Warm up
Build a Max Heap from the following Elements:
4, 15, 22, 6, 18, 30, 14, 21

Heap
- Heap Property: Each node must be larger than its children

Today’s Keywords
- Sorting
- Quicksort
- Sorting Algorithm Characteristics
- Insertion Sort
- Bubble Sort
- Heap Sort
- Linear time Sorting
- Counting Sort
- Radix Sort
CLRS Readings

- Chapter 6
- Chapter 8

Homeworks

- HW3 due 11pm Wednesday Feb. 20
  - Divide and conquer
  - Written (use LaTeX!)
- HW4 coming on Wednesday
- Grading Notes
  - HW0 has been graded and released
  - HW1 grades (and solutions) released on Wednesday
  - HW2 is currently being graded (released tomorrow!)

Generic Divide and Conquer Solution

```python
def myDCalgo(problem):
    if baseCase(problem):
        solution = solve(problem) # brute force if necessary
        return solution
    subproblems = Divide(problem)
    for subproblem in problem:
        subsolutions.append(myDCalgo(subproblem))
    solution = Combine(subsolutions)
    return solution
```
Generic Divide and Conquer Solution

```
def mergesort(list):
    if len(list) < 2:
        return list  # list of size 1 is sorted!
    (listL, listR) = Divide_by_median(list)
    for list in (listL, listR):
        sortedSubLists.append(mergesort(list))
    solution = merge(sortedL, sortedR)
    return solution
```

MergeSort Divide and Conquer Solution

```
def mergesort(list):
    if len(list) < 2:
        return list  # list of size 1 is sorted!
    (listL, listR) = Divide_by_median(list)
    for list in (listL, listR):
        sortedSubLists.append(mergesort(list))
    solution = merge(sortedL, sortedR)
    return solution
```
### Strategy: Decision Tree

- Conclusion: Worst Case Optimal run time of sorting is $\Theta(n \log n)$
  - There is no (comparison-based) sorting algorithm with run time $O(n \log n)$

### Sorting, so far

- Sorting algorithms we have discussed:
  - Mergesort $O(n \log n)$ Optimal!
  - Quicksort $O(n \log n)$ Optimal!
- Other sorting algorithms
  - Bubblesort $O(n^2)$
  - Insertionsort $O(n^2)$
  - Heapsort $O(n \log n)$ Optimal!

### 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
Mergesort

- **Divide:** Break an element list into two lists of \( \frac{n}{2} \) 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!</td>
<td>Yes! (usually)</td>
</tr>
</tbody>
</table>

Merge

- **Combine:** Merge sorted sublists into one sorted list.
- We have:
  - 2 sorted lists \( L_1, L_2 \)
  - 1 output list \( L_{out} \)

While \( L_1 \) and \( L_2 \) not empty:

- **Stable:** If elements are equal, leftmost comes first
- If \( L_1[0] \leq L_2[0] \):
  - Append \( L_1 \) to \( L_{out} \) and pop.
- Else:
  - Append \( L_2 \) to \( L_{out} \) and pop.
  - Append \( L_1 \) to \( L_{out} \).
  - Append \( L_2 \) to \( L_{out} \).

Mergesort

- **Divide:** Break an element list into two lists of \( \frac{n}{2} \) 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!</td>
<td>Yes! (usually)</td>
</tr>
</tbody>
</table>
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

---

**Mergesort (Sequential)**

\[ T(n) = 2T\left(\frac{n}{2}\right) + n \]

- $n$: total / level
- $\log_2 n$: levels of recursion

Run Time: $\Theta(n \log n)$

---

**Mergesort (Parallel)**

\[ T(n) = T\left(\frac{n}{2}\right) + n \]

Done in Parallel

Run Time: $\Theta(n)$
Quicksort

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

<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

Run Time?
\( \Theta(n \log n) \) (almost always)
Better constants than Mergesort

Bubble Sort

- Idea: March through list, swapping adjacent elements if out of order, repeat until sorted

```
8 5 7 9 12 10 1 2 4 3 6 11
5 8 7 9 12 10 1 2 4 3 6 11
5 7 8 9 12 10 1 2 4 3 6 11
5 7 8 9 12 10 1 2 4 3 6 11
```

Run Time?
\( \Theta(n^2) \)
Constants worse than Insertion Sort

“Compared to straight insertion [...], bubble sorting requires a more complicated program and takes about twice as long!”

—Donald Knuth

In Place? Adaptive?
Yes Kinda
Bubble Sort is “almost” Adaptive

- **Idea**: March through list, swapping adjacent elements if out of order
  - Only makes one “pass”
  - After one “pass”
  - Requires \( n \) passes, thus is \( \Theta(n^2) \)

#### Bubble Sort

- **Idea**: March through list, swapping adjacent elements if out of order, repeat until sorted
- **In Place?** Yes!
- **Adaptive?** Kinda
- **Stable?** Yes
- **Parallelizable?** No
- "the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems" – Donald Knuth, The Art of Computer Programming

#### Insertion Sort

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

Insertion Sort

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

In Place?
Yes!

Adaptive?
Yes

\[\begin{array}{c}
\text{Sorted Prefix} \\
1 \ 2 \ 3 \ 4 \ 5 \\
\end{array}\]

Only one comparison needed per element!
Runtime: \( O(n) \)

---

Insertion Sort is Adaptive

- Idea: Maintain a \textit{sorted list prefix}, extend that prefix by "inserting" the next element

\[\begin{array}{c}
\text{Sorted Prefix} \\
1 \ 2 \ 3 \ 4 \ 5 \\
\end{array}\]

---

Insertion Sort

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

In Place?
Yes!

