COMP20230 Data Structures Algorithms

Time Complexity Report

Brian Manning

Student ID: 17324576

A report submitted as part of the degree of

MSc. in Computer Science (Conv.)

Lecturer: Professor Madhusanka Liyanage

UCD School of Computer Science

University College Dublin

March 26, 2021

Table of Contents

Chapter 1: **Introduction and Overview**

The following report will outline my work from constructing algorithms in pseudocode to the implementation of these algorithms in Python code within the attached Jupyter Notebook file.

Using time complexity analysis techniques, I show how we can estimate the running time of algorithms and following this using various techniques in Python - can implement and record the actual running time for comparison.

I have used a number of Python packages to complete this project - these include the following:

- line_profiler - Record both the running time of each line of code in a function along with the number of times that code is run.
- Pandas - used for analysing large datasets, I use it to save the running times over a large amount of iterations and to ultimately calculate the cleaned mean running time for various inputs.
- Plotly - A plotting package - used for creating graphs to visually represent the data and results
- Pyheat - create visual heatmap of the running time of Python code
- scipy - For fitting curves to the data

You can install the packages using:

```
pip install -r requirements.txt
```

The project folder is laid out as follows:

- data - CSV files of the data collected and generate throughout the project
- graphs - Static & Interactive Plots
- profiling - files created when profiling the code
- python_files - .py versions of functions created in the notebook, used for profiling

Chapter 2: Factorial Algorithms

The factorial of an integer n is the product of all positive integers less than or equal to n.
We can calculate the factorial of an integer using different types of algorithms.

2.0.1 Iterative

An iterative function is one that loops to repeat some part of the code. The first example below shows the mathematical formula for calculating the factorial iteratively.

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \cdots \cdot 3 \cdot 2 \cdot 1\,.$$

Below you can see my written algorithm for calculating the factorial iteratively:

Algorithm 1: Factorial - Iterative

Data: An integer n, greater than or equal to 0
Result: The product of all positive integers less than or equal to n, represented as factorial below

```
Function iterative_factorial(n):
    factorial = 1;
    for i ←n to 0 do
        if i = 0 then
            factorial = factorial * 1;
        else
            factorial = factorial * i;
        end
    end
    return factorial;
```

The above algorithm has the time complexity of O(n). This is because the function must multiply the starting number (1) by each number up to the number n, which is n operations. This gives us a linear time complexity.

2.0.2 Recursive

Non-Tail Recursive

A recursive function is one that calls itself again to repeat a section of the code. The following example shows how the same result (the factorial) can be achieve using a recursive function.

$$n! = n \cdot (n-1)!$$

Below you can see my written algorithm for calculating the factorial recursively:

Algorithm 2: Factorial - Non-Tail Recursive

Data: An integer n, greater than or equal to 0
Result: The product of all positive integers less than or equal to n
Function `recursive_factorial(`n`)`:
    if $n = 0$ **then**
        return 1
    else
        return n * `recursive_factorial(`$n-1$`)`
    end

Tail Recursive

The following example algorithm shows how the factorial can also be calculated with a tail recursive algorithm. A tail recursive algorithm is one where the recursive call is the last logic instruction in the recursive algorithm.

Algorithm 3: Factorial - Tail Recursive

Data: An integer n, greater than or equal to 0
Result: The product of all positive integers less than or equal to n
Function `recursive_factorial_tail(`*n, accumulator=1*`)`:
    if $n = 0$ **then**
        return accumulator
    else
        return `recursive_factorial_tail(`*n-1, accumulator * n*`)`
    end

2.0.3 Time Complexity

By studying our non-tail recursive algorithm above, we can prove that the time complexity is O(n) as follows: The factorial of 0 is only 1 comparison. This means that T(0) = 1. The factorial of n involves 1 comparison (if n = 0), 1 multiplication and one subtraction - and also the time for the factorial of n - 1. This leaves us with T(n) = T(n-1) + 3. Below we can solve this equation for k[Tou].

$$\begin{aligned} T(n) &= T(n-1) + 3 \\ &= T(n-2) + 6 \\ &= T(n-3) + 9 \\ &= T(n-4) + 12 \\ &= ... \\ &= T(n-k) + 3k \end{aligned} \tag{2.1}$$

as we know

$$T(0) = 1 \tag{2.2}$$

we need to find the value of k for which

$$\begin{aligned} n - k = 0, k = n \\ T(n) = T(0) + 3n, k = n \\ = 1 + 3n \end{aligned} \tag{2.3}$$

that gives us a time complexity of O(n)

Chapter 3: **Search Functions**

Below I will discuss the implementation of various search algorithms.

