Comp 2210 Empirical Analysis Assignment

Haden Stuart

October 2, 2018

Abstract

Efficient coding is one of the most important parts of developing algorithms but finding out just how efficient your code is can take some extra work. Using analysis and some testing we can develop times and ratios to examine exactly how efficient something can be. Running the SortingLabClient.java will provide enough information to empirically decide how efficient the code is.

Problem Overview

In this experiment we will be using empirical analysis and the scientific method to solve the provided puzzle. We are provided with a **SortingLabClient.java** that will be used to run five different sorting methods which are: selection sort, insertion sort, merge sort, randomized quicksort, and non-randomized quicksort. Each of these sorting methods will run in a certain amount of time which we will use to calculate a ratio for the method. We are not allowed to access the actual SortingLab file, but we can use the debugger and canvas viewer to observe which elements are being moved in each of the sorting methods. Using this information, we will be able to figure out the Big-O for each of the methods which will provide us with enough information to determine which sort is being used. This program starts out asking for a key in the form of a student id number, mine being (903725103). Using this id key, the program will develop different values to use in the sort methods to provide the data for analysis.

Experimental Procedure

Using the table below we will be able to record each of the timings for the methods and then get each of the ratios based on the times. To get these timings we must run the **SortingLabClient.java** that will process the values using each of the sorting methods. To get these times, we will need to test each sort method individually to find out exactly which sort method is being used. Once we have calculated the ratios we can calculate the Big-O value by using the equation provided on the problem worksheet (also posted below). Another very useful piece of information that we will be looking at is the debugger and the canvas viewer. Using these helpful tools, we will be able to follow exactly which values in the array will be sorted into their places. With both processes combined we should have no difficulty determining which sorting methods are being used.

Numbers	Time	Ratio
N_0	$T(N_0)$	None
N_1	$T(N_{1)}$	$T(N_1)/ T(N_0)$
N_2	$T(N_2)$	$T(N_2)/ T(N_1)$
N_n	$T(N_n)$	$T(N_n)/ T(N_{n-1})$

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

Data Collection and Analysis

I started the process by running each sort method a few times and then recording the times. After this process I calculated the ratio for each to find the Big-O value. Once that was completed, I went back to the program and watched the canvas viewer for each sorting method and then recorded the changed values with orange. Another very important piece of information when it comes to finding out efficiency is what finding out what your computer can handle. With more recent hardware and software you will be able to get much faster times for larger processes. Since we need as much information as possible to get the most accurate result, recording computer specifications will also be beneficial to this collection.

Computer Specs:

- Intel® Core™ i7-6500U CPU @ 2.50GHz with 8.00GB installed ram
- Windows 10 Operating System

Sort 1.)

Below we have the timings for sort 1 in the table on the left and the results from the canvas viewer in the table on the right.

N	Time	Ratio
10000	0.004	None
20000	0.008	2.00
40000	0.016	2.00
80000	0.037	2.31
160000	0.070	1.89
320000	0.213	3.04
640000	0.380	1.78

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

Following the array of characters in the canvas viewer from the table on the right we can start to narrow down which sort is being used. We see on the first swap that it is swapping the first two characters on the left side of the array. Then it moves to the next two non-sorted values in the list and swaps those. It continues to do this on the left side until the first four characters are sorted, then it moves on to the right side. It makes two changes to the right side which then leaves the right side sorted as well. For the last change it places both sorted sides back together making the entire list sorted. We can now see that this method is merge sort based on it sorting each side and then merging the two sorted sides together. The time complexities for merge sort are Best: O(NlogN), Average: O(NlogN), and Worst: O(NlogN). Using the ratios we calculated above, we can see that the values are consistent around 2. Now we can figure out the Big-O value by doing: $2^k = 2$. By solving for k we get k = 1. This means that the Big-O value would be NlogN which is consistent to what we concluded.

Sort 2.)

Below we have the timings for sort 2 in the table on the left and the results from the canvas viewer in the table on the right.

N	Time	Ratio
10000	0.004	None
20000	0.007	1.75
40000	0.016	2.29
80000	0.034	2.13
160000	0.057	1.68
320000	0.151	2.65
640000	0.355	2.35

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

If we observe the canvas viewer results on the right, we see that the first swap that occurs is the A and the E elements. Since these two are still not in their correct placements, this rules out selection sort. Since A and E were the values swapped, we could hypothesize that a pivot point is being used with a value between these two characters (B, C, D). From the problem worksheet, we know that both quicksort methods will choose their pivot point from the left most element. In this case D is the left most element which also coincides with our theory before. As we follow the sorting we see that D gets moved to its correct place by swapping with C. For the rest of the sort, we can see that D stays in its correct place while all elements larger than it are placed to the right and all elements smaller than it are placed to the left. With this information we can conclude that this is the non-randomized quicksort. The time complexities for quicksort are Best: O(NlogN), Average: O(NlogN), and Worst: $O(N^2)$. Using the ratios we calculated above, we can see that the values are consistent around 2. Now we can figure out the Big-O value by doing: $2^k = 2$. By solving for k, we get k = 1. This means that the Big-O value would be NlogN which is consistent to what we concluded.

