CSE332: Data Abstractions
Lecture 13: Comparison Sorting

Dan Grossman
Spring 2012
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

Start with: How would “normal people (?)” sort?
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 $k^{\text{th}}$ element in the correct position among the first $k$ elements

- Alternate way of saying this:
  - Sort first two elements
  - Now insert 3$^{\text{rd}}$ element in order
  - Now insert 4$^{\text{th}}$ element in order
  - ...

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

- Time?
  
<table>
<thead>
<tr>
<th>Start Condition</th>
<th>Best-case</th>
<th>Worst-case</th>
<th>“Average” case</th>
</tr>
</thead>
<tbody>
<tr>
<td>start sorted</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>start reverse sorted</td>
<td></td>
<td></td>
<td>(see text)</td>
</tr>
</tbody>
</table>

Spring 2012 CSE332: Data Abstractions
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 1st
  – Find next smallest element, put it 2nd
  – Find next smallest element, put it 3rd
  – …

• “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\(^{\text{st}}\)
  – Find next smallest element, put it 2\(^{\text{nd}}\)
  – Find next smallest element, put it 3\(^{\text{rd}}\)
  – …

• “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 common 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

- As you saw on Project 2, 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 \( \text{arr}[n-i] \)
  - That array location isn’t needed for the heap anymore!

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

heap part

sorted part

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

heap part

sorted part

arr[\(n-i\)] =
`deleteMin()`
“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

• Don’t even think about trying to sort with a hash table
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

(The name “divide and conquer” is rather clever.)
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

8 2 9 4 5 3 1 6

- 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:  

```
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:

| 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 |   |   |   |   |   |   |

(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

(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

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

(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

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

(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:  

| 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

(After merge, copy back to original array)

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

Spring 2012 CSE332: Data Abstractions
Example, Showing Recursion

```
8   2   9   4   5   3   1   6
Divide 8 2 1 6 5 3 1 6
Divide 8 2 9 4 5 3 1 6
Divide 8 2 9 4 5 3 1 6
Divide 8 2 9 4 5 3 1 6
1 Element 8 2 9 4 5 3 1 6
Merge 2 8 4 9 1 6 3 5
Merge 2 4 8 9 1 6 3 5
Merge 2 4 8 9 1 6 3 5
Merge 2 4 8 9 1 6 3 5
```
Some details: saving a little time

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

![Diagram]

- 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:

$T(1) = c_1$

$T(n) = 2T(n/2) + c_2n$
One of the recurrence classics...

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

\[ T(1) = 1 \]
\[ T(n) = 2T(n/2) + n \]
\[ = 2(2T(n/4) + n/2) + n \]
\[ = 4T(n/4) + 2n \]
\[ = 4(2T(n/8) + n/4) + 2n \]
\[ = 8T(n/8) + 3n \]
\[ \ldots \]
\[ = 2^k T(n/2^k) + kn \]

So total is \( 2^k T(n/2^k) + kn \) 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 halves
  - 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

select pivot value

partition S

Quicksort(S₁) and Quicksort(S₂)

Presto! S is sorted

[Weiss]
Example, Showing Recursion

Divide
Divide
Divide
1 Element
Conquer
Conquer
Conquer

8 2 9 4 5 3 1 6

Spring 2012
CSE332: Data Abstractions
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 $\text{arr}[\text{lo}]$
  2. Use two fingers $i$ and $j$, starting at $\text{lo}+1$ and $\text{hi}-1$
  3. while ($i < j$)
     
     if ($\text{arr}[j] > \text{pivot}$) $j--$
     else if ($\text{arr}[i] < \text{pivot}$) $i++$
     else swap $\text{arr}[i]$ with $\text{arr}[j]$
  4. Swap pivot with $\text{arr}[i]$ *

*skip step 4 if pivot ends up being least element
Example

• Step one: pick pivot as median of 3
  – $lo = 0$, $hi = 10$
  
  \[
  \begin{array}{ccccccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
  \hline
  8 & 1 & 4 & 9 & 0 & 3 & 5 & 2 & 7 & 6
  \end{array}
  \]

• Step two: move pivot to the $lo$ position
  
  \[
  \begin{array}{ccccccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
  \hline
  6 & 1 & 4 & 9 & 0 & 3 & 5 & 2 & 7 & 8
  \end{array}
  \]
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