Using algorithms to compute solutions to problems is at the heart of computer science. Consequently, computer scientists are well-versed in the theoretical and practical aspects of algorithmic computation. Of paramount importance are questions related to the definition, capabilities and limitations of algorithmic computation, as well as, what general strategies exist for design algorithms and what specific algorithms, and associated data structures, exist for solving commonly occurring problems. Computer scientists must also consider the impact on society of using specific algorithms when solving real-world problems. This knowledge area focuses on these questions.

An algorithm is essentially a set of unambiguous instructions written in some language that can be executed by some machine for the purpose of transforming data. As such machines are defined by mathematical models of computation, referred to as abstract machines, computer scientists are conversant with the definitions, capabilities, and limitations of common abstract machines. As the Church-Turing Thesis hypothesizes that a Turing Machine provides the most capable model of algorithmic calculation, the computational capability of all future programming languages and computers will be a subset of those provided by a Turing Machine. Interestingly, there are known problems, such as the Halting Problem, which cannot be solved by any Turing Machine. As a result, computer scientists recognize the possibility that no suitable algorithm may exist to solve a plethora of known problems. Computability theory is the branch of computer science that focuses on abstract machines and their computational capabilities.

Alternatively, complexity theory focuses on the practical efficiency of an algorithm to solve a computational problem. The real-world performance of any computing system depends on: (1) the algorithms chosen and (2) the suitability, capability, and efficiency of the various layers of implementation. Therefore, good algorithm design is crucial for the performance of all computing systems. Consequently, an important part of computing is the ability to select algorithms appropriate to particular purposes and apply them. This ability relies on understanding the range of algorithms that address an important set of well-defined problems, recognizing their strengths, weaknesses, and suitability in particular contexts. Efficiency is a pervasive theme in this knowledge area with the distinction between tractable and intractable algorithms being of paramount importance since computer scientists recognize that intractable algorithms are limited in their ability to practically solve increasingly larger problems.

In general, the study of algorithms and their practical and theoretical limitations provides insight into the intrinsic nature of problems and problem-solving techniques, independent of any programming language, programming paradigm, specific hardware, or implementation aspect.

When compared to CS 2013, this knowledge area returns to early CS curricular recommendations that required more insight into Automata and Computability Theory in the non-elective CS Core. It also includes a new focus on considering the impact algorithmic computing has on society, as well as, algorithms focused on protecting individuals.
## Allocation of Core Hours

<table>
<thead>
<tr>
<th>Knowledge Unit</th>
<th>CS Core</th>
<th>KA Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL/Fundamental Data Structures and Algorithms</td>
<td>30</td>
<td>12</td>
</tr>
<tr>
<td>AL/Algorithmic Strategies</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>AL/Complexity Analysis</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>AL/Automata and Computability Theory</td>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td>AL/Algorithms and Society</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

- Twenty-three of the CS Core hours in this area intersect with the SFD/Fundamental Data Structures and SDF/Algorithms and Design knowledge unit hours, which are targeted as a first-year experience. The remaining twenty-five hours are targeted for the second year.
- The AL and SDF areas are complementary in their approach to algorithms and data structures with the SDF area focusing heavily on implementation and the AL area focusing more on an in-depth understanding of algorithmic computation.
- Algorithms in this area also overlap with topics in the Artificial Intelligence and Information Assurance knowledge areas. Graphs and trees coverage also overlaps with the Mathematical Foundations area.
- While not strictly course based, the CS Core hours in this knowledge area can be distributed across the curriculum with the majority of the topics presented in first-year Data Structures (CS2) and second-year Algorithms (CS3) type courses. If a third course is required by a program, typically Computation Theory, then in-depth coverage of Automata and Computability Theory can be included, which frees up lecture hours to cover a selection of additional important algorithms, as part of the KA core.
AL/Fundamental Data Structures and Algorithms
[30 CS Core hours, 12 KA Core Hours]

Algorithms and data structures are fundamental to computer science. The collection of data structures and algorithms specified in the CS Core of this unit are fundamental precisely because every computer science graduate is expected to be able to articulate their characteristics and use them in solving various computational problems. This unit complements the AL/Algorithmic Strategies unit since the listed algorithms represent instances of the various strategies in that unit. Many of the topics in this unit also overlap those recommended in the Software Development Fundamentals (SDF), Math Foundations (MF), Artificial Intelligence (AI), and Information Assurance and Security (IAD) knowledge areas.

