CSCI 261
Computer Science II
Search is one of the most frequently-used operations in all of CS

- "Make the common case fast!"
- To make search fast, we need to learn how to sort a list

After all, why do you sort things?

- Your wardrobe...
- Hand of playing cards when dealt...
- Contacts in your phone...
- Organizing books on your shelf...
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>Worst Case Time Complexity</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

<table>
<thead>
<tr>
<th>i = 0</th>
<th>i = 1</th>
<th>i = 2</th>
<th>i = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="State 0" /></td>
<td><img src="image2" alt="State 1" /></td>
<td><img src="image3" alt="State 2" /></td>
<td><img src="image4" alt="State 3" /></td>
</tr>
</tbody>
</table>

**CSCI 261: Computer Science II - 10 - Efficient Sorting Algorithms**
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

1. i = 0
   - 19  7  12  13
   - j-1  j

2. i = 1
   - 7  19  12  13
   - j-1  j

3. i = 2
   - 7  12  13  19
   - j-1  j

4. i = 3
   - 7  12  13  19
   - j-1  j
   - (stop!)

5. i = 4 (done!)
   - 7  12  13  19
Bubble Sort Example 2*

i = 0

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

j-1  j

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

\( \rightarrow \)

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

j-1  j

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

j-1  j

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

j-1  j

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

j (stop!)

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

j (stop!)

i = 1

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

j-1  j

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

j-1  j

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

j-1  j

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

j (stop!)

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

j (stop!)

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](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
```

**j-1**  
**j**

```
1 5 7 0
```

**j-1**  
**j**

```
1 5 7 0
```

**j-1**  
**j**

```
1 5 0 7
```

**j**  
**j+1**

```
1 0 5 7
```

**i = 1**

```
0 1 5 7
```

**j-1**  
**j**

```
0 1 5 7
```

**j-1**  
**j**

```
0 1 5 7
```

**j**  
**j+1**

```
0 1 5 7
```

**i = 2**

```
0 1 5 7
```

**j-1**  
**j**

```
0 1 5 7
```

**j-1**  
**j**

```
0 1 5 7
```

(No swaps! Done)
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>
</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/
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 another variant of Bubble Sort
  - Why? Obama already told us that he wouldn't use bubble sort
    - It's good for multi-core CPUs!

- 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
Odd-Even Sort Example

```
Odd-Even Sort Example

8 5 1 3 0 5 8 0
0 1 2 3 4 5 6 7

i=0
5 8 1 3 0 5 0 8
0 1 2 3 4 5 6 7

i=1
5 1 8 0 3 0 5 8
0 1 2 3 4 5 6 7

i=2
1 5 0 8 0 3 5 8
0 1 2 3 4 5 6 7

i=3
1 0 5 0 8 3 5 8
0 1 2 3 4 5 6 7

i=4
0 1 0 5 3 8 5 8
0 1 2 3 4 5 6 7

i=5
0 0 1 3 5 5 8 8
0 1 2 3 4 5 6 7

i=6
0 0 1 3 5 5 8 8
0 1 2 3 4 5 6 7

i=7
0 0 1 3 5 5 8 8
0 1 2 3 4 5 6 7
```
Odd-Even 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;
}

private static void swap(int[] list, int i, int j) {
    int tmp = list[i];
    list[i] = list[j];
    list[j] = tmp;
}
```
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"...
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
Sorted? true

FastOddEven sort took: 1780.808583ms
Sorted? true

----------------------------------------
n = 100000
Odd-Even sort took: 6547.624747ms
Sorted? true

FastOddEven sort took: 18145.69779ms
Sorted? true

----------------------------------------
n = 1000000
Odd-Even sort took: 660046.62797ms
Sorted? true

FastOddEven sort took: 540041.537718ms
Sorted? true

97x slowdown!!!
(Yikes, wasn't it supposed to beat bubble sort?)

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

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

- Under ideal conditions, 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...

- But $m$ is generally small (i.e., a constant) unless you have access to 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
Insertion Sort (Cont.)

- 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 fallen off the 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)
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;
}
Insertion Sort Example (Best Case)

R = 1

(R = 1)

(R = 2)

(R = 3)

(R = 4)

(Done in outer loop)
Analysis of Insertion Sort

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

- Pros:
  - Also optimized for mostly-sorted or already-sorted lists
    - In practice, faster than bubble sort (not in theory)
  - 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/]