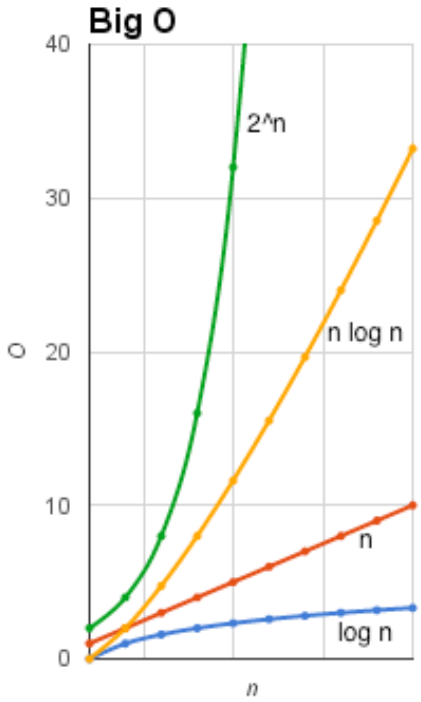
CECS 277 – Lecture 6 – Sorting

Sorting Algorithms – There are many different types of algorithms that will sort an array or collection of items (ints, doubles, strings, objects, etc.). Some algorithms are faster, some are more efficient, and some are just easier to program. Most algorithms are compared by the amount of time it would take to process the collection given the worst-case scenario. Big-O notation allows us to express the efficiencies of different algorithms so that they can be easily compared.

Figures: Ranking of some Big-O notation times.

Big-O Notation	Name
O(1)	Constant
O(log(n))	Log
O(n)	Linear
O(n log(n))	Log-Linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
O(n!)	Factorial



Compareable Interface – Any object that implements Comparable can be sorted. The class must override the compareTo() method so that two objects of that type can be compared. Common classes that implement Comparable: Boolean (False < True), Double, Integer (numerical order), Character, String (alphabetical order), etc.

The compareTo() Method – compares the implicit and explicit parameters of the method. CompareTo() returns a negative integer if the two objects are in order, a zero if they are equal, and a positive integer if they are out of order.

```
public class Student implements Comparable<Student> {
      private int idNum;
      public Student( int i ) {
            idNum = i;
      }
      @Override
      public int compareTo( Student s ) {
            if(this.idNum == s.idNum) { return 0; }
            if(this.idNum <  s.idNum) { return -1;}
            if(this.idNum >  s.idNum) { return 1; }
      }
}
```

Selection Sort – An intuitive sorting algorithm that is easy to implement. Its complexity is on the order of $O(n^2)$ time.

Algorithm:

1. Find the minimum value in the list.
2. Swap with the element in the first position.
3. Move to the next position and repeat for the rest of the list.

Given a list: `list => [4, 2, 6, 5, 1, 3, 9, 0, 7, 8]`

On the first pass of the for loop, lowest is initialized to 0, and then the value stored at index 0 (4) is compared with the other elements of the array until the lowest value is found at index 7 (0). The values of index 0 and index 7 are swapped resulting in:

```
list => [0, 2, 6, 5, 1, 3, 9, 4, 7, 8]
```

On the second pass, lowest is initialized to 1, and then the value stored at index 1 (2) is compared with the other elements of the array until the lowest value is found at index 4 (1). The values are swapped: `list => [0, 1, 6, 5, 2, 3, 9, 4, 7, 8]`

This process continues until the outer for loop finishes and the list is sorted, in all, it will take 10 passes through the loop to finish sorting.

```
public static void selectionSort(ArrayList<Student> list) {
     for( int i = 0; i < list.size(); i++ ) {
          int low = i;
          for( int j = i + 1; j < list.size(); j++ ) {
               if(list.get(j).compareTo(list.get(low))<0) {
                    low = j;
               }
          }
          Student swap = list.get( i );
          list.set( i, list.get( low ) );
          list.set( low, swap );
     }
}
```

Merge Sort – A recursive divide and conquer algorithm. The code is more complex, but it is a much faster algorithm than Selection Sort with a worst case of O(n log (n)).

Algorithm –

1. Recursively split the list into two halves until there is one element in each list.
2. Repeatedly sort and merge the lists together until there is only one sorted list.

The list is broken up into halves until each list contains one element.

```
lists => [4], [2], [6], [5], [1], [3], [9], [0], [7], [8]
```

The lists are then merged back together in sorted order.

```
lists => [2, 4], [6], [1, 5], [3, 9], [0], [7, 8]
lists => [2, 4], [1, 5, 6], [3, 9], [0, 7, 8]
lists => [1, 2, 4, 5, 6], [0, 3, 7, 8, 9]
sorted=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
public static void mergeSort( ArrayList<Student> list ) {
      ArrayList<Student> left = new ArrayList<Student>();
      ArrayList<Student> right = new ArrayList<Student>();
      int center;
      if( list.size() > 1 ) {
            center = list.size() / 2;
            for( int i = 0; i < center; i++ ) {
                  left.add( list.get( i ) );
            }
            for( int i = center; i < list.size(); i++ ) {
                  right.add( list.get( i ) );
            }
            mergeSort( left );
            mergeSort( right );
            merge( left, right, list );
      }
}
private static void merge( ArrayList<Student> left,
      ArrayList<Student> right, ArrayList<Student> list ) {
      int iLeft = 0;
      int iRight = 0;
      int iList = 0;
      while( iLeft < left.size() && iRight < right.size() ) {
            if((left.get(iLeft).compareTo(right.get(iRight)))<0){
                  list.set( iList, left.get( iLeft ) );
                  iLeft++;
            } else {
                  list.set( iList, right.get( iRight ) );
                  iRight++;
            }
            iList++;
     }
      if ( iLeft >= left.size() ) {
            for ( int i = iRight; i < right.size(); i++ ) {
                  list.set( iList, right.get( i ) );
                  iList++;
            }
      } else {
            for ( int i = iLeft; i < left.size(); i++ ) {
                  list.set( iList, left.get( i ) );
                  iList++;
            }
      }
}
```