Warm up

Define:

– In Place
– Adaptive
– Stable
– Parallelizable
Mergesort

- **Divide:**
  - Break $n$-element list into two lists of $n/2$ elements

- **Conquer:**
  - If $n > 1$: Sort each sublist recursively
  - If $n = 1$: List is already sorted (base case)

- **Combine:**
  - Merge together sorted sublists into one sorted list

**Run Time?**
$\Theta(n \log n)$

**Optimal!**

<table>
<thead>
<tr>
<th>In Place?</th>
<th>Adaptive?</th>
<th>Stable?</th>
<th>Parallelizable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes! (usually)</td>
<td>Yes!</td>
</tr>
</tbody>
</table>
Speed Isn’t Everything

• Important properties of sorting algorithms:
  • **Run Time**
    – Asymptotic Complexity
    – Constants
  • **In Place (or In-Situ)**
    – Done with only constant additional space
  • **Adaptive**
    – Faster if list is nearly sorted
  • **Stable**
    – Equal elements remain in original order
  • **Parallelizable**
    – Runs faster with multiple computers
**Quicksort**

- Idea: pick a partition element, recursively sort two sublists around that element
- **Divide**: select an element \( p \), `Partition(p)`
- **Conquer**: recursively sort left and right sublists
- **Combine**: Nothing!

**Run Time?**
\( \Theta(n \log n) \)
(almost always)

Better constants than Mergesort

<table>
<thead>
<tr>
<th>In Place?</th>
<th>Adaptive?</th>
<th>Stable?</th>
<th>Parallelizable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>kinda</td>
<td>No!</td>
<td>No</td>
<td>Yes!</td>
</tr>
</tbody>
</table>

Uses stack for recursive calls
Insertion Sort

• Idea: Maintain a sorted list prefix, extend that prefix by “inserting” the next element
Insertion Sort

- Idea: Maintain a sorted list prefix, extend that prefix by “inserting” the next element

**Run Time?**
\[ \Theta(n^2) \]
(but with very small constants)
Great for short lists!

**In Place?**
Yes!

**Adaptive?**
Yes
Insertion Sort is Adaptive

- **Idea**: Maintain a *sorted list prefix*, extend that prefix by “inserting” the next element

Only one comparison needed per element!  
Runtime: $O(n)$
**Insertion Sort**

- **Idea:** Maintain a sorted list prefix, extend that prefix by “inserting” the next element

<table>
<thead>
<tr>
<th>In Place?</th>
<th>Adaptive?</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes!</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Run Time?**

$\Theta(n^2)$

(but with very small constants)

Great for short lists!
Insertion Sort is Stable

• **Idea:** Maintain a *sorted list prefix*, extend that prefix by “inserting” the *next element*.

![Sorted Prefix Diagram]

The “second” 10 will stay to the right.
Insertion Sort

- Idea: Maintain a sorted list prefix, extend that prefix by “inserting” the next element

Run Time?
\( \Theta(n^2) \)
(but with very small constants)
Great for short lists!

**In Place?**
Yes!

**Adaptive?**
Yes

**Stable?**
Yes

**Parallelizable?**
No

**Online?**
Yes

Can sort a list as it is received, i.e., don’t need the entire list to begin sorting

“All things considered, it’s actually a pretty good sorting algorithm!” –Nate Brunelle
Heap Sort

- **Idea**: Build a Heap, repeatedly extract max element from the heap to build sorted list Right-to-Left

Max Heap Property: Each node is larger than its children
Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call Heapify(root)

Max Heap Property: Each node is larger than its children

Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree
Heap Sort

• Remove the Max element (i.e. the root) from the Heap: replace with last element, call Heapify(root)

Max Heap Property: Each node is larger than its children

Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree
Heap Sort

• Remove the Max element (i.e. the root) from the Heap: replace with last element, call Heapify(root)

Max Heap Property: Each node is larger than its children

Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree
Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call Heapify(root)

Max Heap Property: Each node is larger than its children

Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree
Heap Sort

- **Idea**: Build a Heap, repeatedly extract max element from the heap to build sorted list Right-to-Left

**Run Time?**
\[ \Theta(n \log n) \]

- Constants worse than Quick Sort

**In Place?**  
**Yes!**

- When removing an element from the heap, move it to the (now unoccupied) end of the list
In Place Heap Sort

• **Idea:** When removing an element from the heap, move it to the (now unoccupied) end of the list.

Max Heap Property: Each node is larger than its children.
In Place Heap Sort

• **Idea**: When removing an element from the heap, move it to the (now unoccupied) end of the list.

