Exam II next Tuesday.
Same format, about 16-18 questions, about 1/3 short answer, 1/3 code reading, 1/3 code writing.
You can have one 2-sided 8.5x11" "cheat sheet" hand-written or printed, no smaller than 10-point.

I suggest you try and answer as best as you can all the Review Questions. Pretend you are taking the exam. Then look at the Key. And make sure you understand anything you got wrong. If you have any questions on this or anything else, ask at the review session, or send the TAs or me email.

Review Session: 4:30 Fri, in C206.

Algorithms:

An algorithm is an explicit (mechanical) description of how to do some process.

Note however that what is "explicit" may depend on the who or what is executing the algorithm. So I can say, "Please go to the store and buy bread," and that will be clear to most people, but not to most computers or robots. We are interested in algorithms that can be executed by computers, and so the explicit operations are those built into the computer or already provided for us in libraries, etc.

Basic elements of an algorithm (LCR 306):

1) Algorithms are step by step recipes that clearly identify inputs and outputs (things provided and results).

You can describe algorithms in many different ways: English or any other natural

language (may be vague), math, diagrams (e.g., flowcharts), pseudocode (mixture of English and PL notation), etc. The key is that however you describe the algorithm, it be sufficiently precise. This means it should be mechanical, and involve no judgment, skill, or detailed knowledge. So a very dull person or machine could carry it out. In general, we don't like doing mechanical processes (they are boring), and we tend to make mistakes, but machines are good at it. So you have to design the algorithm so that works correctly in the absence of judgment, skill, and detailed knowledge. "Computers are dumb."

To run an algorithm on a computer, you have to translate it into a PL, then you have a program (as opposed to an algorithm).

2) Algorithms name the entities that are manipulated or used, e.g., variables, functions, objects, and classes.

3) Generally, the steps in an algorithm are executed in the order in which they're written. (Exception: Invoking a function jumps to another part of the algorithm, but then returns to do the rest.)

4) Some of the steps specify decisions (if-then, switch, etc.), which affect which steps we do.

5) Some of the steps specify repetition of steps (loops), e.g., for-loops, while-loops, recursion.

6) All of the above can be combined in any way, subject to the syntax of the language. This is the combinatorial power or compositional power of a PL.

Because of these common features a program in one PL can almost always be

translated into another PL. In this sense all PLs are equally powerful (in a mathematical sense). However, it may be much easier to program it in some PLs than in others, or it may be more efficient, or more reliable to program, or easier to read; also some PLs encourage or discourage good programming style; some are better for big programs or little ones; some are better for beginners or experts, etc.

As I've mentioned, there are thousands of PLs. You cannot learn them all. And there are new ones being invented every year. A few may be come popular in the future. (Example: F#)

The bottom line is, unless your programming of computers is going to be very casual, *you will have to learn new PLs*. However, the basics of algorithm construction are pretty much the same in all of them. When you want or need to learn a new language, look for the familiar stuff, and build on that to learn the new stuff. This is part of the skill of being a computer scientist.

Developing an Algorithm:

0) See if you can modify/translate an existing algorithm to do what you want. If not, then...

1) Think about how you would do the process by hand. What are the steps? What are the decisions you have to make? What scraps paper do you need (to make notes on, etc.). When you get an idea, think of describing it to a very dull person (i.e., make it explicit at whatever level is necessary — what "skills" are available).

Remember: Pick the right data structure and the algorithm will design itself.

Transfer your physical intuitions about how to do something to the computer. Work

out some examples by hand.

2) Look for common patterns. If you find yourself doing the same thing or very similar things over and over, that may be a loop or a function.

3) Think carefully about general and special cases.

4) Write it out carefully in English, pseudocode, a diagram, etc. Look for ambiguities, incompleteness (cases not covered) etc.

5) Code reading / code review. Have someone else read it. Explain it to them.

6) Testing.

7) Start with simple example that you can check by hand.

8) Make sure to test boundary conditions. Your knowledge of the algorithm can help you decide what these are. You wrote it, you know how to break it. Try to exercise every path in the program. Give it bad data.