CSE 332: Data Abstractions
Lecture 13: Beyond Comparison Sorting

Ruth Anderson
Winter 2015
Today

- Sorting
  - Comparison sorting
  - Beyond comparison sorting
The Big Picture

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?

- Heapsort & mergesort have $O(n \log n)$ worst-case running time
- Quicksort has $O(n \log n)$ average-case running times
- 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 Different View of Sorting

• Assume we have \( n \) elements to sort
  – And for simplicity, none are equal (no duplicates)

• How many \textit{permutations} (possible orderings) of the elements?

• Example, \( n=3 \),
A Different View of Sorting

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

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

• Example, $n=3$, six possibilities
  

• In general, $n$ choices for least element, then $n-1$ for next, then $n-2$ for next, …
  – $n(n-1)(n-2)...(2)(1) = n!$ possible orderings
Describing every comparison sort

• A different way of thinking of sorting is that the sorting algorithm has to “find” the right answer among the n! possible answers
  – Starts “knowing nothing”, “anything is possible”
  – Gains information with each comparison, eliminating some possibilities
    • Intuition: At best, each comparison can eliminate half of the remaining possibilities
  – In the end narrows down to a single possibility
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”
  – At root, anything is possible; no option eliminated
  – 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** $a < c < b$

- **possible orders**
  - $a < b < c$, $b < c < a$, $a < c < b$, $c < a < b$,
  - $b < a < c$, $c < b < a$

- **actual order**
  - $a < b < c$, $b < c < a$, $b < a < c$
What the decision tree tells us

• A binary tree because each comparison has 2 outcomes
  – Perform only comparisons between 2 elements; binary result
    • Ex: Is a<b? Yes or no?
  – We assume no duplicate elements
  – Assume algorithm doesn’t ask redundant questions
• 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
- Fine, *how tall is a binary tree with \( n! \) leaves?*

**Now:** Show that a binary tree with \( n! \) leaves has height \( \Omega(n \log n) \)

- That is, \( n \log n \) is the lower bound, the height must be at least this, could be more, (in other words your comparison sorting algorithm could take longer than this, but it won’t be faster)
- Factorial function grows very quickly

Then we’ll conclude that: *(Comparison) Sorting is \( \Omega(n \log n) \)*

- This is an amazing computer-science result: proves all the clever programming in the world can’t sort in linear time!
Lower bound on Height

- A binary tree of height $h$ has at most how many leaves?
  $L \leq \underline{\phantom{0}\phantom{0}\phantom{0}\phantom{0}}$

- A binary tree with $L$ leaves has height at least:
  $h \geq \underline{\phantom{0}\phantom{0}\phantom{0}\phantom{0}}$

- The decision tree has how many leaves: ______
- So the decision tree has height:
  $h \geq \underline{\phantom{0}\phantom{0}\phantom{0}\phantom{0}}$
Lower bound on Height

- A binary tree of height $h$ has at most how many leaves?
  $L \leq 2^h$

- A binary tree with $L$ leaves has height at least:
  $h \geq \log_2 L$

- The decision tree has how many leaves: $N!$
- So the decision tree has height:
  $h \geq \log_2 N!$
**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, $h$:

\[
h \geq \log_2 (n!)
\]

\[
= \log_2 (n^*(n-1)^*(n-2)*...*(2)*(1))
\]

\[
= \log_2 n + \log_2 (n-1) + ... + \log_2 1
\]

\[
\geq \log_2 n + \log_2 (n-1) + ... + \log_2 (n/2)
\]

\[
\geq (n/2) \log_2 (n/2)
\]

\[
= (n/2)(\log_2 n - \log_2 2)
\]

\[
= (1/2)n \log_2 n - (1/2)n
\]

"\(=\)" \(\Omega\ (n \log n)\)
The Big Picture

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???
- Change the model – assume more than ‘compare(a,b)’
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 $K$, and 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 many times that bucket has been used
- Output result via linear pass through array of buckets

```
count array
1
2
3
4
5
```

- Example:
  - $K=5$
  - Input: $(5,1,3,4,3,2,1,1,5,4,5)$
  - output:
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 $K$, and 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 many 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: $1,1,1,2,3,3,4,4,5,5,5$

What is the running time?
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 range, $K$, is smaller (or not much larger) than $n$
  – (We don’t spend time doing lots of 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
**Bucket Sort with Data**

