learning outcomes. Each learning outcome has an associated level of mastery. In defining different levels we drew from other curriculum approaches, especially Bloom's Taxonomy, which has been well explored within computer science. We did not directly apply Bloom's levels in part because several of them are driven by pedagogic context, which would introduce too much plurality in a document of this kind; in part because we intend the mastery levels to be indicative and not to impose theoretical constraint on users of this document.

We use three levels of mastery, defined as:

- ***Familiarity***: The student understands what a concept is or what it means. This level of mastery concerns a basic awareness of a concept as opposed to expecting real facility with its application. It provides an answer to the question “What do you know about this?”
- ***Usage***: The student is able to use or apply a concept in a concrete way. Using a concept may include, for example, appropriately using a specific concept in a program, using a particular proof technique, or performing a particular analysis. It provides an answer to the question “What do you know how to do?”
- ***Assessment***: The student is able to consider a concept from multiple viewpoints and/or justify the selection of a particular approach to solve a problem. This level of mastery implies more than using a concept; it involves the ability to select an appropriate approach from understood alternatives. It provides an answer to the question “Why would you do that?”

As a concrete, although admittedly simplistic, example of these levels of mastery, we consider the notion of iteration in software development, for example for-loops, while-loops, and iterators. At the level of “Familiarity,” a student would be expected to have a definition of the concept of iteration in software development and know why it is a useful technique. In order to show mastery at the “Usage” level, a student should be able to write a program properly using a form of iteration. Understanding iteration at the “Assessment” level would require a student to understand multiple methods for iteration and be able to appropriately select among them for different applications.

The descriptions we have included for learning outcomes may not exactly match those used by institutions, in either specifics or emphasis. Institutions may have different learning outcomes that capture the same level of mastery and intent for a given topic. Nevertheless, we believe that by giving descriptive learning outcomes, we both make our intention clear and facilitate interpretation of what outcomes mean in the context of a particular curriculum.

Appendix A: The Body of Knowledge

Algorithms and Complexity (AL)

Algorithms are fundamental to computer science and software engineering. The real-world performance of any software system depends on: (1) the algorithms chosen and (2) the suitability and efficiency of the various layers of implementation. Good algorithm design is therefore crucial for the performance of all software systems. Moreover, the study of algorithms provides insight into the intrinsic nature of the problem as well as possible solution techniques independent of programming language, programming paradigm, computer hardware, or any other implementation aspect.

An important part of computing is the ability to select algorithms appropriate to particular purposes and to apply them, recognizing the possibility that no suitable algorithm may exist. This facility relies on understanding the range of algorithms that address an important set of well-defined problems, recognizing their strengths and weaknesses, and their suitability in particular contexts. Efficiency is a pervasive theme throughout this area.

This knowledge area defines the central concepts and skills required to design, implement, and analyze algorithms for solving problems. Algorithms are essential in all advanced areas of computer science: artificial intelligence, databases, distributed computing, graphics, networking, operating systems, programming languages, security, and so on. Algorithms that have specific utility in each of these are listed in the relevant knowledge areas. Cryptography, for example, appears in the new Knowledge Area on Information Assurance and Security (IAS), while parallel and distributed algorithms appear the Knowledge Area in Parallel and Distributed Computing (PD).

As with all knowledge areas, the order of topics and their groupings do not necessarily correlate to a specific order of presentation. Different programs will teach the topics in different courses and should do so in the order they believe is most appropriate for their students.

AL. Algorithms and Complexity (19 Core-Tier1 hours, 9 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
AL/Basic Analysis	2	2	N
AL/Algorithmic Strategies	5	1	N
AL/Fundamental Data Structures and Algorithms	9	3	N
AL/Basic Automata, Computability and Complexity	3	3	N
AL/Advanced Computational Complexity			Y
AL/Advanced Automata Theory and Computability			Y
AL/Advanced Data Structures, Algorithms, and Analysis			Y

AL/Basic Analysis

[2 Core-Tier1 hours, 2 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Differences among best, expected, and worst case behaviors of an algorithm
- Asymptotic analysis of upper and expected complexity bounds
- Big O notation: formal definition
- Complexity classes, such as constant, logarithmic, linear, quadratic, and exponential
- Empirical measurements of performance
- Time and space trade-offs in algorithms

[Core-Tier2]

- Big O notation: use
- Little o, big omega and big theta notation
- Recurrence relations
- Analysis of iterative and recursive algorithms
- Some version of a Master Theorem

