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]

AL/Advanced Computational Complexity

[Elective]

Topics:

- Review of the classes P and NP; introduce P-space and EXP
- Polynomial hierarchy
- NP-completeness (Cook's theorem)
- Classic NP-complete problems
- Reduction Techniques

Learning Outcomes:

1. Define the classes P and NP. (Also appears in AL/Basic Automata, Computability, and Complexity). [Familiarity]
2. Define the P-space class and its relation to the EXP class. [Familiarity]
3. Explain the significance of NP-completeness. (Also appears in AL/Basic Automata, Computability, and Complexity). [Familiarity]
4. Provide examples of classic NP-complete problems. [Familiarity]
5. Prove that a problem is NP-complete by reducing a classic known NP-complete problem to it. [Usage]

AL/Advanced Automata Theory and Computability

[Elective]

Topics:

- Sets and languages
    - Regular languages
    - Review of deterministic finite automata (DFAs)
    - Nondeterministic finite automata (NFAs)
    - Equivalence of DFAs and NFAs
    - Review of regular expressions; their equivalence to finite automata
    - Closure properties
    - Proving languages non-regular, via the pumping lemma or alternative means
- Context-free languages
    - Push-down automata (PDAs)
    - Relationship of PDAs and context-free grammars
    - Properties of context-free languages
- Turing machines, or an equivalent formal model of universal computation
- Nondeterministic Turing machines
- Chomsky hierarchy
- The Church-Turing thesis
- Computability
- Rice's Theorem
- Examples of uncomputable functions
- Implications of uncomputability

Learning Outcomes:

1. Determine a language's place in the Chomsky hierarchy (regular, context-free, recursively enumerable). [Assessment]
2. Convert among equivalently powerful notations for a language, including among DFAs, NFAs, and regular expressions, and between PDAs and CFGs. [Usage]
3. Explain the Church-Turing thesis and its significance. [Familiarity]
4. Explain Rice's Theorem and its significance. [Familiarity]
5. Provide examples of uncomputable functions. [Familiarity]
6. Prove that a problem is uncomputable by reducing a classic known uncomputable problem to it. [Usage]

AL/Advanced Data Structures Algorithms and Analysis

[Elective]

Many programs will want their students to have exposure to more advanced algorithms or methods of analysis. Below is a selection of possible advanced topics that are current and timely but by no means exhaustive.

Topics:

- Balanced trees (e.g., AVL trees, red-black trees, splay trees, treaps)
- Graphs (e.g., topological sort, finding strongly connected components, matching)
- Advanced data structures (e.g., B-trees, Fibonacci heaps)
- String-based data structures and algorithms (e.g., suffix arrays, suffix trees, tries)
- Network flows (e.g., max flow [Ford-Fulkerson algorithm], max flow – min cut, maximum bipartite matching)
- Linear Programming (e.g., duality, simplex method, interior point algorithms)
- Number-theoretic algorithms (e.g., modular arithmetic, primality testing, integer factorization)
- Geometric algorithms (e.g., points, line segments, polygons. [properties, intersections], finding convex hull, spatial decomposition, collision detection, geometric search/proximity)
- Randomized algorithms
- Stochastic algorithms
- Approximation algorithms
- Amortized analysis
- Probabilistic analysis
- Online algorithms and competitive analysis

Learning Outcomes:

1. Understand the mapping of real-world problems to algorithmic solutions (e.g., as graph problems, linear programs, etc.). [Assessment]
2. Select and apply advanced algorithmic techniques (e.g., randomization, approximation) to solve real problems. [Assessment]
3. Select and apply advanced analysis techniques (e.g., amortized, probabilistic, etc.) to algorithms. [Assessment]

Architecture and Organization (AR)

Computing professionals should not regard the computer as just a black box that executes programs by magic. The knowledge area Architecture and Organization builds on Systems Fundamentals (SF) to develop a deeper understanding of the hardware environment upon which all computing is based, and the interface it provides to higher software layers. Students should acquire an understanding and appreciation of a computer system's functional components, their characteristics, performance, and interactions, and, in particular, the challenge of harnessing parallelism to sustain performance improvements now and into the future. Students need to understand computer architecture to develop programs that can achieve high performance through a programmer's awareness of parallelism and latency. In selecting a system to use, students should be able to understand the tradeoff among various components, such as CPU clock speed, cycles per instruction, memory size, and average memory access time.

The learning outcomes specified for these topics correspond primarily to the core and are intended to support programs that elect to require only the minimum 16 hours of computer architecture of their students. For programs that want to teach more than the minimum, the same AR topics can be treated at a more advanced level by implementing a two-course sequence. For programs that want to cover the elective topics, those topics can be introduced within a two-course sequence and/or be treated in a more comprehensive way in a third course.

AR. Architecture and Organization (0 Core-Tier1 hours, 16 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 Hours	Includes Elective
AR/Digital Logic and Digital Systems		**3**	**N**
AR/Machine Level Representation of Data		**3**	**N**
AR/Assembly Level Machine Organization		**6**	**N**
AR/Memory System Organization and Architecture		**3**	**N**
AR/Interfacing and Communication		**1**	**N**
AR/Functional Organization			**Y**
AR/Multiprocessing and Alternative Architectures			**Y**
AR/Performance Enhancements			**Y**

AR/Digital Logic and Digital Systems

[3 Core-Tier2 hours]

Topics:

- Overview and history of computer architecture
- Combinational vs. sequential logic/Field programmable gate arrays as a fundamental combinational + sequential logic building block
- Multiple representations/layers of interpretation (hardware is just another layer)
- Computer-aided design tools that process hardware and architectural representations
- Register transfer notation/Hardware Description Language (Verilog/VHDL)
- Physical constraints (gate delays, fan-in, fan-out, energy/power)

Learning outcomes:

1. Describe the progression of computer technology components from vacuum tubes to VLSI, from mainframe computer architectures to the organization of warehouse-scale computers. [Familiarity]
2. Comprehend the trend of modern computer architectures towards multi-core and that parallelism is inherent in all hardware systems. [Familiarity]
3. Explain the implications of the "power wall" in terms of further processor performance improvements and the drive towards harnessing parallelism. [Familiarity]
4. Articulate that there are many equivalent representations of computer functionality, including logical expressions and gates, and be able to use mathematical expressions to describe the functions of simple combinational and sequential circuits. [Familiarity]
5. Design the basic building blocks of a computer: arithmetic-logic unit (gate-level), registers (gate-level), central processing unit (register transfer-level), memory (register transfer-level). [Usage]
6. Use CAD tools for capture, synthesis, and simulation to evaluate simple building blocks (e.g., arithmetic-logic unit, registers, movement between registers) of a simple computer design. [Usage]

7. Evaluate the functional and timing diagram behavior of a simple processor implemented at the logic circuit level. [Assessment]

AR/Machine Level Representation of Data

[3 Core-Tier2 hours]

Topics:

- Bits, bytes, and words
- Numeric data representation and number bases
- Fixed- and floating-point systems
- Signed and twos-complement representations
- Representation of non-numeric data (character codes, graphical data)
- Representation of records and arrays

Learning outcomes:

1. Explain why everything is data, including instructions, in computers. [Familiarity]
2. Explain the reasons for using alternative formats to represent numerical data. [Familiarity]
3. Describe how negative integers are stored in sign-magnitude and twos-complement representations. [Familiarity]
4. Explain how fixed-length number representations affect accuracy and precision. [Familiarity]
5. Describe the internal representation of non-numeric data, such as characters, strings, records, and arrays. [Familiarity]
6. Convert numerical data from one format to another. [Usage]
7. Write simple programs at the assembly/machine level for string processing and manipulation. [Usage]

AR/Assembly Level Machine Organization

[6 Core-Tier2 hours]

Topics:

- Basic organization of the von Neumann machine
- Control unit; instruction fetch, decode, and execution
- Instruction sets and types (data manipulation, control, I/O)
- Assembly/machine language programming
- Instruction formats
- Addressing modes
- Subroutine call and return mechanisms (cross-reference PL/Language Translation and Execution)
- I/O and interrupts
- Heap vs. Static vs. Stack vs. Code segments
- Shared memory multiprocessors/multicore organization
- Introduction to SIMD vs. MIMD and the Flynn Taxonomy

***Learning outcomes*:**

1. Explain the organization of the classical von Neumann machine and its major functional units. [Familiarity]
2. Describe how an instruction is executed in a classical von Neumann machine, with extensions for threads, multiprocessor synchronization, and SIMD execution. [Familiarity]

3. Describe instruction level parallelism and hazards, and how they are managed in typical processor pipelines. [Familiarity]
4. Summarize how instructions are represented at both the machine level and in the context of a symbolic assembler. [Familiarity]
5. Demonstrate how to map between high-level language patterns into assembly/machine language notations. [Familiarity]
6. Explain different instruction formats, such as addresses per instruction and variable length vs. fixed length formats. [Familiarity]
7. Explain how subroutine calls are handled at the assembly level. [Familiarity]
8. Explain the basic concepts of interrupts and I/O operations. [Familiarity]
9. Write simple assembly language program segments. [Usage]
10. Show how fundamental high-level programming constructs are implemented at the machine-language level. [Usage]

AR/Memory System Organization and Architecture

[3 Core-Tier2 hours]

Cross-reference OS/Memory Management/Virtual Machines

Topics:

- Storage systems and their technology
- Memory hierarchy: importance of temporal and spatial locality
- Main memory organization and operations
- Latency, cycle time, bandwidth, and interleaving
- Cache memories (address mapping, block size, replacement and store policy)
- Multiprocessor cache consistency/Using the memory system for inter-core synchronization/atomic memory operations
- Virtual memory (page table, TLB)
- Fault handling and reliability
- Error coding, data compression, and data integrity (cross-reference SF/Reliability through Redundancy)

Learning outcomes:

1. Identify the main types of memory technology (e.g., SRAM, DRAM, Flash, magnetic disk) and their relative cost and performance. [Familiarity]
2. Explain the effect of memory latency on running time. [Familiarity]
3. Describe how the use of memory hierarchy (cache, virtual memory) is used to reduce the effective memory latency. [Familiarity]
4. Describe the principles of memory management. [Familiarity]
5. Explain the workings of a system with virtual memory management. [Familiarity]
6. Compute Average Memory Access Time under a variety of cache and memory configurations and mixes of instruction and data references. [Usage]

AR/Interfacing and Communication

[1 Core-Tier2 hour]

Cross-reference Operating Systems (OS) Knowledge Area for a discussion of the operating system view of input/output processing and management. The focus here is on the hardware mechanisms for supporting device interfacing and processor-to-processor communications.

Topics:

- I/O fundamentals: handshaking, buffering, programmed I/O, interrupt-driven I/O
- Interrupt structures: vectored and prioritized, interrupt acknowledgment
- External storage, physical organization, and drives
- Buses: bus protocols, arbitration, direct-memory access (DMA)
- Introduction to networks: communications networks as another layer of remote access
- Multimedia support
- RAID architectures

***Learning outcomes*:**

1. Explain how interrupts are used to implement I/O control and data transfers. [Familiarity]
2. Identify various types of buses in a computer system. [Familiarity]
3. Describe data access from a magnetic disk drive. [Familiarity]
4. Compare common network organizations, such as ethernet/bus, ring, switched vs. routed. [Familiarity]
5. Identify the cross-layer interfaces needed for multimedia access and presentation, from image fetch from remote storage, through transport over a communications network, to staging into local memory, and final presentation to a graphical display. [Familiarity]
6. Describe the advantages and limitations of RAID architectures. [Familiarity]

AR/Functional Organization

[Elective]

Note: elective for computer scientist; would be core for computer engineering curriculum.

Topics:

- Implementation of simple datapaths, including instruction pipelining, hazard detection and resolution
- Control unit: hardwired realization vs. microprogrammed realization
- Instruction pipelining
- Introduction to instruction-level parallelism (ILP)

Learning outcomes:

1. Compare alternative implementation of datapaths. [Familiarity]
2. Discuss the concept of control points and the generation of control signals using hardwired or microprogrammed implementations. [Familiarity]
3. Explain basic instruction level parallelism using pipelining and the major hazards that may occur. [Familiarity]
4. Design and implement a complete processor, including datapath and control. [Usage]
5. Determine, for a given processor and memory system implementation, the average cycles per instruction. [Assessment]

AR/Multiprocessing and Alternative Architectures

[Elective]

The view here is on the hardware implementation of SIMD and MIMD architectures.

Cross-reference PD/Parallel Architecture.

Topics:

- Power Law
- Example SIMD and MIMD instruction sets and architectures
- Interconnection networks (hypercube, shuffle-exchange, mesh, crossbar)
- Shared multiprocessor memory systems and memory consistency
- Multiprocessor cache coherence

***Learning outcomes*:**

1. Discuss the concept of parallel processing beyond the classical von Neumann model. [Familiarity]
2. Describe alternative parallel architectures such as SIMD and MIMD. [Familiarity]
3. Explain the concept of interconnection networks and characterize different approaches. [Familiarity]
4. Discuss the special concerns that multiprocessing systems present with respect to memory management and describe how these are addressed. [Familiarity]
5. Describe the differences between memory backplane, processor memory interconnect, and remote memory via networks, their implications for access latency and impact on program performance. [Familiarity]

AR/Performance Enhancements

[Elective]

Topics:

- Superscalar architecture
- Branch prediction, Speculative execution, Out-of-order execution
- Prefetching
- Vector processors and GPUs
- Hardware support for multithreading
- Scalability
- Alternative architectures, such as VLIW/EPIC, and Accelerators and other kinds of Special-Purpose Processors

Learning outcomes:

1. Describe superscalar architectures and their advantages. [Familiarity]
2. Explain the concept of branch prediction and its utility. [Familiarity]
3. Characterize the costs and benefits of prefetching. [Familiarity]
4. Explain speculative execution and identify the conditions that justify it. [Familiarity]
5. Discuss the performance advantages that multithreading offered in an architecture along with the factors that make it difficult to derive maximum benefits from this approach. [Familiarity]
6. Describe the relevance of scalability to performance. [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]

CN/Modeling and Simulation

[Elective]

Topics:

- Purpose of modeling and simulation including optimization; supporting decision making, forecasting, safety considerations; for training and education
- Tradeoffs including performance, accuracy, validity, and complexity
- The simulation process; identification of key characteristics or behaviors, simplifying assumptions; validation of outcomes
- Model building: use of mathematical formulas or equations, graphs, constraints; methodologies and techniques; use of time stepping for dynamic systems

- Formal models and modeling techniques: mathematical descriptions involving simplifying assumptions and avoiding detail. Examples of techniques include:
    - Monte Carlo methods
    - Stochastic processes
    - Queuing theory
    - Petri nets and colored Petri nets
    - Graph structures such as directed graphs, trees, networks
    - Games, game theory, the modeling of things using game theory
    - Linear programming and its extensions
    - Dynamic programming
    - Differential equations: ODE, PDE
    - Non-linear techniques
    - State spaces and transitions
- Assessing and evaluating models and simulations in a variety of contexts; verification and validation of models and simulations
- Important application areas including health care and diagnostics, economics and finance, city and urban planning, science, and engineering
- Software in support of simulation and modeling; packages, languages

Learning Outcomes:

1. Explain and give examples of the benefits of simulation and modeling in a range of important application areas. [Familiarity]
2. Demonstrate the ability to apply the techniques of modeling and simulation to a range of problem areas. [Usage]
3. Explain the constructs and concepts of a particular modeling approach. [Familiarity]
4. Explain the difference between validation and verification of a model; demonstrate the difference with specific examples[1]. [Assessment]
5. Verify and validate the results of a simulation. [Assessment]
6. Evaluate a simulation, highlighting the benefits and the drawbacks. [Assessment]
7. Choose an appropriate modeling approach for a given problem or situation. [Assessment]
8. Compare results from different simulations of the same situation and explain any differences. [Assessment]
9. Infer the behavior of a system from the results of a simulation of the system. [Assessment]
10. Extend or adapt an existing model to a new situation. [Assessment]

[1] *Verification* means that the computations of the model are correct. If we claim to compute total time, for example, the computation actually does that. *Validation* asks whether the model matches the real situation.

CN/Processing

[Elective]

The processing topic area includes numerous topics from other knowledge areas. Specifically, coverage of processing should include a discussion of hardware architectures, including parallel systems, memory hierarchies, and interconnections among processors. These are covered in AR/Interfacing and Communication, AR/Multiprocessing and Alternative Architectures, AR/Performance Enhancements.

Topics:

- Fundamental programming concepts:
  - The concept of an algorithm consisting of a finite number of well-defined steps, each of which completes in a finite amount of time, as does the entire process.
  - Examples of well-known algorithms such as sorting and searching.
  - The concept of analysis as understanding what the problem is really asking, how a problem can be approached using an algorithm, and how information is represented so that a machine can process it.
  - The development or identification of a workflow.
  - The process of converting an algorithm to machine-executable code.
  - Software processes including lifecycle models, requirements, design, implementation, verification and maintenance.
  - Machine representation of data computer arithmetic.
- Numerical methods
  - Algorithms for numerically fitting data (e.g., Newton's method)
  - Architectures for numerical computation, including parallel architectures
- Fundamental properties of parallel and distributed computation:
  - Bandwidth.
  - Latency.
  - Scalability.
  - Granularity.
  - Parallelism including task, data, and event parallelism.
  - Parallel architectures including processor architectures, memory and caching.
  - Parallel programming paradigms including threading, message passing, event driven techniques, parallel software architectures, and MapReduce.
  - Grid computing.
  - The impact of architecture on computational time.
  - Total time to science curve for parallelism: continuum of things.
- Computing costs, e.g., the cost of re-computing a value vs. the cost of storing and lookup.

Learning Outcomes:

1. Explain the characteristics and defining properties of algorithms and how they relate to machine processing. [Familiarity]
2. Analyze simple problem statements to identify relevant information and select appropriate processing to solve the problem. [Assessment]
3. Identify or sketch a workflow for an existing computational process such as the creation of a graph based on experimental data. [Familiarity]
4. Describe the process of converting an algorithm to machine-executable code. [Familiarity]
5. Summarize the phases of software development and compare several common lifecycle models. [Familiarity]
6. Explain how data is represented in a machine. Compare representations of integers to floating point numbers. Describe underflow, overflow, round off, and truncation errors in data representations. [Familiarity]

7. Apply standard numerical algorithms to solve ODEs and PDEs. Use computing systems to solve systems of equations. [Usage]
8. Describe the basic properties of bandwidth, latency, scalability and granularity. [Familiarity]
9. Describe the levels of parallelism including task, data, and event parallelism. [Familiarity]
10. Compare and contrast parallel programming paradigms recognizing the strengths and weaknesses of each. [Assessment]
11. Identify the issues impacting correctness and efficiency of a computation. [Familiarity]
12. Design, code, test and debug programs for a parallel computation. [Usage]

CN/Interactive Visualization

[Elective]

This sub-area is related to modeling and simulation. Most topics are discussed in detail in other knowledge areas in this document. There are many ways to present data and information, including immersion, realism, variable perspectives; haptics and heads-up displays, sonification, and gesture mapping.

Interactive visualization in general requires understanding of human perception (GV/Basics); graphics pipelines, geometric representations and data structures (GV/Fundamental Concepts); 2D and 3D rendering, surface and volume rendering (GV/Rendering, GV/Modeling, and GV/Advanced Rendering); and the use of APIs for developing user interfaces using standard input components such as menus, sliders, and buttons; and standard output components for data display, including charts, graphs, tables, and histograms (HCI/GUI Construction, HCI/GUI Programming).

Topics:

- Principles of data visualization
- Graphing and visualization algorithms
- Image processing techniques
- Scalability concerns

Learning Outcomes:

1. Compare common computer interface mechanisms with respect to ease-of-use, learnability, and cost. [Assessment]
2. Use standard APIs and tools to create visual displays of data, including graphs, charts, tables, and histograms. [Usage]
3. Describe several approaches to using a computer as a means for interacting with and processing data. [Familiarity]
4. Extract useful information from a dataset. [Assessment]
5. Analyze and select visualization techniques for specific problems. [Assessment]
6. Describe issues related to scaling data analysis from small to large data sets. [Familiarity]

CN/Data, Information, and Knowledge

[Elective]

Many topics are discussed in detail in other knowledge areas in this document, specifically Information Management (IM/Information Management Concepts, IM/Database Systems, and IM/Data Modeling), Algorithms and Complexity (AL/Basic Analysis, AL/Fundamental Data Structures and Algorithms), and Software Development Fundamentals (SDF/Fundamental Programming Concepts, SDF/Development Methods).

Topics:

- Content management models, frameworks, systems, design methods (as in IM. Information Management)
- Digital representations of content including numbers, text, images (e.g., raster and vector), video (e.g., QuickTime, MPEG2, MPEG4), audio (e.g., written score, MIDI, sampled digitized sound track) and animations; complex/composite/aggregate objects; FRBR
- Digital content creation/capture and preservation, including digitization, sampling, compression, conversion, transformation/translation, migration/emulation, crawling, harvesting
- Content structure / management, including digital libraries and static/dynamic/stream aspects for:
    - Data: data structures, databases
    - Information: document collections, multimedia pools, hyperbases (hypertext, hypermedia), catalogs, repositories
    - Knowledge: ontologies, triple stores, semantic networks, rules
- Processing and pattern recognition, including indexing, searching (including: queries and query languages; central / federated / P2P), retrieving, clustering, classifying/categorizing, analyzing/mining/extracting, rendering, reporting, handling transactions
- User / society support for presentation and interaction, including browse, search, filter, route, visualize, share, collaborate, rate, annotate, personalize, recommend
- Modeling, design, logical and physical implementation, using relevant systems/software

Learning Outcomes:

1. Identify all of the data, information, and knowledge elements and related organizations, for a computational science application. [Assessment]
2. Describe how to represent data and information for processing. [Familiarity]
3. Describe typical user requirements regarding that data, information, and knowledge. [Familiarity]
4. Select a suitable system or software implementation to manage data, information, and knowledge. [Assessment]
5. List and describe the reports, transactions, and other processing needed for a computational science application. [Familiarity]
6. Compare and contrast database management, information retrieval, and digital library systems with regard to handling typical computational science applications. [Assessment]
7. Design a digital library for some computational science users/societies, with appropriate content and services. [Usage]

CN/Numerical Analysis

[Elective]

Cross-reference AR/Machine Level Representation of Data

Topics:

- Error, stability, convergence, including truncation and round-off
- Function approximation including Taylor's series, interpolation, extrapolation, and regression

- Numerical differentiation and integration (Simpson’s Rule, explicit and implicit methods)
- Differential equations (Euler’s Method, finite differences)

Learning Outcomes:

1. Define error, stability, machine precision concepts and the inexactness of computational approximations. [Familiarity]
2. Implement Taylor series, interpolation, extrapolation, and regression algorithms for approximating functions. [Usage]
3. Implement algorithms for differentiation and integration. [Usage]
4. Implement algorithms for solving differential equations. [Usage]

Discrete Structures (DS)

Discrete structures are foundational material for computer science. By foundational we mean that relatively few computer scientists will be working primarily on discrete structures, but that many other areas of computer science require the ability to work with concepts from discrete structures. Discrete structures include important material from such areas as set theory, logic, graph theory, and probability theory.

The material in discrete structures is pervasive in the areas of data structures and algorithms but appears elsewhere in computer science as well. For example, an ability to create and understand a proof—either a formal symbolic proof or a less formal but still mathematically rigorous argument—is important in virtually every area of computer science, including (to name just a few) formal specification, verification, databases, and cryptography. Graph theory concepts are used in networks, operating systems, and compilers. Set theory concepts are used in software engineering and in databases. Probability theory is used in intelligent systems, networking, and a number of computing applications.

Given that discrete structures serves as a foundation for many other areas in computing, it is worth noting that the boundary between discrete structures and other areas, particularly Algorithms and Complexity, Software Development Fundamentals, Programming Languages, and Intelligent Systems, may not always be crisp. Indeed, different institutions may choose to organize the courses in which they cover this material in very different ways. Some institutions may cover these topics in one or two focused courses with titles like "discrete structures" or "discrete mathematics," whereas others may integrate these topics in courses on programming, algorithms, and/or artificial intelligence. Combinations of these approaches are also prevalent (e.g., covering many of these topics in a single focused introductory course and covering the remaining topics in more advanced topical courses).

DS. Discrete Structures (37 Core-Tier1 hours, 4 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
DS/Sets, Relations, and Functions	**4**		**N**
DS/Basic Logic	**9**		**N**
DS/Proof Techniques	**10**	**1**	**N**
DS/Basics of Counting	**5**		**N**
DS/Graphs and Trees	**3**	**1**	**N**
DS/Discrete Probability	**6**	**2**	**N**

DS/Sets, Relations, and Functions

[4 Core-Tier1 hours]

Topics:

- Sets
    - Venn diagrams
    - Union, intersection, complement
    - Cartesian product
    - Power sets
    - Cardinality of finite sets
- Relations
    - Reflexivity, symmetry, transitivity
    - Equivalence relations, partial orders
- Functions
    - Surjections, injections, bijections
    - Inverses
    - Composition

Learning Outcomes:

1. Explain with examples the basic terminology of functions, relations, and sets. [Familiarity]
2. Perform the operations associated with sets, functions, and relations. [Usage]
3. Relate practical examples to the appropriate set, function, or relation model, and interpret the associated operations and terminology in context. [Assessment]

DS/Basic Logic

[9 Core-Tier1 hours]

Topics:

- Propositional logic (cross-reference: Propositional logic is also reviewed in IS/Knowledge Based Reasoning)
- Logical connectives
- Truth tables
- Normal forms (conjunctive and disjunctive)
- Validity of well-formed formula
- Propositional inference rules (concepts of modus ponens and modus tollens)
- Predicate logic
    - Universal and existential quantification
- Limitations of propositional and predicate logic (e.g., expressiveness issues)

Learning Outcomes:

1. Convert logical statements from informal language to propositional and predicate logic expressions. [Usage]
2. Apply formal methods of symbolic propositional and predicate logic, such as calculating validity of formulae and computing normal forms. [Usage]
3. Use the rules of inference to construct proofs in propositional and predicate logic. [Usage]
4. 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]
5. 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]
6. Describe the strengths and limitations of propositional and predicate logic. [Familiarity]

DS/Proof Techniques

[10 Core-Tier1 hours, 1 Core-Tier2 hour]

Topics:

[Core-Tier1]

- Notions of implication, equivalence, converse, inverse, contrapositive, negation, and contradiction
- The structure of mathematical proofs
- Direct proofs
- Disproving by counterexample
- Proof by contradiction
- Induction over natural numbers
- Structural induction
- Weak and strong induction (i.e., First and Second Principle of Induction)
- Recursive mathematical definitions

[Core-Tier2]

- Well orderings

Learning Outcomes:

[Core-Tier1]

1. Identify the proof technique used in a given proof. [Familiarity]
2. Outline the basic structure of each proof technique (direct proof, proof by contradiction, and induction) described in this unit. [Usage]
3. Apply each of the proof techniques (direct proof, proof by contradiction, and induction) correctly in the construction of a sound argument. [Usage]
4. Determine which type of proof is best for a given problem. [Assessment]
5. Explain the parallels between ideas of mathematical and/or structural induction to recursion and recursively defined structures. [Assessment]
6. Explain the relationship between weak and strong induction and give examples of the appropriate use of each. [Assessment]

[Core-Tier2]

7. State the well-ordering principle and its relationship to mathematical induction. [Familiarity]

DS/Basics of Counting

[5 Core-Tier1 hours]

Topics:

- Counting arguments
    - Set cardinality and counting
    - Sum and product rule
    - Inclusion-exclusion principle
    - Arithmetic and geometric progressions
- The pigeonhole principle
- Permutations and combinations
    - Basic definitions
    - Pascal's identity
    - The binomial theorem
- Solving recurrence relations (cross-reference: AL/Basic Analysis)
    - An example of a simple recurrence relation, such as Fibonacci numbers
    - Other examples, showing a variety of solutions
- Basic modular arithmetic

Learning Outcomes:

1. Apply counting arguments, including sum and product rules, inclusion-exclusion principle and arithmetic/geometric progressions. [Usage]
2. Apply the pigeonhole principle in the context of a formal proof. [Usage]
3. Compute permutations and combinations of a set, and interpret the meaning in the context of the particular application. [Usage]
4. 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]
5. Solve a variety of basic recurrence relations. [Usage]
6. Analyze a problem to determine underlying recurrence relations. [Usage]
7. Perform computations involving modular arithmetic. [Usage]

DS/Graphs and Trees

[3 Core-Tier1 hours, 1 Core-Tier2 hour]

Cross-reference: AL/Fundamental Data Structures and Algorithms, especially with relation to graph traversal strategies.

