CPSC 111 Course Learning Goals

| By the end of the course students can… | 1. Write and modify code to "express understanding" of basic programming constructs (including sequential exeuction, conditional execution, iteration, arrays, methods/parameter passing, object-orientation and inheritance principles). | 2. Read and hand-execute (trace) provided code to "express understanding" of basic programming constructs and memory models (including sequential exeuction, conditional execution, iteration, arrays, methods/parameter passing, object-orientation and inheritance principles). | 3. Write code to solve moderately-difficult problems (moderately difficult will be defined through example in an appendix) | 4. Recognize, create, and manipulate various models of programs including memory tracing and UML class diagrams | 5. Explain Java language features (e.g. classes, visibility, fields, and methods) which support OO design principles such as modularity, encapsulation, abstraction and inheritance. | 6. Explain the major components of a computing system and how a program compiles and executes to a non-computer scientist. |
|---|---|---|---|---|---|---|
| Computing Systems (2) | | | | | | A, B |
| Programming Language Basics (4) | C, D | C | | | | E, (F?) |
| Classes and Objects (3) | G, H | G, H, I | | | I | |
| Conditionals (3) | J, K, L | J, K, L | (J?, K?), L | | | |
| Designing and Defining Classes (4/2) | M, N, O, (Q?) | (O?), R | N, Q | O, R | M, P | |
| Iteration (3) | S, U | T | S, U | T | | |
| Arrays (3,1) | V, X | | V, W, X | | | |
| Sorting (2) | | Z | | (Z?) | | AA* (not done by everyone who teaches course) |
| Advanced Class Design (3) | AB1, AC1, AC3 | AB4, AC4, AC5 | AD | | AB2, AB3, AC2, (AC4?), (AC5?) | |
| Graphics (2) | AE2, AF | | | | AE1 | |

| Topic | ID | Assessed in? | Goals |
|---|---|---|---|
| | | | **Students can…** |
| Computing Systems (2) | A | M1 | define and give real world examples of key components of the computer (input, output, processor, memory). |
| | B | | can distinguish and describe how layers of abstraction are supported in computing problem solving through algorithms, programming languages, assembly, and computer hardware. |
| Programming Language Basics (4) | C | M1, M2, F, L, A | apply with basic competence simple programming constructs such as sequential execution, variable typing and declaration, naming, algebraic operations, operation precedence. |
| | D | M1, M2, F, L, A | create programs which translate explicit English problem statements (an algorithm) into short series of sequential Java instructions. |
| | E | | describe the multiple ways in which a natural language paragraph can be interpreted and contrast to the single way an algorithm can be interpreted. |
| | F | | explain why a particular numeric type can only represent numbers in a particular range. |
| Classes and Objects (3) | G | M1, M2, F | define the relationship between classes and objects. |
| | H | M1, M2, F, L, A | read and write code utilizing the API of key Java classes (e.g. String, Scanner). |
| | | | explain how control flow and data pass on a method call. |
| | I | | identify specific standard methods like accessors and mutators and describe why these operations are needed for non-primitive data types. |
| Conditionals (3) | J | M2, F, L, A | hand-trace and create programs which use if-statement conditionals to model behavior of input-driven programs. |
| | K | M2, F, L, A | utilize Boolean expressions, relational, and logical operators to control conditional execution. |
| | L | M2, F, L, A | utilize block statements, short-circuit evaluation(?), and nested ifs to create code to solve problems in Java. |
| Designing and Defining Classes (4/2) | M | M1, M2, F, L, A | create a simple class (with instance variables, accessors and setters) utilizing basic components of OO design (including encapsulation, visibility modifiers, and overloading) to model a real world entity (including it's actions and state). |
| | N | M2, F, L, A | use that class in a simple program. |
| | O | M2, F, | apply their understanding of references and objects by writing standard constructors and drawing diagrams of memory after an object is constructed. |
| | P | | explain how encapsulation (as implemented with visibility modifiers) supports data integrity and good interface design. |
| | Q | M2, F, L, A | apply with more expert competence simple programming constructs such as sequential execution, variable typing and declaration, naming, algebraic operations, operation precedence. |
| | R | | describe how reference objects differ from primitive variables and describe problem solving scenarios which are best supported by each. |
| Iteration (3) | S | M2, F, L, A | solve problems by creating code where repeated actions are controlled with looping structures (for and while loops). |
