CPSC-335

# Project 1: Empirical Analysis

Authors:

Cristopher Hernandez, Reza Mansouri, Christian Medina

Contact Information:

cristopherh@csu.fullerton.edu, rmansouri@csu.fullerton.edu, christian.medina@csu.fullerton.edu

The goal of this report will be to empirically test the Big O notation growth rates of three algorithms. Each algorithm was implemented in C++ using provided skeleton code and functions. In addition, data was collected by compiling and running code on the Cal State Fullerton Titan Server. The hypothesis in question is thusly:

*For large values of n, the mathematically-derived efficiency class of an algorithm accurately predicts the observed running time of an implementation of that algorithm.*

There were three algorithms for testing:

- *character mode* algorithm

| *character mode problem* |
|---|
| **input:** a vector V of n strings that collectively contain at least one character<br>**output:** the character that appears the most times among all characters in V; when two characters are tied, the output is the tie with the least ASCII code |

- *longest mirrored string* algorithm

| *longest mirrored string problem* |
|---|
| **input:** a vector V of n strings<br>**output:** a string a from V whose mirror is also in V, of maximum length; or the empty string |

- *longest subset trio* algorithm

| *longest subset trio problem* |
|---|
| **input:** a vector V of n strings<br>**output:** a vector containing exactly three strings a, b, c comprising the substring trio of maximum total length; or a vector containing three empty strings when V contains no substring trio |

## Algorithm 1

The first problem we will discuss will be the character mode problem. This algorithm produced the most frequently occurring character in a set of strings. The pseudo-code that was provided is presented again here:

```
def character_mode(V):
    counts = Array(256, 0)
    for string in V:
        for character in string:
            counts[character]++
    c = <find the character c with the greatest value of
    counts[c] >
    return c
```

looking at the pseudocode for this algorithm, we can ascertain an equation for proving an efficiency class. Using chronological step counting, we find:

$$f(n) = n * m + 3$$

Where *n* is the number of strings given in *V* and *m* is the length of a string in *V* that is being examined.

Using limits we can prove that the derived equation is of efficiency class O(n).

Let g(n) = n

$$\lim_{n \to \infty} \frac{nm}{n}$$

$$\lim_{n \to \infty} m$$

Since *m* represents the length of a string and the longest word in the English dictionary is 45 characters (as stated in the given information for the algorithm), we can reason that m -> 45 as n -> ∞.

Therefore: $\lim_{n \to \infty} 45$

The result is a constant, therefore f(n) = g(n) and f(n) is in the efficiency class O(n).

Here is a scatterplot of the data obtained:

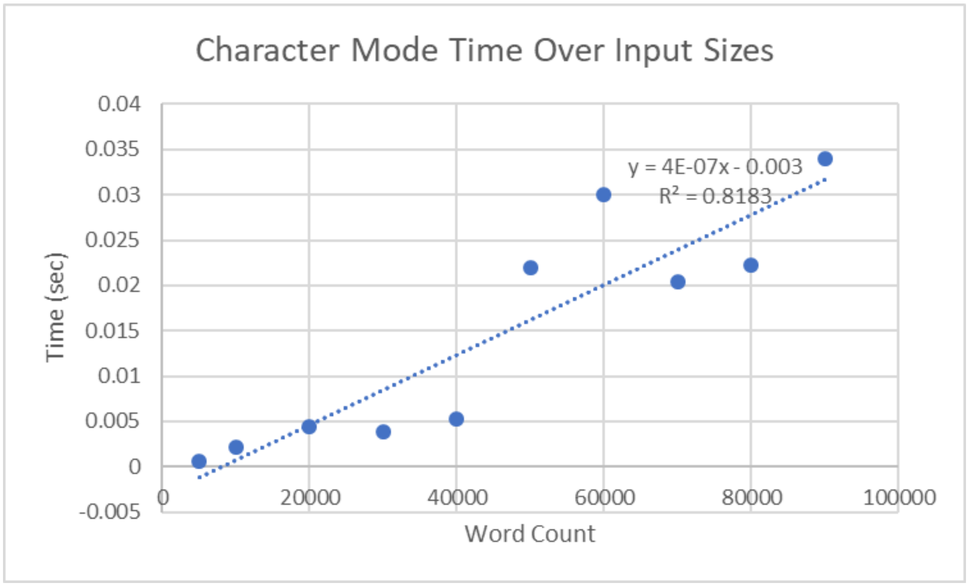

*Figure 1*

Examining this graph, the equation and growth rate appears to conform to the expectation of an O(n) efficiency class. This is because the resulting best fit line equation is linear in growth (following the y = mx + b form; see the equation in figure 1), just as O(n) is also linear with respect to input size. This brings us to the next algorithm.

