COMP 2210 Empirical Analysis Assignment – Part B

Zachary Parker

September 26, 2016

Abstract

The concept of sorting is one that pervades the software engineering world, and an understanding of different sorting algorithms allows us to viably tackle large-scale problems in a more streamlined way. Although there exists a myriad of different sorting algorithms, we will be focusing on five: selection sort, insertion sort, merge sort, and two variations of quick sort (randomized and non-randomized). This experiment aims to identify the sorting behavior of five methods within the SortingLab class using empirical analysis and to make conclusions based on gathered data. Through the use of the provided SortingLabClient and analysis of the results of running the client, it can be concluded that sort1 implements merge sort, sort2 implements selection sort, sort3 implements insertion sort, sort4 implements non-randomized quicksort, and sort5 implements randomized quicksort.

1 Problem Overview

The task at hand involves identifying the sorting algorithm used in each of the five sorting methods in the SortingLab class. In order to make these identifications, we will look at the time complexities of each method, in addition to their stability. Once we know of each individual method's approximate time complexity and its presence of stability, we will be able to successfully discern them.

The sorting algorithm assigned to each method will be randomized according to the "key" parameter used in the SortingLab(int key) constructor. We will each use our Auburn University Banner IDs as a means of generating a randomized set of five sorting algorithms, each tied to a specific sort method. One common theme among every algorithm's time complexity is the following equation, which tells us that as we double N, we will reach a constant equal to 2^k (where the log of this constant equals k, thus giving us the time complexity):

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

In order to find each method's underlying sorting algorithm, we will need to obtain the time complexity of each method by analyzing timing data and finding k as it converges. In addition, we will need to identify if a certain method is stable. An algorithm is considered stable if it successfully maintains the relative order of objects with equal values. The following chart will

lay out the following assumptions we must make about the four sorting algorithms we will be dealing with, including (but not limited to) their time complexities and stability:

Chart 1: Four Sorting Algorithms and Their Behavior

	Selection	Insertion	Mergesort	Quicksort
Worst case	$O(N^2)$	$O(N^2)$	O(N log N)	$O(N^2)$
Average case	$O(N^2)$	$O(N^2)$	O(N log N)	O(N log N)
Best case	$O(N^2)$	O(N)	O(N log N)	O(N log N)
In-place?	Yes	Yes	No	Yes
Stable?	No	Yes	Yes	No
Adaptive?	No	Yes	No	No

Analyzing this chart will help us discern selection sort/insertion sort from merge sort/quicksort due to their vastly differing time complexities. It will also help us discern selection sort from insertion sort, as well as merge sort from quicksort, due to their differences in stability. We also know from this chart that non-randomized quicksort will yield a worst-case scenario, $O(N^2)$, rather than randomized quicksort which will yield a more average case. With these things in mind, we can begin to collect data.

2 Experimental Procedure

In the SortingLabClient class, the "key" must be changed to one's own individual AU Banner ID. This ensures that each new SortingLab object is unique, even if the final order of the sort methods and their respective sorting algorithms are not. The default key is 9031111111, but I change it to my own ID as follows:

```
int key = 903550375;
```

After the key is changed, the client can be run and the results observed. The results will print an N (problem size) column and an elapsed time column. It should be inherently clear that as the problem size increases, the elapsed time will increase as well. This means that each successive line of code being printed will take longer than the last.

The problem size defaults at ten thousand to begin, and the size doubles in every iteration. Additionally, a limit of two million is set so that the client ceases running when the limit is reached. This is demonstrated by the following loop in the SortingLabClient class:

```
int M = 2000000; // max capacity for array
int N = 10000;   // initial size of array
for (; N < M; N *= 2) {
   ...
}
```

When running the sort methods, this starting point and limit is viable in the case of sort2 and sort3. However, in the other cases where the sorts are being completed extraordinarily quickly, it is better to start with a higher N value and a higher limit to compensate. The starting value and limit can be changed for sort1, sort4, and sort5 as such:

```
int M = 40000000 // forty million
int N = 500000; // five hundred thousand
```

These values yield much cleaner results for sort2 and sort3. With these more discernible elapsed time values, we can go on to find the time complexity of each method.