| | T | M2, F, L, A | identify and debug a loop that never stops (an infinite loop). |
| | U | M2, F, L, A | solve problems which requires a loop within a loop where the inner loop iteration does not depend on the outer loop iterator (e.g. to draw a rectangle of stars). |
| | | | solve problems which requires a loop within a loop where the inner loop iteration does depend on the outer loop iterator (e.g. to draw a triangle of stars). |
| Arrays (3) | V | M2, F, L, A | solve problems with collections of same-type data using arrays (including primitive type collections (e.g a collection of class grades) and collections of objects (e.g. a collection of String names or a deck of cards). |
| | W | M2, F, L, A | apply with more expert competence branching, looping, and nested loops through practice solving problems using arrays and 2-D arrays. |
| | X | M2, F, L, A | solve problems by creating code which require the creation and use of 2-D arrays (e.g. graphics and averaging scores of students and other data that can be stored in matrix form). |
| Sorting (2) | Z | F | identify a simple sorting algorithm. |
| | AA | F | explain that a simple sorting algorithm can be analyzed through simple techniques such as comparison counting and that different sorting algorithms can have different execution time costs and that the number of elements sorted is important in making these analyses. |
| Advanced Class Design (10) | AB1 | M2, F, L, A | create codes which require the use of advanced class syntax and semantics including static methods and variables, scoping, primitive and non-primitive parameter passing. |
| | AB2 | | explain the difference between static and non-static fields and give an example of when each should be used. |
| | AB3 | | explain the difference between static and non-static methods and give an example of when each should be used. |
| | AB4 | | given a piece of code, identify the scope of a variable (locals, class-level, or global). |
| | AC1 | F, L, (A) | create codes which require the use of advanced OO concepts such as inheritance, class hierarchy, and polymorphism. |
| | AC2 | | explain how inheritance is a form of code re-use that can be valuable in large systems. |
| | AC3 | | given a parent class and a specification for a subclass, implement the subclass, including method overriding, calls to the super class constructor and calls to the super class's version of the overwritten method. |
| | AC4 | | explain what happens when polymorphic assignment happens. |
| | AC5 | | explain what happens when a polymorphic method call is made. |
| | AD | F, L, (A) | apply with more expert competence class design and usage through practice with programs implementing inheritance, class hierarchy and polymorphism. |
| Graphics (3) | AE1 | | explain how graphics applications use inheritance and interfaces. |
| | AE2 | | create codes which require the use of basic graphical user interface APIs in Java. |
| | AF | | create codes which utilize an event-driven execution model. |

| By the end of the course, students can… | 1. Apply the formal systems we discussed to model computational systems (like programs and circuits), including reasoning about them, proving relevant properties, and communicating about them clearly and precisely with fellow Computer Scientists. Learn and apply new formalisms, specifically be able to connect between features and conclusions in the formal and informal (English language, sketch-based, pseudo-code, etc.) representations. | 2. Justify the behaviour and correctness of some algorithms (e.g. at the level of selection sort and recursive binary searc or quicksort), but especially for algorithms with singly and doubly nested loops in order to prove them correct or bound their running time. | 3. Translate easily among English language, simple formal representations (i.e., propositional and shallowly nested predicate logic statements), and closely related equivalent formal representations (in order to identify alternate methods to solve or simplify a variety of problems, such as writing conditionals, as you work with them). Write proofs for simple theorems by translating the theorem into first-order logic, decomposing the statement into its components, and then using the proof techniques discussed in class (direct proofs, indirect proofs by contrapositive, indirect proofs by contradiction, proofs by weak and strong mathematical induction). | 4. Read a proof, and justify why each step of the proof is correct. | 5. Create regular expressions and DFAs to solve problems that are important to them in programming. |
|---|---|---|---|---|---|
| Propositional Logic and Circuits (3) | C | | A, B | | |
| Proofs (4) | (G)? | (F)? | D, **E,** F | G | |
| Arithmetic Circuits (2) | H, I | | | | |
| Sets and functions (2) | J | | K | | |
| Finite Automata (3) | L, M, N | | | | L, M |
