

Sorting I

Michael C. Hackett
Assistant Professor, Computer Science

Lecture Topics

- Bubble Sort
 - Bubble Sort Improvements
 - Cocktail Shaker Sort
- Insertion Sort
- Selection Sort
- Comparable Interface (Java)
- Comparator Interface (Java)

Sorting Algorithms

- A **sorting algorithm** is an algorithm that organizes a sequence of data (like an array) into some type of order.
 - Usually ascending or descending order.
- An **ascending sort** means arranging the values from smallest to largest.
 - {3, 4, 2, 5, 1} {1, 2, 3, 4, 5}
- A **descending sort** means arranging the values from largest to smallest.
 - {3, 4, 2, 5, 1} {5, 4, 3, 2, 1}

Sorting Algorithms

- A **comparative sorting algorithm** sorts the contents of a sequence through a repeated process of comparing values in the sequence.
 - Usually less-than or greater-than comparisons.
- A **non-comparative sorting algorithm** sorts the contents of a sequence using characters from the values to be sorted.
 - Non-comparative sorting is an upcoming lecture.

Bubble Sort Algorithm

- The Bubble Sort algorithm is a comparative sorting algorithm.
- Neighboring pairs of values are compared and swapped so that they are in the correct order.
- The algorithm repeats this process n -number of times, where n is the length of the array.

Bubble Sort Algorithm

- Here is a link to an online tool that visualizes the Bubble Sort sorting an array of numbers.

<https://www.sortvisualizer.com/bubblesort/>

Bubble Sort

```
public void bubbleSort(int[] a[]) {  
    for(int i = 0; i < a.length; i++) {  
        for(int j = 1; j < a.length; j++) {  
            if(a[j-1] > a[j]) {  
                int temp = a[j-1];  
                a[j-1] = a[j];  
                a[j] = temp;  
            }  
        }  
    }  
}
```

Bubble Sort Algorithm Improvements

- One variation of the Bubble Sort allows it to end early if the sequence is in order.
 - The algorithm begins by assuming the sequence is in order.
 - It traverses the sequence, making any necessary swaps.
 - If it made at least one swap, the process repeats.
 - If it made no swaps, the algorithm terminates.

Bubble Sort (Improved 1)

```
public void bubbleSort(int[] a[]) {  
    boolean sorted;  
    do {  
        sorted = true;  
        for(int i = 0; i < a.length-1; i++) {  
            if(a[i] > a[i+1]) {  
                int temp = a[i];  
                a[i] = a[i+1];  
                a[i+1] = temp;  
                sorted = false;  
            }  
        }  
    } while(!sorted);  
}
```

Bubble Sort Algorithm Improvements

- Another improvement of the original Bubble Sort allows it to only traverse up to the last element that was sorted.
 - The algorithm is almost identical to the original, but the inner loop only continues while the inner loop's counter is $< \text{length} - \text{the outer loop's counter}$
 - For example, when the sequence is half-way sorted there is no need to check the second half of the sequence, since those elements are already in order.
 - The original Bubble Sort (and the alternative implementation previously shown) unnecessarily checks those already sorted elements.

Bubble Sort (Improved 2) (C++ Function)

```
public void bubbleSort(int[] a) {  
    for(int i = 0; i < a.length; i++) {  
        for(int j = 1; j < a.length - i; j++) {  
            if(a[j-1] > a[j]) {  
                int temp = a[j-1];  
                a[j-1] = a[j];  
                a[j] = temp;  
            }  
        }  
    }  
}
```

Bubble Sort Variations Compared

- Original Bubble Sort

$$\underline{\text{Total Comparisons}} = n * (n-1)$$

- Length of 5: Total Comparisons = $5 * (5-1) = 20$
- Length of 10: Total Comparisons = $10 * (10-1) = 90$

- Bubble Sort Improved 1

$$n-1 \leq \underline{\text{Total Comparisons}} \leq n * (n-1)$$

- Length of 5: $4 \leq \text{Total Comparisons} \leq 20$
- Length of 10: $9 \leq \text{Total Comparisons} \leq 90$

