CISC320 algorithms, 10F, final exam review notes

Exam is Tue, Dec 14 10:30AM-12:30PM in our classroom.
Overall, the reading covered is chapters 1-6 and 8 and 9 of Skienna.
First half (Chapters 1-5, 6.2, 6.3) is summarized on the midterm review sheet. this sheet continues from there.
Chapter 6.1: Minimal spanning trees
Key points:
IF e is the minimal weight edge on a cut, then e is on some MST (minimal spanning tree).
Prim's algorithm: build spanning tree out from a start node. similar to Dijkstra's SSSP (single source shortest path) algorithm. Cut: separate fringe from tree, take shortest edge bridging this cut. Data structure: priority queue of fringe nodes.
Kruskal's algorithm: sort edges by weight, process by increasing weight. forest of disjoint trees is gradually combined into one tree. Cut: separate a tree from the rest. min weight edge bridges this cut. Data structure: union-find for dynamic disjoint sets (key actions: merge sets (using union). Determine if two nodes are in the same set (using find).
Union-find costs O(log(n)) time per operation if union by weight is used. Union-find costs O(log*(n)), essentially constant, time per operation if also path compression is used.
Chapter 6.5: Network flows and bipartite matching
Key points:
Max flow, min cut theorem: Max flow from node s to node t = min cut that separates s from t.
Ford-Fulkerson algorithm: inital flow = 0; find a path allowing positive flow, add it to the flow, compute residual graph until no more residual flow is possible. Trick: residual graph must allow some backward flow (you gain sometimes by undoing some flow you put in before).
Algorithm variants have to do with how to select flow paths to add.
Bipartite matching (section 6.5): reduce to network flow by adding two nodes s, t, with s connected to each node on one side of bipartite graph, t to the others. Make all weights 1. Max flow indicated the maximal matching.
Chapter 8, Dynamic Programming:
Strategy: Find a function of a set of parameters that

1. Has a clear conceptual definition
2. Has a recurrence relation expressing it's value at parameters in terms of other values at parameters in which at least one is decreased.
3. Solves the given problem for some set of parameter values.

Then figure out the table of values needed and the order of construction.
The run time cost is the size of the table times the cost to compute one entry. The latter is typically dominated by the number of other entries looked up to compute the given entry.

Memoization can do the table work for you. In the recursive function do: (1) first see if the parameters hash to a table entry already computed. If so, use it. Otherwise compute the value according to the recursive definition and put it in the table just before returning it.

Many problems yield good dynamic programming solutions.

- Longest increasing subsequence
- Edit distance (note relevance to bioinformatics)
- Traveling Salesman in O($n^2 2^n$)

- The hw4 exercises

Chapter 9: P and NP and NPC
Optimization problem vs decision problem:
Opt: What is the best (max or min) that property y of input x can be?
Decision: For input x, can property y of size k be achieved?
Example, Vertex cover: A vertex cover in a graph is a set of vertices such that all edges have at least one end in the cover.
Optimization problem: for given graph, find the minimum size of a vertex cover.
Decision problem: Does given graph have a vertex cover of size k? Decision problem vs optimization problem.
For simplicity, we discuss decision problems.
P: the class of problems that have an algorithm (A(x), for input x of size n) running in time O(n^k) for some constant k.
NP: the class of problems that have a nondeterministic algorithm (B(x,y), for input x of size n and hint y), running in time O(n^k) for some constant k and giving the answer "true" only if P(x) should be true and y is sufficient hint to guarantee that. Typically y is a solution to the corresponding optimization problem and B just verifies that.
Evidently, every problem in P is also in NP. Proof: If B is in P it has a polytime (deterministic) algorithm A(x). We have to show it has a non-deterministic algorithm. Let B(x,y) be A(x). In other words, a special case of a hint is no hint – or an ignored hint.
No one knows if there is any problem in NP that is not also in P.
NPC: The class of problems that are in NP and have the property that if any one of them is in P (has a polytime algorithm), then P = NP.
Cook't theorem: SAT is in NPC.
Reductions: Show problem L has an algorithm that works by calling an algorithm for problem M together with polynomially much additional work. If L is a hard problem, reduction to M is a way to show that B must also be hard.
Note, many reductions have this form: Poblem L(x) is reduced to problem M(z) by transforming input s of L into input z(x) for M in polynomial time. Then return M(z(x)). In showing that such a reduction works (that L(x) = M(z(x))), it is necessary to show two things, (1) that if L(x) is true then M(z(x)) is true, and (2) vice versa [either show M(z(x) implies L(x) or show that if L(x) is false then the same holds for M(z(x))].
Some reductions we have shown:
Bipartite matching to network flow (these problems are in P) [purpose: to get a good alg for bipartite matching]
SAT to 3-SAT (expand each k-clause to k-2 3-clauses) [purpose: to show 3-SAT is not easier than general SAT.]
3-SAT to Vertex cover (make widgets)
HAM cycle to HAM path (for each edge, remove it and ask about a HAM path)
Any problem in NP to SAT (Cook's theorem)
Easy reductions (where one problem is a subproblem of the other):
3-SAT to SAT (a 3-SAT input is a valid SAT input)
HAM to TSP (a graph with all edges of weight 1 is a weighted graph)

Principle: if (1) problem L is in NP and (2) SAT (or any other NPC problem) can be reduced to L, then L is in NPC.
Chapter 9 NP-complete problems
Approximation: It is easy to do to find a vertex cover for a graph that is no more than twice the size of a minimal cover (see section 9.10.1).
Examples of reductions about which exam questions could be asked

- Hamiltonian cycle to Hamiltonian path
- SAT to 3SAT
- 3SAT to Vertex Cover

Reduction categories (generalizations, uses of gadgets).

Cook's theorem: reduction of any NP problem to SAT.

Overall: Things to know about each algorithm:

- Why is it correct (vis a vis it's input/output specification)? What are the theorems and properties used to explain it's workings?
- What measure, n, of it's input is used as basis for analysis (for instance, n = bound on number of bits in numbers, or n = size of an array)?
- What formula (function of that n) estimates it's runtime? Usually the formula is a recurrence relation.
- What is the solution of that formula/recurrence? (often Master theorem helps here)

Kinds of questions: multiple choice, short answer, write algorithm, track algorithm.
Up to midterm: roughly 1/4 of final. After midterm: roughly 3/4 of final.