Algorithms
Section 2 Spring 2003
Study Guide for Final

Below is a list of topics to help you prepare for the final 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 comprehensive; expect questions like the ones you saw on Midterms 1 and 2 as well as questions concerning NP Completeness. You may use an 8.5×11 sheet (both sides) that you have made yourself (if you desire, this may be done in Latex). No other materials (books, people, oracles, etc.) may be consulted during the exam.

1. Fundamental Algorithmic Analysis Tools
   (a) Understand asymptotic notation. A good way to make sure you understand the differences between Ω, $\mathcal{O}$, and Θ is to derive why $\log n! \in \Theta(n \log n)$. Remember that for the lower bound constants, n_0 and c must be determined by hand. Why are lower bounds so valuable and what makes them so valuable? Why are upper bounds typically easier to derive?
   (b) Understand divide-and-conquer. Know how to set up these problems' recurrence relations and solve them using the recursion tree method. Understand how to deal with trees that aren't balanced and understand when a tree's height has no impact on run-time (e.g. recall the order statistics recursion when sorts are done on groups of 3 versus 5).
   (c) Understand sorting and order statistics – insertion sort, heapsort, quicksort, counting sort, radix sort. Know how to analyze the runtime and argue the correctness of a particular sorting paradigm (this usually requires that one can argue a particular invariant is maintained throughout an algorithm's execution, c.f. HW1b, Problem 4). Which algorithms rely on a partitioning technique and what is this (p.s. by he way, partitioning *can* be done in place, see page 146 of CLRS)? Why are heaps used so often (reason one: sorting, reason two: think abstract data types!). Understand how to derive a lower-bound for a comparison-based sorting algorithm (as well as other similar paradigms, e.g. HW 2b, Problem 4).
   (d) Understand dynamic programming. When can you use it to make a computation more efficient and when is it unable to help (recall the basic "components" required for dynamic programming to be useful, Section 15.3 in CLRS). Perhaps the most salient place to review this material is in the context of course scheduling (e.g. HW 5a, Problem 2, CLRS Section 16.1). This context is particularly illuminating because, while one *could* write a dynamic program (or a recursive algorithm) to solve this problem, greed provides much better performance here! As we did in class and on numerous HWs, know how to derive a worst-case (back of the envelope) running time for a recursive solution and be able to contrast this with a corresponding more efficient dynamic programming solution. Aside: what does the term "programming" refer to here? Table lookup!

(e) Understand greed. Because it is so easy to "greedify" something when greed is not optimal, you should *always* convince yourself that a greedy method is safe before using such a technique. Safety requires that one show that, given a solution is currently part of *some* optimal solution, the greedy choice transforms this solution into something that is also part of *some* optimal solution. So, having just stated this, why does safety give us a by-induction proof of correctness for free?

(f) Understand abstract data types (ADT), e.g. heaps, queues, (dynamic) arrays, linked lists, (balanced) binary search trees, the union-find data structure. Review the derivations for why B-trees and Red-Black trees are height balanced. Understand the difference between B-trees and Red-Black trees (each one cleans up the "unbalanced slop" a little differently). Understand how to use amortization to make an ADT look as good as possible (Kruskal and the Union-Find amortized analysis is an excellent case in point).

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? Also recall the incidence matrix used for deriving that $VC \leq_P SSUM$.

   (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 $\mathcal{O}(V + E)$, the size of the adjacency list itself). How can DFS be used to detect cycles? Can it find all cycles, or just the existance of at least one cycle (and just how many cycles can there be in a graph, anyway)?

   (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). How can you use this algorithm to determine if a graph is a DAG?

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 (FW) Dynamic Programming Algorithm. How does it work? Why does it work? How did we derive the running time? How can FW be used to detect the existance of negative cycles?

(b) Don Johnson's Reweighting Algorithm (hey, this stuff is as hot as *Miami Vice*). How and why does it work? When is this better to use than FW?

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? Why did we want to use it here?

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) 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?

7. NP-Completeness

   (a) What is an optimization problem? What is a decision problem? What are the precise definitions of P and NP?

   (b) What is a polynomial-time reduction?

   (c) What is the definition of "NP-complete"? What was the first known NP-complete problem and why is this such a big deal?

(d) If a problem is known to be NP-complete and a polynomial-time algorithm is found for that problem, what major consequence can you infer?

(e) Make sure that you understand all of the steps necessary to prove that a problem is NP-complete. Make sure that you completely understand the proofs of NP-completeness for the problems that we saw in class.