1. Solve the midterm. All of it! If you got all the points for a question, you can just copy-paste it. Otherwise scratch your answer and solve it again. Attach your answers to the rest of the HW. A fresh copy is given as a handout.

2. Exercise 15.4-1 in the text (page 396). Of course you could probably solve this problem easily enough just by looking at it. What I want you to do is to explicitly use the algorithm presented in class. Write out your derivation neatly, including the table.

3. Exercise 3.2 in the Lecture 8 handout (on page 12).

4. Exercise 3.3 in the Lecture 8 handout (on page 13).

5. Exercise 15.5-2 (page 404).

6. Let T be a rooted tree. T is not necessarily a binary tree. That is, each node may have any finite number of children. We are going to consider a method of visiting each node of the tree. The method works like this:

   At Step 1, we visit the root. The root is then marked as “visited”.

   At Step n, all nodes that have already been marked “visited” can visit at most one of their children. Of course if all the children of a node have already been visited, there is nothing for that node to do. But other nodes may still do something at that step. (And so in particular, more than one node might be visited in a single step of the process.)

   The question is this: what is the minimum number of steps necessary to visit every node in the tree?

   (a) Draw a picture of a tree in which more than one node is visited at some step of this process, and show why that happens.

   (b) Show that this problem contains optimal substructure. Please be careful when you do this. You have to state precisely and clearly what the substructure is and show why it is optimal.

   (c) Use that result to write a recursive algorithm to solve this problem.

   (d) Turn that algorithm into a dynamic programming algorithm.

   (e) What is the cost of the algorithm? (By that, I mean, “What is the cost of the algorithm that computes the minimum cost?”, not “What is the minimum cost?”). In doing this problem, please write as little pseudo-code as possible. I would much rather read something that is clearly explained using ordinary language.
7. (a) Read Section 10.1. This should be very easy, since I assume you already know about stacks and queues. The text has an implementation of a queue in terms of an array of fixed size, in which the operations Enqueue and Dequeue each have cost $O(1)$. (Remember that this is just a fancy way of saying that the cost is uniformly bounded by some constant.) You don't have to write anything - just read this and understand it. Let me know if you find anything confusing.

(b) The only problem with that implementation of queues is that the size of a queue is bounded by the size of the array. It would be nicer to have a queue which, like a stack, had no upper bound on its size. (Now of course in practice, a stack does have an upper bound on its size - that's why we talk about "stack overflow". But also in practice we have ways of dealing with this so that except in very unusual circumstances, stacks act as if their size was not constrained.)

So assuming that we had an implementation for stacks in which their size was not constrained, and in which the operations Push and Pop each have cost 1, I first want you to do Exercise 10.1-6 (page 236). The idea of this exercise is to provide an implementation of a queue in which the size of the queue is not constrained. In particular, you should derive the worst-case costs of Enqueue and Dequeue, and it should be clear that the cost of Dequeue is not $O(1)$. Presumably, that's the price we pay for having a queue of unlimited size.

(c) Then finally, do Exercise 17.3-6 (page 463). The idea of this exercise is show that in the implementation of the queue in Exercise 10.1-6 (which you just did), the amortized cost of both Enqueue and Dequeue for this data structure is $O(1)$. So things really are not so bad - in fact, they're pretty good.