Experimental discovery of running Time

Abstract

Finding the efficiency of an algorithm is important to the development process of writing software, but the way of calculating the efficiency can be very subjective in relation to how the calculating is done. By calculating the efficiency of an algorithm empirically, through experimenting and observation, one has to rely on the processing of the computer that is running the algorithm. The processing of a computer can be different from one system to the next. Thus this experiment to discover the big-Oh running time of RunningTime.timeTrial() will be subjective based on the calculating process of my system and the algorithm that was used to achieve the efficiency of the big-Oh in the experiment.

Introduction

This experiment of finding the big-Oh in RunningTime.timeTrial() is being conducted using an empirical approach which is by experimenting and observation of the time timeTrial() method and the special method it calls to when giving a parameter of timeTrial(N, 31), which 'N' is the problem size and “31” is the 'seed' that corresponds to the call to the special method that is needed to take time to process in order to find the big-Oh running time of the method. The objective of this experiment is to provide the client with the data needed to discover the big-Oh running time of the method established earlier. The importance of this experiment is to have an experience of applying the scientific method to find the big-Oh running time of an algorithm.

Procedures

When starting this experiment I had to figure out a way to run the experiment a number of times without having to change the problem size each time. The instructor provided a simple example of a client to use, which would require changing the problem size each time, which was the opposite of what was need to be done. I looked at the power point slide 4 of “ Algorithm_Analysis.pptx.pdf” which

gave an example of what might be expected as an output. I needed to write a loop that would execute a number of times and calculate the ratio and log ratio for each iteration of the problem size change and print the output to the console. In order for the ratio to be calculated there must be a previous run which that meant that the first iteration would not have a ratio also it was apparent in the power point slide

mentioned earlier. The formula used to calculate the ratio is $Ratio = \frac{Current\ Time}{Previous\ Time}$, where current time is the present iteration time and the previous time is the last iteration's time. To calculate the log ratio I needed to use the Math class and the static methods of it in order to calculate the formula

for the log ratio which is $\log Ratio = \log_2(\frac{Current\ Time}{Previous\ Time})$. After being let down that the Math class does not provide a way to obtain base two easily I resorted to use algebra to provide a way using the static methods provided to obtain the base two equivalent of the ratio. The formula used to obtain this is

$\log_2 Ratio = \frac{\log(\frac{Current\ Time}{Previous\ Time})}{\log(2)}$. After having the math worked out the rest of writing the code to obtain the big-Oh of the RunningTime.class was a breeze. After writing the code to obtain the big-Oh, I executed the byte code that was complied and up and behold it worked and on the other hand it did not work. It worked in the aspect of it complied and done what it was meant to do. It did not work in the way of that it didn't get me meaningful data to work with (Figure 3). I needed to slow down the iterations of the loop to obtain meaningful data to work with. In order to achieve this I used the debug feature on JGrasp. I set a break-point at the for loop to and set the delay to '0' and enabled 'Auto Resume' and resumed the execution of the written code, this method worked to delay the loop and provided me with meaningful data to use. The explanation to why I choose to start the problem size to 1 and double in size to 128 has to do with other executions that were made while debugging and testing

the class RunningTimeClient.java. The other executions that were made resulted in prolonged durations of time spent waiting and the expectancy of the result to come in two hours time and the resulting time would not be needed to come to the same result.

Results and Discussion

Run	N	T(N)	Ratio	lg Ratio
0	1	.031		
1	2	.031	1.000	0
2	4	.232	7.484	2.970
3	8	1.333	5.746	2.522
4	16	10.441	7.833	2.969
5	32	83.477	7.995	2.999
6	64	667.589	7.997	2.999
7	128	5323.471	7.974	2.995

Run	N	T(N)	Ratio	lg Ratio
0	1	.032		
1	2	.031	.969	-.046
2	4	.223	7.194	2.847
3	8	1.335	5.987	2.582
4	16	10.466	7.840	2.971
5	32	83.519	7.980	2.996
6	64	668.003	7.998	3.000

Figure 1. The data that was generated by the source code to find the big-Oh, there are five different columns: Run, N, T(N), Ratio, lg Ratio. Each column represents essential data needed to empirically discover the big-Oh. The first generated result is from the initial test which has a added time that took approximately 1 hour and 47 minutes to generate. The second result is to see if the result from above could be replicated again with much success it did.

To discover the big-Oh by using figure 1 table of information one must look at the 'lg Ratio' column, this column represents the exponent that is required to have the result in the Ratio column. The Ratio column can be used to multiply the T(N) column which is in terms of seconds with three significant figures are to the millisecond, the result of such calculation is the expected result of the next iteration. In figure 1 such calculation can be done to not prove, but to demonstrate such result.

$$(668.003)\cdot(7.998)=5342.687\approx5343$$

Example 1. Shown to demonstrate how one could get the future result of how much time it would take to get the result of a problem size of N.

To prove such calculation is out of scope of this report. To demonstrate this same table in a graphical instance can be far more appealing then a table of numbers.