Learning Outcomes:

[Core-Tier1]

1. Explain what is meant by “best”, “expected”, and “worst” case behavior of an algorithm. [Familiarity]
2. In the context of specific algorithms, identify the characteristics of data and/or other conditions or assumptions that lead to different behaviors. [Assessment]
3. Determine informally the time and space complexity of simple algorithms. [Usage]

4. State the formal definition of big O. [Familiarity]
5. List and contrast standard complexity classes. [Familiarity]
6. Perform empirical studies to validate hypotheses about runtime stemming from mathematical analysis. Run algorithms on input of various sizes and compare performance. [Assessment]
7. Give examples that illustrate time-space trade-offs of algorithms. [Familiarity]

[Core-Tier2]

8. Use big O notation formally to give asymptotic upper bounds on time and space complexity of algorithms. [Usage]
9. Use big O notation formally to give expected case bounds on time complexity of algorithms. [Usage]
10. Explain the use of big omega, big theta, and little o notation to describe the amount of work done by an algorithm. [Familiarity]
11. Use recurrence relations to determine the time complexity of recursively defined algorithms. [Usage]
12. Solve elementary recurrence relations, e.g., using some form of a Master Theorem. [Usage]

AL/Algorithmic Strategies

[5 Core-Tier1 hours, 1 Core-Tier2 hours]

An instructor might choose to cover these algorithmic strategies in the context of the algorithms presented in "Fundamental Data Structures and Algorithms" below. While the total number of hours for the two knowledge units (18) could be divided differently between them, our sense is that the 1:2 ratio is reasonable.

Topics:

[Core-Tier1]

- Brute-force algorithms
- Greedy algorithms
- Divide-and-conquer (cross-reference SDF/Algorithms and Design/Problem-solving strategies)
- Recursive backtracking
- Dynamic Programming

[Core-Tier2]

- Branch-and-bound
- Heuristics
- Reduction: transform-and-conquer

Learning Outcomes:

[Core-Tier1]

1. For each of the strategies (brute-force, greedy, divide-and-conquer, recursive backtracking, and dynamic programming), identify a practical example to which it would apply. [Familiarity]
2. Use a greedy approach to solve an appropriate problem and determine if the greedy rule chosen leads to an optimal solution. [Assessment]
3. Use a divide-and-conquer algorithm to solve an appropriate problem. [Usage]
4. Use recursive backtracking to solve a problem such as navigating a maze. [Usage]
5. Use dynamic programming to solve an appropriate problem. [Usage]
6. Determine an appropriate algorithmic approach to a problem. [Assessment]

[Core-Tier2]

7. Describe various heuristic problem-solving methods. [Familiarity]
8. Use a heuristic approach to solve an appropriate problem. [Usage]
9. Describe the trade-offs between brute force and heuristic strategies. [Assessment]
10. Describe how a branch-and-bound approach may be used to improve the performance of a heuristic method. [Familiarity]

AL/Fundamental Data Structures and Algorithms

[9 Core-Tier1 hours, 3 Core-Tier2 hours]

This knowledge unit builds directly on the foundation provided by Software Development Fundamentals (SDF), particularly the material in SDF/Fundamental Data Structures and SDF/Algorithms and Design.

Topics:

[Core-Tier1]

- Simple numerical algorithms, such as computing the average of a list of numbers, finding the min, max, and mode in a list, approximating the square root of a number, or finding the greatest common divisor
- Sequential and binary search algorithms
- Worst case quadratic sorting algorithms (selection, insertion)
- Worst or average case O(N log N) sorting algorithms (quicksort, heapsort, mergesort)
- Hash tables, including strategies for avoiding and resolving collisions
- Binary search trees
  - Common operations on binary search trees such as select min, max, insert, delete, iterate over tree
- Graphs and graph algorithms
  - Representations of graphs (e.g., adjacency list, adjacency matrix)
  - Depth- and breadth-first traversals

[Core-Tier2]

- Heaps
- Graphs and graph algorithms
  - Shortest-path algorithms (Dijkstra's and Floyd's algorithms)
  - Minimum spanning tree (Prim's and Kruskal's algorithms)
- Pattern matching and string/text algorithms (e.g., substring matching, regular expression matching, longest common subsequence algorithms)

Learning Outcomes:

[Core-Tier1]

