CSE332: Data Abstractions
Lecture 11: Beyond Comparison Sorting

James Fogarty
Winter 2012
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
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 ]
Quicksort 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)
  – \( O(n \log n) \) (see text)
Quicksort Cutoffs

• For small $n$, recursion tends to cost more than a quadratic sort
  – Remember asymptotic complexity is for large $n$
  – Also, recursive calls add a lot of overhead for small $n$

• Common 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
void quicksort(int[] arr, int lo, int hi) {
    if(hi - lo < CUTOFF)
        insertionSort(arr,lo,hi);
    else
        ...
}

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

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

Mergesort can very nicely work directly on linked lists
- Heapsort and Quicksort do not
- Insertion sort and Selection sort can, but they are slower

Mergesort is also the sort of choice for external sorting
- Quicksort and Heapsort jump all over the array
- Mergesort scans linearly through arrays
- In-memory sorting of blocks can be combined with larger sorts
- Mergesort can leverage multiple disks
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 we prove that this is impossible when the primary operation is comparison of pairs of elements
Permutations

- Assume we have \( n \) elements to sort
  - And for simplicity, assume none are equal (i.e., no duplicates)

- How many permutations of the elements (possible orderings)?

- Example, \( n=3 \)
  
  \[
  \begin{align*}
  \end{align*}
  \]
  
  6 possible orderings

- In general, \( n \) choices for first, \( n-1 \) for next, \( n-2 \) for next, etc.
  - \( n(n-1)(n-2)\ldots(2)(1) = n! \) possible orderings
Representing Every Comparison Sort

- Algorithm must “find” the right answer among $n!$ possible answers

- Starts “knowing nothing” and gains information with each comparison
  - Intuition is that each comparison can, at best, eliminate half of the remaining possibilities

- Can represent this process as a decision tree
  - Nodes contain “remaining possibilities”
  - Edges are “answers from a comparison”
  - This is not a data structure, it’s what our proof uses to represent “the most any algorithm could know”
Decision Tree for $n = 3$

The leaves contain all the possible orderings of $a, b, c$
What the Decision Tree Tells Us

• A binary tree because each comparison has 2 outcomes
  – No duplicate elements
  – Assume algorithm not so dumb as to ask redundant questions

• Because any data is possible, any algorithm needs to ask enough questions to decide among all n! answers
  – Every answer is a leaf (no more questions to ask)
  – So the tree must be big enough to have n! leaves
  – Running any algorithm on any input will at best correspond to one root-to-leaf path in the decision tree
  – So no algorithm can have worst-case running time better than the height of the decision tree
Example

possible orders

actual order
Where are We

Proven: No comparison sort can have worst-case better than: 
the height of a binary tree with \( n! \) leaves 
– Turns out average-case is same asymptotically 
– So how tall is a binary tree with \( n! \) leaves?

Now: Show that a binary tree with \( n! \) leaves has height \( \Omega(n \log n) \) 
– \( n \log n \) is the lower bound, the height must be at least this 
– It could be more (in other words, your comparison sorting algorithm could take longer than this, but can not be faster) 
– Factorial function grows very quickly

Conclude that: \((\text{Comparison}) \text{ Sorting is } \Omega(n \log n)\) 
– 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

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

$$h \geq \log_2 (n!)
= \log_2 (n^*(n-1)^*(n-2)...(2)(1))$$

$$= \log_2 n + \log_2 (n-1) + ... + \log_2 1$$

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

$$\geq (n/2) \log_2 (n/2)$$

each of the $n/2$ terms left is $\geq \log_2 (n/2)$

$$\geq (n/2)(\log_2 n - \log_2 2)$$

$$\geq (1/2)n \log_2 n - (1/2)n$$

“=“ $\Omega (n \log n)$
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
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 anything 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 array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

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 $K$
  - Put each element in its proper bucket (a.k.a. bin)
  - *If* data is only integers, no need to store anything 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 array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

**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 $K$
  - Put each element in its proper bucket (a.k.a. bin)
  - *If* data is only integers, no need to store anything more than a *count* of how times that bucket has been used
- Output result via linear pass through array of buckets

| count array |
|---|---|
| 1 | 3 |
| 2 | 1 |
| 3 | 2 |
| 4 | 2 |
| 5 | 3 |

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 \( K \) 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
Bucket Sort with Data

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

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

• Bucket sort illustrates a more general trick
  – Imagine a heap for a small range of integer priorities
**Radix Sort**

- Radix = “the base of a number system”
  - Examples will use 10 because we are familiar with 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
  - After $k$ passes, the last $k$ digits are sorted

- Aside: Origins go back to the 1890 U.S. census
Example: Radix Sort: Pass #1

Bucket sort
by 1’s digit

Input data

<table>
<thead>
<tr>
<th>478</th>
<th>537</th>
<th>9</th>
<th>721</th>
<th>3</th>
<th>38</th>
<th>123</th>
<th>67</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>721</td>
<td>3</td>
<td>123</td>
<td>537</td>
<td>67</td>
</tr>
</tbody>
</table>

After 1st pass

<table>
<thead>
<tr>
<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>721</td>
<td>123</td>
<td></td>
<td>537</td>
<td>67</td>
<td>478</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This example uses B=10 and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.
Example: Radix Sort: Pass #2

After 1\textsuperscript{st} pass

<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>721</td>
<td>03</td>
<td>721</td>
<td>537</td>
<td>67</td>
<td>478</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>09</td>
<td>123</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>537</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>
<tr>
<td>478</td>
<td></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>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>

Bucket sort by 10's digit

After 2\textsuperscript{nd} pass

<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>721</td>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>537</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>537</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></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>
<tr>
<td>478</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Example: Radix Sort: Pass #3**

<table>
<thead>
<tr>
<th>After 2\textsuperscript{nd} pass</th>
<th>Bucket sort by 100’s digit</th>
<th>After 3\textsuperscript{rd} pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>721</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>38</td>
</tr>
<tr>
<td>123</td>
<td>003 123</td>
<td>67</td>
</tr>
<tr>
<td>537</td>
<td>009 478 537 721</td>
<td>123</td>
</tr>
<tr>
<td>38</td>
<td>038</td>
<td>478</td>
</tr>
<tr>
<td>67</td>
<td>067</td>
<td>537</td>
</tr>
<tr>
<td>478</td>
<td></td>
<td>721</td>
</tr>
</tbody>
</table>

Invariant: after k passes the low order k digits are sorted.
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
  – $15*(52 + n)$
  – This is less than $n \log n$ only if $n > 33,000$
  • Of course, cross-over point depends on constant factors of the implementations
Last Slide on Sorting

- Simple $O(n^2)$ sorts can be fastest for small $n$
  - selection sort, insertion sort (which is 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
- $\Omega (n \log n)$ worst and average bound for comparison sorting
- 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!