| Induction (3) | | O, P | O | Q | |
| Relations (1) | R | | R | | |

| Topic | ID | Assessed in? | Learning Goals<br>Students can… |
|---|---|---|---|
| Propositional Logic and Circuits | A | Implicitly assessed with B, *should* be assessed on a quiz, assignment | express simple natural language statements using propositional logic. |
| | B | midterm (sometimes on a quiz, but too long) | distinguish between statements that express the same information about the world versus statements that don't using logical equivalences. |
| | C | Lab(1-2), quiz or midterm, sometimes assignment | translate back and forth between propositional logic statement and circuits that assesses the truth or falsehood of those statements. |
| Proofs | D | F6a, midterm, assignments (with variety of domains), quiz | express natural language statements which require the use of predicate logic to describe, for example, the result of algorithms that use loops. |
| | E | Assignment, sometimes quiz | make statements about the relationships between properties of various objects (e.g. every candidate got votes from at least three people in every province). |
| | F | F5, F6b, F7b, F9, quizzes, assignments, midterm | create simple direct and indirect proofs, to be able to prove the correctness of operations that can be performed in programs. As another example, supports the development of data type representations (e.g. rational numbers). |
| | G | Not directly assessed (now one on quiz and one assignment), maybe occasionally on an assignment. Suggestion, use the web. | evaluate when a proof fails to satisfy as a communication between people – that is identify inaccuracies or missing steps in proofs. |
| Arithmetic Circuits | H | F1a, F1b, labs a lot, lightly on assignment | describe how the arithmetic operations of the computer break down into simpler logical operations as this is understanding one step of the layered structure of computers. |
| | I | F1b?, F3a, F3b, F3c, lab, breakdown not assessed otherwise | recognize why the numerical systems that we work with on computers behave the way they do, especially in cases where they break down such as floating point representation being inaccurate, overflow, and limitations of integral numerical types (longs, ints, etc.). |
| Sets and Functions | J1 | F2a (simpler), F2b, F7a, F7b, not | apply previously developed formalism to proofs about sets and functions as applied in Java collection classes and in databases. |
| | J2 | really the application to Java or | give examples of function that have certain properties and vice versa (e.g. injective, surjective, bijective). |
| | K | Continue to do questions like D/E and they understand better. Assignments, quizzes | more precisely explain the meaning of quantified statements. (elaboration of D/E) |
| Finite Automata | L | F11a, lab (adding a new one), assignment, | model and solve real world problems such as control circuits (traffic lights), matching problems, validating input, and (in the abstract) modeling the capabilities of a computer using real circuits/DFAs. |
| | M | F11b, assignment, quiz, lab | Students can create regular expressions which produce DFAs to solve problems that are important to them in programming. |
| Induction | O | F8(not prog), F10b but easier, too hard to assess, not convinced that we have a simple enough problem that they can do. Save for 221., assignment (a lot), quiz | prove things about programs that the use loops and recursion. |
| | P | F8(not prog) | justify the correctness of a reasonably complex recursive algorithm (like quicksort or mergesort). An example of O. |
| | Q | F8(not prog), F10a, talk a lot about in class, but not on assignment, the application can just be done mechanically (NOT ASSESSED BEFORE FINAL) | be able to list out the exhaustive steps from a proof that should prove that -- given a property that they want to prove and given any specific value to prove that property at. |
| Relations | R | Sometimes we get to it and sometimes we don't. | prove that a relation is symmetric, transitive or reflexive. |

CPSC 211 Course Learning Goals

| After this class students can… | Move from personal software development methodologies to professional standards and practices (e.g. create programs that interact with their environment (files etc.) and human users according to standard professional norms). | Given an API, write code that conforms to the API to perform a given task. | Identify and evaluate trade-offs in design and implementation decisions for systems of an intermediate size. | Read and write programs in Java using advanced features | Extend their mental model of computation from that developed in CPSC111 | Work with an existing codebase, including reading and understanding given code, and augment its functionality. [Happens only with assignments] |
|---|---|---|---|---|---|---|