1. Implement basic numerical algorithms. [Usage]
2. Implement simple search algorithms and explain the differences in their time complexities. [Assessment]
3. Be able to implement common quadratic and O(N log N) sorting algorithms. [Usage]
4. Describe the implementation of hash tables, including collision avoidance and resolution. [Familiarity]
5. Discuss the runtime and memory efficiency of principal algorithms for sorting, searching, and hashing. [Familiarity]
6. Discuss factors other than computational efficiency that influence the choice of algorithms, such as programming time, maintainability, and the use of application-specific patterns in the input data. [Familiarity]
7. Explain how tree balance affects the efficiency of various binary search tree operations. [Familiarity]
8. Solve problems using fundamental graph algorithms, including depth-first and breadth-first search. [Usage]

9. Demonstrate the ability to evaluate algorithms, to select from a range of possible options, to provide justification for that selection, and to implement the algorithm in a particular context. [Assessment]

[Core-Tier2]

10. Describe the heap property and the use of heaps as an implementation of priority queues. [Familiarity]
11. Solve problems using graph algorithms, including single-source and all-pairs shortest paths, and at least one minimum spanning tree algorithm. [Usage]
12. Trace and/or implement a string-matching algorithm. [Usage]

AL/Basic Automata Computability and Complexity

[3 Core-Tier1 hours, 3 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Finite-state machines
- Regular expressions
- The halting problem

[Core-Tier2]

- Context-free grammars (cross-reference PL/Syntax Analysis)
- Introduction to the P and NP classes and the P vs. NP problem
- Introduction to the NP-complete class and exemplary NP-complete problems (e.g., SAT, Knapsack)

Learning Outcomes:

[Core-Tier1]

1. Discuss the concept of finite state machines. [Familiarity]
2. Design a deterministic finite state machine to accept a specified language. [Usage]
3. Generate a regular expression to represent a specified language. [Usage]
4. Explain why the halting problem has no algorithmic solution. [Familiarity]

[Core-Tier2]

5. Design a context-free grammar to represent a specified language. [Usage]
6. Define the classes P and NP. [Familiarity]
7. Explain the significance of NP-completeness. [Familiarity]

Computational Science (CN)

Computational Science is a field of applied computer science, that is, the application of computer science to solve problems across a range of disciplines. In the book *Introduction to Computational Science* [3], the authors offer the following definition: "the field of computational science combines computer simulation, scientific visualization, mathematical modeling, computer programming and data structures, networking, database design, symbolic computation, and high performance computing with various disciplines." Computer science, which largely focuses on the theory, design, and implementation of algorithms for manipulating data and information, can trace its roots to the earliest devices used to assist people in computation over four thousand years ago. Various systems were created and used to calculate astronomical positions. Ada Lovelace's programming achievement was intended to calculate Bernoulli numbers. In the late nineteenth century, mechanical calculators became available, and were immediately put to use by scientists. The needs of scientists and engineers for computation have long driven research and innovation in computing. As computers increase in their problem-solving power, computational science has grown in both breadth and importance. It is a discipline in its own right [2] and is considered to be "one of the five college majors on the rise [1]." An amazing assortment of sub-fields have arisen under the umbrella of Computational Science, including computational biology, computational chemistry, computational mechanics, computational archeology, computational finance, computational sociology and computational forensics.

Some fundamental concepts of computational science are germane to every computer scientist (e.g., modeling and simulation), and computational science topics are extremely valuable components of an undergraduate program in computer science. This area offers exposure to many valuable ideas and techniques, including precision of numerical representation, error analysis, numerical techniques, parallel architectures and algorithms, modeling and simulation, information visualization, software engineering, and optimization. Topics relevant to computational science include fundamental concepts in program construction (SDF/Fundamental Programming Concepts), algorithm design (SDF/Algorithms and Design), program testing (SDF/Development Methods), data representations (AR/Machine Representation of Data), and basic computer architecture (AR/Memory System Organization and Architecture). At the same time, students who take courses in this area have an opportunity to apply these techniques in a wide range of application areas, such as molecular and fluid dynamics, celestial mechanics, economics, biology, geology, medicine, and social network analysis. Many of the techniques used in these areas require advanced mathematics such as calculus, differential equations, and linear algebra. The descriptions here assume that students have acquired the needed mathematical background elsewhere.

In the computational science community, the terms *run*, *modify*, and *create* are often used to describe levels of understanding. This chapter follows the conventions of other chapters in this volume and uses the terms *familiarity*, *usage*, and *assessment*.

