CS 100 Project 2: Bridge Hands (SUPPLEMENTAL)
Summer 2001
Due in lecture, Thursday August 2

This handout serves as a supplement to the original Project 2 handout, and explains the Sorting.java and Analysis.java components in more detail.

1. Sorting

We are interested in looking at how different sorting algorithms perform. **Sorting.java** will eventually contain three sorting methods: insertionSort(), selectionSort(), and bubbleSort(). Insertion Sort has already been implemented for you and provides a good example of what we're looking for with the other two.

Each method takes in an array of Comparable objects and returns an integer indicating how many comparisons were done to sort the array. Comparable is a Java **interface**, and any class can be defined as "implements Comparable". This is different from "extends Otherclass", which indicates that the class inherits variables and methods from Otherclass. The "implements" keyword indicates that the class commits to providing certain method(s) that appear in the Comparable interface. (For more details, see Appendix 7 in Savitch.) In this case, it means that the class must provide a **compareTo()** method. You'll implement this method in BridgeHand.java, and it already exists for Card.java. The question, then, is what you should do with it when implementing the sorting methods.

First, let's take a look at insertionSort():

```
public static int insertionSort(Comparable[] a)
```

As you can see, it takes in an array of Comparable objects and returns an integer.

```
{
  int numComps = 0;
  // Make a copy of the array to insert into
  Comparable[] c = new Comparable[a.length];
```

This version of insertion sort makes a copy, **c**, of the input array **a** and writes the sorted elements into this array, then copies the sorted version back into array **a.** This is a somewhat clearer version of the algorithm, although it is possible to do the sorting in-place (all in the same array).

```
  // Now sort 'a' into 'c'
  // For each element in a, insert it in the right place in c
  for (int i=0; i<a.length; i++)
  {
      int j;
      for (j=0; j<i; j++)
      {
        // Increment comparisons
        numComps++;
        // if c[j] >= a[i], a[i] belongs here, so break
        if (c[j].compareTo(a[i]) > -1) break;
      }
```

The variable **j** is declared outside of the **for** loop so that we can access its value after the loop finishes. The purpose of this loop is to find the right place in **c** to insert **a[i]**. If compareTo() returns –1, then **c[j]** < **a[i]**, and we keep moving down the array. However, if compareTo() returns 0 or 1, then we've found the right place to insert **a[i]**, so we break out of the loop. Each comparison we do is accompanied by an increment of **numComps**.

```
      // Move everything from j on, down one
      for (int k=i; k>j; k--)
      {
        c[k] = c[k-1];
      }
```

Now that we've found the right place to put **a[i]**, we need to move everything from **j** to the end of the sorted part of **c** down one, to make room. This loop does that.

```
            // Now put the ith element in at j
            c[j] = a[i];
```

We can now insert **a[i]** in the proper place.

```
            // INVARIANT: array c is now sorted from 0 to i.
```

An **invariant** is a statement that is always true. At the end of each iteration of the outer loop, we know that array **c** must be sorted from index 0 to **i.** Thus, after the loop has completed, array **c** must be sorted from 0 to **a.length-1**, which is what we wanted.

```
        }
        // Now copy c back into a, since it's sorted
        for (int i=0; i<a.length; i++) a[i] = c[i];

        return numComps;
    }
}
```

Finally, we copy the sorted array **c** back into **a** and return the computed number of comparisons that the algorithm had to do.

You should implement both **selectionSort()** and **bubbleSort()** in similar ways. The algorithms are provided in comments in Sorting.java. If you do not understand the algorithms, please ask!

Java supplies a **sort()** method which is a fourth alternative that can sort any array of Comparable objects. We encourage you to use this method to check the results of your sorting algorithms against the "correct" answer.

2. Analysis (MATLAB)

Now that you have implemented the sorting routines, we'd like to compare the three algorithms in terms of how much work they do to sort an array.

Download **Analysis.java** from the website. In this file, you'll run a bunch of experiments sorting different numbers of BridgeHands with each of the three algorithms. We know that it will take more work to sort bigger arrays.. but how much more work? Here, you'll record statistics for how many comparisons it takes to sort 10 hands, 20 hands, 30 hands... all the way up to MAXHANDS hands. This information will be output to a file, and you'll use MATLAB to plot it.

1. **Read** through Analysis.java. Currently, the main() method runs one experiment: insertionSort() on an array of 10 BridgeHands, and writes out the result to insertion-comps.txt.

2. Now **modify the main() method** so that it loops from 10 to MAXHANDS, incrementing by 10 each time, and runs insertionSort() on an array of the proper size. You should store the number of comparisons each experiment takes – then **modify the try block** so that it writes out one line for each experiment that includes the **size of the array** and the **number of comparisons** that were required to sort it.

3. Now add code to do the same thing with the other two sorting routines (**selectionSort()** and **bubbleSort()**), writing out their results to selection-comps.txt and bubble-comps.txt respectively.

4. Finally, use MATLAB to **read in** the contents of insertion-comps.txt and **plot it** so that the x axis indicates the size of the array and the y axis indicates the number of comparisons. Add **an informative title (describing what you're plotting, plus your name and CUid)** and **x and y labels,** then **print out** your plot. Then do the same thing with selection-comps.txt and bubble-comps.txt. You should turn in three plots.
   NOTE: Arrays (matrices) in MATLAB number their entries from 1, not from 0.

5. Also print out a **listing of the MATLAB commands** you used to generate your plots.