Data Structures and Algorithms
More Sorting

Chris Brooks

Department of Computer Science
University of San Francisco
Base Case:
△ A list of length 1 or length 0 is already sorted

Recursive Case:
△ Split the list in half
△ Recursively sort two halves
△ Merge sorted halves together
14-1: Quick Sort

6 Pick a pivot element
6 Reorder the list:
   △ All elements < pivot
   △ Pivot element
   △ All elements > pivot
6 Recursively sort elements < pivot
6 Recursively sort elements > pivot
14-2: Quick Sort with constant memory

Can we avoid making a duplicate of the list we’re sorting?

6 Swap pivot element out of the way (we’ll swap it back later)

6 Maintain two pointers, \( i \) and \( j \)
  \( i \) points to the beginning of the list
  \( j \) points to the end of the list

6 Move \( i \) and \( j \) in to the middle of the list – ensuring that all elements to the left of \( i \) are \(<\) the pivot, and all elements to the right of \( j \) are greater than the pivot

6 Swap pivot element back to middle of list
14-3: Quick Sort - Partitioning

Pseudocode:

1. Pick a pivot index
2. Swap $A[pivotIndex]$ and $A[high]$
3. Set $i \leftarrow low, j \leftarrow high - 1$
4. while $(i \leq j)$
   - swap $A[i]$ and $A[j]$
   - increment $i$, decrement $j$
5. swap $A[i]$ and $A[pivot]$

Department of Computer Science — University of San Francisco – p.5/69
Coming up with a recurrence relation for quicksort is harder than mergesort.

How the problem is divided depends upon the data.

- Break list into:
  - size 0, size \( n - 1 \)
  - size 1, size \( n - 2 \)
  - \( \ldots \)
  - size \( \lfloor (n - 1)/2 \rfloor \), size \( \lceil (n - 1)/2 \rceil \)
  - \( \ldots \)
  - size \( n - 2 \), size 1
  - size \( n - 1 \), size 0
Worst case performance occurs when break list into size $n - 1$ and size 0

$$T(0) = c_1$$

for some constant $c_1$

$$T(1) = c_2$$

for some constant $c_2$

$$T(n) = nc_3 + T(n - 1) + T(0)$$

for some constant $c_3$

$$T(n) = nc_3 + T(n - 1) + T(0)$$

$$= T(n - 1) + nc_3 + c_2$$
Worst case: \( T(n) = T(n - 1) + nc_3 + c_2 \)

\[
\begin{align*}
T(n) &= T(n - 1) + nc_3 + c_2 \\
\end{align*}
\]
Worst case: \( T(n) = T(n - 1) + nc_3 + c_2 \)

\[
\begin{align*}
T(n) &= T(n - 1) + nc_3 + c_2 \\
&= [T(n - 2) + (n - 1)c_3 + c_2] + nc_3 + c_2 \\
&= T(n - 2) + (n + (n - 1))c_3 + 2c_2
\end{align*}
\]

14-7: \( \Theta() \) for Quick Sort
Worst case: \( T(n) = T(n - 1) + nc_3 + c_2 \)

\[
T(n) \\
= T(n - 1) + nc_3 + c_2 \\
= [T(n - 2) + (n - 1)c_3 + c_2] + nc_3 + c_2 \\
= T(n - 2) + (n + (n - 1))c_3 + 2c_2 \\
= [T(n - 3) + (n - 2)c_3 + c_2] + (n + (n - 1))c_3 + 2c_2 \\
= T(n - 3) + (n + (n - 1) + (n - 2))c_3 + 3c_2
\]
Worst case: \( T(n) = T(n - 1) + nc_3 + c_2 \)

\[
T(n) \\
= T(n - 1) + nc_3 + c_2 \\
= [T(n - 2) + (n - 1)c_3 + c_2] + nc_3 + c_2 \\
= T(n - 2) + (n + (n - 1))c_3 + 2c_2 \\
= [T(n - 3) + (n - 2)c_3 + c_2] + (n + (n - 1))c_3 + 2c_2 \\
= T(n - 3) + (n + (n - 1) + (n - 2))c_3 + 3c_2 \\
= T(n - 4) + (n + (n - 1) + (n - 2) + (n - 3))c_3 + 4c_2
\]
Worst case: \( T(n) = T(n - 1) + nc_3 + c_2 \)

\[

t(n) \\
= T(n - 1) + nc_3 + c_2 \\
= [T(n - 2) + (n - 1)c_3 + c_2] + nc_3 + c_2 \\
= T(n - 2) + (n + (n - 1))c_3 + 2c_2 \\
= [T(n - 3) + (n - 2)c_3 + c_2] + (n + (n - 1))c_3 + 2c_2 \\
= T(n - 3) + (n + (n - 1) + (n - 2))c_3 + 3c_2 \\
= T(n - 4) + (n + (n - 1) + (n - 2) + (n - 3))c_3 + 4c_2 \\
\ldots \\
= T(n - k) + \left( \sum_{i=0}^{k-1} (n - i)c_3 \right) + kc_2
\]
Worst case:

\[
T(n) = T(n - k) + \left( \sum_{i=0}^{k-1} (n - i)c_3 \right) + kc_2
\]