**Algorithm 2**

The second problem examined was the longest mirrored string algorithm. The pseudo-code that was provided for this algorithm is reproduced here:

```
def longest_mirrored_string(V):
     best = “”
     for a in V:
          for b in V:
               if a is the mirror of b and len(a) > len(best):
                    best = a
     return best
```

Examining this algorithm using chronological step counting, we find:

$$f(n) = n^2 + 2$$

Using limits we can prove that this equation belongs to the efficiency class O($n^2$).

Let g(n) = $n^2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

$$\lim_{n\to\infty} \frac{n^2 + 2}{n^2}$$

$$\lim_{n\to\infty} \frac{n^2}{n^2} + \frac{2}{n^2}$$

$$\lim_{n\to\infty} 1$$

The result is a constant. Therefore, f(n) = g(n) and the derived equation is of the efficiency class O($n^2$). In addition, the data collected when executing the algorithm appears to back up this conclusion.

Here is the scatterplot of the data obtained:

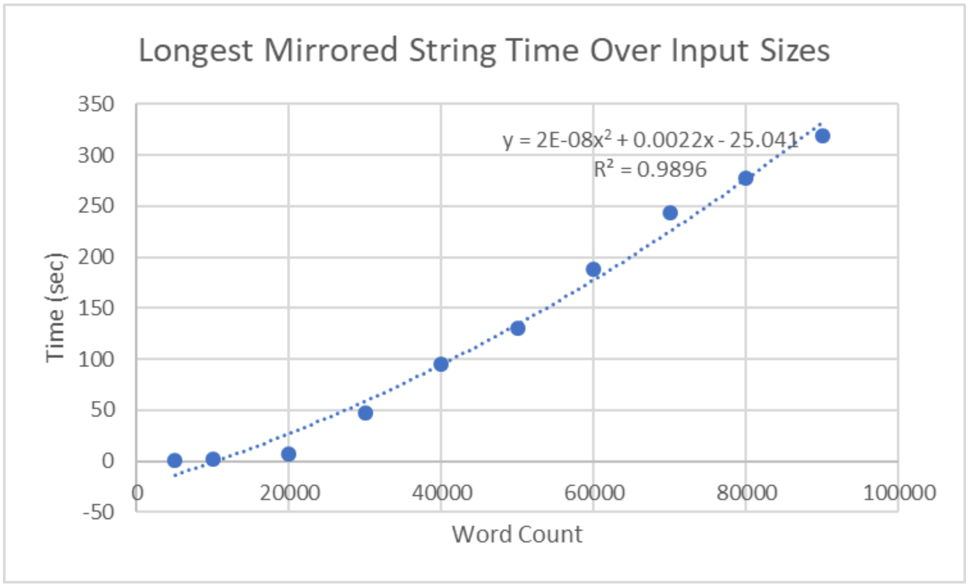

*Figure 2*

Plotting a line through the scatterplot, we find the best fit is a 2nd order polynomial function (see the equation in figure 2). This appears to be within the expectation of an $O(n^2)$ order of growth. This is because the polynomial function displays growth over time that is dominated by its highest order element (i.e. the $x^2$). As such, as the data grows, it should conform to an $O(n^2)$ growth rate over time. With this algorithm analyzed, the final algorithm is next.

**Algorithm 3**

The third problem examined was the longest substring trio algorithm. The pseudo-code that was provided for that algorithm is reproduced here:

```
def longest_subset_trio(V):
    best_length = 0
    trio = Vector(3, "")
    for a in V:
        for b in V:
            for c in V:
                abc_length = |a|+|b|+|c|
                if <a, b, c are a subset trio> and
                  abc_length > best_length:
                    best_length = abc_length
                    trio[0] = a
                    trio[1] = b
                    trio[2] = c
    return trio
```

Using chronological step counting, we produce this formula:

$$f(n) = n^3 + 3$$

Using limits we can prove that the derived equation is in O($n^3$) efficiency class.

