CSE373: Data Structure & Algorithms
Lecture 19: Comparison Sorting

Lauren Milne
Summer 2015
Admin

• Homework 5 due next Wednesday

• START SOON!!

• Homework 6 assigned next Wednesday (due the week after)

• Final will be last day in class (Friday 8/21)

• Pick up any midterms after class today or in office hours
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
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 $A[i] \leq A[j]$
- (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

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 data sets
- External sorting
**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

- Let’s see a visualization ([http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html](http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html))

- 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

- Let’s see a visualization ([http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html](http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html))

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

• Let’s see a visualization (http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html)

• 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$^{st}$
  - 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 (http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html)

- Time?
  
  Best-case $O(n^2)$  
  Worst-case $O(n^2)$  
  “Average” case $O(n^2)$
  
  Always $T(1) = 1$ and $T(n) = n + T(n-1)$
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
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] = 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 `arr[n-i]`
  - That array location isn’t needed for the heap anymore!

```
4 7 5 9 8 6 10 3 2 1
```

But this reverse sorts – how would you fix that?

```
5 7 6 9 8 10 4 3 2 1
```

arr[n-i] = `deleteMin()`
“AVL sort”

- We can also use a balanced tree to:
  - \textbf{insert} each element: total time $O(n \log n)$
  - Repeatedly \textbf{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
“Hash sort”???

• Nope!

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

• And selection sort is 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
**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
Merge sort

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:
\( \text{(not magic 😊)} \)

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

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

\[
\begin{array}{cccc}
1 & 2 \\
\end{array}
\]

(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:

![Array](8 2 9 4 5 3 1 6)

After recursion:

(not magic 😊)

![Array](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

(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:

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:

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

(not magic 😊)

Merge:

Use 3 “fingers”
and 1 more array

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

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

Start with:

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

After recursion:
(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:

```
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)

```
1 2 3 4 5 6 8 9
```
Example, Showing Recursion

```
8 2 9 4 5 3 1 6

Divide
Divide
Divide
1 Element
Merge
Merge
Merge
```
Merge sort visualization

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:

  ![Diagram of left-side finishing first]

• If right-side finishes first, copy dregs into right then copy back

  ![Diagram of right-side finishing first]
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:
\[
T(1) = c_1 \\
T(n) = 2T(n/2) + c_2 n
\]
Analysis 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 $n$
Analysis more formally
(One of the recurrence classics)

For simplicity, ignore constants (let constants be )

\[ T(1) = 1 \]
\[ T(n) = 2T(n/2) + n \]
\[ = 2(2T(n/4) + n/2) + n \]
\[ = 4T(n/4) + 2n \]
\[ = 4(2T(n/8) + n/4) + 2n \]
\[ = 8T(n/8) + 3n \]
\[ \ldots \]
\[ = 2^k T(n/2^k) + kn \]
Analysis more formally
(One of the recurrence classics)

For simplicity, ignore constants (let constants be )

\[ T(1) = 1 \]
\[ T(n) = 2T(n/2) + n \]

\[ = 2(2T(n/4) + n/2) + n \]
\[ = 4T(n/4) + 2n \]
\[ = 4(2T(n/8) + n/4) + 2n \]
\[ = 8T(n/8) + 3n \]

\[ \ldots \]
\[ = 2^k T(n/2^k) + kn \]

We will continue to recurse until we reach the base case, i.e. \( T(1) \)
for \( T(1), \ n/2^k = 1, \ i.e., \log n = k \)

So the total amount of work will be
\[ = 2^k T(n/2^k) + kn = 2^{\log n} T(1) + n \log n = n + n \log n = O(n \log n) \]
Next lecture

• Quick sort 😊