**Assignment 3 Lab Report - Part 1**

**Empirical Discovery of “big-Oh” Running Time**

**Nicholas DiChiara - Spring 2013 COMP-2210**

**Abstract.** In this experiment, I am testing whether it is possible to determine the “big-Oh” time complexity of an unknown method and make assumptions on its efficiency. By running trials on the given class and doubling the amount of data to process with each iteration, I hypothesize that I can examine the recorded data and make an educated guess on the time complexity without seeing the source code.

**Introduction**

When writing algorithms involving large sets of objects and information, efficiency can make all the difference in speed and usability. By using “big-Oh” time complexity, we are able to analyze software and modify it for efficiency. When dealing with software without the source code available we must find other means of determining the time complexity. In this experiment I will run trials on the class “RunningTime” and examine the data yielded and make assumptions as to what is in the source code.

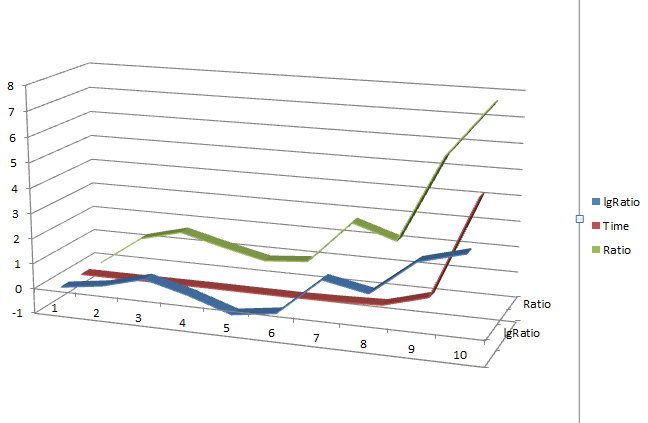
**Methods**

To determine the “big-Oh” running time of the RunningTime class, I began by running the program with an N value of two. After that, I began trials of the driver program with a doubling N value and recorded the time for each. Then, by using the times gathered, I drew the ratio between each trial by dividing the the time of the current trial by the time of the previous trial. With this set of ratios, I took the log(2) of each and recorded them into the table below. This data provided the necessary tools to make assumptions on what the “big-Oh” complexity was.

**Results and Discussion**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Run** | **N** | **Time** | **Ratio** | **lg Ratio** |
| 0 | 2 | 00.011s | N/A | N/A |
| 1 | 4 | 00.013s | 1.18 | 0.24 |
| 2 | 8 | 00.021s | 1.62 | 0.69 |
| 3 | 16 | 00.025s | 1.19 | 0.25 |
| 4 | 32 | 00.020s | 0.80 | -0.32 |
| 5 | 64 | 00.019s | 0.95 | -0.07 |
| 6 | 128 | 00.051s | 2.68 | 1.42 |
| 7 | 256 | 00.108s | 2.12 | 1.08 |
| 8 | 512 | 00.598s | 5.54 | 2.47 |
| 9 | 1024 | 04.588s | 7.67 | 2.94 |
| 10 | 2048 | 36.024s | 7.85 | 2.97 |
| 11 | 4096 | 292.148 | 8.12 | 3.02 |

As you can see from the table above and the chart below, the “lgRatio” was steadily approaching “3”. With the methods I mentioned above and with this table of data, I am able to safely determine that the method “timeTrial” has a “big-Oh” time complexity of O(N^3).



**Conclusion**

In conclusion, I tested whether it was possible to determine the “big-Oh” time complexity of an unknown method and make assumptions on its efficiency. By running trials on the given class and doubling the amount of data to process with each iteration, I was able to examine the recorded data and make an educated guess on the time complexity without seeing the source code.

**Assignment 3 Lab Report - Part 3**

**Experimental Identification of Sorting Algorithms**

**Nicholas DiChiara - Spring 2013 COMP-2210**

**Abstract.** In this experiment, I attempt to identify the sorting methods used in a class without access to the source code. By writing a driver class to run all of the sorting methods, I can record the run-time and ratio of the data and attempt to match that information to characteristics of known sorting algorithms and determine a match.

**Introduction**

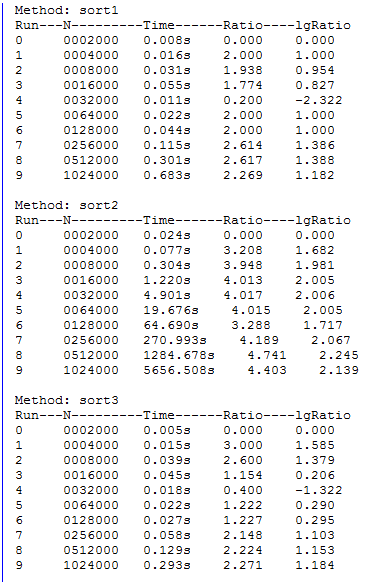
When writing algorithms involving large sets of objects and information, efficiency can make all the difference in data-processing and determine usability and run-time. To identify the sorting methods used in the SortTrials class, I began with the fact that I needed to observe these methods in action and view their results. The first step taken was to write a driver program that would run each sort method ten times, each time doubling the number of values to be sorted. Then the program must record the times of each, and determine the ratio between each run.

