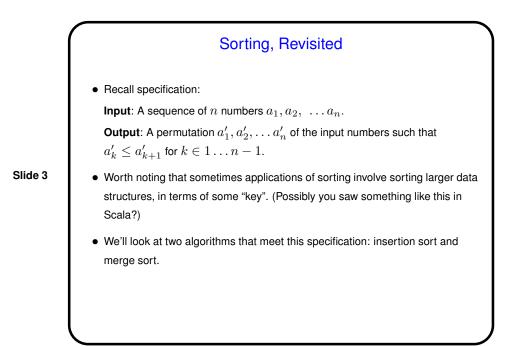


A Few Words About Pseudocode

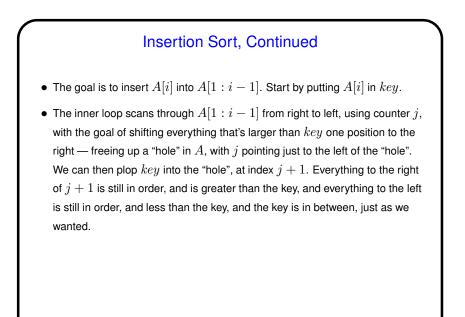
• Textbook presents algorithms in *pseudocode* — notation that's meant to be readable to anyone familiar with one of many popular programming languages, but not constrained by them. Emphasis is on "readable" — idea is to use whatever makes the ideas clearest, and at times that may be natural language.

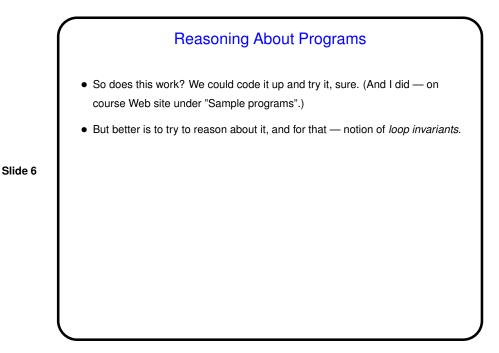
Slide 2

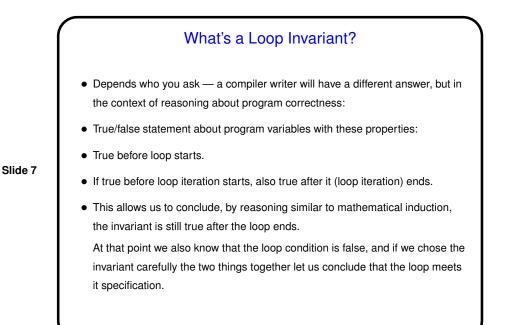
• More details in Chapter 2 (2.1). Probably most noticeable thing for someone coming from a language based on C is the use of indices that start with 1 rather than 0.



ſ	Insertion Sort
	• Algorithm to sort array $A$ of $n$ elements, In pseudocode:
	Insertion-Sort $(A,n)$
	for $i=2$ to $n$
	key = A[i]
	// Insert $A[i]$ into sorted subarray $A[1:i-1]$
	j = i - 1
	while $j>0$ and $A[j]>key$
	A[j+1] = A[j]
	j=j-1
	A[j+1] = key
	• Informally, we move left to right through the array, keeping a "hand" of sorted elements and inserting elements of the original array into it, one at a time.
	• This probably seems fairly straightforward, except for the inner loop maybe. next slide







# Loop Invariants for Insertion Sort

- Comment at start of loop is meant to suggest the intended invariant: A[1:i-1] is a permutation of the original values of A[1:i-1], in sorted order.
- Slide 8
- True before start? Trivially, yes (i = 2).
- If A[1:i-1] true before loop iteration, does the loop "insert" A[i] into the sorted "hand"?

Yes. (Note that the inner loop does rely on A[1:i-1] being in order.)

- If true at end, i = n + 1, so yes, this implies that the whole array (A[i:n]) is sorted.
- (As the textbook points our, really that inner loop needs its own invariant, but coming up with one is a little tedious, and perhaps a needless formalism .)

