Sorting Lower Bound
Radix Sort

CSE 373
Data Structures
Lecture 15
Reading

• Reading
  › Sections 7.8-7.11
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\cdot2\cdot1 = 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
Decision Tree

The leaves contain all the possible orderings of a, b, c
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 Trees and Sorting

- Every sorting algorithm corresponds to a decision tree
  - Finds correct leaf by choosing edges to follow
    - i.e., 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
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  
c < a < b  
a < c < b

a < c  
a > c

b < c  
b > c

b < c < a  
b < a < c

b < c < a  
c < b < a

b < c < a  
c < b < a

b < c < a  
b < a < c

b < c < a  
b < a < c
How many leaves on a tree?

• Suppose you have a binary tree of height $d$. How many leaves can the tree have?
  › $d = 1 \rightarrow$ at most 2 leaves,
  › $d = 2 \rightarrow$ at most 4 leaves, etc.
Lower bound on Height

- A binary tree of height $d$ has at most $2^d$ leaves
  - depth $d = 1 \rightarrow 2$ leaves, $d = 2 \rightarrow 4$ leaves, etc.
  - Can prove by induction
- Number of leaves, $L \leq 2^d$
- Height $d \geq \log_2 L$
- The decision tree has $N!$ leaves
- So the decision tree has height $d \geq \log_2(N!)$. 
\[ \log(N!) = \log(N \cdot (N-1) \cdot (N-2) \cdots (2) \cdot (1)) \]
\[ = \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \]
\[ \geq \log N + \log(N-1) + \log(N-2) + \cdots + \log \frac{N}{2} \]
\[ \geq \frac{N}{2} \log \frac{N}{2} \]
\[ \geq \frac{N}{2} (\log N - \log 2) = \frac{N}{2} \log N - \frac{N}{2} \]
\[ = \Omega(N \log N) \]
\( \Omega(N \log N) \)

- Run time of any comparison-based sorting algorithm is \( \Omega(N \log N) \)
- Can we do better if we don’t use comparisons?
Radix Sort: Sorting integers

- Historically goes back to the 1890 census.
- Radix sort = multi-pass bucket sort of integers in the range 0 to $B^P - 1$
- Bucket-sort from least significant to most significant “digit” (base B)
- Requires $P(B+N)$ operations where $P$ is the number of passes (the number of base $B$ digits in the largest possible input number).
- If $P$ and $B$ are constants then $O(N)$ time to sort!
Radix Sort Example

Input data

<table>
<thead>
<tr>
<th></th>
<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>1's digit</td>
<td>721</td>
<td>3</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>478</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Bucket sort by 1's 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></td>
<td>721</td>
<td>3</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After 1st pass

<table>
<thead>
<tr>
<th></th>
<th>721</th>
<th>3</th>
<th>123</th>
<th>537</th>
<th>67</th>
<th>478</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>721</td>
<td>3</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>478</td>
<td>9</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.
Radix Sort Example

<table>
<thead>
<tr>
<th>After 1\textsuperscript{st} pass</th>
<th>Bucket sort by 10’s digit</th>
<th>After 2\textsuperscript{nd} pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>721</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>123</td>
<td></td>
<td>721</td>
</tr>
<tr>
<td>537</td>
<td></td>
<td>123</td>
</tr>
<tr>
<td>67</td>
<td></td>
<td>537</td>
</tr>
<tr>
<td>478</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>67</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>478</td>
</tr>
</tbody>
</table>
Radix Sort Example

<table>
<thead>
<tr>
<th>After 2^{nd} pass</th>
<th>Bucket sort by 100's digit</th>
<th>After 3^{rd} pass</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<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>
</tbody>
</table>

<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>123</td>
<td></td>
<td></td>
<td>123</td>
<td></td>
<td>478</td>
<td>537</td>
<td></td>
<td>721</td>
<td></td>
<td></td>
</tr>
<tr>
<td>003</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>009</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>478</td>
<td></td>
</tr>
<tr>
<td>038</td>
<td></td>
<td></td>
<td>123</td>
<td></td>
<td>478</td>
<td>537</td>
<td></td>
<td>721</td>
<td></td>
<td></td>
</tr>
<tr>
<td>067</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>537</td>
<td></td>
</tr>
</tbody>
</table>

Invariant: after k passes the low order k digits are sorted.
Implementation Options

- **List**
  - List of data, bucket array of lists.
  - Concatenate lists for each pass.

- **Array / List**
  - Array of data, bucket array of lists.

- **Array / Array**
  - Array of data, array for all buckets.
  - Requires counting.
### Array / Array

<table>
<thead>
<tr>
<th>Data Array</th>
<th>Count Array</th>
<th>Address Array</th>
<th>Target Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 721</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1 3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2 123</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3 537</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4 67</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
<td>5 478</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>6</td>
<td>6 38</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>7</td>
<td>7 9</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bucket i ranges from add[i] to add[i+1]-1

add[0] := 0
add[i] := add[i-1] + count[i-1], i > 0
Array / Array

• Pass 1 (over A)
  › Calculate counts and addresses for 1st “digit”
• Pass 2 (over T)
  › Move data from A to T
  › Calculate counts and addresses for 2nd “digit”
• Pass 3 (over A)
  › Move data from T to A
  › Calculate counts and addresses for 3rd “digit”
• …
• In the end an additional copy may be needed.
Choosing Parameters for Radix Sort

- N number of integers – given
- m bit numbers - given
- B number of buckets
  - \( B = 2^r \) – calculations can be done by shifting.
  - N/B not too small, otherwise too many empty buckets.
  - \( P = m/r \) should be small.
- Example – 1 million 64 bit numbers. Choose \( B = 2^{16} = 65,536 \). 1 Million / B \( \approx \) 15 numbers per bucket. \( P = 64/16 = 4 \) passes.
Properties of Radix Sort

- Not in-place
  - needs lots of auxiliary storage.
- Stable
  - equal keys always end up in same bucket in the same order.
- Fast
  - $B = 2^r$ buckets on $m$ bit numbers
  - $O\left(\frac{m}{r} (n + 2^r)\right)$ time
Internal versus External Sorting

- So far assumed that accessing A[i] is fast – Array A is stored in internal memory (RAM)
  - Algorithms so far are good for internal sorting
- What if A is so large that it doesn’t fit in internal memory?
  - Data on disk or tape
  - Delay in accessing A[i] – e.g. need to spin disk and move head
Internal versus External Sorting

• Need sorting algorithms that minimize disk/tape access time
  › External sorting – Basic Idea:
    • Load chunk of data into RAM, 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)
    • Text gives some examples
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.