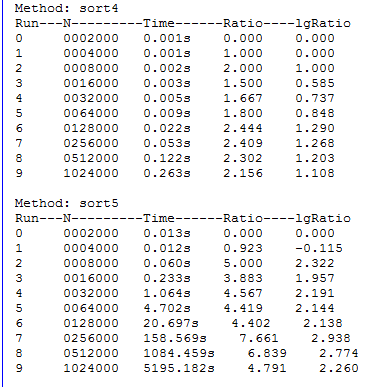
**Methods**

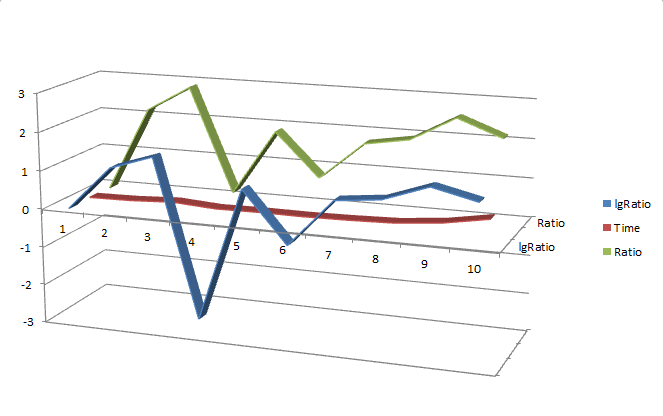
To test this, I began with a reflection method to identify the names of the methods within the SortTrials class and populating them into an array. After retrieving the method names, I added a for loop to iterate through the declared methods and continue only if the current method name being examined began with the word “sort.” This would allow the program to focus only on what is being tested and begin trials on each.

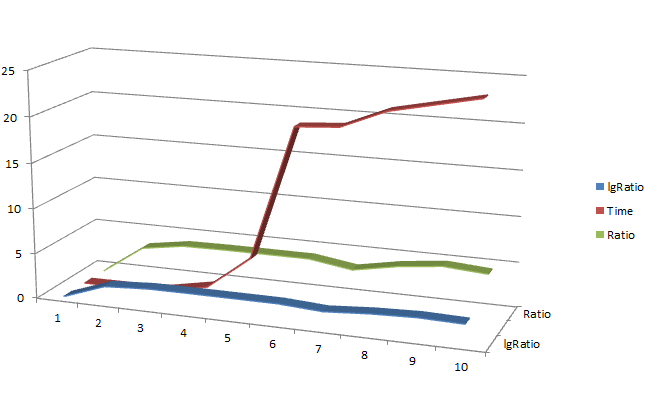
After isolating the methods, I needed to provide an array to test on each. This array must have the ability to double in size with each trial, and must populate itself with random integers that are not already sorted. To do this, I created an empty array of size “N” and then a for-loop that utilizes a Math.random() method call to populate the array with random numbers. Now that the test data has been generated, the array is tested to ensure that it is not already sorted before proceeding. The program then invokes the clock class to begin a timer and immediately proceeds to the sort method being tested. The sort method sorts the array, and then the driver program stops the clock and stores the time. From there, the program takes that time, divides it by the previous time and concludes the ratio between the two trials. By taking the log(2) of that ratio, I can begin to see a simplified extrapolation of that data which yields to what degree of complexity the method is performing. After these calculations, the program prints them and begins with the next trial. Once all necessary trials have been performed, the program resets the “prevTime” and the “N” value back to its original starting point at 2000 and repeats the process on the next four sort methods.

**Results and Discussion**

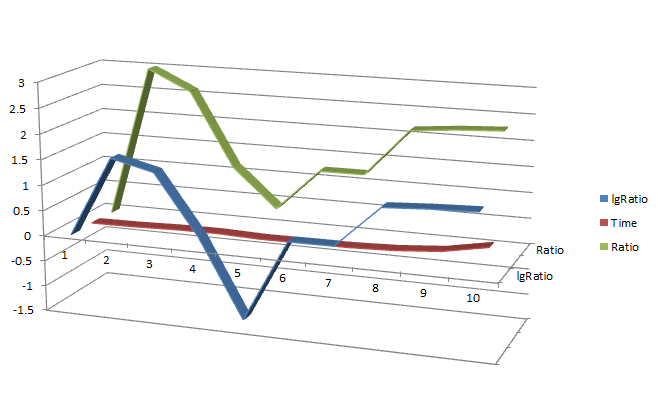




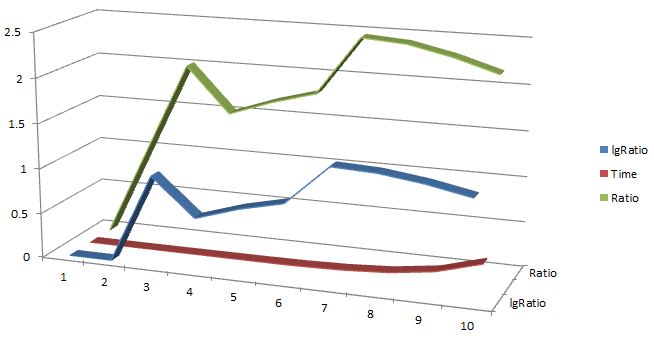
sort1:

sort2: (time was modified in this chart to better depict the ratio)

sort3:



sort4:



sort5:

