Lab 6 Distributed: Wednesday, Oct 13th **Due:** Tuesday, Oct 19th @ 11pm.

Lab Objectives:

- Work with the Random class.
- Develop two sort classes using an interface to allow an estimated complexity to be determined.
- Develop two search classes using an interface to allow an estimated complexity to be determined.
- Create an exploration infrastructure to determine the overall performance of the different algorithms.
- Develop a report to display your findings and present an analysis of these findings.

Random

The Random class allows for the generation of pseudo-random values. For this class to correctly function a seed value needs to be passed when the object is created. Note, if this seed value is not different each time, then the same sequence will be generated. This is shown with the example code below.

```
for(int cnt1 = 0 ; cnt1 < 4; cnt1++) {
    rand = new Random(847638740);
    System.out.println("==============================");
    for(int cnt2 = 0; cnt2 < 5; cnt2++) {
        System.out.println(rand.nextDouble());
    }
}
```

The code will create a new object of type Random and then generate five random values. The code will do this four times. The output below shows the four runs side-by-side, which are four duplicate sets of values. So, each time the Random object is created it will generate the same sequence.

```
0.8481209093836806    0.8481209093836806    0.8481209093836806    0.8481209093836806
0.10770274476341235   0.10770274476341235   0.10770274476341235   0.10770274476341235
0.6978139119794761    0.6978139119794761    0.6978139119794761    0.6978139119794761
0.544835665068649     0.544835665068649     0.544835665068649     0.544835665068649
0.3126296907550802    0.3126296907550802    0.3126296907550802    0.3126296907550802
```

If a different random seed is used, even if there is a difference of 1, then a different sequence will be generated. The following code is the same as the code above, but the static seed `847638740` is incremented by the variable `cnt1`, creating the seeds: `847638740`, `847638741`, `847638742`, `847638743`.

```
for(int cnt1 = 0 ; cnt1 < 4; cnt1++) {
    rand = new Random(847638740 + cnt1);
    System.out.println("++++++++++++++++++++++++++++++");
    for(int cnt2 = 0; cnt2 < 5; cnt2++) {
        System.out.println(rand.nextDouble());
    }
}
```

The numbers from the above code will result in the four sequences of five. As can be seen, the first numbers are very similar, but the second numbers are quite different, as is the next after that.

```
0.8481209093836806    0.8480313276114756    0.8483000729280905    0.8482104911558855
0.10770274476341235   0.27724714058412814   0.7686139531219809    0.9381583489426967
0.6978139119794761    0.2681113279390849    0.5572190651590974    0.12751648111870606
0.544835665068649     0.32711570100936505   0.9802755931872167    0.7625556142267716
0.3126296907550802    0.6828403074808379    0.5722084424024035    0.9424190740293225
```

The Random class has a variety of “next” methods, such as: nextDouble, nextBoolean, and nextInt. The method “nextInt(int bound)” is very useful for drawing from a defined range of integers, see the Java API to learn more.

For this course, you will need to use different seed values when working with random, however during debugging, it is good to use a few common seed values, so the behavior remains consistent.

Bubble-Sort and Fast-Bubble-Sort

The bubble sort is well known as the worst sorting algorithm and runs at an unhealthy $O(N^2)$, where the “fast” bubble sort does not do much better. You will be using these two sorts for this lab. The basic code for bubble sort is given here.

```
for (int i = 0; i < data.length - 1; i++) {
    for (int j = 0; j < data.length - 1; j++) {

        if (data[j] > data[j+1]) {
            temp        = data[j];
            data[j]     = data[j + 1];
            data[j + 1] = temp;
        }
    }
}
```

The fast bubble sort is given here.

```
        for (int i = 0; i < data.length - 1; i++) {
            for (int j = 0; j < data.length - i - 1; j++) {

                if (data[j] > data[j+1]) {
                    temp        = data[j];
                    data[j]     = data[j + 1];
                    data[j + 1] = temp;
                }

            }
        }
```

The difference between the two different approaches is that the inner loop ignores the tail of the array where the elements are already sorted.

Linear Search and Binary Search.

Introduced in lecture were the linear search and binary search. The linear search looks through the array of elements from the front to the end. The elements should be sorted for either search to function in a reasonable way. The example static code for each type of search is given here.

```
    public static int linear_search(int[] data, int key) {

        for(int i = 0; i < data.length; i++) {

            if(key == data[i]) {
                return data[i];
            }
        }

        return -1;
    }
```

The binary search is more sophisticate than the linear search. In the following code the array is repeatedly sliced in half, allowing for a faster search time. But again, the array of values must be sorted in advance to the search being performed or else this algorithm will not function.

```
    public static int binary_search(int[] d, int k) {
        int mid  = 0;
        int low  = 0;
        int high = d.length - 1;

        while(high >= low) {
```

```
            mid = (high + low)/2;

            if (d[mid] < k) {       // search to the right
                low = mid + 1;
            } else if (k < d[mid]) {// search to the left
                high = mid - 1;
            } else {
                return d[mid];      // item found
            }
        }

        return -1;                  // item not found
    }
```

Complexity Exploration by Counting.

