Logistics

- Survey on main web page!
- Homework 6 (due on Friday)
- Project 3, Project 3, Project 3.
- Reading: finish Weiss Chapter 7, start Chapter 9
Sorting: The Big Picture

Given *n* comparable elements in an array, sort them in an increasing (or decreasing) order.

- **Simple algorithms:** $O(n^2)$
  - Insertion sort
  - Selection sort
  - Bubble sort
  - Shell sort
  - ...
- **Fancier algorithms:** $O(n \log n)$
  - Heap sort
  - Merge sort
  - Quick sort
  - ...
- **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, 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 \( a, b, c, d, e, \ldots \)
- 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 = 3·2·1 = 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

\[
\frac{N \cdot (N-1) \cdot (N-2) \cdots (2) \cdot (1)}{N = N!} \\
\text{ grows fast.}
\]
Decision Tree

- **Binary tree**: node-set of possible orderings
- **edge**: 1-comparison

- **Nodes**:
  - **a < b < c**
  - **c < a < b**
  - **a < c < b**
  - **b < c**
  - **b > c**

- **Leaves**:
  - **a < b < c**
  - **a < c < b**
  - **b < c < a**
  - **b < a < c**
  - **c < b < a**

- **Orderings**:
  - **a < b < c**
  - **b < c < a**
  - **c < a < b**
  - **a < c < b**
  - **b < a < c**
  - **c < b < a**

The leaves contain all the possible orderings of a, b, c.
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:
  \[ \log_2 L \leq \log_2 2^h = h \]

- The decision tree has how many leaves:
  \[ L = N! \]

- So the decision tree has height:
  \[ h \geq \log_2 (N!) \text{ (worst case)} \]
\[ \log(N!) \text{ is } \Omega(N\log N) \]

\[ \log a + \log b = \log(ab) \]

\[
\begin{align*}
\log(N!) &= \log(N \cdot (N-1) \cdot (N-2) \cdots (2) \cdot (1)) \\
&= \log N + \log(N-1) + \log(N-2) + \cdots + \log 2 + \log 1 \\
&\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log\frac{N}{2} \\
&\geq \frac{N}{2} \log\frac{N}{2} \\
&\geq \frac{N}{2} (\log N - \log 2) = \frac{N}{2} \log N - \frac{N}{2} \\
&= \Omega(N \log N)
\end{align*}
\]
$\Omega(N \log N)$

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

$\text{Comparisons: } a < b$
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</th>
<th>array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3 * 2</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>* 2</td>
</tr>
</tbody>
</table>

Running time to sort $n$ items? $O(n + K)$
BucketSort Complexity: $O(n+K)$

\[ k=1000 \quad \sigma(n + 1000) \]

- 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. $2^{32}$)
  - ???

$O(n + 2^{32})$
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 (1st pass)

Input data

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

After 1st pass

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)

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

```
<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>03</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>478</td>
<td>38</td>
<td>67</td>
<td>478</td>
<td>3</td>
<td>9</td>
</tr>
</tbody>
</table>
```
Radix Sort Example (3rd pass)

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

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

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

**BucketSort on lsd:**

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

**BucketSort on next-higher digit:**

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

**BucketSort on msd:**

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
Radixsort: Complexity

- How many passes?
  \[ \rho = \log_2 (\text{max number}) \]
- How much work per pass?
  \[ O(n + k) \]
- Total time?
  \[ O(\rho (n + k)) \]
- Conclusion?
  \[ \rho \text{ is large} \]
- In practice
  - RadixSort only good for large number of elements with relatively small values
  - Hard on the cache compared to MergeSort/QuickSort
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
Graphs

Chapter 9 in Weiss
Graph... ADT?

- Not quite an ADT... operations not clear

- A formalism for representing relationships between objects

  Graph \( g = (V, E) \)
  - **Set of vertices:**
    \[ V = \{v_1, v_2, \ldots, v_n\} \]
  - **Set of edges:**
    \[ E = \{e_1, e_2, \ldots, e_m\} \]
    where each \( e_i \) connects two vertices \((v_{i1}, v_{i2})\)

\[ V = \{\text{Han, Leia, Luke}\} \]
\[ E = \{\text{(Luke, Leia), (Han, Leia), (Leia, Han)}\} \]
Graph Definitions

In *directed* graphs, edges have a specific direction:

In *undirected* graphs, they don’t (edges are two-way):

\( v \) is adjacent to \( u \) if \( (u, v) \in E \)
More Definitions: Simple Paths and Cycles

A *simple path* repeats no vertices (except that the first can be the last):

\[ p = \{\text{Seattle, Salt Lake City, San Francisco, Dallas}\} \]
\[ p = \{\text{Seattle, Salt Lake City, Dallas, San Francisco, Seattle}\} \]

A *cycle* is a path that starts and ends at the same node:

\[ p = \{\text{Seattle, Salt Lake City, Dallas, San Francisco, Seattle}\} \]
\[ p = \{\text{Seattle, Salt Lake City, Seattle, San Francisco, Seattle}\} \]

A *simple cycle* is a cycle that repeats no vertices except that the first vertex is also the last (in undirected graphs, no edge can be repeated)
Trees as Graphs

• Every tree is a graph!
• Not all graphs are trees!

A graph is a tree if
  – There are no cycles (directed or undirected)
  – There is a path from the root to every node
Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no (directed) cycles.

Aside: If program call graph is a DAG, then all procedure calls can be in-lined.
Graph Representations

0. List of vertices + list of edges
1. 2-D matrix of vertices (marking edges in the cells)  "adjacency matrix"
2. List of vertices each with a list of adjacent vertices  "adjacency list"

Things we might want to do:
• iterate over vertices
• iterate over edges
• iterate over vertices adj. to a vertex
• check whether an edge exists

Vertices and edges may be labeled
Representation 1: Adjacency Matrix

A $|V| \times |V|$ array in which an element $(u, v)$ is true if and only if there is an edge from $u$ to $v$.

Space requirements: 

Runtime: