Outline

- Quadratic-Time Sorting
  - Bubble/Shaker Sort
  - Insertion Sort
  - Odd-Even Sort

- Linearithmic-Time Sorting
  - Heap Sort
  - Merge Sort
  - Quick Sort

- Conclusion

Check out this link for animation of various sorting algorithms:
http://cs.pugetsound.edu/~aasmith/sorters/
Quick Sort

- Input: two indices, start and end, of current list
  - If the list isn't empty \( (i.e., start < end) \):
  - Use last item in the list as the **pivot**
    - Rearrange (partition) all items in the list so that:
      - Items greater than it appear on its right
      - Items less than pivot appear on its left
  - Then recurse on both **unsorted** sublists
Pivot and Partitioning

- Trace for: 7 3 9 6 1 3 5, 6, and 3 3 0 8 1 7

```java
/**
 * Rearranges items in the list and returns the pivot's position
 */

private static int partition(int[] list, int start, int end) {
    int pivot = list[end];  // use last item in the sublist as pivot
    int i = start;
    for (int j = start; j < end; j++) {
        if (list[j] < pivot) {
            swap(list, i, j);
            i++;
        }
    }
    // swap pivot into proper position
    swap(list, i, end);
    return i;
}

private static void swap(int[] list, int i, int j) {
    int tmp = list[i];
    list[i] = list[j];
    list[j] = tmp;
}
```
Analysis of Partition

- What is the purpose of `partition()`?
  - It sorts the pivot in place, and returns the index of the pivot
  - Important byproduct: It split the list into two sublists
    - Items < pivot are swapped to its left
    - Items >= pivot are swapped to its right

- Complexity:
  \[ T_{\text{part}}(n) \approx 4n + 4 \]
  \[ T_{\text{part}}(n) = O(n) \]
QuickSort

```java
/**
 * QuickSort
 * @param list A list of ints
 * @return a sorted list of ints
 */
public static int[] quickSort(int[] list) {
    return quickSort(list, 0, list.length - 1);
}

private static int[] quickSort(int[] list, int start, int end) {
    if (start < end) {
        // partitions the list, and gets the position of the pivot
        int pivot_idx = partition(list, start, end);

        // sort the two other halves
        quickSort(list, start, pivot_idx - 1);
        quickSort(list, pivot_idx + 1, end);
    }
    return list;
}
```
Quick Sort Trace

List 8 6 9 5 1 3 2 4
Quick Sort Trace: First partition(list,0,7)

List: 8 6 9 5 1 3 2 4

Quick Sort Trace: First partition(list,0,7)
Quick Sort Trace: First partition(list,0,7)

List

\[\begin{array}{c}
1 & 6 & 9 & 5 & 8 & 3 & 2 & 4 \\
\end{array}\]

\(\text{swap}\)

\(\text{pivot}\)

\(\text{quicksort}(\text{list}, 0, 7)\)

\[\begin{array}{c}
1 & 6 & 9 & 5 & 8 & 3 & 2 & 4 \\
\end{array}\]
Quick Sort Trace: First partition(list,0,7)

List: 1 3 9 5 8 6 2 4

1. Partition the list into two parts: i = 0, j = 7.
2. Swap the pivot element with the element at index i.
3. Recursively call quicksort on the left part (list, 0, i-1) and the right part (list, i+1, j).

