Computer Science Curricula 2013

Algorithms and Complexity (AL)	Basic Analysis	[Tier1] 2	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
		[Tier2] 2	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
	Algorithmic Strategies	[Tier1] 5	Brute-force algorithms
			Greedy algorithms
			Divide-and-conquer (cross-reference SDF/Algorithms and Design/Problem-solving strategies)
			Recursive backtracking
			Dynamic Programming
		[Tier2] 1	Branch-and-bound
			Heuristics
			Reduction: transform-and-conquer
	Fundamental Data Structures and Algorithms	[Tier1] 9	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
		[Tier2] 3	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)
	Basic Automata Computability and Complexity	[Tier1] 3	Finite-state machines
			Regular expressions
			The halting problem
		[Tier2] 3	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)
	Advanced Computational Complexity	[Elective]	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
	Advanced Automata Theory and Computability	[Elective]	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
	Advanced Data Structures Algorithms and Analysis	[Elective]	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
Discrete Structures (DS)	Sets, Relations, and Functions	[Tier1] 4	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
	Basic Logic	[Tier1] 9	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)
	Proof Techniques	[Tier1] 10	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
		[Tier2] 1	Well orderings
	Basics of Counting	[Tier1] 5	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
	Graphs and Trees	[Tier1] 3	Trees: Properties; Traversal strategies
			Undirected graphs
			Directed graphs
			Weighted graphs
		[Tier2] 1	Spanning trees/forests
			Graph isomorphism
	Discrete Probability	[Tier1] 6	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
		[Tier2] 2	Variance
			Conditional Independence
Programming Languages (PL)	Object-Oriented Programming	[Tier1] 4	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
		[Tier2] 6	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
	Functional Programming	[Tier1] 3	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)
		[Tier2] 4	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
	Event-Driven and Reactive Programming	[Tier2] 2	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
	Basic Type Systems	[Tier1] 1	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
		[Tier2] 4	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
	Program Representation	[Tier2] 1	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
	Language Translation and Execution	[Tier2] 3	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
	Syntax Analysis	[Elective]	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
	Compiler Semantic Analysis	[Elective]	High-level program representations such as abstract syntax trees
			Scope and binding resolution
			Type checking
			Declarative specifications such as attribute grammars
	Code Generation	[Elective]	Procedure calls and method dispatching
			Separate compilation; linking
			Instruction selection
			Instruction scheduling
			Register allocation
			Peephole optimization
	Runtime Systems	[Elective]	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.
	Static Analysis	[Elective]	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
	Advanced Programming Constructs	[Elective]	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
	Concurrency and Parallelism	[Elective]	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
	Type Systems	[Elective]	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
	Formal Semantics	[Elective]	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
	Language Pragmatics	[Elective]	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
	Logic Programming	[Elective]	Clausal representation of data structures and algorithms
			Unification
			Backtracking and search
			Cuts

CS 250 Discrete Structures I

250.1 Describe basic properties of sets, bags, tuples, relations, graphs, trees, and functions.	
250.2 Perform traversals of graphs and trees; construct simple functions by composition of known functions; determine whether simple functions are injective, surjective, or bijective; and classify simple functions by rate of growth.	
250.3 Describe the concepts of countable and uncountable sets, and apply the diagonalization method to construct elements that are not in certain countable sets.	
250.4 Construct inductive definitions for sets, construct grammars for languages (sets of strings), and construct recursive definitions for functions and procedures.	
250.5 Determine whether a binary relation is reflexive, symmetric, or transitive and construct closures with respect to these properties.	
250.6 Construct a topological sort of a partially ordered set and determine whether a partially ordered set is well-founded.	
250.7 Use elementary counting techniques to count simple finite structures that are either ordered or unordered, to count the worst case number of comparisons and, with discrete probability, to count the average number of comparisons for simple decision trees.	
250.8 Find closed form solutions for simple recurrences using the techniques of substitution, cancellation, and generating functions.	
250.9 Demonstrate standard proof techniques and the technique of inductive proof by writing short informal proofs about simple properties of numbers, sets, and ordered structures.	

