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 things” in some order
  – Humans can sort, but computers can sort fast
  – Very common to need data sorted somehow
    • Alphabetical list of people
    • List of countries ordered by population
    • Search engine results by relevance
  • 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

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 things” in some order
  – Humans can sort, but computers can sort fast
  – Very common to need data sorted somehow
    • Alphabetical list of people
    • List of countries ordered by population
    • Search engine results by relevance
  • 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 elements in logarithmic time

Whether the performance of the preprocessing matters depends on
  – How often the data will change (and how much it will change)
  – How much data there is

Why Study Sorting in this Class?

• You might never need to reimplement a sorting algorithm yourself
  – Standard libraries will generally implement one or more (Java implements 2)
• You will almost certainly use sorting algorithms
  – Important to understand relative merits and expected performance
• Excellent set of algorithms for practicing analysis and comparing design techniques
  – Classic part of a data structures class, so you’ll be expected to know it

The main problem, stated carefully

For now, 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)

Effect:
  – Reorganize the elements of $A$ such that for any $i$ and $j$,
    if $i < j$ then $A[i] \leq A[j]$
  – (Also, $A$ must have exactly 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 ties need to be resolved by “original array position”
  – Sorts that do this naturally are called stable sorts
  – Others could tag each item with its original position and adjust comparisons accordingly (non-trivial constant factors)
3. Maybe we must not use more than $O(1)$ “auxiliary space”
  – Sorts meeting this requirement are called in-place sorts
4. Maybe we can do more with elements than just compare
  – Sometimes leads to faster algorithms
5. Maybe we have too much data to fit in memory
  – Use an “external sorting” algorithm
**Sorting: The Big Picture**

Surprising amount of neat stuff to say about sorting:

- **Simple algorithms:** $O(n^2)$
  - Insertion sort
  - Selection sort
  - Shell sort
  - ...
- **Fancier algorithms:** $O(n \log n)$
  - Heap sort
  - Merge sort
  - Quick sort
  - ...
- **Comparison lower bound:** $\Omega(n \log n)$
- **Specialized algorithms:** $O(n)$
  - Bucket sort
  - Radix sort
- **Handling huge datasets**

---

**Insertion Sort**

- **Idea:** At step $k$, put the $k^{th}$ element in the correct position among the first $k$ elements
- **Alternate way of saying this:**
  - Sort first two elements
  - Now insert $3^{rd}$ element in order
  - Now insert $4^{th}$ element in order
  - ...
- **"Loop invariant":** when loop index is $1$, first $1$ elements are sorted
- **Let’s see a visualization:** [link]
- **Time?**
  - Best-case $O(n)$
  - Worst-case $O(n^2)$
  - "Average" case $O(n^2)$

---

**Selection sort**

- **Idea:** At step $k$, find the smallest element among the not-yet-sorted elements and put it at position $k$
- **Alternate way of saying this:**
  - Find smallest element, put it $1^{st}$
  - Find next smallest element, put it $2^{nd}$
  - Find next smallest element, put it $3^{rd}$ ...
- **"Loop invariant":** when loop index is $i$, first $i$ elements are the $i$ smallest elements in sorted order
- **Let’s see a visualization:** [link]
- **Time?**
  - Best-case $O(n^2)$
  - Worst-case $O(n^2)$
  - "Average" case $O(n^2)$

---

**Insertion Sort vs. Selection Sort**

- Different algorithms
- Solve the same problem
- Have the same worst-case and average-case asymptotic complexity
  - Insertion-sort has better best-case complexity; preferable when input is “mostly sorted”
- Other algorithms are more efficient for large arrays that are not already almost sorted
  - Insertion sort may do well on small arrays
Aside: We Will Not Cover Bubble Sort

- It is not, in my opinion, what a “normal person” would think of
- It doesn’t have good asymptotic complexity: $O(n^2)$
- It’s not particularly efficient with respect to constant factors

Basically, almost everything it is good at some other algorithm is at least as good at:
- Perhaps people teach it just because someone taught it to them?

Fun, short, optional read:

The Big Picture

Surprising amount of juicy computer science: 2-3 lectures...

Simple algorithms: $O(n^2)$
Fancier algorithms: $O(n \log n)$
Comparison lower bound: $\Omega(n \log n)$
Specialized algorithms: $O(n)$
Handling huge data sets

Insertion sort
Selection sort
Heap sort
Shell sort
Merge sort
Bucket sort
Radix sort
Quick sort (avg)
... (not shown)

Heap sort

- Sorting with a heap is easy:
  - insert each arr[i], or better yet use buildHeap
  - for i in range(len(arr)):
    arr[i] = deleteMin()

- Worst-case running time: $O(n \log n)$

- We have the array-to-sort and the heap
  - So this is not an in-place sort
  - There’s a trick to make it in-place...

In-place heap sort

But this reverse sorts – how would you fix that?

- Treat the initial array as a heap (via buildHeap)
- When you delete the 1st element, put it at arr[n-i]
- That array location isn’t needed for the heap anymore!

