**COMP 2210 Empirical Analysis Assignment – Part B**

Austin Keelin

September 25, 2016

**Abstract**

This experiment was done to determine if it is possible to experimentally verify which sorting method is being used, when only the results are observable, not the method. This is done by recording the time for the method to run, therefore obtaining the big-Oh running time; and by observing, in this case, the array as it is being modified by the sort method. For this experiment, there were five potential sort methods which were: merge sort, insertion, randomized quicksort, selection, and non-randomized quicksort, which is the experimentally determined order the methods were implemented.

1. **Problem Overview**

In general, the problem was to verify the identity of a sort method using only timing data, and also by observing the array as it is sorted. Similar to part A, each student uses their banner ID as a key required for the SortingLab constructor, which creates a SortingLab object whose sort methods are randomized, meaning each student will have their methods in a different, random order. The time complexity for each student was guaranteed to be proportional to Nk where k is a positive integer. The work in this experiment rests on the property of polynomial time complexity function as shown and explained in figure 1.

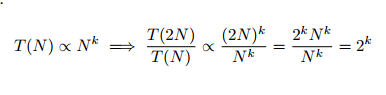


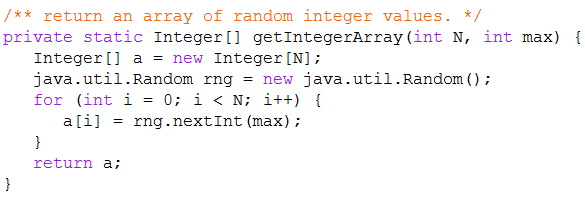
Figure 1. This property shows that as N is doubled, the ratio of the run time of

the method on the current value of N to the method’s run time on the previous

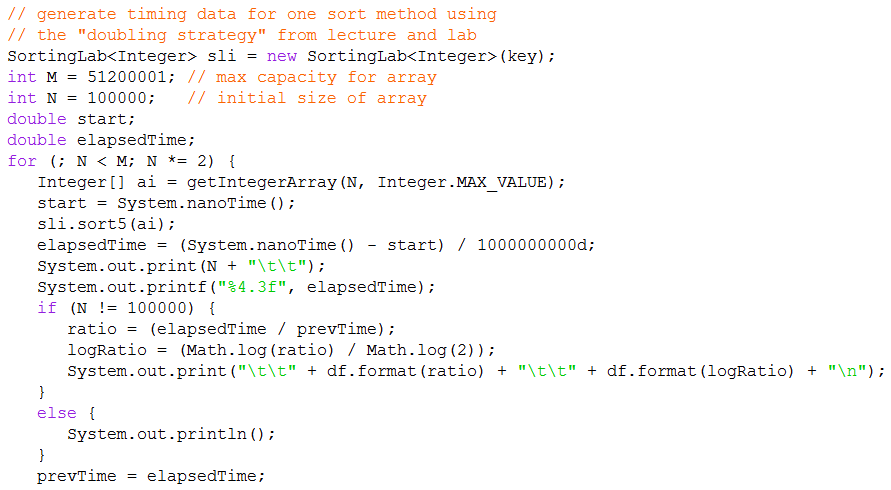
value of N converges to a constant R, which is equal to 2k, and thus k = log2R.

1. **Experimental Procedure**

I decided editing the provided SortingLabClient would be easier then creating my own class. The class before editing included the getIntegerArray method which creates an integer array with size of N, and fills with random numbers. This array would be useful for the main method of SortingLabClient which, under a for loop, calls getIntegerArray starting with N, and each run determining the time it takes for the sort method to sort the array up until M, the max size of the array entered by the user.



Listing 1: Code which shows the getIntegerArray method.



Listing 2: Code showing the main method used in determining the

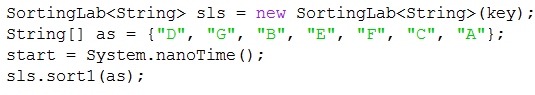
time complexity for each method. Sort methods 1, 3, and 5 used the

same N, and M values, while methods 2, and 4 had separate N and M values.

This particular code was used for determining the complexity for method 5.

The main method for SortingLabClient would print the value of N and the time elapsed, but did not print the ratio or the log ratio. Therefore, code was added to find the ratio (elapsedTime / prevTime). Log ratio is used for finding k, where k = log2R. Since there is no method in the Math class to find log base 2 of R, instead k = logR / log2 was used, which will give you the same result. Finding the ratio and log ratio for each iteration was set under an if statement to avoid finding these values for the first iteration, as they couldn’t exist. This is shown in Listing 2.

