COMP 2210 Empirical Analysis Assignment – Part A

Zachary Parker

September 26, 2016

Abstract

It is emphatically important that we, as programmers, create code that is efficient and scalable. An algorithm that lacks viability in the context of large problems is one that would, in most of the situations that we can imagine, lack any semblance of viability at all. This experiment aims to identify the time complexity of a method within the TimingLab class using empirical analysis and to make conclusions based on gathered data. Through the use of the provided TimingLabClient and analysis of the results of running the client, it can be concluded that the time complexity of this specific instance of the method is $O(N^3)$.

1 Problem Overview

The task at hand involves finding the time complexity (or "Big-Oh") of a given method. The method in this case will be timeTrial(int N) method within the TimingLab class, where N is the problem size. As the problem size increases—specifically, as N doubles with each iteration—we should observe a clear increase in running time, although the rate of this increase could vary.

The variance in running time will stem from the "key" parameter used in the TimingLab(int key) constructor. We will each use our Auburn University Banner IDs as a means of generating a specifically-tailored time complexity. One common theme among every possible 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):

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

The goal of the experimental procedure will be to obtain the time complexity of the method by analyzing timing data and finding k as it converges.

2 Experimental Procedure

In the TimingLabClient class, the "key" must be changed to one's own individual AU Banner ID. This ensures that each new TimingLab object is unique, even if the final time complexity of the method is 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.

A total of five calculations will be made when the program is run. This is demonstrated by the following loop in the TimingLabClient class:

```
for (int i = 0; i < 5; i++) {
     start = System.nanoTime();
     tl.timeTrial(N);
     elapsedTime = (System.nanoTime() - start) / BILLION;
     System.out.print(N + "\t");
     System.out.printf("%4.3f\n", elapsedTime);
     N = N * 2;
}
```

The number of successive runs can be changed (e.g. i < 7 for a total of seven runs) so that more data can be procured, but this should not be necessary. In fact, patterns can be recognized before five results have even been printed. This is convenient, as the last result can likely take an amount of time not deemed practical. Note the last line of the code in which the problem size is doubled before moving to the next iteration in the loop.

It's important to note that 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 the method. 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 can vary from machine to machine, but every machine should be able to reach the same conclusion of the method's time complexity. Once a

few instances of N and the corresponding values have been printed by the program and recorded, we can begin to come to a conclusion.

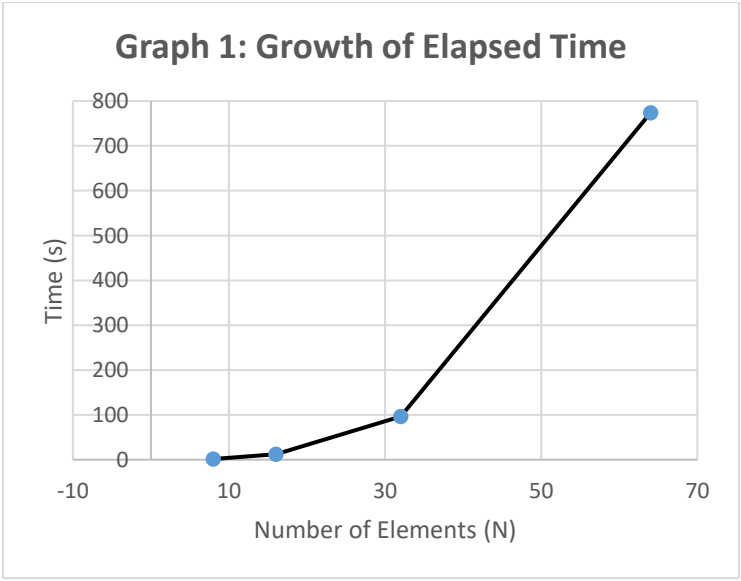
3 Data Collection and Analysis

Only four iterations of the loop were necessary to draw a substantial conclusion about the nature of the timeTrial() method. Therefore, there were only four values of N that were analyzed. The following table (Table 1) shows these values of N, along with the corresponding elapsed time, R values, and k values:

Table 1: Running-time Data

N	Time (s)	R	k
8	1.56	-	-
16	12.12	7.77	2.96
32	96.22	7.94	2.99
64	773.71	8.04	3.01

As N doubles in each iteration, the elapsed time increases exponentially. The ratio discussed earlier seems relatively constant, which means that k converges quickly and conspicuously. Looking at the k column, it is evident almost immediately that the time complexity function for this method is $O(N^3)$, as k nearly equals 3 in every measured instance. As N doubles in every instance, the elapsed time will be approximately 2^3 (or 8) times the previous time. The cubic nature of the method's time complexity can be more easily seen when N is plotted against the elapsed time on a graph:



Whereas a constant time complexity would be represented with a horizontal line and a linear time complexity would be represented with a straight line rising to the right, this line instead has a constantly increasing slope, more specifically representing a time complexity of $O(N^3)$. As N increases, the time will increase at a much faster rate, leading to a largely inefficient and impractical solution as the number of elements increases.

4 Interpretation

The main conclusion to gather from this experiment is how inefficient a cubic time complexity truly is. Based on the information found in Table 1, a list of a mere 64 elements took minutes to complete – nearly 13 minutes, to be precise. Graph 1 tells a similar story and allows us to picture how horrendously inapplicable this version of the timeTrial() method would be in large-scale situations. If the number of elements were increased to thousands or (let us try not to imagine) millions of elements, we would certainly find ourselves with an immediate call to rework our algorithmic approach.

Being able to work with this algorithm in real-time puts into perspective the importance of algorithm management and scalability. If we do not include efficiency within our definition of robust code, then surely we will run into some Big-Oh problems along the way.