COMP 2210 Empirical Analysis Assignment – Part B

Marco A. Zuniga

September 26, 2016

Abstract

This lab report is intended to show the different lab experiments performed in order to identify all five of the sorting methods that were given. Below one can find exactly how each method was determined with data, charts, and graphs. To go along with the code used to determine each sorting type, another class was provided in order to determine which methods were stable in order to determine each unknown sorting method.

1. Problem Overview

The research being studied in this assignment is to perform multiple time cases in order to identify all of the sorting methods being used. Whether it be selection sort, insertion sort, merge sort, randomized quicksort or non-randomized quicksort. It was the student's job to perform any experiment necessary to correctly identify each sorting method. Every student was required to run each sorting method based off of their Banner ID. No matter what Banner ID is used, the experiment performed is guaranteed to follow the property of polynomial time complexity function found below. The value of k can be determined by taking the log of each ratio.

$$T(N) \propto \mathrm{N}^{\mathrm{k}} ==> \frac{T(2N)}{T(N)} \propto \frac{(2N)^k}{N^k} = \frac{2^k N^k}{N^k} = 2^k$$

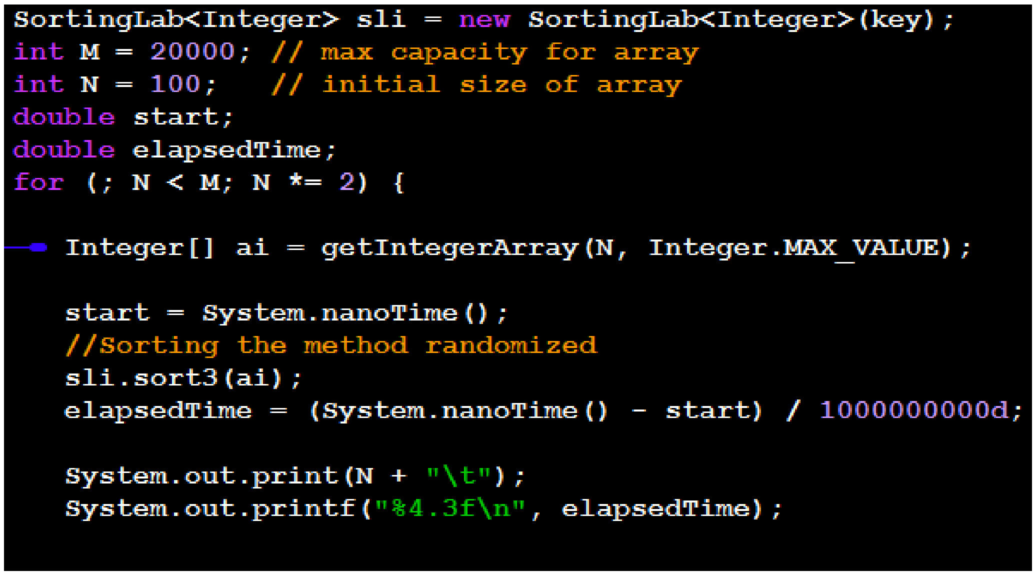
$$k = \log_2(R)$$

2. Experimental Procedure

In order to get enough data, a for-loop was used to iterate until the value of N (10,000) reached the value of M (2,000,000). For some of the sorting methods, the size of N was reduced in order the speed up the process of recording data. One can find the environment used to complete this experiment in Table 1 and also an image of the for-loop used for this experiment in Figure 1 below.

