Final Exam: Monday, May 9, 12:30-2:30 here.
Review session on May 2 (TAs will announce).
Review Questions coming soon.

Algorithms and Programs:

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

There is some vagueness in the idea of whether something is explicit or mechanical. In deciding what's explicit, we have to consider who or what is going executing the algorithm. What's explicit for one "execution vehicle" might not be for another.

"Mechanical" means it could be done in principle by a machine. Or it can be done by a person "mechanically" (without the use of judgment, understanding, or specialized knowledge).

These are the sorts of things that are boring to people, and when people get bored they start to make mistakes. That's why we make machines to do them.

More specifically, algorithms have the following features (LCR, p. 306):

(1) Algorithms are step-by-step procedures with clearly defined inputs and outputs.

Algorithms can be expressed in many different ways, such as writing in a natural language, such as English, using flowcharts or diagrams, expressing them in mathematics, writing them in pseudocode (a mixture of English and programming language-like notations), or in a programming language (such as C++).

If you express an algorithm in a programming language, you have a program. Then

it can be executed on a computer.

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

(3) Generally the steps of an algorithm are executed in the order they're written, from beginning to end.

(4) Some of the steps may specify decisions, which determine which steps will be executed (e.g., if-statements, switches).

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

(6) All of the above can be combined in any way, subject to the syntax of the language (combinatorial power).

Virtually all PLs allow you to do all of these things (have all of these characteristics). So this means that in a fundamental sense all PLs are equal in power: any algorithm you can express in one, can also be expressed in another. PLs differ in the ease of expression, in the reliability of programming in them, in their efficiency, and other factors. So if you are going to do much programming, you will have to learn more programming languages, but they all have the same basic characteristics.

Developing an Algorithm:

(0) See if you can use or modify an existing algorithm (don't re-invent the wheel — unless you are doing it for practice). If, not, proceed as follows:

(1) Think about how you would do it by hand. Put yourself in the place of the computer. What are the steps I would have to take? What are the decisions? What sorts of scrap paper do I need? How would I organize the information? (This helps you decide on data structures. "If you pick the right data structure, the algorithm will design itself.").

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

(3) Think carefully about general and special cases.

(4) Write the algorithm in English, pseudocode, a diagram, or something similar. Now you are trying to make it explicit (mechanical). Look for ambiguity, incompleteness (cases not covered), etc.

(5) Code-reading or code-review: Have someone else look at your algorithm and see if it makes sense. Explain it, step by step, to someone else, who should try to understand it.

(6) Testing. Start with simple examples that you do by hand, or for which you already know the correct answer.

(7) Make sure to test boundary cases. These are cases that are peculiar in some way. For example, they cause some for-loop to execute 0 times. Your knowledge of the algorithm can often help you to identify boundary cases. You wrote it, you know how to break it. Give it bad data.

(8) Try to make sure that you test cases exercise every path through the program.

Linked Lists:

Choice of data structures: different data structures are better/worse for different purposes.

For example, vectors and linked lists have different characteristics:

Vectors (and arrays):

* Efficient "random" access: equally efficient to get to any element of the vector. (Because the vector uses contiguous storage)

* Inefficient insertion or deletion. (Because the vector uses contiguous storage)

Linked List:

* Inefficient sequential access: I have to start at the beginning and count my way down to the element I want. (Because the elements are stored non-contiguously)

* Efficient insertion and deletion: just have to change a few pointers. (Because the elements are stored non-contiguously)

Which is better? It depends on what you think you are going to be doing.

But remember, what you are doing could change as the program evolves, which is why the Information Hiding Principles are important.