Project 1 Report

CPSC 335 SUMMER 2021, SESSION A

Sean Mitchell & Jason Mora-Mendoza
CALIFORNIA STATE UNIVERSITY, FULLERTON | SMITCHELL36@CSU.FULLERTON.EDU

Table of Contents

1. Introduction

The purpose of this project is to analyze the time complexity functions of two algorithms found in the book (exercises 3-14a and 3-15b) and to prove their efficiency classes. Additionally, we implement these two algorithms in C++, and time their execution times for various sizes of n. Using these n values and their associated time, we plot them onto a scatterplot to confirm our efficiency classes that we concluded upon in the mathematical analysis section.

2. Mathematical Analysis

The Mean Algorithm

```
def mean(L):
    total = 0
    for x in L:
        total += x
    return total / len(L)
```

Figure 1 - Mean pseudocode from Dr. Wortman

Finding a complexity function for running time, T(n)

We have 2 single step statements outside of the loop:

```
total = 0
return total/len(L)
```

So far, we have T(n) = 2. But now we must include the loop which executes n amount of times, where n = the length of list L. The loop itself includes 1 single step statement: `total+=x`. However, we must also add 1 more statement to account for loop overhead. So for the loop we have T(n) = 2n, but we must also include the 2 single step statements outside of the loop, giving us: <u>T(n) = 2n + 2.</u>

Proof by Properties of Big O:

$2n + 2 \in O(2n + 2)$ (trivial)

$= O(2n)$ (Dropping additive constants)

$= O(n)$ (Dropping multiplicative constants)

<u>Thus, $T(n) \in O(n)$. (Linear time complexity)</u>

Square Matrix Construction Algorithm

```python
def construct_square_matrix(n, x):
    rows = []
    for r in range(n):
        rows.append([])
        for c in range(n):
            rows[r].append(x)
    return rows
```

Figure 2 - Square matrix construction pseudocode from Dr. Wortman

Finding a complexity function for running time, T(n)

We have 2 single step statements outside of both loops:

```
rows = []

return rows
```

Which gives us T(n) = 2. The outer loop contains 1 single step statement, which gives us T(n) = 2n, including an additional statement for loop overhead. The inner loop also contains only 1 single statement, so we would also get T(n) = 2n. If we combine both loops, we get $T(n) = 4n^2$, however we must include the 2 single statements outside of the loop as additive constants.

Finally resulting in $\underline{T(n) = 4n^2 + 2}$.

Proof by Properties of Big O:

$4n^2 + 2 \in O(4n^2 + 2)$	(trivial)
$= O(4n^2)$	(Dropping additive constants)
$= O(n^2)$	(Dropping multiplicative constants)
Thus, $T(n) \in O(n^2)$	(Quadratic time complexity)

3. Implementation in C++

This project implements two algorithms in C++17 and a problem instance generator function for each algorithm. Both algorithms are derived from the book's pseudocode listed in exercises 3-14 (a) and 3-14 (b). The following two subsections will go over the implementation of each function, and its problem definition.

The Mean Algorithm

The implementation for this code is simple. The function takes in a vector of type int, L as an argument. Then we initialize the total variable of type double, with an initial value of 0. Then we loop through the vector L, adding each element of L to the running sum of the total variable. Finally, we return the total sum divided by the vector size to get the mean as a double. The C++ code and problem definitions for both the algorithm and instance generator are listed below.

```
double mean(std::vector<int> L) {
    double total = 0;
    for(int i=0; i<L.size(); i++) {
        total+=L[i];
    }
    return total/L.size();
}
```

Mean problem
Input: a non-empty list L of n numbers **Output**: the mean (average) of L

Mean problem instance generation
Input: a positive integer n **Output**: a non-empty list L of n random integers

The Square Matrix Construction Algorithm

The implementation for this algorithm is slightly more complicated than the mean algorithm, however still simple. The function takes two arguments: an integer n which represents the size of the square matrix (an n x n matrix) and a double x which is the value that each element in the square matrix will store. First, we initialize a vector of vectors type int, called rows. Then we loop n amount of times, and for each n we push an empty vector into rows. We then loop n amount of times within the current iteration of the outer loop, pushing x into the empty vector we pushed earlier in the outer loop. Finally, we return the n x n matrix. The C++ implementation and problem definitions for the function and problem instance generation are listed below.

```cpp
std::vector<std::vector<int>>  construct_square_matrix(int n, int x) {
    std::vector<std::vector<int>> rows;
    for(int i=0; i<n; i++) {
        rows.push_back({});
        for(int j=0; j<n; j++) {
            rows[i].push_back(x);
        }
    }
    return rows;
}
```