3.0.1 Linear Search

Linear Search works by searching through each element one after the other and checking whether that element matches the key. If it matches the key the index of that element in the array is returned. If the element is not found -1 is returned.

Algorithm 4: Linear Search

Data: A sorted array of integers, Arr & a search key Key which is to be found within the array
Result: The index of the search key in the array, -1 if Key not found
Function `linear_search`(Arr, Key)**:**

```
for i ←0 to len(Arr) do
    if Arr[i] = Key then
        return i;
    else
        continue;
    end
end
return -1;
```

Time Complexity

Linear Search in the worst case (i.e. element not actually in the array) must search through every element in the array before finishing. This means it must do n operations, giving us O(n). In the best case linear search must only search through one elements (i.e. if the search key is the first element in the array), this is $\Theta(1)$. The average case for linear search is also O(n).

In comparison to Binary Search below, this algorithm is very inefficient when comparing against finding a key in a sorted array. Linear Search has the advantage that it can be done on unsorted arrays and it can also be used on linked lists.

3.0.2 Binary Search

Binary Search, also known as logarithmic search[Bor06], is a search algorithm which finds the position of a key within a sorted array. Binary search compares the middle value of the array with the key, if the value at the middle is equal to the key that value is returned, if not it checks whether the value is greater than or less than the key. If the value is greater than the key, it searches through the half of the array which is less than that value or else it searches through the higher half of the array. It follows this pattern until either the whole list has been searched through or the key has been found.

Iterative

The iterative binary search algorithm works by halving the search array until the middle element is equal to the Key. It uses a while loop to halve the search space in the array until the Key is found.

Algorithm 5: Binary Search - Iterative

Data: A sorted array of integers, Arr & a search key Key which is to be found within the array
Result: The index of the search key in the array, -1 if Key not found
Function `binary_search_iterative(`Arr, Key`):`

```
low = 0;
high = len(Arr) - 1;
while low ≤ high do
    mid = low + (high + low) // 2;
    if Arr[mid] = Key then
        return mid;
    else if Arr[mid] > Key then
        high = mid - 1;
    else if Arr[mid] < Key then
        low = mid + 1;
end
return -1;
```

Recursive

The Recursive Binary Search algorithms works in much the same way as the Iterative method, but rather than iterating with a while loop, the function is called recursively until the index is found.

Time Complexity

Binary Search algorithms have a time complexity of $O(\log n)$. It is clear to see from the Big O values that Binary Search will be much quicker on sorted arrays than Linear Search.

Chapter 4: Running Time Analysis

The actual running times of the above algorithms were tested using the timeit package. Pandas & Plotly were used to analyse the dataset and plot various graphs. Scikit learn was then used for curve fitting on the results.

4.0.1 Factorial Functions

These functions were ran 1000 times with the input parameters ranging from 0 to 2499, 2500 inputs in total (totalling 2500*1000 = 2,500,000 total runs). The times were recorded for each of the input parameters on each run. I then cleaned the dataset from outliers using a standard deviation approach where any values which fell outside of 3 standard deviations of the mean were removed. The mean value of the cleaned running times for each value of n were then found.

