After this course students can...	Analyze design tradeoffs and constraints (e.g. through space/time complexity analysis) and make appropriate choices in data structures and algorithms when solving problems. (Students care because a good programmer may not be able to do this, but a good computer scientist does -- a good computer scientist has broader design goals (e.g. proof of correctness, resource constraints, performance and scalability issues)).	Expand your programming language repertoire with the addition of C++. Through learning a new language, gain experience in identifying and exploiting high-level properties across programming languages (as opposed to language-specific properties). For example, the use of general data structures in multiple languages, the commonalities of dynamic memory allocation, parameter passing conventions, templates, etc.)	Gain an appreciation for the role of mathematical formalisms (such as discrete mathematics, functions, sets, Big-O notation, proofs, trees, graphs) in expressing and solving problems in computer science (e.g. link the principles of loops, recursion, and induction to establish loop/program correctness).	Begin to form a clear conception of the integration of the topics seen previously (such as introductory programming techniques, recursion, etc) as the greater science of computers. Be able to recognize the bigger picture and how the topics learned in your courses so far come together to serve computer science at large; be able to justify why you have learned the topics you have learned so far.	Manipulate data structures algorithmically, without a specific implementation	Doesn't fit in available course goals
Introduction and Motivation, Foundations	A1	A1		A1		
C++ Programming	B3	B1,B2,B3		B1,B2		
Review of Sets and Functions	C7		C1,C3,C4,C5,C6,C7	C2,C4		
Induction and Recursion	D3,D4,D7	D4,D5,D6	D1,D2	D2,D3		
Loop Invariants			E1,E2			
Big-O, Big-Omega, Big-Theta Complexity	F1,F2,F7,F8,F9,F10	F5	F1,F2,F3,F4,F5,F6,F7			
NP-Completeness ** (optional)						G1, G3, G4
Space Complexity	H1,H2,H3			H2		
Memory Layout		I1,I2,I3,I4		I1,I2,I3,I4		
Linked Lists (Including Stacks, Queues, and Deques), Introduction to Pointers	J2,J4,J6	J4,J5,J6			J1,J8	
Insertion Sort, Mergesort, Quicksort	K1,K2,K3				K5	
Introduction to Trees and Tree Traversal	L2,L4	L3	L1, L3		L5	
Priority Queues, Heaps, Heapsort	M1,M3				M2	
Hashing	N1,N2,N3,N4,N5	N6		N1	N6	
B+ Trees	O1,O4,O5,O6,O7		O3	O4,O6	O2	
Counting: Product Rule, Sum Rule, Inclusion-Exclusion, Tree Diagrams, Combinations, Permutations			P1,P2,P3			
Binomial Theorem, Combinatorial Identities			Q1,Q2			Q2
Binomial Distribution and Basic Probability (new)			R1,R3			R2,R3
Pigeonhole Principle			S1	S1		
Graph Theory: Introduction and Terminology			T1,T2			
Graph Representation, Isomorphism, Graph Connectivity			U1,U2**,U3			
Euler/Hamilton Paths/Cycles**			V1,V2			
Graph Traversals	W1		W2			
Planar Graphs**			X1,X2,X3			

CPSC 221 Topic Learning Goals

Topic	ID	Students Can
Introduction and Motivation, Foundations	A1	Compare abstract and concrete data structures and implications for implementations.
C++ Programming	B1	Effectively pick up a new programming language on their own similar to the first language of instruction (Java). (e.g., code assignments in C++ with minmal help
	B2	Implement basic data structures in the C++ programming language -- the programs (up to several pages long) should effectively use arrays, lists, pointers, recursion, trees, dynamic memory allocation, and classes in C++.
	B3	Analyze C++ programs and functions to determine their algorithmic complexity
Review of Sets and Functions	C1	Demonstrate mathematical literacy (competence, familiarity, ability to use to solve problems) in sets, functions, and mathematical symbols
	C2	Be prepared for further computing studies in fields such as database management systems, algorithm analysis, information retrieval, logic/AI courses (binding of symbols), and functional programming.
	C3	Communicate effectively through set parlance and notation (e.g., be able to translate general problem into rigorous problem statements throughout the course).
	C4	Apply sets and functions to the topic areas in the course including (hashing, complexity analysis, counting, and generally supporting exact problem expression throughout the course).
