Evan Sheehan

Empirical Analysis of Time Complexity

Purpose

The purpose of this report is to analyze and identify the time complexity of a provided method, in this case “methodToTime()” from the “ProvidedClass” class. This must be accomplished through empirical analysis; raw data must be recorded and summarized to more clearly illustrate any patterns that might be found.

Procedure

The instructor provided a client for measuring the time complexity of methodToTime() already, but it was recommended that we modify or create our own method to analyze the method’s time complexity. I wrote my own method, though I used a very similar format to the method the instructor provided. I wanted to have at least one other variable under my control for my own method. In this case, I decided to run methodToTime() an “n” number of times, average those times together, and use that average as the output for methodToTime()’s overall time. Where the provided client uses the variable “numRuns” to represent the number of times methodToTime() is ran and the problem size is doubled, I used a similarly named variable to represent the number of times methodToTime() is ran but with the same problem size. I then used another variable, numIncrements, to represent the number of time the problem size is incremented.

With these two variables, we can use a nested loop structure to control both the number of times a specific problem size is ran and the number of times the problem size is incremented, doubled in this case.

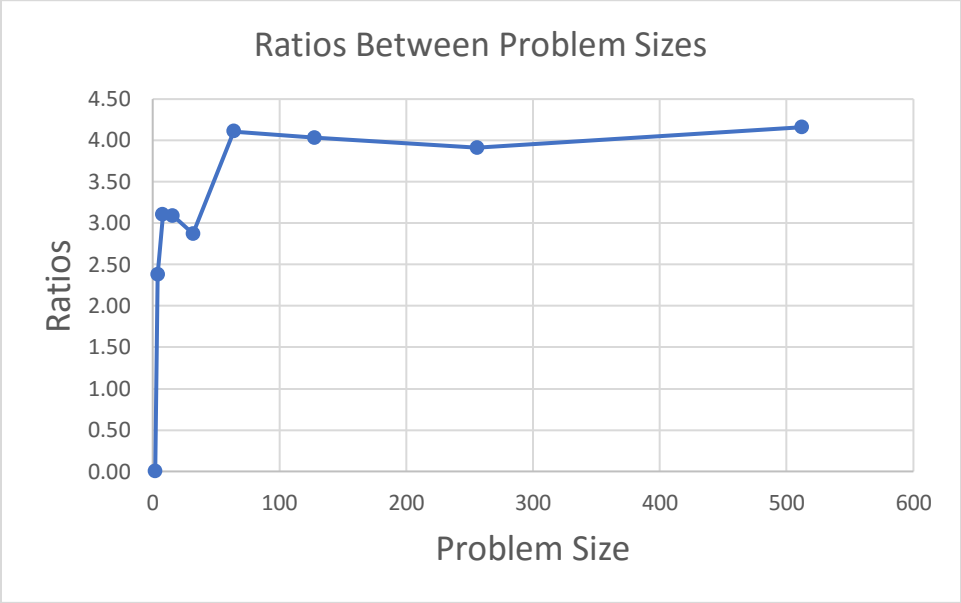
```
for (int i = 0; i < numIncrements; i++) {
   double averageTime = 0;

   for (int j = 0; j < numRuns; j++) {
      long startTime = System.nanoTime();
      provClass.methodToTime(n);
      long endTime = System.nanoTime();
      averageTime += (endTime - startTime) / SECONDS;
   }
```

Once these variables are decided, this method will run methodToTime() for however many increments the user defines, and will run each increment (problem size) a variable number of times, given the value the user defines for numRuns. Each execution of methodToTime() is measured by running System.nanoTime() directly before and after the method execution. The difference between these times results in the total runtime for the method. We convert this number to seconds using the “SECONDS” constant and add the runtime to a double value named “averageTime”. The value of averageTime is later divided by the variable numRuns, resulting in the average runtime for a given problem size. The last piece of data I collected in the actual runtime was the ratio between problem sizes. This could have been calculated by hand post-runtime, but I decided to go ahead and include it in my method. It is calculated by dividing the current measured runtime by the previous measured runtime. The two following charts illustrate my results.

Evan Sheehan

Results

Problem Size	No. of Runs	Average Runtime	Ratio
2	10	0.00874	0.00
4	10	0.02086	2.38
8	10	0.06475	3.10
16	10	0.20050	3.09
32	10	0.57654	2.87
64	10	2.37092	4.11
128	10	9.57005	4.03
256	10	37.49822	3.91
512	10	156.07538	4.16



The results illustrate a clear pattern, which is that methodToTime() has a quadratic, or $O(N^2)$, time complexity. I started with an n = 2 problem size which is incremented 8 times, with each problem size being ran 10 times, then averaged. As expected, lower problem sizes are more difficult to measure accurately. It can be seen that the method isn't approaching any specific ratio until the problem size is about 64, and afterwards methodToTime() quickly approaches a ratio of 4. Identifying that methodToTime() runs with quadratic time complexity involves taking the increase in problem size and comparing it to the ratios. In this case, we're doubling the problem size with each increment, but the average elapsed time nearly quadruples each time. This means that the relationship between the problem size and the average elapsed time is N^2, where N is the problem size; a quadratic time complexity.

Evan Sheehan

Conclusion

Through empirical analysis, solving a method's time complexity appears relatively straightforward in this context. The graph that was generated in Excel helps to show how time complexities for methods approach a certain ratio. Looking at the raw data, the ratios outputted can be discouraging. How can I say the ratio is approaching 4 when half of my problem sizes aren't close to that ratio? I definitely underestimated how the problem size affects the ratio, and feel more satisfied with my results after generating the line graph. One last takeaway is that running a certain problem size more than once and averaging the results doesn't seem to help that much with more accurately generating data, as I thought maybe it would. With larger problem sizes, the ratios become so consistent regardless if their results were averaged or not that it doesn't make a significant difference to average more than one run of a certain size.