Sort 3.)

Below we have the timings for sort 3 in the table on the left and the results from the canvas viewer in the table on the right.

N	Time	Ratio
10000	0.485	None
20000	2.005	4.13
40000	7.350	3.67
80000	29.820	4.06
160000	110.356	3.70
320000	661.971	6.00
640000	2670.540	4.03

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

If we follow the swaps in the table on the right, we should be able to determine exactly which sort is being used. The first swap occurs with the smallest character in the list (A) being moved from the next to last position, to the first position. The next smallest value is already in place, so the sort continues by selecting the smallest then swapping it to the right place and moving on to the next value. Based on these movements we can determine that the sort being used is selection sort. The time complexities for selection sort are Best: $O(N^2)$, Average: $O(N^2)$, and Worst: $O(N^2)$. Using the ratios we calculated above, we can see that the values are consistent around 4. Now we can figure out the Big-O value by doing: $2^k = 4$. By solving for k we get k = 2. This means that the Big-O would be N^2 which is consistent with what we concluded.

Sort 4.)

Below we have the timings for sort 4 in the table on the left and the results from the canvas viewer in the table on the right.

N	Time	Ratio
10000	0.004	None
20000	0.006	1.50
40000	0.012	2.00
80000	0.027	2.25
160000	0.060	2.22
320000	0.140	2.33
640000	0.361	2.58

D	B	E	C	G	A	F
F	B	E	C	G	A	D
F	B	A	C	G	E	D
F	B	C	A	G	E	D
C	B	F	A	G	E	D
C	B	A	F	G	E	D
~~C~~	B	A	F	G	E	D
A	B	~~C~~	F	G	E	D
A	B	C	~~F~~	D	E	G
A	B	C	E	D	~~F~~	G
A	B	C	D	E	F	G
A	B	C	D	E	F	G

With the results in the right table we can begin to figure out what elements are being moved around. We see that the elements begin switching with each other, but they don't appear to be following any certain pattern. This process appears to be randomizing the list instead of sorting it, which proves that every movement above the gray line is part of a randomization. Once the randomization is finished, the sort picks the left most value as the pivot point (C). The sort then swaps C with A which moves C to its correct location. Now that the set of characters on the left of C are smaller and the set on the right of C are larger, the sort must pick a new pivot point. Using the same process as above, it picks the left most element in the non-sorted list (F) and then begins to move each character to its respective place. With this information we can deduce that the sort being used is the randomized quicksort. The time complexities for quicksort are Best: O(NlogN), Average: O(NlogN), and Worst: $O(N^2)$. Using the ratios we calculated above, we can see that the values are consistent around 2. Now we can figure out the Big-O value by doing: $2^k = 2$. By solving for k we get k = 1. This means that the Big-O value would be NlogN which is consistent to what we concluded.

Sort 5.)

Below we have the timings for sort 5 in the table on the left and the results from the canvas viewer in the table on the right.

N	Time	Ratio
10000	0.240	None
20000	0.972	4.05
40000	3.892	4.00
80000	16.315	4.19
160000	67.434	4.13
320000	442.676	6.56
640000	2823.613	6.38

D	B	E	C	G	A	F
B	D	E	C	G	A	F
B	D	C	E	G	A	F
B	C	D	E	G	A	F
B	C	D	E	A	G	F
B	C	D	A	E	G	F
B	C	A	D	E	G	F
B	A	C	D	E	G	F
A	B	C	D	E	G	F
A	B	C	D	E	F	G

In the table on the right we can watch each of the characters as they begin to be shifted around. We see that the first swap starts at the beginning of the list as B is smaller than D. The sort continues and swaps the next two elements in the list but then it continues to move the smaller character until it is sorted in its correct place. Whenever the smallest value (A) is found near the end of the list, it is slowly moved from place to place until it is finally placed at the front of the list. These movements lead us to believe that this is the insertion sort due to it inserting an element in front of the previous element until it reaches its correct place. The time complexities for insertion sort are Best: O(N), Average: $O(N^2)$, and Worst: $O(N^2)$. Using the ratios, we calculated above, we can see that the values are consistent around 4. Now we can figure out the Big-O value by doing: $2^k = 4$. By solving for k we get k = 2. This means that the Big-O would be N^2 which is consistent with what we concluded.

Conclusion

Time complexity can be a very important factor when it comes to writing code. Making a solution to a problem is only beneficial if the time it takes to solve the problem is feasible. As we saw above, each sort method solved the given problem, but some had much quicker return times. Looking back at each sort we can finalize the experiment and say that:

- Sort 1 == Merge sort
- Sort 2 == Non-randomized Quicksort
- Sort 3 == Selection sort
- Sort 4 == Randomized Quicksort
- Sort 5 == Insertion Sort