CSE373: Data Structure & Algorithms
Lecture 20: Comparison Sorting

Aaron Bauer
Winter 2014
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?

• Unlikely you will ever 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
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 (avg)
  - ...

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

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

- **Handling huge data sets**
  - External sorting

---

Winter 2014  
CSE373: Data Structures & Algorithms
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 $i$, first $i$ elements are sorted

• Time?
  Best-case _____  Worst-case _____  “Average” case _____
**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 $i$, first $i$ elements are sorted.

- **Time?**
  
<table>
<thead>
<tr>
<th>Start Sorted</th>
<th>Start Reverse Sorted</th>
<th>“Average” Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

(Winter 2014)
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 _____  Worst-case _____  “Average” case _____
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\textsuperscript{st}
  – Find next smallest element, put it 2\textsuperscript{nd}
  – Find next smallest element, put it 3\textsuperscript{rd}
  – ...

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

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

  \[Always \ T(1) = 1 \text{ and } T(n) = n + T(n-1)\]
Mystery

This is one implementation of which sorting algorithm (for ints)?

```java
void mystery(int[] arr) {
    for(int i = 1; i < arr.length; i++) {
        int tmp = arr[i];
        int j;
        for(j=i; j > 0 && tmp < arr[j-1]; j--)
            arr[j] = arr[j-1];
        arr[j] = tmp;
    }
}
```

Note: Like with heaps, “moving the hole” is faster than unnecessary swapping (constant-factor issue)
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 non-small 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:

*Bubble Sort: An Archaeological Algorithmic Analysis*, Owen Astrachan, SIGCSE 2003

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)

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

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

Handling huge data sets
- External sorting
Heap sort

- Sorting with a heap is easy:
  - insert each \( arr[i] \), or better yet use buildHeap
  - for(\( i=0; \ i < \ arr.length; \ i++ \))
    \[ arr[i] = \text{deleteMin}(); \]

- 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

- 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

• 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
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

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
Mergesort

- To sort array from position $lo$ to position $hi$:
  - If range is 1 element long, it is already sorted! (Base case)
  - 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
Example, Focus on Merging

Start with:  

\[
\begin{array}{cccccccc}
8 & 2 & 9 & 4 & 5 & 3 & 1 & 6 \\
\end{array}
\]

After recursion: (not magic 😊)

\[
\begin{array}{cccccccc}
2 & 4 & 8 & 9 & 1 & 3 & 5 & 6 \\
\end{array}
\]

Merge:  
Use 3 “fingers” and 1 more array  

(After merge, copy back to original array)
Example, focus on merging

Start with:

\[ \begin{array}{cccccccc}
8 & 2 & 9 & 4 & 5 & 3 & 1 & 6 \\
\end{array} \]

After recursion:
(not magic 😊)

\[ \begin{array}{cccccccc}
2 & 4 & 8 & 9 & 1 & 3 & 5 & 6 \\
\end{array} \]

Merge:

Use 3 “fingers”
and 1 more array

(After merge, copy back to original array)
Example, focus on merging

Start with:

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

After recursion:
(not magic 😊)

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

Merge:
Use 3 “fingers”
and 1 more array

(After merge, copy back to original array)
Example, focus on merging

Start with:  

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

After recursion:  
(not magic 😊)

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

Merge:  
Use 3 “fingers”  
and 1 more array

| 1 | 2 | 3 |

(After merge,  
copy back to  
original array)
Example, focus on merging

Start with:

\[
\begin{array}{cccccccc}
8 & 2 & 9 & 4 & 5 & 3 & 1 & 6 \\
\end{array}
\]

After recursion:
(not magic 😊)

\[
\begin{array}{cccccccc}
2 & 4 & 8 & 9 & 1 & 3 & 5 & 6 \\
\end{array}
\]

Merge:

Use 3 “fingers”
and 1 more array

(After merge, copy back to original array)
Example, focus on merging

Start with:

\[
\begin{array}{cccccccc}
8 & 2 & 9 & 4 & 5 & 3 & 1 & 6 \\
\end{array}
\]

After recursion:
\( \text{not magic } \)

\[
\begin{array}{cccccccc}
2 & 4 & 8 & 9 & 1 & 3 & 5 & 6 \\
\end{array}
\]

Merge:
Use 3 “fingers”
and 1 more array

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & & & \\
\end{array}
\]

(After merge, copy back to original array)
Example, focus 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:

```
1 2 3 4 5 6
```

Use 3 “fingers” and 1 more array

(After merge, copy back to original array)
**Example, focus 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:

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

Use 3 “fingers” and 1 more array

(After merge, copy back to original array)
Example, focus on merging

Start with:

8 2 9 4 5 3 1 6

After recursion:
(not magic 😊)

2 4 8 9 1 3 5 6

Merge:
Use 3 “fingers”
and 1 more array

1 2 3 4 5 6 8 9

(After merge, copy back to original array)
Example, focus on merging

Start with:

After recursion: (not magic 😊)

Merge:
Use 3 “fingers” and 1 more array

(After merge, copy back to original array)
Example, Showing Recursion

\[ \begin{array}{cccccccc}
8 & 2 & 9 & 4 & 5 & 3 & 1 & 6 \\
\end{array} \]

Divide

Divide

Divide

1 Element

Merge

Merge

Merge

Merge
Some details: saving a little time

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

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

Main array

```
  1 2 3 4 5 6
```

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

(Arguably easier to code up without recursion at all)
**Linked lists and big data**

We defined sorting over an array, but sometimes you want to sort linked lists.

One approach:
- Convert to array: $O(n)$
- Sort: $O(n \log n)$
- Convert back to list: $O(n)$

Or: merge sort works very nicely on linked lists directly
- Heapsort and quicksort do not
- Insertion sort and selection sort do but they’re slower

Merge sort is also the sort of choice for external sorting
- Linear merges minimize disk accesses
- And can leverage multiple disks to get streaming accesses
Analysis

Having defined an algorithm and argued it is correct, we should analyze its running time and space:

To sort \( n \) elements, we:
- Return immediately if \( n=1 \)
- Else do 2 subproblems of size \( n/2 \) and then an \( O(n) \) merge

Recurrence relation:
\[
T(1) = c_1 \\
T(n) = 2T(n/2) + c_2 n
\]
One of the recurrence classics…

For simplicity let constants be 1 – no effect on asymptotic answer

\[
\begin{align*}
T(1) &= 1 \\
T(n) &= 2T(n/2) + n \\
&= 2(2T(n/4) + n/2) + n \\
&= 4T(n/4) + 2n \\
&= 4(2T(n/8) + n/4) + 2n \\
&= 8T(n/8) + 3n \\
&= \ldots \\
&= 2^kT(n/2^k) + kn
\end{align*}
\]

So total is \(2^kT(n/2^k) + kn\) where \(n/2^k = 1\), i.e., \(\log n = k\)

That is, \(2^{\log n} T(1) + n \log n\)

\[= n + n \log n\]

\[= O(n \log n)\]
Or more intuitively…

This recurrence is common you just “know” it’s $O(n \log n)$

Merge sort is relatively easy to intuit (best, worst, and average):
- The recursion “tree” will have $\log n$ height
- At each level we do a total amount of merging equal to $n$
Quicksort

• Also uses divide-and-conquer
  – Recursively chop into two pieces
  – Instead of doing all the work as we merge together, we will do all the work as we recursively split into halves
  – Unlike merge sort, does not need auxiliary space

• $O(n \log n)$ on average 😊, but $O(n^2)$ worst-case 😞

• Faster than merge sort in practice?
  – Often believed so
  – Does fewer copies and more comparisons, so it depends on the relative cost of these two operations!
Quicksort 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”

(Alas, there are some details lurking in this algorithm)
Think in Terms of Sets

S

S₁

S₂

select pivot value

partition S

Quicksort(S₁) and Quicksort(S₂)

Presto! S is sorted

[Weiss]
Example, Showing Recursion

Divide

Conquer

1 Element

Divide

Conquer

Divide

Conquer
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 two partitions 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 \texttt{arr} from \texttt{lo} (inclusive) to \texttt{hi} (exclusive)...

- Pick \texttt{arr[lo]} or \texttt{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., \texttt{arr[lo]}, \texttt{arr[hi-1]}, \texttt{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 \texttt{arr[lo]}
  2. Use two fingers \(i\) and \(j\), starting at \(lo+1\) and \(hi-1\)
  3. \textbf{while} \(i < j\)
     \hspace{1em} \textbf{if} \ (\texttt{arr[j]} > \texttt{pivot}) \ j--
     \hspace{1em} \textbf{else if} \ (\texttt{arr[i]} < \texttt{pivot}) \ i++
     \hspace{1em} \textbf{else} \ swap \ \texttt{arr[i]} \ with \ \texttt{arr[j]}
  4. Swap pivot with \texttt{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

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
8 & 1 & 4 & 9 & 0 & 3 & 5 & 2 & 7 & 6
\end{array}
\]

• Step two: move pivot to the lo position

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
6 & 1 & 4 & 9 & 0 & 3 & 5 & 2 & 7 & 8
\end{array}
\]
Example

Now partition in place

Move fingers

Swap

Move fingers

Move pivot

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

• Best-case: Pivot is always the median
  \[ T(0)=T(1)=1 \]
  \[ T(n)=2T(n/2) + n \quad -- \text{linear-time partition} \]
  Same recurrence as mergesort: \( O(n \log n) \)

• Worst-case: Pivot is always smallest or largest element
  \[ T(0)=T(1)=1 \]
  \[ T(n) = 1T(n-1) + n \]
  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 skeleton

```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