	C5	Understand the notion of mapping between sets.
	C6	Prove one to one and onto for finite and infinite sets.
	C7	Recognize the different classes of functions in terms of their complexity.
Induction and Recursion	D1	Prove that a loop invariant holds for a given code or algorithm example.
	D2	Describe the relationship between recursion and induction (e.g., take a recursive code fragment and express it mathematically in order to prove it's correctness inductively)
	D3	Evaluate the effect of recursion on space complexity (e.g., explain why a recursively defined method takes more space then an equivalent iteratively defined method.)
	D4	Describe how tail recursive algorithms can require less space complexity than non-tail recursive algorithms.
	D5	Recognize algorithms as being iterative or recursive.
	D6	Convert recursive solutions to iterative solutions and vice versa.
	D7	Draw a recursion tree and relate the depth to a) the number of recursive calls and b) the size of the runtime stack. Identify and/or produce an example of infinite recursion
Loop Invariants	E1	Take a loop code fragment and express it mathematically in order to prove it's correctness inductively (specifically describing that the induction is on the iteration variable)
	E2	In simpler cases, determine the loop invariant.
Big-O, Big-Omega, Big-Theta Complexity	F1	Define which program operations we measure in an algorithm in order to approximate its efficiency (e.g., number of instructions, steps, function calls, comparisons, swaps, I/Os, network accesses).
	F2	Define "input size" and determine the effect (in terms of performance) that input size has on an algorithm
	F3	Give examples of common practical limits of problem size for each complexity class.
	F4	Give examples of tractable, intractable, and undecidable problems.**
	F5	Given a code, write a formula which measures the number of steps executed as a function of the size of the input (N)
	F6	Compute the worst-case asymptotic complexity of an algorithm (e.g., the worst possible running time based on the size of the input (N))
	F7	Categorize an algorithm into one of the common complexity classes (e.g. constant, logarithmic, linear, quadratic, etc.).
	F8	Explain the differences between best, worst, and average case analysis.
	F9	Describe why best-case analysis is rarely relevant and how worst-case analysis may never be encountered in practice.
	F1	Given two or more algorithms, rank them in terms of their time and space complexity
NP-Completeness ** (optional)	G1	State the basic properties of NP-Complete problems and explain why they are hard to solve computationally
	G3	Give examples of NP-Complete problems.
	G4	Explain the significance of NP-Completeness to Big-O, Big-Omega, and Big-Theta complexity
	G5	Explain the difference between the complexity of a problem and the complexity of a particular algorithm for solving that problem
Space Complexity	H1	Compare and contrast space and time complexity.
	H2	Discuss the tradeoffs in algorithm performance with respect to space and time complexity. E.g., Compare and contrast the space requirements for a linked list (single, double) vs. an array-based implementation.
	H3	Compare and contrast the space requirements for Mergesort versus Quicksort.
Memory Layout	I1	Describe general layout of program memory (e.g. the locations or program, stack, and heap).
	I2	Diagram how the stack and heap grow in relation to each other in the context of a code example
	I3	Explain how stack overflow may arise as a result of recursion.
	I4	Explain the low level implementation of methods calls and returns by describing an activation record and how it is pushed and popped from the stack
Linked Lists (Including Stacks, Queues, and Deques), Introduction to Pointers	J1	Differentiate an abstraction from an implementation.
	J2	Define and give examples of problems that can be solved using the abstract data types stacks, queues and deques.
	J4	Compare and contrast the implementations of these abstract data types using linked lists and circular arrays in C++.
	J5	Demonstrate how dynamic memory management is handled in C++ (e.g., allocation, deallocation, memory heap, run-time stack)
	J6	Gain experience with pointers in C++ and their tradeoffs and risks (dangling pointers, memory leaks)
	J7	Explain the difference between the complexity of a problem (sorting) and the complexity of a particular algorithm for solving that problem
	J8	Manipulate data in stacks, queues, and deques (irrespective of any implementation).
Insertion Sort, Mergesort, Quicksort	K1	Describe and apply various sorting algorithms; Compare and contrast their tradeoffs.
	K2	State differences in performance for large datasets versus small datasets on various sorting algorithms.
	K3	Analyze the complexity of these sorting algorithms.