| **Programming by contract** | A1, A2, A3, A4 | | | | | |
| **Exception handling** | B1, B5 | | B1, B6 | B2, B3, B4, B5 | | |
| **Streams, I/O** | C3 | | | C2, C3 | C1 | |
| **Testing** | D1, D2, D3 | | | D4 | | |
| **Software Design** | E2, E3, E4, E5, E6 | | E1, E7, E8, E10 | E9 | | |
| **Java Collections Framework** | | F3, F8, F11, F15, F18, F19 | F1, F2, F4, F12, F16, F20 | F3, F6, F7, F10, F13, F17, F21 | | |
| **Graphical User Interfaces** | G1 | | G1 | G2, G3, G5, G6 | G4 | |
| **Multi-threaded programming** | | H6 | | H4, H5, H6 | H1, H2, H3 | |
| **Recursion** | | | I5 | I1, I4, I6 | I2, I3 | |
| **Implementing basic collection classes** | | | | J1, J2, J3 | | |

| Topic | ID | Assessed in? | Students can: |
|---|---|---|---|
| **Programming by contract** | A1 | | write client code that adheres to the contract specified for a class using invariants, preconditions and postconditions |
| | A2 | | implement a class given a contract specified by invariants, preconditions and postconditions |
| | A3 | | describe the benefits of programming by contract for client and developer |
| | A4 | | use assertions appropriately in code |
| **Exception handling** | B1 | | incorporate exception handling into the design of a method's contract |
| | B2 | | trace code that makes use of exception handling |
| | B3 | | write code to throw, catch or propagate an exception |
| | B4 | | write code that uses a finally block |
| | B5 | | write code to define a new exception class |
| | B6 | | compare and contrast checked and unchecked exceptions |
| **Streams, I/O** | C1 | | describe stream abstraction used in Java for byte and character input/output |
| | C2 | | write programs that use streams to read and write data |
| | C3 | | incorporate data persistence in a program using Java's serialization mechanism |
| **Testing** | D1 | | compare and contrast blackbox and whitebox testing (at the level of what each type of testing provides) |
| | D2 | | use blackbox testing with equivalence classes to test a method and from that a suite of test cases |
| | D3 | | describe how unit testing is applied to a class (describe a hierarchy of tests that you could apply) |
| | D4 | | write a suite of tests to apply unit testing to a class using JUnit (putting the above into practice with a particular tool) |
| **Software Design** | E1 | | describe the basic design principles of low coupling and high cohesion |
| | E2 | | design a software system (expressed in UML) from a given specification that adheres to basic design principles (lc and hc) |
| | E3 | | interpret UML class diagrams to identify relationships between classes |
| | E4 | | draw a UML class diagram to represent the design of a software system |
| | E5 | | describe the Liskov Substitution Principle |
| | E6 | | explain whether or not a given design adheres to the LSP |
| | E7 | | incorporate inheritance into the design of software systems so that the LSP is respected |
| | E8 | | compare and contrast the use of inheritance and delegation |
| | E9 | | use delegation and interfaces to realize multiple inheritance in design (e.g. to support the implementation of multiple types) |
| | E10 | | identify elements of a given design that violate the basic design principles of low coupling, high cohesion, the LSP |
| **Java Collections Framework** | F1 | | use big-O notation to categorize an algorithm as constant, linear, quadratic or logarithmic time |
| | F2 | | given two or more algorithms, rank them in terms of their time efficiency |
| | F3 | | program to the generic List interface including read and use the List API (e.g. use Lists in ways similar to arrays) |
| | F4 | | compare and contrast ArrayList and LinkedList implementations of the List interface |
| | F6 | | compare and contrast assignment with various generic collections under specific subclass scenarios |
| | F7 | | use wildcards appropriately in generic type parameters to enable assignment in sub and super class scenarios |
| | F8 | | program to the generic Iterator and ListIterator interfaces including reading and using the APIs |
| | F10 | | read and write code that uses a for-each loop to iterate over a collection |
| | F11 | | program to the generic Set and SortedSet interfaces inclduign read and use the API |
| | F12 | | compare and contrast the HashSet and TreeSet classes (benefits of using each, basic run time analysis) |
| | F13 | | design and implement a class in such a way that it can be used with the Java collections framework (overrides equals in HashCode, implement the generic Comparable and Comparator interfaces to account for multiple sorting criteria) |
| | F15 | | program to the generic Map and SortedMap interfaces by reading and using the API |
| | F16 | | compare and contrast HashMap and TreeMap classes (benefits of using each, basic run time analysis) |