Let g(n) = $n^3$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

$$\lim_{n\to\infty} \frac{n^3 + 3}{n^3}$$

$$\lim_{n\to\infty} \frac{n^3}{n^3} + \frac{3}{n^3}$$

$$\lim_{n\to\infty} 1$$

The result is a constant. Therefore f(n) = g(n) and the derived function is in the O($n^3$) efficiency class.

Here is the scatterplot for the physical execution data that was obtained:

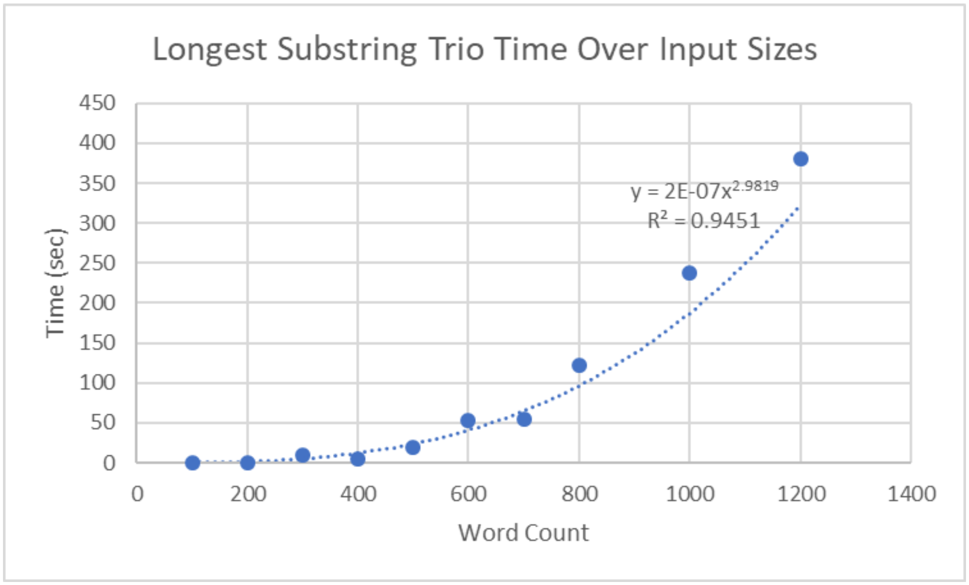

*Figure 3*

Looking at this graph, the line which fits the data best is exponential in nature. Specifically, the resulting equation of the line has an exponent of 2.9819 (see the equation in figure 3), which is very close to the measure of 3. This growth rate is very similar to the derived efficiency class, which is also exponential. As a result, this appears to conform to our expectation for the algorithm.

## Conclusion

After analyzing all three algorithms, some notable features about their performance arises. To elaborate, the performance differences between these algorithms are very noticeable. This is especially true as the input size increases. Some algorithms were faster than others, with the substring trio algorithm being easily the slowest. By contrast, the character mode algorithm appeared to be the fastest. This is not altogether surprising, as it is in the fastest efficiency class. In fact, this algorithm is faster by a large margin. For comparison, the slowest run of the mirrored string algorithm required about 320 seconds. Whereas, for the character mode algorithm, the longest time needed was roughly 0.035 seconds for the same input size of n = 90,000. That is an incredible change in running time. These observations exemplify how much difference there really is between the different efficiency classes.

The experiments with these algorithms appear to be consistent with the hypothesis originally stated at the beginning of this project. Essentially, the overall trends appeared to conform to their efficiency classes. For example, the first algorithm shows a linear growth consistent with what you might expect of an O(n) efficiency class algorithm. The second algorithm appears to fit a 2nd degree polynomial which is also consistent with an $O(n^2)$ efficiency class algorithm. Finally, the third algorithm is perhaps the closest fit, being that the equation for the best fit curve is dominated by $x^{2.9819}$. This kind of exponential equation is to be expected with an $O(n^3)$ efficiency class algorithm. As a result, the conclusion can be made with confidence that the data collected is consistent with the hypothesis. Put plainly, for large values of n, the mathematically-derived efficiency class of an algorithm appears to accurately predict the observed running time of an implementation of that algorithm.