

**The CS2110 Final is optional.** As soon as possible, we will post grades for A6 and tentative course letter grades. You will answer a 1-question assignment on the CMS: Do you accept the letter grade or will you take the final? This question will become available at the same time that we post the tentative letter grades.

If you walk into the final room, you must complete the final. For example, you may not decide half way through the final that you don't want to take it.

A few students missed one of the two prelims. These students are required to take the final unless other arrangements have been made.

Taking the final may lower or raise your course grade. Our past experience: taking the final changes the letter grade for very few —fewer than 10 in a course of 200, mostly raises.

**Conflicts.** If there is a chance you may take the exam and you have a conflict, please email Maria Witlox, [Mpr13@cornell.edu](mailto:Mpr13@cornell.edu), immediately. How we handle conflicts depends on how many conflicts there are. We define “conflict” as having another final at the same time or having more than two finals within a 24-hour period, thus following university policy.

**Quiet room / extra time.** If Cornell allows you extra time or a quiet room for the exam, please email Maria Witlox, [Mpr13@cornell.edu](mailto:Mpr13@cornell.edu), so that we know how many there are of you. You will take the exam in Gates 114. Students needing extra time should start at 6PM.

**Review session 1.** Fri, 9 May. 1:00–3:00. Phillips 101.

**Overall length and balance of the exam.** Similar to past CS2110 exams, but it will be 150 minutes long.

**Programming.** Being able to write correct code in Java is absolutely critical. You should have skill in using arrays, Strings, loops (while, for, and for-each), writing procedures, functions, constructors, etc. You will be asked to write at least one recursive method.

**JAVA.** Know the basics of Java. Use the summary given for both prelims and the JavaSummary (ppt slides) on the course website. You may be asked about anything in these references. In addition, know about inner classes and nested static classes and about enums.

**Java Collection Hierarchy.** Basic understanding of how to use `Collection<T>`, `List<T>`, `Set<T>`, `ArrayList<T>`, `Vector<T>`, `HashSet<T>`, and `HashMap<T>`. Know the basic methods in each: how to create an object of that class, add an element, remove an element, determine the size (number of elements). If a question requires any methods other than the basic ones, they will be given to you.

**Specific Java Interfaces.** Be able to use interface `Comparable` (what method is required?).

Know that if a `Collection` or one of its subinterfaces/subclasses implements interface `Iterable`, the `foreach` statement (e.g. `for (Integer d : arraylistofinteger) {...}`) can be used to iterate over such collections.

Know how to use interface `Iterator<T>`.

**JUnit testing.** Know how to write a procedure to test a method, using procedure `assertEquals(expected-value, computed-value)`.

**Javadoc.** What it is. What a javadoc comment is. Where one places javadoc comments. What javadoc extracts from a class (or bunch of classes) and puts into an html page.

**Java GUI.** Be able to read and understand basic GUI code. Know that after adding components to the GUI method `pack()` must be called.

Know the three basic containers and their layout managers and how they work: `JFrame` uses a `BorderLayout` manager, `JPanel` uses a `FlowLayout` manager, and `Box` uses a `BoxLayout` manager. Know how to add a component to these containers and how the components appear in the GUI.

Know the three basic things needed to be able to listen to some event: (1) implement a particular interface, which means, (2) writing the method that will be called when the event happens, and (3) registering an object that contains the method with the component on which the event occurs.

We do not expect you to remember the interface and method needed for each kind of event. It's the overall process, the concept, that is important.

**Analysis of algorithms.** Big-O complexity notation, the definition of  $O(f(n))$ . Definition of best-case, worst-case, and expected (average) case order of execution. The notion that this may count execution not of every statement but important actions (e.g. number of array comparisons, number of array-element swaps).

**Proofs.** Rigorous arguments for establishing the big-O complexity of algorithms. Be prepared to show that an algorithm is correct through clear logical reasoning.

**Searching.** Linear search. Binary search in an ordered array. Be able to write these. Know their expected- and worst-case time (which of these is  $O(n)$  or  $O(\log n)$  for an array of size  $n$ ?)

**Sorting.** Understand the following sorting methods, their worst-case and expected-case execution times, and their space complexity: insertion sort, selection sort, merge sort, quick sort, heap sort. Be able to write each

of these, perhaps leaving parts of the algorithms in English (see end of this review for what we expect).

Concept of stable sorting.

Lower bound for comparison sorts.

**Abstract data types (ADTs).** An ADT is just a set of values together with (primitive, or predefined) operations on them and the idea that we may expect the primitive operations to have certain properties, e.g.  $O(1)$  cost and  $O(n)$  space. How we can define an ADT in Java by declaring an interface. Know the meaning of *amortizing* the cost of an operation.

**Knowledge of particular ADTs.** Know the following ADTs and their basic operations: stack, queue, set, priority queue, heap.

Be able to design and write: An implementation of a stack in an array; an implementation of a queue in an array using wraparound, so that removing the first and putting something at the end are constant time operations; an implementation of a heap and priority queue in an array.

**Linked lists.** Describe the data structure used to implement singly linked lists, doubly linked lists, and circular lists, and each of these with a header. Know the advantages of each. Be able to write code on how to insert or delete an element.

**Hashing.** The idea of a hash function and the basic idea of hashing. Solving collisions using linear or quadratic probing; why, when using such probing, one can't simply remove an element by setting the array element in which it appears to null. Load factor, and the fact that if it is  $\frac{1}{2}$ , at most 2 probes are expected in searching for a value. Solving collisions using a linked list of values that hash to the same address.

**Trees.** Definition of a tree and the general terminology associated with trees —node, child, sibling, parent, leaf. Notion of a balanced tree.

Data structures to implement trees. Be able to write a tree traversal —both breadth-first and depth-first. Be able to write code to perform operations on trees, such as finding the depth of tree, finding an element, etc.

Binary search trees (BSTs): Definition. Write algorithm to search a BST for a value, know its order of execution time.

**Graphs.** Two major data structures for a graph: adjacency matrix and adjacency list. The advantages/disadvantages of each (in terms of the complexity of the primitive graph operations).

Topological sorting of a DAG.

Be able to code breadth-first and depth-first search and understand when each is useful.

**Dijkstra's shortest path algorithm.** While we will not ask you to write it, we may ask you questions about it —e.g. for a node  $w$  in each of the three sets of nodes, what does  $L[w]$  contain? For a connected graph of  $n$  nodes, what is the order of execution of the algorithm? What data structure do you use for one of the sets to achieve it? Which representation of the graph leads to a better execution time —adjacency matrix or adjacency list?

**Minimal spanning trees.** Be able to express Prim's algorithm and Kruskal's algorithm —how do they pick the next edge to add to the spanning tree?

**Proofs by induction.** Be able to prove theorems by induction. Be able to express a theorem in the form “for all  $n, \geq 0, P(n)$ . Relation between recursion and induction: Prove that a recursive function with a precise specification is correct by arguing that (1) the base case is correct, (2) recursive calls have smaller arguments (in some sense) than the parameters of the function, so that termination is ensured, and (3) If the recursive calls in the body perform as per the specification of the method (this is the inductive assumption), the body is correct.

**Understand key concepts that arise when proving the correctness of programs.** Know what we mean by specifications, preconditions, postconditions, and invariants (class invariant, loop invariant). Be familiar with the lectures that used these terms and be able to answer questions about them.

**Concurrency.** Be familiar with the issues seen when two or more threads of execution are running concurrently, perhaps both referring to and changing shared variables.

**Java implementation of threads.** Know how one creates a thread, starts it running, stops it. Be familiar with the use of synchronized, wait, and notify.

Be prepared to edit existing code using synchronized to make it thread safe.