CS 251 Discrete Structures II

251.1 Apply the properties of propositional calculus to: determine whether a wff is a tautology, a contradiction, or a contingency by truth tables and by Quine's method; construct equivalence proofs; and transform truth functions and wffs into conjunctive or disjunctive normal form.
251.2 Describe the basic inference rules and use them to write formal proofs in propositional calculus.
251.3 Apply the properties of first-order predicate calculus to: determine whether a wff is valid, invalid, satisfiable, or unsatisfiable; construct equivalence proofs; and transform first-order wffs into prenex conjunctive or disjunctive normal form.
251.4 Describe the rules of inference for quantifiers and use them along with the basic inference rules to write formal proofs in first-order predicate calculus.
251.5 Write formal proofs in first-order predicate calculus with equality.
251.6 Construct partial correctness proofs of simple imperative programs and construct termination proofs for simple loops.
251.7 Transform first-order wffs into clausal form; and unify atoms from a set of clauses.
251.8 Describe the resolution inference rule; use it to write formal proofs in first-order logic; and describe how resolution is used to execute a logic program.
251.9 Transform simple English sentences into formal logic (propositional, first-order, or higher-order).
251.10 Apply appropriate algebraic properties to: simplify Boolean expressions; simplify regular expressions; write recursive definitions for simple functions in terms of operations for abstract data types; write expressions to represent relations constructed in terms of operations for relational databases; and work with congruences.

CS 311 Computational Structures

311.1 Find regular grammars and context-free grammars for simple languages whose strings are described by given properties.
311.2 Apply algorithms to: transform regular expressions to NFAs, NFAs to DFAs, and DFAs to minimum-state DFAs; construct regular expressions from NFAs or DFAs; and transform between regular grammars and NFAs.
311.3 Apply algorithms to transform: between PDAs that accept by final state and those that accept by empty stack; and between context-free grammars and PDAs that accept by empty stack.
311.4 Describe LL(k) grammars; perform factorization if possible to reduce the size of k; and write recursive descent procedures and parse tables for simple LL(1) grammars.
311.5 Transform grammars by removing all left recursion and by removing all possible productions that have the empty string on the right side.
311.6 Apply pumping lemmas to prove that some simple languages are not regular or not context-free.
311.7 State the Church-Turing Thesis and solve simple problems with each of the following models of computation: Turing machines (single-tape and multi-tape); while-loop programs; partial recursive functions; Markov algorithms; Post algorithms; and Post systems.
311.8 Describe the concepts of unsolvable and partially solvable; state the halting problem and prove that it is unsolvable and partially solvable; and use diagonalization to prove that the set of total computable functions cannot be enumerated.
311.9 Describe the hierarchy of languages and give examples of languages at each level that do not belong in a lower level.
311.10 Describe the complexity classes P, NP, and PSPACE.

CS 350 Algorithms and Complexity

350.1 Analyze the running time and space complexity of algorithms.
350.2 Use the big Oh notation. (e.g., O(n lg n).)
350.3 Describe how to prove the correctness of an algorithm.
350.4 Use the mathematical techniques required to prove the time complexity of a program/algorithm. (e.g., limits and sums of series.)
350.5 Perform inductive proofs.
350.6 Prove and apply the Master Theorem.
350.7 Describe the notions of P, NP, NPC, and NP-hard.
350.8 Compare the rates of growth of functions.
350.9 Apply algorithmic complexity principles in the design of programs.
350.10 Design divide and conquer and dynamic programming algorithms.

CS 321 Languages and Compiler Design I

321.1 Explain the phase structure of a typical compiler and the role of each phase.
321.2 Describe and apply mechanisms for defining the lexical structure of a programming language.
321.3 Use context-free grammars to define syntax of a programming language.
321.4 Describe and apply mechanisms for transforming grammars to make them suitable for predictive parsing, and for building LL(1) parsing tables.
321.5 Explain the operation of a shift-reduce bottom-up parser.
321.6 Construct abstract syntax trees for programs in a simple language.
321.7 Distinguish between inherited attributes and synthesized attributes, and use them to define simple syntax-directed actions.
321.8 Describe and apply the basic concepts of data abstraction, encapsulation, object-oriented classes, and modules.
321.9 Describe and apply the basic concepts of type systems, including primitive types, aggregate and recursive types, abstract data types, and type equivalence models.
321.10 Contrast the main features of different programming paradigms, including procedural, object-oriented, and functional.
321.11 Implement a lexical analyzer, parser, and type-checker for a simple but realistic language.

