Slide 1

Administrivia

- Reminder: Quiz 5 Thursday. Likely topics are arrays and using files.
- Reminder: Homework 6 Thursday.
- How many will be here next Tuesday?

Slide 2

Sorting and Searching — Overview

- Something we often want to do is put things in order — similar to "alphabetizing" a list of names. Techspeak for this is *sorting*, and it can be done to anything for which you can define an ordering.
- A related problem is *searching* ("does this array contain a specified element?"). One motivation for sorting is that it makes searching much faster. (Why? Well — how would you search for something in a list, if the items are in no particular order? How does it help to know that they *are* in order? Think about searching for a particular word in a dictionary.)
- So, if you have a list of things, how would you put them in order?

Slide 3

Sorting

- Many ways to put a list of things in order. Some are simple to understand and to code, but slow. Others are somewhat more complicated, but faster. (What do we mean by “slow”? More about that later.)
- Simple-but-slow methods:
  - Bubble sort: Repeatedly go through the list exchanging adjacent elements that are out of order.
  - Selection sort: Find the largest (or smallest) element and put it at the appropriate end. Repeat with the next largest (smallest) element, putting it next to the end, and so forth.
  - Insertion sort: Start with one element, and “insert” subsequent elements into a sorted-list-so-far.

  All of these have running time proportional to N^2, where N is the number of things to sort. (Better algorithms have time proportional to $N \log N$.)

Slide 4

Comparing Algorithms

- We’re talking here about different ways of solving the same problem (putting a list of things in order) — different *algorithms*. Which is “better”, or is there any way to compare?
- One comparison is simplicity / readability — the simpler the algorithm, the more likely it is you can turn it into code and get it right.
- Another, though, is resource use — memory use, running time. Actually measuring these depends on a lot of factors, hardware and software. Is there some way to estimate, *before* writing the code and trying it?

Slide 5

Order of Magnitude of Algorithms

- An estimate — “order of magnitude”, a.k.a. “big-oh notation”. Similar to order of magnitude of numbers — crude estimate, but good enough to be useful in many situations.
- Idea is to estimate how work (execution time) for algorithm varies as a function of “problem size” (e.g., for sorting, size of array). (Similar idea can be applied to how much memory is required.)
- Usually do this by counting something that represents most of the “work” in the algorithm and varies with problem size (e.g., for sorting, how many comparisons).

Slide 6

Order of Magnitude of Algorithms, Continued

- Informally, $O(N)$ means work/time is proportional to N (problem size). $O(N^2)$ means . . . ?
  (Compare aN and bN^2 as N increases, for different values of a and b. bN^2 larger for larger enough N.)
- Formal definition (from CSCI 1323): $g(n)$ is $O(f(n))$ if there are positive constants n_0 and c such that for $n \geq n_0$,

$$g(n) \leq cf(n)$$

Slide 7

Order of Magnitude for Sorting Algorithms

- For sorting algorithms, we usually count the number of times we compare two elements.
- For selection sort: Finding the largest element (of N) requires how many comparisons? Finding the next largest takes how many? and so forth ...
- For bubble sort: The first pass through the data involves how many comparisons? The next pass? and so forth ...

Slide 8

Searching

- Another thing we often want to do is find out whether a given element is in an array.
- Obvious way is *sequential search* — start at one end, look at each element until you either find what you want or get to the end.
- Less obvious way is *binary search* and works only if array is sorted — compare the thing you're looking for (`search_elem`) to the element in the middle of the array (`a[mid]`). Three cases:
  - `search_elem == a[mid]`. Done!
  - `search_elem < a[mid]`. Search elements to the left (recursively).
  - `search_elem > a[mid]`. Search elements to the right (recursively).

Slide 9

Order of Magnitude for Searching Algorithms

- For searching algorithms, we also usually count the number of times we compare two elements.
- For sequential search: Looking for something in a list of N elements requires how many comparisons? best case? worst case?
- For binary search: Looking for something in a list of N elements requires how many comparisons? best case? worst case?

Slide 10

Sorting and Searching — Code

- Before we start writing code, think a minute about how to test it. Certainly we can get input from a human user. But if we just want to know if the sorting function works, we might have the program generate its own data, and check its own results. This would also let us easily observe how running time (or something related) increases as a function of number of elements.
- How to generate data? We could use `rand` to generate “random” data. (More on next slide.)
- How to check results?

Slide 11

Pseudo-Random Number Generators

- The last slide put quotes around "random". Why? Partly because exactly what random means is difficult to pin down, partly because the best we can do in deterministic code is a good fake — *pseudo-random*.
- Mathematically interesting topic, classic reference is in one of the volumes of Donald Knuth's *The Art of Computer Programming*.
- Also used more than one might think in programming! for example, in simulating various things in the physical world. Textbook examples often involve simulating rolling dice, shuffling cards, etc.

Slide 12

Minute Essay

- We said that for selection sort and bubble sort the (worst case) number of comparisons for N elements is

$$(N-1)+(N-2)+\ldots+1$$

Try to come up with a similar formula for insertion sort (where you first insert the second element into a list consisting of the first element, then insert the third element into a list containing the first two, etc.).

Minute Essay Answer

- The analogous formula for insertion sort is

$$1 + 2 + \ldots + (N - 1)$$

- Since this is the same sum as the other two, but in reverse order, overall number of comparisons is the same — $N(N-1)/2$, i.e., $O(N^2)$.