Topics:

[Core-Tier1]

- Trees
    - Properties
    - Traversal strategies
- Undirected graphs
- Directed graphs
- Weighted graphs

[Core-Tier2]

- Spanning trees/forests
- Graph isomorphism

Learning Outcomes:

[Core-Tier1]

1. 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]
2. Demonstrate different traversal methods for trees and graphs, including pre-, post-, and in-order traversal of trees. [Usage]
3. 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]
4. Show how concepts from graphs and trees appear in data structures, algorithms, proof techniques (structural induction), and counting. [Usage]

[Core-Tier2]

5. Explain how to construct a spanning tree of a graph. [Usage]
6. Determine if two graphs are isomorphic. [Usage]

DS/Discrete Probability

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

Topics:

[Core-Tier1]

- Finite probability space, events
- Axioms of probability and probability measures
- Conditional probability, Bayes' theorem
- Independence
- Integer random variables (Bernoulli, binomial)
- Expectation, including Linearity of Expectation

[Core-Tier2]

- Variance
- Conditional Independence

Learning Outcomes:

[Core-Tier1]

1. Calculate probabilities of events and expectations of random variables for elementary problems such as games of chance. [Usage]
2. Differentiate between dependent and independent events. [Usage]
3. Identify a case of the binomial distribution and compute a probability using that distribution. [Usage]
4. Apply Bayes theorem to determine conditional probabilities in a problem. [Usage]
5. Apply the tools of probability to solve problems such as the average case analysis of algorithms or analyzing hashing. [Usage]

[Core-Tier2]

6. Compute the variance for a given probability distribution. [Usage]
7. Explain how events that are independent can be conditionally dependent (and vice-versa). Identify real-world examples of such cases. [Usage]

Graphics and Visualization (GV)

Computer graphics is the term commonly used to describe the computer generation and manipulation of images. It is the science of enabling visual communication through computation. Its uses include cartoons, film special effects, video games, medical imaging, engineering, as well as scientific, information, and knowledge visualization. Traditionally, graphics at the undergraduate level has focused on rendering, linear algebra, and phenomenological approaches. More recently, the focus has begun to include physics, numerical integration, scalability, and special-purpose hardware. In order for students to become adept at the use and generation of computer graphics, many implementation-specific issues must be addressed, such as file formats, hardware interfaces, and application program interfaces. These issues change rapidly, and the description that follows attempts to avoid being overly prescriptive about them. The area encompassed by Graphics and Visualization is divided into several interrelated fields:

- Fundamentals: Computer graphics depends on an understanding of how humans use vision to perceive information and how information can be rendered on a display device. Every computer scientist should have some understanding of where and how graphics can be appropriately applied as well as the fundamental processes involved in display rendering.
- Modeling: Information to be displayed must be encoded in computer memory in some form, often in the form of a mathematical specification of shape and form.
- Rendering: Rendering is the process of displaying the information contained in a model.
- Animation: Animation is the rendering in a manner that makes images appear to move and the synthesis or acquisition of the time variations of models.
- Visualization: The field of visualization seeks to determine and present underlying correlated structures and relationships in data sets from a wide variety of application areas. The prime objective of the presentation should be to communicate the information in a dataset so as to enhance understanding
- Computational Geometry: Computational Geometry is the study of algorithms that are stated in terms of geometry.

Graphics and Visualization is related to machine vision and image processing, which are found in the Intelligent Systems (IS) KA, and algorithms such as computational geometry, which are found in the Algorithms and Complexity (AL) KA. Topics in virtual reality are found in the Human-Computer Interaction (HCI) KA.

This description assumes students are familiar with fundamental concepts of data representation, abstraction, and program implementation.

GV. Graphics and Visualization (2 Core-Tier1 hours, 1 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
GV/Fundamental Concepts	**2**	**1**	**Y**
GV/Basic Rendering			**Y**
GV/Geometric Modeling			**Y**
GV/Advanced Rendering			**Y**
GV/Computer Animation			**Y**
GV/Visualization			**Y**

GV/Fundamental Concepts

[2 Core-Tier1 and 1 Core-Tier2 hours]

For nearly every computer scientist and software developer, an understanding of how humans interact with machines is essential. While these topics may be covered in a standard undergraduate graphics course, they may also be covered in introductory computer science and programming courses. Part of our motivation for including immediate and retained modes is that these modes are analogous to polling vs. event driven programming. This is a fundamental question in computer science: Is there a button object, or is there just the display of a button on the screen? Note that most of the outcomes in this section are at the knowledge level, and many of these topics are revisited in greater depth in later sections.

Topics:

[Core-Tier1]

- Media applications including user interfaces, audio and video editing, game engines, cad, visualization, virtual reality
- Digitization of analog data, resolution, and the limits of human perception, e.g., pixels for visual display, dots for laser printers, and samples for audio (HCI/Foundations)
- Use of standard APIs for the construction of UIs and display of standard media formats (see HCI/GUI construction)
- Standard media formats, including lossless and lossy formats

[Core-Tier2]

- Additive and subtractive color models (CMYK and RGB) and why these provide a range of colors
- Tradeoffs between storing data and re-computing data as embodied by vector and raster representations of images
- Animation as a sequence of still images

[Elective]

- Double buffering

Learning Outcomes:

[Core-Tier1]

1. Identify common uses of digital presentation to humans (e.g., computer graphics, sound). [Familiarity]
2. Explain in general terms how analog signals can be reasonably represented by discrete samples, for example, how images can be represented by pixels. [Familiarity]
3. Explain how the limits of human perception affect choices about the digital representation of analog signals. [Familiarity]
4. Construct a simple user interface using a standard API. [Usage]
5. Describe the differences between lossy and lossless image compression techniques, for example as reflected in common graphics image file formats such as JPG, PNG, MP3, MP4, and GIF. [Familiarity]

[Core-Tier2]

6. Describe color models and their use in graphics display devices. [Familiarity]
7. Describe the tradeoffs between storing information vs. storing enough information to reproduce the information, as in the difference between vector and raster rendering. [Familiarity]

[Elective]

8. Describe the basic process of producing continuous motion from a sequence of discrete frames (sometimes called "flicker fusion"). [Familiarity]
9. Describe how double-buffering can remove flicker from animation. [Familiarity]

GV/Basic Rendering

[Elective]

This section describes basic rendering and fundamental graphics techniques that nearly every undergraduate course in graphics will cover and that are essential for further study in graphics. Sampling and anti-aliasing are related to the effect of digitization and appear in other areas of computing, for example, in audio sampling.

Topics:

- Rendering in nature, e.g., the emission and scattering of light and its relation to numerical integration
- Forward and backward rendering (i.e., ray-casting and rasterization)
- Polygonal representation
- Basic radiometry, similar triangles, and projection model
- Affine and coordinate system transformations
- Ray tracing
- Visibility and occlusion, including solutions to this problem such as depth buffering, Painter's algorithm, and ray tracing
- The forward and backward rendering equation
- Simple triangle rasterization
- Rendering with a shader-based API
- Texture mapping, including minification and magnification (e.g., trilinear MIP-mapping)
- Application of spatial data structures to rendering
- Sampling and anti-aliasing
- Scene graphs and the graphics pipeline

Learning Outcomes:

1. Discuss the light transport problem and its relation to numerical integration i.e., light is emitted, scatters around the scene, and is measured by the eye. [Familiarity]
2. Describe the basic graphics pipeline and how forward and backward rendering factor in this. [Familiarity]
3. Create a program to display 3D models of simple graphics images. [Usage]
4. Derive linear perspective from similar triangles by converting points (x, y, z) to points (x/z, y/z, 1). [Usage]
5. Obtain 2-dimensional and 3-dimensional points by applying affine transformations. [Usage]
6. Apply 3-dimensional coordinate system and the changes required to extend 2D transformation operations to handle transformations in 3D. [Usage]
7. Contrast forward and backward rendering. [Assessment]
8. Explain the concept and applications of texture mapping, sampling, and anti-aliasing. [Familiarity]
9. Explain the ray tracing/rasterization duality for the visibility problem. [Familiarity]
10. Implement simple procedures that perform transformation and clipping operations on simple 2-dimensional images. [Usage]
11. Implement a simple real-time renderer using a rasterization API (e.g., OpenGL) using vertex buffers and shaders. [Usage]
12. Compare and contrast the different rendering techniques. [Assessment]
13. Compute space requirements based on resolution and color coding. [Assessment]
14. Compute time requirements based on refresh rates, rasterization techniques. [Assessment]

GV/Geometric Modeling

[Elective]

Topics:

- Basic geometric operations such as intersection calculation and proximity tests
- Volumes, voxels, and point-based representations
- Parametric polynomial curves and surfaces
- Implicit representation of curves and surfaces
- Approximation techniques such as polynomial curves, Bezier curves, spline curves and surfaces, and non-uniform rational basis (NURB) spines, and level set method
- Surface representation techniques including tessellation, mesh representation, mesh fairing, and mesh generation techniques such as Delaunay triangulation, marching cubes
- Spatial subdivision techniques
- Procedural models such as fractals, generative modeling, and L-systems
- Graftals, cross referenced with programming languages (grammars to generated pictures)
- Elastically deformable and freeform deformable models
- Subdivision surfaces
- Multiresolution modeling
- Reconstruction
- Constructive Solid Geometry (CSG) representation

Learning Outcomes:

1. Represent curves and surfaces using both implicit and parametric forms. [Usage]
2. Create simple polyhedral models by surface tessellation. [Usage]
3. Generate a mesh representation from an implicit surface. [Usage]
4. Generate a fractal model or terrain using a procedural method. [Usage]
5. Generate a mesh from data points acquired with a laser scanner. [Usage]
6. Construct CSG models from simple primitives, such as cubes and quadric surfaces. [Usage]
7. Contrast modeling approaches with respect to space and time complexity and quality of image. [Assessment]

GV/Advanced Rendering

[Elective]

Topics:

- Solutions and approximations to the rendering equation, for example:
    - Distribution ray tracing and path tracing
    - Photon mapping
    - Bidirectional path tracing
    - Reyes (micropolygon) rendering
    - Metropolis light transport
- Time (motion blur), lens position (focus), and continuous frequency (color) and their impact on rendering
- Shadow mapping
- Occlusion culling
- Bidirectional Scattering Distribution function (BSDF) theory and microfacets
- Subsurface scattering
- Area light sources
- Hierarchical depth buffering
- The Light Field, image-based rendering

- Non-photorealistic rendering
- GPU architecture
- Human visual systems including adaptation to light, sensitivity to noise, and flicker fusion

Learning Outcomes:

1. Demonstrate how an algorithm estimates a solution to the rendering equation. [Assessment]
2. Prove the properties of a rendering algorithm, e.g., complete, consistent, and unbiased. [Assessment]
3. Analyze the bandwidth and computation demands of a simple algorithm. [Assessment]
4. Implement a non-trivial shading algorithm (e.g., toon shading, cascaded shadow maps) under a rasterization API. [Usage]
5. Discuss how a particular artistic technique might be implemented in a renderer. [Familiarity]
6. Explain how to recognize the graphics techniques used to create a particular image. [Familiarity]
7. Implement any of the specified graphics techniques using a primitive graphics system at the individual pixel level. [Usage]
8. Implement a ray tracer for scenes using a simple (e.g., Phong model) BRDF plus reflection and refraction. [Usage]

GV/Computer Animation

[Elective]

Topics:

- Forward and inverse kinematics
- Collision detection and response
- Procedural animation using noise, rules (boids/crowds), and particle systems
- Skinning algorithms
- Physics based motions including rigid body dynamics, physical particle systems, mass-spring networks for cloth and flesh and hair
- Key-frame animation
- Splines
- Data structures for rotations, such as quaternions
- Camera animation
- Motion capture

Learning Outcomes:

1. Compute the location and orientation of model parts using a forward kinematic approach. [Usage]
2. Compute the orientation of articulated parts of a model from a location and orientation using an inverse kinematic approach. [Usage]
3. Describe the tradeoffs in different representations of rotations. [Assessment]
4. Implement the spline interpolation method for producing in-between positions and orientations. [Usage]
5. Implement algorithms for physical modeling of particle dynamics using simple Newtonian mechanics, for example Witkin & Kass, snakes and worms, symplectic Euler, Stormer/Verlet, or midpoint Euler methods. [Usage]
6. Discuss the basic ideas behind some methods for fluid dynamics for modeling ballistic trajectories, for example for splashes, dust, fire, or smoke. [Familiarity]
7. Use common animation software to construct simple organic forms using metaball and skeleton. [Usage]

GV/Visualization

[Elective]

Visualization has strong ties to the Human-Computer Interaction (HCI) knowledge area as well as Computational Science (CN). Readers should refer to the HCI and CN KAs for additional topics related to user population and interface evaluations.

Topics:

- Visualization of 2D/3D scalar fields: color mapping, isosurfaces
- Direct volume data rendering: ray-casting, transfer functions, segmentation
- Visualization of:
  - Vector fields and flow data
  - Time-varying data
  - High-dimensional data: dimension reduction, parallel coordinates,
  - Non-spatial data: multi-variate, tree/graph structured, text
- Perceptual and cognitive foundations that drive visual abstractions
- Visualization design
- Evaluation of visualization methods
- Applications of visualization

Learning Outcomes:

1. Describe the basic algorithms for scalar and vector visualization. [Familiarity]
2. Describe the tradeoffs of visualization algorithms in terms of accuracy and performance. [Assessment]
3. Propose a suitable visualization design for a particular combination of data characteristics and application tasks. [Assessment]
4. Analyze the effectiveness of a given visualization for a particular task. [Assessment]
5. Design a process to evaluate the utility of a visualization algorithm or system. [Assessment]
6. Recognize a variety of applications of visualization including representations of scientific, medical, and mathematical data; flow visualization; and spatial analysis. [Familiarity]

Human-Computer Interaction (HCI)

Human-computer interaction (HCI) is concerned with designing interactions between human activities and the computational systems that support them, and with constructing interfaces to afford those interactions.

Interaction between users and computational artefacts occurs at an interface that includes both software and hardware. Thus interface design impacts the software life-cycle in that it should occur early; the design and implementation of core functionality can influence the user interface – for better or worse.

Because it deals with people as well as computational systems, as a knowledge area HCI demands the consideration of cultural, social, organizational, cognitive and perceptual issues. Consequently it draws on a variety of disciplinary traditions, including psychology, ergonomics, computer science, graphic and product design, anthropology and engineering.

HCI: Human Computer Interaction (4 Core-Tier1 hours, 4 Core-Tier2 hours)

	Core-Tier1 hours	**Core-Tier2 hours**	**Includes Electives**
HCI/Foundations	**4**		**N**
HCI/Designing Interaction		**4**	**N**
HCI/Programming Interactive Systems			**Y**
HCI/User-Centered Design & Testing			**Y**
HCI/New Interactive Technologies			**Y**
HCI/Collaboration & Communication			**Y**
HCI/Statistical Methods for HCI			**Y**
HCI/Human Factors & Security			**Y**
HCI/Design-Oriented HCI			**Y**
HCI/Mixed, Augmented and Virtual Reality			**Y**

HCI/Foundations

[4 Core-Tier1 hours]

Motivation: For end-users, the interface *is* the system. So design in this domain must be interaction-focused and human-centered. Students need a different repertoire of techniques to address this than is provided elsewhere in the curriculum.

Topics:

- Contexts for HCI (anything with a user interface, e.g., webpage, business applications, mobile applications, and games)
- Processes for user-centered development, e.g., early focus on users, empirical testing, iterative design
- Different measures for evaluation, e.g., utility, efficiency, learnability, user satisfaction
- Usability heuristics and the principles of usability testing
- Physical capabilities that inform interaction design, e.g., color perception, ergonomics
- Cognitive models that inform interaction design, e.g., attention, perception and recognition, movement, and memory; gulfs of expectation and execution
- Social models that inform interaction design, e.g., culture, communication, networks and organizations
- Principles of good design and good designers; engineering tradeoffs
- Accessibility, e.g., interfaces for differently-abled populations (e.g., blind, motion-impaired)
- Interfaces for differently-aged population groups (e.g., children, 80+)

Learning Outcomes:

1. Discuss why human-centered software development is important. [Familiarity]
2. Summarize the basic precepts of psychological and social interaction. [Familiarity]
3. Develop and use a conceptual vocabulary for analyzing human interaction with software: affordance, conceptual model, feedback, and so forth. [Usage]
4. Define a user-centered design process that explicitly takes account of the fact that the user is not like the developer or their acquaintances. [Usage]
5. Create and conduct a simple usability test for an existing software application. [Assessment]

HCI/Designing Interaction

[4 Core-Tier2 hours]

Motivation: CS students need a minimal set of well-established methods and tools to bring to interface construction.

Topics:

- Principles of graphical user interfaces (GUIs)
- Elements of visual design (layout, color, fonts, labeling)
- Task analysis, including qualitative aspects of generating task analytic models
- Low-fidelity (paper) prototyping
- Quantitative evaluation techniques, e.g., keystroke-level evaluation
- Help and documentation
- Handling human/system failure
- User interface standards

Learning Outcomes:

1. For an identified user group, undertake and document an analysis of their needs. [Assessment]
2. Create a simple application, together with help and documentation, that supports a graphical user interface. [Usage]
3. Conduct a quantitative evaluation and discuss/report the results. [Usage]
4. Discuss at least one national or international user interface design standard. [Familiarity]

HCI/Programming Interactive Systems

[Elective]

Motivation: To take a user-experience-centered view of software development and then cover approaches and technologies to make that happen.

Topics:

- Software Architecture Patterns, e.g., Model-View controller; command objects, online, offline (cross reference PL/Event Driven and Reactive Programming, where MVC is used in the context of event-driven programming)
- Interaction Design Patterns: visual hierarchy, navigational distance
- Event management and user interaction
- Geometry management (cross-reference GV/Geometric Modelling)
- Choosing interaction styles and interaction techniques
- Presenting information: navigation, representation, manipulation
- Interface animation techniques (e.g., scene graphs)
- Widget classes and libraries
- Modern GUI libraries (e.g. iOS, Android, JavaFX) GUI builders and UI programming environments (cross-reference PBD/Mobile Platforms)
- Declarative Interface Specification: Stylesheets and DOMs
- Data-driven applications (database-backed web pages)
- Cross-platform design
- Design for resource-constrained devices (e.g. small, mobile devices)

Learning Outcomes:

1. Explain the importance of Model-View controller to interface programming. [Familiarity]
2. Create an application with a modern graphical user interface. [Usage]
3. Identify commonalities and differences in UIs across different platforms. [Familiarity]
4. Explain and use GUI programming concepts: event handling, constraint-based layout management, etc. [Familiarity]

HCI/User-Centered Design and Testing

[Elective]

Motivation: An exploration of techniques to ensure that end-users are fully considered at all stages of the design process, from inception to implementation.

Topics:

- Approaches to, and characteristics of, the design process
- Functionality and usability requirements (cross-reference to SE/Requirements Engineering)
- Techniques for gathering requirements, e.g., interviews, surveys, ethnographic and contextual enquiry
- Techniques and tools for the analysis and presentation of requirements, e.g., reports, personas
- Prototyping techniques and tools, e.g., sketching, storyboards, low-fidelity prototyping, wireframes
- Evaluation without users, using both qualitative and quantitative techniques, e.g., walkthroughs, GOMS, expert-based analysis, heuristics, guidelines, and standards
- Evaluation with users, e.g., observation, think-aloud, interview, survey, experiment
- Challenges to effective evaluation, e.g., sampling, generalization
- Reporting the results of evaluations
- Internationalization, designing for users from other cultures, cross-cultural

Learning Outcomes:

1. Explain how user-centered design complements other software process models. [Familiarity]
2. Use lo-fi (low fidelity) prototyping techniques to gather, and report, user responses. [Usage]
3. Choose appropriate methods to support the development of a specific UI. [Assessment]
4. Use a variety of techniques to evaluate a given UI. [Assessment]
5. Compare the constraints and benefits of different evaluative methods. [Assessment]

HCI/New Interactive Technologies

[Elective]

Motivation: As technologies evolve, new interaction styles are made possible. This knowledge unit should be considered extensible, to track emergent technology.

Topics:

- Choosing interaction styles and interaction techniques
- Representing information to users: navigation, representation, manipulation
- Approaches to design, implementation and evaluation of non-mouse interaction
    - Touch and multi-touch interfaces
    - Shared, embodied, and large interfaces
    - New input modalities (such as sensor and location data)
    - New Windows, e.g., iPhone, Android
    - Speech recognition and natural language processing (cross reference IS/Natural Language Processing)
    - Wearable and tangible interfaces
    - Persuasive interaction and emotion
    - Ubiquitous and context-aware interaction technologies (Ubicomp)
    - Bayesian inference (e.g. predictive text, guided pointing)
    - Ambient/peripheral display and interaction

Learning Outcomes:

1. Describe when non-mouse interfaces are appropriate. [Familiarity]
2. Understand the interaction possibilities beyond mouse-and-pointer interfaces. [Familiarity]
3. Discuss the advantages (and disadvantages) of non-mouse interfaces. [Assessment]

HCI/Collaboration and Communication

[Elective]

Motivation: Computer interfaces not only support users in achieving their individual goals but also in their interaction with others, whether that is task-focused (work or gaming) or task-unfocused (social networking).

Topics:

- Asynchronous group communication, e.g., e-mail, forums, social networks
- Synchronous group communication, e.g., chat rooms, conferencing, online games
- Social media, social computing, and social network analysis
- Online collaboration, 'smart' spaces, and social coordination aspects of workflow technologies
- Online communities
- Software characters and intelligent agents, virtual worlds and avatars (cross-reference IS/Agents)
- Social psychology

Learning Outcomes:

1. Describe the difference between synchronous and asynchronous communication. [Familiarity]
2. Compare the HCI issues in individual interaction with group interaction. [Assessment]
3. Discuss several issues of social concern raised by collaborative software. [Familiarity]
4. Discuss the HCI issues in software that embodies human intention. [Familiarity]

HCI/Statistical Methods for HCI

[Elective]

Motivation: Much HCI work depends on the proper use, understanding and application of statistics. This knowledge is often held by students who join the field from psychology, but less common in students with a CS background.

Topics:

- t-tests
- ANOVA
- Randomization (non-parametric) testing, within vs. between-subjects design
- Calculating effect size
- Exploratory data analysis
- Presenting statistical data
- Combining qualitative and quantitative results

Learning Outcomes:

1. Explain basic statistical concepts and their areas of application. [Familiarity]
2. Extract and articulate the statistical arguments used in papers that quantitatively report user studies. [Usage]
3. Design a user study that will yield quantitative results. [Usage]
4. Conduct and report on a study that utilizes both qualitative and quantitative evaluation. [Usage]

HCI/Human Factors and Security

[Elective]

Motivation: Effective interface design requires basic knowledge of security psychology. Many attacks do not have a technological basis, but exploit human propensities and vulnerabilities. “Only amateurs attack machines; professionals target people” (Bruce Schneier, https://www.schneier.com/blog/archives/2013/03/phishing_has_go.h.)

Topics:

- Applied psychology and security policies
- Security economics
- Regulatory environments – responsibility, liability and self-determination
- Organizational vulnerabilities and threats
- Usability design and security
- Pretext, impersonation and fraud, e.g., phishing and spear phishing (cross-reference IAS/Threats and Attacks)
- Trust, privacy and deception
- Biometric authentication (camera, voice)
- Identity management

Learning Outcomes:

1. Explain the concepts of phishing and spear phishing, and how to recognize them. [Familiarity]
2. Describe the issues of trust in interface design with an example of a high and low trust system. [Assessment]
3. Design a user interface for a security mechanism. [Assessment]
4. Explain the concept of identity management and its importance. [Familiarity]
5. Analyze a security policy and/or procedures to show where they consider, or fail to consider, human factors. [Usage]

HCI/Design-Oriented HCI

[Elective]

Motivation: Some curricula will want to emphasize an understanding of the norms and values of HCI work itself as emerging from, and deployed within specific historical, disciplinary and cultural contexts.

Topics:

- Intellectual styles and perspectives to technology and its interfaces
- Consideration of HCI as a design discipline
  - Sketching
  - Participatory design
- Critically reflective HCI
  - Critical technical practice
  - Technologies for political activism
  - Philosophy of user experience
  - Ethnography and ethnomethodology
- Indicative domains of application
  - Sustainability
  - Arts-informed computing

Learning Outcomes:

1. Explain what is meant by "HCI is a design-oriented discipline". [Familiarity]
2. Detail the processes of design appropriate to specific design orientations. [Familiarity]
3. Apply a variety of design methods to a given problem. [Usage]

HCI/Mixed, Augmented and Virtual Reality

[Elective]

Motivation: A detailed consideration of the interface components required for the creation and development of immersive environments, especially games.

Topics:

- Output
  - Sound
  - Stereoscopic display
  - Force feedback simulation, haptic devices
- User input
  - Viewer and object tracking
  - Pose and gesture recognition
  - Accelerometers
  - Fiducial markers
  - User interface issues
- Physical modelling and rendering
  - Physical simulation: collision detection & response, animation
  - Visibility computation
  - Time-critical rendering, multiple levels of details (LOD)
- System architectures

    - Game engines
    - Mobile augmented reality
    - Flight simulators
    - CAVEs
    - Medical imaging
- Networking
    - p2p, client-server, dead reckoning, encryption, synchronization
    - Distributed collaboration

Learning Outcomes:

1. Describe the optical model realized by a computer graphics system to synthesize stereoscopic view. [Familiarity]
2. Describe the principles of different viewer tracking technologies. [Familiarity]
3. Describe the differences between geometry- and image-based virtual reality. [Familiarity]
4. Describe the issues of user action synchronization and data consistency in a networked environment. [Familiarity]
5. Determine the basic requirements on interface, hardware, and software configurations of a VR system for a specified application. [Usage]
6. Describe several possible uses for games engines, including their potential and their limitations. [Familiarity]

Information Assurance and Security (IAS)

In CS2013, the Information Assurance and Security KA is added to the Body of Knowledge in recognition of the world's reliance on information technology and its critical role in computer science education. Information assurance and security as a domain is the set of controls and processes both technical and policy intended to protect and defend information and information systems by ensuring their confidentiality, integrity, and availability, and by providing for authentication and non-repudiation. The concept of assurance also carries an attestation that current and past processes and data are valid. Both assurance and security concepts are needed to ensure a complete perspective. Information assurance and security education, then, includes all efforts to prepare a workforce with the needed knowledge, skills, and abilities to protect our information systems and attest to the assurance of the past and current state of processes and data. The importance of security concepts and topics has emerged as a core requirement in the Computer Science discipline, much like the importance of performance concepts has been for many years.