Iterative Factorial

Below we can see the average running times plotted as a function of n:

While we would expect the above to be linear, due to added complexities in Python, such as the way multiplication is implemented using the Karatsuba (time complexity of $n^{1.585}$) and also the time complexity of large numbers - we see that as the value of n gets larger, the increase in running time increases greater than proportionally. The fitted function is has the follow parameters:

$$-4.3 \times 10^{-6} + 8.3 \times 10^{-8}x + 2.2 \times 10^{-10}x^2 + 1 \times 10^{-14}x^3$$

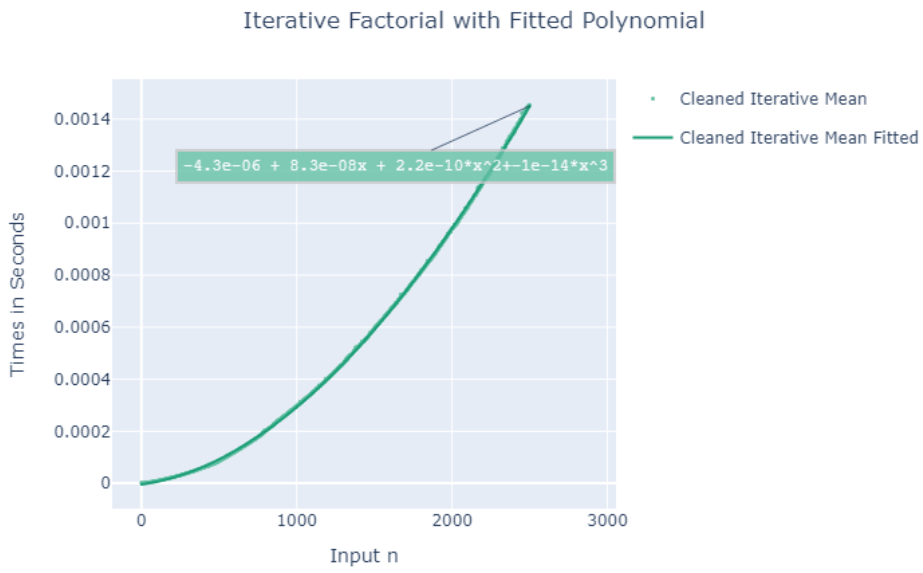

Figure 4.1: Iterative Factorial Running Times

Below we can see that both tail and non-tail factorial running times follow a similar trend to the above iterative algorithm. For the non-tail recursive algorithm, the running times are slightly greater than the iterative algorithm, while for the tail recursive algorithm we can see that the running times are about 33% greater than either of the above.

Non-Tail Recursive Factorial

$$-8.6 \times 10^{-6} + 1.1 \times 10^{-7}x + 2.2 \times 10^{-10}x^2 + 4 \times 10^{-15}x^3$$

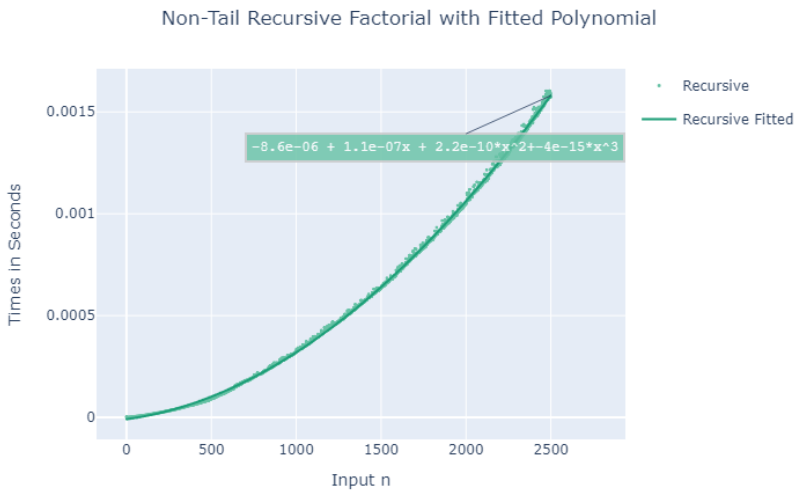

Figure 4.2: Non-Tail Recursive Factorial Running Times

Tail Recursive Factorial

$$-5 \times 10^{-6} + 1.1 \times 10^{-7}x + 3.1 \times 10^{-10}x^2 + 9.3 \times 10^{-15}x^3$$