| | F17 | | write code (solve problems) that uses the generic algorithms provided in the Collections class |
| | F18 | | program to the generic Queue interface |
| | F19 | | program to the API of the generic Stack class |
| | F20 | | identify (in words or through code) appropriate types for collections of data needed in a given software system |
| | F21 | | write code that implements unidirectional, bidirectional, 1-1 and 1-many associations |
| **Graphical User Interfaces** | G1 | | describe basic principles of good user interface design (user interface hall of shame) |
| | G2 | | use layout managers to produce a well designed GUI |
| | G3 | | write code to produce a well designed GUI that includes frames, panels, menus and buttons |
| | G4 | | describe the event driven model |
| | G5 | | describe and apply scoping rules that apply to the use of inner classes |
| | G6 | | write code that uses inner classes (including anonymous inner classes) to handle events raised by GUI elements |
| **Multi-threaded programming** | H1 | | Describe the multi-threaded programming model including thread scheduler, thread priority, and time slices. |
| | H2 | | describe the various states that a Java thread can achieve and the events that lead to transition from one state to another |
| | H3 | | define the terms deadlock, race condition and critical section |
| | H4 | | identify possible legal traces of a multithreaded program |
| | H5 | | identify deadlock and race conditions in a multithreaded program |
| | H6 | | write a thread-safe class using Lock and Condition objects |
| **Recursion** | I1 | | trace code that uses recursion to determine what the code does |
| | I2 | | draw a recursion tree corresponding to a recursive method call |
| | I3 | | draw a stack trace of code that uses single and multi-branch recursion |
| | I4 | | write recursive methods |
| | I5 | | compare and contrast iterative and recursive solutions to a problem |
| | I6 | | replace a recursive implementation of a method with an iterative solution that uses a stack to model the run-time stack |
| **Implementing basic collection classes** | J1 | | write code to perform search, insertion and removal operations on a singly or doubly linked list |
| | J2 | | implement a class (e.g., list, stack or queue) that stores data in a linked list |
| | J3 | | implement a class (e.g., list, stack or queue) that stores data in an array |

| After this class students can… | Be a better programmer because, you will have a deeper understanding of the features of a programming langauge in order to be able to a) understand in detail how your programs are executed, b) be able to more easily learn new programming languages and c) be able evaluate design tradeoffs in considering languages most appropriate for solving a given problem. | Appreciate that system design is a complex set of tradeoffs which, while are important to be able to analyze will not have exactly one optimal answer (while there are often many sub-optimal answer). Tradeoffs exist at a range of levels including the hardware level, programming language level, etc. Experience with these tradeoffs prepares the student to deal with tradeoffs in desin in real world programming scenarios. | Develop distinctions between the static and dynamic components of programs and systems and be able to describe their implications. | Utilize synchronization primitives to control interaction in various situations including among processes, threads, and networked communication. | Understand how computing systems work including networking. |
|---|---|---|---|---|---|
| ALU/Registers/ Memory | A1 | | | | A1 |
| Machine Level Instructions | B1, B2, B6 | B6 | B1, B6 | | B2, B3, B4, B5, B7 |
| ISA Design | | C1, C2, C3, C4 | | | |
| Variables | D1, D2, D3 | D1 | D1, D2, D3 | | |
| Flow of Control | E4, E5, E6 | E5, E6 | E3, E4 | | E1, E2, E3, E5 |
| Language Design and Tradeoffs | F1, F2, F3, F4 | F1, F2, F3, F4, F5, F7, F8, F9 | F1, F3 | | F4, F7 |
| External Devices | | G1 | | | G1 |
| Devices and Files | H1, H8 | H1, H4, H7 | | | H1, H2, H3, H5, H8 |
| Networking | I2, I3 | | | I2, I3 | I1, I4 |
| Processes | J12, J13 | J2, | | J6, J7, J8, J9, J10 | J1, J2, J3, J4, J6, J9 |
| Java and C comparative understanding* | K1, K2, K3, K4, K6, K7, K8, K10, K11 | K6, K8, K9, K10 | | | K5, K9 |

| Topic | ID | **Learning Goals Students Can…** |
|---|---|---|
| ALU/Registers/ Memory | A1 | Describe a basic computer with basic components (ALU, Registers, Memory) and explain how instructions execute and data flows. |
| Machine Level Instructions | B1 | Trace execution of a simple C program and translate to a set of machine level instructions to emulate that C program |
| | B2 | Identify and group Gold Assembly instructions based on their utility for programming(control flow of execution, access memory, arithmetic operations, etc.) |
