COMP 2210 Assignment 3 Part 2
Group 28

Tyler Jewell **Jentry Chesnut**

February 23, 2016

Abstract

The motivation for this experiment is to observe empirical data to determine the identity of each sorting algorithm being used in a3resources.jar. The five possible sorts are selection sort, insertion sort, mergesort, quicksort with no randomization, and randomized quicksort. One way to identify the sorts is to utilize timing data gathered from SortingLabClient.java, which invokes the sorting methods in a3resources.jar. Another way is to observe how the list is being modified as it is sorted, since every sorting algorithm has a unique signature to how it sorts we can distinguish each sort correctly.

1. Problem Overview

The objective of the experiment is to determine which of the five sorts from a3resources.jar are being used in each of the five different calls to the sort() method in the SortingLabClient.java. The determination of the sorts should be based off of empirical data and methods. To determine the different types of sorting methods the constant value k, from Formula 1 below, can be observed as a means to determine the algorithms big-Oh notation, which can be compared to the theoretical big-Oh values of each sort. This method will allow us to eliminate some of the sorts from consideration. In the event that we can't determine the sort from the big-Oh notation, we observe the modifications of the list that is being sorted by a single sort() call in order to identify it.

$$T(N) \propto N^K \Rightarrow T(2N)/T(N) \propto (2N)^k/N^k = 2^k N^k/N^k = 2^k$$

Formula 1

2. Experimental Procedure

The machine used for this data is a Windows 8.1 OS with an Intel i5 processor and 12 GB of RAM running JDK 8 and JRE 8.

```
for (; N < M; N *= 2) {
                Integer[] ai = getIntegerArray(N, Integer.MAX_VALUE);
                start = System.nanoTime();
                sli.sort3(ai);
                elapsedTime = (System.nanoTime() - start) / 1000000000d;
                System.out.print(N + "\t");
                System.out.printf("%4.3f\n", elapsedTime);
}
```

Snippet 1: Loop used to gather timing data for the various sorts. In each run the sort() method was changed.

Snippet 1 above is a loop used to sort algorithm and print the problem size, N, as well as its runtime. This snippet of code instantiates a new array of type Integer, starts a timer in nanoseconds, sorts the array that is given using an undetermined sorting algorithm, and prints the result. Using this data, we can narrow down the possible sorting algorithm being used. After gathering timing data from every sort we chose a word to be sorted by the array in order to observe the modifications. The word chosen was Sephirot, which is the Tree of Life. With each run of the program, we made a canvas in JGrasp and dragged the List that was being sorted through and observed as it changed and recorded each change that occurred. By taking this approach, distinguishing between each sort was somewhat trivialized.

3. Data Collection and Analysis

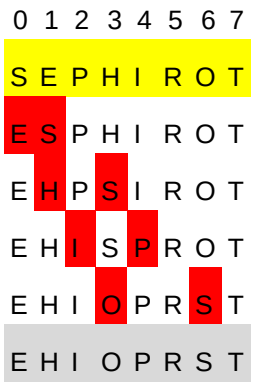

Figure 1: Visual Representation of Sort1() given an array {S,E,P,H,I,R,O,T}. The numbers represent the element number, the yellow row is the starting array, the red represents the modifications to the array, and the gray row is the array in its sorted order.

Sort 1			
N	**Elapsed Time (sec)**	**R**	**k**
10000	0.099	—	—
20000	0.421	4.253	2.088
40000	1.617	3.841	1.841
80000	8.517	5.045	2.335
160000	28.457	3.341	1.74
320000	158.289	5.562	2.476
640000	1132.822	7.157	2.839
1280000	6011.037	5.306	2.408

Table 1: The first set of sorting data that was generated by the SortingLabClient.java, Where N is the problem size, R is the ratio, and k is the constant value.

	Selection	Insertion	Mergesort	Quicksort
Best-Case	$O(N^2)$	O(N)	O(NlogN)	O(NlogN)
Worst-Case	$O(N^2)$	$O(N^2)$	O(NlogN)	$O(N^2)$
Average-Case	$O(N^2)$	$O(N^2)$	O(NlogN)	O(NlogN)
In-Place	Yes	Yes	No	Yes
Stable	No	Yes	Yes	No
Adaptive	No	Yes	No	No

Diagram 1: Each sorting algorithm's properties

To Identify the sorting algorithm being applied in sort1() we can observe figure 1. In the figure, the first modification is between elements 0 and 1, which doesn't rule out the insertion sort, but the second modification does because it 'H' is the second lowest order in the array. As the sort continues, it can be determined that it is a Selection sort based on the fact that it is scanning the whole array before it makes a modification, so every modification of the array puts

the element in the place it should be in sorted order. To further back up this claim, according to the data gathered in the table k is trending around two which gives a big-Oh notation of $O(N^2)$, which is consistent with Selection sorts best, worst, and average case.

