QuickSort(array A, int p, int r)
1   if (p < r)
2       then q ← Partition(A, p, r)
3       QuickSort(A, p, q − 1)
4       QuickSort(A, q + 1, r)

To sort array call QuickSort(A, 1, length[A]).

Partition(array A, int p, int r)
1   x ← A[r] ▷ Choose pivot
2   i ← p − 1
3   for j ← p to r − 1
4       do if (A[j] ≤ x)
5           then i ← i + 1
7   exchange A[i + 1] ← A[r]
8   return i + 1
Analysis of QuickSort

- **Average case**
  - $T(n) \leq 2T(n/2) + O(n)$
  - $T(n) = O(n \log n)$

- **Worst case**
  - $T(n) = T(n-1) + O(n)$
  - $T(n) = O(n^2)$

- **“IN PLACE” sorting algorithm**
  - Which sorting algorithm is not an “IN PLACE” sorting algorithm?
Solving Recurrence Relations

Page 62, [CLR]

<table>
<thead>
<tr>
<th>Recurrence; Cond</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(n)$</td>
<td>$T(n) = O(n^2)$</td>
</tr>
<tr>
<td>$T(n) = T(n-c) + O(1)$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = T(n-c) + O(n)$</td>
<td>$T(n) = O(n^2)$</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>$T(n) = O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + O(n)$; $a = b$</td>
<td>$T(n) = O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + O(n)$; $a &lt; b$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a - \epsilon})$</td>
<td>$T(n) = O(n)$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a})$</td>
<td>$T(n) = \Theta(n^{\log_b a \log n})$</td>
</tr>
<tr>
<td>$T(n) = aT(n/b) + f(n)$; $f(n) = \Theta(f(n))$; $af(n/b) \leq cf(n)$</td>
<td>$T(n) = \Omega(n^{\log_b a \log n})$</td>
</tr>
</tbody>
</table>
Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- ShakerSort
- QuickSort
- MergeSort
- HeapSort
- Bucket & Radix Sort
- Counting Sort
HeapSort

• First convert array into a heap (**BUILD-MAX-HEAP**, p133)
• Then convert heap into sorted array (**HEAPSORT**, p136)
Storing binary trees as arrays
Heaps (Max-Heap)

HEAP represents a binary tree stored as an array such that:
- Tree is filled on all levels except last
- Last level is filled from left to right
- Left & right child of i are in locations $2i$ and $2i+1$
- HEAP PROPERTY:
  Parent value is at least as large as child’s value
HeapSort: Part 1

\[\text{Max-Heapify}(\text{array } A, \text{int } i)\]

\(\triangleright\) Assume subtree rooted at \(i\) is not a heap;
\(\triangleright\) but subtrees rooted at children of \(i\) are heaps

1. \(l \leftarrow \text{LEFT}[i]\)
2. \(r \leftarrow \text{RIGHT}[i]\)
3. \(\text{if } ((l \leq \text{heap-size}[A]) \text{ and } (A[l] > A[i]))\)
   \(\text{then } \text{largest} \leftarrow l\)
   \(\text{else } \text{largest} \leftarrow i\)
4. \(\text{if } ((r \leq \text{heap-size}[A]) \text{ and } (A[r] > A[\text{largest}]))\)
   \(\text{then } \text{largest} \leftarrow r\)
5. \(\text{if } (\text{largest} \neq i)\)
   \(\text{then } \text{exchange } A[i] \leftrightarrow A[\text{largest}]\)
6. \(\text{MAX-HEAPIFY}(A, \text{largest})\)

\(O(\text{height of node in location } i) = O(\log(\text{size of subtree}))\)
HeapSort: Part 2

\begin{verbatim}
BUILD-MAX-HEAP(array A)
1 heap-size[A] ← length[A]
2 for i ← \lfloor length[A]/2 \rfloor \ downto 1
3 \hspace{1em} do MAX-HEAPIFY(A, i)
\end{verbatim}
HeapSort: Part 2

**Build-Max-Heap**\( (array \ A) \)

1. \( \text{heap-size}[A] \leftarrow \text{length}[A] \)
2. \textbf{for} \( i \leftarrow \lfloor \text{length}[A]/2 \rfloor \) \textbf{downto} 1
3. \quad \textbf{do} \text{Max-Heapify}(A, i)

**HeapSort**\( (array \ A) \)

1. \textbf{Build-Max-Heap}(A)
2. \textbf{for} \( i \leftarrow \text{length}[A] \) \textbf{downto} 2
3. \quad \textbf{do} exchange \( A[1] \leftarrow A[i] \)
4. \qquad \text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1
5. \quad \textbf{Max-Heapify}(A, 1)

\text{Total: } O(n \log n)
For the HeapSort analysis, we need to compute:

\[ \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \]

We know from the formula for geometric series that

\[ \sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \]

Differentiating both sides, we get

\[ \sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1 - x)^2} \]

Multiplying both sides by \(x\) we get

\[ \sum_{k=0}^{\infty} kx^k = \frac{x}{(1 - x)^2} \]

Now replace \(x = 1/2\) to show that

\[ \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \frac{1}{2} \]
Visualizing Algorithms 1

What algorithms are $A$ and $B$?
Visualizing Algorithms 2

Value

Position

Unsorted

Sorted
Visualizing Comparisons 3
Animations

- [http://cg.scs.carleton.ca/~morin/misc/sortalg/](http://cg.scs.carleton.ca/~morin/misc/sortalg/)
- [http://home.westman.wave.ca/~rhenry/sort/](http://home.westman.wave.ca/~rhenry/sort/)
  - time complexities on best, worst and average case
  - runs on almost sorted, reverse, random, and unique inputs; shows code with invariants
  - comparisons, movements & stepwise animations with user data
- [http://maven.smith.edu/~thiebaut/java/sort/demo.html](http://maven.smith.edu/~thiebaut/java/sort/demo.html)
  - comparisons & data movements and step by step execution
Problems to think about!

• What is the least number of comparisons you need to sort a list of 3 elements? 4 elements? 5 elements?

• How to arrange a tennis tournament in order to find the tournament **champion** with the least number of matches? How many tennis matches are needed?