| | B3 | Describe in what ways instructions and data are the same at the bit level. |
| | B4 | Translate a Gold Assembly instruction into machine representation (in bits) |
| | B5 | Decipher according to Gold Assembly language rules the various parts of an instruction (opcode, operands, etc) from the bit |
| | B6 | Identify what information is available to an instruction statically and what must be calculated dynamically at run time. For example, instructions are created ahead of time and live in memory and are static but that the data they access, including the memory addresses to be accessed may be only calculated or available at run time |
| | B7 | Recognize that subtracting a number from another involves taking the twos complement of the number and adding it. Be able to apply the principles of twos complement to be able to correctly implement sign extension.* |
| ISA Design | C1 | Describe the minimal set of addressing modes needed for an instruction set to be complete. |
| | C2 | Compare and contrast various addressing modes (e.g. the limitations of not supporting a particular mode in an instruction set, why dynamically generated addressing is necessary). |
| | C3 | Compare and contrast the performance impact of addressing modes -- specifically be able to discuss the design trade offs in instruction size, memory versus register access, and direct versus indirect addressing. |
| | C4 | Evaluate tradeoffs in instruction set design. This involves discussion of minimalness, orthogonality, and simplicity, and performance. This should be done for pairs of instructions up to the point of evaluating the differences in CISC and RISC instruction sets. |
| Variables | D1 | Describe the differences between dynamic and static variables in terms of what the compiler can do for each in creating assembly instructions. |
| | D2 | Give examples of both dynamic and static variables in both Java and C |
| | D3 | State for different kinds of variables what information is statically known and what information is dynamically known. |
| Flow of Control | E1 | Keep track of program counter when code using control flow (jumps) is executed |
| | E2 | Calculate jump targets based on the address of the program counter. |
| | E3 | Explain why conditional control flow (loops) is needed enable static programs to compute dynamically sized results. |
| | E4 | Compare and contrast scenarios which require static versus dynamic jump targets. |
| | E5 | Give C or Java code examples which require direct versus indirect jumps and vice versa |
| | E6 | Describe how performance can be affected by dynamic jumps (e.g. be able to show how you can use jump tables to make switch statements faster) |
| Language Design and Tradeoffs | F1 | Explain why procedure return in C/Java must be dynamic – consider the case of a programming language whose procedure RETURN was a static jump |
| | F2 | Explain the consequences to programming if local variables were allocated statically |
| | F3 | Explain the consequences to programming of eliminating dynamically allocated local variables and/or dynamic return. |
| | F4 | Explain the advantage of using the stack for local variables as opposed to just using the heap, including describing how the stack is not required (e.g. you can just have a heap – and that having the stack is a design tradeoff). |
| | F5 | Show how procedure call implementation is different if you use the heap instead of the stack. |
| | F6 | [Understand advantage of maintaining a closure after a procedure returns and that this would require using the heap instead of the stack. Advanced students only |
| | F7 | Show the machine instructions necessary to implement a procedure call and return and describe the format of the stack |
| | F8 | Explain why a procedure-calling convention exists and the design tradeoffs of having it implemented by the compiler and not imbedded in the instruction architecture alone |
| | F9 | Explain how the independence of callers and callees complicates the planning of register usage (e.g. what values to store in register). For example, describe how storing all values in the caller is rarely optimal. |
| | F10 | [Develop a heuristic that a compiler could use to determine when to use a callee-save register and alternatively when to use a caller-save register by giving examples in machine code that benefit from each choice.] |
| External Devices and Files | G1 | Explain what PIO and DMA are and how they differ and are similar to each other |
| | H1 | Explain what disk drive characteristics contribute to how quickly information can be retrieved from disk |
| | H2 | Calculate average disk access time |
| | H3 | Explain how sectors are identified (head, track sector) |
| | H4 | Explain and compare the tradeoffs disk scheduling algorithms make |