The Information Assurance and Security KA is unique among the set of KAs presented here given the manner in which the topics are pervasive throughout other Knowledge Areas. The topics germane to only IAS are presented in the IAS section; other topics are noted and cross-referenced in the IAS KA. In the IAS KA the many topics are represented with only 9 hours of Core-Tier1 and Tier2 coverage. This is balanced with the level of mastery primarily at the familiarity level and the more indepth coverage distributed in the referenced KAs where they are applied. The broad application of the IAS KA concepts (63.5 hours) across all other KAs provides the depth of coverage and mastery for an undergraduate computer science student.

The IAS KA is shown in two groups: (1) concepts where the depth is unique to Information Assurance and Security and (2) IAS topics that are integrated into other KAs that reflect naturally implied or specified topics with a strong role in security concepts and topics. For completeness, the total distribution of hours is summarized in the table below.

IAS. Information Assurance and Security “Core” and Distributed

	Core-Tier1 hours	Core-Tier2 hours	Elective Topics
IAS	3	6	Y
IAS distributed in other KA’s	32	31.5	Y

IAS. Information Assurance and Security (3 Core-Tier1 hours, 6 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
IAS/Foundational Concepts in Security	1		N
IAS/Principles of Secure Design	1	1	N
IAS/Defensive Programming	1	1	Y
IAS/Threats and Attacks		1	N
IAS/Network Security		2	Y
IAS/Cryptography		1	N
IAS/Web Security			Y
IAS/Platform Security			Y
IAS/Security Policy and Governance			Y
IAS/Digital Forensics			Y
IAS/Secure Software Engineering			Y

The following table shows the distribution of hours throughout all other KA’s in CS2013 where security is appropriately addressed either as fundamental to the KU topics (for example, OS/Security or Protection or SE/Software Construction) or as a supportive use case for the topic (for example, HCI/Foundations or NC/Routing and Forwarding or SP/Intellectual Property). The hours represent the set of hours in that KA/KU where the topics are particularly relevant to Information Assurance and Security.

IAS. Information Assurance and Security (distributed) (32 Core-Tier1 hours, 31.5 Core-Tier2 hours)

Knowledge Area and Topic	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
AR/Assembly Level Machine Organization		1	
AR/Memory System Organization and Architecture		0.5	
AR/Multiprocessing and Alternative Architectures			Y
HCI/Foundations	1		
HCI/Human Factors and Security			Y
IM/Information Management Concepts	0.5	0.5	
IM/Transaction Processing			Y
IM/Distributed Databases			Y
IS/Reasoning Under Uncertainty			Y
NC/Introduction	1		
NC/Networked Applications	0.5		
NC/Reliable Data Delivery		1.5	
NC/Routing and Forwarding		1	
NC/Local Area Networks		1	
NC/Resource Allocation		0.5	
NC/Mobility		1	
OS/Overview of OS	2		
OS/OS Principles	1		

OS/Concurrency		**1.5**	
OS/Scheduling and Dispatch		**2**	
OS/Memory Management		**2**	
OS/Security and Protection		**2**	
OS/Virtual Machines			**Y**
OS/Device Management			**Y**
OS/File Systems			**Y**
OS/Real Time and Embedded Systems			**Y**
OS/Fault Tolerance			**Y**
OS/System Performance Evaluation			**Y**
PBD/Web Platforms			**Y**
PBD/Mobile Platforms			**Y**
PBD/Industrial Platforms			**Y**
PD/Parallelism Fundamentals	**1**		
PD/Parallel Decomposition	**0.5**		
PD/Communication and Coordination	**1**	**1**	**Y**
PD/Parallel Architecture	**0.5**		**Y**
PD/Distributed Systems			**Y**
PD/Cloud Computing			**Y**
PL/Object-Oriented Programming	**1**	**3**	
PL/Functional Programming	**1**		

PL/Basic Type Systems	**0.5**	**2**	
PL/Language Translation and Execution		**1**	
PL/Runtime Systems			**Y**
PL/Static Analysis			**Y**
PL/Concurrency and Parallelism			**Y**
PL/Type Systems			**Y**
SDF/Fundamental Programming Concepts	**1**		
SDF/Development Methods	**8**		
SE/Software Processes	**1**		
SE/Software Project Management		**1**	**Y**
SE/Tools and Environments		**1**	
SE/Software Construction		**2**	**Y**
SE/Software Verification and Validation		**1**	**Y**
SE/Software Evolution		**1.5**	
SE/Software Reliability	**1**		
SF/Cross-Layer Communications	**3**		
SF/Parallelism	**1**		
SF/Resource Allocation and Scheduling	**0.5**		
SF/Virtualization and Isolation		**1**	
SF/Reliability through Redundancy		**2**	

SP/Social Context	**0.5**		
SP/Analytical Tools	**1**		
SP/Professional Ethics	**1**	**0.5**	
SP/Intellectual Property	**2**		**Y**
SP/Privacy and Civil Liberties	**0.5**		
SP/Security Policies, Laws and Computer Crimes			**Y**

IAS/Foundational Concepts in Security

[1 Core-Tier1 hour]

Topics:

- CIA (Confidentiality, Integrity, Availability)
- Concepts of risk, threats, vulnerabilities, and attack vectors (cros- reference SE/Software Project Management/Risk)
- Authentication and authorization, access control (mandatory vs. discretionary)
- Concept of trust and trustworthiness
- Ethics (responsible disclosure). (cross-reference SP/Professional Ethics/Accountability, responsibility and liability)

Learning outcomes:

1. Analyze the tradeoffs of balancing key security properties (Confidentiality, Integrity, and Availability). [Usage]
2. Describe the concepts of risk, threats, vulnerabilities and attack vectors (including the fact that there is no such thing as perfect security). [Familiarity]
3. Explain the concepts of authentication, authorization, access control. [Familiarity]
4. Explain the concept of trust and trustworthiness. [Familiarity]
5. Describe important ethical issues to consider in computer security, including ethical issues associated with fixing or not fixing vulnerabilities and disclosing or not disclosing vulnerabilities. [Familiarity]

IAS/Principles of Secure Design

[1 Core-Tier1 hour, 1 Core-Tier2 hour]

Topics:

. [Core-Tier1]

- Least privilege and isolation (cross-reference OS/Security and Protection/Policy/mechanism separation and SF/Virtualization and Isolation/Rationale for protection and predictable performance and PL/Language Translation and Execution/Memory management)
- Fail-safe defaults (cross-reference SE/Software Construction/ Coding practices: techniques, idioms/patterns, mechanisms for building quality programs and SDF/Development Methods/Programming correctness)
- Open design (cross-reference SE/Software Evolution/ Software development in the context of large, pre-existing code bases)
- End-to-end security (cross-reference SF/Reliability through Redundancy/ How errors increase the longer the distance between the communicating entities; the end-to-end principle)
- Defense in depth (e.g., defensive programming, layered defense)
- Security by design (cross-reference SE/Software Design/System design principles)
- Tensions between security and other design goals

[Core-Tier2]

- Complete mediation
- Use of vetted security components
- Economy of mechanism (reducing trusted computing base, minimize attack surface) (cross-reference SE/Software Design/System design principles and SE/Software Construction/Development context: “green field” vs. existing code base)
- Usable security (cross-reference HCI/Foundations/Cognitive models that inform interaction design)
- Security composability
- Prevention, detection, and deterrence (cross-reference SF/Reliability through Redundancy/Distinction between bugs and faults and NC/Reliable Data Delivery/Error control and NC/Reliable Data Delivery/Flow control)

Learning outcomes:

[Core-Tier1]

1. Describe the principle of least privilege and isolation as applied to system design. [Familiarity]
2. Summarize the principle of fail-safe and deny-by-default. [Familiarity]
3. Discuss the implications of relying on open design or the secrecy of design for security. [Familiarity]
4. Explain the goals of end-to-end data security. [Familiarity]
5. Discuss the benefits of having multiple layers of defenses. [Familiarity]
6. For each stage in the lifecycle of a product, describe what security considerations should be evaluated. [Familiarity]
7. Describe the cost and tradeoffs associated with designing security into a product. [Familiarity]

[Core-Tier2]

8. Describe the concept of mediation and the principle of complete mediation. [Familiarity]
9. Describe standard components for security operations, and explain the benefits of their use instead of re-inventing fundamentals operations. [Familiarity]
10. Explain the concept of trusted computing including trusted computing base and attack surface and the principle of minimizing trusted computing base. [Familiarity]

11. Discuss the importance of usability in security mechanism design. [Familiarity]
12. Describe security issues that arise at boundaries between multiple components. [Familiarity]
13. Identify the different roles of prevention mechanisms and detection/deterrence mechanisms. [Familiarity]

IAS/Defensive Programming

[1 Core-Tier1 hour, 1 Core-Tier2 hour]

Topics in defensive programming are generally not thought about in isolation, but applied to other topics particularly in SDF, SE and PD Knowledge Areas.

Topics:

[Core-Tier1]

- Input validation and data sanitization (cross-reference SDF/Development Methods/Program Correctness)
- Choice of programming language and type-safe languages
- Examples of input validation and data sanitization errors (cross-reference SDF/Development Methods/Program Correctness and SE/Software Construction/Coding Practices)
  - Buffer overflows
  - Integer errors
  - SQL injection
  - XSS vulnerability
- Race conditions (cross-reference SF/Parallelism/Parallel programming and PD/Parallel Architecture/Shared vs. distributed memory and PD/Communication and Coordination/Shared Memory and PD/Parallelism Fundamentals/Programming errors not found in sequential programming)
- Correct handling of exceptions and unexpected behaviors (cross-reference SDF/Development Methods/program correctness)

[Core-Tier2]

- Correct usage of third-party components (cross-reference SDF/Development Methods/program correctness and Operating System Principles/Concepts of application program interfaces (APIs)
- Effectively deploying security updates (cross-reference OS/Security and Protection/Security methods and devices)

[Electives]

- Information flow control
- Correctly generating randomness for security purposes
- Mechanisms for detecting and mitigating input and data sanitization errors
- Fuzzing
- Static analysis and dynamic analysis
- Program verification
- Operating system support (e.g., address space randomization, canaries)
- Hardware support (e.g., DEP, TPM)

Learning outcomes:

[Core-Tier1]

1. Explain why input validation and data sanitization is necessary in the face of adversarial control of the input channel. [Familiarity]
2. Explain why you might choose to develop a program in a type-safe language like Java, in contrast to an unsafe programming language like C/C++. [Familiarity]
3. Classify common input validation errors, and write correct input validation code. [Usage]
4. Demonstrate using a high-level programming language how to prevent a race condition from occurring and how to handle an exception. [Usage]
5. Demonstrate the identification and graceful handling of error conditions. [Usage]

[Core-Tier2]

6. Explain the risks with misusing interfaces with third-party code and how to correctly use third-party code. [Familiarity]
7. Discuss the need to update software to fix security vulnerabilities and the lifecycle management of the fix. [Familiarity]

[Elective]

8. List examples of direct and indirect information flows. [Familiarity]
9. Explain the role of random numbers in security, beyond just cryptography (e.g. password generation, randomized algorithms to avoid algorithmic denial of service attacks). [Familiarity]
10. Explain the different types of mechanisms for detecting and mitigating data sanitization errors. [Familiarity]
11. Demonstrate how programs are tested for input handling errors. [Usage]
12. Use static and dynamic tools to identify programming faults. [Usage]
13. Describe how memory architecture is used to protect runtime attacks. [Familiarity]

IAS/Threats and Attacks

[1 Core-Tier2 hour]

Topics:

[Core-Tier2]

- Attacker goals, capabilities, and motivations (such as underground economy, digital espionage, cyberwarfare, insider threats, hacktivism, advanced persistent threats)
- Examples of malware (e.g., viruses, worms, spyware, botnets, Trojan horses or rootkits)
- Denial of Service (DoS) and Distributed Denial of Service (DDoS)
- Social engineering (e.g., phishing) (cross-reference SP/Social Context/Social implications of computing in a networked world and HCI/Designing Interaction/Handling human/system failure)

[Elective]

- Attacks on privacy and anonymity (cross-reference HCI/Foundations/Social models that inform interaction design: culture, communication, networks and organizations (cross-reference SP/Privacy and Civil Liberties/technology-based solutions for privacy protection)
- Malware/unwanted communication such as covert channels and steganography

Learning outcomes:

[Core-Tier2]

1. Describe likely attacker types against a particular system. [Familiarity]
2. Discuss the limitations of malware countermeasures (e.g., signature-based detection, behavioral detection). [Familiarity]
3. Identify instances of social engineering attacks and Denial of Service attacks. [Familiarity]
4. Discuss how Denial of Service attacks can be identified and mitigated. [Familiarity]

[Elective]

5. Describe risks to privacy and anonymity in commonly used applications. [Familiarity]
6. Discuss the concepts of covert channels and other data leakage procedures. [Familiarity]

IAS/Network Security

[2 Core-Tier2 hours]

Discussion of network security relies on previous understanding on fundamental concepts of networking, including protocols, such as TCP/IP, and network architecture/organization (cross-reference NC/Network Communication).

Topics:

[Core-Tier2]

- Network specific threats and attack types (e.g., denial of service, spoofing, sniffing and traffic redirection, man-in-the-middle, message integrity attacks, routing attacks, and traffic analysis)
- Use of cryptography for data and network security
- Architectures for secure networks (e.g., secure channels, secure routing protocols, secure DNS, VPNs, anonymous communication protocols, isolation)
- Defense mechanisms and countermeasures (e.g., network monitoring, intrusion detection, firewalls, spoofing and DoS protection, honeypots, tracebacks)

. [Elective]

- Security for wireless, cellular networks (cross-reference NC/Mobility/Principles of cellular networks; cross-reference NC/Mobility/802.11)
- Other non-wired networks (e.g., ad hoc, sensor, and vehicular networks)
- Censorship resistance
- Operational network security management (e.g., configure network access control)

Learning outcomes:

[Core-Tier2]

1. Describe the different categories of network threats and attacks. [Familiarity]
2. Describe the architecture for public and private key cryptography and how public key infrastructure (PKI) supports network security. [Familiarity]
3. Describe virtues and limitations of security technologies at each layer of the network stack. [Familiarity]
4. Identify the appropriate defense mechanism(s) and its limitations given a network threat. [Familiarity]

[Elective]

5. Discuss security properties and limitations of other non-wired networks. [Familiarity]
6. Identify the additional threats faced by non-wired networks. [Familiarity]

7. Describe threats that can and cannot be protected against using secure communication channels. [Familiarity]
8. Summarize defenses against network censorship. [Familiarity]
9. Diagram a network for security. [Familiarity]

IAS/Cryptography

[1 Core-Tier2 hour]

Topics:

[Core-Tier2]

- Basic Cryptography Terminology covering notions pertaining to the different (communication) partners, secure/unsecure channel, attackers and their capabilities, encryption, decryption, keys and their characteristics, signatures
- Cipher types (e.g., Caesar cipher, affine cipher) together with typical attack methods such as frequency analysis
- Public Key Infrastructure support for digital signature and encryption and its challenges

[Elective]

- Mathematical Preliminaries essential for cryptography, including topics in linear algebra, number theory, probability theory, and statistics
- Cryptographic primitives:
    - pseudo-random generators and stream ciphers
    - block ciphers (pseudo-random permutations), e.g., AES
    - pseudo-random functions
    - hash functions, e.g., SHA2, collision resistance
    - message authentication codes
    - key derivations functions
- Symmetric key cryptography
    - Perfect secrecy and the one time pad
    - Modes of operation for semantic security and authenticated encryption (e.g., encrypt-then-MAC, OCB, GCM)
    - Message integrity (e.g., CMAC, HMAC)
- Public key cryptography:
    - Trapdoor permutation, e.g., RSA
    - Public key encryption, e.g., RSA encryption, EI Gamal encryption
    - Digital signatures
    - Public-key infrastructure (PKI) and certificates
    - Hardness assumptions, e.g., Diffie-Hellman, integer factoring
- Authenticated key exchange protocols, e.g., TLS
- Cryptographic protocols: challenge-response authentication, zero-knowledge protocols, commitment, oblivious transfer, secure 2-party or multi-party computation, secret sharing, and applications
- Motivate concepts using real-world applications, e.g., electronic cash, secure channels between clients and servers, secure electronic mail, entity authentication, device pairing, voting systems.
- Security definitions and attacks on cryptographic primitives:
    - Goals: indistinguishability, unforgeability, collision-resistance
    - Attacker capabilities: chosen-message attack (for signatures), birthday attacks, side channel attacks, fault injection attacks.
- Cryptographic standards and references implementations
- Quantum cryptography

Learning outcomes:

[Core-Tier2]

1. Describe the purpose of cryptography and list ways it is used in data communications. [Familiarity]
2. Define the following terms: cipher, cryptanalysis, cryptographic algorithm, and cryptology, and describe the two basic methods (ciphers) for transforming plain text in cipher text. [Familiarity]
3. Discuss the importance of prime numbers in cryptography and explain their use in cryptographic algorithms. [Familiarity]
4. Explain how public key infrastructure supports digital signing and encryption and discuss the limitations/vulnerabilities. [Familiarity]

[Elective]

5. Use cryptographic primitives and describe their basic properties. [Usage]
6. Illustrate how to measure entropy and how to generate cryptographic randomness. [Usage]
7. Use public-key primitives and their applications. [Usage]
8. Explain how key exchange protocols work and how they fail. [Familiarity]
9. Discuss cryptographic protocols and their properties. [Familiarity]
10. Describe real-world applications of cryptographic primitives and protocols. [Familiarity]
11. Summarize security definitions related to attacks on cryptographic primitives, including attacker capabilities and goals.[Familiarity]
12. Apply appropriate known cryptographic techniques for a given scenario. [Usage]
13. Appreciate the dangers of inventing one's own cryptographic methods. [Familiarity]
14. Describe quantum cryptography and the impact of quantum computing on cryptographic algorithms. [Familiarity]

IAS/Web Security

[Elective]

Topics:

- Web security model
  - Browser security model including same-origin policy
  - Client-server trust boundaries, e.g., cannot rely on secure execution in the client
- Session management, authentication
  - Single sign-on
  - HTTPS and certificates
- Application vulnerabilities and defenses
  - SQL injection
  - XSS
  - CSRF
- Client-side security
  - Cookies security policy
  - HTTP security extensions, e.g. HSTS
  - Plugins, extensions, and web apps
  - Web user tracking
- Server-side security tools, e.g. Web Application Firewalls (WAFs) and fuzzers

Learning outcomes:

1. Describe the browser security model including same-origin policy and threat models in web security. [Familiarity]

2. Discuss the concept of web sessions, secure communication channels such as TLS and importance of secure certificates, authentication including single sign-on such as OAuth and SAML. [Familiarity]
3. Describe common types of vulnerabilities and attacks in web applications, and defenses against them. [Familiarity]
4. Use client-side security capabilities in an application. [Usage]

IAS/Platform Security

[Elective]

Topics:

- Code integrity and code signing
- Secure boot, measured boot, and root of trust
- Attestation
- TPM and secure co-processors
- Security threats from peripherals, e.g., DMA, IOMMU
- Physical attacks: hardware Trojans, memory probes, cold boot attacks
- Security of embedded devices, e.g., medical devices, cars
- Trusted path

Learning outcomes:

1. Explain the concept of code integrity and code signing and the scope it applies to. [Familiarity]
2. Discuss the concept of root of trust and the process of secure boot and secure loading. [Familiarity]
3. Describe the mechanism of remote attestation of system integrity. [Familiarity]
4. Summarize the goals and key primitives of TPM. [Familiarity]
5. Identify the threats of plugging peripherals into a device. [Familiarity]
6. Identify physical attacks and countermeasures. [Familiarity]
7. Identify attacks on non-PC hardware platforms. [Familiarity]
8. Discuss the concept and importance of trusted path. [Familiarity]

IAS/Security Policy and Governance

[Elective]

See general cross-referencing with the SP/Security Policies, Laws and Computer Crimes.

Topics:

- Privacy policy (cross-reference SP/Social Context/Social implications of computing in a networked world; cross-reference SP/Professional Ethics/Accountability, responsibility and liability; cross-reference SP/Privacy and Civil Liberties/Legal foundations of privacy protection)
- Inference controls/statistical disclosure limitation
- Backup policy, password refresh policy
- Breach disclosure policy
- Data collection and retention policies
- Supply chain policy
- Cloud security tradeoffs

Learning outcomes:

1. Describe the concept of privacy including personally private information, potential violations of privacy due to security mechanisms, and describe how privacy protection mechanisms run in conflict with security mechanisms. [Familiarity]
2. Describe how an attacker can infer a secret by interacting with a database. [Familiarity]
3. Explain how to set a data backup policy or password refresh policy. [Familiarity]
4. Discuss how to set a breach disclosure policy. [Familiarity]
5. Describe the consequences of data retention policies. [Familiarity]
6. Identify the risks of relying on outsourced manufacturing. [Familiarity]
7. Identify the risks and benefits of outsourcing to the cloud. [Familiarity]

IAS/Digital Forensics

[Elective]

Topics:

- Basic Principles and methodologies for digital forensics
- Design systems with forensic needs in mind
- Rules of Evidence – general concepts and differences between jurisdictions and Chain of Custody
- Search and Seizure of evidence: legal and procedural requirements
- Digital Evidence methods and standards
- Techniques and standards for Preservation of Data
- Legal and Reporting Issues including working as an expert witness
- OS/File System Forensics
- Application Forensics
- Web Forensics
- Network Forensics
- Mobile Device Forensics
- Computer/network/system attacks
- Attack detection and investigation
- Anti-forensics

Learning outcomes:

1. Describe what a digital investigation is, the sources of digital evidence, and the limitations of forensics. [Familiarity]
2. Explain how to design software to support forensics. [Familiarity]
3. Describe the legal requirements for use of seized data. [Familiarity]
4. Describe the process of evidence seizure from the time when the requirement was identified to the disposition of the data. [Familiarity]
5. Describe how data collection is accomplished and the proper storage of the original and forensics copy. [Familiarity]
6. Conduct data collection on a hard drive. [Usage]
7. Describe a person's responsibility and liability while testifying as a forensics examiner. [Familiarity]
8. Recover data based on a given search term from an imaged system. [Usage]
9. Reconstruct application history from application artifacts. [Usage]
10. Reconstruct web browsing history from web artifacts. [Usage]
11. Capture and interpret network traffic. [Usage]
12. Discuss the challenges associated with mobile device forensics. [Familiarity]
13. Inspect a system (network, computer, or application) for the presence of malware or malicious activity. [Usage]
14. Apply forensics tools to investigate security breaches. [Usage]
15. Identify anti-forensic methods. [Familiarity]

IAS/Secure Software Engineering

[Elective]

Fundamentals of secure coding practices covered in other knowledge areas, including SDF and SE. For example, see SE/Software Construction; Software Verification and Validation.

Topics:

- Building security into the software development lifecycle (cross-reference SE/Software Processes)
- Secure design principles and patterns
- Secure software specifications and requirements
- Secure software development practices (cross-reference SE/Software Construction)
- Secure testing - the process of testing that security requirements are met (including static and dynamic analysis).
- Software quality assurance and benchmarking measurements

Learning outcomes:

1. Describe the requirements for integrating security into the software development lifecycle. [Familiarity]
2. Apply the concepts of the Design Principles for Protection Mechanisms, the Principles for Software Security [2], and the Principles for Secure Design [1] on a software development project. [Usage]
3. Develop specifications for a software development effort that fully specify functional requirements and identifies the expected execution paths. [Usage]
4. Describe software development best practices for minimizing vulnerabilities in programming code. [Familiarity]
5. Conduct a security verification and assessment (static and dynamic) of a software application. [Usage]

References

[1] Gasser, M. *Building a Secure Computer System*, Van Nostrand Reinhold, 1988.

[2] Viega, J. and McGraw, G. *Building Secure Software: How to Avoid Security Problems the Right Way*, Addison-Wesley, 2002.

Information Management (IM)

Information Management is primarily concerned with the capture, digitization, representation, organization, transformation, and presentation of information; algorithms for efficient and effective access and updating of stored information; data modeling and abstraction; and physical file storage techniques. The student needs to be able to develop conceptual and physical data models, determine which IM methods and techniques are appropriate for a given problem, and be able to select and implement an appropriate IM solution that addresses relevant design concerns including scalability, accessibility and usability.

We also note that IM is related to fundamental information security concepts that are described in the Information Assurance and Security (IAS) topic area, IAS/Fundamental Concepts.

IM. Information Management (1 Core-Tier1 hour; 9 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
IM/Information Management Concepts	1	2	N
IM/Database Systems		3	Y
IM/Data Modeling		4	N
IM/Indexing			Y
IM/Relational Databases			Y
IM/Query Languages			Y
IM/Transaction Processing			Y
IM/Distributed Databases			Y
IM/Physical Database Design			Y
IM/Data Mining			Y
IM/Information Storage And Retrieval			Y
IM/MultiMedia Systems			Y

IM. Information Management-related topics (distributed) (1 Core-Tier1 hour, 2 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
IAS/Fundamental Concepts*	1	2	N

* See Information Assurance and Security Knowledge Area for a description of this topic area.

IM/Information Management Concepts

[1 Core-Tier1 hour; 2 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Information systems as socio-technical systems
- Basic information storage and retrieval (IS&R) concepts
- Information capture and representation
- Supporting human needs: searching, retrieving, linking, browsing, navigating

[Core-Tier2]

- Information management applications
- Declarative and navigational queries, use of links
- Analysis and indexing
- Quality issues: reliability, scalability, efficiency, and effectiveness

Learning Outcomes:

[Core-Tier1]

1. Describe how humans gain access to information and data to support their needs. [Familiarity]
2. Describe the advantages and disadvantages of central organizational control over data. [Assessment]
3. Identify the careers/roles associated with information management (e.g., database administrator, data modeler, application developer, end-user). [Familiarity]
4. Compare and contrast information with data and knowledge. [Assessment]
5. Demonstrate uses of explicitly stored metadata/schema associated with data. [Usage]
6. Identify issues of data persistence for an organization. [Familiarity]

[Core-Tier2]

7. Critique an information application with regard to satisfying user information needs. [Assessment]
8. Explain uses of declarative queries. [Familiarity]
9. Give a declarative version for a navigational query. [Familiarity]
10. Describe several technical solutions to the problems related to information privacy, integrity, security, and preservation. [Familiarity]
11. Explain measures of efficiency (throughput, response time) and effectiveness (recall, precision). [Familiarity]
12. Describe approaches to scale up information systems. [Familiarity]
13. Identify vulnerabilities and failure scenarios in common forms of information systems. [Usage]

IM/Database Systems

[3 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Approaches to and evolution of database systems
- Components of database systems

- Design of core DBMS functions (e.g., query mechanisms, transaction management, buffer management, access methods)
- Database architecture and data independence
- Use of a declarative query language
- Systems supporting structured and/or stream content

[Elective]

- Approaches for managing large volumes of data (e.g., noSQL database systems, use of MapReduce).

Learning Outcomes:

[Core-Tier2]

1. Explain the characteristics that distinguish the database approach from the approach of programming with data files. [Familiarity]
2. Describe the most common designs for core database system components including the query optimizer, query executor, storage manager, access methods, and transaction processor. [Familiarity]
3. Cite the basic goals, functions, and models of database systems. [Familiarity]
4. Describe the components of a database system and give examples of their use. [Familiarity]
5. Identify major DBMS functions and describe their role in a database system. [Familiarity]
6. Explain the concept of data independence and its importance in a database system. [Familiarity]
7. Use a declarative query language to elicit information from a database. [Usage]
8. Describe facilities that datatbases provide supporting structures and/or stream (sequence) data, e.g., text. [Familiarity]

[Elective]

9. Describe major approaches to storing and processing large volumes of data. [Familiarity]

IM/Data Modeling

[4 Core-Tier2 hours]

Topics:

- Data modeling
- Conceptual models (e.g., entity-relationship, UML diagrams)
- Spreadsheet models
- Relational data models
- Object-oriented models (cross-reference PL/Object-Oriented Programming)
- Semi-structured data model (expressed using DTD or XML Schema, for example)

Learning Outcomes:

1. Compare and contrast appropriate data models, including internal structures, for different types of data. [Assessment]
2. Describe concepts in modeling notation (e.g., Entity-Relation Diagrams or UML) and how they would be used. [Familiarity]
3. Define the fundamental terminology used in the relational data model. [Familiarity]
4. Describe the basic principles of the relational data model. [Familiarity]
5. Apply the modeling concepts and notation of the relational data model. [Usage]
6. Describe the main concepts of the OO model such as object identity, type constructors, encapsulation, inheritance, polymorphism, and versioning. [Familiarity]

7. Describe the differences between relational and semi-structured data models. [Assessment]
8. Give a semi-structured equivalent (e.g., in DTD or XML Schema) for a given relational schema. [Usage]

IM/Indexing

[Elective]

Topics:

- The impact of indices on query performance
- The basic structure of an index
- Keeping a buffer of data in memory
- Creating indexes with SQL
- Indexing text
- Indexing the web (e.g., web crawling)

Learning Outcomes:

1. Generate an index file for a collection of resources. [Usage]
2. Explain the role of an inverted index in locating a document in a collection. [Familiarity]
3. Explain how stemming and stop words affect indexing. [Familiarity]
4. Identify appropriate indices for given relational schema and query set. [Usage]
5. Estimate time to retrieve information, when indices are used compared to when they are not used. [Usage]
6. Describe key challenges in web crawling, e.g., detecting duplicate documents, determining the crawling frontier. [Familiarity]

IM/Relational Databases

[Elective]

Topics:

- Mapping conceptual schema to a relational schema
- Entity and referential integrity
- Relational algebra and relational calculus
- Relational Database design
- Functional dependency
- Decomposition of a schema; lossless-join and dependency-preservation properties of a decomposition
- Candidate keys, superkeys, and closure of a set of attributes
- Normal forms (BCNF)
- Multi-valued dependency (4NF)
- Join dependency (PJNF, 5NF)
- Representation theory

Learning Outcomes:

1. Prepare a relational schema from a conceptual model developed using the entity- relationship model. [Usage]
2. Explain and demonstrate the concepts of entity integrity constraint and referential integrity constraint (including definition of the concept of a foreign key). [Usage]

3. Demonstrate use of the relational algebra operations from mathematical set theory (union, intersection, difference, and Cartesian product) and the relational algebra operations developed specifically for relational databases (select (restrict), project, join, and division). [Usage]
4. Write queries in the relational algebra. [Usage]
5. Write queries in the tuple relational calculus. [Usage]
6. Determine the functional dependency between two or more attributes that are a subset of a relation. [Assessment]
7. Connect constraints expressed as primary key and foreign key, with functional dependencies. [Usage]
8. Compute the closure of a set of attributes under given functional dependencies. [Usage]
9. Determine whether a set of attributes form a superkey and/or candidate key for a relation with given functional dependencies. [Assessment]
10. Evaluate a proposed decomposition, to say whether it has lossless-join and dependency-preservation. [Assessment]
11. Describe the properties of BCNF, PJNF, 5NF. [Familiarity]
12. Explain the impact of normalization on the efficiency of database operations especially query optimization. [Familiarity]
13. Describe what is a multi-valued dependency and what type of constraints it specifies. [Familiarity]

IM/Query Languages

[Elective]

Topics:

- Overview of database languages
- SQL (data definition, query formulation, update sublanguage, constraints, integrity)
- Selections
- Projections
- Select-project-join
- Aggregates and group-by
- Subqueries
- QBE and 4th-generation environments
- Different ways to invoke non-procedural queries in conventional languages
- Introduction to other major query languages (e.g., XPATH, SPARQL)
- Stored procedures

Learning Outcomes:

1. Create a relational database schema in SQL that incorporates key, entity integrity, and referential integrity constraints. [Usage]
2. Use SQL to create tables and retrieve (SELECT) information from a database. [Usage]
3. Evaluate a set of query processing strategies and select the optimal strategy. [Assessment]
4. Create a non-procedural query by filling in templates of relations to construct an example of the desired query result. [Usage]
5. Embed object-oriented queries into a stand-alone language such as C++ or Java (e.g., SELECT Col.Method() FROM Object). [Usage]
6. Write a stored procedure that deals with parameters and has some control flow, to provide a given functionality. [Usage]

IM/Transaction Processing

[Elective]

Topics:

- Transactions
- Failure and recovery
- Concurrency control
- Interaction of transaction management with storage, especially buffering

Learning Outcomes:

1. Create a transaction by embedding SQL into an application program. [Usage]
2. Explain the concept of implicit commits. [Familiarity]
3. Describe the issues specific to efficient transaction execution. [Familiarity]
4. Explain when and why rollback is needed and how logging assures proper rollback. [Assessment]
5. Explain the effect of different isolation levels on the concurrency control mechanisms. [Assessment]
6. Choose the proper isolation level for implementing a specified transaction protocol. [Assessment]
7. Identify appropriate transaction boundaries in application programs. [Assessment]

IM/Distributed Databases

[Elective]

Topics:

- Distributed DBMS
  - Distributed data storage
  - Distributed query processing
  - Distributed transaction model
  - Homogeneous and heterogeneous solutions
  - Client-server distributed databases (cross-reference SF/Computational Paradigms)
- Parallel DBMS
  - Parallel DBMS architectures: shared memory, shared disk, shared nothing;
  - Speedup and scale-up, e.g., use of the MapReduce processing model (cross-reference CN/Processing, PD/Parallel Decomposition)
  - Data replication and weak consistency models

Learning Outcomes:

1. Explain the techniques used for data fragmentation, replication, and allocation during the distributed database design process. [Familiarity]
2. Evaluate simple strategies for executing a distributed query to select the strategy that minimizes the amount of data transfer. [Assessment]
3. Explain how the two-phase commit protocol is used to deal with committing a transaction that accesses databases stored on multiple nodes. [Familiarity]
4. Describe distributed concurrency control based on the distinguished copy techniques and the voting method. [Familiarity]
5. Describe the three levels of software in the client-server model. [Familiarity]

IM/Physical Database Design

[Elective]

Topics:

- Storage and file structure
- Indexed files
- Hashed files
- Signature files
- B-trees
- Files with dense index
- Files with variable length records
- Database efficiency and tuning

Learning Outcomes:

1. Explain the concepts of records, record types, and files, as well as the different techniques for placing file records on disk. [Familiarity]
2. Give examples of the application of primary, secondary, and clustering indexes. [Familiarity]
3. Distinguish between a non-dense index and a dense index. [Assessment]
4. Implement dynamic multilevel indexes using B-trees. [Usage]
5. Explain the theory and application of internal and external hashing techniques. [Familiarity]
6. Use hashing to facilitate dynamic file expansion. [Usage]
7. Describe the relationships among hashing, compression, and efficient database searches. [Familiarity]
8. Evaluate costs and benefits of various hashing schemes. [Assessment]
9. Explain how physical database design affects database transaction efficiency. [Familiarity]

IM/Data Mining

[Elective]

Topics:

- Uses of data mining
- Data mining algorithms
- Associative and sequential patterns
- Data clustering
- Market basket analysis
- Data cleaning
- Data visualization (cross-reference GV/Visualization and CN/Interactive Visualization)

Learning Outcomes:

1. Compare and contrast different uses of data mining as evidenced in both research and application. [Assessment]
2. Explain the value of finding associations in market basket data. [Familiarity]
3. Characterize the kinds of patterns that can be discovered by association rule mining. [Assessment]
4. Describe how to extend a relational system to find patterns using association rules. [Familiarity]
5. Evaluate different methodologies for effective application of data mining. [Assessment]
6. Identify and characterize sources of noise, redundancy, and outliers in presented data. [Assessment]
7. Identify mechanisms (on-line aggregation, anytime behavior, interactive visualization) to close the loop in the data mining process. [Familiarity]
8. Describe why the various close-the-loop processes improve the effectiveness of data mining. [Familiarity]

IM/Information Storage and Retrieval

[Elective]

Topics:

- Documents, electronic publishing, markup, and markup languages
- Tries, inverted files, PAT trees, signature files, indexing
- Morphological analysis, stemming, phrases, stop lists
- Term frequency distributions, uncertainty, fuzziness, weighting
- Vector space, probabilistic, logical, and advanced models
- Information needs, relevance, evaluation, effectiveness
- Thesauri, ontologies, classification and categorization, metadata
- Bibliographic information, bibliometrics, citations
- Routing and (community) filtering
- Multimedia search, information seeking behavior, user modeling, feedback
- Information summarization and visualization
- Faceted search (e.g., using citations, keywords, classification schemes)
- Digital libraries
- Digitization, storage, interchange, digital objects, composites, and packages
- Metadata and cataloging
- Naming, repositories, archives
- Archiving and preservation, integrity
- Spaces (conceptual, geographical, 2/3D, VR)
- Architectures (agents, buses, wrappers/mediators), interoperability
- Services (searching, linking, browsing, and so forth)
- Intellectual property rights management, privacy, and protection (watermarking)

Learning Outcomes:

1. Explain basic information storage and retrieval concepts. [Familiarity]
2. Describe what issues are specific to efficient information retrieval. [Familiarity]
3. Give applications of alternative search strategies and explain why the particular search strategy is appropriate for the application. [Assessment]
4. Design and implement a small to medium size information storage and retrieval system, or digital library. [Usage]
5. Describe some of the technical solutions to the problems related to archiving and preserving information in a digital library. [Familiarity]

IM/Multimedia Systems

[Elective]

Topics:

- Input and output devices, device drivers, control signals and protocols, DSPs
- Standards (e.g., audio, graphics, video)
- Applications, media editors, authoring systems, and authoring
- Streams/structures, capture/represent/transform, spaces/domains, compression/coding
- Content-based analysis, indexing, and retrieval of audio, images, animation, and video

- Presentation, rendering, synchronization, multi-modal integration/interfaces
- Real-time delivery, quality of service (including performance), capacity planning, audio/video conferencing, video-on-demand

Learning Outcomes:

1. Describe the media and supporting devices commonly associated with multimedia information and systems. [Familiarity]
2. Demonstrate the use of content-based information analysis in a multimedia information system. [Usage]
3. Critique multimedia presentations in terms of their appropriate use of audio, video, graphics, color, and other information presentation concepts. [Assessment]
4. Implement a multimedia application using an authoring system. [Usage]
5. For each of several media or multimedia standards, describe in non-technical language what the standard calls for, and explain how aspects of human perception might be sensitive to the limitations of that standard. [Familiarity]
6. Describe the characteristics of a computer system (including identification of support tools and appropriate standards) that has to host the implementation of one of a range of possible multimedia applications. [Familiarity]

Intelligent Systems (IS)

Artificial intelligence (AI) is the study of solutions for problems that are difficult or impractical to solve with traditional methods. It is used pervasively in support of everyday applications such as email, word-processing and search, as well as in the design and analysis of autonomous agents that perceive their environment and interact rationally with the environment.
The solutions rely on a broad set of general and specialized knowledge representation schemes, problem solving mechanisms and learning techniques. They deal with sensing (e.g., speech recognition, natural language understanding, computer vision), problem-solving (e.g., search, planning), and acting (e.g., robotics) and the architectures needed to support them (e.g., agents, multi-agents). The study of Artificial Intelligence prepares the student to determine when an AI approach is appropriate for a given problem, identify the appropriate representation and reasoning mechanism, and implement and evaluate it.

IS. Intelligent Systems (10 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
IS/Fundamental Issues		1	Y
IS/Basic Search Strategies		4	N
IS/Basic Knowledge Representation and Reasoning		3	N
IS/Basic Machine Learning		2	N
IS/Advanced Search			Y
IS/Advanced Representation and Reasoning			Y
IS/Reasoning Under Uncertainty			Y
IS/Agents			Y
IS/Natural Language Processing			Y
IS/Advanced Machine Learning			Y
IS/Robotics			Y
IS/Perception and Computer Vision			Y

IS/Fundamental Issues

[1 Core-Tier2 hours]

Topics:

- Overview of AI problems, examples of successful recent AI applications
- What is intelligent behavior?
    - The Turing test
    - Rational versus non-rational reasoning
- Problem characteristics
    - Fully versus partially observable
    - Single versus multi-agent
    - Deterministic versus stochastic
    - Static versus dynamic
    - Discrete versus continuous
- Nature of agents
    - Autonomous versus semi-autonomous
    - Reflexive, goal-based, and utility-based
    - The importance of perception and environmental interactions
- Philosophical and ethical issues. [elective]

Learning Outcomes:

1. Describe Turing test and the “Chinese Room” thought experiment. [Familiarity]
2. Differentiate between the concepts of optimal reasoning/behavior and human-like reasoning/behavior. [Familiarity]
3. Determine the characteristics of a given problem that an intelligent system must solve. [Assessment]

IS/Basic Search Strategies

[4 Core-Tier2 hours]

Cross-reference AL/Basic Analysis, AL/Algorithmic Strategies, AL/Fundamental Data Structures and Algorithms

Topics:

- Problem spaces (states, goals and operators), problem solving by search
- Factored representation (factoring state into variables)
- Uninformed search (breadth-first, depth-first, depth-first with iterative deepening)
- Heuristics and informed search (hill-climbing, generic best-first, A*)
- Space and time efficiency of search
- Two-player games (introduction to minimax search)
- Constraint satisfaction (backtracking and local search methods)

Learning Outcomes:

1. Formulate an efficient problem space for a problem expressed in natural language (e.g., English) in terms of initial and goal states, and operators. [Usage]
2. Describe the role of heuristics and describe the trade-offs among completeness, optimality, time complexity, and space complexity. [Familiarity]
3. Describe the problem of combinatorial explosion of search space and its consequences. [Familiarity]
4. Select and implement an appropriate uninformed search algorithm for a problem, and characterize its time and space complexities. [Usage]
5. Select and implement an appropriate informed search algorithm for a problem by designing the necessary heuristic evaluation function. [Usage]
6. Evaluate whether a heuristic for a given problem is admissible/can guarantee optimal solution. [Assessment]
7. Formulate a problem specified in natural language (e.g., English) as a constraint satisfaction problem and implement it using a chronological backtracking algorithm or stochastic local search. [Usage]
8. Compare and contrast basic search issues with game playing issues. [Familiarity]

IS/Basic Knowledge Representation and Reasoning

[3 Core-Tier2 hours]

Topics:

- Review of propositional and predicate logic (cross-reference DS/Basic Logic)
- Resolution and theorem proving (propositional logic only)
- Forward chaining, backward chaining
- Review of probabilistic reasoning, Bayes theorem (cross-reference with DS/Discrete Probability)

Learning Outcomes:

1. Translate a natural language (e.g., English) sentence into predicate logic statement. [Usage]
2. Convert a logic statement into clause form. [Usage]
3. Apply resolution to a set of logic statements to answer a query. [Usage]
4. Make a probabilistic inference in a real-world problem using Bayes' theorem to determine the probability of a hypothesis given evidence. [Usage]

IS/Basic Machine Learning

[2 Core-Tier2 hours]

Topics:

- Definition and examples of broad variety of machine learning tasks, including classification
- Inductive learning
- Simple statistical-based learning, such as Naive Bayesian Classifier, decision trees
- The over-fitting problem
- Measuring classifier accuracy

Learning Outcomes:

1. List the differences among the three main styles of learning: supervised, reinforcement, and unsupervised. [Familiarity]
2. Identify examples of classification tasks, including the available input features and output to be predicted. [Familiarity]
3. Explain the difference between inductive and deductive learning. [Familiarity]

4. Describe over-fitting in the context of a problem. [Familiarity]
5. Apply the simple statistical learning algorithm such as Naive Bayesian Classifier to a classification task and measure the classifier's accuracy. [Usage]

IS/Advanced Search

[Elective]

Note that the general topics of Branch-and-bound and Dynamic Programing are listed in AL/Algorithmic Strategies.

Topics:

- Constructing search trees, dynamic search space, combinatorial explosion of search space
- Stochastic search
  - Simulated annealing
  - Genetic algorithms
  - Monte-Carlo tree search
- Implementation of A* search, beam search
- Minimax search, alpha-beta pruning
- Expectimax search (MDP-solving) and chance nodes

Learning Outcomes:

1. Design and implement a genetic algorithm solution to a problem. [Usage]
2. Design and implement a simulated annealing schedule to avoid local minima in a problem. [Usage]
3. Design and implement A*/beam search to solve a problem. [Usage]
4. Apply minimax search with alpha-beta pruning to prune search space in a two-player game. [Usage]
5. Compare and contrast genetic algorithms with classic search techniques. [Assessment]
6. Compare and contrast various heuristic searches vis-a-vis applicability to a given problem. [Assessment]

IS/Advanced Representation and Reasoning

[Elective]

Topics:

- Knowledge representation issues
  - Description logics
  - Ontology engineering
- Non-monotonic reasoning (e.g., non-classical logics, default reasoning)
- Argumentation
- Reasoning about action and change (e.g., situation and event calculus)
- Temporal and spatial reasoning
- Rule-based Expert Systems
- Semantic networks
- Model-based and Case-based reasoning
- Planning:
  - Partial and totally ordered planning
  - Plan graphs
  - Hierarchical planning
  - Planning and execution including conditional planning and continuous planning
  - Mobile agent/Multi-agent planning

Learning Outcomes:

1. Compare and contrast the most common models used for structured knowledge representation, highlighting their strengths and weaknesses. [Assessment]
2. Identify the components of non-monotonic reasoning and its usefulness as a representational mechanism for belief systems. [Familiarity]
3. Compare and contrast the basic techniques for representing uncertainty. [Assessment]
4. Compare and contrast the basic techniques for qualitative representation. [Assessment]
5. Apply situation and event calculus to problems of action and change. [Usage]
6. Explain the distinction between temporal and spatial reasoning, and how they interrelate. [Familiarity]
7. Explain the difference between rule-based, case-based and model-based reasoning techniques. [Familiarity]
8. Define the concept of a planning system and how it differs from classical search techniques. [Familiarity]
9. Describe the differences between planning as search, operator-based planning, and propositional planning, providing examples of domains where each is most applicable. [Familiarity]
10. Explain the distinction between monotonic and non-monotonic inference. [Familiarity]

IS/Reasoning Under Uncertainty

[Elective]

Topics:

- Review of basic probability (cross-reference DS/Discrete Probability)
- Random variables and probability distributions
    - Axioms of probability
    - Probabilistic inference
    - Bayes' Rule
- Conditional Independence
- Knowledge representations
    - Bayesian Networks
        - Exact inference and its complexity
        - Randomized sampling (Monte Carlo) methods (e.g. Gibbs sampling)
    - Markov Networks
    - Relational probability models
    - Hidden Markov Models
- Decision Theory
    - Preferences and utility functions
    - Maximizing expected utility

Learning Outcomes:

1. Apply Bayes' rule to determine the probability of a hypothesis given evidence. [Usage]
2. Explain how conditional independence assertions allow for greater efficiency of probabilistic systems. [Assessment]
3. Identify examples of knowledge representations for reasoning under uncertainty. [Familiarity]
4. State the complexity of exact inference. Identify methods for approximate inference. [Familiarity]
5. Design and implement at least one knowledge representation for reasoning under uncertainty. [Usage]
6. Describe the complexities of temporal probabilistic reasoning. [Familiarity]
7. Design and implement an HMM as one example of a temporal probabilistic system. [Usage]
8. Describe the relationship between preferences and utility functions. [Familiarity]
9. Explain how utility functions and probabilistic reasoning can be combined to make rational decisions. [Assessment]

IS/Agents

[Elective]

Cross-reference HCI/Collaboration and Communication

Topics:

- Definitions of agents
- Agent architectures (e.g., reactive, layered, cognitive)
- Agent theory
- Rationality, game theory
    - Decision-theoretic agents
    - Markov decision processes (MDP)
- Software agents, personal assistants, and information access
    - Collaborative agents
    - Information-gathering agents
    - Believable agents (synthetic characters, modeling emotions in agents)
- Learning agents
- Multi-agent systems
    - Collaborating agents
    - Agent teams
    - Competitive agents (e.g., auctions, voting)
    - Swarm systems and biologically inspired models

Learning Outcomes:

1. List the defining characteristics of an intelligent agent. [Familiarity]
2. Characterize and contrast the standard agent architectures. [Assessment]
3. Describe the applications of agent theory to domains such as software agents, personal assistants, and believable agents. [Familiarity]
4. Describe the primary paradigms used by learning agents. [Familiarity]
5. Demonstrate using appropriate examples how multi-agent systems support agent interaction. [Usage]

IS/Natural Language Processing

[Elective]

Cross-reference HCI/New Interactive Technologies

Topics:

- Deterministic and stochastic grammars
- Parsing algorithms
    - CFGs and chart parsers (e.g. CYK)
    - Probabilistic CFGs and weighted CYK
- Representing meaning / Semantics
    - Logic-based knowledge representations
    - Semantic roles
    - Temporal representations
    - Beliefs, desires, and intentions
- Corpus-based methods
- N-grams and HMMs
- Smoothing and backoff

- Examples of use: POS tagging and morphology
- Information retrieval (Cross-reference IM/Information Storage and Retrieval)
    - Vector space model
        - TF & IDF
    - Precision and recall
- Information extraction
- Language translation
- Text classification, categorization
    - Bag of words model

Learning Outcomes:

1. Define and contrast deterministic and stochastic grammars, providing examples to show the adequacy of each. [Assessment]
2. Simulate, apply, or implement classic and stochastic algorithms for parsing natural language. [Usage]
3. Identify the challenges of representing meaning. [Familiarity]
4. List the advantages of using standard corpora. Identify examples of current corpora for a variety of NLP tasks. [Familiarity]
5. Identify techniques for information retrieval, language translation, and text classification. [Familiarity]

IS/Advanced Machine Learning

[Elective]

Topics:

- Definition and examples of broad variety of machine learning tasks
- General statistical-based learning, parameter estimation (maximum likelihood)
- Inductive logic programming (ILP)
- Supervised learning
    - Learning decision trees
    - Learning neural networks
    - Support vector machines (SVMs)
- Ensembles
- Nearest-neighbor algorithms
- Unsupervised Learning and clustering
    - EM
    - K-means
    - Self-organizing maps
- Semi-supervised learning
- Learning graphical models (Cross-reference IS/Reasoning under Uncertainty)
- Performance evaluation (such as cross-validation, area under ROC curve)
- Learning theory
- The problem of overfitting, the curse of dimensionality
- Reinforcement learning
    - Exploration vs. exploitation trade-off
    - Markov decision processes
    - Value and policy iteration
- Application of Machine Learning algorithms to Data Mining (cross-reference IM/Data Mining)

Learning Outcomes:

1. Explain the differences among the three main styles of learning: supervised, reinforcement, and unsupervised. [Familiarity]
2. Implement simple algorithms for supervised learning, reinforcement learning, and unsupervised learning. [Usage]
3. Determine which of the three learning styles is appropriate to a particular problem domain. [Usage]
4. Compare and contrast each of the following techniques, providing examples of when each strategy is superior: decision trees, neural networks, and belief networks. [Assessment]
5. Evaluate the performance of a simple learning system on a real-world dataset. [Assessment]
6. Characterize the state of the art in learning theory, including its achievements and its shortcomings. [Familiarity]
7. Explain the problem of overfitting, along with techniques for detecting and managing the problem. [Usage]

IS/Robotics

[Elective]

Topics:

- Overview: problems and progress
    - State-of-the-art robot systems, including their sensors and an overview of their sensor processing
    - Robot control architectures, e.g., deliberative vs. reactive control and Braitenberg vehicles
    - World modeling and world models
    - Inherent uncertainty in sensing and in control
- Configuration space and environmental maps
- Interpreting uncertain sensor data
- Localizing and mapping
- Navigation and control
- Motion planning
- Multiple-robot coordination

Learning Outcomes:

1. List capabilities and limitations of today's state-of-the-art robot systems, including their sensors and the crucial sensor processing that informs those systems. [Familiarity]
2. Integrate sensors, actuators, and software into a robot designed to undertake some task. [Usage]
3. Program a robot to accomplish simple tasks using deliberative, reactive, and/or hybrid control architectures. [Usage]
4. Implement fundamental motion planning algorithms within a robot configuration space. [Usage]
5. Characterize the uncertainties associated with common robot sensors and actuators; articulate strategies for mitigating these uncertainties. [Familiarity]
6. List the differences among robots' representations of their external environment, including their strengths and shortcomings. [Familiarity]
7. Compare and contrast at least three strategies for robot navigation within known and/or unknown environments, including their strengths and shortcomings. [Assessment]
8. Describe at least one approach for coordinating the actions and sensing of several robots to accomplish a single task. [Familiarity]

IS/Perception and Computer Vision

[Elective]

Topics:

- Computer vision
    - Image acquisition, representation, processing and properties
    - Shape representation, object recognition and segmentation
    - Motion analysis
- Audio and speech recognition
- Modularity in recognition
- Approaches to pattern recognition (cross-reference IS/Advanced Machine Learning)
    - Classification algorithms and measures of classification quality
    - Statistical techniques

Learning Outcomes:

1. Summarize the importance of image and object recognition in AI and indicate several significant applications of this technology. [Familiarity]
2. List at least three image-segmentation approaches, such as thresholding, edge-based and region-based algorithms, along with their defining characteristics, strengths, and weaknesses. [Familiarity]
3. Implement 2d object recognition based on contour- and/or region-based shape representations. [Usage]
4. Distinguish the goals of sound-recognition, speech-recognition, and speaker-recognition and identify how the raw audio signal will be handled differently in each of these cases. [Familiarity]
5. Provide at least two examples of a transformation of a data source from one sensory domain to another, e.g., tactile data interpreted as single-band 2d images. [Familiarity]
6. Implement a feature-extraction algorithm on real data, e.g., an edge or corner detector for images or vectors of Fourier coefficients describing a short slice of audio signal. [Usage]
7. Implement an algorithm combining features into higher-level percepts, e.g., a contour or polygon from visual primitives or phoneme hypotheses from an audio signal. [Usage]
8. Implement a classification algorithm that segments input percepts into output categories and quantitatively evaluates the resulting classification. [Usage]
9. Evaluate the performance of the underlying feature-extraction, relative to at least one alternative possible approach (whether implemented or not) in its contribution to the classification task (8), above. [Assessment]
10. Describe at least three classification approaches, their prerequisites for applicability, their strengths, and their shortcomings. [Familiarity]

Networking and Communication (NC)

The Internet and computer networks are now ubiquitous and a growing number of computing activities strongly depend on the correct operation of the underlying network. Networks, both fixed and mobile, are a key part of the computing environment of today and tomorrow. Many computing applications that are used today would not be possible without networks. This dependency on the underlying network is likely to increase in the future.

