Algorithms Spring 2003
Study Guide for Exam 2

Below is a list of topics to help you prepare for the second Algorithm's exam. In addition to this study guide, you should review your class notes and homework problems carefully to make sure that you understand all of the concepts.

The exam will mostly cover graph algorithms but may include some basic ideas from Abstract Data Types and linear programming. You may use an 8.5×11 sheet (both sides) that you have made yourself. No other materials (books, people, oracles, etc.) may be consulted during the exam.

1. Abstract Data Types (ADTs)
   (a) What are the notable properties of an ADT?
   (b) What is amortization? What is the difference between the accounting and the aggregate scheme?
   (c) Why is amortization such a powerful technique for analyzing an ADT's runtime? Why do we care about ADT's when building graph algorithms?
   (d) Review the ADTs we've described in class, e.g. Heaps, Trees, Balanced Trees, Fancy Queues, Dynamic Arrays, etc.
2. Fundamental Graph Algorithms
   (a) Representations of graphs as data structures: adjacency lists and adjacency matrices. What are they and what are the advantages and disadvantages of each?
   (b) Depth-first search. How does the algorithm work, what does it do, and exactly how do you analyze the running time? The running time is easy to remember (since it's just $O(V + E)$, the size of the adjacency list itself).
   (c) Breadth-first search. Same comments as for DFS above.
   (d) Topological sorting. Same comments as for DFS and BFS above. How did we prove the correctness of the algorithm (it relies on the proof that every directed acyclic graph has at least one vertex of in-degree 0).
3. Single-source Shortest Path Algorithms
   (a) The DAG Algorithm. How does it work? Make sure that you know how to prove that it is correct. Make sure that you can also derive the running time.
   (b) Dijkstra's Algorithm. How does it work? Be able to prove carefully and formally that it is correct (using the "Stronger Claim" technique from class). How does the implementation work using a heap? Make sure that you can derive the running time.
   (c) Bellman-Ford Algorithm. What can it do that Dijkstra can't? How does it work? How do we formally prove its correctness? How do we derive the running time? How can you use it to determine if a graph has a negative cost cycle? Why does this work?

4. All-pairs Shortest Path Algorithms
   (a) The Floyd-Warshall Dynamic Programming Algorithm. How does it work? Why does it work? How did we derive the running time?
   (b) Don Johnson's re-weighting Algorithm. How and why does it work?
5. Minimum Spanning Trees
   (a) What is the Minimum Spanning Tree problem?
   (b) What is the "General Theory of MST's"? What is a *cut* in this context? What does "respect" mean? What does the "Big Theorem" state? Make sure that you can prove it.
   (c) How does Prim's Algorithm work? Why is it correct? How does the implementation using a heap work? Make sure that you can derive the running time.
   (d) How does Kruskal's Algorithm work? Why is it correct? How did the Union-Find data structure work and how did we derive its running time? When is Kruskal's Algorithm preferable to Prim's?
6. Network Flow
   (a) What is the network flow problem? How is a flow function defined? What do the capacity constraint, skew symmetry, and conservation of flow mean? What is the *value* of a flow?
   (b) How are *cuts* defined in this context? Given the statements of the Cut Theorem, Capacity Theorem, and the Max Flow Min Cut Theorem, make sure that you can prove each one formally.
   (c) How does the Ford-Fulkerson Algorithm work? Using the above results, be able to prove formally that the Ford-Fulkerson Algorithm always finds a maximum flow. How did we derive the running time of the simplest implementation of the Ford-Fulkerson Algorithm?
   (d) How does the Edmonds-Karp work? What is its runtime? How is it different from Ford-Fulkerson?
   (e) Applications of network flow: How can we find the minimum cut in a network? Why does this work? How can it be used for matching in bipartite graphs (as in the PIT registrar problem)? How can it be used in baseball-prediction-type problems?