Square Matrix Construction
Input: a positive integer n and number x **Output**: an n x n matrix with each element equal to x

Square matrix construction problem instance generation
Input: a positive integer n **Output**: a number x

4. Empirical Analysis

In this section, we use empirical, or experimental, analysis to confirm the efficiency class of both algorithms.

Implementation of Experiment

To accurately time each function, we used std::chrono to capture the start and end time in milliseconds for each trial. We tested n values from 1 to 5000, with 3 random problem instances for each value of n. These results were outputted to two csv files: “mean.csv” and “matrix.csv”. Then we used R and ggplot2 to plot this data on a scatterplot for each function.

Benchmark Results

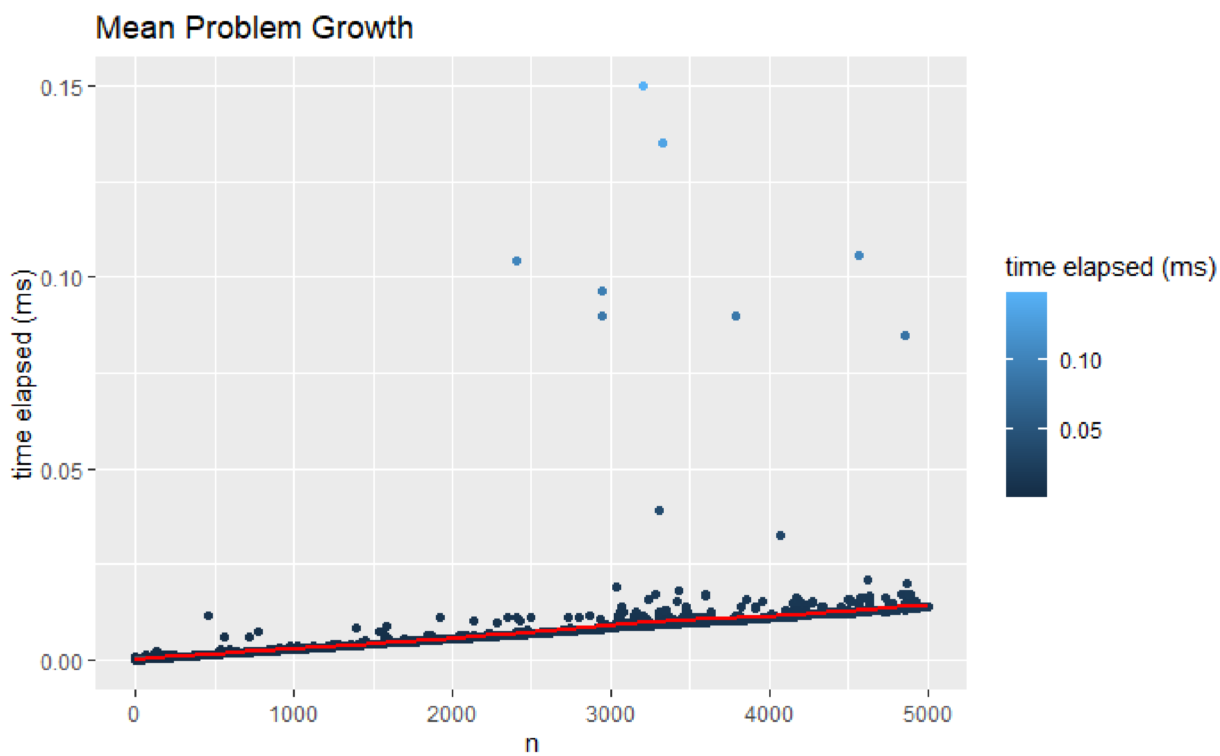

Figure 3 - Mean Scatterplot

The scatterplot above clearly shows that there is a linear relationship between n and time elapsed (ms) as n grows larger. Each dot represents a single test trial, for a total of 15,000 total test trials (3 random trials for each value of n). The red line shows the trendline for each trial, which best fits with a linear line, O(n). Based off this information, we can conclude that the mean function runs in O(n) time, confirming our findings in the previous section.

