We began with the usual class details. Two important ones are that class materials and information are available on the CSC 421 site found from http://facweb.cs.depaul.edu/jrogers and that the COL web site (http://col.cdm.depaul.edu) will be used only for assignment submissions and grading.

**Chapter 1: Role of algorithms in computing**

After echoing the book’s ideas that algorithms are technology and what an algorithm is (page 5) I mentioned and briefly explained the following two algorithms:

- **The Agrawal-Kayal-Saxena algorithm for determining whether a natural number is prime.** This algorithm was discovered only a few years ago. I did not explain how it works but did say that we have always had algorithms to determine whether a number is prime (one example being the Sieve of Eratosthenes) but none of these were efficient. I left aside for the time being what we mean formally when we say an algorithm is efficient. The AKS algorithm is the first primality testing algorithm that is efficient. You can find the original paper describing it here: http://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf.

  One must be careful to note that the algorithm efficiently determines primality but it cannot be used to factor. It does not threaten the security of public key cryptosystems like RSA.

- **The Shor algorithm for factoring…on a quantum computer.** There an efficient factoring algorithm but it works only a quantum computer, which exist as yet only in the abstract. Physical quantum computers capable of very short computations on very small inputs do exist but it is abundantly unclear whether these can be scaled up to machines able to work with large inputs.

I then presented two algorithms.

The first was **Prim’s minimal spanning tree** algorithm which, when given as input a weighted graph, returns a spanning tree of the graph whose total weight (that is, the total of the weights on the edges in the tree) is minimal among all possible spanning trees. We will see the pseudocode for this later. I showed informally how it works on an example graph.
The second was Dijkstra’s shortest path algorithm. Dijkstra’s algorithm determines the shortest path from one node to another in a weighted graph. I defined what that is and sketched how the algorithm solves the problem.

Both of these are examples of greedy algorithms and we will go into greater depth on each.

**Chapter 2: Getting started**

Using the insertion sort as an example, I demonstrated how to determine the correctness of an algorithm and how to analyze its running time. I wrote on the board (and slightly modified) the pseudocode for insertion sort, which appear on page 18 of the textbook. I did not demonstrate how it works but that can be seen in the book.

To determine correctness we often use **loop invariants**, a statement that, if it’s true before the loop executes, true each time the loop executes, and true when the loop terminates then the algorithm is doing what we say it is doing.

The three properties a loop invariant must have for the algorithm to be correct are:

- Initialization property: the statement is true before the first iteration of the loop;
- Maintenance property: if the statement is true before an iteration of the loop, it remains true before the next iteration;
- Termination property: when the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

For insertion sort the loop invariant is:

> At the start of each iteration of the **for** loop of lines 1-8, the sub-array A[1..j-1] consists of elements originally in A[1..j-1] but in sorted order.

I showed that the three properties hold for this invariant and, thus, insertion sort is correct (p. 19).

I then looked at determining how many steps the sort needs to sort an array of length n. I followed closely the book’s presentation of this. The conclusion is that the sort takes \( O(n^2) \) steps.

Insertion sort is an **incremental** algorithm. A better approach is **divide-and-conquer**. There are three phases to such algorithms: divide, conquer, and combine. Merge sort is one such algorithm:

- Divide: divide \( n \) elements in the array to be sorted into two sub-arrays of
length \( n/2 \);
- Conquer: sort the two sub-arrays recursively;
- Combine: merge the two sorted sub-arrays to produce a single sorted array.

I explained why the merge step (the combine step) performs \( n \) comparisons when merging together two sub-arrays of size \( n/2 \). Using that I put on the board a call tree showing what happens, in terms of recursive calls and calls to the merge method, when merge sort is applied to an array of length 8. From this I concluded that its running time is \( O(n \log n) \).

**Highlighted points**

- The study of algorithms reaches back 3,000 years and right up to now has yielded surprising and extremely useful techniques for problem-solving.

- There are three main approaches to designing algorithms: divide-and-conquer (which is a top-down approach), dynamic programming (which is a bottom-up approach), and greedy (which is local choice approach). Each of these will be discussed in the coming weeks.

- The two questions we want to answer about an algorithm: Is it correct? How efficient is it? We determine both using mathematical methods and proofs.

**Help with the assignment**

To help with exercise 2.2-2, where you are asked to write a loop invariant for and to analyze selection sort, consider this example with bubble sort.

1 Bubble-sort(a)
2   for i = 1 to n
3     for j = i+1 to n
5     end
6   end
7 end Bubble-sort

Loop invariant: At the start of each iteration of the for-loop of lines 2-6, the subarray \( A[n - i + 2..n] \) contains the largest \( i - 1 \) values in the array and they’re sorted.

Because the exercise only asks you to state the loop invariant, I will hold off until lecture next week on showing that the initialization, maintenance, and termination properties hold for bubble sort’s invariant.
We analyze the running time by looking at how often the statement at line 4 is executed. As you can see, that count is governed by the number of times the inner loop executes and that, in turn, is governed by the value of $i$, the index variable on the outer loop and a little algebra shows that the inner loop executes $n - i$ times for each value of $i$. So, overall, statement 4 is executed $n - 1$ times for $i = 1$, $n - 2$ times for $i = 2$, and so forth down to 0 times for $i = n$.

So we need to sum up $n - 1, n - 2, n - 3, \ldots, 0$. Writing this using the summation notation from discrete math, we find that we need to find the closed form of the sum:

$$\sum_{i=1}^{n} n - i$$

We can break that up into the difference of two summations:

$$\sum_{i=1}^{n} n - \sum_{i=1}^{n} i$$

Finding closed forms for these we get:

$$n^2 - \frac{n(n + 1)}{2}$$

Multiplying out and combining like terms, we end up with:

$$n^2 - \frac{1}{2}n^2 - \frac{1}{2}n = \frac{1}{2}n^2 - \frac{1}{2}n$$

And we're done.