Table 1: Environment

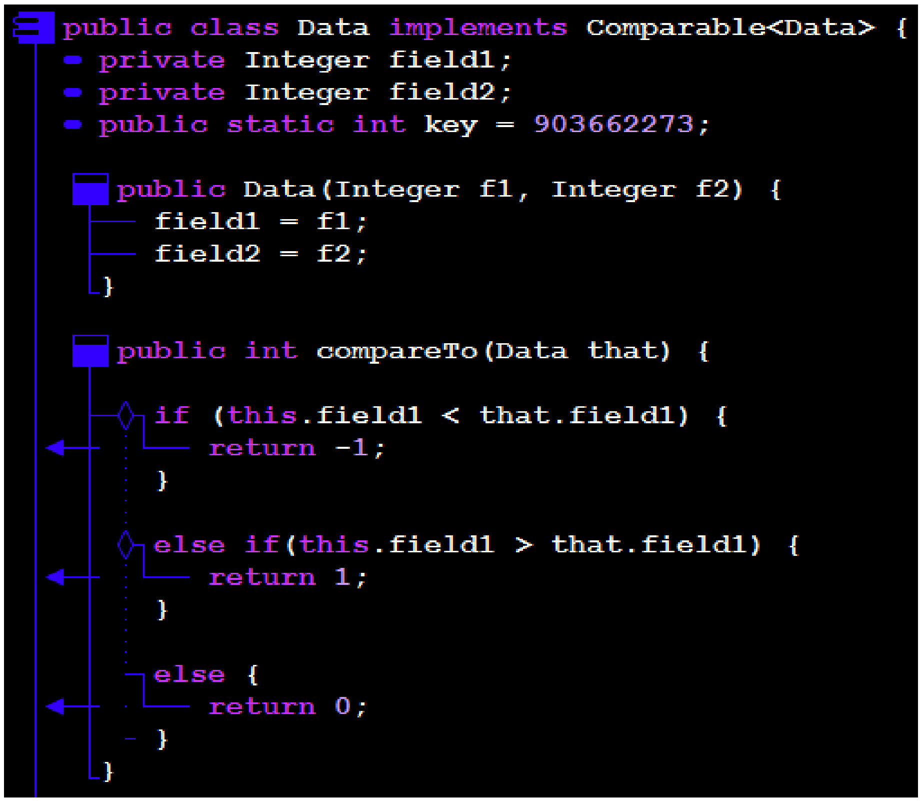
Laptop Used	*Lenovo IdeaPad P400*
Operating System	*Windows 10*
Hard Drive Capacity	*1 TB*
RAM	*8 GB*
Processor	*Intel Core i7 – 3632QM*
GHZ	*2.20 GHZ*
Java Version	*8 Update 65*
Programming Software	*JGrasp*



```java
SortingLab<Integer> sli = new SortingLab<Integer>(key);
int M = 20000; // max capacity for array
int N = 100;   // initial size of array
double start;
double elapsedTime;
for (; N < M; N *= 2) {

    Integer[] ai = getIntegerArray(N, Integer.MAX_VALUE);

    start = System.nanoTime();
    //Sorting the method randomized
    sli.sort3(ai);
    elapsedTime = (System.nanoTime() - start) / 1000000000d;

    System.out.print(N + "\t");
    System.out.printf("%4.3f\n", elapsedTime);

```

Figure 1: For-loop used to in order to calculate times.

The code found below was used in order to test which methods were stable. By creating an object that holds 2 integers, the compareTo method was used in order to sort all objects based off of the 1st integer. One can then check the second integer in order to determine which method is stable or not. The stability tests were run through all methods below in the data collections section of the lab report.



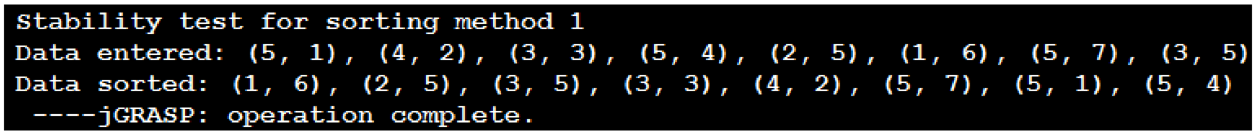
```java
public class Data implements Comparable<Data> {
    private Integer field1;
    private Integer field2;
    public static int key = 903662273;

    public Data(Integer f1, Integer f2) {
        field1 = f1;
        field2 = f2;
    }

    public int compareTo(Data that) {

        if (this.field1 < that.field1) {
            return -1;
        }

        else if(this.field1 > that.field1) {
            return 1;
        }

        else {
            return 0;
        }
    }
```

Figure 2: Class used to determine stability

3. Data Collection and Analysis

Sorting Method 1:

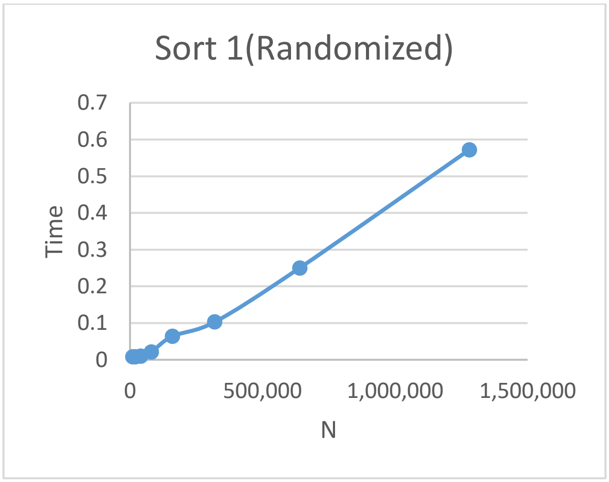
Below one can find all of the times recorded in order to calculate a worst time case analysis. Being able to determine whether or not the sorting method being used was stable or not was a great help as it narrowed down the possible sorting methods. One can see below in Figure 3 that the method is not stable. Narrowing it down to selection, and quicksort.



```
Stability test for sorting method 1
Data entered: (5, 1), (4, 2), (3, 3), (5, 4), (2, 5), (1, 6), (5, 7), (3, 5)
Data sorted: (1, 6), (2, 5), (3, 5), (3, 3), (4, 2), (5, 7), (5, 1), (5, 4)
 ----jGRASP: operation complete.
```

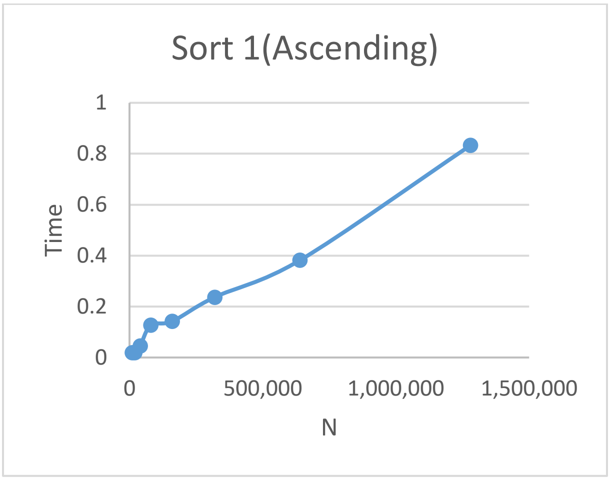
Figure 3: sort1 stability test

The ratio of the randomized array of integers seemed to be approaching a value of 2.3, the ascending array of integers seemed to be approaching a value of 2.2, and the descending array of integers seemed to be approaching a value of 5. All ratios can be found by dividing the time in question with the previous time. The real giveaway was running the ascending array through the method and seeing that the results came out to Nlog$_2$(N). Since quicksort picks the very first index as the pivot point, that means the array must have been randomized to get the big-Oh time complexity it got, therefore, this method can be identified as the *randomized quicksort* method.



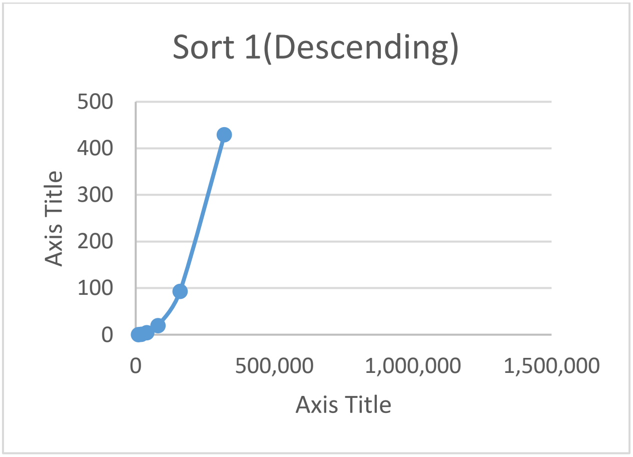
N	Time	Ratio
10,000	0.008	
20,000	0.008	1
40,000	0.01	1.25
80,000	0.021	2.1
160,000	0.064	3.05
320,000	0.103	1.61
640,000	0.25	2.45
1,280,000	0.572	2.29

Figure 4: sort1 with randomized integers.



N	Time	Ratio
10,000	0.019	0
20,000	0.019	1
40,000	0.045	2.368
80,000	0.127	2.8
160,000	0.142	1.12
320,000	0.237	1.67
640,000	0.382	1.6
1,280,000	0.833	2.18

Figure 5: sort2 with ascending integers.