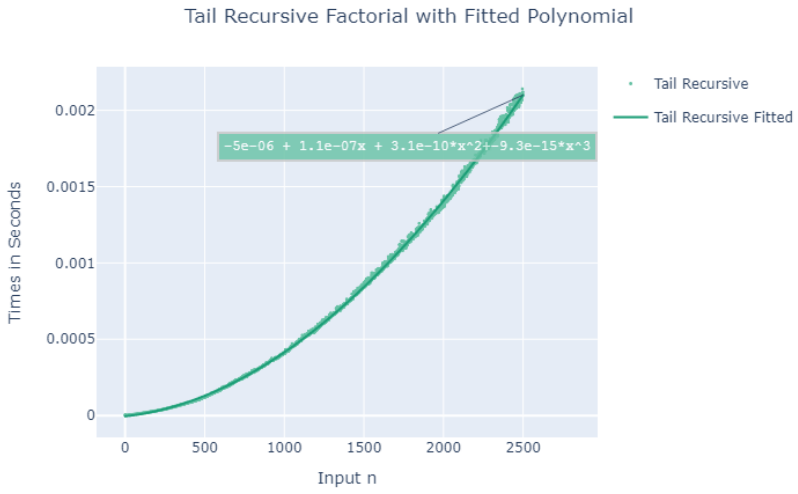

Figure 4.3: Tail Recursive Factorial Running Times

Comparison of Factorial Running Times

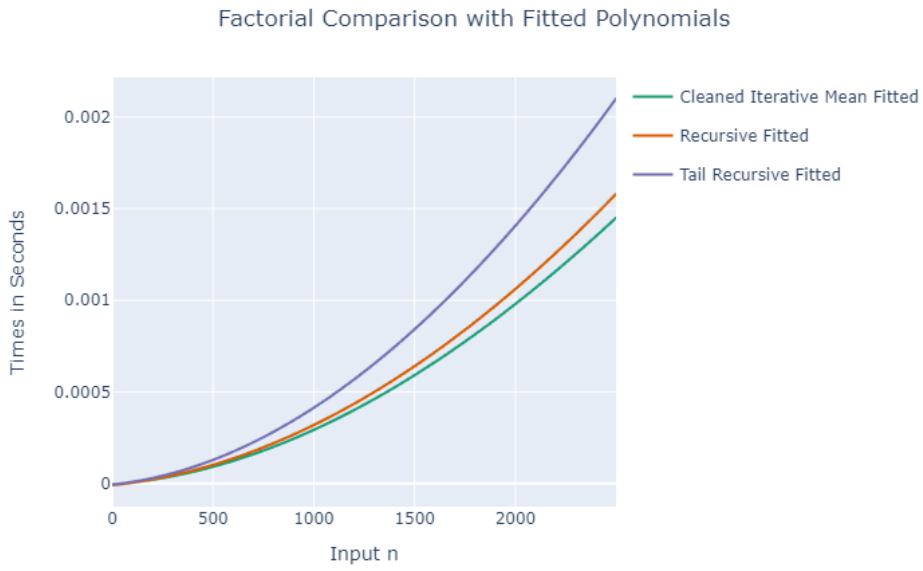

Figure 4.4: Factorial Running Time Comparison

4.0.2 Search Functions

These functions were ran 1000 times with the input arrays ranging from length 1 to 2500. The same random element was searched for within the arrays using the various search functions. The time to return the output was recorded and stored in respective dataframes for each function. The dataset was then cleaned from outliers as above and the mean running time for each array length was found.

Linear Search

I decided to plot the running times of the linear search algorithm for the best, average and worst cases. The linear trend is clear to see across both the Worst and Average cases as we would expect. Linear search has time complexity $O(\mathrm{n})$ so we expect the running time figures to be linear. The best case also looks as expected, near constant running time $O(\mathrm{n})$.

Figure 4.5: Linear Search Running Times Scatter

Iterative Binary Search

I have fitted three curves to the Iterative binary search running time data. Both the Worst and Average cases follow a log2n curve which they have been fitted with. The various parameters of the curve can be seen in the plot below. It is clear to see that the running time data closely follows these fitted curves. The best case can be seen to run in constant time, with some that take slightly more than shown in the curve. This is to be expected with Python and if more iterations were ran these would be expected to be averaged out.