CS 322 Languages and Compiler Design II

322.1 Explain basic approaches to formal specification of languages.
322.2 Explain the differences between interpreters and compilers.
322.3 Describe and apply the basic concepts of sequential control abstraction, including structured control constructs and subroutines.
322.4 Explain runtime organization of program execution, including activation records, static and dynamic access links, and frame and stack pointers.
322.5 Describe standard runtime representations for arrays, records, objects, and first-class functions.
322.6 Explain different parameter-passing modes and their implementation implications.
322.7 Give examples of peephole and global optimizations, and explain the general techniques involved in each of them.
322.8 Explain the concept of garbage collection and the typical algorithms.
322.9 Implement an interpreter for a simple but realistic language.
322.10 Implement a code generator for a simple but realistic language,producing native code for a real machine.

CS 250 Discrete Structures I

250.1 Describe basic properties of sets, bags, tuples, relations, graphs, trees, and functions.
250.2 Perform traversals of graphs and trees; construct simple functions by composition of known functions; determine whether simple functions are injective, surjective, or bijective; and classify simple functions by rate of growth.
250.3 Describe the concepts of countable and uncountable sets, and apply the diagonalization method to construct elements that are not in certain countable sets.
250.4 Construct inductive definitions for sets, construct grammars for languages (sets of strings), and construct recursive definitions for functions and procedures.
250.5 Determine whether a binary relation is reflexive, symmetric, or transitive and construct closures with respect to these properties.
250.6 Construct a topological sort of a partially ordered set and determine whether a partially ordered set is well-founded.
250.7 Use elementary counting techniques to count simple finite structures that are either ordered or unordered, to count the worst case number of comparisons and, with discrete probability, to count the average number of comparisons for simple decision trees.
250.8 Find closed form solutions for simple recurrences using the techniques of substitution, cancellation, and generating functions.
250.9 Demonstrate standard proof techniques and the technique of inductive proof by writing short informal proofs about simple properties of numbers, sets, and ordered structures.

CS 251 Discrete Structures II

251.1 Apply the properties of propositional calculus to: determine whether a wff is a tautology, a contradiction, or a contingency by truth tables and by Quine's method; construct equivalence proofs; and transform truth functions and wffs into conjunctive or disjunctive normal form.
251.2 Describe the basic inference rules and use them to write formal proofs in propositional calculus.
251.3 Apply the properties of first-order predicate calculus to: determine whether a wff is valid, invalid, satisfiable, or unsatisfiable; construct equivalence proofs; and transform first-order wffs into prenex conjunctive or disjunctive normal form.
251.4 Describe the rules of inference for quantifiers and use them along with the basic inference rules to write formal proofs in first-order predicate calculus.
251.5 Write formal proofs in first-order predicate calculus with equality.
251.6 Construct partial correctness proofs of simple imperative programs and construct termination proofs for simple loops.
251.7 Transform first-order wffs into clausal form; and unify atoms from a set of clauses.
251.8 Describe the resolution inference rule; use it to write formal proofs in first-order logic; and describe how resolution is used to execute a logic program.
251.9 Transform simple English sentences into formal logic (propositional, first-order, or higher-order).
251.10 Apply appropriate algebraic properties to: simplify Boolean expressions; simplify regular expressions; write recursive definitions for simple functions in terms of operations for abstract data types; write expressions to represent relations constructed in terms of operations for relational databases; and work with congruences.

CS 311 Computational Structures

311.1 Find regular grammars and context-free grammars for simple languages whose strings are described by given properties.
311.2 Apply algorithms to: transform regular expressions to NFAs, NFAs to DFAs, and DFAs to minimum-state DFAs; construct regular expressions from NFAs or DFAs; and transform between regular grammars and NFAs.
311.3 Apply algorithms to transform: between PDAs that accept by final state and those that accept by empty stack; and between context-free grammars and PDAs that accept by empty stack.
311.4 Describe LL(k) grammars; perform factorization if possible to reduce the size of k; and write recursive descent procedures and parse tables for simple LL(1) grammars.
311.5 Transform grammars by removing all left recursion and by removing all possible productions that have the empty string on the right side.
311.6 Apply pumping lemmas to prove that some simple languages are not regular or not context-free.
311.7 State the Church-Turing Thesis and solve simple problems with each of the following models of computation: Turing machines (single-tape and multi-tape); while-loop programs; partial recursive functions; Markov algorithms; Post algorithms; and Post systems.
311.8 Describe the concepts of unsolvable and partially solvable; state the halting problem and prove that it is unsolvable and partially solvable; and use diagonalization to prove that the set of total computable functions cannot be enumerated.
311.9 Describe the hierarchy of languages and give examples of languages at each level that do not belong in a lower level.
311.10 Describe the complexity classes P, NP, and PSPACE.

