COMP 2210 Assignment 3 Part A

Group 46

Xing Wang, Redatu Semeon

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. We were guaranteed that the method has asymptotically binomial time complexity. We used input doubling method to find the upper bound for the method and were convinced that its time complexity is quadratic function..

1. Problem Review

In this experiment, we tried to develop and perform a repeatable experimental procedure that allow us to empirically discover the big-Oh time complexity of the timeTrial(int N) method in the TimingLab class. The parameter N represents the problem size. Thus, by iteratively calling timeTrial with successively larger values of N, we collected timing data that is useful for characterizing the method's time complexity.

According to the instruction for this assignment, we are guaranteed that no matter what key value is used, the associated time complexity will be proportional to N_k for some positive integer k. Thus, we can take advantage of the following property of polynomial time complexity functions T(N).

$$T(N) \propto N_k \Longrightarrow R = \frac{T(2N)}{T(N)} = \frac{(2N)^k}{N^k} = 2^k \times \frac{N^k}{N^k} = 2^k$$

$$k = \log_2 R = \log_2 2^k$$

This property tells us that as N is successively doubled, the ratio of the method's running time on the current problem size to the method's running time on the previous problem size (i.e., T(2N)/T (N)) converges to a numerical constant, call it R, that is equal to 2^k, and thus $k = \log_2 R = \log_2 2^k$.

2. Experimental Procedures

We are required to use System.nanoTime() to generate timing data and the running time values

would be expressed in seconds. All calculations were performed on Intel(R) Core™ i5-4210 2.40GHz CPU, with a 6GB RAM in a 64-bit Windows 10 operating system on a personal HP Pavilion laptop.

The constructor TimingLab(int key) creates a TimingLab object whose timeTrial method is tailored specifically to the given key value. We used our group number in Canvas as the key required by the constructor, and thus invoke the constructor TimingLab(46).

We have to say that the program takes much longer time for the key assigned to our group than others. After several trials, we set our initial N value as 1, and double it in each later run. By understanding the mathematics behind this procedure, we can ask the computer to calculate the ratio R and the logarithm k for us automatically. Note that in order to obtain the ratio of the elapsed times of the current run and of the previous run, we used an array called t to record all the elapsed time we have already got. Also note that in order to compute k, since the math library in Java only provides us the logarithm method for natural logarithms, we used the change of base formula for logarithm:

$$\log_a x = \frac{\log_b x}{\log_b a}$$

Thus, we have

$$\log_2 R = \frac{\log R}{\log 2}.$$

And our code is as follows:

```
public class TimingLabClient {

  /** Drives execution. */
    public static void main(String[] args) {

        // some useful variables that you will need
        double BILLION = 1000000000d; // nanoseconds to seconds
        double start = 0;              // start time of the current run
        double elapsedTime = 0;      // elapsed time of current run
        double prevTime = 0;          // elapsed time of previous run
        double ratio = 1;              // currentTime / prevTime
        double lgratio = 0;           // log base 2 of ratio
        int N = 1;                        // problem size parameter
        int key = 46;                    // selects internal method of RunningTime

        double[] t = new double[100];
        double R;
       double k;
```

```
        // time a single method in this class
        start = System.nanoTime();
        foo();
        elapsedTime = (System.nanoTime() - start) / BILLION;
        System.out.print("This call to method foo() took ");
        System.out.printf("%4.3f", elapsedTime);
        System.out.println(" seconds.");

        // measure elapsed time for a single call to timeTrial
        TimingLab tl = new TimingLab(key);
        start = System.nanoTime();
        tl.timeTrial(N);
        elapsedTime = (System.nanoTime() - start) / BILLION;
        System.out.print("This call to method TimingLab.timeTrial("
            + N + ") took ");
        System.out.printf("%4.3f", elapsedTime);
        System.out.println(" seconds.");

        // measure elapsed time for multiple calls to timeTrial
        // with increasing N values
        System.out.print("Timing multiple calls to timeTrial(N) ");
        System.out.println("with increasing N values.");
        System.out.println("N\tElapsed Time (sec) \tR\tK");
        for (int i = 0; i < 10; i++) {
            start = System.nanoTime();
            tl.timeTrial(N);
           elapsedTime = (System.nanoTime() - start) / BILLION;
            System.out.print(N + "\t");
            System.out.printf("%4.3f\t", elapsedTime);

            t[i] = elapsedTime;
            if (i >= 1) {
                R = t[i] / t[i - 1];
                k = Math.log(R) / Math.log(2);
                System.out.printf("%4.3f\t", R);
                System.out.printf("%4.3f\n", k);
            }
           else {System.out.print("\n");}

            N *= 2;
        }

    }
```

```
    /**
     * Something that will hopefully take time >= 0.001 seconds
     * so that the program output looks better.
     */
    private static void foo() {
        for (int i = 0; i < 100000; i++) {
            String s1 = "War";
            String s2 = "Eagle";
            String s3 = s1 + s2;
            s1 = null;
            s2 = null;
            s3 = null;
        }
    }

}
```