The high-level learning objective of this module can be summarized as follows:

- Thinking in a networked world. The world is more and more interconnected and the use of networks will continue to increase. Students must understand how the networks behave and the key principles behind the organization and operation of the networks.
- Continued study. The networking domain is rapidly evolving and a first networking course should be a starting point to other more advanced courses on network design, network management, sensor networks, etc.
- Principles and practice interact. Networking is real and many of the design choices that involve networks also depend on practical constraints. Students should be exposed to these practical constraints by experimenting with networking, using tools, and writing networked software.

There are different ways of organizing a networking course. Some educators prefer a top-down approach, i.e., the course starts from the applications and then explains reliable delivery, routing and forwarding. Other educators prefer a bottom-up approach where the students start with the lower layers and build their understanding of the network, transport and application layers later.

NC. Networking and Communication (3 Core-Tier1 hours, 7 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
NC/Introduction	**1.5**		**N**
NC/Networked Applications	**1.5**		**N**
NC/Reliable Data Delivery		**2**	**N**
NC/Routing And Forwarding		**1.5**	**N**
NC/Local Area Networks		**1.5**	**N**
NC/Resource Allocation		**1**	**N**
NC/Mobility		**1**	**N**
NC/Social Networking			**Y**

NC/Introduction

[1.5 Core-Tier1 hours]

Cross-reference IAS/Network Security, which discusses network security and its applications.

Topics:

- Organization of the Internet (Internet Service Providers, Content Providers, etc.)
- Switching techniques (e.g., circuit, packet)
- Physical pieces of a network, including hosts, routers, switches, ISPs, wireless, LAN, access point, and firewalls
- Layering principles (encapsulation, multiplexing)
- Roles of the different layers (application, transport, network, datalink, physical)

Learning Outcomes:

1. Articulate the organization of the Internet. [Familiarity]
2. List and define the appropriate network terminology. [Familiarity]
3. Describe the layered structure of a typical networked architecture. [Familiarity]
4. Identify the different types of complexity in a network (edges, core, etc.). [Familiarity]

NC/Networked Applications

[1.5 Core-Tier1 hours]

Topics:

- Naming and address schemes (DNS, IP addresses, Uniform Resource Identifiers, etc.)
- Distributed applications (client/server, peer-to-peer, cloud, etc.)
- HTTP as an application layer protocol
- Multiplexing with TCP and UDP
- Socket APIs

Learning Outcomes:

1. List the differences and the relations between names and addresses in a network. [Familiarity]
2. Define the principles behind naming schemes and resource location. [Familiarity]
3. Implement a simple client-server socket-based application. [Usage]

NC/Reliable Data Delivery

[2 Core-Tier2 hours]

This knowledge unit is related to Systems Fundamentals (SF). Cross-reference SF/State and State Machines and SF/Reliability through Redundancy.

Topics:

- Error control (retransmission techniques, timers)
- Flow control (acknowledgements, sliding window)
- Performance issues (pipelining)
- TCP

Learning Outcomes:

1. Describe the operation of reliable delivery protocols. [Familiarity]
2. List the factors that affect the performance of reliable delivery protocols. [Familiarity]
3. Design and implement a simple reliable protocol. [Usage]

NC/Routing and Forwarding

[1.5 Core-Tier2 hours]

Topics:

- Routing versus forwarding
- Static routing
- Internet Protocol (IP)
- Scalability issues (hierarchical addressing)

Learning Outcomes:

1. Describe the organization of the network layer. [Familiarity]
2. Describe how packets are forwarded in an IP network. [Familiarity]
3. List the scalability benefits of hierarchical addressing. [Familiarity]

NC/Local Area Networks

[1.5 Core-Tier2 hours]

Topics:

- Multiple Access Problem
- Common approaches to multiple access (exponential-backoff, time division multiplexing, etc)
- Local Area Networks
- Ethernet
- Switching

Learning Outcomes:

1. Describe how frames are forwarded in an Ethernet network. [Familiarity]
2. Describe the differences between IP and Ethernet. [Familiarity]
3. Describe the interrelations between IP and Ethernet. [Familiarity]
4. Describe the steps used in one common approach to the multiple access problem. [Familiarity]

NC/Resource Allocation

[1 Core-Tier2 hours]

Topics:

- Need for resource allocation
- Fixed allocation (TDM, FDM, WDM) versus dynamic allocation
- End-to-end versus network assisted approaches
- Fairness
- Principles of congestion control
- Approaches to Congestion (e.g., Content Distribution Networks)

Learning Outcomes:

1. Describe how resources can be allocated in a network. [Familiarity]
2. Describe the congestion problem in a large network. [Familiarity]
3. Compare and contrast fixed and dynamic allocation techniques. [Assessment]
4. Compare and contrast current approaches to congestion. [Assessment]

NC/Mobility

[1 Core-Tier2 hours]

Topics:

- Principles of cellular networks
- 802.11 networks
- Issues in supporting mobile nodes (home agents)

Learning Outcomes:

1. Describe the organization of a wireless network. [Familiarity]
2. Describe how wireless networks support mobile users. [Familiarity]

NC/Social Networking

[Elective]

Topics:

- Social networks overview
- Example social network platforms
- Structure of social network graphs
- Social network analysis

Learning Outcomes:

1. Discuss the key principles (such as membership, trust) of social networking. [Familiarity]
2. Describe how existing social networks operate. [Familiarity]
3. Construct a social network graph from network data. [Usage]
4. Analyze a social network to determine who the key people are. [Usage]
5. Evaluate a given interpretation of a social network question with associated data. [Assessment]

Operating Systems (OS)

An operating system defines an abstraction of hardware and manages resource sharing among the computer's users. The topics in this area explain the most basic knowledge of operating systems in the sense of interfacing an operating system to networks, teaching the difference between the kernel and user modes, and developing key approaches to operating system design and implementation. This knowledge area is structured to be complementary to the Systems Fundamentals (SF), Networking and Communication (NC), Information Assurance and Security (IAS), and the Parallel and Distributed Computing (PD) knowledge areas. The Systems Fundamentals and Information Assurance and Security knowledge areas are the new ones to include contemporary issues. For example, Systems Fundamentals includes topics such as performance, virtualization and isolation, and resource allocation and scheduling; Parallel and Distributed Systems includes parallelism fundamentals; and and Information Assurance and Security includes forensics and security issues in depth. Many courses in Operating Systems will draw material from across these knowledge areas.

OS. Operating Systems (4 Core-Tier1 hours; 11 Core Tier2 hours)

	Core-Tier1 hours	**Core-Tier2 hours**	**Includes Electives**
OS/Overview of Operating Systems	**2**		**N**
OS/Operating System Principles	**2**		**N**
OS/Concurrency		**3**	**N**
OS/Scheduling and Dispatch		**3**	**N**
OS/Memory Management		**3**	**N**
OS/Security and Protection		**2**	**N**
OS/Virtual Machines			**Y**
OS/Device Management			**Y**
OS/File Systems			**Y**
OS/Real Time and Embedded Systems			**Y**
OS/Fault Tolerance			**Y**
OS/System Performance Evaluation			**Y**

OS/Overview of Operating Systems

[2 Core-Tier1 hours]

Topics:

- Role and purpose of the operating system
- Functionality of a typical operating system
- Mechanisms to support client-server models, hand-held devices
- Design issues (efficiency, robustness, flexibility, portability, security, compatibility)
- Influences of security, networking, multimedia, windowing systems

Learning Outcomes:

1. Explain the objectives and functions of modern operating systems. [Familiarity]
2. Analyze the tradeoffs inherent in operating system design. [Usage]
3. Describe the functions of a contemporary operating system with respect to convenience, efficiency, and the ability to evolve. [Familiarity]
4. Discuss networked, client-server, distributed operating systems and how they differ from single user operating systems. [Familiarity]
5. Identify potential threats to operating systems and the security features design to guard against them. [Familiarity]

OS/Operating System Principles

[2 Core-Tier1 hours]

Topics:

- Structuring methods (monolithic, layered, modular, micro-kernel models)
- Abstractions, processes, and resources
- Concepts of application program interfaces (APIs)
- The evolution of hardware/software techniques and application needs
- Device organization
- Interrupts: methods and implementations
- Concept of user/system state and protection, transition to kernel mode

Learning Outcomes:

1. Explain the concept of a logical layer. [Familiarity]
2. Explain the benefits of building abstract layers in hierarchical fashion. [Familiarity]
3. Describe the value of APIs and middleware. [Assessment]
4. Describe how computing resources are used by application software and managed by system software. [Familiarity]
5. Contrast kernel and user mode in an operating system. [Usage]
6. Discuss the advantages and disadvantages of using interrupt processing. [Familiarity]
7. Explain the use of a device list and driver I/O queue. [Familiarity]

OS/Concurrency

[3 Core-Tier2 hours]

Topics:

- States and state diagrams (cross-reference SF/State and State Machines)
- Structures (ready list, process control blocks, and so forth)
- Dispatching and context switching
- The role of interrupts
- Managing atomic access to OS objects
- Implementing synchronization primitives
- Multiprocessor issues (spin-locks, reentrancy) (cross-reference SF/Parallelism)

Learning Outcomes:

1. Describe the need for concurrency within the framework of an operating system. [Familiarity]
2. Demonstrate the potential run-time problems arising from the concurrent operation of many separate tasks. [Usage]
3. Summarize the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each. [Familiarity]
4. Explain the different states that a task may pass through and the data structures needed to support the management of many tasks. [Familiarity]
5. Summarize techniques for achieving synchronization in an operating system (e.g., describe how to implement a semaphore using OS primitives). [Familiarity]
6. Describe reasons for using interrupts, dispatching, and context switching to support concurrency in an operating system. [Familiarity]
7. Create state and transition diagrams for simple problem domains. [Usage]

OS/Scheduling and Dispatch

[3 Core-Tier2 hours]

Topics:

- Preemptive and non-preemptive scheduling (cross-reference SF/Resource Allocation and Scheduling, PD/Parallel Performance)
- Schedulers and policies (cross-reference SF/Resource Allocation and Scheduling, PD/Parallel Performance)
- Processes and threads (cross-reference SF/Computational paradigms)
- Deadlines and real-time issues

Learning Outcomes:

1. Compare and contrast the common algorithms used for both preemptive and non-preemptive scheduling of tasks in operating systems, such as priority, performance comparison, and fair-share schemes. [Usage]
2. Describe relationships between scheduling algorithms and application domains. [Familiarity]
3. Discuss the types of processor scheduling such as short-term, medium-term, long-term, and I/O. [Familiarity]
4. Describe the difference between processes and threads. [Usage]
5. Compare and contrast static and dynamic approaches to real-time scheduling. [Usage]
6. Discuss the need for preemption and deadline scheduling. [Familiarity]
7. Identify ways that the logic embodied in scheduling algorithms are applicable to other domains, such as disk I/O, network scheduling, project scheduling, and problems beyond computing. [Usage]

OS/Memory Management

[3 Core-Tier2 hours]

Topics:

- Review of physical memory and memory management hardware
- Working sets and thrashing
- Caching (cross-reference AR/Memory System Organization and Architecture)

Learning Outcomes:

1. Explain memory hierarchy and cost-performance trade-offs. [Familiarity]
2. Summarize the principles of virtual memory as applied to caching and paging. [Familiarity]
3. Evaluate the trade-offs in terms of memory size (main memory, cache memory, auxiliary memory) and processor speed. [Assessment]
4. Defend the different ways of allocating memory to tasks, citing the relative merits of each. [Assessment]
5. Describe the reason for and use of cache memory (performance and proximity, different dimension of how caches complicate isolation and VM abstraction). [Familiarity]
6. Discuss the concept of thrashing, both in terms of the reasons it occurs and the techniques used to recognize and manage the problem. [Familiarity]

OS/Security and Protection

[2 Core-Tier2 hours]

Topics:

- Overview of system security
- Policy/mechanism separation
- Security methods and devices
- Protection, access control, and authentication
- Backups

Learning Outcomes:

1. Articulate the need for protection and security in an OS (cross-reference IAS/Security Architecture and Systems Administration/Investigating Operating Systems Security for various systems). [Assessment]
2. Summarize the features and limitations of an operating system used to provide protection and security (cross-reference IAS/Security Architecture and Systems Administration). [Familiarity]
3. Explain the mechanisms available in an OS to control access to resources (cross-reference IAS/Security Architecture and Systems Administration/Access Control/Configuring systems to operate securely as an IT system). [Familiarity]
4. Carry out simple system administration tasks according to a security policy, for example creating accounts, setting permissions, applying patches, and arranging for regular backups (cross-reference IAS/Security Architecture and Systems Administration). [Usage]

OS/Virtual Machines

[Elective]

Topics:

- Types of virtualization (including Hardware/Software, OS, Server, Service, Network)
- Paging and virtual memory
- Virtual file systems
- Hypervisors
- Portable virtualization; emulation vs. isolation
- Cost of virtualization

Learning Outcomes:

1. Explain the concept of virtual memory and how it is realized in hardware and software. [Familiarity]
5. Differentiate emulation and isolation. [Familiarity]
6. Evaluate virtualization trade-offs. [Assessment]
2. Discuss hypervisors and the need for them in conjunction with different types of hypervisors. [Usage]

OS/Device Management

[Elective]

Topics:

- Characteristics of serial and parallel devices
- Abstracting device differences
- Buffering strategies
- Direct memory access
- Recovery from failures

Learning Outcomes:

1. Explain the key difference between serial and parallel devices and identify the conditions in which each is appropriate. [Familiarity]
2. Identify the relationship between the physical hardware and the virtual devices maintained by the operating system. [Usage]
3. Explain buffering and describe strategies for implementing it. [Familiarity]
4. Differentiate the mechanisms used in interfacing a range of devices (including hand-held devices, networks, multimedia) to a computer and explain the implications of these for the design of an operating system. [Usage]
5. Describe the advantages and disadvantages of direct memory access and discuss the circumstances in which its use is warranted. [Usage]
6. Identify the requirements for failure recovery. [Familiarity]
7. Implement a simple device driver for a range of possible devices. [Usage]

OS/File Systems

[Elective]

Topics:

- Files: data, metadata, operations, organization, buffering, sequential, nonsequential
- Directories: contents and structure
- File systems: partitioning, mount/unmount, virtual file systems
- Standard implementation techniques
- Memory-mapped files
- Special-purpose file systems
- Naming, searching, access, backups
- Journaling and log-structured file systems

Learning Outcomes:

1. Describe the choices to be made in designing file systems. [Familiarity]
2. Compare and contrast different approaches to file organization, recognizing the strengths and weaknesses of each. [Usage]
3. Summarize how hardware developments have led to changes in the priorities for the design and the management of file systems. [Familiarity]
4. Summarize the use of journaling and how log-structured file systems enhance fault tolerance. [Familiarity]

OS/Real Time and Embedded Systems

[Elective]

Topics:

- Process and task scheduling
- Memory/disk management requirements in a real-time environment
- Failures, risks, and recovery
- Special concerns in real-time systems

Learning Outcomes:

1. Describe what makes a system a real-time system. [Familiarity]
2. Explain the presence of and describe the characteristics of latency in real-time systems. [Familiarity]
3. Summarize special concerns that real-time systems present, including risk, and how these concerns are addressed. [Familiarity]

OS/Fault Tolerance

[Elective]

Topics:

- Fundamental concepts: reliable and available systems (cross-reference SF/Reliability through Redundancy)
- Spatial and temporal redundancy (cross-reference SF/Reliability through Redundancy)
- Methods used to implement fault tolerance
- Examples of OS mechanisms for detection, recovery, restart to implement fault tolerance, use of these techniques for the OS's own services

Learning Outcomes:

1. Explain the relevance of the terms fault tolerance, reliability, and availability. [Familiarity]
2. Outline the range of methods for implementing fault tolerance in an operating system. [Familiarity]
3. Explain how an operating system can continue functioning after a fault occurs. [Familiarity]

OS/System Performance Evaluation

[Elective]

Topics:

- Why system performance needs to be evaluated (cross-reference SF/Performance/Figures of performance merit)
- What is to be evaluated (cross-reference SF/Performance/Figures of performance merit)
- Systems performance policies, e.g., caching, paging, scheduling, memory management, and security
- Evaluation models: deterministic, analytic, simulation, or implementation-specific
- How to collect evaluation data (profiling and tracing mechanisms)

Learning Outcomes:

1. Describe the performance measurements used to determine how a system performs. [Familiarity]
2. Explain the main evaluation models used to evaluate a system. [Familiarity]

Platform-Based Development (PBD)

Platform-based development is concerned with the design and development of software applications that reside on specific software platforms. In contrast to general purpose programming, platform-based development takes into account platform-specific constraints. For instance web programming, multimedia development, mobile computing, app development, and robotics are examples of relevant platforms that provide specific services/APIs/hardware that constrain development. Such platforms are characterized by the use of specialized APIs, distinct delivery/update mechanisms, and being abstracted away from the machine level. Platform-based development may be applied over a wide breadth of ecosystems.

While we recognize that some platforms (e.g., web development) are prominent, we are also cognizant of the fact that no particular platform should be specified as a requirement in the CS2013 curricular guidelines. Consequently, this Knowledge Area highlights many of the platforms that have become popular, without including any such platform in the core curriculum. We note that the general skill of developing with respect to an API or a constrained environment is covered in other Knowledge Areas, such as Software Development Fundamentals (SDF). Platform-based development further emphasizes such general skills within the context of particular platforms.

PBD. Platform-Based Development (Elective)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
PBD/Introduction			**Y**
PBD/Web Platforms			**Y**
PBD/Mobile Platforms			**Y**
PBD/Industrial Platforms			**Y**
PBD/Game Platforms			**Y**

PBD/Introduction

[Elective]

This knowledge unit describes the fundamental differences that Platform-Based Development has over traditional software development.

Topics:

- Overview of platforms (e.g., Web, Mobile, Game, Industrial)
- Programming via platform-specific APIs
- Overview of Platform Languages (e.g., Objective C, HTML5)
- Programming under platform constraints

Learning Outcomes:

1. Describe how platform-based development differs from general purpose programming. [Familiarity]
2. List characteristics of platform languages. [Familiarity]
3. Write and execute a simple platform-based program. [Usage]
4. List the advantages and disadvantages of programming with platform constraints. [Familiarity]

PBD/Web Platforms

[Elective]

Topics:

- Web programming languages (e.g., HTML5, Java Script, PHP, CSS)
- Web platform constraints
- Software as a Service (SaaS)
- Web standards

Learning Outcomes:

1. Design and Implement a simple web application. [Usage]
2. Describe the constraints that the web puts on developers. [Familiarity]
3. Compare and contrast web programming with general purpose programming. [Assessment]
4. Describe the differences between Software-as-a-Service and traditional software products. [Familiarity]
5. Discuss how web standards impact software development. [Familiarity]
6. Review an existing web application against a current web standard. [Assessment]

PBD/Mobile Platforms

[Elective]

Topics:

- Mobile programming languages
- Challenges with mobility and wireless communication
- Location-aware applications
- Performance / power tradeoffs
- Mobile platform constraints
- Emerging technologies

Learning Outcomes:

1. Design and implement a mobile application for a given mobile platform. [Usage]
2. Discuss the constraints that mobile platforms put on developers. [Familiarity]
3. Discuss the performance vs. power tradeoff. [Familiarity]
4. Compare and contrast mobile programming with general purpose programming. [Assessment]

PBD/Industrial Platforms

[Elective]

This knowledge unit is related to IS/Robotics.

Topics:

- Types of Industrial Platforms (e.g., Mathematic, Robotic, Industrial Control)
- Robotic software and its architecture
- Domain-specific languages
- Industrial platform constraints

Learning Outcomes:

1. Design and implement an industrial application on a given platform (e.g., using Lego Mindstorms or Matlab). [Usage]
2. Compare and contrast domain specific languages with general purpose programming languages. [Assessment]
3. Discuss the constraints that a given industrial platforms impose on developers. [Familiarity]

PBD/Game Platforms

[Elective]

Topics:

- Types of game platforms (e.g., XBox, Wii, PlayStation)
- Game platform languages (e.g., C++, Java, Lua, Python)
- Game platform constraints

Learning Outcomes:

1. Design and implement a simple application on a game platform. [Usage]
2. Describe the constraints that game platforms impose on developers. [Familiarity]
3. Compare and contrast game programming with general purpose programming. [Assessment]

Parallel and Distributed Computing (PD)

The past decade has brought explosive growth in multiprocessor computing, including multi-core processors and distributed data centers. As a result, parallel and distributed computing has moved from a largely elective topic to become more of a core component of undergraduate computing curricula. Both parallel and distributed computing entail the logically simultaneous execution of multiple processes, whose operations have the potential to interleave in complex ways. Parallel and distributed computing builds on foundations in many areas, including an understanding of fundamental systems concepts such as concurrency and parallel execution, consistency in state/memory manipulation, and latency. Communication and coordination among processes is rooted in the message-passing and shared-memory models of computing and such algorithmic concepts as atomicity, consensus, and conditional waiting. Achieving speedup in practice requires an understanding of parallel algorithms, strategies for problem decomposition, system architecture, detailed implementation strategies, and performance analysis and tuning. Distributed systems highlight the problems of security and fault tolerance, emphasize the maintenance of replicated state, and introduce additional issues that bridge to computer networking.

Because parallelism interacts with so many areas of computing, including at least algorithms, languages, systems, networking, and hardware, many curricula will put different parts of the knowledge area in different courses, rather than in a dedicated course. While we acknowledge that computer science is moving in this direction and may reach that point, in 2013 this process is still in flux and we feel it provides more useful guidance to curriculum designers to aggregate the fundamental parallelism topics in one place. Note, however, that the fundamentals of concurrency and mutual exclusion appear in the Systems Fundamentals (SF) Knowledge Area. Many curricula may choose to introduce parallelism and concurrency in the same course (see below for the distinction intended by these terms). Further, we note that the topics and learning outcomes listed below include only brief mentions of purely elective coverage. At the present time, there is too much diversity in topics that share little in common (including for example, parallel scientific computing, process calculi, and non-blocking data structures) to recommend particular topics be covered in elective courses.

Because the terminology of parallel and distributed computing varies among communities, we provide here brief descriptions of the intended senses of a few terms. This list is not exhaustive or definitive, but is provided for the sake of clarity.

- *Parallelism:* Using additional computational resources simultaneously, usually for speedup.
- *Concurrency:* Efficiently and correctly managing concurrent access to resources.
- *Activity*: A computation that may proceed concurrently with others; for example a program, process, thread, or active parallel hardware component.
- *Atomicity*: Rules and properties governing whether an action is observationally indivisible; for example, setting all of the bits in a word, transmitting a single packet, or completing a transaction.
- *Consensus*: Agreement among two or more activities about a given predicate; for example, the value of a counter, the owner of a lock, or the termination of a thread.
- *Consistency*: Rules and properties governing agreement about the values of variables written, or messages produced, by some activities and used by others (thus possibly exhibiting a *data race*); for example, *sequential consistency*, stating that the values of all variables in a shared memory parallel program are equivalent to that of a single program performing some interleaving of the memory accesses of these activities.
- *Multicast*: A message sent to possibly many recipients, generally without any constraints about whether some recipients receive the message before others. An *event* is a multicast message sent to a designated set of *listeners* or *subscribers*.

As multi-processor computing continues to grow in the coming years, so too will the role of parallel and distributed computing in undergraduate computing curricula. In addition to the guidelines presented here, we also direct the interested reader to the document entitled "NSF/TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates", available from the website: http://www.cs.gsu.edu/~tcpp/curriculum/.

General cross-referencing note: Systems Fundamentals also contains an introduction to parallelism (SF/Computational Paradigms, SF/System Support for Parallelism, SF/Performance).

The introduction to parallelism in SF complements the one here and there is no ordering constraint between them. In SF, the idea is to provide a unified view of the system support for simultaneous execution at multiple levels of abstraction (parallelism is inherent in gates, processors, operating systems, and servers), whereas here the focus is on a preliminary understanding of parallelism as a computing primitive and the complications that arise in parallel and concurrent programming. Given these different perspectives, the hours assigned to each are not redundant: the layered systems view and the high-level computing concepts are accounted for separately in terms of the core hours.

PD. Parallel and Distributed Computing (5 Core-Tier1 hours, 10 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
PD/Parallelism Fundamentals	**2**		**N**
PD/Parallel Decomposition	**1**	**3**	**N**
PD/Communication and Coordination	**1**	**3**	**Y**
PD/Parallel Algorithms, Analysis, and Programming		**3**	**Y**
PD/Parallel Architecture	**1**	**1**	**Y**
PD/Parallel Performance			**Y**
PD/Distributed Systems			**Y**
PD/Cloud Computing			**Y**
PD/Formal Models and Semantics			**Y**

PD/Parallelism Fundamentals

[2 Core-Tier1 hours]

Build upon students' familiarity with the notion of basic parallel execution—a concept addressed in Systems Fundamentals—to delve into the complicating issues that stem from this notion, such as race conditions and liveness.

Cross-reference SF/Computational Paradigms and SF/System Support for Parallelism.

Topics:

- Multiple simultaneous computations
- Goals of parallelism (e.g., throughput) versus concurrency (e.g., controlling access to shared resources)
- Parallelism, communication, and coordination
  - Programming constructs for coordinating multiple simultaneous computations
  - Need for synchronization
- Programming errors not found in sequential programming
  - Data races (simultaneous read/write or write/write of shared state)
  - Higher-level races (interleavings violating program intention, undesired non-determinism)
  - Lack of liveness/progress (deadlock, starvation)

Learning outcomes:

1. Distinguish using computational resources for a faster answer from managing efficient access to a shared resource. (Cross-reference GV/Fundamental Concepts, outcome 5.) [Familiarity]
2. Distinguish multiple sufficient programming constructs for synchronization that may be inter-implementable but have complementary advantages. [Familiarity]
3. Distinguish data races from higher level races. [Familiarity]

PD/Parallel Decomposition

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

(Cross-reference SF/System Support for Parallelism)

Topics:

[Core-Tier1]

- Need for communication and coordination/synchronization
- Independence and partitioning

[Core-Tier2]

- Basic knowledge of parallel decomposition concepts (cross-reference SF/System Support for Parallelism)
- Task-based decomposition
  - Implementation strategies such as threads
- Data-parallel decomposition
  - Strategies such as SIMD and MapReduce
- Actors and reactive processes (e.g., request handlers)

Learning outcomes:

[Core-Tier1]

1. Explain why synchronization is necessary in a specific parallel program. [Usage]
2. Identify opportunities to partition a serial program into independent parallel modules. [Familiarity]

[Core-Tier2]

3. Write a correct and scalable parallel algorithm. [Usage]
4. Parallelize an algorithm by applying task-based decomposition. [Usage]
5. Parallelize an algorithm by applying data-parallel decomposition. [Usage]
6. Write a program using actors and/or reactive processes. [Usage]

PD/Communication and Coordination

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

Cross-reference OS/Concurrency for mechanism implementation issues.