References

[1] Fischer, K. and Glenn, D., "5 College Majors on the Rise," *The Chronicle of Higher Education*, August 31, 2009.

[2] President's Information Technology Advisory Committee, 2005: p. 13. http://www.nitrd.gov/pitac/reports/20050609_computational/computational.pdf

[3] Shiflet, A. B. and Shiflet, G. W. *Introduction to Computational Science: Modeling and Simulation for the Sciences*, Princeton University Press, 2006: p. 3.

CN. Computational Science (1 Core-Tier1 hours, 0 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
CN/Introduction to Modeling and Simulation	1		N
CN/Modeling and Simulation			Y
CN/Processing			Y
CN/Interactive Visualization			Y
CN/Data, Information, and Knowledge			Y
CN/Numerical Analysis			Y

CN/Introduction to Modeling and Simulation

[1 Core-Tier1 hours]

Abstraction is a fundamental concept in computer science. A principal approach to computing is to abstract the real world, create a model that can be simulated on a machine. The roots of computer science can be traced to this approach, modeling things such as trajectories of artillery shells and the modeling cryptographic protocols, both of which pushed the development of early computing systems in the early and mid-1940's.

Modeling and simulation of real world systems represent essential knowledge for computer scientists and provide a foundation for computational sciences. Any introduction to modeling and simulation would either include or presume an introduction to computing. In addition, a general set of modeling and simulation techniques, data visualization methods, and software testing and evaluation mechanisms are also important.

Topics:

- Models as abstractions of situations
- Simulations as dynamic modeling
- Simulation techniques and tools, such as physical simulations, human-in-the-loop guided simulations, and virtual reality
- Foundational approaches to validating models (e.g., comparing a simulation's output to real data or the output of another model)
- Presentation of results in a form relevant to the system being modeled

Learning Outcomes:

1. Explain the concept of modeling and the use of abstraction that allows the use of a machine to solve a problem. [Familiarity]
2. Describe the relationship between modeling and simulation, i.e., thinking of simulation as dynamic modeling. [Familiarity]
3. Create a simple, formal mathematical model of a real-world situation and use that model in a simulation. [Usage]
4. Differentiate among the different types of simulations, including physical simulations, human-guided simulations, and virtual reality. [Familiarity]
5. Describe several approaches to validating models. [Familiarity]
6. Create a simple display of the results of a simulation. [Usage]

PL/Object-Oriented Programming

[4 Core-Tier1 hours, 6 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Object-oriented design
    - Decomposition into objects carrying state and having behavior
    - Class-hierarchy design for modeling
- Definition of classes: fields, methods, and constructors
- Subclasses, inheritance, and method overriding
- Dynamic dispatch: definition of method-call

[Core-Tier2]

- Subtyping (cross-reference PL/Type Systems)
    - Subtype polymorphism; implicit upcasts in typed languages
    - Notion of behavioral replacement: subtypes acting like supertypes
    - Relationship between subtyping and inheritance
- Object-oriented idioms for encapsulation
    - Privacy and visibility of class members
    - Interfaces revealing only method signatures
    - Abstract base classes
- Using collection classes, iterators, and other common library components

Learning outcomes:

[Core-Tier1]

1. Design and implement a class. [Usage]
2. Use subclassing to design simple class hierarchies that allow code to be reused for distinct subclasses. [Usage]
3. Correctly reason about control flow in a program using dynamic dispatch. [Usage]
4. Compare and contrast (1) the procedural/functional approach (defining a function for each operation with the function body providing a case for each data variant) and (2) the object-oriented approach (defining a class for each data variant with the class definition providing a method for each operation). Understand both as defining a matrix of operations and variants. [Assessment] *This outcome also appears in PL/Functional Programming.*

[Core-Tier2]

5. Explain the relationship between object-oriented inheritance (code-sharing and overriding) and subtyping (the idea of a subtype being usable in a context that expects the supertype). [Familiarity]
6. Use object-oriented encapsulation mechanisms such as interfaces and private members. [Usage]
7. Define and use iterators and other operations on aggregates, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language. [Usage] *This outcome also appears in PL/Functional Programming.*

PL/Event-Driven and Reactive Programming

[2 Core-Tier2 hours]

This material can stand alone or be integrated with other knowledge units on concurrency, asynchrony, and threading to allow contrasting events with threads.