Set \( k = n \):

\[
T(n) = T(n - k) + \left( \sum_{i=0}^{k-1} (n - i)c_3 \right) + kc_2
= T(n - n) + \left( \sum_{i=0}^{n-1} (n - i)c_3 \right) + kc_2
= T(0) + \left( \sum_{i=0}^{n-1} ic_3 \right) + kc_2
= T(0) + c_3n(n + 1)/2 + kc_2
\in \Theta(n^2)
\]
Best case performance occurs when break list into size 
\[ \lceil (n - 1)/2 \rceil \] and size \[ \lceil (n - 1)/2 \rceil \]

\[ T(0) = c_1 \] for some constant \( c_1 \)
\[ T(1) = c_2 \] for some constant \( c_2 \)
\[ T(n) = nc_3 + 2T(n/2) \] for some constant \( c_3 \)

This is the same as Merge Sort: \( \Theta(n \log n) \)
If Quicksort is $\Theta(n^2)$ on some lists, why is it called *quick*?

1. Most lists give running time of $\Theta(n \lg n)$
   - Average case running time is $\Theta(n \lg n)$

2. Constants are very small
   - Constants don’t matter when complexity is different
   - Constants *do* matter when complexity is the same

What lists will cause Quick Sort to have $\Theta(n^2)$ performance?
Quick Sort has worst-case performance when:

- The list is sorted (or almost sorted)
- The list is inverse sorted (or almost inverse sorted)

Many lists we want to sort are almost sorted!

How can we fix Quick Sort?
14-15: Better Partitions

- Pick the middle element as the pivot
  - Sorted and reverse sorted lists give good performance

- Pick a random element as the pivot
  - No single list always gives bad performance

- Pick the median of 3 elements
  - First, Middle, Last
  - 3 Random Elements
Improving Quick Sort

- Insertion Sort runs faster than Quick Sort on small lists
  - Why?

- We can combine Quick Sort & Insertion Sort
  - When lists get small, run Insertion Sort instead of a recursive call to Quick Sort
  - When lists get small, stop! After call to Quick Sort, list will be almost sorted – finish the job with a single call to Insertion Sort
14-17: Heap Sort

- Build a heap out of the data
- Repeat:
  - Remove the largest element from the list, place it at end of heap
- Until all elements have been removed from the heap
- The list is now sorted

Example: 3 1 7 2 5 4
Building the heap takes time $\Theta(n)$

Each of the $n$ RemoveMax calls takes time $O(\lg n)$

Total time: $(n \lg n)$ (also $\Theta(n \lg n)$)
Comparison sorts work by comparing elements

- Can only compare 2 elements at a time
- Check for $<$, $>$, $\equiv$.

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.

All comparison sorting algorithms have worst-case running time $\Omega(n \lg n)$.  

14-20: 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
14-21: Counting Sort Example

3135216781

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</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>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>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 14-22: Counting Sort Example

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

135216781
14-23: Counting Sort Example

35216781

<table>
<thead>
<tr>
<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>
<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>
<td></td>
</tr>
</tbody>
</table>
### 14-24: Counting Sort Example

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

5216781
### 14-25: Counting Sort Example

#### Input Array:

216781

#### Counting Array:

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

#### Output Array:

<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>
<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>
<td></td>
</tr>
</tbody>
</table>
14-26: Counting Sort Example

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

```
0 1 1 2 0 1 0 0 0
0 1 2 3 4 5 6 7 8 9
```
14-27: Counting Sort Example

6781

0  2  1  2  0  1  0  0  0  0  0  0
0  1  2  3  4  5  6  7  8  9
14-28: 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>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>
14-29: 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>
14-30: Counting Sort Example
14-31: Counting Sort Example

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</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>
<td></td>
</tr>
</tbody>
</table>
14-32: Counting Sort Example

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

11112335678
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 its 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 $\Omega(n \log n)$ 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 $i$. 
**Binsort Example**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<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>
</tr>
</tbody>
</table>

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</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>
</tr>
</tbody>
</table>

keydata
data
14-37: Binsort Example

keydata

mark | john | mary | sue | julie | rachel | pixel | shadow | alex | james

Department of Computer Science — University of San Francisco – p.39/69
14-38: Binsort Example

keydata: john mary julie mark shadow rachel pixel sue james alex

data: 1 2 3 4 5 6 7 8 9

Binsort process:
1. Divide the data into bins based on the keys.
2. Sort the keys within each bin.
3. Combine the sorted bins to form the final sorted list.

