Selection Sort, Insertion Sort, and Bubble Sort all have a worst-case time of $O(n^2)$, making them impractical for large arrays. But they are easy to program, easy to debug. Insertion Sort also has good performance when the array is nearly sorted to begin with. But more sophisticated sorting algorithms are needed when good performance is needed in all cases for large arrays.

**Sorting Algorithms and Average Case Number of Comparisons**

- **Simple Sorts**
  - Insertion Sort
  - Selection Sort
  - Bubble Sort
  - $O(N^2)$

- **More Complex Sorts**
  - Heap Sort
  - Quick Sort
  - Merge Sort
  - $O(N \log N)$

**Heaps**

- A heap has two properties.
- Heaps can be implemented by arrays.
- Convert an array into a heap.
- A top down method
- A bottom up method

**Convert an Array into a Heap**

The Floyd Algorithm (bottom-up)

```
FloydAlgorithm(heap[])
for (i = index of last nonleaf; i>=0; i--) do
    restore the heap property for the tree whose root is heap[i] by calling movedown(heap, i, n-1);
endfor
```

**Heap Sort Approach**

First, make the unsorted array into a heap by satisfying the order property. Then repeat the steps below until there are no more unsorted elements.

- **Take the root (maximum) element off the heap** by swapping it into its correct place in the array at the end of the unsorted elements.
- **Reheap the remaining unsorted elements.** (This puts the next-largest element into the root position).
Example: After creating the original heap

Heap Sort: How many comparisons?

Quicksort Algorithm

Pick Pivot Element

Partitioning Array
**Partition Result**

```
<table>
<thead>
<tr>
<th>7</th>
<th>20</th>
<th>10</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
<td>[6]</td>
<td>[7]</td>
<td>[8]</td>
</tr>
</tbody>
</table>
```

<= data[pivot]  > data[pivot]

---

**Recursion: Quicksort Sub-arrays**

```
<table>
<thead>
<tr>
<th>7</th>
<th>20</th>
<th>10</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
<td>[6]</td>
<td>[7]</td>
<td>[8]</td>
</tr>
</tbody>
</table>
```

<= data[pivot]  > data[pivot]

---

**Quicksort Analysis**

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size \( n/2 \)
    2. Quicksort each sub-array
  - Depth of recursion tree? \( O(\log_2 n) \)
  - Number of accesses in partition? \( O(n) \)

---

**Quicksort Analysis**

- Assume that keys are random, uniformly distributed.
- Best case running time: \( O(n \log_2 n) \)
- Worst case running time?

---

**Quicksort: Worst Case**

- Assume first element is chosen as pivot.
- Assume we get array that is already in order:

```
| 2 | 4 | 10 | 12 | 13 | 50 | 57 | 63 | 100 |

pivot_index = 0
```

\[ [0] [1] [2] [3] [4] [5] [6] [7] [8] \]

too_big_index  too_small_index

---

**Quicksort: Worst Case**

```
| 2 | 4 | 10 | 12 | 13 | 50 | 57 | 63 | 100 |

pivot_index = 0
```

<= data[pivot]  > data[pivot]
Assume that keys are random, uniformly distributed.

Best case running time: $O(n \log_2 n)$

Worst case running time?

- Recursion:
  1. Partition splits array in two sub-arrays:
     - one sub-array of size 0
     - the other sub-array of size $n-1$
  2. Quicksort each sub-array
- Depth of recursion tree? $O(n)$
- Number of accesses per partition? $O(n)$

Worst case running time: $O(n^2)$!!!

What can we do to avoid worst case?