Since the sort methods can have the same time complexity, one would also have to observe the changes made to the array in order to accurately determine the order of the sort methods. This was done by removing iteration: the sort method was observed on one pre-determined array which was used for each method. Using canvas in jGRASP, it is possible to observe each step of the sort method, which shows how the array is changed. Listing 3 shows what this method looked like (for this I ran the method under the for loop, but it could have been run anywhere in the main method as iteration was not necessary) and Figures 2 and 3 shows the canvas viewer and how change in the array is observed.



Listing 3: Code which shows how the main method is changed to

allow a String array to be sorted.

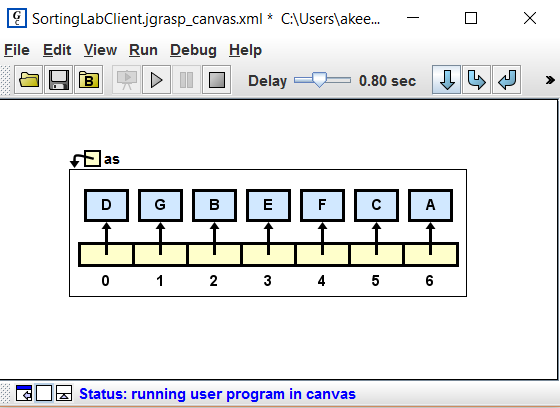


Figure 2: Shows the canvas viewer and how the array

is displayed before the array has been changed.

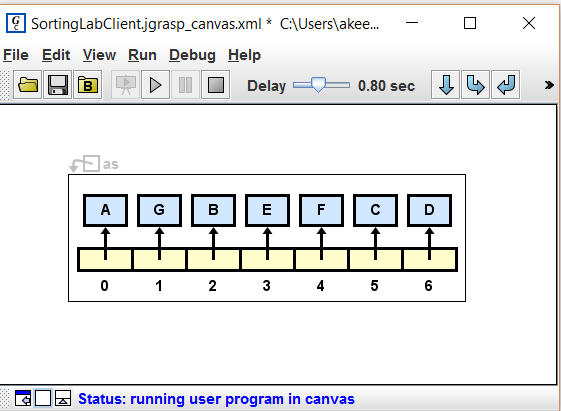


Figure 3: Shows the canvas viewer after one change has been made

in sorting the array. Value “A” has switched with value “D”.

Now that data can be collected, the next steps became, in the simplest sense, comparing it to known values. The time complexities can be compared to the best case, worst case, average case table (as shown in figure 4) to help determine the method; and the observed changes can be compared to the array changes present in the Sotring.pdf PowerPoint viewable on the Canvas for the course (it would be far too much to post the example array changes for each sort method).

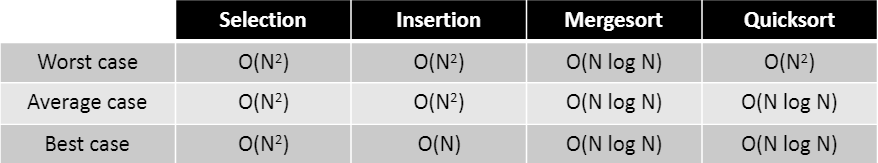


Figure 4: Shows the table which lists the time complexity for each sort method.

Lastly, it is important to note the environment in which this experiment was performed. All of the data was collected through use of jGRASP 2.0.2\_01 on an ASUS laptop with Windows 10 and a 64-bit operating system.

1. **Data Collection and Analysis**

The first run began with using the integer array to determine the time complexity for sort 1. Testing began with a value of 10,000 for N and M of 200,000. The run finished rather quickly and I realized I would need to increase both N and M to receive meaningful data. After much trial and error in choosing N and M, I decided on N being 100,000 and M being 51,200,000. This surprisingly still took under a minute to reach the max and did apply meaningful data. Note: for each table, N will tell the problem set value for each iteration, T will tell the elapsed time in seconds, Ratio will tell the ratio (elapsed time / previous time) for each iteration, and finally Log Ratio will tell the value of k, where k = log2R.

Table 1: Running time data for sort 1.

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **T (sec)** | **Ratio** | **Log Ratio** |
| 100000 | 0.189 |  |  |
| 200000 | 0.067 | 0.351 | -1.509 |
| 400000 | 0.155 | 2.333 | 1.222 |
| 800000 | 0.317 | 2.04 | 1.029 |
| 1600000 | 0.863 | 2.727 | 1.447 |
| 3200000 | 2.055 | 2.381 | 1.252 |
| 6400000 | 4.874 | 2.371 | 1.246 |
| 12800000 | 11.342 | 2.327 | 1.219 |
| 25600000 | 29.07 | 2.563 | 1.358 |
| 51200000 | 54.164 | 1.863 | 0.898 |

The log ratio values for this were surprisingly consistent and gave sort method 1 a time complexity of O (N log N), which means it could be merge, or either of the quicksort methods. So the next step was to compare the changes of the array to how the methods are known to alter the arrays as observed in lecture.

