Slide 1

Administrivia

- These presentations will be linked from the schedule page. Usually I'll post a preliminary version before class, a more-final version later (sometimes there are typo fixes, e.g.).
- I've also made a Google Doc with links to Zoom recordings and shared it with all of you. There's a link to it at the bottom of the schedule page too.
- Also, you may notice in your Google Drive "shared with me" a folder with name "CSCI-3322-*your_name*"? This is the shared-only-by-you-and-me folder where I plan to put information about grades and where I want you to put things you're turning in.
- I'm hoping to get the first reading quiz (over chapter 1 and 2) posted over the long weekend.

Slide 2

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.
- 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.

Slide 3

Sorting, Revisited

- Recall specification:

  Input: A sequence of n numbers $a_1, a_2, \ \ldots a_n$.

  Output: A permutation $a'_1, a'_2, \ldots a'_n$ of the input numbers such that $a'_k \leq a'_{k+1}$ for $k \in 1 \ldots n-1$.
- Worth noting that sometimes applications of sorting involve sorting larger data structures, in terms of some "key". (Possibly you saw something like this in Scala?)
- We'll look at two algorithms that meet this specification: insertion sort and merge sort.

Slide 4

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 ...

Slide 5

Insertion Sort, Continued

- The goal is to insert $A[i]$ into $A[1:i-1]$. Start by putting $A[i]$ in key.
- The inner loop scans through $A[1:i-1]$ from right to left, using counter j, with the goal of shifting everything that's larger than key one position to the right — freeing up a "hole" in A, with j pointing just to the left of the "hole". We can then plop key into the "hole", at index $j+1$. Everything to the right of $j+1$ is still in order, and is greater than the key, and everything to the left is still in order, and less than the key, and the key is in between, just as we wanted.

Slide 6

Reasoning About Programs

- So does this work? We could code it up and try it, sure. (And I did — on course Web site under "Sample programs".)
- But better is to try to reason about it, and for that — notion of *loop invariants*.

Slide 7

What's a Loop Invariant?

- Depends who you ask — a compiler writer will have a different answer, but in the context of reasoning about program correctness:
- True/false statement about program variables with these properties:
- True before loop starts.
- If true before loop iteration starts, also true after it (loop iteration) ends.
- This allows us to conclude, by reasoning similar to mathematical induction, the invariant is still true after the loop ends.

  At that point we also know that the loop condition is false, and if we chose the invariant carefully the two things together let us conclude that the loop meets it specification.

Slide 8

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.
- 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 .)

Slide 9

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 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`.

Slide 11

Sidebar: Programming in C++, Continued

- You may already have a preferred development environment for editing and compiling. For small programs I like `vim` for editing (yes, really!) and `g++` for compiling, invoked via `make`. Sample `Makefile` under "Sample programs".
- In implementing algorithms from the textbook, I switch from the pseudocode's 1-based indexing to C++'s 0-based indexing.
- Example in `InsertionSort.cpp` under "Sample programs".

Slide 12

"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 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.

"Analysis" of Insertion Sort

Slide 13

- The idea, then, is to go through the algorithm one step at a time and figure out how many instructions it will execute. We won't try to be too specific except to note how this count is proportional to problem size (size of array here).
- Line by line on next slide . . .

"Analysis" of Insertion Sort, Continued

Slide 14

- **for** $i = 2$ **to** n // $c_1 n$

  $key = A[i]$ // $c_2(n-1)$

  $j = i - 1$ // $c_4(n-1)$

  while $j > 0$ and $A[j] > key$ // $c_5 \sum_{k=2}^{n} t_i$

  $A[j+1] = A[j]$ // $c_6 \sum_{k=2}^{n} t_i - 1$

  $j = j - 1$ // $c_7 \sum_{k=2}^{n} t_i - 1$

  $A[j+1] = key$ // $c_8(n-1)$

  where t_i is how many times the inner loop executes for that value of i — at least once, and at most i times.

Slide 15

“Analysis” of Insertion Sort, Continued

- In the best case (inner loop executes once for all i):

$$
\begin{aligned}
\sum_{k=2}^{n} t_i &= \sum_{k=2}^{n} 1 \\
&= \sum_{k=1}^{n} 1 - 1 \\
&= n - 1 \\
\sum_{k=2}^{n} t_i - 1 &= \sum_{k=2}^{n} 0 \\
&= 0
\end{aligned}
$$

So total time is $an + b$, i.e., a linear function of n. (Details in textbook.)

Slide 16

“Analysis” of Insertion Sort, Continued

- In the worst case (inner loops executes i times for all i,

$$
\begin{aligned}
\sum_{k=2}^{n} t_i &= \sum_{k=2}^{n} i \\
&= \sum_{k=1}^{n} i - 1 \\
&= \frac{n(n+1)}{2} - 1 \\
&= \frac{n(n-1)}{2}
\end{aligned}
$$

(Some of this is from results reviewed/summarized in Appendix A.) Details are again in the textbook, but total time is $an^2 + bn + c$, i.e.,. a quadratic function of n.

Slide 17

Analysis of Algorithms, Continued

- In the analysis of insertion sort we considered both the best case and the worst case. Usually we focus on worst-case behavior. Why?
- It's usually useful to know, as an upper bound on actual behavior.
- The textbook claims that worst-case behavior occurs surprisingly often, and approximates average-case behavior (which would seem to be more meaningful, and which sometimes we *are* interested in).

Slide 18

Order of Growth

- In the example just done, we end up with very simplified functions that summarize/hide a lot of the details.
- But in practice this is what we're interested in — and indeed, what we really care about is how fast execution time increases as problem size grows — i.e., the "order of growth", and for that, for a polynomial-time function, all that really matters is the highest-order term (n, n^2, etc.). If you try plotting functions with different-order terms, for varying ranges of x, it's not hard to see that for large enough x, the function with the highest-order term is larger, independent of lower-order terms and multiplicative constants.
- For this we use the Greek capital Θ, e.g., $\Theta(n)$, or $\Theta(n^2)$.
- (We will look at this more formally in Chapter 3.)

Another Example: Merge Sort

- Another algorithm for solving the same problem — "'merge sort" — next time.

Slide 19

Minute Essay

- Who was your instructor for CSCI 2320? What platform did you write code on/for, and what tools did you use?
- Any questions about today's material?

Slide 20