Topics:

- Events and event handlers
- Canonical uses such as GUIs, mobile devices, robots, servers
- Using a reactive framework
    - Defining event handlers/listeners
    - Main event loop not under event-handler-writer's control
- Externally-generated events and program-generated events
- Separation of model, view, and controller

Learning outcomes:

1. Write event handlers for use in reactive systems, such as GUIs. [Usage]
2. Explain why an event-driven programming style is natural in domains where programs react to external events. [Familiarity]
3. Describe an interactive system in terms of a model, a view, and a controller. [Familiarity]

Software Development Fundamentals (SDF)

Fluency in the process of software development is a prerequisite to the study of most of computer science. In order to use computers to solve problems effectively, students must be competent at reading and writing programs in multiple programming languages. Beyond programming skills, however, they must be able to design and analyze algorithms, select appropriate paradigms, and utilize modern development and testing tools. This knowledge area brings together those fundamental concepts and skills related to the software development process. As such, it provides a foundation for other software-oriented knowledge areas, most notably Programming Languages, Algorithms and Complexity, and Software Engineering.

It is important to note that this knowledge area is distinct from the old Programming Fundamentals knowledge area from CC2001. Whereas that knowledge area focused exclusively on the programming skills required in an introductory computer science course, this new knowledge area is intended to fill a much broader purpose. It focuses on the entire software development process, identifying those concepts and skills that should be mastered in the first year of a computer science program. This includes the design and simple analysis of algorithms, fundamental programming concepts and data structures, and basic software development methods and tools. As a result of its broader purpose, the Software Development Fundamentals knowledge area includes fundamental concepts and skills that could naturally be listed in other software-oriented knowledge areas (e.g., programming constructs from Programming Languages, simple algorithm analysis from Algorithms & Complexity, simple development methodologies from Software Engineering). Likewise, each of these knowledge areas will contain more advanced material that builds upon the fundamental concepts and skills listed here.

While broader in scope than the old Programming Fundamentals, this knowledge area still allows for considerable flexibility in the design of first-year curricula. For example, the Fundamental Programming Concepts unit identifies only those concepts that are common to all programming paradigms. It is expected that an instructor would select one or more programming paradigms (e.g., object-oriented programming, functional programming, scripting) to illustrate these programming concepts, and would pull paradigm-specific content from the Programming Languages knowledge area to fill out a course. Likewise, an instructor could choose to emphasize formal analysis (e.g., Big-Oh, computability) or design methodologies (e.g., team projects, software life cycle) early, thus integrating hours from the Programming Languages, Algorithms and Complexity, and/or Software Engineering knowledge areas. Thus, the 43 hours of material in this knowledge area will typically be augmented with core material from one or more of these knowledge areas to form a complete and coherent first-year experience.

When considering the hours allocated to each knowledge unit, it should be noted that these hours reflect the minimal amount of classroom coverage needed to introduce the material. Many software development topics will reappear and be reinforced by later topics (e.g., applying iteration constructs when processing lists). In addition, the mastery of concepts and skills from this knowledge area requires a significant amount of software development experience outside of class.

SDF. Software Development Fundamentals (43 Core-Tier1 hours)

	Core-Tier1 hours	**Core-Tier2 hours**	**Includes Electives**
SDF/Algorithms and Design	**11**		**N**
SDF/Fundamental Programming Concepts	**10**		**N**
SDF/Fundamental Data Structures	**12**		**N**
SDF/Development Methods	**10**		**N**

SDF/Algorithms and Design

[11 Core-Tier1 hours]

This unit builds the foundation for core concepts in the Algorithms and Complexity Knowledge Area, most notably in the Basic Analysis and Algorithmic Strategies knowledge units.

Topics:

- The concept and properties of algorithms
    - Informal comparison of algorithm efficiency (e.g., operation counts)
- The role of algorithms in the problem-solving process
- Problem-solving strategies
    - Iterative and recursive mathematical functions
    - Iterative and recursive traversal of data structures
    - Divide-and-conquer strategies
- Fundamental design concepts and principles
    - Abstraction
    - Program decomposition
    - Encapsulation and information hiding
    - Separation of behavior and implementation

Learning Outcomes:

1. Discuss the importance of algorithms in the problem-solving process. [Familiarity]
2. Discuss how a problem may be solved by multiple algorithms, each with different properties. [Familiarity]
3. Create algorithms for solving simple problems. [Usage]
4. Use a programming language to implement, test, and debug algorithms for solving simple problems. [Usage]
5. Implement, test, and debug simple recursive functions and procedures. [Usage]
6. Determine whether a recursive or iterative solution is most appropriate for a problem. [Assessment]
7. Implement a divide-and-conquer algorithm for solving a problem. [Usage]
8. Apply the techniques of decomposition to break a program into smaller pieces. [Usage]
9. Identify the data components and behaviors of multiple abstract data types. [Usage]
10. Implement a coherent abstract data type, with loose coupling between components and behaviors. [Usage]
11. Identify the relative strengths and weaknesses among multiple designs or implementations for a problem. [Assessment]