Table 2: Changes in the array for sort 1. The value which changes is highlighted to show the sort pattern.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| D | G | B | E | F | C | A |
| B | G | B | E | F | C | A |
| B | D | B | E | F | C | A |
| B | D | E | E | F | C | A |
| B | D | E | G | F | C | A |
| B | D | E | G | C | C | A |
| B | D | E | G | C | F | A |
| B | D | E | G | A | F | A |
| B | D | E | G | A | C | A |
| B | D | E | G | A | C | F |
| A | D | E | G | A | C | F |
| A | B | E | G | A | C | F |
| A | B | C | G | A | C | F |
| A | B | C | D | A | C | F |
| A | B | C | D | E | C | F |
| A | B | C | D | E | F | F |
| A | B | C | D | E | F | G |

Sort 1 correctly orders the left side of the array (including G in the left) and then orders the right side of the array. After sorting both sides separately, the two halves are then combined in the correct order. Based on how the array is sorted, and the complexity being O (N log N), sort 1 must be merge sort.

While only knowing that sort 2 isn’t merge, I left the values of N and M the same as in sort 1, and began testing. After 30 seconds with nothing printed yet, I realized N and M would need to be significantly lowered. After more trial and error, I decided on N being 250, and M being 256,000. Again, testing for time complexity went very well, resulting in consistent log ratio values.

Table 3: Running time data for sort 2.

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **T (sec)** | **Ratio** | **Log Ratio** |
| 250 | 0.003 |  |  |
| 500 | 0.005 | 1.854 | 0.891 |
| 1000 | 0.006 | 1.164 | 0.219 |
| 2000 | 0.02 | 1.923 | 0.943 |
| 4000 | 0.019 | 1.582 | 0.661 |
| 8000 | 0.095 | 4.915 | 2.297 |
| 16000 | 0.444 | 4.687 | 2.229 |
| 32000 | 1.784 | 4.02 | 2.007 |
| 64000 | 10.204 | 5.718 | 2.51 |
| 128000 | 41.256 | 4.043 | 2.015 |
| 256000 | 206.205 | 4.998 | 2.321 |

Starting at problem size of 8,000 the method began to give a consistent log ratio around 2. This would give a time complexity of O (N2), which means it could be either insertion, selection, or possibly worst case non-randomized quicksort (although worst case for quicksort would need an “edge” value to be first, I still analyzed the data as though it could happen). Again, I would have to observe how the array is sorted to determine the sort method.

Table 4: Changes in the array for sort 2.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| D | G | B | E | F | C | A |
| B | D | G | E | F | C | A |
| B | D | E | G | F | C | A |
| B | D | E | F | G | C | A |
| B | D | E | F | C | G | A |
| B | D | E | C | F | G | A |
| B | D | C | E | F | G | A |
| B | C | D | E | F | G | A |
| B | C | D | E | F | A | G |
| B | C | D | E | A | F | G |
| B | C | D | A | E | F | G |
| B | C | A | D | E | F | G |
| B | A | C | D | E | F | G |
| A | B | C | D | E | F | G |

Sort 2 goes through the array in order, placing each value in order as it goes. When it gets to a value, it places the value in order, with the values that have already been scanned; it does not search the array for the correct value, it simply sorts as it goes through the array. Based on this pattern, and the time complexity being O (N2), sort 2 must be insertion sort.

Similar to what was done with sort 2, I left the values of N and M for the first test of sort 3 to what was used for sort 2. After the method returned the time for each value almost immediately, I decided to use the values from sort 1, which would work perfectly (these values were N of 100,000 and M of 51,200,000).

Table 5: Running time data for sort 3.

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **T (sec)** | **Ratio** | **Log Ratio** |
| 100000 | 0.1 |  |  |
| 200000 | 0.071 | 0.708 | -0.498 |
| 400000 | 0.152 | 2.152 | 1.105 |
| 800000 | 0.555 | 3.652 | 1.869 |
| 1600000 | 1.292 | 2.327 | 1.218 |
| 3200000 | 2.978 | 2.306 | 1.205 |
| 6400000 | 7.363 | 2.472 | 1.306 |
| 12800000 | 15.427 | 2.095 | 1.067 |
| 25600000 | 370713 | 2.445 | 1.29 |
| 51200000 | 91.834 | 2.435 | 1.284 |

The log ratio data for sort 3 give it a time complexity of O (N log N). This time complexity leaves two possibilities for determining sort, randomized quicksort or non-randomized quicksort. The easy way to tell between these methods was to have three runs for each sort and view the first changes the method made. While running sort 3, the first change the program made was different each time: first trial it switched “C” and “G”; second trial it switched “C” and “B”; and finally for the third trial it switched “A” and “F”. While running sort 5, the first change the program made was the same for each run: each time “A” and “G” were switched. Based on the time complexity being O (N log N) and how the method is randomized, sort 3 must be randomized quicksort. Although not needed, I will still provide what a full instance of randomized quicksort looks like (note: this sort method would switch values instead of shifting, so two values are highlighted).