In the case of each method, we need to find the ratio of the current and previous times (denoted as "R") and $\log_2 R$ (denoted as "k", and the prime indicator of the method's time complexity). Finding R in each iteration of the loop is simple:

```
ratio = (elapsedTime / prevTime);
```

The double "ratio" is simply the elapsed time of the current run divided by the elapsed time of the previous run. Now, to find k, we must find $\log_2 R$. The Math class does not supply a way of finding $\log_2 x$, and so we must find an algebraic workaround. This results in the following code, where k is calculated as ln(R)/ln(2):

```
k = (Math.log(ratio) / Math.log(2));
```

These are all of the pieces necessary to find the time complexity of each sort method. However, in addition to finding their time complexity, we also need to determine whether or not they are stable in order to differentiate the methods that are otherwise indistinguishable. In order to achieve this, a separate class needs to be created. The Tuple class will act as a way to place two fields into a single, sortable Tuple object. Only the first field will be compared, and the second field will act as a stability test; if the second objects stay in their original order, then the method is considered to be stable. The Tuple class needs to implement Comparable on itself and pass in two fields to its constructor. In this case, the fields were of type Integer:

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

Of course, the class also requires a compareTo() method, as per its implied contract with Comparable. All of its comparisons should be made in terms of the Integer field1 so that field2 will be uninvolved in any comparisons. This way, if field2 appears out of order in any case, the method being used is guaranteed to not be stable. In order to ensure results are accurate, three Tuple arrays are created and separately called with a SortingLab<Tuple> object.

```
SortingLab<Tuple> pair = new SortingLab<Tuple>(key);

Tuple[] tupleArray = {new Tuple(5, 2), new Tuple(5, 1), ... };
Tuple[] tupleArray2 = { ... };
Tuple[] tupleArray3 = { ... };

pair.sort5(tupleArray);
pair.sort5(tupleArray2);
pair.sort5(tupleArray3);
```

With these two ways of identifying each method's sorting algorithm—a test of time complexity and a test of stability—we can ensure that our results and conclusions regarding the data will be accurate.

In my case, this experiment was conducted on a laptop computer running Windows 8.1, using Version 8, Update 101 of Java and jGRASP v2.0.3. Results in terms of determining time complexity can vary from machine to machine, but every machine should be able to reach the same conclusions. Of course, the test of stability should be identical on every machine. With these two ways of identifying each method's sorting algorithm, we can begin to collect data and come to a conclusion.

3 Data Collection and Analysis

Three sets of data are taken for each sort method to help ensure that each method's k values are not comparatively misleading in any way. The data set with the most apparently average k values is chosen in each case. The following five charts show each method's problem size (N), along with the corresponding elapsed time, R values, and k values:

Timing Table 1: sort1

N	Time (s)	R	k
500,000	0.66	-	-
1,000,000	1.14	1.73	0.79
2,000,000	2.44	2.14	1.10
4,000,000	6.08	2.49	1.32
8,000,000	11.84	1.95	0.96
16,000,000	27.66	2.34	1.23
32,000,000	58.85	2.13	1.09

Timing Table 2: sort2

N	Time (s)	R	k
10,000	0.51	-	-
20,000	2.08	4.08	2.03
40,000	5.10	2.45	1.29
80,000	25.37	4.97	2.31
160,000	323.79	12.76	3.67

Timing Table 3: sort3

N	Time (s)	R	k
10,000	0.54	-	-
20,000	1.97	3.65	1.87
40,000	7.18	3.64	1.86
80,000	27.63	3.85	1.94
160,000	147.50	5.34	2.42

Timing Table 4: sort4

N	Time (s)	R	k
500,000	0.74	-	-
1,000,000	1.43	1.93	0.95
2,000,000	3.04	2.13	1.09
4,000,000	6.60	2.17	1.12
8,000,000	14.06	2.13	1.09
16,000,000	31.23	2.22	1.15
32,000,000	75.17	2.41	1.27

