From Midterm Post-Mortem vs. Historical Average for this Course
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.
How fast can we sort?

- Heapsort, Mergesort, and Quicksort all run in $O(N \log N)$ best case running time.
- Can we do any better?
- No, if the basic action is a comparison.
Sorting Model

• Recall our basic assumption: we can only compare two elements at a time
  – we can only reduce the possible solution space by half each time we make a comparison

• Suppose you are given N elements
  – Assume no duplicates

• How many possible orderings can you get?
  – Example: a, b, c (N = 3)
Permutations

• How many possible orderings can you get?
  – Example: a, b, c \( (N = 3) \)
  – (a b c), (a c b), (b a c), (b c a), (c a b), (c b a)
  – 6 orderings = 3 \( \cdot 2 \cdot 1 = 3! \) (ie, “3 factorial”)
  – All the possible permutations of a set of 3 elements

• For N elements
  – N choices for the first position, (N-1) choices for the second position, …, (2) choices, 1 choice
  – \( N(N-1)(N-2) \cdots (2)(1) = N! \) possible orderings
The leaves contain all the possible orderings of $a$, $b$, and $c$. 

The orderings include:

- $a < b < c$
- $b < c < a$
- $c < a < b$
- $a < c < b$
- $b < a < c$
- $c < b < a$
Decision Trees

• A Decision Tree is a Binary Tree such that:
  – Each node = a set of orderings
    • ie, the remaining solution space
  – Each edge = 1 comparison
  – Each leaf = 1 unique ordering
  – How many leaves for N distinct elements?
    • N!, ie, a leaf for each possible ordering
• Only 1 leaf has the ordering that is the desired correctly sorted arrangement
Decision Tree Example

Possible orders:
- $a < b < c$
- $b < c < a$
- $c < a < b$
- $a < c < b$
- $b < a < c$
- $c < b < a$

Actual order:
- $a < b < c$
- $b < c < a$
- $c < a < b$
- $a < c < b$
- $b < a < c$
- $c < b < a$
Decision Trees and Sorting

• Every sorting algorithm corresponds to a decision tree
  – Finds correct leaf by choosing edges to follow
    • ie, by making comparisons
  – Each decision reduces the possible solution space by one half

• Run time is $\geq$ maximum no. of comparisons
  – maximum number of comparisons is the length of the longest path in the decision tree, i.e. the height of the tree
Lower bound on Height

• A binary tree of height \( h \) has at most how many leaves?

\[ L \leq 2^h \]

• The decision tree has how many leaves:

\[ L = N! \]

• A binary tree with \( L \) leaves has height at least:

\[ h \geq \log_2 L \]

• So the decision tree has height:

\[ h \geq \log_2 (N!) \]
\[ \log(N!) \text{ is } \Omega(N \log N) \]

\[
\log(N!) = \log(N \cdot (N - 1) \cdot (N - 2) \cdots (2) \cdot (1)) \\
= \log N + \log(N - 1) + \log(N - 2) + \cdots + \log 2 + \log 1 \\
\geq \log N + \log(N - 1) + \log(N - 2) + \cdots + \log \frac{N}{2} \\
\geq \frac{N}{2} \log \frac{N}{2} \\
\geq \frac{N}{2} (\log N - \log 2) = \frac{N}{2} \log N - \frac{N}{2} \\
= \Omega(N \log N) \]
\( \Omega(N \log N) \)

- Run time of any comparison-based sorting algorithm is \( \Omega(N \log N) \)
- Can we do better if we don’t use comparisons?
BucketSort (aka BinSort)

If all values to be sorted are *known* to be between 1 and $K$, create an array count of size $K$, *increment* counts while traversing the input, and finally output the result.

**Example**  
$K=5$. Input = (5,1,3,4,3,2,1,1,5,4,5)

<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>
BucketSort Complexity: $O(n+K)$

- Case 1: $K$ is a constant
  - BinSort is linear time
- Case 2: $K$ is variable
  - Not simply linear time
- Case 3: $K$ is constant but large (e.g. $2^{32}$)
  - ???
Fixing impracticality: RadixSort

• Radix = “The base of a number system”
  – We’ll use 10 for convenience, but could be anything

• **Idea**: BucketSort on each digit, least significant to most significant (lsd to msd)
Radix Sort Example (1\textsuperscript{st} pass)

Bucket sort by 1’s digit

Input data

| 478 | 537 | 9  | 721 | 3  | 38 | 123 | 67  |

| 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |

| 721 | 38 | 123 | 537 | 478 | 9  |

After 1\textsuperscript{st} pass

| 721 | 3  | 123 | 537 | 478 | 9  |

This example uses B=10 and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.
Radix Sort Example (2\textsuperscript{nd} pass)

After 1\textsuperscript{st} pass

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

Bucket sort by 10’s digit

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>03</td>
<td>09</td>
<td>721</td>
<td>123</td>
<td>537</td>
<td>38</td>
<td>67</td>
<td>478</td>
<td></td>
</tr>
</tbody>
</table>

After 2\textsuperscript{nd} pass

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 3  | 9  | 721| 123| 537| 38 | 67 | 478|    |
Radix Sort Example (3\textsuperscript{rd} pass)

<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></td>
<td>38</td>
</tr>
<tr>
<td>123</td>
<td></td>
<td>67</td>
</tr>
<tr>
<td>537</td>
<td></td>
<td>123</td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>478</td>
</tr>
<tr>
<td>67</td>
<td></td>
<td>537</td>
</tr>
<tr>
<td>478</td>
<td></td>
<td>721</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>003</td>
<td>009</td>
<td>123</td>
</tr>
<tr>
<td>038</td>
<td>067</td>
<td></td>
</tr>
<tr>
<td>067</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textbf{Invariant}: after k passes the low order k digits are sorted.
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:

<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>
Radixsort: Complexity

• How many passes?
• How much work per pass?
• Total time?
• Conclusion?

• In practice
  – RadixSort only good for large number of elements with relatively small values
  – Hard on the cache compared to MergeSort/QuickSort
Summary of sorting

• Sorting choices:
  – $O(N^2)$ – Bubblesort, Insertion Sort
  – $O(N \log N)$ average case running time:
    • Heapsort: In-place, not stable.
    • Mergesort: $O(N)$ extra space, stable.