3. Data Collection and Analysis

We have to say that the program takes much longer time for the key assigned to our group than others. After several trials, we set our initial N value as 1, and double it in each later run. As you can see, when the N size grows to 128, the elapsed time for that single run became 2997 seconds, i.e., near one hour. Since the running time becomes longer, and more importantly, we notice that our K, i.e., ***log(Ratio)*** value seems to converge, and thus the running time pattern seems to be clear, we decided to stop there. And the output is as follows:

```
This call to method foo() took 0.026 seconds.
    This call to method TimingLab.timeTrial(1) took 0.001 seconds.
    Timing multiple calls to timeTrial(N) with increasing N values.
```

Table 1 Running time data and calculations

N	***Elapsed Time (sec)***	***Ratio***	***K***
1	0.001	-	-
2	0.011	9.486	**3.246**
4	0.119	10.633	**3.411**
8	0.757	6.359	**2.669**
16	4.550	6.011	**2.588**
32	38.330	8.423	**3.074**
64	307.003	8.009	**3.002**
128	2996.816	9.762	**3.287**

Each row in this table records data for a given run of some method being timed. The first column (N)

records the problem size, the second column (Time) records the time taken for the method to run on this problem size, the third column (R) is the ratio discussed above (i.e., $^{Time_i}/_{Time_{i-1}}$), and the third column (K) is $\log_2 R$.

We also provide a plot of the running time versus size N, which is shown in Figure 1 below.

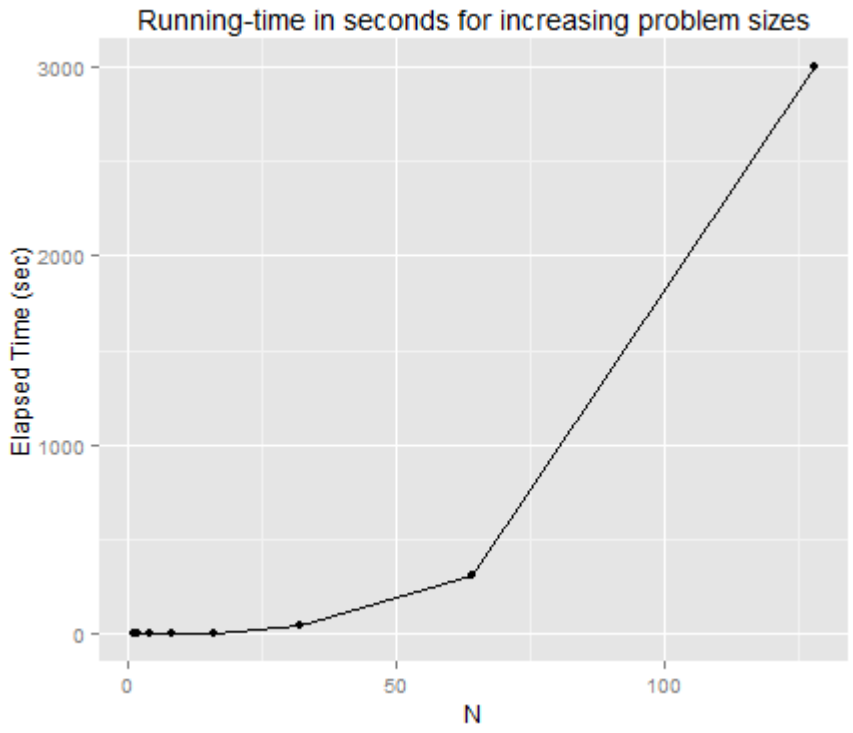

Figure 1

4. Interpretation

To discover the big-Oh by using Table 1 of information one must look at column of K value, this column represents the logarithm or 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 decimals, the result of such calculation is the expected result of the next iteration. The detailed analysis of mathematical relations is shown in Section 1.

As we can see, after the first a few runs, the K value seems to converge to 3, i.e., it oscillated around the value of 3. As we have already known that we are guaranteed to have an integer K, we can hypothesize that the method being timed has $O(N^3)$ time complexity. The plot of the running time versus size N seems to be a demonstration of our hypothesis. Therefore, we conclude that the method being timed has computational complexity of order 3.

Reference

1. Dr. Heandrix Dean, *Algorithm_Analysis.pptx*.
2. Venugopal, S. (2006). *Data Structures Outside-In with Java* (1st edition.). Prentice Hall. ISBN 0-13-198619-8.
3. Lewis&Loftus ,*Java Software solutions foundations of program design* (7th edition),Addison Welsey. ISBN 978-0-13-214918-1