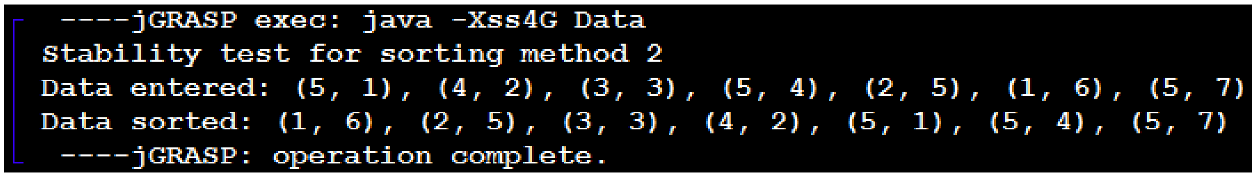


N	Time	Ratio
10,000	0.331	0
20,000	1.284	3.88
40,000	4.86	3.79
80,000	20.055	4.13
160,000	93.558	4.67
320,000	429.269	4.59
640,000		
1,280,000		

Figure 6: sort1 with descending integers.

Sorting Method 2

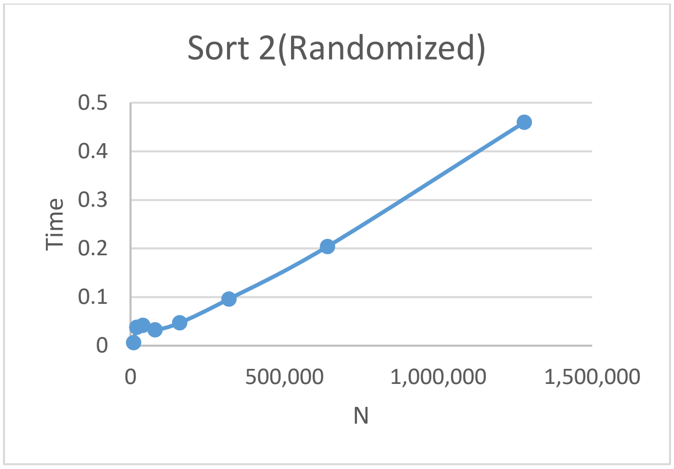
Looking at the stability test below in Figure 7, and identifying that this method is stable, one can narrow down the choices for sort2 to the possibilities of insertion sort and merge sort.



```
  ----jGRASP exec: java -Xss4G Data
 Stability test for sorting method 2
 Data entered: (5, 1), (4, 2), (3, 3), (5, 4), (2, 5), (1, 6), (5, 7)
 Data sorted: (1, 6), (2, 5), (3, 3), (4, 2), (5, 1), (5, 4), (5, 7)
  ----jGRASP: operation complete.
```

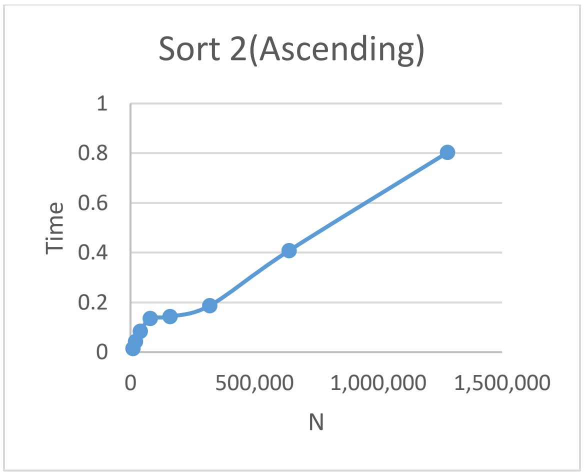
Figure 7: sort2 stability test

Both the randomized and ascending seemed to approach the value of 2, which would mean that the big-Oh time complexity would have to be $N\log_2(N)$. Considering the fact that this method is stable, this method must be the *merge sorting* method.



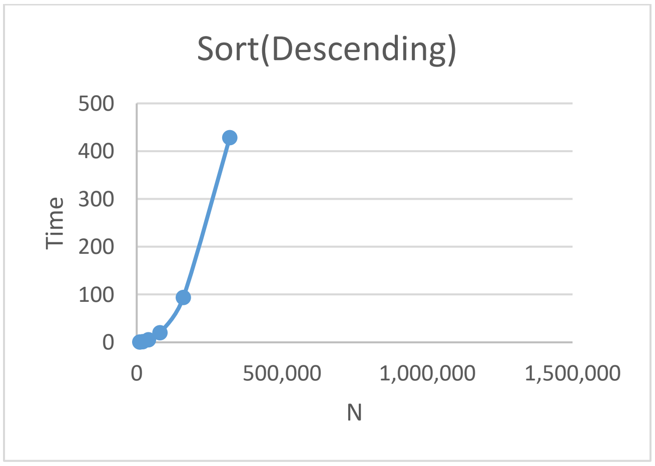
N	Time	Ratio
10,000	0.006	0
20,000	0.038	6.33
40,000	0.042	1.12
80,000	0.033	0.79
160,000	0.047	1.4
320,000	0.096	2.04
640,000	0.204	2.13
1,280,000	0.46	2.25

