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!)
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 list.length < 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

\[ \frac{n}{4} \]

\[ \frac{n}{4} \]

\[ \frac{n}{4} \]

\[ \frac{n}{4} \]

\[ 1 \]

\[ 1 \]

\[ 1 \]

\[ 1 \]

\[ \cdots \]

\[ 1 \]

\[ 1 \]

\[ 1 \]
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 $n$-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>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</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):

If $L_1[0] \leq L_2[0]$:

$$L_{out}.append(L_1.pop())$$

Else:

$$L_{out}.append(L_2.pop())$$

$L_{out}.append(L_1)$

$L_{out}.append(L_2)$

**Stable**: If elements are equal, leftmost comes first
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!</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

Parallelizable: Allow different machines to work on each sublist
Mergesort (Sequential)

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

Run Time: \( \Theta(n \log n) \)
Mergesort (Parallel)

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

Run Time: $\Theta(n)$
QuickSort

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

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

**In Place?**
kinda
Uses stack for recursive calls

**Adaptive?**
No!

**Stable?**
No

**Parallelizable?**
Yes!
Bubble Sort

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

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

Run Time?

$\Theta(n^2)$

Constants worse than Insertion Sort

In Place? Yes

Adaptive? Kinda

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

–Donald Knuth
Bubble Sort is “almost” Adaptive

- **Idea**: March through list, swapping adjacent elements if out of order

<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>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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></td>
</tr>
</tbody>
</table>

Only makes one “pass”

<table>
<thead>
<tr>
<th></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>
<th>10</th>
<th>11</th>
<th>12</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<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>1</td>
<td></td>
</tr>
</tbody>
</table>

After one “pass”

<table>
<thead>
<tr>
<th></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>
<th>10</th>
<th>11</th>
<th>12</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<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>1</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Requires $n$ passes, thus is $O(n^2)$
# Bubble Sort

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

<table>
<thead>
<tr>
<th>In Place?</th>
<th>Adaptive?</th>
<th>Stable?</th>
<th>Parallelizable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes!</td>
<td>Kinda</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Not really</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

"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

- 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?  Adaptive?
Yes!  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?
Θ(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**

The “second” 10 will stay to the right
**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>
<th>Parallelizable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes!</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

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

“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

**In Place?**

Yes!

When removing an element from the heap, move it to the (now unoccupied) end of the list

**Run Time?**

$\Theta(n \log n)$

Constants worse than Quick Sort
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
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

Run Time? \( \Theta(n \log n) \)
Constants worse than Quick Sort

<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>
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

  \[ L = \begin{array}{cccccccc}
  3 & 6 & 6 & 1 & 3 & 4 & 1 & 6 \\
  1 & 2 & 3 & 4 & 5 & 6 & 7 & 8
  \end{array} \]

  1. Range is \([1, k]\) (here \([1,6]\))
     make an array \(C\) of size \(k\)
     populate with counts of each value
     
     \[
     \text{For } i \text{ in } L: \quad C[L[i]]++
     \]

  2. Take “running sum” of \(C\)
     to count things less than each value
     
     \[
     \text{For } i = 1 \text{ to } \text{len}(C): \quad C[i] = C[i - 1] + C[i]
     \]

  \[ C = \begin{array}{ccccccc}
  2 & 0 & 2 & 1 & 0 & 3 \\
  1 & 2 & 3 & 4 & 5 & 6
  \end{array} \]

  running sum

  \[ C = \begin{array}{ccccccc}
  2 & 2 & 4 & 5 & 5 & 8 \\
  1 & 2 & 3 & 4 & 5 & 6
  \end{array} \]

To sort: last item of value 3 goes at index 4
Counting Sort

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

\[
L = \begin{bmatrix}
3 & 6 & 6 & 1 & 3 & 4 & 1 & 6 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
2 & 2 & 4 & 5 & 5 & 7 \\
1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}
\]

Last item of value 6 goes at index 8

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

\[
B = \begin{bmatrix}
& & & & & & 6 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8
\end{bmatrix}
\]

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

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

$$L = \begin{array}{cccccccc}
3 & 6 & 6 & 1 & 3 & 4 & 1 & 6 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}$$

$$C = \begin{array}{cccccccc}
1 & 2 & 4 & 5 & 5 & 7 \\
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}$$

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

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

$$B = \begin{array}{cccccccc}
1 & & & & & & 6 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}$$

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
12 Exabytes
Radix Sort

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

Place each element into a “bucket” according to its 1’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 10’s place

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

<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>800</td>
<td>801</td>
<td>401</td>
<td>101</td>
<td>901</td>
<td>103</td>
<td>512</td>
<td>121</td>
<td>323</td>
<td>823</td>
<td>018</td>
</tr>
<tr>
<td></td>
<td>255</td>
<td>555</td>
<td></td>
<td></td>
<td></td>
<td>245</td>
<td></td>
<td></td>
<td></td>
<td>999</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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: $O(d(n + b))$

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