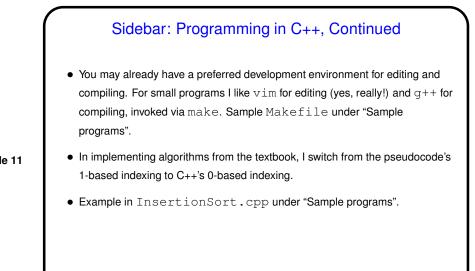
## Loop Termination

- Note that we also need to prove, or at least reason about why, the loops terminate. But that seems straightforward:
- The outer loop executes n-1 times (*i* starts with 2, increases by 1 every iteration, and ends when i = n + 1). The inner loop executes at most i 1 times (*j* starts with i 1 and ends when j = 0, if not before.)

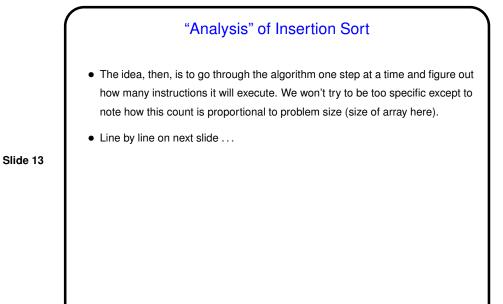
Slide 9

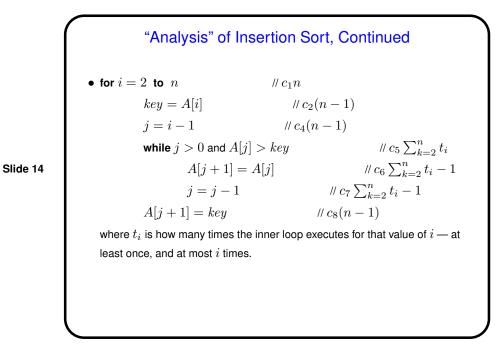
Slide 10

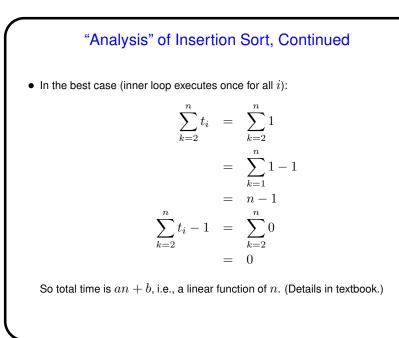
# Sidebar: Programming in C++ The department likes to use C++ for this course, so you get two semesters' worth of programming in another language (in addition to Scala). So you should have learned (some!) C++ in CSCI 2320 (Data Abstraction). C++ has been through many versions, with each one making a claim to being a better language. We will use version C++11, and I will try to make reasonable use of newish features. Note that for best results on our department's older builds (everything except the classroom machines, at present), you will need module load gcc-latest before compiling. You can avoid typing this every time you log in by putting it in your file ~/.bash\_profile.

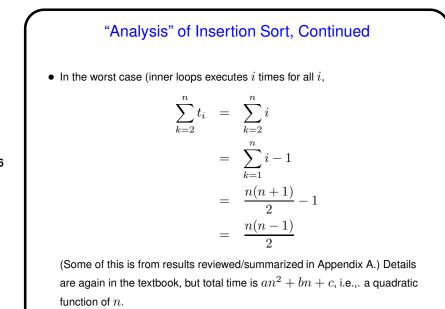


"Analysis" of Algorithms • Another thing it's useful to know is an estimate of resource usage - both memory and execution time - and how they relate to program size (count of elements being sorted, here). • To do this, we use the "random-access-machine (RAM)" model, in which there is one processor, and every instruction takes the same amount of time. The Slide 12 model doesn't go much into specifics of what an instruction is, except to require that it be realistic about what a single instruction can do. (So for example it's unlikely that the processor would provide one to sort a whole list or array.) Worth noting too that this model ignores subtleties of memory use, which can make a difference in performance in real hardware.









Slide 16

