Administrivia

Slide 1

- Homework 7 due date extended to Friday.
- Quiz 4 moved to next Friday.

Sorting and Searching

Slide 2

- Traditional topics in CS1 courses. Arguably not of first importance to people more interested in using computers as tools, but still interesting . . . :
- Both are good examples of problems that can be solved in different ways.
- Both are good examples for introducing the idea of "order of magnitude" for algorithms.
- (But if you actually need to do one of these operations, look first for a library function!)

Slide 3

Sorting — The Problem and Some Solutions

- Problem: Given an array (or list) of elements for which there is a sensible "less than" operator, put them in order.
- Simple solutions include bubble sort, selection sort, insertion sort. Easy to program but not "fast" (more shortly).

  Textbook has good discussions.

  (Examples of doing bubble sort and selection sort.)
- More-complex but "faster" solutions exist, and two of the best-known use recursion(!). More about them later.

Slide 4

Searching — The Problem and Some Solutions

- Problem: Given an array (or list) and an element, search the array for the element.
- Simplest solution is sequential search. Easy to program and works for any array but not "fast".
- Slightly more-complex solution is binary search. "Faster" but requires array to be in order.

Slide 5

Order of Magnitude of Algorithms

- Conventional wisdom (among computer scientists) is to write programs in a way that humans can understand, and let the compiler turn them into something that will run fast.
- One exception is “order of magnitude” of algorithm, however.
- Key idea is to think about how execution time (or some other measure, such as memory requirements) scales with “problem size”.
- Roughly analogous to order of magnitude of numbers — provide a way of grouping into classes in which all members of one class are sort of “the same” but members of different classes are not.

Slide 6

Order of Magnitude of Algorithms, Continued

- Typically written using “big-O” notation (e.g., $O(N)$, $O(N^2)$, etc.). Formal definition possible, but informally, $O(f(N))$ means that execution time (or whatever) for problem size N scales as $f(N)$. Examples:
- $f(N) = N$, $f(N) = 10N$, and $f(N) = N + 1000$ are all $O(N)$ (“linear”).
- $f(N) = N^2$, $f(N) = 2N^2$, and $f(N) = N^2 + 2N + 1$ are all $O(N^2)$.
- $f(N) = 2^N$ and $f(N) = 2^N + N$ are both $O(2^N)$ (“exponential”).
- (Compare using `gnuplot`.)

Slide 7

Order of Magnitude of Algorithms, Continued

- A key idea: For large enough problem sizes, algorithms with smaller orders of magnitude are faster, though this may not be true for small problem sizes.
- Another key idea: Some orders of magnitude (e.g., $O(2^N)$) are sufficiently "big" that solving problems of any non-trivial size is simply not feasible, so "wait until computers get faster" is probably not a good strategy. "Hm!"?
- Can help rule out algorithms that would not be practical/feasible for large problems.

  A famous(?) example — "traveling salesperson problem", for which all known algorithms require considering, for N cities, all possible permutations, making them $O(N!)$. Not reasonable! (Worth noting that there apparently *are* practical approximations. Still!)

Slide 8

Order of Magnitude of Algorithms, Continued

- As an example, look at bubble sort and selection sort.
- For both, "problem size" is the number of elements to sort, and a rough measure of how execution time scales with problem size is based on how many comparisons are needed, in the worst case.
- Again for both, total number of comparisons is $N(N-1)/2$, making them "$O(N^2)$".
- As another example, look at sequential search and binary search. The first is $O(N)$, but the second is . . . What? ($O(\log N)$)

Minute Essay

- None really — just sign in, unless questions?

Slide 9