COMP 2210 Empirical Analysis Assignment

Lucas W. Peacock

September 27, 2018

Abstract

We identify through empirical evidence the big-Oh time complexity of the **`timeTrial`** method in the **`TimingLab`** class. Timing data relative to this method was collected for different problem sizes, and observations of growth rate were made using a *power law* assumption. Analysis of the data determined that **`timeTrial`** has $O(N^3)$ time complexity.

1 Problem Overview

In this exercise, resourses.jar contains five different sorting methods in random order. The student is tasked with identifying which sorting method is used in each of the five methods. The methods used include merge sort, randomized quicksort, un-randomized quicksort, insertion sort, and selection sort. The student should begin to approach the problem by first developing a time-trial test which can test each sorting method individually by passing in a variety of three different arrays. The first time array should be in completely random order, following by an array in ascending order, and finished by an array in descending order. The time complexity of each sort method will be calculated by the property of doubling the size of the input array. This property is expressed below.

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

This property allows us to record running times for m calls to **`timeTrial(N)`** with the value of **`N`** doubling on each successive call (i.e., $N_i = 2N_{i-1}$), and then calculate the $m-1$ running time ratios. This data is described in the table below, where column N records the problem size, column $T(N)$ records the time required by the **`timeTrial`** method on the current problem size, and column R is the ratio of the time required for current run to the time required for the previous run (i.e., $T(N_i)/T(N_{i-1})$). Per the power law property above, the values of R will converge to a value $\hat{R} = 2^k$, making the time complexity of the **`timeTrial`** method $O(N^{\log_2 \hat{R}})$.

Table 1: Description of running time data

N	$T(N)$	R
N_0	$T(N_0)$	–
N_1	$T(N_1)$	$T(N_1)/T(N_0)$
N_2	$T(N_2)$	$T(N_2)/T(N_1)$
...	...	...
N_i	$T(N_i)$	$T(N_i)/T(N_{i-1})$
...	...	...
N_m	$T(N_m)$	$T(N_m)/T(N_{m-1})$

2 Experimental Procedure

The following is an outline of our experimental procedure, which follows directly from the preceding discussion.

1. Collect running time data for the **sort1-5** method with random input arrays.
2. Analyze the timing data to identify $\hat{R}$.
3. Repeat with presorted arrays in ascending and descending value.
4. Anaylze how certain methods change based off of the input array.

The following subsections discuss the experimental materials and context and elaborate on each step of the experimental procedure.

2.1 Experimental materials

Two Java classes constitute the experimental materials: **SortingLab**, which contains the **sort** methods under study, and **SortingLabClient**, which generates timing data for the **sort** methods. I have writeen **SortlingLabClient** to implement the experimental procedure.

The student was provided with a jar file (**`resources.jar`**) that contained the **SortingLab** class. The method under study (**sort1-5**) is a static method of this class. The **SortingLab** constructor takes an `int` parameter (the *key*) that is used to map the **sort methods** to a particular polynomial-time algorithm. We were required to use our Banner ID number as the key when instantiating a **SortingLab** object.

2.2 Collecting running time data

Note that, per the assignment requirements, **`System.nanoTime()`** was used to measure elapsed time. The identifier **`BILLION`** is a double with value 1,000,000 and is used to convert nanoseconds to seconds.

2.3 Analyzing running time data

Analysis of the running time data produced by **SortingLabClient** involves computing the ratio of the running time for the current run to the running time of the previous run across all runs, and then estimating the “steady state” value of this ratio ($\hat{R}$). The power law property above tells us that $\log_2 \hat{R}$ is the exponent k in the big-Oh expression $O(N^k)$ for the method **timeTrial**.

3 Data Collection and Analysis

Data was collected by the Java class "SortingLabClient," which tested all five sorting methods in sets with a specific input array, and repeats the process with the next type of array.

- Computer: Dell XPS 15 (15-inch Mid 2017), 2.7GHz Intel Core i5 processor, 32GB 1333 MHz DDR3 memory
- Operating system: Windows 10
- Java: 1.8.0
  - **javac -version**: javac 1.8.0
  - **java -version**: java version “1.8.0” Java(TM) SE Runtime Environment (build 1.8.0-b132) Java HotSpot(TM) 64-Bit Server VM (build 25.0-b70, mixed mode)
  - **System.nanoTime**() used to measure elapsed time

3.1 Timing Data

Timing data was generated by running **SortingLabClient** with problem sizes 1000 to 128000 inclusive. The command issued to run the program and the resulting program output (both copied from a terminal window) are shown below. All times are reported in seconds. (Note that due to the rapid growth rate in running times, the program was aborted after only half of the planned runs (10) were executed, giving data for **N** = 16 to 256 inclusive.)

```
$ java -cp .:A3.jar TimingLabSolution 23 16 10
Run  N    Time
1    16   4.55
2    32   34.48
3    64   276.53
4    128  2204.73
5    256  17531.89
```

4 Interpretation

After running the first data set with a random input array, we can immediately indicate which sorting algorithms are of order N^2 and which sorting algorithms are of order NlogN.

Sort1, Sort2, and Sort 4 all finish significantly faster than Sort3 and Sort 5; therefore these algorithms inlcude mergesort, and both types of quicksort.

Sort3 and Sort5 must be quadratic order sorting algorithms; therefore these methods include selection sort and mergesort.

With $\mathrm{R} = 8$, $k = \log_2 \mathrm{R} = 3$, and thus the method **sort1()** has time complexity $O(N^3)$, therefore it is.

With R = 8, k = log2 R = 3, and thus the method sort1() has time complexity O(N3), therefore it is.

With R = 8, k = log2 R = 3, and thus the method sort1() has time complexity O(N3), therefore it is.

With R = 8, k = log2 R = 3, and thus the method sort1() has time complexity O(N3), therefore it is.

With R = 8, k = log2 R = 3, and thus the method sort1() has time complexity O(N3), therefore it is.