Diagram shows the process of sorting keys into bins and then combining them.
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.
## 14-40: Bucket Sort Example

<table>
<thead>
<tr>
<th>114</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>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>

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

---

Department of Computer Science — University of San Francisco – p.42/69
**14-41: Bucket Sort Example**

<table>
<thead>
<tr>
<th>Key</th>
<th>Data</th>
<th>Key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>mary</td>
<td>50</td>
<td>julie</td>
</tr>
<tr>
<td>180</td>
<td>mark</td>
<td>44</td>
<td>shadow</td>
</tr>
<tr>
<td>111</td>
<td>rachel</td>
<td>4</td>
<td>pixel</td>
</tr>
<tr>
<td>95</td>
<td>sue</td>
<td>196</td>
<td>james</td>
</tr>
<tr>
<td>170</td>
<td>alex</td>
<td>111</td>
<td>114</td>
</tr>
</tbody>
</table>
14-42: 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>

Key: 26
Data: mary

Key: 114
Data: john

0-19 / 40-59 / 80-99 / 120-139 / 160-179
20-39 / 60-79 / 100-119 / 140-159 / 180-199
14-43: Bucket Sort Example

Here is an example of bucket sort with a key and data set:

<table>
<thead>
<tr>
<th>Key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-19</td>
<td>50</td>
</tr>
<tr>
<td>20-39</td>
<td>26</td>
</tr>
<tr>
<td>40-59</td>
<td>114</td>
</tr>
<tr>
<td>60-79</td>
<td>111</td>
</tr>
<tr>
<td>80-99</td>
<td>44</td>
</tr>
<tr>
<td>100-119</td>
<td>180</td>
</tr>
<tr>
<td>120-139</td>
<td>44</td>
</tr>
<tr>
<td>140-159</td>
<td>111</td>
</tr>
<tr>
<td>160-179</td>
<td>95</td>
</tr>
<tr>
<td>180-199</td>
<td>196</td>
</tr>
</tbody>
</table>

In this example, the keys are numbers, and the data associated with each key is a name. The numbers are placed into buckets based on their value range, and the names are then sorted within each bucket.
# 14-44: Bucket Sort Example

## Bucket Sort Example

Here is an example of bucket sort with the given data:

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Key Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-19</td>
<td>26 mary</td>
</tr>
<tr>
<td>20-39</td>
<td></td>
</tr>
<tr>
<td>40-59</td>
<td>50 julie</td>
</tr>
<tr>
<td>60-79</td>
<td></td>
</tr>
<tr>
<td>80-99</td>
<td>114 john</td>
</tr>
<tr>
<td>100-119</td>
<td></td>
</tr>
<tr>
<td>120-139</td>
<td></td>
</tr>
<tr>
<td>140-159</td>
<td></td>
</tr>
<tr>
<td>160-179</td>
<td></td>
</tr>
<tr>
<td>180-199</td>
<td></td>
</tr>
</tbody>
</table>

The sorted data is as follows:

- 26 mary
- 50 julie
- 114 john
- 180 mark

This demonstrates how bucket sort can be used to sort a list of numbers by distributing them into buckets corresponding to their ranges and then sorting the buckets individually.
14-45: Bucket Sort Example

Key data

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-19</td>
<td>26</td>
<td>Mary</td>
</tr>
<tr>
<td>20-39</td>
<td>44</td>
<td>Shadow</td>
</tr>
<tr>
<td>40-59</td>
<td>114</td>
<td>John</td>
</tr>
<tr>
<td>60-79</td>
<td>111</td>
<td>Rachel</td>
</tr>
<tr>
<td>80-99</td>
<td>4</td>
<td>Pixel</td>
</tr>
<tr>
<td>100-119</td>
<td>95</td>
<td>Sue</td>
</tr>
<tr>
<td>120-139</td>
<td>196</td>
<td>James</td>
</tr>
<tr>
<td>140-159</td>
<td>170</td>
<td>Alex</td>
</tr>
</tbody>
</table>

