This final exam is comprehensive, but there will be more emphasis on the last third of the semester. Only the topics listed below from the earlier material will be tested. You will again be permitted to bring in a "cheat-sheet" of notes, consisting of one double-sided sheet of paper with hand-written (or typed) notes. (You may not copy someone else's cheat sheet or print out lecture notes and use that as a cheat sheet.) You should turn in the cheat sheet with the exam, with your name on it.

For the listed algorithms, you should know the algorithm’s runtime, and you should be able to demonstrate that you know how it works, e.g. by showing what it would do on sample input.

Chapter 1: Algorithm analysis
- Asymptotic notation: \( \Omega, \Theta, O \)
- Runtime analysis: worst, best, and average-case analysis; upper/lower/tight bounds
- Writing the runtime of a non-recursive function
- Writing a recurrence relation for the runtime of a recursive function
- Writing the solution for a recurrence relation from the Master Theorem

Chapter 2: Elementary data structures
- Abstract data types: what is an ADT? How do you decide what ADT your program needs?
- Trees: terminology. Binary trees. Tree traversal: preorder, inorder, postorder
- ADT priority queue
- Heaps, heapsort
- Hash tables: hash functions, collision resolution: chaining, linear probing, double hashing; Dynamic hashing
- Selecting a data structure for an ADT based on the application specifics

Chapter 3: Balanced binary search trees
- (2,4) trees: definition, insert/find operations
- Red-Black trees: definitions, runtimes, concepts, when to use

Chapter 4: Sorting, sets, and selection
- Merge sort, Quicksort, Radix sort
- Lower bound on comparison sorting
- Parallel terminology. Parallel max. Odd-even merge and parallel mergesort.

Chapter 5: Design techniques
- Greedy: What is it? When does it apply?
- Greedy examples: making change, fractional knapsack, task scheduling
- Divide and conquer. Examples: Binary search, merge sort, min&max
- Dynamic programming. What is it, and when does it apply? Bottom-up and memoization.
- DP examples: Fibonacci, making change, 0/1 knapsack

Chapter 6. Graphs
- Depth-first search; applications: connectivity, reachability, DFS spanning tree
- Breadth-first search; applications: the same, plus shortest path (number of edges)
- Digraphs: Transitive closure (Floyd-Warshall algorithm), Topological sort

Chapter 7. Graph Algorithms
- Shortest path: Dijkstra’s algorithm, Bellman-Ford
- All-Pairs Shortest Path

Artificial Intelligence
- AI paradigms: neural nets, symbolic AI
- Setting up a problem for a symbolic AI solution
- Minimax search
- Backtracking, Greedy, and A* search, e.g. for shortest-path problems
- Admissible heuristics for A* search

Chapter 9. Text algorithms.
- Regular expressions and finite automata
Sample Questions

See the pertinent sample questions from the first two tests. These are linked on the web page for the class. Also review the first two tests and the homework problems. There will probably be a few questions on asymptotic notation and algorithm runtimes and several questions on data structures, algorithms, and algorithmic techniques we studied.

1. Show how Dijkstra’s algorithm would find the shortest path from the start vertex to all other vertices in this [given] graph. What is the worst-case runtime for a graph with n vertices, m edges?

2. Show how the all-pairs shortest path algorithm would work on the graph. What is its worst-case runtime? Can it handle negative-weight edges? Under what conditions?

3. What is the value of the root of the following graph, using a depth-2 minimax search?

4. Explain how to set up the 8-tiles problem for a solution by the techniques of symbolic AI. What would the states and transitions be in the search space?

5. How does A* search differ from Dijkstra’s algorithm? Explain what a heuristic is in this context, what it means that the heuristic is admissible. What would be a good, non-trivial heuristic for finding a solution to the 8-tiles problem?

6. Suppose you are writing a program that uses many insert, delete, find, and max operations. Of the data structures we studied, which would give the best average-case performance when using about an equal number of each operation? How would your answer change if we were to do many n insert, delete, and find operations and 17 max operations? How might your answer change if there were no delete or find operations?