COMP 200: Elements of Computer Science
Fall 2004
Lecture 3: August 26, 2004

On the Board

Reading: Start Chapter 2
Homework 1 due Monday

Expressing Algorithms (or Algorithmic Problems)

The book makes a distinction between an *algorithmic problem* (or a *computational problem*) and the expression of an *algorithm* that solves the problem. The problem is abstract, as in "*sort yourselves into ascending order by first name,*" while the algorithm is concrete — a clear definition of how to sort the set of people attending lecture on Wednesday. (The difference is akin to the distinction between a patent and a copyright. Patents cover novel inventions — ideas — while copyright protects the expression of an idea — its written or spoken form.)

An algorithm must specify, in unambiguous form, a set of detailed steps that can be followed to solve an instance of the computational problem. (Whoa — an *instance*? Sorting by first names is a computational problem. Sorting the set of people who attended class on Wednesday is an instance of a sorting problem.) Your homework requires you to write an algorithm for sorting the class into ascending order by first names, so that example is out of bounds for today's lecture. Let's pick another one — our binary search of the phonebook.

The algorithm is conceptually simple:

1. Consider the active range of the search to be the entire phonebook
2. Find the middle of the range
3. Compare the string being sought against the middle entry
    a. If the target string is less than the middle entry, narrow the active range to the first half of the current active range
    b. Otherwise, narrow the active range to the second half
4. Repeat until the active range contains one element
5. If that element matches the target string, report success; otherwise report failure.

Note that we introduced an abstraction — the active range of the phonebook. This abstraction makes it easier to talk about the algorithm. It also works well for the simplification that limits the search to a single letter in the book.

Unfortunately, this description is not sufficiently detailed for a computer to execute it. Actions such as "find the middle of the range" make sense to an educated human being, but are far more abstract than a computer can understand. (A good working model is that a computer can do basic arithmetic, compare two quantities, and choose among alternatives.) How would we tell the computer to "find the middle of the range"?

Assume that the phonebook contains *k* names and that we number them from 1 to *k*. (Draw 11 item range on the board.)

We might compute the middle element's index as the average of the bounds:

$$(1+11)/2 = 6$$

That works because we have an odd number of elements. If the range is 1 to 10, the formula produces 5.5, which is not a counting number. If we round up to 6, the formula works. (Rounding upward to the next counting number — or integer — is sometimes called taking the ceiling of a number.)

This formula suggests that we can represent a "range" as a pair of numbers, the index of the lowest element in the range and the index of the highest element in the range. Our algorithm, then, needs to set the initial bounds for a range, and successively reduce it until it has one element. Our search might look like

```
set range as [low, high]
while(low < high)
{
  middle ← ceiling((low+high)/2)
  if (entry[middle] ≤ target)
     then low ← middle
     else high ← middle
}
```

This description matches, sort of, our verbal description of the algorithm. It compares the middle entry against the target string and shrinks the range [low,high] accordingly. (Geeks call this notation "pseudocode".)

Does this work? Let's try it.

Non-convergence because low can never equal high. The rounding implied by "ceiling" ensures that a two-element range remains two elements.

⇒ *Change the termination condition on the loop and test one of the two elements.*

```
set range as [low, high]
while(low + 1 < high)
{
  middle ← ceiling((low+high)/2)
  if (entry[middle] ≤ target)
     then low ← middle
     else high ← middle
}
```

At this point, the range will be two elements. We need to test one or both of them.

The easy thing to do is test *entry*[low] and *entry*[high] and return success or failure accordingly.

Thus, the full algorithm is:

```
low ← 1
high ← number of entries in the phonebook
while(low + 1 < high)
{
  middle ← ceiling((low+high)/2)
  if (entry[middle] ≤ target)
     then low ← middle
     else high ← middle
}
if (entry[low] = target OR entry[high] = target)
   then report success
   else report failure
```

There are, of course, alternate solutions.

⇒ where low ← middle, use low ← middle+1. Change convergence test to "middle = high". Always test the entry for down. Convergence from above yields a range of 2 elements & tests the lower one. Convergence from below yields a range of 1 element and tests it.

⇒ start out with high set to 1 more than the number of elements. It can never converge to the original upper bound (& it never compares against that upper bound), so this solution works correctly.