Department of Computer Science — University of San Francisco — p.47/69
14-46: Bucket Sort Example

Key: pixel, sue, james, alex

Data: 4, 95, 196, 170

Departments of Computer Science — University of San Francisco — p. 48/69
14-47: Bucket Sort Example

Bucket Sort Example
14-48: Bucket Sort Example

key

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

james  alex

julie

mary

shadow

sue

rachel

mark

pixel

26

44

95

111

180

Department of Computer Science — University of San Francisco – p.50/69
Bucket Sort Example

0-19  40-59  80-99  120-139  160-179
20-39  60-79  100-119  140-159  180-199
14-50: Bucket Sort Example

Bucket Sort Example

key data


0   1   2   3   4   5   6   7   8   9

4  pixel  26  mary  44  shadow  95  sue  111  rachel  170  alex  196  james

50  julie

114  john

180  mark

Department of Computer Science — University of San Francisco – p.52/69
### 14-51: Bucket Sort Example

<table>
<thead>
<tr>
<th>Key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>44</td>
<td>8</td>
</tr>
<tr>
<td>50</td>
<td>11</td>
</tr>
<tr>
<td>95</td>
<td>114</td>
</tr>
<tr>
<td>111</td>
<td>170</td>
</tr>
<tr>
<td>114</td>
<td>180</td>
</tr>
<tr>
<td>196</td>
<td></td>
</tr>
</tbody>
</table>

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

- **Bucket Sort Example**
  - Bucket 0: pixel
  - Bucket 2: mary
  - Bucket 4: shadow
  - Bucket 4: julie
  - Bucket 5: sue
  - Bucket 11: rachel
  - Bucket 11: john
  - Bucket 12: alex
  - Bucket 14: mark
  - Bucket 19: james
  - Bucket 19: mark
  - Bucket 12: alex
  - Bucket 11: rachel
  - Bucket 4: julie
  - Bucket 2: mary
  - Bucket 1: pixel
Create the array $C[\cdot]$, such that $C[i] = \# \text{ of times key } i$ appears in the array.

Modify $C[\cdot]$ such that $C[i] = \text{ the index of key } i$ in the sorted array. (assume no duplicate keys, for now)

If $x \not\in A$, we don’t care about $C[x]$
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 \( x \not\in A \), we don’t care about \( C[x] \)

\[
\text{for}(i=1; \ i<C\.\text{length}; \ i++) \\
\quad C[i] = C[i] + C[i-1];
\]

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=0; i<A.length; i++)
    B[C[A[i].key()]] = A[i];
for (i=0; i<A.length; 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 = 0; i < A.length; 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?
If a list has duplicate elements, and we create $C$ as before:

```java
for (i=0; i<A.length; 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=0; i<A.length; i++)
    C[A[i].key()]++; 
for (i=1; i<C.length; i++)
    C[i] = C[i] + C[i-1];

for (i=0; i<A.length; i++) {
    B[C[A[i].key()]] = A[i];
    C[A[i].key()]--; 
}

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

Example: 3 1 2 4 2 2 9 1 6
for (i = 0; i < A.length; i++)
    C[A[i].key()]++;
for (i = 1; i < C.length; i++)
    C[i] = C[i] + C[i-1];

for (i = 0; i < A.length; i++) {
    B[C[A[i].key()]] = A[i];
    C[A[i].key()]--;
}
for (i = 0; i < A.length; i++)
    A[i] = B[i];

Example: 3 1 2 4 2 2 9 1 6

Is this a Stable sorting algorithm?
for (i=0; i<A.length; i++)
    C[A[i].key()]+=;
for (i=1; i<C.length; i++)
    C[i] = C[i] + C[i-1];

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

for (i=0; i<A.length; i++)
    A[i] = B[i];
14-61: Radix Sort

1. Sort a list of numbers one digit at a time
   - Sort by 1st digit, then 2nd digit, etc

2. Each sort can be done in linear time, using counting sort

3. 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
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?
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
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 $y$. 
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 $y$.
“Digit” does not need to be base ten

For any value $r$:
- Sort the list based on $(\text{key} \mod r)$
- Sort the list based on $((\text{key} / r) \mod r)$
- Sort the list based on $((\text{key} / r^2) \mod r)$
- Sort the list based on $((\text{key} / r^3) \mod r)$
  
  …
- Sort the list based on $((\text{key} / r^{\log_k(\text{largest value in array})}) \mod r)$

Code on other screen