- **Topics**
  - [CS Core]
    - Fundamental Abstract Data Types (ADTs)
      - Arrays
      - Graphs: (Un)Directed, (A)Cyclic, (Un)Connected, and (Un)Weighted
      - Hash Tables/Maps
      - Lists: Single and Doubly Linked Lists
      - Matrices: Single (vectors) and Multi-Dimensional
      - Objects
      - Queues: First-In First-Out (FIFO), Priority Queues
      - Records/Structs/Tuples
      - Sets
      - Stacks: Last-In First-Out (LIFO)
      - Strings
      - Trees: Binary, (Un)Ordered, and (Un)Balanced; Heaps
    - Fundamental Algorithms
      - Search: Sequential and Binary
      - Sorting: Selection, Quick, Insertion, Merge, and Heap
      - Graph Algorithms
        - Searching using Depth-First, Best-First, and Breadth-First
        - Finding Single-Source, Shortest-path (Dijkstra’s, Bellman-Ford)
        - Finding All-Pairs Shortest-Paths (Floyd-Warshall)
        - Determining Strongly Connectivity (Depth-First Search)
        - Determining Transitive Closure (Warshall’s)
        - Finding Minimal Spanning Tree (Prim’s and Kruskal’s)
        - Topological sorting (Depth-First Search)
    - Hash Table Collision Resolution
      - Linear and Quadratic Probing, Chaining, and Rehashing
  - Trees
    - Searching using Depth-First, Best-First, and Breadth-First Search
    - Balancing during insertion and deletion (AVL or Red-Black; 2-3 or B)
  - Strings
    - String Matching (Boyer-Moore), Lexicographical Ordering
  - Differential Privacy
- Map Reduce
- Public-Key Encryption (e.g. RSA)
- Secure Hash Algorithm 256 (SHA-256)
- Blockchain Algorithms (e.g. Bitcoin)

- [KA Core]
  - Bloom Filter
  - Consensus Algorithms
  - Digital Fingerprint (e.g. Rabin's Algorithm)
  - Fast Fourier Transform
  - Geometric Algorithms (e.g., Graham Scan for Convex Hull)
  - Linear Programming (e.g. Simplex algorithm)
  - Network Flow Algorithms (e.g. Ford-Fulkerson max-flow)
  - Number Theoretic
  - Randomized Algorithms (e.g. Genetic Algorithms, Simulated Annealing)
  - Quantum Algorithms (e.g., Deutsch-Jozsa, Bernstein-Vazirani, Simon’s, Shor’s Grover’s)

- Illustrative Learning Outcomes
- [CS Core]
  1. For each Fundamental ADT/Data Structure
     a. Articulate the definitions, properties, representation, and common interface operations supported by the ADT including how these operations work (e.g. create, retrieve, update, and delete operations)
     b. Compare the usage and efficiency of the data structure with other ADTs
  2. Represent graphs (various types) using adjacency lists and adjacency matrices.
  3. For each of the following algorithms show step-by-step how the algorithm
     a. Sequential search find elements in an array or list
     b. Binary search finds elements in a sorted array.
     c. Selection, quick, merge, and insertion sorts sort an unsorted array.
     d. Depth-first and breadth-first search find a node/vertex in a tree/graph including calculating the cost of reaching the node/ vertex.
     e. Dijkstra’s algorithm solves the single-source shortest-paths problem for a weighted directed graph.
     f. Bellman-Ford algorithm solves the single-source shortest paths problem on a directed weighted graph with positive or negative weights.
     g. Floyd-Warshall algorithm solves the all-pairs shortest-paths problem on a directed graph or the transitive closure of a graph.
     h. Prims and Kruskal’s algorithms find a minimal spanning tree
     i. Two depth-first searches determine if a graph is strongly connected.
     j. Depth-first search can topologically sort the vertices in a graph.
     k. Stack can be used to determine whether parentheses are balanced in a string
     l. Stack supports last-in, first-out (LIFO) data handling
     m. Queue supports first-in, first-out (FIF) data handling
     n. AVL, 2-3, or a Red-Black tree handles inserting and delete elements
o. Linear probing, chaining, and rehashing handle collision resolution.

p. Heaps handle inserting and deleting elements

q. Brute-force, Boyer-Moore, and Knuth-Morris-Pratt match strings

r. Calculate a hash key for a string using a variety of approaches (e.g. simple modulo arithmetic, MD5, and SHA-256)

