COMP 2210 Assignment 3 Part A

Group 25

Benjamin A. Cyr Mark A. Gallagher Jr

September 18, 2014

Abstract

This document presents an experiment to empirically determine the big-Oh time complexity of the `timeTrial(int N)` method in the `TimingLab` class. This was done by running the method with various problem sizes and recording the running time. The relationship between the ratios of consecutive running times determined the big-Oh time complexity to be $O(N^2)$.

1 Problem Overview

The purpose of this experiment is to empirically determine the big-Oh time complexity of the `timeTrial(int N)` method in the `TimingLab` class. This is done by measuring the time each call to the method takes. Each successive call to the method doubles the problem size `N`. The ratio of two successive times is used to calculate the value *k*, which is the degree of the time complexity of the method. This relationship is shown in equation (1) below.

$$T(N) \propto N^k \implies \frac{T(2N)}{T(N)} \propto \frac{(2N)^k}{N^k} = \frac{2^k N^k}{N^k} = 2^k \quad (1)$$

The ratio $T(2N) / T(N)$ converges to a constant R that is equal to 2^k. The value k is then calculated as $k = \log_2 R$. The result is a positive number that describes the big-Oh time complexity of the method `timeTrial(int N)` as $O(N^k)$.

2 Experimental Procedure

To find the time complexity of the `timeTrial` method, the method needed to be executed with different problem sizes and its running times recorded. This was done in the experimental environment described below.

ASUS Q550L Laptop – Installation "EAGLE"

- Intel Core i7-4500U CPU @ 1.80 GHz, 2.40 GHz Turbo
- Nvidia GeForce GT 745M
- 8 GB RAM
- Power Setting: Performance
- OS: Windows 8.1 – all updates installed
- Java version: Version 8 Update 11 (build 1.8.0_11-b12)
- Integrated Development Environment: IntelliJ IDEA 13.1.4

The `TimingLab` class that was being tested is contained in the A3.jar file provided. To run the experiment, the Java code in listing 1 was developed.

Listing 1: Source code to record timing data

```
1     TimingLab tl = new TimingLab(key);
2     System.out.print("Timing multiple calls to timeTrial(N) ");
3     System.out.println("with increasing N values.");
4     System.out.println("N\t  Time\t\tR\t\tk");
5     for (int i = 0; i < 7; i++) {
6        start = System.nanoTime();
7        tl.timeTrial(N);
8        elapsedTime = (System.nanoTime() - start) / BILLION;
9        System.out.print(N + "\t");
10       System.out.printf("%7.3f\t\t", elapsedTime);
11       if (prevTime == 0) System.out.print("-\t\t-\n");
12       else {
13           ratio = elapsedTime / prevTime;
14           lgratio = Math.log10(ratio) / Math.log10(2);
15           System.out.printf("%4.2f\t%4.2f\n", ratio, lgratio);
16       }
17       prevTime = elapsedTime;
18       N *= 2;
19    }
```

The code in line 1 initiates an object of the class we are trying to test. The `key` variable represents a unique key that changes the time complexity of the method. The value of this variable is determined by the group number, which in this case, is 25. The for loop iterates through different trials, starting with a problem size of N = 8 and doubling it every time the loop iterates. The method `System.nanoTime()` returns the current system time, which is recorded in the `start` variable to represent the time that the method starts. Line 7 executes the method `timeTrial` with the current problem size. The next line calculates the total elapsed time for the method by subtracting the start time from the current system time and dividing by 1 billion nanoseconds to give the time in seconds. The current problem size and the elapsed time are then printed to standard output.

The code from lines 11 to 17 is used to calculate the values for the ratio *R* of two consecutive times and the value of *k*. The if statement is used to ensure that the program does not divide by 0 and throw an exception. If this passes, the ratio of the current recorded time and previous time is calculated and stored in the `ratio` variable. The static method `Math.log10` is used to calculate the value for *k*. The values for `ratio` and `lgratio` are then displayed to standard output.

3 Data Collection and Analysis

The result of running the code in Listing 1 in IntelliJ is shown in fig. 1 below.

```
Timing multiple calls to timeTrial(N) with increasing N values.
N      Time      R       k
8      0.146     -       -
16     0.410     2.80    1.49
32     1.026     2.50    1.32
64     4.104     4.00    2.00
128   16.507     4.02    2.01
256   67.664     4.10    2.04
512  281.140     4.15    2.05

Process finished with exit code 0
```

Figure 1. Results of running source code in IntelliJ

Each row represents one call to the `timeTrial` method. The first column, *N*, describes the problem size for a particular trial. The next column shows the elapsed time of the method call. The third column, *R*, displays the ratio of the elapsed time for that run to the elapsed time for the previous run. The final column *k* shows the calculated value for the degree of *N* time complexity of the tested method. Figure 2 shows the graph produced by the data collected.

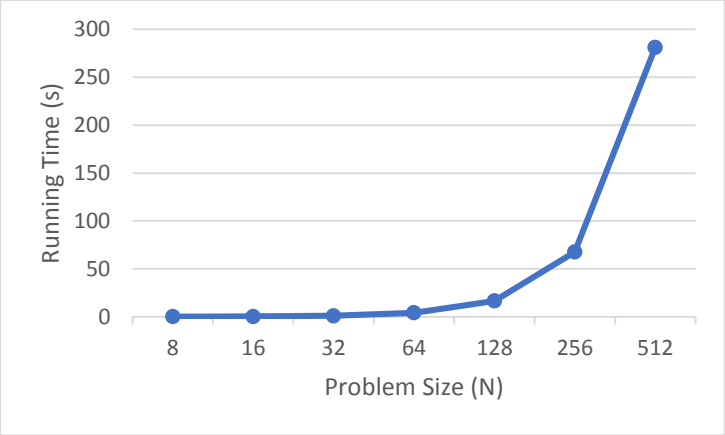

Figure 2. Graph of running time in seconds for increasing problem sizes

4 Interpretation

The graph in fig. 2 shows that the running time for each method call grows exponentially as the problem size doubles. The calculations for *R* and *k* as shown in fig. 1 support this. The value for *R* converges to 4. This means that each consecutive run takes 4 times the run time of the previous run. This value can be plugged into the equation for *k* to find that *k* is equal to 2. Again, the results in fig. 1 show that *k* appears to converge to 2. As shown in section 1, the value of *k* relates to the method's big-Oh time complexity as $O(N^k)$. Therefore, the big-Oh time complexity of the `timeTrial(int N)` method is $O(N^2)$.