Topics:

[Core-Tier1]

- Shared Memory
- Consistency, and its role in programming language guarantees for data-race-free programs

[Core-Tier2]

- Message passing
    - Point-to-point versus multicast (or event-based) messages
    - Blocking versus non-blocking styles for sending and receiving messages
    - Message buffering (cross-reference PF/Fundamental Data Structures/Queues)
- Atomicity
    - Specifying and testing atomicity and safety requirements
    - Granularity of atomic accesses and updates, and the use of constructs such as critical sections or transactions to describe them
    - Mutual Exclusion using locks, semaphores, monitors, or related constructs
        - Potential for liveness failures and deadlock (causes, conditions, prevention)
    - Composition
        - Composing larger granularity atomic actions using synchronization
        - Transactions, including optimistic and conservative approaches

[Elective]

- Consensus
    - (Cyclic) barriers, counters, or related constructs
- Conditional actions
    - Conditional waiting (e.g., using condition variables)

Learning outcomes:

[Core-Tier1]

1. Use mutual exclusion to avoid a given race condition. [Usage]
2. Give an example of an ordering of accesses among concurrent activities (e.g., program with a data race) that is not sequentially consistent. [Familiarity]

[Core-Tier2]

3. Give an example of a scenario in which blocking message sends can deadlock. [Usage]
4. Explain when and why multicast or event-based messaging can be preferable to alternatives. [Familiarity]
5. Write a program that correctly terminates when all of a set of concurrent tasks have completed. [Usage]
6. Use a properly synchronized queue to buffer data passed among activities. [Usage]
7. Explain why checks for preconditions, and actions based on these checks, must share the same unit of atomicity to be effective. [Familiarity]
8. Write a test program that can reveal a concurrent programming error; for example, missing an update when two activities both try to increment a variable. [Usage]
9. Describe at least one design technique for avoiding liveness failures in programs using multiple locks or semaphores. [Familiarity]
10. Describe the relative merits of optimistic versus conservative concurrency control under different rates of contention among updates. [Familiarity]
11. Give an example of a scenario in which an attempted optimistic update may never complete. [Familiarity]

[Elective]

12. Use semaphores or condition variables to block threads until a necessary precondition holds. [Usage]

PD/Parallel Algorithms, Analysis, and Programming

[3 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Critical paths, work and span, and the relation to Amdahl's law (cross-reference SF/Performance)
- Speed-up and scalability
- Naturally (embarrassingly) parallel algorithms
- Parallel algorithmic patterns (divide-and-conquer, map and reduce, master-workers, others)
    - Specific algorithms (e.g., parallel MergeSort)

[Elective]

- Parallel graph algorithms (e.g., parallel shortest path, parallel spanning tree) (cross-reference AL/Algorithmic Strategies/Divide-and-conquer)
- Parallel matrix computations
- Producer-consumer and pipelined algorithms
- Examples of non-scalable parallel algorithms

Learning outcomes:

[Core-Tier2]

1. Define “critical path”, “work”, and “span”. [Familiarity]
2. Compute the work and span, and determine the critical path with respect to a parallel execution diagram. [Usage]
3. Define “speed-up” and explain the notion of an algorithm’s scalability in this regard. [Familiarity]
4. Identify independent tasks in a program that may be parallelized. [Usage]
5. Characterize features of a workload that allow or prevent it from being naturally parallelized. [Familiarity]
6. Implement a parallel divide-and-conquer (and/or graph algorithm) and empirically measure its performance relative to its sequential analog. [Usage]
7. Decompose a problem (e.g., counting the number of occurrences of some word in a document) via map and reduce operations. [Usage]

[Elective]

8. Provide an example of a problem that fits the producer-consumer paradigm. [Familiarity]
9. Give examples of problems where pipelining would be an effective means of parallelization. [Familiarity]
10. Implement a parallel matrix algorithm. [Usage]
11. Identify issues that arise in producer-consumer algorithms and mechanisms that may be used for addressing them. [Familiarity]

PD/Parallel Architecture

[1 Core-Tier1 hour, 1 Core-Tier2 hour]

The topics listed here are related to knowledge units in the Architecture and Organization (AR) knowledge area (AR/Assembly Level Machine Organization and AR/Multiprocessing and Alternative Architectures). Here, we focus on parallel architecture from the standpoint of applications, whereas the Architecture and Organization knowledge area presents the topic from the hardware perspective.

[Core-Tier1]

- Multicore processors
- Shared vs. distributed memory

[Core-Tier2]

- Symmetric multiprocessing (SMP)
- SIMD, vector processing

[Elective]

- GPU, co-processing
- Flynn’s taxonomy
- Instruction level support for parallel programming
    - Atomic instructions such as Compare and Set
- Memory issues
    - Multiprocessor caches and cache coherence
    - Non-uniform memory access (NUMA)

- Topologies
    - Interconnects
    - Clusters
    - Resource sharing (e.g., buses and interconnects)

Learning outcomes:

[Core-Tier1]

1. Explain the differences between shared and distributed memory. [Familiarity]

[Core-Tier2]

2. Describe the SMP architecture and note its key features. [Familiarity]
3. Characterize the kinds of tasks that are a natural match for SIMD machines. [Familiarity]

[Elective]

4. Describe the advantages and limitations of GPUs vs. CPUs. [Familiarity]
5. Explain the features of each classification in Flynn's taxonomy. [Familiarity]
6. Describe assembly-level support for atomic operations. [Familiarity]
7. Describe the challenges in maintaining cache coherence. [Familiarity]
8. Describe the key performance challenges in different memory and distributed system topologies. [Familiarity]

PD/Parallel Performance

[Elective]

Topics:

- Load balancing
- Performance measurement
- Scheduling and contention (cross-reference OS/Scheduling and Dispatch)
- Evaluating communication overhead
- Data management
    - Non-uniform communication costs due to proximity (cross-reference SF/Proximity)
    - Cache effects (e.g., false sharing)
    - Maintaining spatial locality
- Power usage and management

Learning outcomes:

1. Detect and correct a load imbalance. [Usage]
2. Calculate the implications of Amdahl's law for a particular parallel algorithm (cross-reference SF/Evaluation for Amdahl's Law). [Usage]
3. Describe how data distribution/layout can affect an algorithm's communication costs. [Familiarity]
4. Detect and correct an instance of false sharing. [Usage]
5. Explain the impact of scheduling on parallel performance. [Familiarity]
6. Explain performance impacts of data locality. [Familiarity]
7. Explain the impact and trade-off related to power usage on parallel performance. [Familiarity]

PD/Distributed Systems

[Elective]

Topics:

- Faults (cross-reference OS/Fault Tolerance)
    - Network-based (including partitions) and node-based failures
    - Impact on system-wide guarantees (e.g., availability)
- Distributed message sending
    - Data conversion and transmission
    - Sockets
    - Message sequencing
    - Buffering, retrying, and dropping messages
- Distributed system design tradeoffs
    - Latency versus throughput
    - Consistency, availability, partition tolerance
- Distributed service design
    - Stateful versus stateless protocols and services
    - Session (connection-based) designs
    - Reactive (IO-triggered) and multithreaded designs
- Core distributed algorithms
    - Election, discovery

Learning outcomes:

1. Distinguish network faults from other kinds of failures. [Familiarity]
2. Explain why synchronization constructs such as simple locks are not useful in the presence of distributed faults. [Familiarity]
3. Write a program that performs any required marshaling and conversion into message units, such as packets, to communicate interesting data between two hosts. [Usage]
4. Measure the observed throughput and response latency across hosts in a given network. [Usage]
5. Explain why no distributed system can be simultaneously consistent, available, and partition tolerant. [Familiarity]
6. Implement a simple server -- for example, a spell checking service. [Usage]
7. Explain the tradeoffs among overhead, scalability, and fault tolerance when choosing a stateful v. stateless design for a given service. [Familiarity]
8. Describe the scalability challenges associated with a service growing to accommodate many clients, as well as those associated with a service only transiently having many clients. [Familiarity]
9. Give examples of problems for which consensus algorithms such as leader election are required. [Usage]

PD/Cloud Computing

[Elective]

Topics:

- Internet-Scale computing
    - Task partitioning (cross-reference PD/Parallel Algorithms, Analysis, and Programming)
    - Data access
    - Clusters, grids, and meshes
- Cloud services
    - Infrastructure as a service
        - Elasticity of resources
        - Platform APIs

    - Software as a service
    - Security
    - Cost management
- Virtualization (cross-reference SF/Virtualization and Isolation and OS/Virtual Machines)
    - Shared resource management
    - Migration of processes
- Cloud-based data storage
    - Shared access to weakly consistent data stores
    - Data synchronization
    - Data partitioning
    - Distributed file systems (cross-reference IM/Distributed Databases)
    - Replication

Learning outcomes:

1. Discuss the importance of elasticity and resource management in cloud computing. [Familiarity]
2. Explain strategies to synchronize a common view of shared data across a collection of devices. [Familiarity]
3. Explain the advantages and disadvantages of using virtualized infrastructure. [Familiarity]
4. Deploy an application that uses cloud infrastructure for computing and/or data resources. [Usage]
5. Appropriately partition an application between a client and resources. [Usage]

PD/Formal Models and Semantics

[Elective]

Topics:

- Formal models of processes and message passing, including algebras such as Communicating Sequential Processes (CSP) and pi-calculus
- Formal models of parallel computation, including the Parallel Random Access Machine (PRAM) and alternatives such as Bulk Synchronous Parallel (BSP)
- Formal models of computational dependencies
- Models of (relaxed) shared memory consistency and their relation to programming language specifications
- Algorithmic correctness criteria including linearizability
- Models of algorithmic progress, including non-blocking guarantees and fairness
- Techniques for specifying and checking correctness properties such as atomicity and freedom from data races

Learning outcomes:

1. Model a concurrent process using a formal model, such as pi-calculus. [Usage]
2. Explain the characteristics of a particular formal parallel model. [Familiarity]
3. Formally model a shared memory system to show if it is consistent. [Usage]
4. Use a model to show progress guarantees in a parallel algorithm. [Usage]
5. Use formal techniques to show that a parallel algorithm is correct with respect to a safety or liveness property. [Usage]
6. Decide if a specific execution is linearizable or not. [Usage]

Programming Languages (PL)

Programming languages are the medium through which programmers precisely describe concepts, formulate algorithms, and reason about solutions. In the course of a career, a computer scientist will work with many different languages, separately or together. Software developers must understand the programming models underlying different languages and make informed design choices in languages supporting multiple complementary approaches. Computer scientists will often need to learn new languages and programming constructs, and must understand the principles underlying how programming language features are defined, composed, and implemented. The effective use of programming languages, and appreciation of their limitations, also requires a basic knowledge of programming language translation and static program analysis, as well as run-time components such as memory management.

PL. Programming Languages (8 Core-Tier1 hours, 20 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
PL/Object-Oriented Programming	**4**	**6**	**N**
PL/Functional Programming	**3**	**4**	**N**
PL/Event-Driven and Reactive Programming		**2**	**N**
PL/Basic Type Systems	**1**	**4**	**N**
PL/Program Representation		**1**	**N**
PL/Language Translation and Execution		**3**	**N**
PL/Syntax Analysis			**Y**
PL/Compiler Semantic Analysis			**Y**
PL/Code Generation			**Y**
PL/Runtime Systems			**Y**
PL/Static Analysis			**Y**
PL/Advanced Programming Constructs			**Y**
PL/Concurrency and Parallelism			**Y**
PL/Type Systems			**Y**
PL/Formal Semantics			**Y**
PL/Language Pragmatics			**Y**
PL/Logic Programming			**Y**

Note:

- Some topics from one or more of the first three Knowledge Units (*Object-Oriented Programming*, *Functional Programming*, *Event-Driven and Reactive Programming*) are likely to be integrated with topics in the SF-Software Development Fundamentals Knowledge Area in a curriculum's introductory courses. Curricula will differ on which topics are integrated in this fashion and which are delayed until later courses on software development and programming languages.
- Some of the most important core learning outcomes are relevant to object-oriented programming, functional programming, and, in fact, all programming. These learning outcomes are *repeated* in the *Object-Oriented Programming* and *Functional Programming* Knowledge Units, with a note to this effect. We do not intend that a

curriculum necessarily needs to cover them multiple times, though some will. We repeat them only because they do not naturally fit in only one Knowledge Unit.

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/Functional Programming

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

Topics:

[Core-Tier1]

- Effect-free programming
  - Function calls have no side effects, facilitating compositional reasoning
  - Variables are immutable, preventing unexpected changes to program data by other code
  - Data can be freely aliased or copied without introducing unintended effects from mutation
- Processing structured data (e.g., trees) via functions with cases for each data variant
  - Associated language constructs such as discriminated unions and pattern-matching over them
  - Functions defined over compound data in terms of functions applied to the constituent pieces
- First-class functions (taking, returning, and storing functions)

[Core-Tier2]

- Function closures (functions using variables in the enclosing lexical environment)
  - Basic meaning and definition -- creating closures at run-time by capturing the environment
  - Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments
  - Using a closure to encapsulate data in its environment
  - Currying and partial application
- Defining higher-order operations on aggregates, especially map, reduce/fold, and filter

Learning outcomes:

[Core-Tier1]

1. Write basic algorithms that avoid assigning to mutable state or considering reference equality. [Usage]
2. Write useful functions that take and return other functions. [Usage]
3. 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/Object-Oriented Programming.*

[Core-Tier2]

4. Correctly reason about variables and lexical scope in a program using function closures. [Usage]
5. Use functional encapsulation mechanisms such as closures and modular interfaces. [Usage]
6. 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/Object-Oriented 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]

PL/Basic Type Systems

[1 Core-Tier1 hour, 4 Core-Tier2 hours]

The core-tier2 hours would be profitably spent both on the core-tier2 topics and on a less shallow treatment of the core-tier1 topics and learning outcomes.

Topics:

[Core-Tier1]

- A type as a set of values together with a set of operations
    - Primitive types (e.g., numbers, Booleans)
    - Compound types built from other types (e.g., records, unions, arrays, lists, functions, references)
- Association of types to variables, arguments, results, and fields
- Type safety and errors caused by using values inconsistently given their intended types
- Goals and limitations of static typing
    - Eliminating some classes of errors without running the program
    - Undecidability means static analysis must conservatively approximate program behavior

[Core-Tier2]

- Generic types (parametric polymorphism)
    - Definition
    - Use for generic libraries such as collections
    - Comparison with ad hoc polymorphism (overloading) and subtype polymorphism
- Complementary benefits of static and dynamic typing
    - Errors early vs. errors late/avoided

- Enforce invariants during code development and code maintenance vs. postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections
- Avoid misuse of code vs. allow more code reuse
- Detect incomplete programs vs. allow incomplete programs to run

Learning outcomes:

[Core-Tier1]

1. For both a primitive and a compound type, informally describe the values that have that type. [Familiarity]
2. For a language with a static type system, describe the operations that are forbidden statically, such as passing the wrong type of value to a function or method. [Familiarity]
3. Describe examples of program errors detected by a type system. [Familiarity]
4. For multiple programming languages, identify program properties checked statically and program properties checked dynamically. [Usage]
5. Give an example program that does not type-check in a particular language and yet would have no error if run. [Familiarity]
6. Use types and type-error messages to write and debug programs. [Usage]

[Core-Tier2]

7. Explain how typing rules define the set of operations that are legal for a type. [Familiarity]
8. Write down the type rules governing the use of a particular compound type. [Usage]
9. Explain why undecidability requires type systems to conservatively approximate program behavior. [Familiarity]
10. Define and use program pieces (such as functions, classes, methods) that use generic types, including for collections. [Usage]
11. Discuss the differences among generics, subtyping, and overloading. [Familiarity]
12. Explain multiple benefits and limitations of static typing in writing, maintaining, and debugging software. [Familiarity]

PL/Program Representation

[1 Core-Tier2 hour]

Topics:

- Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators
- Abstract syntax trees; contrast with concrete syntax
- Data structures to represent code for execution, translation, or transmission

Learning outcomes:

1. Explain how programs that process other programs treat the other programs as their input data. [Familiarity]
2. Describe an abstract syntax tree for a small language. [Usage]
3. Describe the benefits of having program representations other than strings of source code. [Familiarity]
4. Write a program to process some representation of code for some purpose, such as an interpreter, an expression optimizer, or a documentation generator. [Usage]

PL/Language Translation and Execution

[3 Core-Tier2 hours]

Topics:

- Interpretation vs. compilation to native code vs. compilation to portable intermediate representation
- Language translation pipeline: parsing, optional type-checking, translation, linking, execution
    - Execution as native code or within a virtual machine
    - Alternatives like dynamic loading and dynamic (or "just-in-time") code generation
- Run-time representation of core language constructs such as objects (method tables) and first-class functions (closures)
- Run-time layout of memory: call-stack, heap, static data
    - Implementing loops, recursion, and tail calls
- Memory management
    - Manual memory management: allocating, de-allocating, and reusing heap memory
    - Automated memory management: garbage collection as an automated technique using the notion of reachability

Learning outcomes:

1. Distinguish a language definition (what constructs mean) from a particular language implementation (compiler vs. interpreter, run-time representation of data objects, etc.). [Familiarity]
2. Distinguish syntax and parsing from semantics and evaluation. [Familiarity]
3. Sketch a low-level run-time representation of core language constructs, such as objects or closures. [Familiarity]
4. Explain how programming language implementations typically organize memory into global data, text, heap, and stack sections and how features such as recursion and memory management map to this memory model. [Familiarity]
5. Identify and fix memory leaks and dangling-pointer dereferences. [Usage]
6. Discuss the benefits and limitations of garbage collection, including the notion of reachability. [Familiarity]

PL/Syntax Analysis

[Elective]

Topics:

- Scanning (lexical analysis) using regular expressions
- Parsing strategies including top-down (e.g., recursive descent, Earley parsing, or LL) and bottom-up (e.g., backtracking or LR) techniques; role of context-free grammars
- Generating scanners and parsers from declarative specifications

Learning outcomes:

1. Use formal grammars to specify the syntax of languages. [Usage]
2. Use declarative tools to generate parsers and scanners. [Usage]
3. Identify key issues in syntax definitions: ambiguity, associativity, precedence. [Familiarity]

PL/Compiler Semantic Analysis

[Elective]

Topics:

- High-level program representations such as abstract syntax trees
- Scope and binding resolution
- Type checking
- Declarative specifications such as attribute grammars

Learning outcomes:

1. Implement context-sensitive, source-level static analyses such as type-checkers or resolving identifiers to identify their binding occurrences. [Usage]
2. Describe semantic analyses using an attribute grammar. [Usage]

PL/Code Generation

[Elective]

Topics:

- Procedure calls and method dispatching
- Separate compilation; linking
- Instruction selection
- Instruction scheduling
- Register allocation
- Peephole optimization

Learning outcomes:

1. Identify all essential steps for automatically converting source code into assembly or other low-level languages. [Familiarity]
2. Generate the low-level code for calling functions/methods in modern languages. [Usage]
3. Discuss why separate compilation requires uniform calling conventions. [Familiarity]
4. Discuss why separate compilation limits optimization because of unknown effects of calls. [Familiarity]
5. Discuss opportunities for optimization introduced by naive translation and approaches for achieving optimization, such as instruction selection, instruction scheduling, register allocation, and peephole optimization. [Familiarity]

PL/Runtime Systems

[Elective]

Topics:

- Dynamic memory management approaches and techniques: malloc/free, garbage collection (mark-sweep, copying, reference counting), regions (also known as arenas or zones)

- Data layout for objects and activation records
- Just-in-time compilation and dynamic recompilation
- Other common features of virtual machines, such as class loading, threads, and security.

Learning outcomes:

1. Compare the benefits of different memory-management schemes, using concepts such as fragmentation, locality, and memory overhead. [Familiarity]
2. Discuss benefits and limitations of automatic memory management. [Familiarity]
3. Explain the use of metadata in run-time representations of objects and activation records, such as class pointers, array lengths, return addresses, and frame pointers. [Familiarity]
4. Discuss advantages, disadvantages, and difficulties of just-in-time and dynamic recompilation. [Familiarity]
5. Identify the services provided by modern language run-time systems. [Familiarity]

PL/Static Analysis

[Elective]

Topics:

- Relevant program representations, such as basic blocks, control-flow graphs, def-use chains, and static single assignment
- Undecidability and consequences for program analysis
- Flow-insensitive analyses, such as type-checking and scalable pointer and alias analyses
- Flow-sensitive analyses, such as forward and backward dataflow analyses
- Path-sensitive analyses, such as software model checking
- Tools and frameworks for defining analyses
- Role of static analysis in program optimization
- Role of static analysis in (partial) verification and bug-finding

Learning outcomes:

1. Define useful static analyses in terms of a conceptual framework such as dataflow analysis. [Usage]
2. Explain why non-trivial sound static analyses must be approximate. [Familiarity]
3. Communicate why an analysis is correct (sound and terminating). [Usage]
4. Distinguish “may” and “must” analyses. [Familiarity]
5. Explain why potential aliasing limits sound program analysis and how alias analysis can help. [Familiarity]
6. Use the results of a static analysis for program optimization and/or partial program correctness. [Usage]

PL/Advanced Programming Constructs

[Elective]

Topics:

- Lazy evaluation and infinite streams
- Control Abstractions: Exception Handling, Continuations, Monads
- Object-oriented abstractions: Multiple inheritance, Mixins, Traits, Multimethods

- Metaprogramming: Macros, Generative programming, Model-based development
- Module systems
- String manipulation via pattern-matching (regular expressions)
- Dynamic code evaluation (“eval”)
- Language support for checking assertions, invariants, and pre/post-conditions

Learning outcomes:

1. Use various advanced programming constructs and idioms correctly. [Usage]
2. Discuss how various advanced programming constructs aim to improve program structure, software quality, and programmer productivity. [Familiarity]
3. Discuss how various advanced programming constructs interact with the definition and implementation of other language features. [Familiarity]

PL/Concurrency and Parallelism

[Elective]

Support for concurrency is a fundamental programming-languages issue with rich material in programming language design, language implementation, and language theory. Due to coverage in other Knowledge Areas, this elective Knowledge Unit aims only to complement the material included elsewhere in the Body of Knowledge. Courses on programming languages are an excellent place to include a general treatment of concurrency including this other material.

Cross-reference PD/Parallel and Distributed Computing, SF/Parallelism.

Topics:

- Constructs for thread-shared variables and shared-memory synchronization
- Actor models
- Futures
- Language support for data parallelism
- Models for passing messages between sequential processes
- Effect of memory-consistency models on language semantics and correct code generation

Learning outcomes:

1. Write correct concurrent programs using multiple programming models, such as shared memory, actors, futures, and data-parallelism primitives. [Usage]
2. Use a message-passing model to analyze a communication protocol. [Usage]
3. Explain why programming languages do not guarantee sequential consistency in the presence of data races and what programmers must do as a result. [Familiarity]

PL/Type Systems

[Elective]

Topics:

- Compositional type constructors, such as product types (for aggregates), sum types (for unions), function types, quantified types, and recursive types
- Type checking
- Type safety as preservation plus progress
- Type inference
- Static overloading

Learning outcomes:

1. Define a type system precisely and compositionally. [Usage]
2. For various foundational type constructors, identify the values they describe and the invariants they enforce. [Familiarity]
3. Precisely specify the invariants preserved by a sound type system. [Familiarity]
4. Prove type safety for a simple language in terms of preservation and progress theorems. [Usage]
5. Implement a unification-based type-inference algorithm for a simple language. [Usage]
6. Explain how static overloading and associated resolution algorithms influence the dynamic behavior of programs. [Familiarity]

PL/Formal Semantics

[Elective]

Topics:

- Syntax vs. semantics
- Lambda Calculus
- Approaches to semantics: Operational, Denotational, Axiomatic
- Proofs by induction over language semantics
- Formal definitions and proofs for type systems (cross-reference PL/Type Systems)
- Parametricity (cross-reference PL/Type Systems)
- Using formal semantics for systems modeling

Learning outcomes:

1. Give a formal semantics for a small language. [Usage]
2. Write a lambda-calculus program and show its evaluation to a normal form. [Usage]
3. Discuss the different approaches of operational, denotational, and axiomatic semantics. [Familiarity]
4. Use induction to prove properties of all programs in a language. [Usage]
5. Use induction to prove properties of all programs in a language that are well-typed according to a formally defined type system. [Usage]
6. Use parametricity to establish the behavior of code given only its type. [Usage]
7. Use formal semantics to build a formal model of a software system other than a programming language. [Usage]

PL/Language Pragmatics

[Elective]

Topics:

- Principles of language design such as orthogonality
- Evaluation order, precedence, and associativity
- Eager vs. delayed evaluation
- Defining control and iteration constructs
- External calls and system libraries

Learning outcomes:

1. Discuss the role of concepts such as orthogonality and well-chosen defaults in language design. [Familiarity]
2. Use crisp and objective criteria for evaluating language-design decisions. [Usage]
3. Give an example program whose result can differ under different rules for evaluation order, precedence, or associativity. [Usage]
4. Show uses of delayed evaluation, such as user-defined control abstractions. [Familiarity]
5. Discuss the need for allowing calls to external calls and system libraries and the consequences for language implementation. [Familiarity]

PL/Logic Programming

[Elective]

Topics:

- Clausal representation of data structures and algorithms
- Unification
- Backtracking and search
- Cuts

Learning outcomes:

1. Use a logic language to implement a conventional algorithm. [Usage]
2. Use a logic language to implement an algorithm employing implicit search using clauses, relations, and cuts. [Usage]

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]

Software Engineering (SE)

In every computing application domain, professionalism, quality, schedule, and cost are critical to producing software systems. Because of this, the elements of software engineering are applicable to developing software in all areas of computing. A wide variety of software engineering practices have been developed and utilized since the need for a discipline of software engineering was first recognized. Many trade-offs between these different practices have also been identified. Practicing software engineers have to select and apply appropriate techniques and practices to a given development effort in order to maximize value. To learn how to do so, they study the elements of software engineering.

Software engineering is the discipline concerned with the application of theory, knowledge, and practice to effectively and efficiently build reliable software systems that satisfy the requirements of customers and users. This discipline is applicable to small, medium, and large-scale systems. It encompasses all phases of the lifecycle of a software system, including requirements elicitation, analysis and specification; design; construction; verification and validation; deployment; and operation and maintenance. Whether small or large, following a traditional plan-driven development process, an agile approach, or some other method, software engineering is concerned with the best way to build good software systems.

Software engineering uses engineering methods, processes, techniques, and measurements. It benefits from the use of tools for managing software development; analyzing and modeling software artifacts; assessing and controlling quality; and for ensuring a disciplined, controlled approach to software evolution and reuse. The software engineering toolbox has evolved over the years. For instance, the use of contracts, with requires and ensure clauses and class invariants, is one good practice that has become more common. Software development, which can involve an individual developer or a team or teams of developers, requires choosing the most appropriate tools, methods, and approaches for a given development environment.

Students and instructors need to understand the impacts of specialization on software engineering approaches. For example, specialized systems include:

- Real time systems
- Client-server systems
- Distributed systems
- Parallel systems
- Web-based systems
- High integrity systems
- Games
- Mobile computing
- Domain specific software (e.g., scientific computing or business applications)

Issues raised by each of these specialized systems demand specific treatments in each phase of software engineering. Students must become aware of the differences between general software engineering techniques and principles and the techniques and principles needed to address issues specific to specialized systems.

