CSE332: Data Abstractions

Lecture 14: Beyond 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
How Fast Can We Sort?

- Heap sort & Merge sort have $O(n \log n)$ worst-case running time
- Quick sort has $O(n \log n)$ average-case running time
- These bounds are all tight, actually $\Theta(n \log n)$
- So maybe we need to dream up another algorithm with a lower asymptotic complexity, such as $O(n)$ or $O(n \log \log n)$
  - Instead: prove that this is impossible
    - Assuming our comparison model: The only operation an algorithm can perform on data items is a 2-element comparison
A General View of Sorting

• Assume we have $n$ elements to sort
  – For simplicity, assume none are equal (no duplicates)

• How many *permutations* of the elements (possible orderings)?

• Example, $n=3$
  
  
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>a1</td>
<td>a2</td>
</tr>
<tr>
<td>a0</td>
<td>a2</td>
<td>a1</td>
</tr>
<tr>
<td>a1</td>
<td>a0</td>
<td>a2</td>
</tr>
<tr>
<td>a1</td>
<td>a2</td>
<td>a0</td>
</tr>
<tr>
<td>a2</td>
<td>a0</td>
<td>a1</td>
</tr>
<tr>
<td>a2</td>
<td>a1</td>
<td>a0</td>
</tr>
</tbody>
</table>

• In general, $n$ choices for least element, $n-1$ for next, $n-2$ for next, …
  – $n(n-1)(n-2)\ldots(2)(1) = n!$ possible orderings
Counting Comparisons

• So every sorting algorithm has to “find” the right answer among the $n!$ possible answers
  – Starts “knowing nothing” and gains information with each comparison
  – Intuition: Each comparison can at best eliminate half the remaining possibilities
  – Must narrow answer down to a single possibility

• What we will show:
  Any sorting algorithm must do at least $(1/2)n \log_2 n - (1/2)n$ (which is $\Omega(n \log n)$) comparisons
  – Otherwise there are at least two permutations among the $n!$ possible that cannot yet be distinguished, so the algorithm would have to guess and could be wrong
Counting Comparisons

• Don’t know what the algorithm is, but it cannot make progress without doing comparisons
  – Eventually does a first comparison “is $a < b$ ?"
  – Can use the result to decide what second comparison to do
  – Etc.: comparison $k$ can be chosen based on first $k-1$ results

• Can represent this process as a decision tree
  – Nodes contain “set of remaining possibilities”
  – Edges are “answers from a comparison”
  – The algorithm does not actually build the tree; it’s what our proof uses to represent “the most the algorithm could know so far” as the algorithm progresses
One Decision Tree for n=3

- The leaves contain all the possible orderings of a, b, c
- A different algorithm would lead to a different tree
Example if \( a < c < b \)
What the Decision Tree Tells Us

• A binary tree because each comparison has 2 outcomes
  – (No duplicate elements)
  – (Would have 1 outcome if a comparison is redundant)

• Because any data is possible, any algorithm needs to ask enough questions to produce all $n!$ answers
  – Each answer is a different leaf
  – So the tree must be big enough to have $n!$ leaves
  – Running any algorithm on any input will at best correspond to a root-to-leaf path in some decision tree with $n!$ leaves
  – So no algorithm can have worst-case running time better than the height of a tree with $n!$ leaves

• Worst-case number-of-comparisons for an algorithm is an input leading to a longest path in algorithm’s decision tree
Where are we

• Proven: No comparison sort can have worst-case running time better than the height of a binary tree with $n!$ leaves
  – Turns out average-case is same asymptotically
  – A comparison sort could be worse than this height, but it cannot be better

• Now: a binary tree with $n!$ leaves has height $\Omega(n \log n)$
  – Factorial function grows very quickly
  – Height could be more, but cannot be less

• Conclusion: **Comparison sorting is $\Omega(n \log n)$**
  – An amazing computer-science result: proves all the clever programming in the world cannot sort in linear time
Lower bound on height

