6.006 Lecture 8: Sorting I

- Review: Insertion sort
- Merge sort
- Review: Insertion sort
- Sorting in place & Selection sorting
- Heaps
- CLRS 2.1, 2.2, 2.3 Ch. 6 (some on Thu)

**Insertion Sort**

<table>
<thead>
<tr>
<th>Sorted</th>
<th>Insert into sorted prefix</th>
<th>Sorted</th>
<th>Insert</th>
<th>Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 2 4 6 1 3</td>
<td>[\downarrow]</td>
<td>2 5 4 6 1 3</td>
<td>[\downarrow]</td>
<td>2 4 5 6 1 3</td>
</tr>
<tr>
<td></td>
<td>[\rightarrow]</td>
<td></td>
<td>[\rightarrow]</td>
<td></td>
</tr>
</tbody>
</table>

Array is sorted in place; only extra memory is 2 indices and 1 element; constant extra storage that does not depend on \( n \) (problem size).

Running time is \( \Theta(n^2) \)

worst case is \( \begin{bmatrix} n & n-1 & n-2 & \cdots & 3 & 2 & 1 \end{bmatrix} \)

can we do better in place?
Merge Sort

Input: 5 2 7 6 1 3

Divide: 5 2 4 6 1 3

Sort recursively: 2 4 5 1 3 6

Merge: 1 2 3 4 5 6

Cannot do this step in place

$\Theta(n)$ work for merging

$\Theta(n \log n)$ work overall

But requires $\Theta(n)$ extra storage!
Selection Sort

Find maximum element, swap with position n-1 (last)
repeat on A[0:n-1]
repeat ...

In place, but finding maximum at every step costs $\Theta(n)$, so total is $\Theta(n^2)$ work.

Can we find the max in $\Theta(\log n)$ time?

Heaps

(term is also used for an area used for dynamic memory allocation; not related to this lecture)

- Abstractly: a binary tree, values at nodes
  - node value larger than values at children
    (compare to a different invariant in BSTs)

- Tree is nearly complete

- Because it is nearly complete, we can store it in an array (Python list) with no explicit pointers.
Example:

```
               16
              /   \
             14    20
            /     / \
           8     9  6
          /     /   /
         2     4   3
```

Not sorted (up or down)!

Root at $A[0]$

$\text{left}(i) = 2i + 1$

$\text{right}(i) = 2i + 2$

$\text{parent}(j) = \left\lfloor \frac{j - 1}{2} \right\rfloor$

The array can be larger than the heap.

In heapsort, the end of the array contains sorted elements (largest), beginning contains a heap.

```
heap | sorted
```

$\text{len}(A)$

$\text{heap-size}(A)$
Heapify: a procedure for fixing a heap with one invariant violation

heapify \((A, i)\)

exchange \(A[i]\) with larger of \(A[\text{left}(i)]\)

\(A[\text{right}(i)]\)

[or stop if \(A[i]\) larger]

heapify \((A, 3)\)

exchange \(A[3]\) with larger of \(A[\text{left}(3)]\)

\(A[\text{right}(3)]\)

if necessary

heapify \((A, 8)\)

no children \(\rightarrow\) nothing to do.

constant \# \text{operation at each level of the tree}

(possibly not every level; we may stop early)

\(\Rightarrow \Theta(\lg n)\) work
**Extract-Max**: return max element & delete it from the heap

```
return 16
```

```
heapify (A, 0)
```

```
put last leaf at A[0],
heap-size = heap-size -
```

```
DONE!
```

```
constant # operations + one call to heapify \(\Rightarrow \Theta \log n)\n```

work.