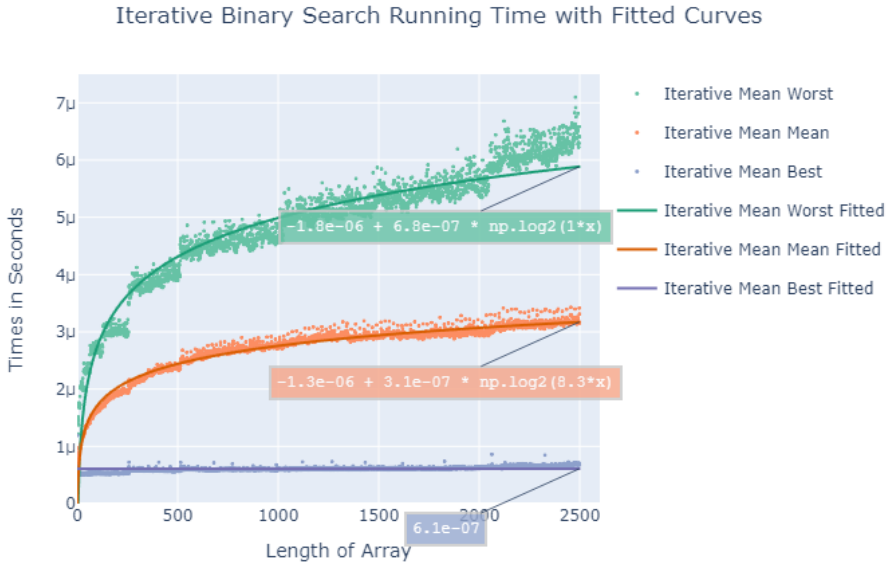

Figure 4.6: Iterative Binary Search Fitted Curves

Recursive Binary Search

I have also fitted three curves to the Recursive binary search running time data. Both the Worst and Average cases follow a log2n curve which they have been fitted with. The various parameters of the curve can be seen in the plot below.

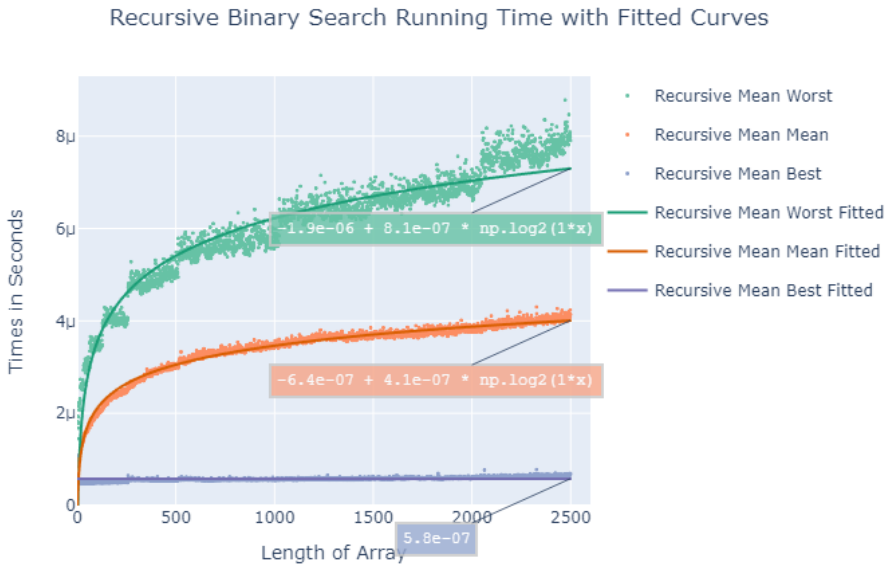

Figure 4.7: Recursive Binary Search Running Time Comparison

Comparison of Binary Search Running Times

Looking at the Binary Search Curves in comparison gives us a clear view of the times taken by each of the algorithms. The iterative search clearly takes less time than the recursive algorithm. This is due to the fact that Recursion is more expensive than Iteration in many ways. Python must add to the stack with each recursive call, where iteration does not need this. Nevertheless both curves follow similar patterns to each other - i.e. both being log2n, even given these differences.