Figure 2: Sort2() #1 retains properties from figure 1.

Sort 2			
N	**Elapsed Time (sec)**	**R**	**k**
10000	0.005	—	—
20000	0.016	3.2	1.678
40000	0.018	1.125	0.17
80000	0.015	0.0008	-10.288
160000	0.032	2.133	1.093
320000	0.06	1.875	0.907
640000	0.132	2.2	1.138
1280000	0.333	2.523	1.335

Table 2: Same as Table 1, but data is for sort2()

Figure 3: Sort2() #2 Worst case assortment

To identify the sorting algorithm being applied in sort2() we can observe figure 2 and figure 3. Figure 2 shows that the elements seem to be moving around element 'R' as if it's a pivot, and this is the case throughout the entire sorting procedure. Given that 'R' is the pivot point, the observation can be made that the goal of the sorting algorithm is to place elements that are less than the pivot to the left of the pivot and elements that are greater than the pivot to the right of the pivot in order to sort the array. By this, it can be concluded that the sorting algorithm being used is either a quicksort with randomization or without randomization. To find out if its randomization or not, we must do another test for the worst case scenario for the quicksort, which is an array sorted in descending order. Figure 3 shows the array being sorted from descending order. It appears that element 'O' is the pivot in this case, which makes the sort behave almost like a Selection sort, which shows that this is a quicksort with randomization.

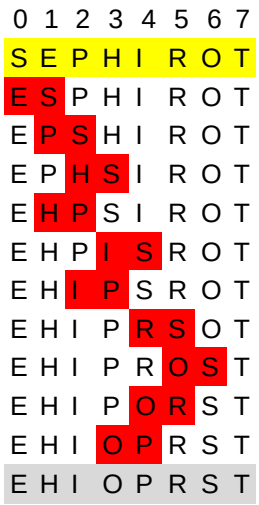

Figure 4: Sort3() retains properties of figure 1

Sort 3			
N	**Elapsed Time (sec)**	**R**	**k**
10000	0.137	—	—
20000	0.519	3.788	1.921
40000	2.034	3.919	1.97
80000	9.738	4.788	2.259
160000	45.691	4.692	2.23
320000	183.478	4.016	2.006
640000	1342.126	7.315	2.871
1280000	7256.37	5.407	2.435

Table 3: Same as Table 1, but data is for sort3()

To identify the sorting algorithm applied in sort3() we can observe figure 4. In figure 4, the first swap is between elements 0 and 1, which is similar to how it occurred in the Selection sort in sort1() since element 'E' is the least element in the array, but the next modification that occurs reveals the nature of the sorting algorithm in sort3(). Element 1 and 2 are swapped. If this was a Selection sort, then element 'P' would be the second least element of the array, but it is not because the second least element of the array is element 'H', so this reveals that this is an Insertion sort. To further back up this assertion, the value k is trending toward two most of the time, which means the big-Oh notation is $O(N^2)$, which is consistent with the worst and average case of an Insertion sort.

Sort 4			
N	**Elapsed Time (sec)**	**R**	**k**
10000	0.006	—	—
20000	0.007	1.167	0.223
40000	0.009	1.286	0.363
80000	0.016	1.778	0.83
160000	0.038	2.375	1.248
320000	0.088	2.316	1.212
640000	0.236	2.682	1.423
1280000	0.551	2.335	1.223

Table 4: Same as Table 1, but data is for sort4()

0	1	2	3	4	5	6	7
S	E	P	H	I	R	O	T
S	T	P	H	I	R	O	E
S	T	P	H	I	O	R	E
S	T	P	O	I	H	R	E
S	I	P	O	T	H	R	E
I	S	P	O	T	H	R	E
I	E	P	O	T	H	R	S
I	E	H	O	T	P	R	S
H	E	I	O	T	R	R	S
E	H	I	O	T	P	R	S
E	H	I	O	S	P	R	T
E	H	I	O	R	P	S	T
E	H	I	O	P	R	S	T
E	H	I	O	P	R	S	T

Figure 5: Sort4() #1 retains properties of figure 1

0	1	2	3	4	5	6	7
S	T	R	P	O	I	H	E
S	T	R	P	O	E	H	I
S	T	R	P	O	H	E	I
S	T	H	P	O	R	E	I
O	T	H	P	S	R	E	I
H	T	O	P	S	R	E	I
T	H	O	P	S	R	E	I
I	H	O	P	S	R	E	T
I	H	E	P	S	R	O	T
E	H	I	P	S	R	O	T
E	H	I	P	O	R	S	T
E	H	I	O	P	R	S	T
E	H	I	O	P	R	S	T