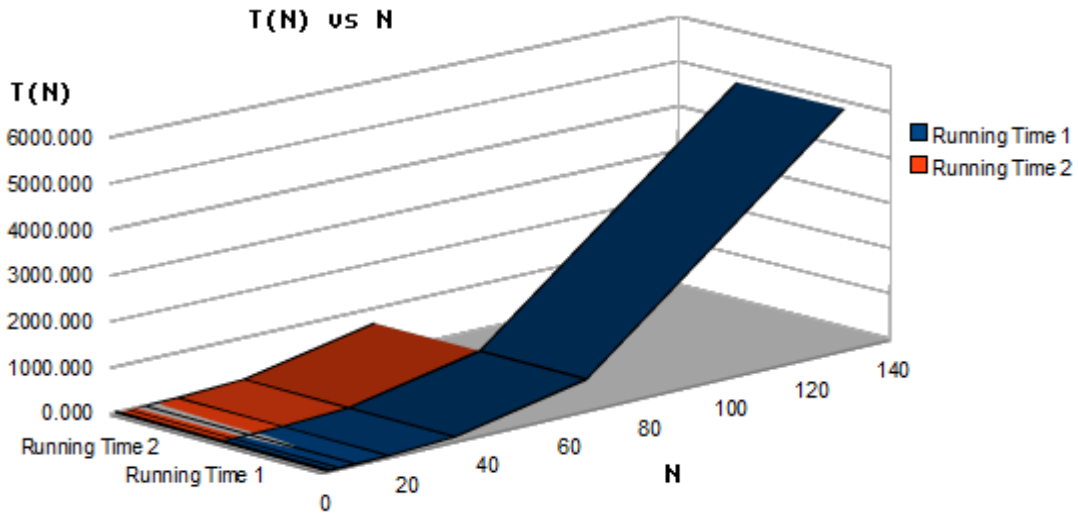

Figure 2. The graph depicts how time increases as the problem size increases. This graph also shows that running time 1 and running time 2 are equivalent and would most likely have close results if continued to execute.

The result of this experiment and the finial conclusion is that the big-Oh of the java class RunningTime.class is that of $O(N^3)$.

Conclusion

The conclusion that is made from this experiment is that it is very difficult to replicate the same results over and over again and not only to replicate it on just one system, but that it would be even more difficult to replicate the same results on an entirely different system. As long as the run time are consistent to result in a ratio of high 7 one can only have the same conclusion of $O(N^3)$. Below is the result the problem that occurred when not running the source code in debug mode.

Run	N	T(N)	Ratio	lg Ratio
0	1	.029		
1	2	.000	.000	-?
2	4	.000	?	?
3	8	.000	?	?
4	16	.001	?	?
5	32	.003	3.000	1.585
6	64	.005	1.667	.737
7	128	.000	.000	-?

Figure 3. This table shows the problem that occurred without running in debug mode to render meaningful data.

Appendices

The source and output of the program are located beyond this page.

```
1    import java.text.DecimalFormat;
2    /**
3     * RunningTimeClient.java. When executed a time trial
4     * is done against RunningTime.class to find the big-Oh.
5     *
6     * Compilation:   %javac RunningTimeClient.java
7     * Execution:     %java -ea RunningTimeClient
8     *
9     * @author    Walter James Conway (wjc0008@auburn.edu)
10    * @version   2013-02-10
11    *
12    */
13    public class RunningTimeClient {
14
15
16       public static void main(String[] args) {
17
18
19         Clock clock;              // measures elapsed time
20         double elapsedTime = 0;   // elapsed time of current run
21         double prevTime = 0;      // elapsed time of previous run
22         double ratio = 0;         // currentTime / prevTime
23         double lgratio = 0;       // log base 2 of ratio
24         int N = 1;               // problem size parameter
25         int seed = 31;          // selects internal method of RunningTime
26         DecimalFormat deci = new DecimalFormat("#.000");
27         System.out.println("----------------------------------"
28                            +"----------------------------------");
29         System.out.println("Run\tN\tT(N)\t\tRatio\t\tlg Ratio");
30         System.out.println("----------------------------------"
31                            +"----------------------------------");
32         for (int i = 0; i < 20; i++) {
33            clock = new Clock();
34            RunningTime.timeTrial(N, seed);
35            elapsedTime = clock.elapsedTime();
36            System.out.print(i + "\t" + N + "\t" + deci.format(elapsedTime));
37            if (i != 0) {
38               ratio = (elapsedTime / prevTime);
39               lgratio = (Math.log(ratio) / Math.log(2));
40               System.out.print("\t\t" + deci.format(ratio)
41                                + "\t\t" + deci.format(lgratio) + "\n");
42            }
43            else {
44               System.out.println();
45            }
46            N *= 2;
47            prevTime = elapsedTime;
48         }
49      }
50   }
```

Run	N	T(N)	Ratio	lg Ratio
0	1	.031		
1	2	.031	1.000	0
2	4	.232	7.484	2.970
3	8	1.333	5.746	2.522
4	16	10.441	7.833	2.969
5	32	83.477	7.995	2.999
6	64	667.589	7.997	2.999
7	128	5323.471	7.974	2.995

Run	N	T(N)	Ratio	lg Ratio
0	1	.032		
1	2	.031	.969	-.046
2	4	.223	7.194	2.847
3	8	1.335	5.987	2.582
4	16	10.466	7.840	2.971
5	32	83.519	7.980	2.996
6	64	668.003	7.998	3.000