The CS2110 Final is optional. As soon as possible, we will post 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 and 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 Jenna Edwards, jls478@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 Jenna Edwards, jls478@cornell.edu, so that we know how many there are of you. Room: TBA.

Review: Sunday, 3 Dec, 12-2PM, Gates Auditorium

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

JavaHyperText entry study/work habits contains useful material. Everything that could be on P1 or P2 could be on the final. We spell this out in more detail below.

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 JavaHyperText on the course website. You may be asked about anything in these references, including access modifiers, abstract classes, and interfaces. In addition, know about inner classes, nested static classes, generics, enums, and JUnit tests. You will not be asked questions about reading/writing files. See the P1 and P2 review handouts for more details.

Java Collection Hierarchy. Basic understanding of how to use Collection<T>, List<T>, Set<T>, ArrayList<T>, HashSet<T>, and HashMap<T>. Know the basic methods in each, including their expected and worst-case runtimes: 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>.

GUIs: 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 of crucial 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.

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.

The *lower bound* for comparison sorts is NOT ON THE FINAL.

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 ½, at most 2 probes are expected in searching for a value. Chaining (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, etc. 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.

Know about heaps and heapsort.

Grammars and Parsing are not covered on the final.

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

Topological sorting of a DAG. Bipartite graphs and vertex coloring. Planar graphs.

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 d[w] contain? What is its loop invariant? The theorem proved from the loop invariant? 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 at a high level –not an implementation —how do they pick the next edge to add to the spanning tree?

Understand key concepts that arise when proving the correctness of programs. Know what we mean by specifications, preconditions, postconditions, the Hoare triple, and invariants (class invariant, loop invariant). Be familiar with the lectures that used these terms and be able to answer questions about them. If the pre- and post-conditions are given by array diagrams (as we did for selection sort, partition algorithm, etc.), be able to draw a diagram that generalizes the pre- and post-condition —as a possible invariant.

Be able to develop a loop given its precondition, post-condition, and loop invariant.

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

Java implementation of threads. Know how one creates a thread and starts it running. Be able to use keyword **synchronized** and the methods wait, notify, and notifyAll within a synchronized block. Understand the bounded-buffer problem as discussed in lecture (see the demo on the lecture-notes page).

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

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, **…**, 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; iterative DFS, BFS, Dijkstra…:

```
ds is a data structure that contains nodes to be processed
while (some vertex is unmarked) {
    v = best vertex in ds (remove it from ds);
    if (v is unmarked) {
        for (each w adjacent to v) {
            update w; add w to ds;
    }
}
```

(1) For breadth-first first 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 G1 composed of edges E and the corresponding nodes is connected. Therefore, the while-loop body must choose an edge whose addition leaves G1 connected.