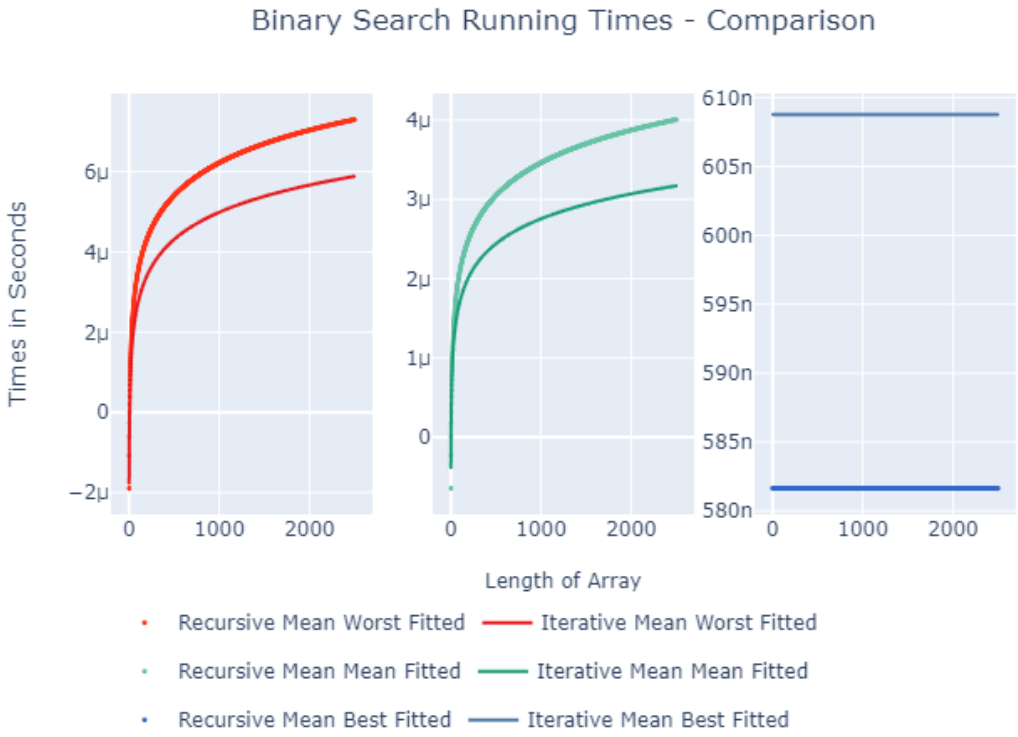

Figure 4.8: Binary Search Running Time Comparison

Chapter 5: **Summary and Conclusions**

In the above, we have analysed the running time of a number of a different algorithms using both Big-Oh notation and then using Python to record the actual running times of these algorithms in practice.

The results shown here show that the time complexity of algorithms relates directly back to the actual running time of these algorithms in computer programs. By studying the above, it is clear to see the importance of time complexity analysis in writing efficient programs.

Further analysis in this could include looking deeper into the implementations of basic operations and data structures in Python to gain a better insight into why certain unexpected times occur - such as the running time of the factorial algorithms which we expected to to be linear but in practice were seen to be of the polynomial form.

I have provided extra comments and analysis in the attached Jupyter notebook, where you can also find unit testing of the algorithms before using and also various profiling techniques to analyse the running times of the code line by line.

Bibliography

[Bor06] Brian Borchers. "The Art of Computer Programming, by D.E. Knuth". In: *Scientific Programming* 14.3-4 (Jan. 2006), pp. 267–268. ISSN: 1058-9244. DOI: 10.1155/2006/281781. URL: https://doaj.org.

[Tou] Syed Tousif Ahmed. *Calculating the factorial of number recursively (Time and Space analysis) | by Syed Tousif Ahmed | Medium*. URL: https://syedtousifahmed.medium.com/calculating-the-factorial-of-number-recursively-time-and-space-analysis-dd47ac5f2607 (visited on 03/24/2021).