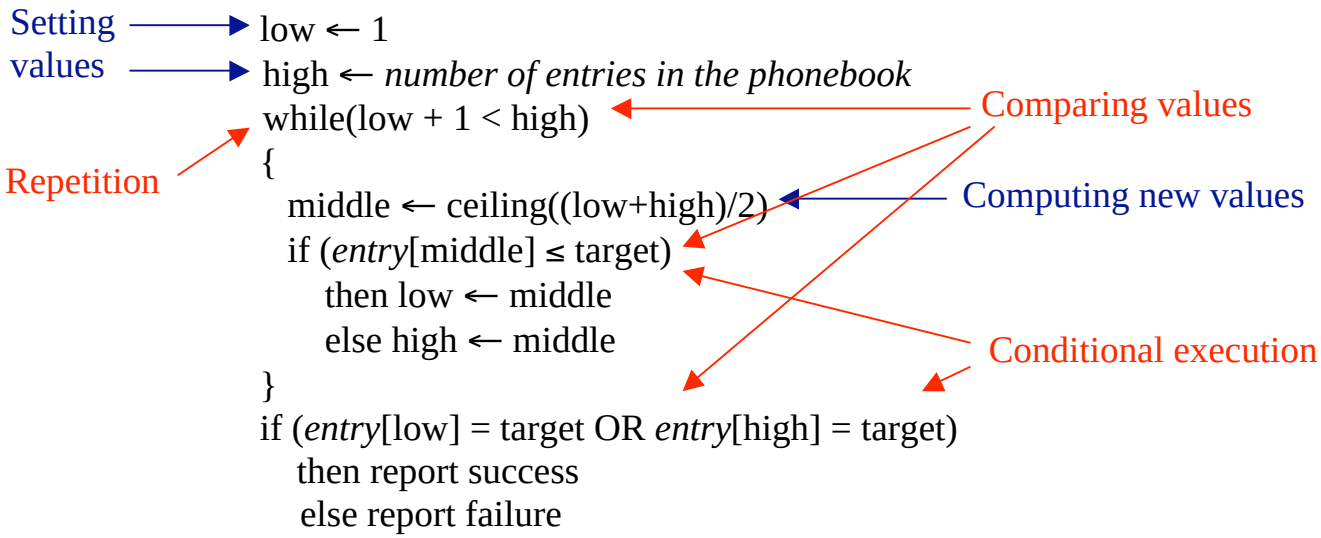
This is the level of detail needed in a program that will execute by computer. Of course, once we have written this code and packaged it as an operation

BinarySearch: <low,high,target,table> → success or failure

then other algorithms can use it as if it were an elementary operation. (Functional abstraction, again)

Programming Language Constructs

This algorithm is sufficiently detailed that it could be converted directly into a computer program and executed. [In fact, I did roughly that for a figure in our book.] Notice the kinds of actions that it uses:



Programming languages must provide constructs for all of these purposes, plus support for functional abstraction (in some form). In class, we will write programs in *pseudocode*, using notations similar to those in the book. In our homework, we will begin to write simple programs in Scheme, a programming language noted for its simple syntax and easily understood behavior.

In developing this short (12 lines) pseudocode program, we went through many mental gyrations. The process was chaotic and disorganized. In this class (and in COMP 210), we will introduce a simple, data-driven methodology for writing small programs that replaces chaos with process. The result is a systematic approach to writing programs.

Why Scheme?

In COMP 200, we will use Scheme for some or most of the programming assignments. Why not use Java — the hot language of the moment — or a more traditional language such as C or Visual Basic?

Scheme has a simple syntax that lets us cover all the syntax rules that you will need in about ten minutes. In contrast, the other languages have complex syntax that forces the programmer to learn (and remember) myriad rules. That complexity forces a focus on syntax, when students are better served by focusing on design, on function, and on debugging.

We will work, in the early stages, in a subset of Scheme that obeys the same basic rules as high-school algebra. This value-based semantics creates a direct connection between the meaning of the programs that you write and the mathematics that you learned in earlier parts of your education. This reinforcement should make some aspects of programming follow your intuitions. In practice, this creates a comfort level with programming that helps students focus on design rather than on trying to puzzle out the meaning of cryptic codes.

Chapter 1 Review (we're done with it)

1. Algorithms lie at the heart of Computer Science. Algorithmics is the study of algorithms and their applications.
2. In describing algorithms, we need abstractions for individuals and for values. We need mechanisms for expressing a sequence of actions, for comparing values and using the result to choose between sequences, and for repeatedly executing the same sequence of actions (with some well-specified termination condition).
3. Algorithms intended for execution on a computer must be specified at a level of detail that is sufficient to allow the computer (which only has fairly low-level operations) to execute it.
4. No obvious relationship exists between a program's textual length and its running time. (In fact, one of the properties of a program that we cannot always analyze is its running time. There exist classes of programs where we cannot even tell if they will halt.) The following program is an example of a short program with a long running time.

```
i ← 1
while (i < 1000000000)
{
  i ← i +1
}
```