Quick Sort Trace: First partition(list,0,7)
Quick Sort Trace: First partition(list, 0, 7)

List

1 3 2 5 8 6 9 4

i
j
pivot
swap

(Stop!)

quicksort(list, 0, 7)

1 3 2 5 8 6 9 4
Quick Sort Trace: First partition(list,0,7)

List: 1 3 2 4 8 6 9 5

Quicksort(list, 0, 7)

Swap pivot into place!

i j
(Stop!)

swap pivot into place!
Quick Sort Trace (Omitting partition() Trace)

List

1 2 3 4 8 6 9 5

quicksort(list, 0, 7)

1 3 2 4 8 6 9 5

quicksort(list, 0, 2)

1 2 3
Quick Sort Trace (Omitting partition() Trace)

List

quicksort(list, 0, 7)

quicksort(list, 0, 2)

quicksort(list, 0, 0)
Quick Sort Trace (Omitting partition() Trace)

List: 1 2 3 4 8 6 9 5

- quicksort(list, 0, 7)
  - quicksort(list, 0, 2)
    - quicksort(list, 0, 0)
    - quicksort(list, 2, 2)
Quick Sort Trace (Omitting partition() Trace)

List: 1 2 3 4 5 6 9 8

quicksort(list, 0, 7)

quicksort(list, 0, 2)

quicksort(list, 4, 7)

quicksort(1 2 3)

quicksort(5 6 9 8)
Quick Sort Trace (Omitting partition() Trace)

List: 1 2 3 4 5 6 8 9

quicksort(list, 0, 7)

quicksort(list, 0, 2)  quicksort(list, 4, 7)

quicksort(list, 0, 2)

quicksort(list, 4, 3)

(base case: do nothing)
Quick Sort Trace (Omitting partition() Trace)

```
List: 1 2 3 4 5 6 8 9
```

```
quicksort(list, 0, 7)
quicksort(list, 0, 2)
quicksort(list, 4, 7)
quicksort(list, 5, 7)
```
Quick Sort Trace (Omitting partition() Trace)

List: 1 2 3 4 5 6 8 9

quicksort(list, 0, 7)

quicksort(list, 0, 2)

quicksort(list, 0, 2)

quicksort(list, 4, 7)

quicksort(list, 5, 7)

quicksort(list, 5, 5)

quicksort(list, 5, 5)
Quick Sort Trace (Omitting partition() Trace)

List: [1, 2, 3, 4, 5, 6, 8, 9]

- `quicksort(list, 0, 7)`
- `quicksort(list, 0, 2)`
- `quicksort(list, 4, 7)`
- `quicksort(list, 5, 5)`
- `quicksort(list, 5, 7)`
- `quicksort(list, 7, 7)`
Class Activity: QuickSort

- Given: A list of numbers
  - If start >= end
    - Hand list back to the student who gave it to you
  - Else:
    - Choose end element as pivot
    - Partition() the sublist indicated from start to end
    - Give left student the list, and:
      - Tell them your start and end = pivot_index - 1
        » Wait to receive the list back
    - Give right student the list, and:
      - Tell them start = pivot_index + 1 and your end
        » Wait to receive the list back
Another Trace!

List: 5 6 8 9

quicksort(list, 0, 3)

| 5 | 6 | 8 | 9 |

quicksort(list, 4, 3)

base case: do nothing
Another Trace!

List: 5 6 8 9

quicksort(list, 0, 3)

quicksort(list, 0, 2)  quicksort(list, 4, 3)

base case: do nothing
Another Trace!

List 5 6 8 9

quicksort(list, 0, 3)

5 6 8 9

quicksort(list, 0, 2)  quicksort(list, 4, 3)

5 6 8

quicksort(list, 3, 2)

base case: do nothing

base case: do nothing
Another Trace!

```
List  5 6 8 9

quicksort(list, 0, 3)
  └──quicksort(list, 0, 2)
     └──quicksort(list, 0, 1)
         5 6
     5 6 8
  5 6 8 9
  └──quicksort(list, 4, 3)
      base case: do nothing
     5 6 8 9
```

base case: do nothing
Another Trace!

```
List:  5 6 8 9

quicksort(list, 0, 3)
  └── quicksort(list, 0, 2)
    │   └── quicksort(list, 0, 1)
    │      └── quicksort(list, 2, 1)
    └── quicksort(list, 3, 2)
          └── quicksort(list, 4, 3)
                        base case: do nothing
```

base case: do nothing
Another Trace!

```
list = [5, 6, 8, 9]
```

```
quicksort(list, 0, 3)
quicksort(list, 0, 2)
quicksort(list, 0, 1)
quicksort(list, 0, 0)
quicksort(list, 4, 3)
quicksort(list, 3, 2)
quicksort(list, 2, 1)
```