- Bubble Sort Improved 2

$$\underline{\text{Total Comparisons}} = \sum_{i=1}^{n-1} i$$

- Length of 5: Total Comparisons = $1 + 2 + 3 + 4 = 10$
- Length of 10: Total Comparisons = $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45$

Bubble Sort Variations Compared

- The following examples use small sequences, but should still illustrate the amount of work required by each algorithm.
 - Each assumes sorting in ascending order.
- {5, 1, 2, 3, 4}
 - Original Bubble Sort: 20 comparisons
 - **Bubble Sort Improved 1: 8 comparisons**
 - Bubble Sort Improved 2: 10 comparisons

Bubble Sort Versions Compared

- {1, 2, 3, 4, 5} (Already sorted)
 - Original Bubble Sort: 20 comparisons
 - **Bubble Sort Improved 1: 4 comparisons**
 - Bubble Sort Improved 2: 10 comparisons
- {5, 4, 3, 2, 1} (Worst Case)
 - Original Bubble Sort: 20 comparisons
 - Bubble Sort Improved 1: 20 comparisons
 - **Bubble Sort Improved 2: 10 comparisons**

Cocktail Shaker Sort Algorithm

- The Cocktail Shaker Sort algorithm is another variation of the Bubble Sort algorithm.
- Neighboring pairs of values in sequence are compared and then swapped, moving the largest values to the right.
 - If performing an ascending sort.
- But then it goes backwards, moving the smallest values to the left.

<https://www.sortvisualizer.com/shakersort/>

Cocktail Shaker Sort

```
public void cocktailSort(int[] a) {  
    boolean sorted;  
    int i;  
    do {  
        Forward {  
            sorted = true;  
            for(int i = 1; i < a.length; i++) {  
                if(a[i-1] > a[i]) {  
                    int temp = a[i-1];  
                    a[i-1] = a[i];  
                    a[i] = temp;  
                    sorted = false;  
                }  
            }  
        }  
        if(sorted) {  
            break;  
        }  
        Reverse {  
            sorted = true;  
            for(int j = a.length-1; j > 0; j--) {  
                if(a[j] < a[j-1]) {  
                    int temp = a[j-1];  
                    a[j-1] = a[j];  
                    a[j] = temp;  
                    sorted = false;  
                }  
            }  
        }  
    } while(!sorted);  
}
```


Cocktail Shaker Sort Algorithm

- The Cocktail Shaker Sort algorithm behaves similarly to the (Improved 1) Bubble Sort algorithm.
 - Terminates early if it didn't make any swaps
- A variation of this algorithm (which sets high and low boundaries) skips the already sorted values at the beginning and end of the sequence.
 - Like the behavior of the (Improved 2) Bubble Sort algorithm

Cocktail Shaker Sort (Improved)

```
public void cocktailSort(int[] a) {  
    int lowBound = 0;  
    int highBound = a.length - 1;  
  
    while(lowBound < highBound) {  
        Forward {  
            for(int i = lowBound; i < highBound; i++) {  
                if(a[i] > a[i+1]) {  
                    int temp = a[i+1];  
                    a[i+1] = a[i];  
                    a[i] = temp;  
                }  
            }  
            highBound--;  
        }  
  
        Reverse {  
            for(int i = highBound; i > lowBound; i--) {  
                if(a[i-1] > a[i]) {  
                    int temp = a[i-1];  
                    a[i-1] = a[i];  
                    a[i] = temp;  
                }  
            }  
            lowBound++;  
        }  
    }  
}
```

Cocktail Shaker Sort Variations Compared

- Original Cocktail Shaker Sort

Odd length: $n-1 \leq \text{Total Comparisons} \leq n * (n-1)$

Even length: $n-1 \leq \text{Total Comparisons} \leq (n+1) * (n-1)$

- Length of 5: $4 \leq \text{Total Comparisons} \leq 20$
- Length of 10: $9 \leq \text{Total Comparisons} \leq 99$

- Cocktail Shaker Sort Improved

Total Comparisons = $\sum_{i=1}^{n-1} i$

- Length of 5: Total Comparisons = $1 + 2 + 3 + 4 = 10$
- Length of 10: Total Comparisons = $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45$