SDF/Fundamental Programming Concepts

[10 Core-Tier1 hours]

This knowledge unit builds the foundation for core concepts in the Programming Languages Knowledge Area, most notably in the paradigm-specific units: Object-Oriented Programming, Functional Programming, and Event-Driven & Reactive Programming.

Topics:

- Basic syntax and semantics of a higher-level language
- Variables and primitive data types (e.g., numbers, characters, Booleans)
- Expressions and assignments
- Simple I/O including file I/O
- Conditional and iterative control structures
- Functions and parameter passing
- The concept of recursion

Learning Outcomes:

1. 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. Identify and describe uses of primitive data types. [Familiarity]
3. Write programs that use primitive data types. [Usage]
4. Modify and expand short programs that use standard conditional and iterative control structures and functions. [Usage]
5. 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]
6. Write a program that uses file I/O to provide persistence across multiple executions. [Usage]
7. Choose appropriate conditional and iteration constructs for a given programming task. [Assessment]
8. Describe the concept of recursion and give examples of its use. [Familiarity]
9. Identify the base case and the general case of a recursively-defined problem. [Assessment]

SDF/Fundamental Data Structures

[12 Core-Tier1 hours]

This unit builds the foundation for core concepts in the Algorithms and Complexity Knowledge Area, most notably in the Fundamental Data Structures and Algorithms and Basic Computability and Complexity knowledge units.

Topics:

- Arrays
- Records/structs (heterogeneous aggregates)
- Strings and string processing
- Abstract data types and their implementation
    - Stacks
    - Queues
    - Priority queues
    - Sets
    - Maps
- References and aliasing
- Linked lists
- Strategies for choosing the appropriate data structure

Learning Outcomes:

1. Discuss the appropriate use of built-in data structures. [Familiarity]
2. Describe common applications for each of the following data structures: stack, queue, priority queue, set, and map. [Familiarity]
3. Write programs that use each of the following data structures: arrays, records/structs, strings, linked lists, stacks, queues, sets, and maps. [Usage]
4. Compare alternative implementations of data structures with respect to performance. [Assessment]
5. Describe how references allow for objects to be accessed in multiple ways. [Familiarity]
6. Compare and contrast the costs and benefits of dynamic and static data structure implementations. [Assessment]
7. Choose the appropriate data structure for modeling a given problem. [Assessment]

SDF/Development Methods

[10 Core-Tier1 hours]

This unit builds the foundation for core concepts in the Software Engineering knowledge area, most notably in the Software Processes, Software Design and Software Evolution knowledge units.

Topics:

- Program comprehension
- Program correctness
    - Types of errors (syntax, logic, run-time)
    - The concept of a specification
    - Defensive programming (e.g. secure coding, exception handling)
    - Code reviews
    - Testing fundamentals and test-case generation
    - The role and the use of contracts, including pre- and post-conditions
    - Unit testing
- Simple refactoring
- Modern programming environments
    - Code search
    - Programming using library components and their APIs
- Debugging strategies
- Documentation and program style

Learning Outcomes:

1. Trace the execution of a variety of code segments and write summaries of their computations. [Assessment]
2. Explain why the creation of correct program components is important in the production of high-quality software. [Familiarity]
3. 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]
4. Conduct a personal code review (focused on common coding errors) on a program component using a provided checklist. [Usage]
5. Contribute to a small-team code review focused on component correctness. [Usage]
6. Describe how a contract can be used to specify the behavior of a program component. [Familiarity]
7. Refactor a program by identifying opportunities to apply procedural abstraction. [Usage]
8. Apply a variety of strategies to the testing and debugging of simple programs. [Usage]
9. Construct, execute and debug programs using a modern IDE and associated tools such as unit testing tools and visual debuggers. [Usage]
10. Construct and debug programs using the standard libraries available with a chosen programming language. [Usage]
11. Analyze the extent to which another programmer's code meets documentation and programming style standards. [Assessment]
12. Apply consistent documentation and program style standards that contribute to the readability and maintainability of software. [Usage]