Sorting Overview

- Some algorithms depend upon sorted data
  - binary search
  - grabbing elements within a value range
- Sort routines often available
  - need to understand properties to choose appropriately
- Sometimes toughest decision is choosing comparison operator
  - Last names
  - characters outside of ASCII, Latin-1
Insertion Sort Review

• Inherent when using sorted-list priority queue
  • enqueue at back
  • move forward until next element > priority
  • dequeueement gives sorted

• On an array
  • increment outer loop from 2\textsuperscript{nd} to last
    • decrement inner loop from current to 1\textsuperscript{st}
    • swap current as needed
Insertion Sort Example and Complexity

http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

• outer loop is set iteration – O(n)
• inner loop is a linear search – O(n)
• O(n**2)

• EXCEPT – what if the data is already sorted?
  • outer loop is still O(n)
  • inner loop is 1 comparison O(1)
  • O(n) – best case
Selection Sort Review

• Inherent in using unsorted-list priority queue
  • enqueue at back
  • dequeue by doing a linear search

• With an array
  • outer loop over entire sequence
    • set min to current
    • loop inner index from outer index to end
      • compare inner current to min, reset if necessary
    • swap outer current with min
Selection Sort Example and Complexity

- outer loop is set iteration, $O(n)$
- inner loop is set iteration, $O(n)$
- $O(n^{**2})$
- There is no best case scenario
Heap Sort Review

• Inherent when using a heap priority queue
  • enqueue at end of complete binary tree
    • bubble if necessary
  • dequeue root
    • re-bubble as necessary

• bubbling aka “heapifying”

• With an array
  • strip out the priority queue API
  • otherwise the same since the heap is 99% going to implemented as an array anyway
Heap Sort Example and Complexity

- enqueue and heapify, $O(\log n)$
- $n$ enqueues – $O(n \log n)$ to build heap
- dequeue and heapify, $O(\log n)$
- $n$ dequeues – $O(n \log n)$ to tear down heap
- $O(n \log n)$ for best and worst case
  - Technically, “bottom-up” vs “top-down” different complexities
  - Best case if already reverse/forward sorted for max/min heaps no heapifying happens $\Omega(n)$

Insertion, Selection, Merge Code

- various_sorts.py
Divide and Conquer

- General algorithm with uses outside of sorting
- Map/Reduce, scatter/gather

1. Recursively divide problem space into smaller, disjoint subsets
2. Recursively conquer the problem by combining previously solved subsets
Merge Sort

• Divide in halves until only 1 item in subset
• Conquer by merging subsets into sorted order
• Drawn as a merge sort tree (see webnotes/textbook)
Merge Sort Example and Complexities

- Merging length of each subset, $O(n + m) \rightarrow O(n)$
- Splitting original length, $O(\log n)$
- $O(n)$ work at $O(\log n)$ levels
- $O(n \log n)$
Merge sort code

- `merge_array.py`
- `merge_queue.py`
- `merge_nonrecurr.py`
  - “bottom up”
  - implicit splits
Quicksort

• Sorts during the split stage
• At current subset
  • select “pivot”
  • sort into L, E, G sets based on pivot
  • recurse on L, G subsets
• On return from recursion
  • merge by copying L, then E, then G
• Quicksort tree (see webnotes/textbook)
Quicksort Example and Complexities


• Compare to pivot, $O(n)$

• Done at each level of quicksort tree, $O(h)$
  • (with merge sort $h$ was guaranteed $\log n$)
  • best case $h$ is $\log n$, total $O(n \log n)$
  • worst case $h$ is $n$, total $O(n^2)$!

• Q: What makes the worst case?
Quicksort – controlling Height

- h factors
  - entropy of data
  - pivot choice
- randomized quicksort
  - random pivots
- median-of-three quicksort
  - median of first, middle, last elements
- hybrid quicksort
  - quicksort until L, G are below threshold, then do insertion sort, return to quicksort to merge
Quicksort Code

• quick_queue.py
• quick_inplace.py