- base case: do nothing
- base case: do nothing
- base case: do nothing
Food for Thought

- Notice how the performance of QuickSort is related to its "Recursion Tree" (results from the trace)

- The first trace, the tree looked more balanced
  - Like MergeSort's recursion tree (which is always balanced)

- The second trace, the tree looked like a linked list
  - Like a worst-case binary search tree...
Important: Performance of QuickSort depends on the pivot

- \texttt{partition()} selects pivot as the last item in the sublist

How to get trees that look like the first trace?

- Want pivot to be the median (or close to it!) item in the sublist
- Splits the list in half (or nearly half!)
How to get trees that look like the second trace?

- When **pivot** is sorted into one of the extreme ends
- Occurs when list is sorted in either ascending or descending order
Pivot: Connection to Performance

- **The fix:** What if we selected a *pivot* randomly?
  - Instead of always being the last item

RNG says: "Choose 5 as pivot"

```plaintext
partition(list, 0, 7)

\[
\begin{array}{cccccccc}
0 & 2 & 2 & 4 & 5 & 5 & 8 & 7 \\
\end{array}
\]

\[\Rightarrow\]

\[
\begin{array}{cccccccc}
0 & 2 & 2 & 4 & 7 & 5 & 8 & 5 \\
\end{array}
\]

\[\Rightarrow\]

\[
\begin{array}{cccccccc}
0 & 2 & 2 & 4 & 5 & 5 & 8 & 7 \\
\end{array}
\]

\[\Rightarrow\]

\[
\begin{array}{cccccccc}
0 & 2 & 2 & 4 & \quad & 5 & 8 & 7 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
5 & 8 & 7 & \quad & 0 & 2 & 2 & 4 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\sim n/2 & & & & \sim n/2 \\
\end{array}
\]

This is awesome!!!
The fix: What if we selected a pivot randomly?

- Instead of always being the last item

RNG says: "Choose 8 as pivot"

Swap pivot to the back

Or... you can still get pretty unlucky
Analysis of QuickSort

- Best case:
  - Median value of the sublist is always picked to be pivot
  - List split in two roughly equal halves

- Best Case Analysis:

\[
T(n) = \begin{cases} 
1, & \text{if } n \leq 1 \\
2T(n/2) + T_{\text{part}}(n), & \text{otherwise} 
\end{cases}
\]

\[
T(n) = O(n \log_2 n)
\]
Analysis of QuickSort

- Worst case:
  - List splits into lopsided "halves"
  - Extreme case is when list size only reduces by 1 item
  - *Can be mitigated by using random pivots*

- Complexity:

\[
T(n) = \begin{cases} 
1, & \text{if } n \leq 1 \\
T(n - 1) + T_{\text{part}}(n) + 1, & \text{otherwise}
\end{cases}
\]

\[
T(n) = O(n^2)
\]
Outline

- Quadratic-Time Sorting
  - Bubble/Shaker Sort
  - Insertion Sort
  - Odd-Even Sort

- Linearithmic-Time Sorting
  - Heap Sort
  - Merge Sort
  - Quick Sort

- Conclusion
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Shaker</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>OddEven</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Merge</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Heap</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quick</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
In Practice...

- The only quadratic-time algorithms in use is probably insertion sort
  - Good for semi-sorted data
  - Good when n is relatively small
  - Doesn't require auxiliary space

- Otherwise it's a toss-up: QuickSort vs. MergeSort
  - QuickSort doesn't require auxiliary space; merge sort does
  - MergeSort doesn't have a quadratic-time worst case