Today

• Sorting
  – Comparison sorting
  – Beyond comparison 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 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
  - \( a[0]<a[1]<a[2] \)
  - \( a[0]<a[2]<a[1] \)
  - \( a[1]<a[0]<a[2] \)
  - \( a[1]<a[2]<a[0] \)
  - \( a[2]<a[0]<a[1] \)
  - \( a[2]<a[1]<a[0] \)
- 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

Representing the Sort Problem

- Can represent this sorting process as a decision tree:
  - Nodes are sets of "remaining possibilities"
  - At root, anything is possible; no option eliminated
  - Edges represent comparisons made, and the node resulting from a comparison contains only consistent possibilities
    - Ex: Say we need to know whether \( a< b \) or \( b< a \); our root for \( n=2 \)
  - A comparison between \( a \& b \) will lead to a node that contains only one possibility (either \( a< b \) or \( b< a \))

Note: This tree 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
  - 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 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: Sorting \( a, b, c \)
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
- 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?
  \( L \leq 2^h \)

• 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:
  \( h \geq \log_2 N! \)

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)
- Bucket sort
- Radix sort
- External sorting

Comparison lower bound: \( \Omega(n \log n) \)
- Bucket sort (a.k.a. BinSort)

Specialized algorithms: \( O(1) \)

Handling huge data sets

Handling huge data sets

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, don’t even 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>

How???

- Change the model – assume more than `compare(a,b)`

2/02/2011
**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, don’t even 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</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>
```

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 number of elements, \( n \)
  - 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 in \( O(1) \) (say, keep a pointer to last element)

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

- Example: Movie ratings; scale 1-5: 1=bad, 5=excellent
  - Input: 5: Casablanca, 3: Harry Potter movies, 5: Star Wars Original Trilogy
  - Output: 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
  - After \( k \) passes, the last \( k \) digits are sorted
- Aside: Origins go back to the 1890 U.S. census

```
<table>
<thead>
<tr>
<th>Input:</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>478</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</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>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>721</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Example**

Radix = 10

```
<table>
<thead>
<tr>
<th>Input:</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>478</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</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>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>721</td>
<td></td>
<td></td>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Example**

Radix = 10

```
<table>
<thead>
<tr>
<th>Input:</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></td>
<td></td>
<td>3</td>
<td>143</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>67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>478</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38</td>
</tr>
</tbody>
</table>
```

**Example**

```
<table>
<thead>
<tr>
<th>Input:</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></td>
<td>3</td>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>537</td>
<td>38</td>
<td>67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>478</td>
<td>38</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Input:</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></td>
<td>3</td>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>537</td>
<td>38</td>
<td>67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>478</td>
<td>38</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Input:</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></td>
<td>3</td>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>537</td>
<td>38</td>
<td>67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>478</td>
<td>38</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Input:</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></td>
<td>3</td>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>537</td>
<td>38</td>
<td>67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>478</td>
<td>38</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
**Example**

Radix = 10

<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>3</td>
<td>9</td>
<td>721</td>
<td>537</td>
<td>143</td>
<td>67</td>
<td>478</td>
<td>724</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Order was: 39 721 537 143 67 478

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:

<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>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:

<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>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:

<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>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>

---

**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: \( B \)
  - Base 10 #: 10; binary #: 2; Alpha-numeric char: 62
- Number of passes = “Digits”: \( P \)
  - 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: \( 15*(52+n) \)
  - 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

---

**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
**Last word 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!