Timing Table 5: sort5

N	Time (s)	R	k
500,000	0.55	-	-
1,000,000	0.75	1.36	0.44
2,000,000	1.73	2.31	1.21
4,000,000	3.73	2.16	1.11
8,000,000	8.12	2.18	1.12
16,000,000	18.60	2.29	1.20
32,000,000	39.70	2.13	1.09

Based on this data, we can see that both sort2 and sort3 have disproportionately higher k values than the others. In both of these methods, the values hover around 2 or greater, pointing to a time complexity of at least $O(N^2)$. Because only insertion sort and selection sort should be maintaining these ranges, we can conclude that sort2 and sort3 implement insertion sort and collection sort in some order. We also see that sort1, sort4, and sort5 maintain much lower k values, pointing to a time complexity of O(NlogN). This means that these three methods are merge sort, randomized quicksort, and non-randomized quicksort in some order.

To further differentiate the methods, we must look at their stability. Using the aforementioned Tuple arrays, we can see if a method maintains stability by looking at the pre-sorted and post-sorted order of the field2 values in an array. Take, for example, the pre-sorted and post-sorted order of values of the following Tuple array when calling the sort2 method:

Chart 2: Tuple[] tupleArray

Before Sort	(5, 2)	(3, 1)	(5, 4)	(3, 3)	(5, 1)	(3, 2)	(1, 1)
After Sort	(1, 1)	(3, 1)	(3, 3)	(3, 2)	(5, 1)	(5, 4)	(5, 2)

Notice that the first value in each pair is correctly sorted in ascending order: 1-3-3-3-5-5-5. For the second values, we must look at their original comparative order and their sorted order. The original order of the values with the first number “3” are as follows:

(3, 1), (3, 3), (3, 2)

Their post-sorted order in terms of the second values was the same: 1-3-2. So far, we can consider this method stable. Now we need to look at the original pairs with a first value of “5”:

(5, 2), (5, 4), (5, 1)

In this case, the secondary values are in an alternate order; the pre-sorted sequence is 2-4-1, but the post-sorted sequence is 1-4-2. Because these secondary values are out of order, we can finally conclude that this method is, in fact, unstable. We already know that sort2 is either insertion

or selection sort based on its time complexity, so we can now conclude that it is selection sort since it is not stable.

Because three Tuple arrays are created, this process is repeated twice for the same method in order to ensure absolute confidence in the procured results. Using this exact method, we can find the stability of every other method in the same way. The following chart provides the results of each method's stability:

Chart 3: Method Stability

sort1	stable
sort2	not stable
sort3	stable
sort4	not stable
sort5	not stable

Now that the timing complexity data and stability procedure's results have been thoroughly explained, we are now able to make final conclusions.

4 Interpretation

The results that have been procured are logically congruent with assumptions that have already been made. As Chart 1 illustrated, the time complexities of insertion sort and selection sort are, generally, $O(N^2)$. After analyzing the k values of the methods in each of the five Time Tables, we found that sort2 and sort3 were the candidates between these two sorting algorithms. We also know from Chart 1 that insertion sort is stable while selection sort is not. Based on Chart 3, we can finally conclude that sort2 implements selection sort and sort3 implements insertion sort.

We can make similar conclusions about sort1, sort4, and sort5. Based on Chart 1 and the five Timing Tables, we know that the three remaining methods have approximate time complexities of O(NlogN). Merge sort will be stable, while both of the quicksort methods will be unstable. Based on these assumptions and the information found in Chart 3, we know that sort1 implements merge sort, and sort4 and sort5 implement some variation of quicksort.

The final logical step in concluding our analysis is to understand that randomized quicksort will tend to be more efficient than non-randomized quicksort, which will be assumed to represent a worst-case scenario. Because sort5's k values and overall runtimes were smaller than those of sort4 (found in Timing Tables 4 and 5), we can finally conclude that sort5 implements randomized quicksort while sort4 implements non-randomized quicksort.

Working with these five sorting methods in real-time puts into perspective the importance of choosing the right algorithm for the given situation. Small-scale problems may not necessitate any particular algorithmic solution, but as problem sizes begin to enlarge, so do the issues we face if we find ourselves utilizing an algorithm that lacks efficiency and scalability. It is for this reason that so much emphasis is placed upon the optimal usage of these various sorting algorithms.