- Most real lists aren’t just #’s; we have data
- Each bucket is a list (say, linked list)
- To add to a bucket, place at end O(1) (keep pointer to last element)

<table>
<thead>
<tr>
<th>count array</th>
<th>Rocky V</th>
<th>Harry Potter</th>
<th>Casablanca</th>
<th>Star Wars</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Example: Movie ratings: 1=bad,… 5=excellent
- Input=
  - 5: Casablanca
  - 3: Harry Potter movies
  - 1: Rocky V
  - 5: Star Wars

**Result:** 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars
This result is **stable**; Casablanca still before Star Wars
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, sort with Bucket Sort
    - Keeping sort stable
  - Do one pass per digit
- **Invariant**: After $k$ passes, 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>721</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>537</td>
<td></td>
<td></td>
</tr>
<tr>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>67</td>
<td></td>
<td></td>
</tr>
<tr>
<td>478</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**First pass:**
1. bucket sort by ones digit
2. Iterate thru and collect into a list
   - List is sorted by first digit

**Order now:**
1. 721
2. 3
3. 143
4. 537
5. 67
6. 478
7. 38
8. 9
**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>1</td>
<td>721</td>
<td>3</td>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td>537</td>
<td>67</td>
<td>478</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>721</td>
<td>537</td>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td>67</td>
<td>478</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Order was: 721 3 143 537 67 478 38 9

Second pass:

- **stable** bucket sort by tens digit

If we chop off the 100’s place, these #s are sorted

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

2/02/2015
### 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>3</td>
<td>143</td>
<td>721</td>
<td>537</td>
<td>143</td>
<td>67</td>
<td>478</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>39</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Order was: 3

Order now: 3

Third pass:

*stable* bucket sort by 100s digit

Only 3 digits: We’re done!
RadixSort

- Input: 126, 328, 636, 341, 416, 131, 328

BucketSort on lsd:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

BucketSort on next-higher digit:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

BucketSort on msd:
Analysis of Radix Sort

Performance depends on:

• Input size: $n$
• Number of buckets = Radix: $B$
  – e.g. Base 10 #: 10; binary #: 2; Alpha-numeric char: 62
• Number of passes = “Digits”: $P$
  – e.g. Ages of people: 3; Phone #: 10; Person’s name: ?

• Work per pass is 1 bucket sort: ___________
  – Each pass is a Bucket Sort
• Total work is ___________
  – We do ‘$P$’ passes, each of which is a Bucket Sort
Analysis of Radix Sort

Performance depends on:

- Input size: \( n \)
- Number of buckets = Radix: \( B \)
  - e.g. Base 10 #: 10; binary #: 2; Alpha-numeric char: 62
- Number of passes = “Digits”: \( P \)
  - e.g. Ages of people: 3; Phone #: 10; Person’s name: ?

- Work per pass is 1 bucket sort: \( O(B+n) \)
  - Each pass is a Bucket Sort
- Total work is \( O(P(B+n)) \)
  - We do ‘P’ passes, each of which is a Bucket Sort
Comparison to Comparison Sorts

Compared to comparison sorts, sometimes a win, but often not

– Example: Strings of English letters up to length 15
  • Approximate run-time: $15 \times (52 + n)$
  • This is less than $n \log n$ only if $n > 33,000$
  • Of course, cross-over point depends on constant factors of the implementations plus $P$ and $B$
    – And radix sort can have poor locality properties
– Not really practical for many classes of keys
  • Strings: Lots of buckets
Recap: Features of Sorting Algorithms

In-place

- Sorted items occupy the same space as the original items. (No copying required, only O(1) extra space if any.)

Stable

- Items in input with the same value end up in the same order as when they began.

Examples:

- Merge Sort - not in place, stable
- Quick Sort - in place, not stable
Sorting massive data: External Sorting

Need sorting algorithms that minimize disk/tape access time:

• Quicksort and Heapsort both jump all over the array, leading to expensive random disk accesses
• Mergesort scans linearly through arrays, leading to (relatively) efficient sequential disk access

Basic Idea:

• Load chunk of data into Memory, sort, store this “run” on disk/tape
• Use the Merge routine from Mergesort to merge runs
• Repeat until you have only one run (one sorted chunk)

• Mergesort can leverage multiple disks
• Weiss gives some examples
Sorting Summary

• 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!