CSE 373: Data Structures and Algorithms

Lecture 19: Comparison Sorting Algorithms

Instructor: Lilian de Greef
Quarter: Summer 2017
Today

• Intro to sorting
• Comparison sorting
  • Insertion Sort
  • Selection Sort
  • Heap Sort
  • Merge Sort
Mini-Announcements

• Homework 4 due today

• Homework 5 coming out today, due Friday 5:00pm
  • Can get started using material covered today
  • Can complete using material covered by Monday
Sorting

Now looking at algorithms instead of data structures!
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
    • List store catalogue by price
    • …

• 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 $k^{\text{th}}$ 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
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

Effect:
- Reorganize the elements of $A$ such that for any $i$ and $j$, if $i < j$ then $A[i] < A[j]$
- (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 sort

3. Maybe we must not use more than $O(1)$ “auxiliary space”
   • Sorts meeting this requirement are called in-place sort

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” 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 data sets**

**External sorting**
Real-world example demo time!

Help me sort some cards!
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 3rd element in order
  • Now insert 4th element in order
  • ...

• “Loop invariant”: when loop index is $i$, first $i$ elements are sorted

• Time?
  - Best-case $O(n)$
  - Worst-case $O(n^2)$
  - “Average” case $O(n^2)$ (see text)
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

• Time?
  Best-case \( \Theta (n^2) \)  Worst-case \( \Theta (n^2) \)  “Average” case \( \Theta (n^2) \)

\[
T(1) = 1 \\
T(n) = n + T(n-1)
\]
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
The Big Picture

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

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

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

• Sorting with a heap:
  • insert each arr[i], or better yet use buildHeap
  • for (i=0; i < arr.length; i++)
    arr[i] =

• 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

- Treat the initial array as a heap (via `buildHeap`)
- When you delete the $i^{th}$ element, put it at $\text{arr}[n-i]$
  - That array location isn't needed for the heap anymore!

But this reverse sorts – how would you fix that?
“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

• Compared to heap sort
  • both are $O(n \log n)$ in worst, best, and average case
  • neither parallelizes well
  • heap sort is can be done in-place, has better constant factors

Design decision: which would you choose between Heap Sort and AVL Sort? Why?
“Hash sort”???

Nope!

Finding min item in a hashtable is $O(n)$, so this would be a slower, more complicated selection sort

already terrible
Divide and conquer

Very important technique in algorithm design

1. Divide problem into smaller parts

2. Independently solve the simpler parts
   • Think recursion
   • Or parallelism

3. Combine solution of parts to produce overall solution

Two great sorting methods are fundamentally divide-and-conquer
(Merge Sort & Quicksort)
Merge Sort

Merge Sort: recursively...

• Sort the left half of the elements
• Sort the right half of the elements
• Merge the two sorted halves into a sorted whole
Real-world example demo time!

Help me sort some cards!
Merge sort

To sort array from position $lo$ to position $hi$:
- If range is 1 element long, it is already sorted!
- Else:
  - Sort from $lo$ to $(hi+lo)/2$
  - Sort from $(hi+lo)/2$ to $hi$
  - Merge the two halves together

Merging takes two sorted parts and sorts everything
- $O(n)$ but requires auxiliary space...
Merge Sort: Example focused on merging

Start with: 8 2 9 4 5 3 1 6

After recursion: 2 4 8 9 1 3 5 6
(not magic 😊)

Merge:
Use 3 “fingers”
and 1 more array

(After merge, copy back to original array)
Merge Sort: Example showing recursion

```
8 2 9 4 5 3 1 6
```

```
8 2 9 4

2 8

9 4

4 9

1 3 5 6

1 2 3 4 5 6 8 9
```
One way to practice on your own time:

- Make yourself an unsorted array
- Try using one of the sorting algorithms on it
- You know you got the right end result if it comes out sorted
- Can use the same example for merge sort as the previous slide to double check in-between steps
Some details: saving a little time

• What if the final steps of our merge looked like this:

<table>
<thead>
<tr>
<th>2</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

Main array

Auxiliary array

• Wasteful to copy to the auxiliary array just to copy back...
Some details: saving a little time

• If left-side finishes first, just stop the merge and copy back:

• If right-side finishes first, copy dregs into right then copy back
Some details: saving space and copying

Simplest / Worst:
  Use a new auxiliary array of size \((hi-lo)\) for every merge

Better:
  Use a new auxiliary array of size \(n\) for every merging stage

Better:
  Reuse same auxiliary array of size \(n\) for every merging stage

Best (but a little tricky):
  Don’t copy back – at 2\(^{nd}\), 4\(^{th}\), 6\(^{th}\), … merging stages, use the original array as the auxiliary array and vice-versa
    • Need one copy at end if number of stages is odd
Swapping Original / Auxiliary Array ("best")

- First recurse down to lists of size 1
- As we return from the recursion, swap between arrays

(ARGUALLY EASIER TO CODE UP WITHOUT RECURSION AT ALL)
Cool Resources

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

• http://www.sorting-algorithms.com/

• https://www.youtube.com/watch?v=t8g-iYGHpEA