4. Explain how Public Key and hashing algorithms support privacy, digital signing, and encryption.

5. Describe the essence of the blockchain algorithm used in crypto-currency applications (e.g. bitcoin).

6. Where appropriate, compare how different algorithms may solve the same problem using a different approach.

7. Justify the selection of a particular algorithm/data structure to solve a specific computational problem by comparing it with other possible solutions.
AL/Algorithmic Strategies
[30 CS Core hours, Elective Hours]
This unit focuses on common design strategies used in algorithmic problem solving. The small number of CS Core hours in this unit reflects the fact that instances of these strategies are exemplified by algorithms in the AL/Fundamental Data Structures and Algorithms unit. Heuristic and branch-and-bound algorithms are also used in the IS/Search Strategies unit.

- **Topics**
  - **[CS Core]**
    - Backtracking
    - Branch and Bound
    - Brute-Force
    - Consensus algorithms
    - Decrease-and-Conquer
    - Divide-and-Conquer
    - Dynamic Programming
    - Greedy Algorithms
    - Heuristic
    - Iterative
    - Recursive
    - Transform-and-Conquer (Reduction)
  - **[KA Topics]**
    - Randomized/Stochastic Algorithms: Genetic Algorithms, Simulated Annealing
  - **Illustrative Learning Outcomes**
    - **[CS Core]**
      1. For each of the algorithms in the AL/Fundamental Data Structures and Algorithms unit, identify the algorithmic strategy it uses and articulate how the algorithm exemplifies the corresponding strategy demonstrating how it solves a specific problem instance:
        a. How binary search and mergesort, quicksort, and insertion sort exemplify a divide-and-conquer algorithmic strategy by using them to find an element in or sort an unsorted array of elements.
        b. How topological sorting using depth-first search exemplifies a decrease-and-conquer algorithmic strategy by using
        c. How Dijkstra's algorithm exemplifies a greedy algorithmic strategy by using it to solve a single-source shortest-path graph problem
        d. How Warshall's Algorithm exemplifies a dynamic programming algorithmic strategy by using it to find the transitive closure of a directed graph.
        e. How Min-max with Alpha-Beta pruning exemplifies heuristic, branch-and-bound, and back-tracking strategies.
        f. How pre-sorting a list exemplifies a transform-and-conquer strategy.
        g. How blockchain, as used by crypto-currency applications (e.g. Bitcoin) exemplifies a consensus algorithmic strategy.
      2. Transform between common recursive and iterative problem-solving algorithms
AL/Complexity Analysis
[12 CS Core hours]

- Topics
  - [CS Core]
    o Best, Average, and Worst case algorithmic behavior
    o Asymptotic analysis of upper and expected complexity bounds
    o Big and little O, Omega, and Theta notation
    o Complexity Classes: Constant, logarithmic, linear, quadratic, cubic, and exponential, P, NP, and NPC
    o Empirical measurements of performance
    o Time and space trade-off in algorithms
    o Analysis of iterative and recursive algorithms
    o Strategies to solve recurrence relations (e.g. Substitution, Master Theorem)
    o P-NP and NPC problems (e.g., SAT, Knapsack, and Hamiltonian Circuit)

- Topics
  - [Illustrative Learning Outcomes]
    1. Explain the use of Big-O, Omega, and Theta notation to describe the amount of work done by an algorithm.
    2. List and contrast standard complexity classes.
    3. Explain what is meant by “best”, “expected”, and “worst” case behavior of an algorithm
    4. Determine and articulate the time and space complexity of simple algorithms, including those associated with the ADTs in the AL/Fundamental Data Structures and Algorithms knowledge unit
    5. Explain the difference in complexity between sequential and binary search
    6. Explain the difference in complexity between quadratic and logarithmic sorting (e.g. Selection vs. Quicksort)
    7. Perform empirical studies to validate hypotheses about runtime stemming from mathematical analysis.
    8. Run various algorithms on input of various sizes and compare performance.
    9. Give examples that illustrate time-space trade-offs of algorithms.
   10. Use recurrence relations to determine the time complexity of recursively defined algorithms (e.g. solve elementary recurrence relations: substitution, Master Theorem).
   11. Define the classes P, NP, and NPC.
   12. Explain the significance of P, NP, and NP-Completeness.
AL/Automata and Computability
[12 CS Core hours, 30 KA Hours]
In addition to serving as models of computation, various automata provide the underlying theoretical basis for lexical, syntactic, and semantic analysis during compilation of a program. Hence, this knowledge unit complements PL Programming Languages