Cocktail Shaker Sort Variations Compared

- The following examples use small sequences, but should still illustrate the amount of work required by each algorithm.
 - Each assumes sorting in ascending order.
- {5, 1, 2, 3, 4}
 - **Original Cocktail Shaker Sort: 8 comparisons**
 - Cocktail Shaker Sort Improved: 10 comparisons

Cocktail Shaker Sort Versions Compared

- {1, 2, 3, 4, 5} (Already sorted)
 - **Original Cocktail Shaker Sort: 4 comparisons**
 - Cocktail Shaker Sort Improved: 10 comparisons
- {5, 4, 3, 2, 1} (Worst Case)
 - Original Cocktail Shaker Sort: 20 comparisons
 - **Cocktail Shaker Sort Improved: 10 comparisons**

Selection Sort Algorithm

- The Selection Sort algorithm is a comparative sorting algorithm.
- The algorithm divides the array into sorted and unsorted partitions.
- The algorithm searches (linearly) through the unsorted partition to find the smallest value.
 - If performing an ascending sort.
- The smallest value found is swapped to the beginning of the unsorted partition, thus extending the sorted partition by 1.

Selection Sort Algorithm

- Here is a link to an online tool that visualizes the Selection Sort sorting an array of numbers.

<https://www.sortvisualizer.com/selectionsort/>

Selection Sort

```
public void selectionSort(int[] a) {  
    for(int i = 0; i < a.length-1; i++) {  
        int smallest = i;  
        for(int j = i+1; j < a.length; j++) {  
            if(a[j] < a[smallest]) {  
                smallest = j;  
            }  
        }  
        if(smallest != i) {  
            int temp = a[smallest];  
            a[smallest] = a[i];  
            a[i] = temp;  
        }  
    }  
}
```


Insertion Sort Algorithm

- The Insertion Sort algorithm is a comparative sorting algorithm.
- Like the Selection Sort, it divides an array into sorted and unsorted partitions.
- The algorithm takes the first element in the unsorted portion, and keeps swapping backwards until it finds a smaller value in the sorted portion.
 - If performing an ascending sort

Insertion Sort Algorithm

- Here is a link to an online tool that visualizes the Insertion Sort sorting an array of numbers.

<https://www.sortvisualizer.com/insertionsort/>

Insertion Sort

```
public void insertionSort(int[] a) {  
    for(int i = 1; i < a.length; i++) {  
        int value = a[i];  
        int j = i-1;  
        while(j >= 0 && a[j] > value) {  
            a[j+1] = a[j];  
            j--;  
        }  
        a[j+1] = value;  
    }  
}
```

Selection Sort and Insertion Sort Compared

- Selection Sort

$$\text{Total Comparisons} = \sum_{i=1}^{n-1} i$$

- Length of 5: Total Comparisons = $1 + 2 + 3 + 4 = 10$
- Length of 10: Total Comparisons = $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45$

- Insertion Sort

$$n-1 \leq \text{Total Comparisons} \leq \sum_{i=1}^{n-1} i$$

- Length of 5: $4 \leq \text{Total Comparisons} \leq 10$
- Length of 10: $10 \leq \text{Total Comparisons} \leq 45$

Selection Sort and Insertion Sort Compared

- The following examples use small sequences, but should still illustrate the amount of work required by each algorithm.
 - Each assumes sorting in ascending order.
- {5, 1, 2, 3, 4}
 - Selection Sort: 10 comparisons
 - **Insertion Sort: 7 comparisons**

Selection Sort and Insertion Sort Compared

- {1, 2, 3, 4, 5} (Already sorted)
 - Selection Sort: 10 comparisons
 - **Insertion Sort: 4 comparisons**
- {5, 4, 3, 2, 1} (Worst Case)
 - **Selection Sort: 10 comparisons**
 - **Insertion Sort: 10 comparisons**

These Algorithms Compared

- {5, 1, 2, 3, 4}
 - **Insertion Sort: 7 comparisons**
 - Original Cocktail Shaker Sort: 8 comparisons
 - Bubble Sort Improved 1: 8 comparisons
 - Selection Sort: 10 comparisons
 - Cocktail Shaker Sort Improved: 10 comparisons
 - Bubble Sort Improved 2: 10 comparisons
 - Original Bubble Sort: 20 comparisons

