CSE332: Data Abstractions
Lecture 12: Introduction to Sorting

Tyler Robison
Summer 2010
Introduction to sorting

- Stacks, queues, priority queues, and dictionaries all focused on providing one element at a time

- But often we know we want “all the data items” in some order
  - Anyone can sort, but a computer can sort faster
  - Very common to need data sorted somehow
    - Alphabetical list of people
    - Population list of countries
    - Search engine results by relevance
    - ...

- Different algorithms have different asymptotic and constant-factor trade-offs
  - No single ‘best’ sort for all scenarios
  - Knowing one way to sort just isn’t enough
More reasons to sort

General technique in computing:

Preprocess data to make subsequent operations faster

- Example: Sort the data so that you can
  - Find the $k^{th}$ largest in constant time for any $k$
  - Perform binary search to find an element in logarithmic time

Whether the performance of the preprocessing matters depends on

- Ways in which you’ll access it later
- How often the data will change
- How much data there is
The main problem, stated carefully

For now we will assume we have $n$ comparable elements in an array and we want to rearrange them to be in increasing order

Input:
- An array $A$ of data records
- A key value in each data record
- A comparison function (consistent and total):
  - Given keys $a$ & $b$, what is their relative ordering? $<$, $=$, $>$?
  - Ex: keys that implement Comparable or have a Comparator that can handle them

Effect:
- Reorganize the elements of $A$ such that for any $i$ and $j$,
  - if $i < j$ then $A[i] \leq A[j]$
- Usually unspoken assumption: $A$ must have all the same data it started with
- Could also sort in reverse order, of course

An algorithm doing this is a comparison sort
Variations on the basic problem

1. Maybe elements are in a linked list (could convert to array and back in linear time, but some algorithms needn’t do so)

2. Maybe in the case of ties we should preserve the original ordering
   - Sorts that do this naturally are called **stable sorts**
   - One way to sort twice, Ex: Sort movies by year, then for ties, alphabetically

3. Maybe we must not use more than $O(1)$ “auxiliary space”
   - Sorts meeting this requirement are called ‘in-place’ sorts
   - Not allowed to allocate extra array (at least not with size $O(n)$), but can allocate $O(1)$ # of variables
   - All work done by swapping around in the array

4. Maybe we can do more with elements than just compare two at a time
   - Comparison sorts assume we work using a binary ‘compare’ operator
   - In special cases we can sometimes get faster algorithms

5. Maybe we have too much data to fit in memory
   - Use an “external sorting” algorithm
The Big Picture

Simple algorithms: $O(n^2)$
- Insertion sort
- Selection sort
- Shell sort
- ...

Fancier algorithms: $O(n \log n)$
- Heap sort
- Merge sort
- Quick sort (avg)
- ...

Comparison lower bound: $\Omega(n \log n)$

Specialized algorithms: $O(n)$
- Bucket sort
- Radix sort

Handling huge data sets
- External sorting