An important effect of specialization is that different choices of material may need to be made when teaching applications of software engineering, such as between different process models, different approaches to modeling systems, or different choices of techniques for carrying out any of the key activities. This is reflected in the assignment of core and elective material, with the core topics and learning outcomes focusing on the principles underlying the various choices, and the details of the various alternatives from which the choices have to be made being assigned to the elective material.

Another division of the practices of software engineering is between those concerned with the fundamental need to develop systems that implement correctly the functionality that is required for them and those concerned with other qualities for systems and the trade-offs needed to balance these qualities. This division too is reflected in the assignment of core and elective material, so that topics and learning outcomes concerned with the basic methods for developing such system are assigned to the core and those that are concerned with other qualities and trade-offs between them are assigned to the elective material.

In general, students can best learn to apply much of the material defined in the Sofware Engineering KA by participating in a project. Such projects should require students to work on a team to develop a software system through as much of its lifecycle as is possible. Much of software engineering is devoted to effective communication among team members and stakeholders. Utilizing project teams, projects can be sufficiently challenging to require students to use effective software engineering techniques and to develop and practice their communication skills. While organizing and running effective projects within the academic framework can be challenging, the best way to learn to apply software engineering theory and knowledge is in the practical environment of a project. The minimum hours specified for some knowledge units in this document may appear insufficient to accomplish associated application-level learning outcomes. It should be understood that these outcomes are to be achieved through project experience that may even occur later in the curriculum than when the topics within the knowledge unit are introduced.

Further, there is increasing evidence that students learn to apply software engineering principles more effectively through an iterative approach, where students have the opportunity to work through a development cycle, assess their work, and then apply the knowledge gained through their assessment to another development cycle. Agile and iterative lifecycle models inherently afford such opportunities.

Software lifecycle terminology in this document is based on that used in earlier sources, such as the Software Engineering Body of Knowledge (SWEBOK) and the ACM/IEEE-CS Software Engineering 2004 Curriculum Guidelines (SE2004). While some terms were originally defined in the context of plan-driven development processes, they are treated here as generic, and thus equally applicable to agile processes.

Note: The SDF/Development Methods knowledge unit includes 9 Core-Tier1 hours that constitute an introduction to certain aspects of software engineering. The knowledge units, topics and core hour specifications in this Software Engineering Knowledge Area must be understood as assuming previous exposure to the material described in SDF/Development Methods.

SE. Software Engineering (6 Core-Tier1 hours; 21 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
SE/Software Processes	**2**	**1**	**Y**
SE/Software Project Management		**2**	**Y**
SE/Tools and Environments		**2**	**N**
SE/Requirements Engineering	**1**	**3**	**Y**
SE/Software Design	**3**	**5**	**Y**
SE/Software Construction		**2**	**Y**
SE/Software Verification and Validation		**4**	**Y**
SE/Software Evolution		**2**	**Y**
SE/Software Reliability		**1**	**Y**
SE/Formal Methods			**Y**

SE/Software Processes

[2 Core-Tier1 hours; 1 Core-Tier2 hour]

Topics:

[Core-Tier1]

- Systems level considerations, i.e., the interaction of software with its intended environment (cross-reference IAS/Secure Software Engineering)
- Introduction to software process models (e.g., waterfall, incremental, agile)
  - Activities within software lifecycles
- Programming in the large vs. individual programming

[Core-Tier2]

- Evaluation of software process models

[Elective]

- Software quality concepts
- Process improvement
- Software process capability maturity models
- Software process measurements

Learning Outcomes:

[Core-Tier1]

1. Describe how software can interact with and participate in various systems including information management, embedded, process control, and communications systems. [Familiarity]
2. Describe the relative advantages and disadvantages among several major process models (e.g., waterfall, iterative, and agile). [Familiarity]
3. Describe the different practices that are key components of various process models. [Familiarity]
4. Differentiate among the phases of software development. [Familiarity]
5. Describe how programming in the large differs from individual efforts with respect to understanding a large code base, code reading, understanding builds, and understanding context of changes. [Familiarity]

[Core-Tier2]

6. Explain the concept of a software lifecycle and provide an example, illustrating its phases including the deliverables that are produced. [Familiarity]
7. Compare several common process models with respect to their value for development of particular classes of software systems taking into account issues such as requirement stability, size, and non-functional characteristics. [Usage]

[Elective]

8. Define software quality and describe the role of quality assurance activities in the software process. [Familiarity]
9. Describe the intent and fundamental similarities among process improvement approaches. [Familiarity]
10. Compare several process improvement models such as CMM, CMMI, CQI, Plan-Do-Check-Act, or ISO9000. [Assessment]
11. Assess a development effort and recommend potential changes by participating in process improvement (using a model such as PSP) or engaging in a project retrospective. [Usage]
12. Explain the role of process maturity models in process improvement. [Familiarity]
13. Describe several process metrics for assessing and controlling a project. [Familiarity]
14. Use project metrics to describe the current state of a project. [Usage]

SE/Software Project Management

[2 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Team participation
    - Team processes including responsibilities for tasks, meeting structure, and work schedule
    - Roles and responsibilities in a software team
    - Team conflict resolution
    - Risks associated with virtual teams (communication, perception, structure)
- Effort Estimation (at the personal level)
- Risk (cross reference IAS/Secure Software Engineering)
    - The role of risk in the lifecycle
    - Risk categories including security, safety, market, financial, technology, people, quality, structure and process

[Elective]

- Team management
    - Team organization and decision-making

    - Role identification and assignment
    - Individual and team performance assessment
- Project management
    - Scheduling and tracking
    - Project management tools
    - Cost/benefit analysis
- Software measurement and estimation techniques
- Software quality assurance and the role of measurements
- Risk
    - Risk identification and management
    - Risk analysis and evaluation
    - Risk tolerance (e.g., risk-adverse, risk-neutral, risk-seeking)
    - Risk planning
- System-wide approach to risk including hazards associated with tools

Learning Outcomes:

[Core-Tier2]

1. Discuss common behaviors that contribute to the effective functioning of a team. [Familiarity]
2. Create and follow an agenda for a team meeting. [Usage]
3. Identify and justify necessary roles in a software development team. [Usage]
4. Understand the sources, hazards, and potential benefits of team conflict. [Usage]
5. Apply a conflict resolution strategy in a team setting. [Usage]
6. Use an *ad hoc* method to estimate software development effort (e.g., time) and compare to actual effort required. [Usage]
7. List several examples of software risks. [Familiarity]
8. Describe the impact of risk in a software development lifecycle. [Familiarity]
9. Describe different categories of risk in software systems. [Familiarity]

[Elective]

10. Demonstrate through involvement in a team project the central elements of team building and team management. [Usage]
11. Describe how the choice of process model affects team organizational structures and decision-making processes. [Familiarity]
12. Create a team by identifying appropriate roles and assigning roles to team members. [Usage]
13. Assess and provide feedback to teams and individuals on their performance in a team setting. [Usage]
14. Using a particular software process, describe the aspects of a project that need to be planned and monitored, (e.g., estimates of size and effort, a schedule, resource allocation, configuration control, change management, and project risk identification and management). [Familiarity]
15. Track the progress of some stage in a project using appropriate project metrics. [Usage]
16. Compare simple software size and cost estimation techniques. [Usage]
17. Use a project management tool to assist in the assignment and tracking of tasks in a software development project. [Usage]
18. Describe the impact of risk tolerance on the software development process. [Assessment]
19. Identify risks and describe approaches to managing risk (avoidance, acceptance, transference, mitigation), and characterize the strengths and shortcomings of each. [Familiarity]
20. Explain how risk affects decisions in the software development process. [Usage]
21. Identify security risks for a software system. [Usage]
22. Demonstrate a systematic approach to the task of identifying hazards and risks in a particular situation. [Usage]
23. Apply the basic principles of risk management in a variety of simple scenarios including a security situation. [Usage]
24. Conduct a cost/benefit analysis for a risk mitigation approach. [Usage]
25. Identify and analyze some of the risks for an entire system that arise from aspects other than the software. [Usage]

SE/Tools and Environments

[2 Core-Tier2 hours]

Topics:

- Software configuration management and version control
- Release management
- Requirements analysis and design modeling tools
- Testing tools including static and dynamic analysis tools
- Programming environments that automate parts of program construction processes (e.g., automated builds)
    - Continuous integration
- Tool integration concepts and mechanisms

Learning Outcomes:

1. Describe the difference between centralized and distributed software configuration management. [Familiarity]
2. Describe how version control can be used to help manage software release management. [Familiarity]
3. Identify configuration items and use a source code control tool in a small team-based project. [Usage]
4. Describe how available static and dynamic test tools can be integrated into the software development environment. [Familiarity]
5. Describe the issues that are important in selecting a set of tools for the development of a particular software system, including tools for requirements tracking, design modeling, implementation, build automation, and testing. [Familiarity]
6. Demonstrate the capability to use software tools in support of the development of a software product of medium size. [Usage]

SE/Requirements Engineering

[1 Core-Tier1 hour; 3 Core-Tier2 hours]

The purpose of requirements engineering is to develop a common understanding of the needs, priorities, and constraints relevant to a software system. Many software failures arise from an incomplete understanding of requirements for the software to be developed or inadequate management of those requirements.

Specifications of requirements range in formality from completely informal (e.g., spoken) to rigorously mathematical (e.g., written in a formal specification language such as Z or first-order logic). In practice, successful software engineering efforts use requirements specifications to reduce ambiguity and improve the consistency and completeness of the development team's understanding of the vision of the intended software. Plan-driven approaches tend to produce formal documents with numbered requirements. Agile approaches tend to favor less formal specifications that include user stories, use cases, and test cases.

Topics:

[Core-Tier1]

- Describing functional requirements using, for example, use cases or users stories
- Properties of requirements including consistency, validity, completeness, and feasibility

[Core-Tier2]

- Software requirements elicitation
- Describing system data using, for example, class diagrams or entity-relationship diagrams
- Non-functional requirements and their relationship to software quality (cross-reference IAS/Secure Software Engineering)
- Evaluation and use of requirements specifications

[Elective]

- Requirements analysis modeling techniques
- Acceptability of certainty / uncertainty considerations regarding software / system behavior
- Prototyping
- Basic concepts of formal requirements specification
- Requirements specification
- Requirements validation
- Requirements tracing

Learning Outcomes:

[Core-Tier1]

1. List the key components of a use case or similar description of some behavior that is required for a system. [Familiarity]
2. Describe how the requirements engineering process supports the elicitation and validation of behavioral requirements. [Familiarity]
3. Interpret a given requirements model for a simple software system. [Familiarity]

[Core-Tier2]

4. Describe the fundamental challenges of and common techniques used for requirements elicitation. [Familiarity]
5. List the key components of a data model (e.g., class diagrams or ER diagrams). [Familiarity]
6. Identify both functional and non-functional requirements in a given requirements specification for a software system. [Usage]
7. Conduct a review of a set of software requirements to determine the quality of the requirements with respect to the characteristics of good requirements. [Usage]

[Elective]

8. Apply key elements and common methods for elicitation and analysis to produce a set of software requirements for a medium-sized software system. [Usage]
9. Compare the plan-driven and agile approaches to requirements specification and validation and describe the benefits and risks associated with each. [Familiarity]
10. Use a common, non-formal method to model and specify the requirements for a medium-size software system. [Usage]
11. Translate into natural language a software requirements specification (e.g., a software component contract) written in a formal specification language. [Usage]
12. Create a prototype of a software system to mitigate risk in requirements. [Usage]
13. Differentiate between forward and backward tracing and explain their roles in the requirements validation process. [Familiarity]

SE/Software Design

[3 Core-Tier1 hours; 5 Core-Tier2 hours]

Topics:

[Core-Tier1]

- System design principles: levels of abstraction (architectural design and detailed design), separation of concerns, information hiding, coupling and cohesion, re-use of standard structures
- Design Paradigms such as structured design (top-down functional decomposition), object-oriented analysis and design, event driven design, component-level design, data-structured centered, aspect oriented, function oriented, service oriented
- Structural and behavioral models of software designs
- Design patterns

[Core-Tier2]

- Relationships between requirements and designs: transformation of models, design of contracts, invariants
- Software architecture concepts and standard architectures (e.g. client-server, n-layer, transform centered, pipes-and-filters)
- Refactoring designs using design patterns
- The use of components in design: component selection, design, adaptation and assembly of components, components and patterns, components and objects (for example, building a GUI using a standard widget set)

[Elective]

- Internal design qualities, and models for them: efficiency and performance, redundancy and fault tolerance, traceability of requirements
- External design qualities, and models for them: functionality, reliability, performance and efficiency, usability, maintainability, portability
- Measurement and analysis of design quality
- Tradeoffs between different aspects of quality
- Application frameworks
- Middleware: the object-oriented paradigm within middleware, object request brokers and marshalling, transaction processing monitors, workflow systems
- Principles of secure design and coding (cross-reference IAS/Principles of Secure Design)
    - Principle of least privilege
    - Principle of fail-safe defaults
    - Principle of psychological acceptability

Learning Outcomes:

[Core-Tier1]

1. Articulate design principles including separation of concerns, information hiding, coupling and cohesion, and encapsulation. [Familiarity]
2. Use a design paradigm to design a simple software system, and explain how system design principles have been applied in this design. [Usage]
3. Construct models of the design of a simple software system that are appropriate for the paradigm used to design it. [Usage]
4. Within the context of a single design paradigm, describe one or more design patterns that could be applicable to the design of a simple software system. [Familiarity]

[Core-Tier2]

5. For a simple system suitable for a given scenario, discuss and select an appropriate design paradigm. [Usage]
6. Create appropriate models for the structure and behavior of software products from their requirements specifications. [Usage]
7. Explain the relationships between the requirements for a software product and its design, using appropriate models. [Assessment]
8. For the design of a simple software system within the context of a single design paradigm, describe the software architecture of that system. [Familiarity]
9. Given a high-level design, identify the software architecture by differentiating among common software architectures such as 3-tier, pipe-and-filter, and client-server. [Familiarity]
10. Investigate the impact of software architectures selection on the design of a simple system. [Assessment]
11. Apply simple examples of patterns in a software design. [Usage]
12. Describe a form of refactoring and discuss when it may be applicable. [Familiarity]
13. Select suitable components for use in the design of a software product. [Usage]
14. Explain how suitable components might need to be adapted for use in the design of a software product. [Familiarity]
15. Design a contract for a typical small software component for use in a given system. [Usage]

[Elective]

16. Discuss and select appropriate software architecture for a simple system suitable for a given scenario. [Usage]
17. Apply models for internal and external qualities in designing software components to achieve an acceptable tradeoff between conflicting quality aspects. [Usage]
18. Analyze a software design from the perspective of a significant internal quality attribute. [Assessment]
19. Analyze a software design from the perspective of a significant external quality attribute. [Assessment]
20. Explain the role of objects in middleware systems and the relationship with components. [Familiarity]
21. Apply component-oriented approaches to the design of a range of software, such as using components for concurrency and transactions, for reliable communication services, for database interaction including services for remote query and database management, or for secure communication and access. [Usage]
22. Refactor an existing software implementation to improve some aspect of its design. [Usage]
23. State and apply the principles of least privilege and fail-safe defaults. [Familiarity]

SE/Software Construction

[2 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Coding practices: techniques, idioms/patterns, mechanisms for building quality programs (cross-reference IAS/Defensive Programming; SDF/Development Methods)
  - Defensive coding practices
  - Secure coding practices
  - Using exception handling mechanisms to make programs more robust, fault-tolerant
- Coding standards
- Integration strategies
- Development context: “green field” vs. existing code base
  - Change impact analysis
  - Change actualization

[Elective]

- Potential security problems in programs
    - Buffer and other types of overflows
    - Race conditions
    - Improper initialization, including choice of privileges
    - Checking input
    - Assuming success and correctness
    - Validating assumptions

Learning Outcomes:

[Core-Tier2]

1. Describe techniques, coding idioms and mechanisms for implementing designs to achieve desired properties such as reliability, efficiency, and robustness. [Familiarity]
2. Build robust code using exception handling mechanisms. [Usage]
3. Describe secure coding and defensive coding practices. [Familiarity]
4. Select and use a defined coding standard in a small software project. [Usage]
5. Compare and contrast integration strategies including top-down, bottom-up, and sandwich integration. [Familiarity]
6. Describe the process of analyzing and implementing changes to code base developed for a specific project. [Familiarity]
7. Describe the process of analyzing and implementing changes to a large existing code base. [Familiarity]

[Elective]

8. Rewrite a simple program to remove common vulnerabilities, such as buffer overflows, integer overflows and race conditions. [Usage]
9. Write a software component that performs some non-trivial task and is resilient to input and run-time errors. [Usage]

SE/Software Verification and Validation

[4 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Verification and validation concepts
- Inspections, reviews, audits
- Testing types, including human computer interface, usability, reliability, security, conformance to specification (cross-reference IAS/Secure Software Engineering)
- Testing fundamentals (cross-reference SDF/Development Methods)
    - Unit, integration, validation, and system testing
    - Test plan creation and test case generation
    - Black-box and white-box testing techniques
    - Regression testing and test automation
- Defect tracking
- Limitations of testing in particular domains, such as parallel or safety-critical systems

[Elective]

- Static approaches and dynamic approaches to verification
- Test-driven development
- Validation planning; documentation for validation
- Object-oriented testing; systems testing
- Verification and validation of non-code artifacts (documentation, help files, training materials)
- Fault logging, fault tracking and technical support for such activities
- Fault estimation and testing termination including defect seeding

Learning Outcomes:

[Core-Tier2]

1. Distinguish between program validation and verification. [Familiarity]
2. Describe the role that tools can play in the validation of software. [Familiarity]
3. Undertake, as part of a team activity, an inspection of a medium-size code segment. [Usage]
4. Describe and distinguish among the different types and levels of testing (unit, integration, systems, and acceptance). [Familiarity]
5. Describe techniques for identifying significant test cases for integration, regression and system testing. [Familiarity]
6. Create and document a set of tests for a medium-size code segment. [Usage]
7. Describe how to select good regression tests and automate them. [Familiarity]
8. Use a defect tracking tool to manage software defects in a small software project. [Usage]
9. Discuss the limitations of testing in a particular domain. [Familiarity]

[Elective]

10. Evaluate a test suite for a medium-size code segment. [Usage]
11. Compare static and dynamic approaches to verification. [Familiarity]
12. Identify the fundamental principles of test-driven development methods and explain the role of automated testing in these methods. [Familiarity]
13. Discuss the issues involving the testing of object-oriented software. [Usage]
14. Describe techniques for the verification and validation of non-code artifacts. [Familiarity]
15. Describe approaches for fault estimation. [Familiarity]
16. Estimate the number of faults in a small software application based on fault density and fault seeding. [Usage]
17. Conduct an inspection or review of software source code for a small or medium sized software project. [Usage]

SE/Software Evolution

[2 Core-Tier2 hour]

Topics:

- Software development in the context of large, pre-existing code bases
    - Software change
    - Concerns and concern location
    - Refactoring
- Software evolution
- Characteristics of maintainable software
- Reengineering systems
- Software reuse

    - Code segments
    - Libraries and frameworks
    - Components
    - Product lines

Learning Outcomes:

1. Identify the principal issues associated with software evolution and explain their impact on the software lifecycle. [Familiarity]
2. Estimate the impact of a change request to an existing product of medium size. [Usage]
3. Use refactoring in the process of modifying a software component. [Usage]
4. Discuss the challenges of evolving systems in a changing environment. [Familiarity]
5. Outline the process of regression testing and its role in release management. [Familiarity]
6. Discuss the advantages and disadvantages of different types of software reuse. [Familiarity]

SE/Software Reliability

[1 Core-Tier2]

Topics:

[Core-Tier2]

- Software reliability engineering concepts
- Software reliability, system reliability and failure behavior (cross-reference SF/Reliability Through Redundancy)
- Fault lifecycle concepts and techniques

[Elective]

- Software reliability models
- Software fault tolerance techniques and models
- Software reliability engineering practices
- Measurement-based analysis of software reliability

Learning Outcomes:

[Core-Tier2]

1. Explain the problems that exist in achieving very high levels of reliability. [Familiarity]
2. Describe how software reliability contributes to system reliability. [Familiarity]
3. List approaches to minimizing faults that can be applied at each stage of the software lifecycle. [Familiarity]

[Elective]

4. Compare the characteristics of three different reliability modeling approaches. [Familiarity]
5. Demonstrate the ability to apply multiple methods to develop reliability estimates for a software system. [Usage]
6. Identify methods that will lead to the realization of a software architecture that achieves a specified level of reliability. [Usage]
7. Identify ways to apply redundancy to achieve fault tolerance for a medium-sized application. [Usage]

SE/Formal Methods

[Elective]

The topics listed below have a strong dependency on core material from the Discrete Structures (DS) Knowledge Area, particularly knowledge units DS/Functions Relations and Sets, DS/Basic Logic and DS/Proof Techniques.

***Topics*:**

- Role of formal specification and analysis techniques in the software development cycle
- Program assertion languages and analysis approaches (including languages for writing and analyzing pre- and post-conditions, such as OCL, JML)
- Formal approaches to software modeling and analysis
    - Model checkers
    - Model finders
- Tools in support of formal methods

Learning Outcomes:

1. Describe the role formal specification and analysis techniques can play in the development of complex software and compare their use as validation and verification techniques with testing. [Familiarity]
2. Apply formal specification and analysis techniques to software designs and programs with low complexity. [Usage]
3. Explain the potential benefits and drawbacks of using formal specification languages. [Familiarity]
4. Create and evaluate program assertions for a variety of behaviors ranging from simple through complex. [Usage]
5. Using a common formal specification language, formulate the specification of a simple software system and derive examples of test cases from the specification. [Usage]

Systems Fundamentals (SF)

The underlying hardware and software infrastructure upon which applications are constructed is collectively described by the term "computer systems." Computer systems broadly span the sub-disciplines of operating systems, parallel and distributed systems, communications networks, and computer architecture. Traditionally, these areas are taught in a non-integrated way through independent courses. However these sub-disciplines increasingly share important common fundamental concepts within their respective cores. These concepts include computational paradigms, parallelism, cross-layer communications, state and state transition, resource allocation and scheduling, and so on. The Systems Fundamentals Knowledge Area is designed to present an integrative view of these fundamental concepts in a unified albeit simplified fashion, providing a common foundation for the different specialized mechanisms and policies appropriate to the particular domain area.

SF. Systems Fundamentals. [18 Core-Tier1 hours, 9 Core-Tier2 hours]

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
SF/Computational Paradigms	3		N
SF/Cross-Layer Communications	3		N
SF/State and State Machines	6		N
SF/Parallelism	3		N
SF/Evaluation	3		N
SF/Resource Allocation and Scheduling		2	N
SF/Proximity		3	N
SF/Virtualization and Isolation		2	N
SF/Reliability through Redundancy		2	N
SF/Quantitative Evaluation			Y

SF/Computational Paradigms

[3 Core-Tier1 hours]

The view presented here is the multiple representations of a system across layers, from hardware building blocks to application components, and the parallelism available in each representation. Cross-reference PD/Parallelism Fundamentals.

Topics:

- Basic building blocks and components of a computer (gates, flip-flops, registers, interconnections; Datapath + Control + Memory)
- Hardware as a computational paradigm: Fundamental logic building blocks; Logic expressions, minimization, sum of product forms
- Application-level sequential processing: single thread
- Simple application-level parallel processing: request level (web services/client-server/distributed), single thread per server, multiple threads with multiple servers
- Basic concept of pipelining, overlapped processing stages
- Basic concept of scaling: going faster vs. handling larger problems

Learning Outcomes:

1. List commonly encountered patterns of how computations are organized. [Familiarity]
2. Describe the basic building blocks of computers and their role in the historical development of computer architecture. [Familiarity]
3. Articulate the differences between single thread vs. multiple thread, single server vs. multiple server models, motivated by real world examples (e.g., cooking recipes, lines for multiple teller machines and couples shopping for food). [Familiarity]
4. Articulate the concept of strong vs. weak scaling, i.e., how performance is affected by scale of problem vs. scale of resources to solve the problem. This can be motivated by the simple, real-world examples. [Familiarity]
5. Design a simple logic circuit using the fundamental building blocks of logic design. [Usage]
6. Use tools for capture, synthesis, and simulation to evaluate a logic design. [Usage]
7. Write a simple sequential problem and a simple parallel version of the same program. [Usage]
8. Evaluate performance of simple sequential and parallel versions of a program with different problem sizes, and be able to describe the speed-ups achieved. [Assessment]

SF/Cross-Layer Communications

Cross-reference NC/Introduction, OS/Operating Systems Principles

[3 Core-Tier1 hours]

Topics:

- Programming abstractions, interfaces, use of libraries
- Distinction between Application and OS services, Remote Procedure Call
- Application-Virtual Machine Interaction
- Reliability

Learning Outcomes:

1. Describe how computing systems are constructed of layers upon layers, based on separation of concerns, with well-defined interfaces, hiding details of low layers from the higher layers. [Familiarity]

2. Describe how hardware, VM, OS, and applications are additional layers of interpretation/processing. [Familiarity]
3. Describe the mechanisms of how errors are detected, signaled back, and handled through the layers. [Familiarity]
4. Construct a simple program using methods of layering, error detection and recovery, and reflection of error status across layers. [Usage]
5. Find bugs in a layered program by using tools for program tracing, single stepping, and debugging. [Usage]

SF/State and State Machines

[6 Core-Tier1 hours]

Cross-reference AL/Basic Computability and Complexity, OS/State and State Diagrams, NC/Protocols

Topics:

- Digital vs. Analog/Discrete vs. Continuous Systems
- Simple logic gates, logical expressions, Boolean logic simplification
- Clocks, State, Sequencing
- Combinational Logic, Sequential Logic, Registers, Memories
- Computers and Network Protocols as examples of state machines

Learning Outcomes:

1. Describe computations as a system characyterized by a known set of configurations with transitions from one unique configuration (state) to another (state). [Familiarity]
2. Describe the distinction between systems whose output is only a function of their input (Combinational) and those with memory/history (Sequential). [Familiarity]
3. Describe a computer as a state machine that interprets machine instructions. [Familiarity]
4. Explain how a program or network protocol can also be expressed as a state machine, and that alternative representations for the same computation can exist. [Familiarity]
5. Develop state machine descriptions for simple problem statement solutions (e.g., traffic light sequencing, pattern recognizers). [Usage]
6. Derive time-series behavior of a state machine from its state machine representation. [Assessment]

SF/Parallelism

[3 Core-Tier1 hours]

Cross-reference PD/Parallelism Fundamentals.