Table 6: Changes in the array for sort 3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| D | G | B | E | F | C | A |
| D | C | B | E | F | G | A |
| F | C | B | E | D | G | A |
| F | E | B | C | D | G | A |
| B | E | F | C | D | G | A |
| E | B | F | C | D | G | A |
| E | B | A | C | D | G | F |
| D | B | A | C | E | G | F |
| C | B | A | D | E | G | F |
| A | B | C | D | E | G | F |
| A | B | C | D | E | F | G |

Sort 4 began with using the N and M values from the previous sort trial, and again the problem size was much too high to obtain usable data without taking very long. The values for N and M were changed to those used in sort 2 (these values were N of 250 and M of 256,000).

Table 7: Running time data for sort 4.

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **T (sec)** | **Ratio** | **Log Ratio** |
| 250 | 0.003 |  |  |
| 500 | 0.005 | 1.612 | 0.689 |
| 1000 | 0.016 | 3.485 | 1.801 |
| 2000 | 0.041 | 2.528 | 1.338 |
| 4000 | 0.031 | 0.759 | -0.399 |
| 8000 | 0.068 | 2.207 | 1.142 |
| 16000 | 0.259 | 3.805 | 1.928 |
| 32000 | 1.376 | 5.308 | 2.408 |
| 64000 | 7.989 | 5.805 | 2.537 |
| 128000 | 41.134 | 4.149 | 2.364 |
| 256000 | 163.149 | 3.966 | 1.988 |

The log ratio data for sort 4 give it a time complexity of O (N2). This time complexity again leaves two possibilities of selection sort and possibly worst case non-randomized quicksort. Again I would have to observe how the array was sorted in order to determine which sort method was used.

Table 8: Changes in the array for sort 4.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| D | G | B | E | F | C | A |
| A | G | B | E | F | C | D |
| A | B | G | E | F | C | D |
| A | B | C | E | F | G | D |
| A | B | C | D | F | G | E |
| A | B | C | D | E | G | F |
| A | B | C | D | E | F | G |

Sort method 4 starts at the first position of the array, scans the array to find which value should be at this position, and moves the right value to this position. Based on this pattern and having a time complexity of O (N2), this must be selection sort.

Although there is only one possibility for what sort 5 could be, the time complexity for this method was still determined, and the sort was observed just as a precautionary measure to ensure accuracy. The values of N and M were changed to those of sort 1, and 3 (N is 100,000 and M is 51,200,000).

Table 9: Running time data for sort 5.

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **T (sec)** | **Ratio** | **Log Ratio** |
| 100000 | 0.079 |  |  |
| 200000 | 0.088 | 1.115 | 0.156 |
| 400000 | 0.1 | 1.128 | 0.174 |
| 800000 | 0.341 | 3.419 | 1.774 |
| 1600000 | 0.726 | 2.13 | 1.091 |
| 3200000 | 1.876 | 2.583 | 1.369 |
| 6400000 | 3.649 | 1.945 | 0.96 |
| 12800000 | 9.893 | 2.711 | 1.439 |
| 25600000 | 21.308 | 2.154 | 1.107 |
| 51200000 | 52.005 | 2.441 | 1.287 |

As was expected, the time complexity for sort 5 was determined to be O (N log N). This is consistent with average or best case for non-randomized quicksort. Again, although the changes to the array were not necessary for determining this method, it will still be provided. Based on the time complexity, and that this method did not randomize through three runs, this must be non-randomized quicksort.

Table 10: Changes in the array for sort 5.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| D | G | B | E | F | C | A |
| D | A | B | E | F | C | G |
| D | A | B | C | F | E | G |
| C | A | B | D | F | E | G |
| B | A | C | D | F | E | G |
| A | B | C | D | F | E | G |
| A | B | C | D | E | F | G |

1. **Interpretation**

Sort method 1 is merge, sort method 2 is insertion, sort method 3 is randomized quicksort, sort method 4 is selection, and sort method 5 is non-randomized quicksort. Each of these assertions has data presented which proves which method each unknown must be. For every sort method, data was presented which determines the time complexity, which would halve the possible choices. Then, each sort method was observed through canvas on jGRASP, and for each method the sorting pattern was presented. These two pieces of data, when analyzed together, give definitive proof for which method is which.