- **Topics**
  - **[CS Core]**
    - Formal Languages, Grammars, and Chomsky Hierarchy
      - Regular, Context-Free, Context-Sensitive, and Recursively Enumerable
      - Regular expressions
    - Formal Automata: Finite State, Pushdown, Linear-Bounded, and Turing Machine
      - Deterministic versus Nondeterministic and equivalencies
      - Relations among formal automata, languages and grammars
      - Decidability and limitations
    - Decidability
    - Uncomputability and the Halting problem
    - The Church-Turing Thesis
    - The P, NP, and NPC complexity
  - **[KA Topics]**
    - Pumping Lemmas for Regular and Context-Free Languages
    - Turing Machine Variations (e.g. multi-tape, non-deterministic)
    - Decidability Proofs for various automata including semi-decidable languages
    - Reducibility Proofs
    - Time Complexity based on Turing Machine
    - Space Complexity (e.g. PSPACE, Savitch’s Theorem)
    - Equivalent Models of Computation
      - Lambda Calculus, Mu-Recursive Functions
    - Turing Machine Variations (e.g. multi-tape, non-deterministic)
    - Arithmetization
    - Rice’s Theorem
- **[Illustrative Learning Outcomes]**
  - **[CS Core]**
    1. Explain the concept of a formal automaton as a model of computation
    2. Compare and contrast the definitions, capabilities, and limitations of finite state, Pushdown Linear-Bounded, and Turing Machine automata.
    3. Design finite state machines, pushdown automata, and Turing Machines that accept a specified language.
    4. Generate a regular expression to represent a specified language.
    5. Compare and contrast regular, context-free, context sensitive, and Recursively Enumerable languages and their uses
    6. Compare and contrast regular, context-free, context sensitive, and Recursively Enumerable languages (e.g. Type-X in the Chomsky Hierarchy) and their uses
    7. Design regular and context-free grammars to represent a specified language.
8. Convert among equivalently powerful notations for a language, including among DFAs, NFAs, and regular expressions, and between PDAs and CFGs.
9. Explain why the halting problem has no algorithmic solution.
10. Explain the Church-Turing thesis and its significance.
11. Define the classes P and NP.
12. Explain the significance of NP-completeness.

AL/Algorithms and Society
[2 CS Core hours]

- Topics
- [CS Core]
  - Context-Aware Computing
  - Social, Ethical, and Secure Algorithms
- [Illustrative Learning Outcomes]
  1. Devise algorithmic solutions to real-world societal problems (e.g., routing an ambulance to a hospital)
  2. Consider and explain the impact that an algorithm’s design will have on society when used to solve real-world problems.
  3. Articulate how differential privacy protects knowledge of an individual’s data.
  4. Articulate how Public Key encryption protects knowledge of secure data.
  5. Articulate the pros and cons of using blockchain algorithms to support crypto-currency (e.g. as used by Bitcoin).

List of Professional Dispositions Appropriate for this KA
Overarching is a growth-mindset
- A curiosity for exploring how software solves real-world problems.
- An enthusiasm for determining efficient software solutions to real-world problems
- Self-confidence in one’s ability to find algorithmic solutions to real-world problems
- An appreciation for the impact of CS theory on solving real-world problems.

Math needed and wanted
- Math Foundations
  - MF/Sets, Relations, Functions
  - MF/Basics of Counting
    - Set cardinality and counting
    - Solving recurrence relations
  - MF/Graphs and Trees
  - MF/Proof Techniques

Crosscutting and Overlapping topics
- Software Development Fundamentals
  - SDF/Fundamental Data Structures
  - SDF/Algorithms and Design
- Programming Languages
- PL/Object-Oriented Programming
- PL/Syntax Analysis
- Information Assurance and Security
  - IAS/Cryptography
- Artificial Intelligence
  - AI/Basic Search Strategies

Subcommittee
Chair: Richard Blumenthal, Regis University, Denver, United States

Subcommittee members
- Cathy Bareiss, Bethel University, Mishawaka, United States
- Tom Blanchet, Hillman Companies Inc., Boulder, United States
- Doug Lea, State University of New York at Oswego, Oswego, United States
- Sara Miner More, John Hopkins University, Baltimore, United States
- Mia Minnes, University of California San Diego, United States
- Atri Rudra, University of Buffalo, Buffalo, United States
- Christian Servin, El Paso Community College, El Paso, United States

Other contributors:
-