| | H5 | Describe and draw pictures of the UNIX file system, basic building blocks and on disk data structures including blocks, inodes, and files |
| | H7 | Apply knowledge about disk performance characteristics to data layout on disk |
| | H8 | Explain how failure of the OS impacts various structures in the file system – at various points of time of failure, depending on the status of the write in a file system. (this will likely be going away). |
| Networking | I1 | Compare and contrast the communication model for procedures on a single machine (the procedure call model) versus networked communication (these differences include: make a connection, transfer data, shut down the connection). |
| | I2 | Write a simple networked program (e.g. perhaps a very simple web server involving a client and server getting connected), including gaining familiarity with networking APIs. |
| | I3 | Describe how networked communication follows an asynchronous communication model in which synchronization needs to be handled explicitly |
| | I4 | Describe how sending a stream of data across a network involves chopping that stream into chunks, sending them independently, chunks can get lost, and that reliability issues arise and must be dealt with. Describe the role that a protocol plays in abstracting these issues. |
| | I5 | [Protocol stack and layering (design, layers of abstraction)not covered in 213] |
| Processes | J1 | Explain that there is a private address space for each process and that that hardware does the translation (via base-bounds). |
| | J2 | Explain the design tradeoffs of why virtual addressing is needed and desirable and also the complicating and performance implications. |
| | J3 | Explain that processes are separate entities with their own address space and that if two processes access the same address location it's different and that this is an example of virtual memory. |
| | J4 | Describe a motivation for processes based on an example of why we need to move from asynchronous access to concurrent access with synchronization primitives |
| | J5 | [Describe at a basic level the tradeoffs (via analysis with examples of round robin, the role of the kernel clock, pre-emption, interrupts) available for scheduling of processes.] |
| | J6 | Trace though code with a producer/consumer relationship. |
| | J7 | Use synchronization primitives to enable mutual exclusion access, e.g using semaphores to control access to a shared array. |
| | J8 | Use synchronization primitives to enable signaling in producer/consumer structures |
| | J9 | Explain how threads and processes differ specifically with regards to shared memory |
| | J10 | Describe real world scenarios which require the use of concurrency via multi-threading |
| | J11 | [Would like: Explain how processes converts asynchrony into concurrency by using synchronization primitives. Interrupts. |
| | J12 | Compare and contrast the synchronization features that students already know from Java with the variations available in C and Unix |
| | J13 | Compare and contrast the threading features that students already know from Java with the variations available in C and Unix |
| | J14 | [Explain how when a program has more than one lock, that it introduces the possibility of deadlock. Give an example of a code that has the possibility of deadlocking and a different example with live-locking. Explain the tradeoffs associated with different granularities of locking. Can explain the standard dining philosophers' problem. Priority inversion and techniques for dealing with it?] |
| Java and C comparable | K1 | Write C code equivalent to known Java code (for the subset of C that is basically the same in both languages – primarily the imperative structures and primitive types |
| | K2 | Describe how arrays are different in C and Java (C arrays are static and Java arrays are dynamic) |
| | K3 | Use C syntax for pointers and compare that to reference variable use in Java. |
| | K4 | Describe the similarities and differences between C structs and Java objects and specifically how their features are addressed in assembly code |
| | K5 | Do pointer arithmetic in C. |
| | K6 | Describe that dynamic memory allocation is the same in C and Java but that type safety is different |
| | K7 | Describe how memory reclamation is different and be able to write C programs that use memory reclamation |
| | K8 | Describe how garbage collectors only solve one of these two memory problems: dangling pointers and memory leaks (including being able to describe these two problems and give code examples which would create them). |
| | K9 | Create a jump table to implement a C switch statement. |
| | K10 | Describe why Java's polymorphism required indirect jumps and discuss the performance implications of that |
| | | Read and understand basic C programs. |

| 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