“AVL sort”

- We can also use a balanced tree to:
  - insert each element: total time $O(n \log n)$
  - Repeatedly deleteMin: total time $O(n \log n)$
    - Better: in-order traversal $O(n)$, but still $O(n \log n)$ overall

- But this cannot be made in-place and has worse constant factors than heap sort
  - both are $O(n \log n)$ in worst, best, and average case
  - neither parallelizes well
  - heap sort is better

“Hash sort”???

- Don’t even think about trying to sort with a hash table!
- Finding min item in a hashtable is $O(n)$, so this would be a slower, more complicated selection sort
- And we’ve already seen that selection sort is pretty bad!
Divide and conquer

Very important technique in algorithm design

1. Divide problem into smaller parts
2. Independently solve the simpler parts
   - Think recursion
   - Or potential parallelism
3. Combine solution of parts to produce overall solution

(This technique has a long history.)

Divide-and-Conquer Sorting

Two great sorting methods are fundamentally divide-and-conquer

1. Mergesort: Sort the left half of the elements (recursively)
   Sort the right half of the elements (recursively)
   Merge the two sorted halves into a sorted whole
   (we covered Mergesort early in the quarter)
2. Quicksort: Pick a “pivot” element
   Divide elements into less-than pivot
   and greater-than pivot
   Sort the two divisions (recursively on each)
   Answer is: sorted-less-than then pivot then sorted-greater-than

Quick sort Overview

1. Pick a pivot element
2. Partition all the data into:
   A. The elements less than the pivot
   B. The pivot
   C. The elements greater than the pivot
3. Recursively sort A and C
4. The answer is, “as simple as A, B, C”

Think in Terms of Sets

Example, Showing Recursion

Details

Have not yet explained:

- How to pick the pivot element
  - Any choice is correct: data will end up sorted
  - But as analysis will show, want the low and high subsets to be about equal in size
- How to implement partitioning
  - In linear time
  - In place
Pivots

- **Best pivot?**
  - Median
  - Halve each time

- **Worst pivot?**
  - Greatest/least element
  - Problem of size \( n - 1 \)
  - \( O(n^2) \)

Potential pivot rules

While sorting \( arr \) from \( lo \) to \( hi - 1 \) …

- **Pick** \( arr[lo] \) or \( arr[hi-1] \)
  - Fast, but worst-case occurs with mostly sorted input

- **Pick random element in the range**
  - Does as well as any technique, but (pseudo)random number generation can be slow
  - Still probably the most elegant approach

- **Median of 3**, e.g., \( arr[lo], arr[hi-1], arr[(hi+lo)/2] \)
  - Common heuristic that tends to work well

Partitioning

- Conceptually simple, but hardest part to code up correctly
  - After picking pivot, need to partition in linear time in-place

- One approach (there are slightly fancier ones):
  1. Swap pivot with \( arr[lo] \)
  2. Use two cursors \( i \) and \( j \), starting at \( lo + 1 \) and \( hi - 1 \)
  3. while \( (i < j) \)
     - if \( (arr[j] > pivot) \) \( j-- \)
     - else if \( (arr[i] < pivot) \) \( i++ \)
     - else swap \( arr[i] \) with \( arr[j] \)
  4. Swap pivot with \( arr[i] \)

*skip step 4 if pivot ends up being least element

Example

- Step one: pick pivot as median of 3
  - \( lo = 0, hi = 10 \)

- Step two: move pivot to the \( lo \) position

Often have more than one swap during partitioning – this is a short example

Quick sort visualization

Analysis

- Best-case: Pivot is always the median
  \[ T(0) = T(1) = 1 \]
  \[ T(n) = 2 \cdot T(n/2) + n \]
  Linear-time partitioning
  Same recurrence as merge sort: \( O(n \log n) \)

- Worst-case: Pivot is always smallest or largest element
  \[ T(0) = T(1) = 1 \]
  \[ T(n) = T(n-1) + n \]
  Basically same recurrence as selection sort: \( O(n^2) \)

- Average-case (e.g., with random pivot)
  \( O(n \log n) \), not responsible for proof (in text)

Cutoffs

- For small \( n \), all that recursion tends to cost more than doing a quadratic sort
  - Remember asymptotic complexity is for large \( n \)

- Common engineering technique: switch algorithm below a cutoff
  - Reasonable rule of thumb: use insertion sort for \( n < 10 \)

- Notes:
  - Could also use a cutoff for merge sort
  - Cutoffs are also the norm with parallel algorithms
  - Switch to sequential algorithm
  - None of this affects asymptotic complexity

Cutoff pseudocode

```c
void quicksort(int[] arr, int lo, int hi) {
    if(hi - lo < CUTOFF)
        insertionSort(arr, lo, hi);
    else
        ...
}
```

Notice how this cuts out the vast majority of the recursive calls
- Think of the recursive calls to quicksort as a tree
- Trims out the bottom layers of the tree