Adaptive?
Yes

Stable?
Yes
Insertion Sort is Stable

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

```
Sorted Prefix: 3 5 7 8 10 10' 12 2 4 6 1 11
```

The "second" 10 will stay to the right.

**Insertion Sort**

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

```
In Place? Yes
Adaptive? Yes
Stable? Yes
Parallelizable? No
Online? Yes
```

- **Run Time?** $\Theta(n^2)$ (but with very small constants)

- **Great for short lists!**

- "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

Run Time?

$\Theta(n \log n)$

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
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

Max Heap
Property: Each node is larger than its children

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

- **Run Time**: $\Theta(n \log n)$
- **Constants worse than Quick Sort**
- **In Place**: Yes!
- **Adaptive**: No
- **Stable**: No
- **Parallelizable**: No

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

\[ L = \begin{bmatrix} 3 & 6 & 6 & 1 & 3 & 4 & 1 & 6 \end{bmatrix} \]

1. Range is \([1,6]\) (here \([1,6]\) )
   populate with counts of each value
   \[
   C = \begin{bmatrix} \text{count of 1} & \text{count of 2} & \cdots & \text{count of 6} \end{bmatrix}
   \]
   For \(i \in L: \quad C[i]++
   
2. Take "running sum" of \(C\) to count things less than each value
   \[
   C[i] = C[i] + C[i-1]
   \]
   \[
   \text{To sort: last item of value 3 goes at index 4}
   \]

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

Radix Sort

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

```
   801 401 101 901 121 800 800
   801 401 101 901 121 800 800
```
Place each element into a "bucket" according to its 1's place

```
   8 9 10 11 12 13 14 15
```

12 Exabytes

```
   801 401 101 901 121 800 800
   801 401 101 901 121 800 800
```

Place each element into a "bucket" according to its 0's place

```
   8 9 10 11 12 13 14 15
```
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

```
<table>
<thead>
<tr>
<th>889</th>
<th>907</th>
<th>990</th>
<th>910</th>
<th>912</th>
<th>913</th>
<th>918</th>
<th>920</th>
<th>928</th>
</tr>
</thead>
<tbody>
<tr>
<td>891</td>
<td>901</td>
<td>991</td>
<td>911</td>
<td>914</td>
<td>916</td>
<td>919</td>
<td>921</td>
<td>929</td>
</tr>
<tr>
<td>892</td>
<td>902</td>
<td>992</td>
<td>912</td>
<td>915</td>
<td>917</td>
<td>920</td>
<td>921</td>
<td>929</td>
</tr>
<tr>
<td>893</td>
<td>903</td>
<td>993</td>
<td>913</td>
<td>916</td>
<td>918</td>
<td>922</td>
<td>924</td>
<td>930</td>
</tr>
<tr>
<td>894</td>
<td>904</td>
<td>994</td>
<td>914</td>
<td>917</td>
<td>920</td>
<td>924</td>
<td>925</td>
<td>930</td>
</tr>
<tr>
<td>895</td>
<td>905</td>
<td>995</td>
<td>915</td>
<td>918</td>
<td>921</td>
<td>925</td>
<td>926</td>
<td>930</td>
</tr>
<tr>
<td>896</td>
<td>906</td>
<td>996</td>
<td>916</td>
<td>920</td>
<td>922</td>
<td>926</td>
<td>927</td>
<td>930</td>
</tr>
<tr>
<td>897</td>
<td>907</td>
<td>997</td>
<td>917</td>
<td>921</td>
<td>923</td>
<td>927</td>
<td>928</td>
<td>930</td>
</tr>
</tbody>
</table>
```

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

Place each element into a "bucket" according to its 100's place

```
<table>
<thead>
<tr>
<th>8901</th>
<th>9001</th>
<th>9901</th>
<th>9101</th>
<th>9102</th>
<th>9103</th>
<th>9108</th>
<th>9201</th>
<th>9208</th>
</tr>
</thead>
<tbody>
<tr>
<td>8911</td>
<td>9011</td>
<td>9911</td>
<td>9111</td>
<td>9114</td>
<td>9116</td>
<td>9119</td>
<td>9211</td>
<td>9219</td>
</tr>
<tr>
<td>8921</td>
<td>9021</td>
<td>9921</td>
<td>9121</td>
<td>9124</td>
<td>9126</td>
<td>9129</td>
<td>9221</td>
<td>9229</td>
</tr>
<tr>
<td>8931</td>
<td>9031</td>
<td>9931</td>
<td>9131</td>
<td>9134</td>
<td>9136</td>
<td>9139</td>
<td>9231</td>
<td>9239</td>
</tr>
<tr>
<td>8941</td>
<td>9041</td>
<td>9941</td>
<td>9141</td>
<td>9144</td>
<td>9146</td>
<td>9149</td>
<td>9241</td>
<td>9249</td>
</tr>
<tr>
<td>8951</td>
<td>9051</td>
<td>9951</td>
<td>9151</td>
<td>9154</td>
<td>9156</td>
<td>9159</td>
<td>9251</td>
<td>9259</td>
</tr>
<tr>
<td>8961</td>
<td>9061</td>
<td>9961</td>
<td>9161</td>
<td>9164</td>
<td>9166</td>
<td>9169</td>
<td>9261</td>
<td>9269</td>
</tr>
<tr>
<td>8971</td>
<td>9071</td>
<td>9971</td>
<td>9171</td>
<td>9174</td>
<td>9176</td>
<td>9179</td>
<td>9271</td>
<td>9279</td>
</tr>
</tbody>
</table>
```

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

- $d =$ digits in largest value
- $b =$ base of representation