CS 350 Algorithms and Complexity

350.1 Analyze the running time and space complexity of algorithms.
350.2 Use the big Oh notation. (e.g., O(n lg n).)
350.3 Describe how to prove the correctness of an algorithm.
350.4 Use the mathematical techniques required to prove the time complexity of a program/algorithm. (e.g., limits and sums of series.)
350.5 Perform inductive proofs.
350.6 Prove and apply the Master Theorem.
350.7 Describe the notions of P, NP, NPC, and NP-hard.
350.8 Compare the rates of growth of functions.
350.9 Apply algorithmic complexity principles in the design of programs.
350.10 Design divide and conquer and dynamic programming algorithms.

CS 321 Languages and Compiler Design I

321.1 Explain the phase structure of a typical compiler and the role of each phase.
321.2 Describe and apply mechanisms for defining the lexical structure of a programming language.
321.3 Use context-free grammars to define syntax of a programming language.
321.4 Describe and apply mechanisms for transforming grammars to make them suitable for predictive parsing, and for building LL(1) parsing tables.
321.5 Explain the operation of a shift-reduce bottom-up parser.
321.6 Construct abstract syntax trees for programs in a simple language.
321.7 Distinguish between inherited attributes and synthesized attributes, and use them to define simple syntax-directed actions.
321.8 Describe and apply the basic concepts of data abstraction, encapsulation, object-oriented classes, and modules.
321.9 Describe and apply the basic concepts of type systems, including primitive types, aggregate and recursive types, abstract data types, and type equivalence models.
321.10 Contrast the main features of different programming paradigms, including procedural, object-oriented, and functional.
321.11 Implement a lexical analyzer, parser, and type-checker for a simple but realistic language.

CS 322 Languages and Compiler Design II

322.1 Explain basic approaches to formal specification of languages.
322.2 Explain the differences between interpreters and compilers.
322.3 Describe and apply the basic concepts of sequential control abstraction, including structured control constructs and subroutines.
322.4 Explain runtime organization of program execution, including activation records, static and dynamic access links, and frame and stack pointers.
322.5 Describe standard runtime representations for arrays, records, objects, and first-class functions.
322.6 Explain different parameter-passing modes and their implementation implications.
322.7 Give examples of peephole and global optimizations, and explain the general techniques involved in each of them.
322.8 Explain the concept of garbage collection and the typical algorithms.
322.9 Implement an interpreter for a simple but realistic language.
322.10 Implement a code generator for a simple but realistic language,producing native code for a real machine.

Discrete Structures (DS)

Sets, Relations, and Functions	[Tier1] 4	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	
Basic Logic	[Tier1] 9	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)	
Proof Techniques	[Tier1] 10	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	
	[Tier2] 1	Well orderings	
Basics of Counting	[Tier1] 5	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	
Graphs and Trees	[Tier1] 3	Trees: Properties; Traversal strategies	
		Undirected graphs	
		Directed graphs	
		Weighted graphs	
	[Tier2] 1	Spanning trees/forests	
		Graph isomorphism	
Discrete Probability	[Tier1] 6	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	
	[Tier2] 2	Variance	
		Conditional Independence	

Algorithms and Complexity (AL)

Basic Analysis	[Tier1] 2	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	
	[Tier2] 2	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	
Algorithmic Strategies	[Tier1] 5	Brute-force algorithms	
		Greedy algorithms	
		Divide-and-conquer (cross-reference SDF/Algorithms and Design/Problem-solving strategies)	
		Recursive backtracking	
		Dynamic Programming	
	[Tier2] 1	Branch-and-bound	
		Heuristics	
		Reduction: transform-and-conquer	
Fundamental Data Structures and Algorithms	[Tier1] 9	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	
	[Tier2] 3	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)	
Basic Automata Computability and Complexity	[Tier1] 3	Finite-state machines	
		Regular expressions	
		The halting problem	
	[Tier2] 3	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)	
Advanced Computational Complexity	[Elective]	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	
Advanced Automata Theory and Computability	[Elective]	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	
Advanced Data Structures Algorithms and Analysis	[Elective]	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	

Programming Languages (PL)

