## InsertionSort

**Idea:**

<table>
<thead>
<tr>
<th></th>
<th>85</th>
<th>24</th>
<th>63</th>
<th>45</th>
<th>17</th>
<th>31</th>
<th>96</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>63</td>
<td>85</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>45</td>
<td>63</td>
<td>85</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
InsertionSort

```plaintext
for j ← 2 to n do
    key ← A[j]
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i = i − 1
    A[i + 1] ← key
```

What is the running time of InsertionSort?

$\Theta(n^2)$
InsertionSort

\[\text{for } j \leftarrow 2 \text{ to } n \text{ do}\]
  \[
  \text{key } \leftarrow A[j]
  \]
  \[
  i \leftarrow j - 1
  \]
\[\textbf{while } i > 0 \text{ and } A[i] > \text{key} \textbf{ do}\]
  \[
  A[i + 1] \leftarrow A[i]
  \]
  \[
  i = i - 1
  \]
  \[
  A[i + 1] \leftarrow \text{key}
  \]

▸ Why is InsertionSort correct?
▸ What is true \textit{every time} we check the \textbf{for} loop?
  (including the time we find \(j > n\) and stop)

\[A[1 \ldots j-1] \text{ is sorted and original element} \]
About that running time ...

- Why are we so concerned with worst case?

- Why not examine average case?

\[ E[\text{inv pairs}] = \frac{1}{2} \cdot \frac{n(n-1)}{2} \leq \frac{n(n-1)}{4} \]

Any is inverted
HeapSort

**Idea:** Use a max heap.

- Find max, put max at end
- Then second-max, etc.
- Use the yet-to-be-sorted array as max heap

**Heapify:** make array into max heap

- Idea 1: insert each into growing heap

\[
\text{for } i = 1 \ldots n \\
\text{H. insert } (A[i]) \} O(n) \text{ space}
\]

\[
\text{for } j = n \ldots 1 \\
A[j] = \text{H. extract Max } \} O(n \log n)
\]
Heapify: Better way

Treat array as heap. Where are leaf nodes?

What should we do with non-leaf nodes?

In which order?

\[ \frac{9}{2} : \text{no comparisons/swap} \]

\[ \frac{9}{4} : \text{one comparison/swap} \]

\[ \frac{9}{8} : \text{two} \]
How much work to heapify?

How many nodes?

\[ \frac{n}{2} \]

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

\[ \frac{n}{8} \]

\[ \frac{n}{16} \]

\[ \frac{n}{64} \]

How many steps (each)?

0

1

2

3

5
How long to heapify?

- The cost to insert varies by height.
- Node at height $h$ costs $\mathcal{O}(h)$.
  
  How many nodes at height $h$?
  
  $\frac{n}{2^{h+1}}$

- How many different height values are there? $\log n$

- Cost for total is:

  $\sum_{h=0}^{\lfloor \log n \rfloor} \left[ \frac{n}{2^{h+1}} \right] \mathcal{O}(h) = \mathcal{O} \left( n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$

  Using formula for infinite geometric series:

  $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1 - 1/2)^2} = 2$
And how, HeapSort