De La Salle University

Posteriori Analysis of Algorithms Through Derivations of Growth Rate Based on Frequency Counts

A Thesis
presented to
the Faculty of the College of Computer Studies
De La Salle University

in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Computer Science

by

Guzman, John Paul

Teresita Limoanco
Adviser

August 21, 2016

De La Salle University

Abstract

Literature has shown that apriori or posteriori estimates are used in order to determine the efficiency of algorithms. Apriori analysis determines the efficiency following the algorithm's logical structure while posteriori analysis accomplishes this by using data from calibrated experiments. The advantage of apriori analysis over posteriori is that it does not depend on other factors aside from the algorithm being analyzed. This makes it more thorough, but is limited by how powerful the current methods of mathematical analysis are. We present an empirical study involving posteriori analysis through the analysis of the measured frequency counts of a given algorithm. The developed method uses a series of formulas that extracts an approximation of the asymptotic behavior from the frequency count measurements. These formulas enable one to get an accurate insight on the asymptotic behavior of an algorithm without the need to do any manual computations or mathematical analysis. The method is initially tested on 26 Python programs involving iterative statements and recursive functions to establish the method's accuracy and correctness in determining programs' time complexity behavior. Results have shown that the developed method outputs accurate approximations of time complexity that correspond to manual apriori calculations.

Keywords: Algorithm Analysis, Class complexity computability theory, time complexity.

Contents

List of Figures

List of Tables

Chapter 1

Research Description

Algorithm analysis is used to measure the efficiency of algorithms. This can be used to compare various algorithms and identify the optimal one. It can also be used to determine if the finite resources of a machine are sufficient to run a particular algorithm within a reasonable span of time. A new method of analyzing algorithms is developed in this study. This developed method automatically outputs an algorithm's asymptotic behavior based on experimental data (in terms of frequency count measurements) gained from running the algorithm multiple times on different input sizes.

1.1 Overview of the Current State of Technology

There are two algorithm analysis methodologies: posteriori and apriori analysis. The apriori analysis determines the efficiency of an algorithm based on its behavior and logical structure while the posteriori analysis determines the efficiency of an algorithm based on experiments (Greene et al., 1982).

The posteriori method analyzes an algorithm's performance through empirical data. This is usually done when there are other algorithms to compare with. Statistical facts on the data gathered are used to draw conclusions on the efficiency of the algorithms. This method is usually automated but different hardware and software used in the experiments may yield different results (Pai, 2008).

The apriori method analyzes the logical flow of the algorithm. It quantifies the amount of resources being consumed by keeping track of it while tracing the logical execution of the algorithm. Its goal is to classify an algorithm based on its asymptotic behavior which serves as the metric for its efficiency (Pai, 2008). The advantage of this method is that it does not depend on external factors like those in posteriori analysis (Cormen et al., 2001). However it is is limited by how powerful the current methods of mathematical analysis are. Because of this limitation, there exists algorithms that are too complex to be analyzed. An example of this is the Ackermann Function which can be implemented through the algorithm shown in Definition 1.1.1. It is notable for being one of the simplest examples of function that is not primitive recursive. This is because it contains a double recursion that is not reducible to a primitive recursion which implies that it is not a linear recursion. Currently, there is no general method to solve nonlinear recursions (Rabinovich et al., 1996).

$$Ack(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ Ack(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ Ack(m-1, Ack(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases} \tag{1.1.1}$$

Another example is a function $Op(a,n,b)$ which encodes the sequence of arithmetic operations in the following way:

$$Op(a,0,b) = b+1$$
$$Op(a,1,b) = a+b$$
$$Op(a,2,b) = a*b$$
$$Op(a,3,b) = a^b$$
$$\vdots$$

This is closely related to Ackermann's three argument function $\phi(m,n,p)$ which can be seen in Ackermann (1928). It is considered to be the predecessor of the Ackermann Function. This function can be implemented using the algorithm shown in Listing 1.1.

