Linear Sorts

- Linear Sorts
- Bucket Sort
- Radix Sort
- Counting Sort
Linear Sorts

• **Linear sorts** are sort algorithms that run in linear time.
  - Require *special limitations*
  - Not useful as general-purpose sorts.

• Contrast: **comparison sorts**
  - General-purpose (sort any dataset)
  - $O(n \ lg \ n)$ worst-case time (proven!)
Linear Sorts

• **Linear sorts** are sort algorithms that run in linear time.
  
  - Require *special limitations*
  - Not useful as general-purpose sorts.

We're not saying that there can't be $O(n)$ special cases. Every algorithm can have those.

But no comparison sort has a **worst case** which is better than $O(n \lg n)$. 
Linear Sorts

- **Linear sorts** are sort algorithms that run in linear time.
  - Require **special limitations**
  - Not useful as general-purpose sorts.

So no comparison sort is asymptotically faster than Merge Sort / Quicksort* / Heapsort!

* Quicksort average time, of course...
Linear Sorts

- Linear Sorts
- Bucket Sort
- Radix Sort
- Counting Sort
What is Bucket Sort?

- Assume that we have a **small number** of key values
  - Or a small number of categories
- Allocate a bucket for each possible key value
- Examine each key, put it in the proper bucket
- Join buckets together to get output
Bucket Sort

Input:

| A_1 | A_0 | A_{37} | B_1 | B_2 | A_{23} |

In this example, we'll assume that our keys can only have two different values.

However, each key is associated with some satellite data, which we'll represent with a subscript.
Bucket Sort

Input:

| A₁ | A₀ | A₃₇ | B₁ | B₂ | A₂₃ |

Buckets:

We have exactly two buckets because we know that there are exactly two possible keys.

(Normally, the buckets would be the same size...but my slide space is limited!)
Bucket Sort

Input:

A_1  A_0  A_{37}  B_1  B_2  A_{23}

Buckets:

A_1  A_0  A_{37}  A_{23}
B_1  B_2

We iterate through the data, and partition the keys into the buckets.

Note that we maintained the relative order of the elements.

Bucket Sort is a stable sort.
Bucket Sort

**Input:**

- $A_1$
- $A_0$
- $A_{37}$
- $B_1$
- $B_2$
- $A_{23}$

**Buckets:**

- $A_1$
- $A_0$
- $A_{37}$
- $A_{23}$
- $B_1$
- $B_2$

**Output:**

- $A_1$
- $A_0$
- $A_{37}$
- $A_{23}$
- $B_1$
- $B_2$

After we've partitioned the data into buckets, we rejoin the data together into the output buffer.

**Total cost:** $O(n+b)$ where $b$ is the number of buckets.

If the number of buckets is small, then this is $O(n)$!
Problems with Bucket Sort

- So what's wrong with Bucket Sort?
  - Doesn't sort the elements within a bucket
  - Requires a small # of keys

- We can solve both with recursion!
Bucket Sort w/ Recursion

Input Data

Sort into buckets using first key
Bucket Sort w/ Recursion

Input Data

Sort into buckets using first key

Sort into buckets using second key
Bucket Sort w/ Recursion

Input Data

Sort into buckets using first key

Sort into buckets using second key

More and more keys....

But where do we get these keys from? What if we have only a single key?
Recognize This?

127 values

63 values       63 values

31 values 31 values 31 values 31 values

15 15 15 15 15 15 15 15

7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
Recognize This?

Quicksort is a lot like bucket sort with recursion...the partitions are buckets.
(I'm not saying they are the same. Just analogous!)

Can we do bucket sort without a Quicksort-style pivot?
# Bucket Sort w/ Recursion

<table>
<thead>
<tr>
<th>If the key type is...</th>
<th>The keys might be...</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>characters</td>
</tr>
<tr>
<td>array</td>
<td>elements</td>
</tr>
<tr>
<td>integer</td>
<td>bits</td>
</tr>
</tbody>
</table>
Bucket Sort w/ Recursion

<table>
<thead>
<tr>
<th>“foo”</th>
<th>“bar”</th>
<th>“baz”</th>
<th>“fred”</th>
<th>“wilma”</th>
</tr>
</thead>
</table>

- “foo” -> “bar” -> “baz” -> “fred” -> “wilma”
Bucket Sort w/ Recursion

So with $w$ layers of recursion, the total cost is $O(wn)$.
Bucket Sort w/ Recursion

W

How large is \( w \)?

We want to split \( n \) into smaller pieces, until the buckets are quite small...
Bucket Sort w/ Recursion

How large is $w$?

We want to split $n$ into smaller pieces, until the buckets are quite small...

...just like Merge Sort...
Bucket Sort w/ Recursion

How large is $w$?

We want to split $n$ into smaller pieces, until the buckets are quite small...

...just like Quicksort...

... so $w = O(\lg n)$
Bucket Sort w/ Recursion

\[ O(wn) = O(n \ lg \ n) \]
Why Bucket Sort?

- Bucket Sort is good if you know that there are a small number of keys
  - Recursion is possible, but you end up with many, many buckets

- If you try to use it for general sorting, it likely just becomes another version of Quicksort or Merge Sort
Linear Sorts

- Linear Sorts
- Bucket Sort
- Radix Sort
- Counting Sort
Why Radix Sort?

- Bucket Sort (when used recursively) creates tons of tiny buckets
- Is there a more elegant way to make it work? Something which doesn't require lots and lots of recursion?

Radix Sort!
What is Radix Sort?

- **Radix Sort** uses the bucket sort method of sorting, but does so *iteratively*, on the same (complete) data set, over and over.
- The trick is: it sorts by the **least significant key** first.
Least Significant Key First

Remember stable vs. unstable sorts?

• A stable sort does not change the order of entries which have the same key
  – Preserves any previous sorting work which had been performed

• Allows us to sort by multiple keys. Example:
  – Sort by first name, then last name
  – Results are grouped by last name, then by first name inside each group
Least Significant Key First

<table>
<thead>
<tr>
<th>Lewis, Russ</th>
<th>Lewis, Emily</th>
<th>Cepin, Eric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lewis, Emily</td>
<td>Cepin, Eric</td>
<td>Cepin, Sue</td>
</tr>
<tr>
<td>Cepin, Sue</td>
<td>Lewis, Russ</td>
<td>Lewis, Emily</td>
</tr>
<tr>
<td>Cepin, Eric</td>
<td>Cepin, Sue</td>
<td>Lewis, Russ</td>
</tr>
</tbody>
</table>

Some Names          Sort by First Name          (Stable) Sort by Last Name

29
Least Significant Key First

The Same Idea Works for Numbers...

<table>
<thead>
<tr>
<th>Some Numbers</th>
<th>Sort by “ones” Digit</th>
<th>(Stable) Sort by “10s” Digit</th>
<th>(Stable) Sort by “100s” Digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>456</td>
<td>801</td>
<td>801</td>
<td>017</td>
</tr>
<tr>
<td>017</td>
<td>123</td>
<td>017</td>
<td>123</td>
</tr>
<tr>
<td>801</td>
<td>456</td>
<td>123</td>
<td>456</td>
</tr>
<tr>
<td>999</td>
<td>017</td>
<td>456</td>
<td>801</td>
</tr>
<tr>
<td>000</td>
<td>999</td>
<td>999</td>
<td>999</td>
</tr>
<tr>
<td>Some Numbers</td>
<td>Sort by “ones” Digit</td>
<td>(Stable) Sort by “10s” Digit</td>
<td>(Stable) Sort by “100s” Digit</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------</td>
<td>-------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>123</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>456</td>
<td>801</td>
<td>801</td>
<td>017</td>
</tr>
<tr>
<td>017</td>
<td>123</td>
<td>017</td>
<td>123</td>
</tr>
<tr>
<td>801</td>
<td>456</td>
<td>123</td>
<td>456</td>
</tr>
<tr>
<td>999</td>
<td>017</td>
<td>456</td>
<td>801</td>
</tr>
<tr>
<td>000</td>
<td>999</td>
<td>999</td>
<td>999</td>
</tr>
</tbody>
</table>

In practice, we'd use bits (or groups of bits)...but the principle remains the same.
Radix Sort

- In theory, Radix Sort has the same problem as Bucket Sort: it eventually becomes $O(n \ lg \ n)$

- In practice, we can sometimes avoid that cost...because integers in real computers have fixed size
  - It's not actually possible to have arbitrarily-large integers.
  - So $\lg n$ is (approximately) a (large) constant!
Linear Sorts

- Linear Sorts
- Bucket Sort
- Radix Sort
- Counting Sort
Why Counting Sort?

- How does Bucket Sort hold its data items?
  - A set of linked lists, like hash table collisions?
    - Lots of memory allocations
    - Lots of total memory: dozens of bytes per element
  - A set of arrays?
    - Need $O(nb)$ capacity, where $b$ is the # of buckets

- How can we reduce the memory cost of bucket sort?
Counting Sort

- **Counting Sort** is a version of Bucket Sort/Radix Sort where we first count the # of elements in each bucket...then move them in a 2nd pass
  - Adds another $O(n)$ (time) pass
  - Adds an $O(b)$ (size) count array
  - But reduces storage overhead to $O(n)$ output array
Counting Sort

Input:

\[
\begin{array}{cccccc}
A_1 & A_0 & A_{37} & C_1 & C_2 & A_{23} \\
\end{array}
\]

This is the same example as we used for Bucket Sort – except that I've renamed the B keys to C.

Let's assume that we have 4 possible keys: A,B,C,D
Counting Sort

Input:

<table>
<thead>
<tr>
<th>A₁</th>
<th>A₀</th>
<th>A₃₇</th>
<th>C₁</th>
<th>C₂</th>
<th>A₂₃</th>
</tr>
</thead>
</table>

Count:

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

The first step is to allocate a count buffer. The size is equal to the # of buckets.
Counting Sort

Input:

<table>
<thead>
<tr>
<th>A₁</th>
<th>A₀</th>
<th>A₃₇</th>
<th>C₁</th>
<th>C₂</th>
<th>A₂₃</th>
</tr>
</thead>
</table>

Count:

<table>
<thead>
<tr>
<th>4</th>
<th>0</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

We now scan through the array, and count the number of times that each key is found.
Next, we rewrite the count buffer so that every slot gives the index where we will put the first element from this bucket.

Note that B and D have no elements, but we fill in the values anyway. These values won't matter to the output.
Counting Sort

**Input:**

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_0$</th>
<th>$A_{37}$</th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$A_{23}$</th>
</tr>
</thead>
</table>

**Count:**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

**Output:**

Now, we allocate the output buffer. It is the **same size** as the input buffer – no linked list overhead, and no duplication.
Counting Sort

Input:

```
| A_1 | A_0 | A_37 | C_1 | C_2 | A_23 |
```

Count:

```
| 0   | 4   | 4   | 6   |
```

Output:

```

```

Finally, we iterate through the input array.

For each key, we look up the proper location for that element in the output array...
Counting Sort

Input:

| $A_1$ | $A_0$ | $A_{37}$ | $C_1$ | $C_2$ | $A_{23}$ |

Count:

| 1  | 4  | 4  | 6  |  |  |

Output:

| $A_1$ |  |  |  |  |  |

Finally, we iterate through the input array.

For each key, we look up the proper location for that element in the output array...

...and copy the value there. We increment the count at the same time.
You notice that the count array now contains the index where we should place the next element with that key.
Counting Sort

So as we iterate through, the count array always tells us where to go.

We write the value to the output, and increment the element in the count array.
Counting Sort

**Input:**

\[
\begin{array}{cccccc}
A_1 & A_0 & A_{37} & C_1 & C_2 & A_{23}
\end{array}
\]

**Count:**

\[
\begin{array}{cccc}
3 & 4 & 4 & 6
\end{array}
\]

**Output:**

\[
\begin{array}{cccccc}
A_1 & A_0 & A_{37}
\end{array}
\]

So as we iterate through, the count array always tells us where to go.

We write the value to the output, and increment the element in the count array.
## Counting Sort

**Input:**

<table>
<thead>
<tr>
<th></th>
<th>A₁</th>
<th>A₀</th>
<th>A₃₇</th>
<th>C₁</th>
<th>C₂</th>
<th>A₂₃</th>
</tr>
</thead>
</table>

**Count:**

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A₀</td>
<td>A₃₇</td>
<td>C₁</td>
<td>C₂</td>
</tr>
</tbody>
</table>

**Output:**

<table>
<thead>
<tr>
<th></th>
<th>A₁</th>
<th>A₀</th>
<th>A₃₇</th>
<th>C₁</th>
</tr>
</thead>
</table>

So as we iterate through, the count array always tells us where to go.

We write the value to the output, and increment the element in the count array.
Counting Sort

Input:

\[
\begin{array}{cccc}
A_1 & A_0 & A_{37} & C_1 & C_2 & A_{23} \\
\end{array}
\]

Count:

\[
\begin{array}{cccc}
3 & 4 & 5 & 6 \\
A & B & C & D \\
\end{array}
\]

Output:

\[
\begin{array}{cccc}
A_1 & A_0 & A_{37} & C_1 \\
\end{array}
\]

Notice that the elements arrive in the output array **out of order**.

However, within each bucket, the elements arrive **in order**.

So Counting Sort is a stable sort.
Counting Sort

Input:

\[
\begin{array}{cccc}
A_1 & A_0 & A_{37} & C_1 \quad C_2 \quad A_{23}
\end{array}
\]

Count:

\[
\begin{array}{cccc}
3 & 4 & 6 & 6
\end{array}
\]

\[
\begin{array}{cccc}
A & B & C & D
\end{array}
\]

Output:

\[
\begin{array}{cccc}
A_1 & A_0 & A_{37} & C_1 \quad C_2
\end{array}
\]

Notice that the elements arrive in the output array out of order.

However, within each bucket, the elements arrive in order.

So Counting Sort is a stable sort.
Counting Sort

Input:

| A_1 | A_0 | A_{37} | C_1 | C_2 | A_{23} |

Count:

| 4   | 4   | 6     | 6   |

A | B | C | D

Output:

| A_1 | A_0 | A_{37} | A_{23} | C_1 | C_2 |

Notice that the elements arrive in the output array **out of order**.

However, within each bucket, the elements arrive **in order**.

So Counting Sort is a stable sort.
Counting Sort

- Total cost of Counting Sort:
  - $O(n+b)$ space
  - $O(n) + O(b) + O(n) = O(n+b)$ time

- Counting Sort is a **stable** sort

- Often, Counting Sort is used as a subroutine inside of Radix Sort
  - Preallocate the memory once, reuse it in each pass