Object-Oriented Programming	[Tier1] 4	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	
	[Tier2] 6	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	
Functional Programming	[Tier1] 3	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)	
	[Tier2] 4	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	
Event-Driven and Reactive Programming	[Tier2] 2	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	
Basic Type Systems	[Tier1] 1	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	
	[Tier2] 4	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	
Program Representation	[Tier2] 1	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	
Language Translation and Execution	[Tier2] 3	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	
Syntax Analysis	[Elective]	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	
Compiler Semantic Analysis	[Elective]	High-level program representations such as abstract syntax trees	
		Scope and binding resolution	
		Type checking	
		Declarative specifications such as attribute grammars	
Code Generation	[Elective]	Procedure calls and method dispatching	
		Separate compilation; linking	
		Instruction selection	
		Instruction scheduling	
		Register allocation	
		Peephole optimization	
Runtime Systems	[Elective]	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.	
Static Analysis	[Elective]	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	
Advanced Programming Constructs	[Elective]	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	
Concurrency and Parallelism	[Elective]	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	
Type Systems	[Elective]	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	
Formal Semantics	[Elective]	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	
Language Pragmatics	[Elective]	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	
Logic Programming	[Elective]	Clausal representation of data structures and algorithms	
		Unification	
		Backtracking and search	
		Cuts	

Discrete Structures (DS)

Sets, Relations, and Functions	[Tier1] 4	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	
Basic Logic	[Tier1] 9	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)	
Proof Techniques	[Tier1] 10	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	
	[Tier2] 1	Well orderings	
Basics of Counting	[Tier1] 5	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	
Graphs and Trees	[Tier1] 3	Trees: Properties; Traversal strategies	
		Undirected graphs	
		Directed graphs	
		Weighted graphs	
	[Tier2] 1	Spanning trees/forests	
		Graph isomorphism	
Discrete Probability	[Tier1] 6	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	
	[Tier2] 2	Variance	
		Conditional Independence	

250.1 Describe basic properties of sets, bags, tuples, relations, graphs, trees, and functions.
250.2 Perform traversals of graphs and trees; construct simple functions by composition of known functions; determine whether simple functions are injective, surjective, or bijective; and classify simple functions by rate of growth.
250.3 Describe the concepts of countable and uncountable sets, and apply the diagonalization method to construct elements that are not in certain countable sets.
250.4 Construct inductive definitions for sets, construct grammars for languages (sets of strings), and construct recursive definitions for functions and procedures.
250.5 Determine whether a binary relation is reflexive, symmetric, or transitive and construct closures with respect to these properties.
250.6 Construct a topological sort of a partially ordered set and determine whether a partially ordered set is well-founded.
250.7 Use elementary counting techniques to count simple finite structures that are either ordered or unordered, to count the worst case number of comparisons and, with discrete probability, to count the average number of comparisons for simple decision trees.
250.8 Find closed form solutions for simple recurrences using the techniques of substitution, cancellation, and generating functions.
250.9 Demonstrate standard proof techniques and the technique of inductive proof by writing short informal proofs about simple properties of numbers, sets, and ordered structures.

251.1 Apply the properties of propositional calculus to: determine whether a wff is a tautology, a contradiction, or a contingency by truth tables and by Quine's method; construct equivalence proofs; and transform truth functions and wffs into conjunctive or disjunctive normal form.
251.2 Describe the basic inference rules and use them to write formal proofs in propositional calculus.
251.3 Apply the properties of first-order predicate calculus to: determine whether a wff is valid, invalid, satisfiable, or unsatisfiable; construct equivalence proofs; and transform first-order wffs into prenex conjunctive or disjunctive normal form.
251.4 Describe the rules of inference for quantifiers and use them along with the basic inference rules to write formal proofs in first-order predicate calculus.
251.5 Write formal proofs in first-order predicate calculus with equality.
251.6 Construct partial correctness proofs of simple imperative programs and construct termination proofs for simple loops.
251.7 Transform first-order wffs into clausal form; and unify atoms from a set of clauses.
251.8 Describe the resolution inference rule; use it to write formal proofs in first-order logic; and describe how resolution is used to execute a logic program.
251.9 Transform simple English sentences into formal logic (propositional, first-order, or higher-order).
251.10 Apply appropriate algebraic properties to: simplify Boolean expressions; simplify regular expressions; write recursive definitions for simple functions in terms of operations for abstract data types; write expressions to represent relations constructed in terms of operations for relational databases; and work with congruences.