Listing 1.1: Function Op(a, n, b) implementation

```
Op(a, n, b){
 if (n==0) //base (operator) case:successor function; f(x)=x+1
      return b+1;
 if (b==0) //returns the identity element of the (n-1)th operator
               //or returns X such that Op(a,n-1,X)=a
      if (n==1)
           return a;
```

```
        if (n==2)
            return 0;
        if (n>2)
            return 1;
return Op(a, n-1, Op(a, n, b-1));}
```

Another example of an algorithm that cannot be analyzed is the Sudan function $F_n(x,y)$ published in Sudan (1927) and it is defined to be:

$$F_n(x,y) = \begin{cases} F_0(x,y) = x + y \\ F_{n+1}(x,0) = x \text{ if } n \geq 0 \\ F_{n+1}(x,y+1) = F_n(F_{n+1}(x,y), F_{n+1}(x,y) + y + 1) \text{ if } n \geq 0 \text{ and } y > 0 \end{cases}$$

If apriori methods such as the iterative method is used on these functions, then there may be steps in converting the expression into a closed form that will need formulas that have not been discovered yet. For instance, in the case of the Ackermann function there is a point where a closed-form formula for the following series is necessary (see Appendix A.3):

$$f(n) = x + x^x + x^{x^x} + x^{x^{x^x}} + ... + x^{\cdot^{\cdot^{\cdot^x}}},$$

where the last term is a real number x exponentiated with itself n many times.

Another motive for analyzing algorithms like these is that there are infinitely many of them. This is easily shown by using arguments of construction. For instance, in the case of Op(a, n, b), it is possible to create another algorithm by changing "return b+1" into "return b+1+1;" or "return b+1+1+1;" or by changing "return Op(a, n-1, Op(a, n, b-1));" into "return 1+Op(a, n-1, Op(a, n, b-1));" or "return 1+1+Op(a, n-1, Op(a, n, b-1));" and so on. This argument can also be used on other properties of the algorithm like the number of recursions, the number of parameters, the number of base cases, and so on. There are infinitely many variations of algorithms like these and each one of these variations produces strange and interesting results that have yet to be fully understood.

Single parameter algorithms containing linear recurrence relations are difficult to analyze through apriori means, but it is possible in theory (Gossett, 2009). A linear recurrence is a recurrence which calls itself with decrementing parameter size for each term with constant coefficients, an example of this can be seen in Equation 1.1.2 where c's are constant real numbers.

$$F_n = c_1F_{n-1} + c_2F_{n-2} + c_3F_{n-3} + c_4F_{n-4} + c_5F_{n-5} \tag{1.1.2}$$

The behavior of linear recurrences could be determined by observing their corresponding characteristic equations. Characteristic equations are polynomial equations to the same degree as its corresponding recurrence relation. The characteristic equation for Equation 1.1.2 is a quintic equation or a polynomial equation of the 5th degree. It is difficult to determine the behavior of this recurrence relation because there is no explicit closed-form formula to identify the roots of a polynomial equation with degree 5 or greater which was proven in Abel (1824). However, it is possible to acquire numerical approximations of these roots.

There are single parameter recurrence relations that are nonlinear. These are much more complex than linear recurrence relations and they are not generally solvable (Rabinovich et al., 1996). Equation 1.1.3 is an example of a nonlinear recurrence relation.

$$F_n = (n)F_{n-1} + (n^2)c_2F_{n-2} + 3n \tag{1.1.3}$$

1.1.1 Statement of the Research Problem

There are algorithms that are too complex for current apriori and posteriori methods to accurately determine asymptotic behaviors which are necessary for measurement of the general efficiency of algorithms and their classification based on complexity classes.

1.2 Research Objectives

1.2.1 General Objective

To develop a method of algorithm analysis that extracts information out of a sequence of frequency count measurements that will yield the asymptotic behavior of an algorithm.