- The height of a binary tree with $L$ leaves is at least $\log_2 L$.
- So the height of our decision tree, $h$:

\[
\begin{align*}
h & \geq \log_2 (n!)
\geq \log_2 n + \log_2 (n-1) + \ldots + \log_2 1
\geq \log_2 n + \log_2 (n-1) + \ldots + \log_2 (n/2)
\geq \log_2 (n/2) + \log_2 (n/2) + \ldots + \log_2 (n/2)
= (n/2)\log_2 (n/2)
= (n/2)(\log_2 n - \log_2 2)
= (1/2)n\log_2 n - (1/2)n
= \Omega (n \log n)
\end{align*}
\]
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

huh???
- Change the model – assume more than items can be compared!
**BucketSort (a.k.a. BinSort)**

- If all values to be sorted are known to be integers between 1 and $K$ (or any small range)
  - Create an array of size $K$
  - Put each element in its proper bucket (a.k.a. bin)
  - *If* data is only integers, no need to store more than a *count* of how times that bucket has been used
- Output result via linear pass through array of buckets

<table>
<thead>
<tr>
<th>count</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

- Example:
  - $K=5$
  - input: $(5,1,3,4,3,2,1,1,5,4,5)$
  - output: $1,1,1,2,3,3,4,4,5,5,5$
Analyzing Bucket Sort

• Overall: $O(n+K)$
  – Linear in $n$, but also linear in $K$
  – $\Omega(n \log n)$ lower bound does not apply because this is not a comparison sort

• Good when $K$ is smaller (or not much larger) than $n$
  – Do not spend time doing comparisons of duplicates

• Bad when $K$ is much larger than $n$
  – Wasted space; wasted time during final linear $O(K)$ pass

• For data in addition to integer keys, use list at each bucket
Radix sort

• Radix = “the base of a number system”
  – Examples will use 10 because we are used to that
  – In implementations use larger numbers
    • For example, for ASCII strings, might use 128

• Idea:
  – Bucket sort on one digit at a time
    • Number of buckets = radix
    • Starting with least significant digit
    • Keeping sort stable
  – Invariant: After $k$ passes (digits), the last $k$ digits are sorted

• Aside: Origins go back to the 1890 U.S. census
# Example

Radix = 10

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>721</td>
<td></td>
<td></td>
<td></td>
<td>537</td>
<td>478</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td>67</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Input: 478</td>
<td>9</td>
<td>721</td>
<td>3</td>
<td>143</td>
<td>537</td>
<td>67</td>
<td>478</td>
<td>38</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

First pass: bucket sort by ones digit

Order now: 721 537 478 38 143 9 67 3
### Example

Radix = 10

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>721</td>
<td></td>
<td>3</td>
<td>143</td>
<td></td>
<td>537</td>
<td>67</td>
<td>478</td>
<td>38</td>
<td>9</td>
</tr>
</tbody>
</table>

Second pass:

**stable** bucket sort by tens digit

Order now:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>9</td>
<td>721</td>
<td>537</td>
<td>38</td>
<td>143</td>
<td>67</td>
<td>478</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Order was:

721 3 143 537 67 478 38 9

---

Spring 2012 CSE332: Data Abstractions
Example

Radix = 10

Order was: 3 9 721 537 38 143 67 478 537 721

Order now: 3 9 38 67 143 67 478 537 721

Third pass:

stable bucket sort by 100s digit
**Analysis**

Input size: \( n \)
Number of buckets = Radix: \( B \)
Number of passes = “Digits”: \( P \)

Work per pass is 1 bucket sort: \( O(B+n) \)

Total work is \( O(P(B+n)) \)

Compared to comparison sorts, sometimes a win, but often not
- Example: Strings of English letters up to length 15
  - \( 15 \times (52 + n) \)
  - This is less than \( n \log n \) only if \( n > 33,000 \)
  - Of course, cross-over point depends on constant factors of the implementations
    - And radix sort can have poor locality properties
Last Slide on Sorting

- Simple $O(n^2)$ sorts can be fastest for small $n$
  - Selection sort, Insertion sort (latter linear for mostly-sorted)
  - Good for “below a cut-off” to help divide-and-conquer sorts
- $O(n \log n)$ sorts
  - Heap sort, in-place but not stable nor parallelizable
  - Merge sort, not in place but stable and works as external sort
  - Quick sort, in place but not stable and $O(n^2)$ in worst-case
    - Often fastest, but depends on costs of comparisons/copies
- $\Omega (n \log n)$ is worst-case and average lower-bound for sorting by comparisons
- Non-comparison sorts
  - Bucket sort good for small number of key values
  - Radix sort uses fewer buckets and more phases
- Best way to sort? It depends!