In lecture we are starting to talk about algorithmic complexity, which can be specified as an equation that specifies how the run time of an algorithm grows with the increasing size of its input data. Another approach is to determine how long the algorithm runs in real time, but with the speed of modern computers, this approach is very limited. A third approach is to count the number of times a specific line in an algorithm is executed. Consider the following annotated linear search algorithm converted into a while-loop.

```
        int i = 0;                  // the assignment will execute once

        while (i < data.length) {   // If the key does not exist or is the last
                                    // item, this condition will be checked N + 1
                                    // times; N true conditions and one false.

          if(key == data[i]) {      // This condition will be checked up till
                                    // the key is found.

             return data[i];        // If the key is found, this will execute once
          }

          i++;                      // This will iterate up till
                                    // the key is found.
        }

        return -1;                  // If the key is not found, this return will
                                    // execute once
```

From this annotated code, placing the count in the loop, but not the if-condition, will be the most effective way of determining how the algorithm is running. As it will capture the number of iterations that occur before the key is found. Then by obtaining these counts for different values in

an array, we can get an estimate of how the algorithm behaves for different situations. To do this a consistent set of methods are needed that will initialize and manipulate a counter that can be checked at the end of an algorithm run. For this lab the following interface will be used by different classes that will be developed.

```
/**
 * The ComplexityCount interface defines several methods that are used
 * to manipulate a counter to determine the number of iterations
 * an algorithm performs for estimating the algorithmic complexity.
 *
 * The class will need to specify the counter that is to be used.
 */
public interface ComplexityCount {

    /**
     * This method will be used to initialize the counter to zero.
     */
    public void initialize();

    /**
     * This method increments the counter by one.
     */
    public void incr();

    /**
     * This will return the counter value.
     *
     * @return The returned counter.
     */
    public int  result();

    /**
     * This will run the algorithm implement within the class.
     */
    public void run_algorithm();
}
```

Once the classes are created that all implement this interface, these classes can be run together in the following way.

```
ComplexityCount tc[] = new ComplexityCount[2];
int[] data           = new int[200000];

// load random numbers into data and sort it

tc[0] = new Fast_Search(data);
tc[1] = new Slow_Search(data);

for(int cnt = 0; cnt < tc.length; cnt++) {
  tc[cnt].initialize();
  tc[cnt].run_algorithm();
  process_result(tc[cnt].run_algorithm());
}
```

This will allow you to run the same set of algorithms for different cases.

The Lab Assignment.

Part 1 – Create the following four-classes. Each will implement the **ComplexityCount** interface and should be capable of manipulating **int** arrays.

- BubbleSort -- this class will take an array of **int**'s and sort them using the provided bubble sort algorithm. The number of iterations to perform the sort should be tracked to have an estimated algorithmic complexity.
- FastBubbleSort -- this class will take an array of **int**'s and sort them using the provided fast bubble sort algorithm. The number of iterations to perform the sort should be tracked to have an estimated algorithmic complexity.
- LinearSearch -- this class will take an array of **int**'s and search for a key using the provided linear search algorithm. The number of iterations to perform the search should be tracked to have an estimated algorithmic complexity.
- BinarySearch -- this class will take an array of **int**'s and search for a key using the provided binary search algorithm. The number of iterations to perform the search should be tracked to have an estimated algorithmic complexity.

Part 2 – Build a class that contains a main method to implement the following steps.

1. Create an array of random integers of N size. This size should be a variable that will be manipulated in later parts of the assignment.
    a. The sequence will be developed using the **Random** class described above.
2. Sort the array of numbers using both the fast and slow bubble sort to determine the estimated complexity for each data set. Be careful that both algorithms start with the same random sequence.
3. Search for N random values using both the linear and binary search to determine the estimated complexity using the sorted data set from the previous step. The same sequence of N values should be used for each search.
4. Compute and display the following information.
    a. The minimum, maximum, and average estimated complexity for each sort and search algorithm. The average is determined by summing the individual estimated values and then dividing by the number of values.
    b. For the search algorithms determine the number of random key values that were not found.
    c. Hint, writing this data to a CSV file will make it easier to get it into spreadsheet software.

Part 3 – Extend part 2 to run for the following N values: 10, 100, 1000, 10000, 100000, 1000000. Then run the program 20 different times with a different random seed each time. Finally, organize the data into a table, then describe how the programs performed across the different values of N and for each random seed. Include a graph where the x-axis represents the values N and y-axis is the estimated complexity, the results displayed in the graph should also be included in the description. **Your report must be submitted in PDF format. (All common modern word processors allow for this.)**

Submitting the lab and grade breakdown.

At the end of each lab, let the lab instructor or student proctor know how much progress you have made. This allows the instructor to help you by understanding where you are and give feedback on your status.

To submit the lab: add your report PDF to the directory that contains your BlueJ project, zip that directory, and upload it to the submission item on Moodle for this Lab.

Grade breakdown:

Program complies.	required for a grade
Poorly organized code.	up to -2 points
Poor comments.	up to -2 points
Implementation of the Bubble and Fast Bubble sorts.	2 pts
Implementation of the Linear and Binary searches.	2 pts
Implementation of the main method.	2 pts
Report organization.	2 pts
Report accuracy.	2 pts