Be familiar with the examples covered in class: producer/consumer and bounded buffer. Understand the code we saw in class and be prepared to answer questions about how that code works. Be prepared for questions about threads that communicate using bounded buffers.

**Topics from the last three lectures.** The last lectures are on advanced topics (in Spring 2014: 2 lectures on quantum computers and the last lecture). Sometimes we have no questions on these kinds of advanced topics, and sometimes they show up on true/false questions. You will not see hard questions on those topics.

**Sorting algorithms (written partially in English)**

```

/** Sort b[h..k]. */
public static insertionSort(int[] b, int h, int k) {
    // invariant: b[h..i-1] is sorted
    for (int i = h; i ≤ k; i = i+1) {
        Push b[i] down its sorted position in b[h..i]
    }
}

/** Sort b[h..k]. */
public static selectionSort(int[] b, int h, int k) {
    // invariant: b[h..i-1] is sorted, b[h..i-1] ≤ b[i..k]
    for (int i = h; i ≤ k; i = i+1) {
        Swap b[i] with smallest element in b[i..k]
    }
}

/** Sort b[h..k]. */
public static mergeSort(int[] b, int h, int k) {
    if b[h..k] has less than 2 elements, return;
    int e = (h+k)/2;
    mergeSort(b, h, e);
    mergeSort(b, e+1, k);
    Merge sorted segments b[h..e] and b[e+1..k]
}

/** Sort b[h..k]. */
public static quickSort(int[] b, int h, int k) {
    if b[h..k] has less than 2 elements, return;
    Partition b[h..k] based on its first value, x, say,
    and store a value in j so that:
        b[h..j-1] ≤ b[j] = x ≤ b[j+1..k]
    quickSort(b, h, j-1);
    quickSort(b, j+1, k);
}

/** Sort b. */
public static heapSort(int[] b) {
    Make b into a heap (with largest value at the root);
    // invariant: b[0..k] is a heap, b[k+1..] is sorted, and
    //           b[0..k] ≤ b[k+1..]
    for (int k = b.length - 1; k > 0; k = k-1) {
        Swap b[0] and b[k];
        Make b[0..k-1] back into a heap by
        bubbling b[0] down
    }
}

```

**Depth-first search:**

Node  $v$  is REACHABLE from node  $u$  if there is a path  $(u, \dots, v)$  in which all nodes of the path are unvisited.

```

/** Node u is unvisited. Visit all nodes
that are REACHABLE from u. */
public static void dfs(int u) {
    visit node u;
    for each edge (u, v) leaving u:
        if v is unvisited then dfs(v);
}

```

**Minimum spanning-tree:**

$ds$  is a data structure that contains nodes to be processed

```

while (some vertex is unmarked) {
    v = best vertex in ds;
    if (v is unmarked) {
        for (each w adjacent to v) {
            update w; add w to ds;
        }
    }
}

```

- (1) For breadth-first search algorithm:
  - $ds$  is a queue;
  - best element is one at front of queue
  - update  $w$ :  $D(w) = D(v) + 1$ ;
- (2) For Dijkstra's algorithm:
  - $ds$  is a priority queue
  - best element is one with highest priority
  - update  $w$ :  $D[w] = \min(D[w], D[v] + c(v,w))$
- (3) For Prim's algorithm:
  - $ds$  is a priority queue
  - best element is one with highest priority
  - update  $w$ :  $D[w] = \min(D[w], c(v,w))$

**Kruskal versus Prim:**

Below is Kruskal's algorithm, written in terms of adding edges to a set  $E$ , which ends up being a minimum spanning tree.

```

E = {};
while (E is not a spanning tree for the graph) {
    Add to E an edge with minimum edge weight
    that does not introduce a cycle into E;
}

```

To change it into Prim's algorithm, include the invariant that the graph  $G_1$  composed of edges  $E$  and the corresponding nodes is connected. Therefore, the body of the while-loop must choose an edge whose addition leaves  $G_1$  connected.