Median with 'Quickselect'

- The first partitioning leaves us with two subarrays, but we need only recurse on the one that contains the median

\[
\begin{align*}
3 & 6 & 4 & 2 & 1 & \underline{7} & 5 \\
3 & 1 & 4 & 2 & 6 & \underline{7} & 5 \\
3 & 1 & 4 & 2 & \underline{5} & 7 & 6 \\
3 & 1 & 4 & 2 \\
1 & 3 & 4 & 2 \\
1 & 2 & 4 & 3 \\
1 & 2 & 3 & 4 \\
\end{align*}
\]

\[n = 7 \implies 4^{th} \text{ pos} \]
\[ f(n) = f\left(\frac{n}{2}\right) + kn \]

\[ = f\left(2^{m-1}\right) + k2^m \]
\[ = f\left(2^{m-2}\right) + k2^{m-1} + k2^m \]
\[ = f\left(2^{m-a}\right) + k \sum_{i=M-a+1}^{m} 2^i \]
\[ = f\left(2^0\right) + k \sum_{i=0}^{m} 2^i \]
\[ = 2^0 + 2^1 + 2^2 \]
\[ = 1 + 2 + 4 \]
\[ = K_2 + k \cdot 2^{m-1} - 1 \]
\[ = K_2 + k \cdot 2 \cdot (n-1) \Rightarrow \quad O(n) \]

- Exercise: show the worst case is still \( O(n^2) \)
Heapsort

- One last interesting algorithm
- Sorts in place and guarantees O(nlogn) for any input!
  - Although the constant of proportionality is greater than for quicksort
- Particularly interesting for us because it is based on a data structure called a heap (which we will use later on too)
Introducing Heaps

- Consider a simple binary tree (two branches out of each node)
- There is only one rule for our heap: the value of the two children must each be less than the value of the parent

Note:
- The root of the tree is always the biggest number
- The root of any subtree is the biggest number in that subtree
Introducing Heaps

- Now we represent this tree using an array in a certain way:
- The children of the node at \([i]\) can be found at \([2i+1]\) and \([2i+2]\) (array starts at zero)
Heapsort proceeds as follows:
1. Make your data into a heap in the array [0...n]
2. for ( k=n-1...0 )
   1. Swap element 0 and element k
   2. Make the array [0...k] a valid heap

Trick is in the 'heapify' bit
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Heapsort: Array View**
Heapsort: Analysis of Step 1

Assume a valid heap except first element

\[ O(h) \] to fix that one element (\( h = \log n \))

Each level \( j \) have \( 2^j \) nodes
Each node \( \Theta(\text{height of node}) \)

For a given level, \( \text{num nodes} \times \text{cost} = 2^j \times (h-j) k \)
Heapsort: Analysis of Step 1

Total cost

\[
\sum_{i=0}^{n} k \cdot 2^j \cdot (h-j)
\]
Fixing up heaps where only first element is bad
\[ \Rightarrow \text{Cost } O(h) = O(\log n) \leq \text{ one fix} \]

How many fixes (iterations)? \( n \)

\[ O(n\log n) \]

Heapsort \( O(n\log n + n) \)
\[ O(n\log n) \]
Heapsort

- Achieves guaranteed $O(n \log n)$ performance
- Sorts in place: $O(1)$ space
- But in the average case, quicksort still beats it :-(
O(n) Sorting :-)

- Sometimes we try so hard to solve something using the tools we think apply that we miss the opportunity to do it completely differently....

\begin{center}
\begin{tabular}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 2 & 1 & 1 & 0 & 4 & 0 \\
\end{tabular}
\end{center}
Limitations

- We need to create an array for the histogram that runs from MIN_NUMBER...MAX_NUMBER
  - If this was just a java `int` than this is $-2^{31} \ldots (2^{31} - 1)$
  - Assuming 4 bytes for each slot, that's a total cost of $2^{31} \text{ B} = 2^{14} \text{ MB} = 16 \text{ GB}$ for the histogram (!)
  - i.e. it's $O(\text{MAX\_NUMBER-\text{MIN\_NUMBER}})$ in space!!

![Diagram]

- Scan max/min  
- Construct hist  
- Read results  
- $O(n)$  
- $O(n)$  
- $O(n)$  
- $O(n)$  

Space $O(b-s)$
Stability

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
</table>
age: \textit{int} |
| name: \textit{String} |

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>p5</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>19</td>
<td>30</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>Bob</td>
<td>Alice</td>
<td>Steve</td>
<td>John</td>
<td>Pete</td>
</tr>
</tbody>
</table>

- Want to sort by age
  - But there are multiple sorted solutions:
    - p2, p4, p1, p5, p3
    - p4, p2, p1, p5, p3
    - p4, p2, p5, p1, p3
    - p2, p4, p5, p1, p3

- \textbf{Stable} algorithms preserve the order found in the input in these cases