Data Structures and Algorithms

CS245-2017S-12

Non-Comparison Sorts

David Galles

Department of Computer Science
University of San Francisco
Comparison Sorting

- Comparison sorts work by comparing elements
  - Can only compare 2 elements at a time
  - Check for $<$, $>$, $=$.
- All the sorts we have seen so far (Insertion, Quick, Merge, Heap, etc.) are comparison sorts
- If we know nothing about the list to be sorted, we need to use a comparison sort
Decision Trees

Insertion Sort on list \( \{a, b, c\} \)

\[
\begin{align*}
\text{a}<\text{b}<\text{c} & \quad \text{b}<\text{c}<\text{a} \\
\text{a}<\text{c}<\text{b} & \quad \text{c}<\text{a}<\text{b} \\
\text{b}<\text{a}<\text{c} & \quad \text{c}<\text{b}<\text{a}
\end{align*}
\]
12-2: Decision Trees

- Every comparison sorting algorithm has a decision tree
- What is the best-case number of comparisons for a comparison sorting algorithm, given the decision tree for the algorithm?
12-3: Decision Trees

- Every comparison sorting algorithm has a decision tree
- What is the best-case number of comparisons for a comparison sorting algorithm, given the decision tree for the algorithm?
  - (The depth of the shallowest leaf) + 1
- What is the worst case number of comparisons for a comparison sorting algorithm, given the decision tree for the algorithm?
Every comparison sorting algorithm has a decision tree

What is the best-case number of comparisons for a comparison sorting algorithm, given the decision tree for the algorithm?

- (The depth of the shallowest leaf) + 1

What is the worst case number of comparisons for a comparison sorting algorithm, given the decision tree for the algorithm?

- The height of the tree – (depth of the deepest leaf) + 1
12-5: Decision Trees

- What is the largest number of nodes for a tree of depth $d$?
12-6: Decision Trees

- What is the largest number of nodes for a tree of depth $d$?
  - $2^d$
- What is the minimum height, for a tree that has $n$ leaves?
12-7: Decision Trees

- What is the largest number of nodes for a tree of depth $d$?
  - $2^d$

- What is the minimum height, for a tree that has $n$ leaves?
  - $\log n$

- How many leaves are there in a decision tree for sorting $n$ elements?
12-8: Decision Trees

- What is the largest number of nodes for a tree of depth $d$?
  - $2^d$

- What is the minimum height, for a tree that has $n$ leaves?
  - $\lg n$

- How many leaves are there in a decision tree for sorting $n$ elements?
  - $n!$

- What is the minimum height, for a decision tree for sorting $n$ elements?
What is the largest number of nodes for a tree of depth $d$?
- $2^d$

What is the minimum height, for a tree that has $n$ leaves?
- $\log n$

How many leaves are there in a decision tree for sorting $n$ elements?
- $n!$

What is the minimum height, for a decision tree for sorting $n$ elements?
- $\log n!$
\(12-10: \lg(n!) \in \Omega(n \lg n)\)

\[
\begin{align*}
\lg(n!) &= \lg(n \times (n - 1) \times (n - 2) \times \ldots \times 2 \times 1) \\
&= (\lg n) + (\lg(n - 1)) + (\lg(n - 2)) + \ldots + (\lg 2) + (\lg 1) \\
&\geq (\lg n) + (\lg(n - 1)) + \ldots + (\lg(n/2)) \\
&\quad \text{(n/2 terms)} \\
&\geq (\lg(n/2) + (\lg(n/2)) + \ldots + \lg(n/2) \\
&\quad \text{(n/2 terms)} \\
&= (n/2) \lg(n/2) \\
&\in \Omega(n \lg n)
\end{align*}
\]
All comparison sorting algorithms can be represented by a decision tree with $n!$ leaves.

Worst-case number of comparisons required by a sorting algorithm represented by a decision tree is the height of the tree.

A decision tree with $n!$ leaves must have a height of at least $n \log n$.

All comparison sorting algorithms have worst-case running time $\Omega(n \log n)$. 
12-12: Counting Sort

- Sorting a list of $n$ integers
- We know all integers are in the range $0 \ldots m$
- We can potentially sort the integers faster than $n \log n$
- Keep track of a “Counter Array” $C$:
  - $C[i] =$ # of times value $i$ appears in the list

Example: $3 \ 1 \ 3 \ 5 \ 2 \ 1 \ 6 \ 7 \ 8 \ 1$

<table>
<thead>
<tr>
<th></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>1</td>
<td></td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The counter array $C$ is initialized with the counts of each integer in the list.
12-13: Counting Sort Example

3135216781

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<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>
### Counting Sort Example

The given array is:

```
1 3 5 2 1 6 7 8 1
```

The count array is initialized as:

```
0 0 0 1 0 0 0 0 0
```

After processing, the count array becomes:

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

The sorted array is:

```
0 0 0 1 0 0 0 0 0
```

Thus, the sorted array is:

```
0 0 0 1 0 0 0 0 0
```
### 12-15: Counting Sort Example

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>

35216781
Counting Sort Example

5216781

0 1 0 2 0 0 0 0 0

0 1 2 3 4 5 6 7 8 9
12-17: Counting Sort Example

216781

<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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
12-18: Counting Sort Example

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</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>
12-19: **Counting Sort Example**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>
### Counting Sort Example

- **Input Array:** 781
- **Count Array:**

```
<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>0</td>
<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>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
12-21: **Counting Sort Example**

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</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>

81
### Counting Sort Example

<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>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</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>
</table>
12-23: Counting Sort Example

<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>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<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>
### Counting Sort Example

<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>0</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

1 1 1 2 3 3 5 6 7 8
What is the running time of Counting Sort?

If the list has $n$ elements, all of which are in the range $0 \ldots m$:
What is the running time of Counting Sort?

If the list has $n$ elements, all of which are in the range $0 \ldots m$:
- Running time is $\Theta(n + m)$

What about the $\Omega(n \lg n)$ bound for all sorting algorithms?
What is the running time of Counting Sort?

If the list has $n$ elements, all of which are in the range $0 \ldots m$:

- Running time is $\Theta(n + m)$

What about the $\Omega(n \lg n)$ bound for all sorting algorithms?

- For *Comparison Sorts*, which allow for sorting arbitrary data. What happens when $m$ is very large?
• Counting Sort will need some modification to allow us to sort *records* with integer keys, instead of just integers.

• Binsort is much like Counting Sort, except that in each index $i$ of the counting array $C$:
  • Instead of storing the *number* of elements with the value $i$, we store a *list* of all elements with the value $i$. 
12-29: **Binsort Example**

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>2</th>
<th>6</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>3</th>
<th>9</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>mark</td>
<td>john</td>
<td>mary</td>
<td>sue</td>
<td>julie</td>
<td>rachel</td>
<td>pixel</td>
<td>shadow</td>
<td>alex</td>
<td>james</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>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>
12-30: Binsort Example

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>2</th>
<th>6</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>3</th>
<th>9</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>mark</td>
<td>john</td>
<td>mary</td>
<td>sue</td>
<td>julie</td>
<td>rachel</td>
<td>pixel</td>
<td>shadow</td>
<td>alex</td>
<td>james</td>
</tr>
</tbody>
</table>

Key data:

- 2: julie
- 3: shadow

Sort process:

1. john
2. mary
3. mark
4. rachel
5. pixel
6. sue
7. james
8. alex
9. /
12-31: Binsort Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>john</td>
<td>mary</td>
<td>julie</td>
<td>mark</td>
<td>shadow</td>
<td>rachel</td>
<td>pixel</td>
<td>sue</td>
<td>james</td>
<td>alex</td>
</tr>
</tbody>
</table>

key data

2 "julie"
3 "shadow"
1 "john"
2 "mary"
3 "mark"
4 "rachel"
5 "pixel"
6 "sue"
7 "james"
9 "alex"
12-32: **Bucket Sort**

- Expand the “bins” in Bin Sort to “buckets”
- Each bucket holds a range of key values, instead of a single key value
- Elements in each bucket are sorted.
**12-33: Bucket Sort Example**

<table>
<thead>
<tr>
<th>Key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>114</td>
<td>john</td>
</tr>
<tr>
<td>26</td>
<td>mary</td>
</tr>
<tr>
<td>50</td>
<td>julie</td>
</tr>
<tr>
<td>180</td>
<td>mark</td>
</tr>
<tr>
<td>44</td>
<td>shadow</td>
</tr>
<tr>
<td>111</td>
<td>rachel</td>
</tr>
<tr>
<td>4</td>
<td>pixel</td>
</tr>
<tr>
<td>95</td>
<td>sue</td>
</tr>
<tr>
<td>196</td>
<td>james</td>
</tr>
<tr>
<td>170</td>
<td>alex</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0-19</td>
</tr>
<tr>
<td>1</td>
<td>20-39</td>
</tr>
<tr>
<td>2</td>
<td>40-59</td>
</tr>
<tr>
<td>3</td>
<td>60-79</td>
</tr>
<tr>
<td>4</td>
<td>80-99</td>
</tr>
<tr>
<td>5</td>
<td>100-119</td>
</tr>
<tr>
<td>6</td>
<td>120-139</td>
</tr>
<tr>
<td>7</td>
<td>140-159</td>
</tr>
<tr>
<td>8</td>
<td>160-179</td>
</tr>
<tr>
<td>9</td>
<td>180-199</td>
</tr>
</tbody>
</table>
### Bucket Sort Example

<table>
<thead>
<tr>
<th></th>
<th>26</th>
<th>50</th>
<th>180</th>
<th>44</th>
<th>111</th>
<th>4</th>
<th>95</th>
<th>196</th>
<th>170</th>
</tr>
</thead>
<tbody>
<tr>
<td>key data</td>
<td>mary</td>
<td>julie</td>
<td>mark</td>
<td>shadow</td>
<td>rachel</td>
<td>pixel</td>
<td>sue</td>
<td>james</td>
<td>alex</td>
</tr>
</tbody>
</table>

The key data are distributed across 10 buckets, each representing a specific range of values:

- **0-19**
- **20-39**
- **40-59**
- **60-79**
- **80-99**
- **100-119**
- **120-139**
- **140-159**
- **160-179**
- **180-199**

The number 114 falls into the fifth bucket, corresponding to the range **80-99**.
# Bucket Sort Example

<table>
<thead>
<tr>
<th>Key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>julie</td>
</tr>
<tr>
<td>180</td>
<td>mark</td>
</tr>
<tr>
<td>44</td>
<td>shadow</td>
</tr>
<tr>
<td>111</td>
<td>rachel</td>
</tr>
<tr>
<td>4</td>
<td>pixel</td>
</tr>
<tr>
<td>95</td>
<td>sue</td>
</tr>
<tr>
<td>196</td>
<td>james</td>
</tr>
<tr>
<td>170</td>
<td>alex</td>
</tr>
</tbody>
</table>

The key and data are sorted into buckets according to their values.
12-36: Bucket Sort Example
12-37: Bucket Sort Example

<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>

- 0-19
- 20-39
- 40-59
- 60-79
- 80-99
- 100-119
- 120-139
- 140-159
- 160-179
- 180-199

**Data**

- 44: shadow
- 111: rachel
- 4: pixel
- 95: sue
- 196: james
- 170: alex

**Keys**

- 26: mary
- 50: julie
- 114: john
- 180: mark
Bucket Sort Example

key
rachel 4 95 196 170
data
pixel sue james alex

0-19 40-59 80-99 120-139 160-179
20-39 60-79 100-119 140-159 180-199

50
julie

26
mary

44
shadow

114
john

180
mark
12-39: Bucket Sort Example

Bucket Sort Example

key
data

0 1 2 3 4 5 6 7 8 9


pixel 95 196 170
sue james alex

50 julie

26 44
mary shadow

114 john

111 rachel

180 mark

Rachel
12-40: Bucket Sort Example

```
<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>pixel</td>
<td>mary</td>
<td>shadow</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>sue</td>
<td>james</td>
<td>alex</td>
<td>95</td>
<td>196</td>
<td>170</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Key

- Pixel: 4
- Mary: 26
- Shadow: 44
- Julie: 50
- John: 114
- Rachel: 111
- Mark: 180
12-41: Bucket Sort Example

<table>
<thead>
<tr>
<th>Key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>196</td>
<td>james</td>
</tr>
<tr>
<td>170</td>
<td>alex</td>
</tr>
<tr>
<td>114</td>
<td>john</td>
</tr>
<tr>
<td>95</td>
<td>sue</td>
</tr>
<tr>
<td>111</td>
<td>rachel</td>
</tr>
<tr>
<td>50</td>
<td>julie</td>
</tr>
<tr>
<td>44</td>
<td>shadow</td>
</tr>
<tr>
<td>26</td>
<td>mary</td>
</tr>
<tr>
<td>4</td>
<td>pixel</td>
</tr>
<tr>
<td>180</td>
<td>mark</td>
</tr>
</tbody>
</table>

Key:
- 0-19
- 20-39
- 40-59
- 60-79
- 80-99
- 100-119
- 120-139
- 140-159
- 160-179
- 180-199
12-43: Bucket Sort Example

key
data


0  1  2  3  4  5  6  7  8  9

pixel  mary  shadow  50  julie  95  sue  114  john  111  rachel  170  alex  180  mark  196  james
12-44: Bucket Sort Example

Key:
- 0-19: pixel
- 20-39: mary
- 40-59: shadow
- 60-79: julie
- 80-99: sue
- 100-119: rachel
- 120-139: john
- 140-159: alex
- 160-179: mark
- 180-199: james

Data:
- 4: pixel
- 26: mary
- 44: shadow
- 50: julie
- 95: sue
- 111: rachel
- 114: john
- 170: alex
- 180: mark
- 196: james
We’re going to look at counting sort again.

For the moment, we will assume that our array is indexed from $1 \ldots n$ (where $n$ is the number of elements in the list) instead of being indexed from $0 \ldots n - 1$, to make the algorithm easier to understand.

Later, we will go back and change the algorithm to allow for an index between $0 \ldots n - 1$. 
Counting Sort Revisited

- Create the array $C[]$, such that $C[i] = \# \text{ of times key } i \text{ appears in the array.}$
- Modify $C[]$ such that $C[i] = \text{ the index of key } i \text{ in the sorted array. (assume no duplicate keys, for now)}$
- If $x \notin A$, we don’t care about $C[x]$
12-47: Counting Sort Revisited

- Create the array $C[]$, such that $C[i] =$ # of times key $i$ appears in the array.
- Modify $C[]$ such that $C[i] =$ the index of key $i$ in the sorted array. (assume no duplicate keys, for now)
- If $x \notin A$, we don’t care about $C[x]$

```
for(i=1; i<C.length; i++)
    C[i] = C[i] + C[i-1];
```

- Example: 3 1 2 4 9 8 7
Once we have a modified $C$, such that $C[i] =$ index of key $i$ in the array, how can we use $C$ to sort the array?
Once we have a modified $C$, such that $C[i]$ = index of key $i$ in the array, how can we use $C$ to sort the array?

```java
for (i=1; i <= n; i++)
    B[C[A[i].key()]] = A[i];
for (i=1; i <= n; i++)
    A[i] = B[i];
```

Example: 3 1 2 4 9 8 7
If a list has duplicate elements, and we create $C$ as before:

```java
for(i=1; i <= n; i++)
    C[A[i].key()]++;
for(i=1; i < C.length; i++)
    C[i] = C[i] + C[i-1];
```

What will the value of $C[i]$ represent?
12-51: Counting Sort & Duplicates

- If a list has duplicate elements, and we create \( C \) as before:

```plaintext
for(i=1; i <= n; i++)
    C[A[i].key()]++;
for(i=1; i < C.length; i++)
    C[i] = C[i] + C[i-1];
```

What will the value of \( C[i] \) represent?

- The *last* index in \( A \) where element \( i \) could appear.
for(i=1; i <= n; i++)
    C[A[i].key()]++;
for(i=1; i < C.length; i++)
    C[i] = C[i] + C[i-1];

for (i=1; i <= n; i++) {
    B[C[A[i].key()]]) = A[i];
    C[A[i].key()]--;
}
for (i=1; i <= n; i++)
    A[i] = B[i];

- Example: 3 1 2 4 2 2 9 1 6
for(i=1; i <= n; i++)
    C[A[i].key()]++;
for(i=1; i<C.length; i++)
    C[i] = C[i] + C[i-1];

for (i=1; i <= n; i++) {
    B[C[A[i].key()]][i] = A[i];
    C[A[i].key()]--;
}
for (i=1; i <= n; i++)
    A[i] = B[i];

- Example: 3 1 2 4 2 2 9 1 6
- Is this a Stable sorting algorithm?
for(i=1; i <= n; i++)
    C[A[i].key()]++;
for(i=1; i < C.length; i++)
    C[i] = C[i] + C[i-1];

for (i = n; i>=1; i++) {
    B[C[A[i].key()]] = A[i];
    C[A[i].key()]--;
}

for (i=1; i < n; i++)
    A[i] = B[i];

• How would we change this algorithm if our arrays were indexed from 0...n – 1 instead of 1...n?
for (i = 0; i < A.length; i++)
    C[A[i].key()]++;  // Counting
for (i = 1; i < C.length; i++)
    C[i] = C[i] + C[i-1];

for (i = A.length - 1; i >= 0; i++) {
    C[A[i].key()]--;  // Decrementing
    B[C[A[i].key()]] = A[i];
}

for (i = 0; i < A.length; i++)
    A[i] = B[i];
Radix Sort

- Sort a list of numbers one digit at a time
  - Sort by 1st digit, then 2nd digit, etc
- Each sort can be done in linear time, using counting sort

- First Try: Sort by most significant digit, then the next most significant digit, and so on
  - Need to keep track of a lot of sublists
12-57: Radix Sort

Second Try:

- Sort by least significant digit first
- Then sort by next-least significant digit, using a Stable sort
  
- Sort by most significant digit, using a Stable sort

At the end, the list will be completely sorted. Why?
Radix Sort

- If (most significant digit of $x$) < (most significant digit of $y$),
  then $x$ will appear in $A$ before $y$. 
If (most significant digit of $x$) <
(most significant digit of $y$),
then $x$ will appear in $A$ before $y$.

• Last sort was by the most significant digit
Radix Sort

- If (most significant digit of $x$) < (most significant digit of $y$),
  then $x$ will appear in $A$ before $y$.
- Last sort was by the most significant digit
- If (most significant digit of $x$) = (most significant digit of $y$) and
  (second most significant digit of $x$) < (second most significant digit of $y$),
  then $x$ will appear in $A$ before $y$.  

Radix Sort

- If (most significant digit of \( x \)) < (most significant digit of \( y \)), then \( x \) will appear in \( A \) before \( y \).
  - Last sort was by the most significant digit
- If (most significant digit of \( x \)) = (most significant digit of \( y \)) and (second most significant digit of \( x \)) < (second most significant digit of \( y \)), then \( x \) will appear in \( A \) before \( y \).
  - After next-to-last sort, \( x \) is before \( y \). Last sort does not change relative order of \( x \) and \( y \).
### Radix Sort

**Original List**

| 982 | 414 | 357 | 495 | 500 | 904 | 645 | 777 | 716 | 637 | 149 | 913 | 817 | 493 | 730 | 331 | 201 |

**Sorted by Least Significant Digit**

| 500 | 730 | 331 | 201 | 982 | 493 | 913 | 414 | 904 | 645 | 495 | 716 | 357 | 777 | 637 | 817 | 149 |

**Sorted by Second Least Significant Digit**

| 500 | 201 | 904 | 913 | 414 | 716 | 817 | 730 | 331 | 637 | 645 | 149 | 357 | 777 | 982 | 493 | 495 |

**Sorted by Most Significant Digit**

| 149 | 201 | 331 | 357 | 414 | 493 | 495 | 500 | 637 | 645 | 716 | 730 | 777 | 817 | 904 | 913 | 982 |
Radix Sort

- We do not need to use a single digit of the key for each of our counting sorts
  - We could use 2-digit chunks of the key instead
  - Our $C$ array for each counting sort would have 100 elements instead of 10
12-64: Radix Sort

<table>
<thead>
<tr>
<th>Original List</th>
<th>Sorted by Least Significant Base-100 Digit (last 2 base-10 digits)</th>
<th>Sorted by Most Significant Base-100 Digit (first 2 base-10 digits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9823, 4376, 2493, 1055, 8502, 4333, 1673, 8442, 8035, 6061, 7004, 3312, 4409, 2338</td>
<td>8502, 7004, 4409, 3312, 9823, 4333, 8035, 2338, 8442, 1055, 6061, 1673, 4376, 2493</td>
<td>1055, 1673, 2338, 2493, 3312, 4333, 4376, 4409, 6061, 7004, 8035, 8442, 8502, 9823</td>
</tr>
</tbody>
</table>
12-65: Radix Sort

- “Digit” does not need to be base ten
- For any value \( r \):
  - Sort the list based on \((key \mod r)\)
  - Sort the list based on \(((key / r) \mod r)\)
  - Sort the list based on \(((key / r^2) \mod r)\)
  - Sort the list based on \(((key / r^3) \mod r)\)
    ...
  - Sort the list based on
    \(((key / r^{\log_k(\text{largest value in array})}) \mod r))\)
- Code on other screen