Max Heap Property: Each node is larger than its children.
In Place Heap Sort

- **Idea**: When removing an element from the heap, move it to the (now unoccupied) end of the list.

Max Heap Property: Each node is larger than its children.
In Place Heap Sort

• **Idea**: When removing an element from the heap, move it to the (now unoccupied) end of the list.

Max Heap Property: Each node is larger than its children.
In Place Heap Sort

- **Idea**: When removing an element from the heap, move it to the (now unoccupied) end of the list.

Max Heap Property: Each node is larger than its children.
# Heap Sort

- **Idea:** Build a Heap, repeatedly extract max element from the heap to build sorted list Right-to-Left

<table>
<thead>
<tr>
<th>In Place?</th>
<th>Adaptive?</th>
<th>Stable?</th>
<th>Parallelizable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes!</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Run Time?**

$\Theta(n \log n)$

Constants worse than Quick Sort
Sorting in Linear Time

• Cannot be comparison-based
• Need to make some sort of assumption about the contents of the list
  – Small number of unique values
  – Small range of values
  – Etc.
Counting Sort

- **Idea:** Count how many things are less than each element

1. Range is $[0, k - 1]$ (here $[0,5]$)
   - make an array $C$ of size $k$
   - populate with counts of each value

   
   For $i$ in $L$:
   
   $$ + +C[L[i]] $$

2. Take “running sum” of $C$
   - to count things less than each value

   For $i = 1$ to len($C$):
   
   $$ C[i] = C[i - 1] + C[i] $$

---

$C[$ running sum $]$ To sort: last item of value 2 is 4th in the list
Counting Sort

• **Idea:** Count how many things are less than each element

$L = \begin{array}{ccccccc}
2 & 5 & 5 & 0 & 2 & 3 & 0 \\
0 & 1 & 2 & 3 & 4 & 5 & 6
\end{array}$

$C = \begin{array}{ccccccccc}
2 & 2 & 4 & 5 & 5 & 8 \\
0 & 1 & 2 & 3 & 4 & 5
\end{array}$

For each element of $L$ (last to first):
- Use $C$ to find its proper place in $B$
- Decrement that position of $C$

$B = \begin{array}{ccccccc}
\text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & 5 \\
0 & 1 & 2 & 3 & 4 & 5 & 6
\end{array}$

For $i = \text{len}(L)$ downto 1:
- $B[C[L[i]] - 1] = L[i]$
- $C[L[i]] = C[L[i]] - 1$

Last item of value 5 goes at index 7
Counting Sort

• Idea: Count how many things are less than each element

For each element of $L$ (last to first):
Use $C$ to find its proper place in $B$
Decrement that position of $C$

Run Time: $O(n + k)$
Memory: $O(n + k)$
Counting Sort

• Why not always use counting sort?
• For 64-bit numbers, requires an array of length $2^{64} > 10^{19}$
  – 5 GHz CPU will require $> 116$ years to initialize the array
  – 18 Exabytes of data
    • Total amount of data that Google has (?)

One Exabyte = $10^{18}$ bytes
1 million terabytes (TB)
1 billion gigabytes (GB)
100,000 x Library of Congress (print)
12 Exabytes
Radix Sort

• **Idea:** Stable sort on each digit, from least significant to most significant

<table>
<thead>
<tr>
<th>103</th>
<th>801</th>
<th>401</th>
<th>323</th>
<th>255</th>
<th>823</th>
<th>999</th>
<th>101</th>
<th>113</th>
<th>901</th>
<th>555</th>
<th>512</th>
<th>245</th>
<th>800</th>
<th>018</th>
<th>121</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>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Place each element into a “bucket” according to its 1’s place

<table>
<thead>
<tr>
<th>800</th>
<th>801</th>
<th>401</th>
<th>101</th>
<th>901</th>
<th>121</th>
<th>512</th>
<th>103</th>
<th>323</th>
<th>823</th>
<th>113</th>
<th>255</th>
<th>555</th>
<th>245</th>
<th>018</th>
<th>999</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>
<td></td>
<td>018</td>
<td>999</td>
</tr>
</tbody>
</table>

33
Radix Sort

• **Idea**: Stable sort on each digit, from least significant to most significant

Place each element into a “bucket” according to its 10’s place
Radix Sort

• **Idea:** **Stable sort** on each digit, from least significant to most significant

Place each element into a “bucket” according to its 100’s place

Run Time: $O(d(n + b))$

$d = \text{digits in largest value}$

$b = \text{base of representation}$