CSCI 261
Computer Science II
Motivation: Enable Binary Search

- Performing search over large collections of data...
  - Binary search is king. Precondition: list is sorted in ascending order

<table>
<thead>
<tr>
<th>List size: n</th>
<th>Binary Search: (O(\log_2 n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>24</td>
</tr>
<tr>
<td>100,000,000</td>
<td>27</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
</tr>
</tbody>
</table>
Outline

- Quadratic-Time Sorting
  - Bubble/Shaker Sort
  - Odd-Even Sort
  - Insertion 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/
Bubble Sort (Cont.)

- A list has: unsorted sublist (left) and a sorted sublist (right)
  - Variable i tracks the number of elements that are in sorted sublist
    - The sorted sublist is empty initially (i.e., i=0)
    - Keep track of index j of the current item in the unsorted sublist
    - Compare pair of items (at j-1 and at j) in the unsorted sublist
    - Swap them if left item is greater than right item
  - Continue until i increments to the size of the list
    - That is, until unsorted sublist becomes empty
Bubble Sort Example

i = 0

19 12 13 7

j-1 j

12 19 13 7

j-1 j

12 13 19 7

j-1 j

12 13 7 19

j-1 j (stop!)

i = 1

12 13 7 19

j-1 j

12 13 7 19

j-1 j

12 7 13 19

j-1 j

12 7 13 19

j-1 j (stop!)

i = 2

12 7 13 19

j-1 j

7 12 13 19

j-1 j

7 12 13 19

j-1 j (stop!)

i = 3

7 12 13 19

j-1 j (stop!)

i = 4 (done!)
Bubble Sort Algorithm

- Naive BubbleSort algorithm
  - Why it is naive?

```java
public static int[] bubbleSort(int[] list) {
    // counts the number of item in the sorted sublist
    for (int i = 0; i < list.length; i++)
        for (int j = 1; j < list.length - i; j++) {
            // compare each pair of items
            if (list[j-1] > list[j]) {
                // need to rise list[j-1] up the list (bubbling)
                // swap the two neighboring items
                swap(list, j-1, j);
            }
        }
    return list;
}

private static void swap(int[] list, int i, int j) {
    int tmp = list[i];
    list[i] = list[j];
    list[j] = tmp;
}
```
Bubble Sort Example 2

i = 0

\[
\begin{array}{cccc}
19 & 7 & 12 & 13 \\
\end{array}
\]

j-1 j

i = 1

\[
\begin{array}{cccc}
7 & 12 & 13 & 19 \\
\end{array}
\]

j-1 j

i = 2

\[
\begin{array}{cccc}
7 & 12 & 13 & 19 \\
\end{array}
\]

j-1 j

(j stop!)

i = 3

\[
\begin{array}{cccc}
7 & 12 & 13 & 19 \\
\end{array}
\]

j-1 j

(j stop!)

i = 4 (done!)
Bubble Sort Example 2*

i = 0

i = 1

i = 2 (done!)

Should end the algorithm now!

(How do we know the list was already sorted?)

Check out this link for animation:
http://cs.pugetsound.edu/~aasmith/sorters/
Bubble Sort (Optimized)

- Optimization:
  - If a pass of the unsorted sublist didn't swap, it implies the list is sorted.
    - Stop the algorithm if a swap was not made in a pass!

```java
public static int[] bubbleSort(int[] list) {
    boolean swapped = true;
    for (int i = 0; swapped && i < list.length; i++) {
        swapped = false; // assume no swaps will happen this time
        for (int j = 1; j < list.length - i; j++) {
            if (list[j-1] > list[j]) {
                swap(list, j-1, j);
                swapped = true; // we swapped; more passes needed
            }
        }
    }
    return list;
}

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

Pros:

• Optimized for mostly-sorted or already-sorted lists

Cons:

• Considered the slowest sorter in the avg/worst case
• Input list is seldom mostly- or already-sorted
  
  - Exception: One badly placed element screws up optimization (see animation)

<table>
<thead>
<tr>
<th></th>
<th>Bubble Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Worst/Avg Case</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

Click on this link for a good time:
https://www.youtube.com/watch?v=koMpGeZpu4Q
Outline

- Quadratic-Time Sorting
  - Bubble/Shaker Sort
  - Odd-Even Sort
  - Insertion 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/
(Cocktail) Shaker Sort

- **Shaker Sort** is a further optimization on Bubble Sort

- Recall: One badly placed item screws up Bubble Sort's optimization
  - (e.g., a small item toward the end of the list)

- Shaker sort fixes this by alternating bubble sort left and right each pass
Shaker Sort Example

i = 0

1  5  7  0

i = 1

1  5  0  7

i = 2

0  1  5  7

j-1  j  j-1  j

(No swaps! Done)
Shaker Sort Algorithm