***Topics**:*

- Sequential vs. parallel processing
- Parallel programming vs. concurrent programming
- Request parallelism vs. Task parallelism
- Client-Server/Web Services, Thread (Fork-Join), Pipelining
- Multicore architectures and hardware support for synchronization

***Learning Outcomes**:*

1. For a given program, distinguish between its sequential and parallel execution, and the performance implications thereof. [Familiarity]
2. Demonstrate on an execution time line that parallelism events and operations can take place simultaneously (i.e., at the same time). Explain how work can be performed in less elapsed time if this can be exploited. [Familiarity]
3. Explain other uses of parallelism, such as for reliability/redundancy of execution. [Familiarity]
4. Define the differences between the concepts of Instruction Parallelism, Data Parallelism, Thread Parallelism/Multitasking, Task/Request Parallelism. [Familiarity]
5. Write more than one parallel program (e.g., one simple parallel program in more than one parallel programming paradigm; a simple parallel program that manages shared resources through synchronization primitives; a simple parallel program that performs simultaneous operation on partitioned data through task parallel (e.g., parallel search terms; a simple parallel program that performs step-by-step pipeline processing through message passing). [Usage]
6. Use performance tools to measure speed-up achieved by parallel programs in terms of both problem size and number of resources. [Assessment]

SF/Evaluation

[3 Core-Tier1 hours]

Cross-reference PD/Parallel Performance.

***Topics**:*

- Performance figures of merit
- Workloads and representative benchmarks, and methods of collecting and analyzing performance figures of merit
- CPI (Cycles per Instruction) equation as tool for understanding tradeoffs in the design of instruction sets, processor pipelines, and memory system organizations.
- Amdahl's Law: the part of the computation that cannot be sped up limits the effect of the parts that can

***Learning Outcomes**:*

1. Explain how the components of system architecture contribute to improving its performance. [Familiarity]
2. Describe Amdahl's law and discuss its limitations. [Familiarity]
3. Design and conduct a performance-oriented experiment. [Usage]
4. Use software tools to profile and measure program performance. [Assessment]

SF/Resource Allocation and Scheduling

[2 Core-Tier2 hours]

***Topics**:*

- Kinds of resources (e.g., processor share, memory, disk, net bandwidth)
- Kinds of scheduling (e.g., first-come, priority)
- Advantages of fair scheduling, preemptive scheduling

***Learning Outcomes**:*

1. Define how finite computer resources (e.g., processor share, memory, storage and network bandwidth) are managed by their careful allocation to existing entities. [Familiarity]

2. Describe the scheduling algorithms by which resources are allocated to competing entities, and the figures of merit by which these algorithms are evaluated, such as fairness. [Familiarity]
3. Implement simple schedule algorithms. [Usage]
4. Use figures of merit of alternative scheduler implementations. [Assessment]

SF/Proximity

[3 Core-Tier2 hours]

Cross-reference AR/Memory Management, OS/Virtual Memory.

***Topics**:*

- Speed of light and computers (one foot per nanosecond vs. one GHz clocks)
- Latencies in computer systems: memory vs. disk latencies vs. across the network memory
- Caches and the effects of spatial and temporal locality on performance in processors and systems
- Caches and cache coherency in databases, operating systems, distributed systems, and computer architecture
- Introduction into the processor memory hierarchy and the formula for average memory access time

***Learning Outcomes**:*

1. Explain the importance of locality in determining performance. [Familiarity]
2. Describe why things that are close in space take less time to access. [Familiarity]
3. Calculate average memory access time and describe the tradeoffs in memory hierarchy performance in terms of capacity, miss/hit rate, and access time. [Assessment]

SF/Virtualization and Isolation

[2 Core-Tier2 hours]

***Topics**:*

- Rationale for protection and predictable performance
- Levels of indirection, illustrated by virtual memory for managing physical memory resources
- Methods for implementing virtual memory and virtual machines

***Learning Outcomes**:*

1. Explain why it is important to isolate and protect the execution of individual programs and environments that share common underlying resources. [Familiarity]
2. Describe how the concept of indirection can create the illusion of a dedicated machine and its resources even when physically shared among multiple programs and environments. [Familiarity]
3. Measure the performance of two application instances running on separate virtual machines, and determine the effect of performance isolation. [Assessment]

SF/Reliability through Redundancy

[2 Core-Tier2 hours]

***Topics**:*

- Distinction between bugs and faults
- Redundancy through check and retry

- Redundancy through redundant encoding (error correcting codes, CRC, FEC)
- Duplication/mirroring/replicas
- Other approaches to fault tolerance and availability

Learning Outcomes:

1. Explain the distinction between program errors, system errors, and hardware faults (e.g., bad memory) and exceptions (e.g., attempt to divide by zero). [Familiarity]
2. Articulate the distinction between detecting, handling, and recovering from faults, and the methods for their implementation. [Familiarity]
3. Describe the role of error correcting codes in providing error checking and correction techniques in memories, storage, and networks. [Familiarity]
4. Apply simple algorithms for exploiting redundant information for the purposes of data correction. [Usage]
5. Compare different error detection and correction methods for their data overhead, implementation complexity, and relative execution time for encoding, detecting, and correcting errors. [Assessment]

SF/Quantitative Evaluation

[Elective]

Topics:

- Analytical tools to guide quantitative evaluation
- Order of magnitude analysis (Big-Oh notation)
- Analysis of slow and fast paths of a system
- Events on their effect on performance (e.g., instruction stalls, cache misses, page faults)
- Understanding layered systems, workloads, and platforms, their implications for performance, and the challenges they represent for evaluation
- Microbenchmarking pitfalls

Learning Outcomes:

1. Explain the circumstances in which a given figure of system performance metric is useful. [Familiarity]
2. Explain the inadequacies of benchmarks as a measure of system performance. [Familiarity]
3. Use limit studies or simple calculations to produce order-of-magnitude estimates for a given performance metric in a given context. [Usage]
4. Conduct a performance experiment on a layered system to determine the effect of a system parameter on figure of system performance. [Assessment]

Social Issues and Professional Practice (SP)

While technical issues are central to the computing curriculum, they do not constitute a complete educational program in the field. Students must also be exposed to the larger societal context of computing to develop an understanding of the relevant social, ethical, legal and professional issues. This need to incorporate the study of these non-technical issues into the ACM curriculum was formally recognized in 1991, as can be seen from the following excerpt [2]:

> *Undergraduates also need to understand the basic cultural, social, legal, and ethical issues inherent in the discipline of computing. They should understand where the discipline has been, where it is, and where it is heading. They should also understand their individual roles in this process, as well as appreciate the philosophical questions, technical problems, and aesthetic values that play an important part in the development of the discipline.*
>
> *Students also need to develop the ability to ask serious questions about the social impact of computing and to evaluate proposed answers to those questions. Future practitioners must be able to anticipate the impact of introducing a given product into a given environment. Will that product enhance or degrade the quality of life? What will the impact be upon individuals, groups, and institutions?*
>
> *Finally, students need to be aware of the basic legal rights of software and hardware vendors and users, and they also need to appreciate the ethical values that are the basis for those rights. Future practitioners must understand the responsibility that they will bear, and the possible consequences of failure. They must understand their own limitations as well as the limitations of their tools. All practitioners must make a long-term commitment to remaining current in their chosen specialties and in the discipline of computing as a whole.*

As technological advances continue to significantly impact the way we live and work, the critical importance of social issues and professional practice continues to increase; new computer-based products and venues pose ever more challenging problems each year. It is our students who must enter the workforce and academia with intentional regard for the identification and resolution of these problems.

Computer science educators may opt to deliver this core and elective material in stand-alone courses, integrated into traditional technical and theoretical courses, or as special units in capstone and professional practice courses. The material in this familiarity area is best covered through a combination of one required course along with short modules in other courses. On the one hand, some units listed as Core Tier-1 (in particular, Social Context, Analytical Tools, Professional Ethics, and Intellectual Property) do not readily lend themselves to being covered in other traditional courses. Without a standalone course, it is difficult to cover these topics appropriately. On the other hand, if ethical and social considerations are covered only in the standalone course and not "in context," it will reinforce the false notion that technical processes are void of these other relevant issues. Because of this broad relevance, it is important that several traditional courses include modules with case studies that analyze the ethical, legal, social and professional considerations in the context of the technical subject matter of the course. Courses in areas such as software engineering, databases, computer networks, information assurance and security, and introduction to computing provide obvious context for analysis of ethical issues. However, an ethics-related module could be developed for almost any course in the curriculum. It would be explicitly against the spirit of the recommendations to have only a standalone course. Running through all of the issues in this area is the need to speak to the computing practitioner's responsibility to proactively address these issues by both moral and technical actions. The ethical issues discussed in any class should be directly related to and arise naturally from the subject matter of that class. Examples include a discussion in the database course of data aggregation or data mining, or a discussion in the software engineering course of the potential conflicts between obligations to the customer and obligations to the user and others affected by their work. Programming assignments built around applications such as controlling the movement of a laser during eye surgery can help to address the professional, ethical and social impacts of computing. Computing faculty who are unfamiliar with the content and/or pedagogy of applied ethics are urged to take advantage of the considerable resources from ACM, IEEE-CS, SIGCAS (special interest group on computers and society), and other organizations.

It should be noted that the application of ethical analysis underlies every subsection of this Social and Professional knowledge area in computing. The ACM Code of Ethics and Professional Conduct (http://www.acm.org/about/code-of-ethics) provides guidelines that serve as the basis for the conduct of our professional work. The General Moral Imperatives provide an understanding of our commitment to personal responsibility, professional conduct, and our leadership roles.

SP. Social Issues and Professional Practice. [11 Core-Tier1 hours, 5 Core-Tier2 hours]

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
SP/Social Context	1	2	N
SP/Analytical Tools	2		N
SP/Professional Ethics	2	2	N
SP/Intellectual Property	2		Y
SP/Privacy and Civil Liberties	2		Y
SP/Professional Communication	1		Y
SP/Sustainability	1	1	Y
SP/History			Y
SP/Economies of Computing			Y
SP/Security Policies, Laws and Computer Crimes			Y

SP/Social Context

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

Computers and the Internet, perhaps more than any other technologies, have transformed society over the past 75 years, with dramatic increases in human productivity; an explosion of options for news, entertainment, and communication; and fundamental breakthroughs in almost every branch of science and engineering. Social Context provides the foundation for all other SP knowledge units, especially Professional Ethics. Also see cross-referencing with Human-Computer Interaction (HCI) and Networking and Communication (NC) Knowledge Areas.

Topics:

[Core-Tier1]

- Social implications of computing in a networked world (cross-reference HCI/Foundations/social models; IAS/Fundamental Concepts/social issues)
- Impact of social media on individualism, collectivism and culture.

[Core-Tier2]

- Growth and control of the Internet (cross-reference NC/Introduction/organization of the Internet)
- Often referred to as the digital divide, differences in access to digital technology resources and its resulting ramifications for gender, class, ethnicity, geography, and/or underdeveloped countries.
- Accessibility issues, including legal requirements
- Context-aware computing (cross-reference HCI/Design for non-mouse interfaces/ ubiquitous and context-aware)

Learning Outcomes:

[Core-Tier1]

1. Describe positive and negative ways in which computer technology (networks, mobile computing, cloud computing) alters modes of social interaction at the personal level. [Familiarity]
2. Identify developers' assumptions and values embedded in hardware and software design, especially as they pertain to usability for diverse populations including under-represented populations and the disabled. [Familiarity]
3. Interpret the social context of a given design and its implementation. [Familiarity]
4. Evaluate the efficacy of a given design and implementation using empirical data. [Assessment]
5. Summarize the implications of social media on individualism versus collectivism and culture. [Usage]

[Core-Tier2]

6. Discuss how Internet access serves as a liberating force for people living under oppressive forms of government; explain how limits on Internet access are used as tools of political and social repression. [Familiarity]
7. Analyze the pros and cons of reliance on computing in the implementation of democracy (e.g. delivery of social services, electronic voting). [Assessment]
8. Describe the impact of the under-representation of diverse populations in the computing profession (e.g., industry culture, product diversity). [Familiarity]
9. Explain the implications of context awareness in ubiquitous computing systems. [Familiarity]

SP/Analytical Tools

[2 Core-Tier1 hours]

Ethical theories and principles are the foundations of ethical analysis because they are the viewpoints from which guidance can be obtained along the pathway to a decision. Each theory emphasizes different points such as predicting the outcome and following one's duties to others in order to reach an ethically guided decision. However, in order for an ethical theory to be useful, the theory must be directed towards a common set of goals. Ethical principles are the common goals that each theory tries to achieve in order to be successful. These goals include beneficence, least harm, respect for autonomy, and justice.

Topics:

- Ethical argumentation
- Ethical theories and decision-making
- Moral assumptions and values

Learning Outcomes:

1. Evaluate stakeholder positions in a given situation. [Assessment]
2. Analyze basic logical fallacies in an argument. [Assessment]
3. Analyze an argument to identify premises and conclusion. [Assessment]
4. Illustrate the use of example and analogy in ethical argument. [Usage]
5. Evaluate ethical/social tradeoffs in technical decisions. [Assessment]

SP/Professional Ethics

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

Computer ethics is a branch of practical philosophy that deals with how computing professionals should make decisions regarding professional and social conduct. There are three primary influences: 1) an individual's own personal code; 2) any informal code of ethical behavior existing in the work place; and 3) exposure to formal codes of ethics. See cross-referencing with the Information Assurance and Security (IAS) Knowledge Area.

Topics:

[Core-Tier1]

- Community values and the laws by which we live
- The nature of professionalism including care, attention and discipline, fiduciary responsibility, and mentoring
- Keeping up-to-date as a computing professional in terms of familiarity, tools, skills, legal and professional framework as well as the ability to self-assess and progress in the computing field
- Professional certification, codes of ethics, conduct, and practice, such as the ACM/IEEE-CS, SE, AITP, IFIP and international societies (cross-reference IAS/Fundamental Concepts/ethical issues)
- Accountability, responsibility and liability (e.g. software correctness, reliability and safety, as well as ethical confidentiality of cybersecurity professionals)

[Core-Tier2]

- The role of the computing professional in public policy
- Maintaining awareness of consequences
- Ethical dissent and whistle-blowing
- The relationship between regional culture and ethical dilemmas
- Dealing with harassment and discrimination
- Forms of professional credentialing
- Acceptable use policies for computing in the workplace
- Ergonomics and healthy computing environments
- Time to market and cost considerations versus quality professional standards

Learning Outcomes:

[Core-Tier1]

1. Identify ethical issues that arise in software development and determine how to address them technically and ethically. [Familiarity]
2. Explain the ethical responsibility of ensuring software correctness, reliability and safety. [Familiarity]
3. Describe the mechanisms that typically exist for a professional to keep up-to-date. [Familiarity]

4. Describe the strengths and weaknesses of relevant professional codes as expressions of professionalism and guides to decision-making. [Familiarity]
5. Analyze a global computing issue, observing the role of professionals and government officials in managing this problem. [Assessment]
6. Evaluate the professional codes of ethics from the ACM, the IEEE Computer Society, and other organizations. [Assessment]

[Core-Tier2]

7. Describe ways in which professionals may contribute to public policy. [Familiarity]
8. Describe the consequences of inappropriate professional behavior. [Familiarity]
9. Identify progressive stages in a whistle-blowing incident. [Familiarity]
10. Identify examples of how regional culture interplays with ethical dilemmas. [Familiarity]
11. Investigate forms of harassment and discrimination and avenues of assistance. [Usage]
12. Examine various forms of professional credentialing. [Usage]
13. Explain the relationship between ergonomics in computing environments and people's health. [Familiarity]
14. Develop a computer usage/acceptable use policy with enforcement measures. [Assessment]
15. Describe issues associated with industries' push to focus on time to market versus enforcing quality professional standards. [Familiarity]

SP/Intellectual Property

[2 Core-Tier1 hours]

Intellectual property refers to a range of intangible rights of ownership in an asset such as a software program. Each intellectual property "right" is itself an asset. The law provides different methods for protecting these rights of ownership based on their type. There are essentially four types of intellectual property rights relevant to software: patents, copyrights, trade secrets and trademarks. Each affords a different type of legal protection. See cross-referencing with the Information Management (IM) Knowledge Area.

Topics:

[Core-Tier1]

- Philosophical foundations of intellectual property
- Intellectual property rights (cross-reference IM/Information Storage and Retrieval/intellectual property and protection)
- Intangible digital intellectual property (IDIP)
- Legal foundations for intellectual property protection
- Digital rights management
- Copyrights, patents, trade secrets, trademarks
- Plagiarism

[Elective]

- Foundations of the open source movement
- Software piracy

Learning Outcomes:

[Core-Tier1]

1. Discuss the philosophical bases of intellectual property. [Familiarity]
2. Discuss the rationale for the legal protection of intellectual property. [Familiarity]
3. Describe legislation aimed at digital copyright infringements. [Familiarity]
4. Critique legislation aimed at digital copyright infringements. [Assessment]
5. Identify contemporary examples of intangible digital intellectual property. [Familiarity]
6. Justify uses of copyrighted materials. [Assessment]
7. Evaluate the ethical issues inherent in various plagiarism detection mechanisms. [Assessment]
8. Interpret the intent and implementation of software licensing. [Familiarity]
9. Discuss the issues involved in securing software patents. [Familiarity]
10. Characterize and contrast the concepts of copyright, patenting and trademarks. [Assessment]

[Elective]

11. Identify the goals of the open source movement. [Familiarity]
12. Identify the global nature of software piracy. [Familiarity]

SP/Privacy and Civil Liberties

[2 Core-Tier1 hours]

Electronic information sharing highlights the need to balance privacy protections with information access. The ease of digital access to many types of data makes privacy rights and civil liberties more complex, differing among the variety of cultures worldwide. See cross-referencing with the Human-Computer Interaction (HCI), Information Assurance and Security (IAS), Information Management (IM), and Intelligent Systems (IS) Knowledge Areas.

Topics:

[Core-Tier1]

- Philosophical foundations of privacy rights (cross-reference IS/Fundamental Issues/philosophical issues)
- Legal foundations of privacy protection
- Privacy implications of widespread data collection for transactional databases, data warehouses, surveillance systems, and cloud computing (cross-reference IM/Database Systems/data independence; IM/Data Mining/data cleaning)
- Ramifications of differential privacy
- Technology-based solutions for privacy protection (cross-reference IAS/Threats and Attacks/attacks on privacy and anonymity)

[Elective]

- Privacy legislation in areas of practice
- Civil liberties and cultural differences
- Freedom of expression and its limitations

Learning Outcomes:

[Core-Tier1]

1. Discuss the philosophical basis for the legal protection of personal privacy. [Familiarity]
2. Evaluate solutions to privacy threats in transactional databases and data warehouses. [Assessment]

3. Describe the role of data collection in the implementation of pervasive surveillance systems (e.g., RFID, face recognition, toll collection, mobile computing). [Familiarity]
4. Describe the ramifications of differential privacy. [Familiarity]
5. Investigate the impact of technological solutions to privacy problems. [Usage]

[Elective]

6. Critique the intent, potential value and implementation of various forms of privacy legislation. [Assessment]
7. Identify strategies to enable appropriate freedom of expression. [Familiarity]

SP/Professional Communication

[1 Core-Tier1 hour]

Professional communication conveys technical information to various audiences who may have very different goals and needs for that information. Effective professional communication of technical information is rarely an inherited gift, but rather needs to be taught in context throughout the undergraduate curriculum. See cross-referencing with Human-Computer Interaction (HCI) and Software Engineering (SE) Knowledge Areas.

Topics:

[Core-Tier1]

- Reading, understanding and summarizing technical material, including source code and documentation
- Writing effective technical documentation and materials
- Dynamics of oral, written, and electronic team and group communication (cross-reference HCI/Collaboration and Communication/group communication; SE/Project Management/team participation)
- Communicating professionally with stakeholders
- Utilizing collaboration tools (cross-reference HCI/Collaboration and Communication/online communities; IS/Agents/collaborative agents)

[Elective]

- Dealing with cross-cultural environments (cross-reference HCI/User-Centered Design and Testing/cross-cultural evaluation)
- Tradeoffs of competing risks in software projects, such as technology, structure/process, quality, people, market and financial (cross-reference SE/Software Project Management/Risk)

Learning Outcomes:

[Core-Tier1]

1. Write clear, concise, and accurate technical documents following well-defined standards for format and for including appropriate tables, figures, and references. [Usage]
2. Evaluate written technical documentation to detect problems of various kinds. [Assessment]
3. Develop and deliver a good quality formal presentation. [Assessment]
4. Plan interactions (e.g. virtual, face-to-face, shared documents) with others in which they are able to get their point across, and are also able to listen carefully and appreciate the points of others, even when they disagree, and are able to convey to others what they have heard. [Usage]
5. Describe the strengths and weaknesses of various forms of communication (e.g. virtual, face-to-face, shared documents). [Familiarity]
6. Examine appropriate measures used to communicate with stakeholders involved in a project. [Usage]
7. Compare and contrast various collaboration tools. [Assessment]

[Elective]

8. Discuss ways to influence performance and results in cross-cultural teams. [Familiarity]
9. Examine the tradeoffs and common sources of risk in software projects regarding technology, structure/process, quality, people, market and financial. [Usage]
10. Evaluate personal strengths and weaknesses to work remotely as part of a multinational team. [Assessment]

SP/Sustainability

[1 Core-Tier1 hour, 1 Core-Tier2 hour]

Sustainability is characterized by the United Nations [1] as “development that meets the needs of the present without compromising the ability of future generations to meet their own needs." Sustainability was first introduced in the CS2008 curricular guidelines. Topics in this emerging area can be naturally integrated into other familiarity areas and units, such as human-computer interaction and software evolution. See cross-referencing with the Human-Computer Interaction (HCI) and Software Engineering (SE) Knowledge Areas.

Topics:

[Core-Tier1]

- Being a sustainable practitioner by taking into consideration cultural and environmental impacts of implementation decisions (e.g. organizational policies, economic viability, and resource consumption).
- Explore global social and environmental impacts of computer use and disposal (e-waste)

[Core-Tier2]

- Environmental impacts of design choices in specific areas such as algorithms, operating systems, networks, databases, or human-computer interaction (cross-reference SE/Software Evaluation/software evolution; HCI/Design-Oriented HCI/sustainability)

[Elective]

- Guidelines for sustainable design standards
- Systemic effects of complex computer-mediated phenomena (e.g. telecommuting or web shopping)
- Pervasive computing; information processing integrated into everyday objects and activities, such as smart energy systems, social networking and feedback systems to promote sustainable behavior, transportation, environmental monitoring, citizen science and activism.
- Research on applications of computing to environmental issues, such as energy, pollution, resource usage, recycling and reuse, food management, farming and others.
- The interdependence of the sustainability of software systems with social systems, including the knowledge and skills of its users, organizational processes and policies, and its societal context (e.g., market forces, government policies).

Learning Outcomes:

[Core-Tier1]

1. Identify ways to be a sustainable practitioner. [Familiarity]
2. Illustrate global social and environmental impacts of computer use and disposal (e-waste). [Usage]

[Core-Tier2]

3. Describe the environmental impacts of design choices within the field of computing that relate to algorithm design, operating system design, networking design, database design, etc. [Familiarity]
4. Investigate the social and environmental impacts of new system designs through projects. [Usage]

[Elective]

5. Identify guidelines for sustainable IT design or deployment. [Familiarity]
6. List the sustainable effects of telecommuting or web shopping. [Familiarity]
7. Investigate pervasive computing in areas such as smart energy systems, social networking, transportation, agriculture, supply-chain systems, environmental monitoring and citizen activism. [Usage]
8. Develop applications of computing and assess through research areas pertaining to environmental issues (e.g. energy, pollution, resource usage, recycling and reuse, food management, farming). [Assessment]

SP/History

[Elective]

This history of computing is taught to provide a sense of how the rapid change in computing impacts society on a global scale. It is often taught in context with foundational concepts, such as system fundamentals and software developmental fundamentals.

Topics:

- Prehistory—the world before 1946
- History of computer hardware, software, networking (cross-reference AR/Digital logic and digital systems/ history of computer architecture)
- Pioneers of computing
- History of the Internet

Learning Outcomes:

1. Identify significant continuing trends in the history of the computing field. [Familiarity]
2. Identify the contributions of several pioneers in the computing field. [Familiarity]
3. Discuss the historical context for several programming language paradigms. [Familiarity]
4. Compare daily life before and after the advent of personal computers and the Internet. [Assessment]

SP/Economies of Computing

[Elective]

Economics of computing encompasses the metrics and best practices for personnel and financial management surrounding computer information systems.

Topics:

- Monopolies and their economic implications
- Effect of skilled labor supply and demand on the quality of computing products
- Pricing strategies in the computing domain
- The phenomenon of outsourcing and off-shoring software development; impacts on employment and on economics
- Consequences of globalization for the computer science profession

- Differences in access to computing resources and the possible effects thereof
- Cost/benefit analysis of jobs with considerations to manufacturing, hardware, software, and engineering implications
- Cost estimates versus actual costs in relation to total costs
- Entrepreneurship: prospects and pitfalls
- Network effect or demand-side economies of scale
- Use of engineering economics in dealing with finances

Learning Outcomes:

1. Summarize the rationale for antimonopoly efforts. [Familiarity]
2. Identify several ways in which the information technology industry is affected by shortages in the labor supply. [Familiarity]
3. Identify the evolution of pricing strategies for computing goods and services. [Familiarity]
4. Discuss the benefits, the drawbacks and the implications of off-shoring and outsourcing. [Familiarity]
5. Investigate and defend ways to address limitations on access to computing. [Usage]
6. Describe the economic benefits of network effects. [Familiarity]

SP/Security Policies, Laws and Computer Crimes

[Elective]

While security policies, laws and computer crimes are important subjects, it is essential they are viewed with the foundation of other Social and Professional knowledge units, such as Intellectual Property, Privacy and Civil Liberties, Social Context, and Professional Ethics. Computers and the Internet, perhaps more than any other technology, have transformed society over the past 75 years. At the same time, they have contributed to unprecedented threats to privacy; whole new categories of crime and anti-social behavior; major disruptions to organizations; and the large-scale concentration of risk into information systems. See cross-referencing with the Human-Computer Interaction (HCI) and Information Assurance and Security (IAS) Knowledge Areas.

Topics:

- Examples of computer crimes and legal redress for computer criminals (cross-reference IAS/Digital Forensics/rules of evidence)
- Social engineering, identity theft and recovery (cross-reference HCI/Human Factors and Security/trust, privacy and deception)
- Issues surrounding the misuse of access and breaches in security
- Motivations and ramifications of cyber terrorism and criminal hacking, “cracking”
- Effects of malware, such as viruses, worms and Trojan horses
- Crime prevention strategies
- Security policies (cross-reference IAS/Security Policy and Governance/policies)

Learning Outcomes:

1. List classic examples of computer crimes and social engineering incidents with societal impact. [Familiarity]
2. Identify laws that apply to computer crimes. [Familiarity]
3. Describe the motivation and ramifications of cyber terrorism and criminal hacking. [Familiarity]
4. Examine the ethical and legal issues surrounding the misuse of access and various breaches in security. [Usage]

5. Discuss the professional's role in security and the trade-offs involved. [Familiarity]
6. Investigate measures that can be taken by both individuals and organizations including governments to prevent or mitigate the undesirable effects of computer crimes and identity theft. [Usage]
7. Write a company-wide security policy, which includes procedures for managing passwords and employee monitoring. [Usage]

References

[1] "Our Common Future." http://grawemeyer.org/worldorder/previous-winners/1991-the-united-nations-world-commission-on-environment-and-development.html

[2] Tucker, A. (ed), B. Barnes, R. Aiken, K. Barker, K. Bruce, J. Cain, S. Conry, G. Engel, R. Epstein, D. Lidtke, M. Mulder, J. Rogers, E. Spafford, A. Turner, *Computing Curricula 1991: Report of the Joint Curriculum Task Force*, ACM Press and IEEE-CS Press, 1991.