Figure 8: sort2 with randomized integers



N	Time	Ratio
10,000	0.014	0
20,000	0.043	3.07
40,000	0.084	1.95
80,000	0.135	1.61
160,000	0.143	1.06
320,000	0.187	1.31
640,000	0.408	2.18
1,280,000	0.803	1.97

Figure 9: sort2 with ascending integers

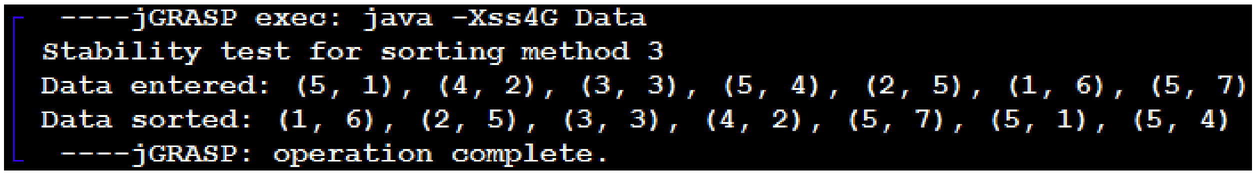


N	Time	Ratio
10,000	0.318	0
20,000	1.328	4.18
40,000	4.861	3.66
80,000	20.088	4.13
160,000	93.359	4.65
320,000	427.975	4.58
640,000		
1,280,000		

Figure 10: sort1 with descending integers

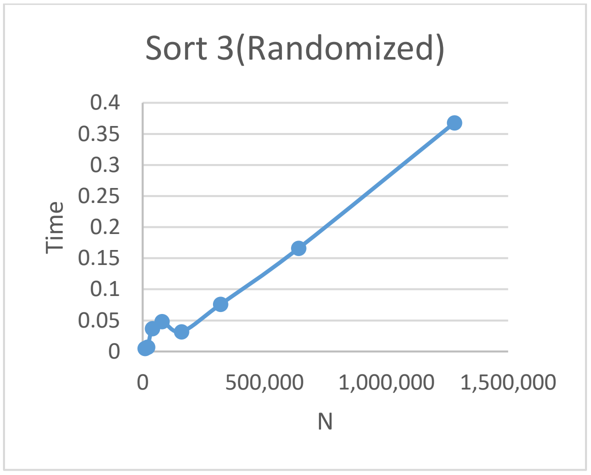
Sorting Method 3:

The third method is not stable based off of the test results in Figure 11. Please note that for this method and the remainder of the experiments, reducing the value of N to 100 for the ascending and descending arrays was necessary in order to receive times quicker. Since the ascending array seems to be growing quadratically, that lets us know this method is the *non-randomized quicksort* method.



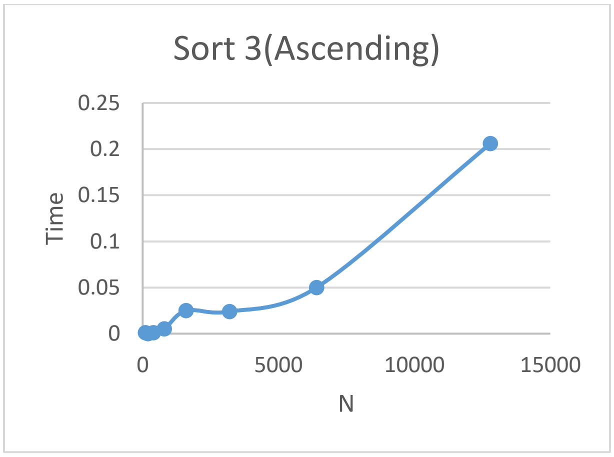
```
 ----jGRASP exec: java -Xss4G Data
Stability test for sorting method 3
Data entered: (5, 1), (4, 2), (3, 3), (5, 4), (2, 5), (1, 6), (5, 7)
Data sorted: (1, 6), (2, 5), (3, 3), (4, 2), (5, 7), (5, 1), (5, 4)
 ----jGRASP: operation complete.
```