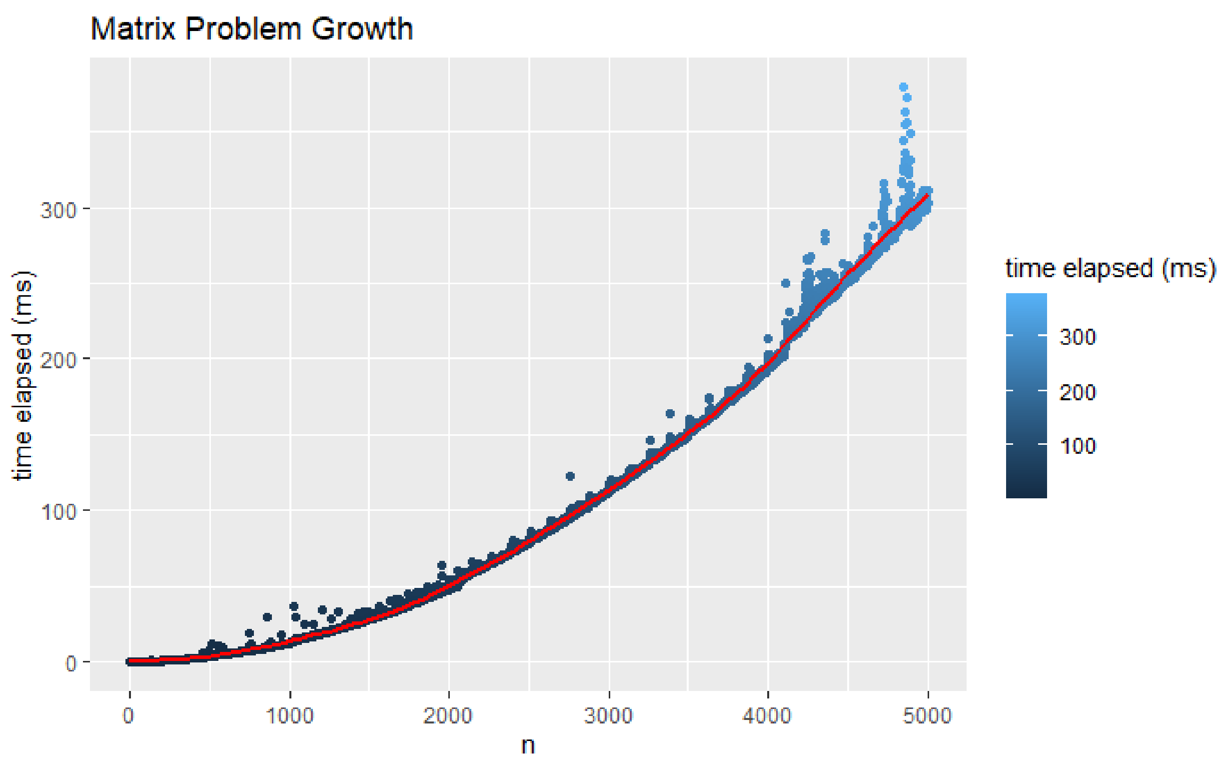

Figure 4 - Square matrix construction scatterplot

The scatterplot above shows a quadratic relationship between n and time elapsed (ms) for the square matrix construction function. Similar to the previous experiment, we tested using values of n ranging from 1 to 5000, with 3 trials per iteration of n. Based off of the trendline, we can conclude that the function runs in $O(n^2)$ time, as we found previously through our proof by properties of Big O.

5. Conclusion

This project covered the mathematical analysis of two functions, `mean` and `construct_square_matrix` using properties of Big O, implemented the pseudocode in C++, and rigorously tested and analyzed the time trials of each function in R. We found that the mean runs in O(n) and `construct_square_matrix` runs in $O(n^2)$. This project served as a good learning exercise on mathematical analysis through properties of Big O and empirical analysis through meticulous testing.

References

"Date and Time Utilities." *Cppreference.com*, en.cppreference.com/w/cpp/chrono#Example.

"Std::Rand." *Cppreference.com*, en.cppreference.com/w/cpp/numeric/random/rand#Example.

Wortman, Kevin A. *Algorithm Design In Three Acts*.