	K4	Explain the difference between the complexity of a problem (sorting) and the complexity of a particular algorithm for solving that problem
	K5	Manipulate data using various sorting algorithms (irrespective of any implementation).
Introduction to Trees and Tree Traversal	L1	Determine if a given tree is an instance of particular type (e.g. heap, binary, etc.) of tree
	L2	Describe and use pre-order, in-order and post-order tree traversal algorithms.
	L3	Describe the properties of binary trees, binary search trees, and more general trees; and implement iterative and recursive algorithms for navigating them in C++.
	L4	Compare and contrast ordered versus unordered trees in terms of complexity and scope of application.
	L5	Insert and delete elements from a binary tree.
Priority Queues, Heaps, Heapsort	M1	Provide examples of appropriate applications for priority queues and heaps.
	M2	Manipulate data in heaps (irregardless of any implementation).
	M3	Describe the Heapify and Heapsort algorithms, and analyze their complexity.
Hashing	N1	Provide examples of the types of problems that can benefit from a hash data structure.
	N2	Compare and contrast open addressing and chaining.
	N3	Evaluate collision resolution policies.
	N4	Describe the conditions under which hashing can degenerate from O(1) expected complexity to O(n).
	N5	Identify the types of search problems that do not benefit from hashing (e.g., range searching) and explain why
	N6	Manipulate data is hash sturctures both irrespective of implementation and also within a given implementation
B+ Trees	O1	Describe the structure, navigation and complexity of an order m B+ tree.
	O2	Insert and delete elements from a B+ tree.
	O3	Explain the relationship among the order of a B+ tree, the number of nodes, and the minimum and maximum capacities of internal and external nodes.
	O4	efficiently)
	O5	Compare and contrast B+ trees and hash data structures.
	O6	Explain and justify the relationship between nodes in a B+ tree and blocks/pages on disk
	O7	Justify why the number of I/Os becomes a more appropriate complexity measure (than the number of operations/steps) when dealing with larger datasets and their indexing structures (e.g., B+ trees).
Counting: Product Rule, Sum Rule, Inclusion-Exclusion, Tree Diagrams,	P1	Apply counting principles to determine the number of arrangements or orderings of discrete objects, with or without repetition, and given various constraints.
	P2	Use appropriate mathematical constructs to express a counting problem (e.g. counting passwords with various restrictions placed on the characters within).
	P3	Identify problems that can be expressed and solved as a combination of smaller sub problems. When necessary, use decision trees to model more complex counting problems
Binomial Theorem, Combinatorial Identities	Q1	Solve problems using combinatorial arguments and algebraic proofs.
	Q2	State the relationship among recursion, Pascal's Triangle, and Pascal's Identity.
Binomial Distribution and Basic Probability (new)	R1	Define binomial distribution and identify applications.
	R2	Model and solve appropriate problems using binomial distribution.
	R3	Apply basic probability theory to problem solving, and identify the parallels between probability and counting.
Pigeonhole Principle	S1	Define various forms of the pigeonhole principle; recognize and solve the specific types of counting and hashing problems to which they apply
Graph Theory: Introduction and	T1	Describe the properties and possible applications of various kinds of graphs (e.g., simple, complete), and the relationships among vertices, edges, and degrees.
	T2	Prove basic theorems about simple graphs (e.g. handshaking theorem).
Graph Representation, Isomorphism, Graph Connectivity	U1	Convert between adjacency matrices / lists and their corresponding graphs.
	U2	Determine whether two graphs are isomorphic.**
	U3	Determine whether a given graph is a subgraph of another.
Euler/Hamilton Paths/Cycles**	V1	Compare and contrast Euler and Hamilton paths/cycles.
	V2	Given an arbitrary graph, determine whether or not a Hamilton path, Hamilton cycle, Euler path, or an Euler cycle exists, and if so, provide an example.
Graph Traversals	W	Perform breadth-first and depth-first searches in graphs.
	W	Explain why graph traversals are more complicated than tree traversals.
Planar Graphs**	X1	Describe the properties and possible applications of planar graphs.
	X2	Use Euler's Formula to solve given planar graph problems.
	X3	Apply the notion of graph colourability to determine if a k-colouring exists for a particular graph

**Optional