Figure 11: sort3 stability test



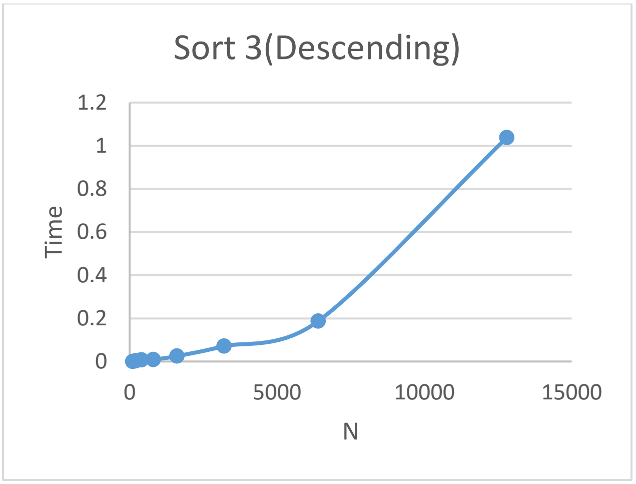
N	Time	Ratio
10,000	0.005	0
20,000	0.007	1.4
40,000	0.037	5.29
80,000	0.048	1.3
160,000	0.032	0.66
320,000	0.076	2.375
640,000	0.166	2.18
1,280,000	0.368	2.22

Figure 12: sort3 with randomized integers



N	Time	Ratio
100	0.001	0
200	0	0
400	0.001	0
800	0.005	5
1600	0.025	5
3200	0.024	0.96
6400	0.05	2.08
12800	0.206	4.12

Figure 13: sort3 with ascending integers

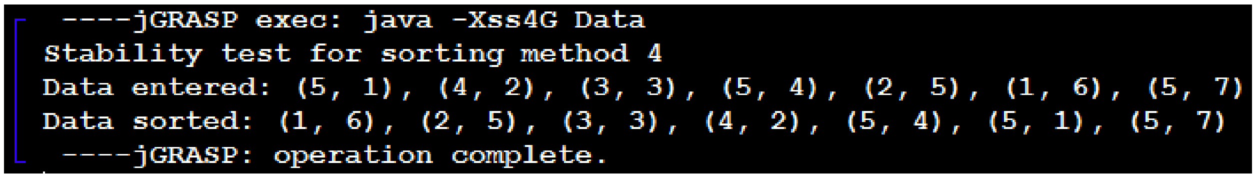


N	Time	Ratio
100	0.001	0
200	0.003	3
400	0.008	2.66
800	0.009	1.13
1600	0.025	2.77
3200	0.071	2.84
6400	0.187	2.63
12800	1.038	5.55

Figure 14: sort1 with descending integers

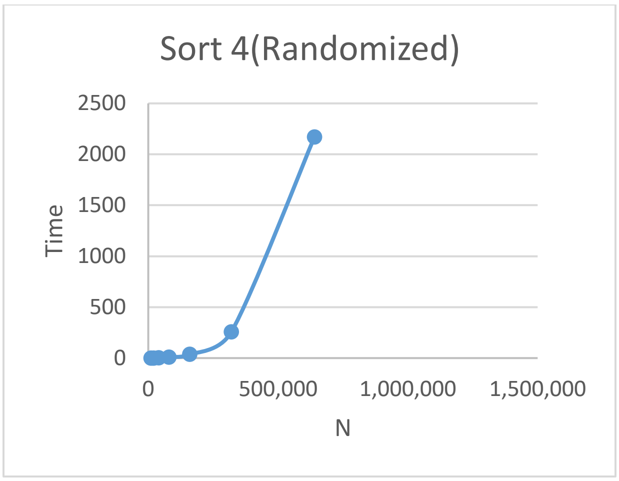
Sorting Method 4:

The fourth sorting method can be identified as not stable. All three tests were found to be approaching a value bigger than four which can then be concluded that this method is the *selection sort* method.