Figure 6: Sort4() #2 Worst case assortment

To identify the sorting algorithm being applied in sort4(), we can observe figures 5 and 6. The initial modification or swap is between element ‘T’ and ‘E’ after several modifications to the array the pivot appears to be at element ‘O’ because after the element ‘H’ swaps with element ‘O’ all modifications to the array take place around the pivot. Given that information it is apparent that sort4() is a quicksort. In order to determine whether it’s a random quicksort or not, we must do another test for the worst case scenario, which is shown in Figure 6. The pivot in this case appears to be at element ‘P’, which would show that this in fact a quicksort without randomization.

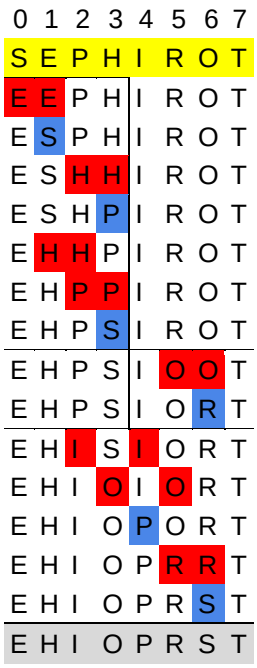

Figure 7: Sort5() retains properties of figure 1

Sort 5			
N	**Elapsed Time (sec)**	**R**	**k**
10000	0.004	—	—
20000	0.025	6.25	2.644
40000	0.038	1.52	0.604
80000	0.021	0.553	-0.855
160000	0.04	1.905	0.93
320000	0.087	2.175	1.121
640000	0.193	2.218	1.149
1280000	0.464	2.404	1.265

Table 5: Same as Table 1, but data is for sort5()

To identify the sorting algorithm being applied by sort5(), we can observe Figure 7. In the figure, the first modification is odd as it places a duplicate of the element that it will apply a swap. Through observing Figure 7, we determined that the array was being split in two, with one side being sorted, then the other side being sorted. This is the “Divide and Conquer” strategy. After being “conquered” the arrays are combine then sorted into their properly sorted order, which makes this a Mergesort.

4. Interpretation

Figure 8 shows graphically the time vs problem size of each. The graph does not consider worst case time complexity for quicksort. In conclusion with the results that were expounded upon in section 3 of the lab report, sort 1 is Selection sort, sort 2 is Quicksort with randomization, sort 3 is Insertion sort, sort 4 is Quicksort without randomization, and sort 5 is Mergesort. Through all of the data gathered throughout the report, we were also able to determine that each sorting algorithm has its own unique sorting pattern that distinguishes it from the other sorts.

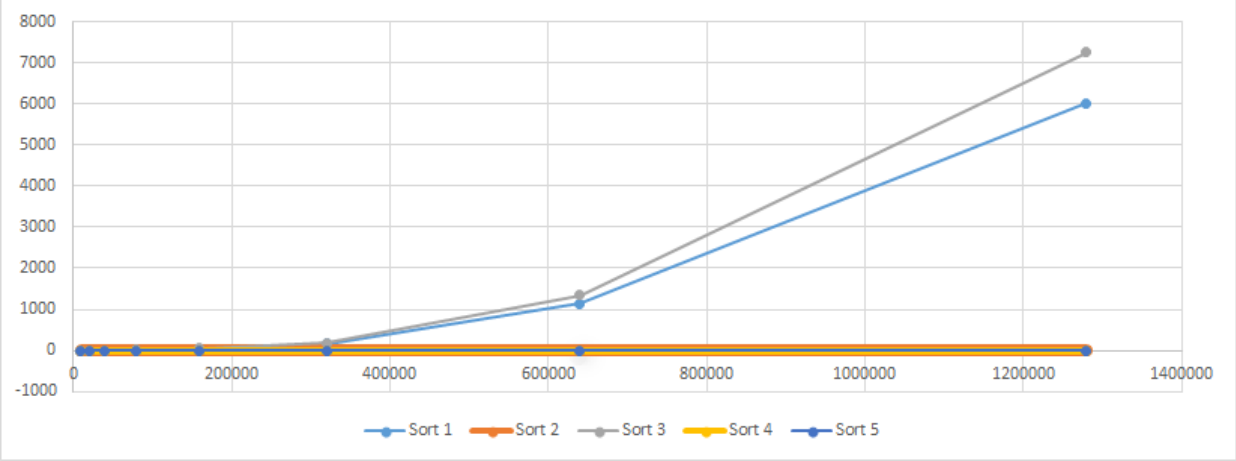

Figure 8: The graph of each sorting method