These Algorithms Compared

- {1, 2, 3, 4, 5} (Already sorted)
 - **Insertion Sort: 4 comparisons**
 - **Original Cocktail Shaker Sort: 4 comparisons**
 - **Bubble Sort Improved 1: 4 comparisons**
 - Selection Sort: 10 comparisons
 - Cocktail Shaker Sort Improved: 10 comparisons
 - Bubble Sort Improved 2: 10 comparisons
 - Original Bubble Sort: 20 comparisons

These Algorithms Compared

- {5, 4, 3, 2, 1} (Worst Case)
 - **Insertion Sort: 10 comparisons**
 - **Cocktail Shaker Sort Improved: 10 comparisons**
 - **Bubble Sort Improved 2: 10 comparisons**
 - **Selection Sort: 10 comparisons**
 - Original Cocktail Shaker Sort: 20 comparisons
 - Original Bubble Sort: 20 comparisons
 - Bubble Sort Improved 1: 20 comparisons
- Of *these* algorithms, **insertion sort will generally be the best choice.**

Sorting Arrays in Java

```
int[] a = {5, 4, 3, 2, 1};  
Arrays.sort(a);
```

- This is helpful (and quick) way to use sorting in a Java program.
- Though, how would this method know how to sort custom objects that you defined?
- If the object has multiple fields, how will the method know which one to use in its sorting process?

Comparable Interface

- Java's Comparable interface contains one abstract method that it requires implementing classes to define.
 - Syntax: **public class MyObj implements Comparable<MyObj>**
- Required method:
public int compareTo(MyObj x)
 - Returns:
 - A positive number if $\geq x$
 - A negative number if $\leq x$
 - 0 if $= x$

Comparable Interface

```
public int compareTo(MyObj x) {  
    return this.field - x.field;  
}
```

- If this object's field was 7 and x's field was 5, the result is 2
 - Positive, so this object is "greater than" x
- If this object's field was 4 and x's field was 8, the result is -4
 - Negative, so this object is "less than" x
- If this object's field was 6 and x's field was 6, the result is 0
 - Zero, so this object is "equal to" x

Comparable Interface

- If the object is implementing Comparable, then methods like `Arrays.sort()` will be able to sort an array of these objects.
 - It knows to use the object's `compareTo` methods, which defined how the objects are to be sorted.
- A drawback is this only allows us one way to sort the objects.
 - There may be multiple fields we wish to sort the objects by.
 - We may want to sort by one field but later sort the same objects using a different field.

Comparator Interface

- The Comparator interface is used to make multiple *comparators*
 - Classes that implement a method to compare two objects.
 - Syntax: **public class myObjComp implements Comparator<MyObj>**
- Required method:
public int compare(MyObj x1, MyObj x2)
 - Returns:
 - A positive number if $x1 \geq x2$
 - A negative number if $x1 \leq x2$
 - 0 if $x1 == x2$

Comparator Interface

```
public class myObjComp implements Comparator<MyObj> {  
    public int compare(MyObj x1, MyObj x2) {  
        return x1.field - x2.field;  
    }  
}
```

- If this x1's field was 7 and x2's field was 5, the result is 2
 - Positive, so x1 is "greater than" x2
- If this x1's field was 4 and x2's field was 8, the result is -4
 - Negative, so x1 is "less than" x2
- If this x1's field was 6 and x'2s field was 6, the result is 0
 - Zero, so x1 is "equal to" x2

Comparator Interface

- We can have many Comparators that handle different comparisons:

```
public class myObjComp implements Comparator<MyObj> {  
    public int compare(MyObj x1, MyObj x2) {  
        return x1.field - x2.field;  
    }  
}
```

```
public class myObjComp2 implements Comparator<MyObj> {  
    public int compare(MyObj x1, MyObj x2) {  
        return x1.field2 - x2.field2;  
    }  
}
```


Comparator Interface

- When calling a method like `Arrays.sort()` we will need to provide it both the array to sort and a comparator to tell the method how to compare/sort the objects.

```
Arrays.sort(myobjs, new MyObjComp());  
Arrays.sort(myobjs, new MyObjComp2());
```

- The first sort would use the `MyObjComp` comparator.
- The second sort would use the `MyObjComp2` comparator.