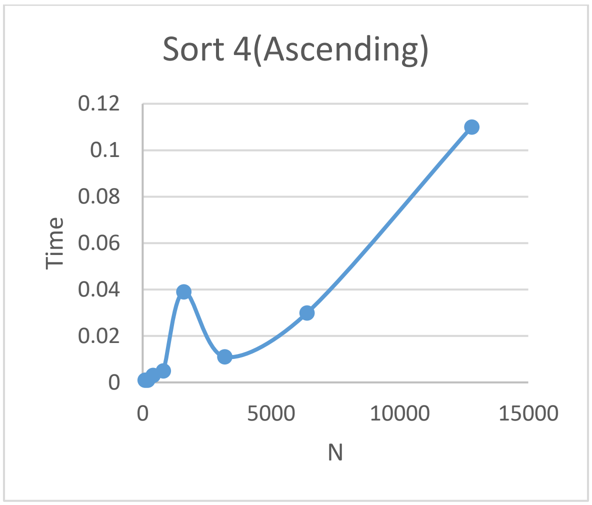
```
  ----jGRASP exec: java -Xss4G Data
 Stability test for sorting method 4
 Data entered: (5, 1), (4, 2), (3, 3), (5, 4), (2, 5), (1, 6), (5, 7)
 Data sorted: (1, 6), (2, 5), (3, 3), (4, 2), (5, 4), (5, 1), (5, 7)
  ----jGRASP: operation complete.
```

Figure 15: sort4 stability test



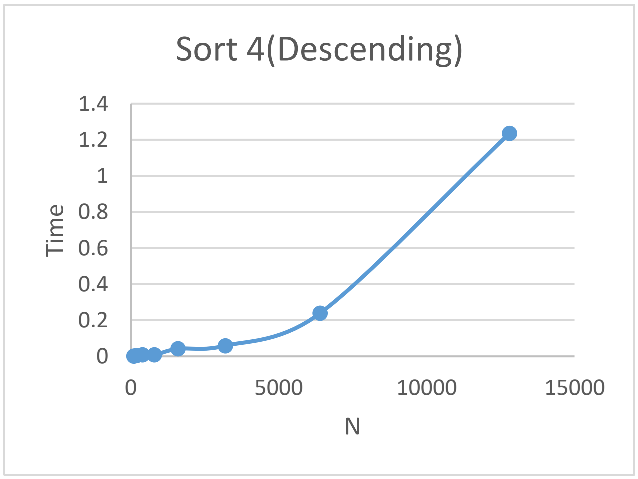
N	Time	Ratio
10,000	0.182	0
20,000	0.687	3.77
40,000	2.194	3.19
80,000	10.31	4.7
160,000	37.101	3.599
320,000	256.37	6.9
640,000	2169.738	8.46
1,280,000		

Figure 16: sort4 with randomized integers



N	Time	Ratio
100	0.001	
200	0.001	
400	0.003	
800	0.005	1.6
1600	0.039	7.
3200	0.011	0.28
6400	0.03	2.7
12800	0.11	3.6

Figure 17: sort4 with ascending integers

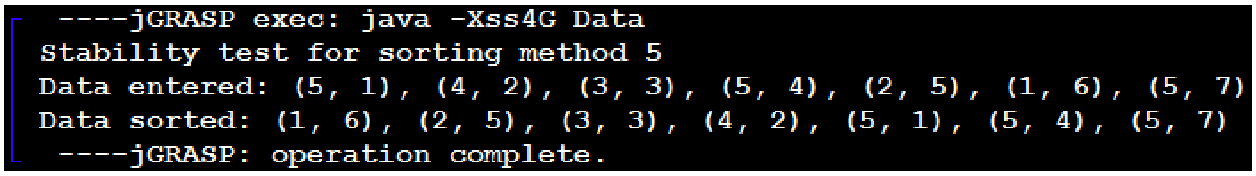


N	Time	Ratio
100	0.002	
200	0.005	2.
400	0.009	1.
800	0.008	0.88
1600	0.042	5.2
3200	0.059	1.
6400	0.239	4.
12800	1.235	5.1

Figure 18: sort1 with descending integers

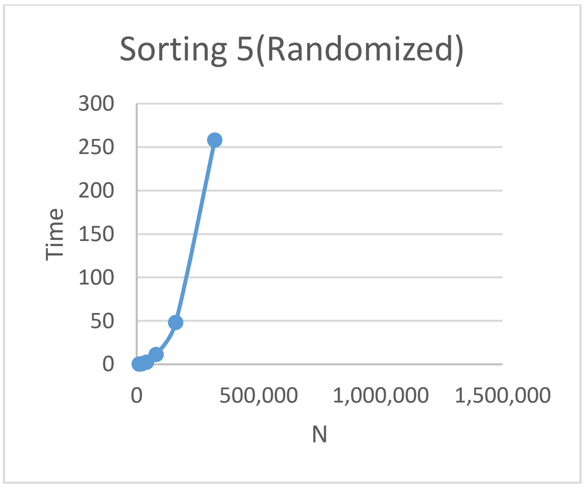
Sorting Method 5:

The last sorting method can be identified to be stable. Since the time seems to be growing quadtratically when the randomized and descending arrays are passed through the method, and growing linearly when almost sorted, this method can be identified as the *insertion sorting* method.



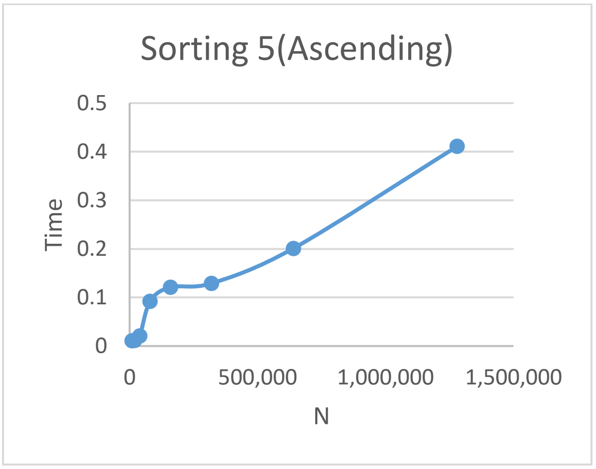
```
 ----jGRASP exec: java -Xss4G Data
Stability test for sorting method 5
Data entered: (5, 1), (4, 2), (3, 3), (5, 4), (2, 5), (1, 6), (5, 7)
Data sorted: (1, 6), (2, 5), (3, 3), (4, 2), (5, 1), (5, 4), (5, 7)
 ----jGRASP: operation complete.
```

Figure 19: sort5 stability test



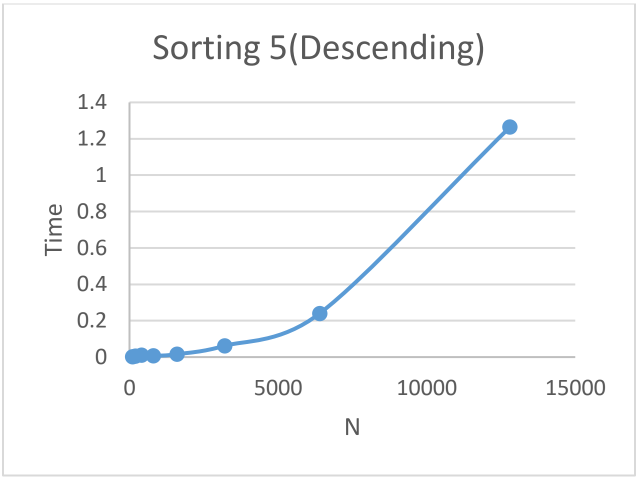
N	Time	Ratio
10,000	0.179	0
20,000	0.649	3.626
40,000	2.637	4.063
80,000	11.543	4.38
160,000	48.198	4.18
320,000	258.004	5.35
640,000		
1,280,000		

Figure 20: sort5 with randomized integers



N	Time	Ratio
10,000	0.011	0
20,000	0.012	1.09
40,000	0.021	1.75
80,000	0.092	4.38
160,000	0.121	1.32
320,000	0.129	1.1
640,000	0.201	1.56
1,280,000	0.411	2.045

Figure 21: sort5 with ascending integers



N	Time	Ratio
100	0.002	0
200	0.005	2.5
400	0.011	2.2
800	0.007	0.636
1600	0.016	2.29
3200	0.062	3.875
6400	0.24	3.87
12800	1.264	5.266

Figure 22: sort1 with descending integer

4. Interpretation

Through this data, one can find that collecting enough data can allow someone to determine hidden codes. This lab allowed the student to gain a better insight on how to find the time complexity of whatever method by simply sunning multiple tests as found in the Figures above. By simply running each method through a for-loop and recording the times of each iteration, the student was able to figure out which sorting method was which. Also figuring out the stability of each method beforehand allowed the student to narrow down the options available for each method.