```java
public static int[] oddEvenSort(int[] list) {
    boolean swapped = true; // lets us short-circuit the loop if no swaps were made
    for (int i = 0; swapped && i < list.length; i++) {
        swapped = false;
        if (i % 2 == 0) { // even pass
            for (int j = 0; j < list.length - 1; j += 2) {
                if (list[j] > list[j + 1]) {
                    swap(list, j, j + 1);
                    swapped = true;
                }
            }
        } else { // odd pass
            for (int j = 1; j < list.length - 1; j += 2) {
                if (list[j] > list[j + 1]) {
                    swap(list, j, j + 1);
                    swapped = true;
                }
            }
        }
    }
    return list;
}
```
Analysis of Shaker Sort

- **Pros:**
  - Optimized for mostly-sorted or already-sorted lists
  - Fixes the "unlucky placement" problem of bubble sort

- **Cons:**
  - Just bubble sort in disguise, so same complexity

<table>
<thead>
<tr>
<th></th>
<th>Bubble Sort</th>
<th>Shaker Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Worst/Avg Case</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

Check out this link for animation:
[http://cs.pugetsound.edu/~aasmith/sorters/](http://cs.pugetsound.edu/~aasmith/sorters/)
Outline

- Quadratic-Time Sorting
  - Bubble/Shaker Sort
  - Odd-Even Sort
  - Insertion Sort
- Linearithmic-Time Sorting
  - Heap Sort
  - Merge Sort
  - Quick Sort
- Conclusion
Odd-Even Sort

- **Odd-Even Sort** is a variant of Bubble Sort

- Given a list of $n$ items, make $n$ passes:
  - On an even pass:
    - Compare each even items to its right neighbor, and swap if out of order
  - On an odd pass:
    - Compare each odd items to its left neighbor, and swap if out of order

- Demonstration on: [12, 41, 7, 0, 33]
Odd-Even Sort Analysis

- Still $O(n^2)$ worst case
  - But that's assuming it's executed on a single CPU...

- Why is odd-even sort used?
  - Most CPUs today have multiple "cores"
    - Think of a "core" as an independent CPU
  - Speed up a program by harnessing multiple cores simultaneously!
    - Share the sorting work across all cores
Multi-Core Processors

- Can run multiple processes (programs in execution) simultaneously!
Multi-Core Processors

- Or one process can spawn multiple "threads" of work ...
Odd-Even Sort (single thread)

Odd-Even Sort is a sorting algorithm that repeatedly sorts pairs of elements in an array. The algorithm consists of two phases: the odd phase and the even phase.

- **Odd Phase**: Elements at odd indices are compared and swapped if they are in the wrong order. This phase can be represented by the sequence $i = 0, 2, 4, \ldots$.
- **Even Phase**: Elements at even indices are compared and swapped if they are in the wrong order. This phase can be represented by the sequence $i = 1, 3, 5, \ldots$.

The process continues until the array is sorted.

The diagrams illustrate the sorting process for an array of numbers. Each diagram shows the array before and after the sorting phase for a particular index $i$. The arrows indicate the elements that are swapped during that phase.
Odd-Even is easily parallelizable, because the swaps are independent of each other!
Performance of Parallel Odd-Even

n = 10000
Odd-Even sort took: 68.970471ms
Parallel OddEven sort took: 1780.808583ms

97x slowdown!!!
(Yikes!)

n = 100000
Odd-Even sort took: 6547.624747ms
Parallel OddEven sort took: 18145.69779ms

64x slowdown!!! (Better, but still yikes!)

n = 1000000
Odd-Even sort took: 660046.62797ms
Parallel OddEven sort took: 540041.537718ms

1.22x speedup!!!
Odd-Even Sort Analysis Again

- Undea "ideal conditions," parallel odd-even sort is:
  - \( T(n) = O(n^2/m) \) where \( m \) is the number of threads
  - There's no such thing as ideal conditions because of:
    - Overhead of creating "threads"
    - Overhead of synchronizing threads
    - Overhead to splitting the work among threads
    - Other overheads...

- And \( m \) is generally small unless you have access to a supercomputer
Outline

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

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

- Conclusion
Insertion Sort

- Suppose you were dealt \( n \) playing cards...

- *Insertion sort* simulates what you would do to sort the cards dealt to you
  - Left hand holds *sorted* pile (initially empty)
  - Right hand holds *unsorted* pile (initially all cards)
  - Take a card from right hand, and insert in place in the left hand
At any point, a list has

- A sorted sublist (blue) and an unsorted sublist (red)
  - right is the start position of the unsorted sublist (initially 1)
  - left is the end position of the sorted sublist. Always starts at right-1

**Algorithm:** While right has not passed end of list:

- Consider the item at right
- Starting from left (and descending) find spot in the sorted sublist for a place to store this item
  - May need to shift sorted item to the right, then insert the item in place
Insertion Sort Example

R = 1

R = 2

R = 3

R = 4 (Done)
Insertion Sort (Cont.)

```java
public static int[] insertionSort(int[] list) {
    for (int right = 1; right < list.length; right++) {
        int toBeInserted = list[right];

        // Need to find the place to put toBeInserted
        int left = right - 1;
        while (left >= 0 && toBeInserted < list[left]) {
            // Shift items to the right to make space for it
            list[left + 1] = list[left];
            left--;
        }

        // Found the place where it should go!
        list[left + 1] = toBeInserted;
    }
    return list;
}
```

Check out this link for animation:
http://cs.pugetsound.edu/~aasmith/sorters/
Insertion Sort Example (Best Case)

R = 1

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>13</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(Skip the inner loop)

R = 2

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>13</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(Skip the inner loop)

R = 3

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>13</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(Skip the inner loop)

R = 4

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>13</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(Done in outer loop)
Analysis of Insertion Sort

- Most of the work done in determining the place to insert (inner loop)

- Pros:
  - Optimized for mostly-sorted or already-sorted lists
  - Poorly placed item in a mostly-sorted list doesn't screw up insertion sort

- Cons:
  - Still an average/worst case $O(n^2)$

<table>
<thead>
<tr>
<th></th>
<th>Insertion Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Case</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Worst/Avg Case</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

Check out this link for animation:
http://cs.pugetsound.edu/~aasmith/sorters/