1.2.2 Specific Objectives

1. To study the properties of different rates of growth, sequence convergence, and properties of asymptotic relationships.
2. To create an preprocessor that parses a Python file and augments frequency count increments for each line that executes an instruction.
3. To create a formula that uses the elements of the sequence of frequency counts which yields approximations of the asymptotic behavior of the algorithm.
4. To create an algorithm that checks if a sequence converges or diverges to a real number.
5. To create an algorithm which utilizes Objectives 1 to 4 to determine and verify the asymptotic behavior of the algorithm that generated the given sequence.
6. To create a prototype that takes in an algorithm implemented in Python and outputs its asymptotic behavior.
7. To create a tool that graphs the output growth of the prototype along with the measured growth of the input.
8. To test the prototype using several algorithms with varying structure and complexity.

1.3 Scope and Limitations of the Research

The developed method covers rates of growth from logarithmic to exponential including constant or no growth. This is because there an infinite number of growth rates and most algorithms fall under this. And anything beyond logarithmic and exponential will grow too slow to detect and explode too fast to compute for accurately, but it is possible to extend the scope further in future work.

The method is not guaranteed to output an exact answer, but it could generate answers up to any arbitrary precision depending on the number of terms in the sequence being analyzed. The next term of a sequence could always be generated, assuming that there is sufficient computing power for the calculation of the next term in the sequence.

This method only works by manipulating numeric sequences and not by manipulating functions; therefore, current convergence tests, which involves taking limits, will not be used. Instead, it will use a discrete and finite type of convergence test which is not as robust as the tests using limits, but it will return an reasonable result given a large enough number of terms in the input sequence.

The method developed in the research focuses on determining the asymptotic behavior given only one parameter. This means that when dealing with multiple parameter functons, the other parameters must be set to some value and only one parameter must have incrementing input size. Algorithm performance depending on multiple parameters would not be considered since it would require manipulating a high dimensional array of numeric sequence; not just one like in the single parameter case. Another limitation of the developed method is when utilizing it on algorithms whose performance depend on external data. Varying the external data will vary the performance of the algorithm without changing its structure. Therefore, the method will not produce a consistent output asymptotic behavior when under these conditions. Furthermore, the method is unable to do the best case and worst case analysis on algorithms of this kind. However, if predefined external data have been manually identified as either the best or worst case structure of data, then the method would be able to do best or worst case analysis.

The input algorithm must be implemented in Python. Every statement that is acceptable by Python is also acceptable by this method except statements with the ”elif” keyword. Note that there are some shorthand statements like the use of ”++” to increment variables or do-while loops which are not accepted by Python. Fortunately, these statements could be simulated using other statements that are valid in Python. For instance, ”elif” could be simulated by nesting an ”if” under an ”else” or ” ””” ” could be simulated using multiple ”#”. This means that method has the ability to process all kinds of complicated nested statements, loops, if-else statements, recursions, and calls to other functions provided that it is defined within the input Python file.

The results of this method will be further supported through the use of graphs. The prototype that will be developed will have a feature where the measured frequency counts and the calculated result are plotted against the size of the input parameter as this will provide a visual and intuitive verification of the results.

1.4 Significance of the Research

There are algorithms that currently cannot be accurately analyzed by current methods due to their complexity. This research study may lead to a more powerful way of algorithm analysis which enables one to accurately analyze these complex functions, and also simple but lengthy algorithms that are very hard to keep track of.

1.5 Research Methodology

This chapter lists and discusses the specific steps and activities that were performed to accomplish the project. The discussion covers the activities from preproposal to final thesis writing. It also includes an initial discussion on the theoretical framework to be followed.

1.5.1 Study of the Prerequisite Concepts

Concepts like the properties of different rates of growth, asymptotic relationships, and sequence convergence were gathered from books and online resources and studied. Asymptotics, algebra, and properties of growth rates within the scope were used in creating the formulas. Calculus was used in the creating an applicable convergence test. Convergence in real analysis was utilized together with asymptotic notations in algorithm analysis was studied to prove the equivalence of two different definitions of asymptotic notations.

1.5.2 Creating the Preprocessor

The input algoritm is parsed and augmented with frequency count incrementers. These incrementers automatically generate the performance measurements of the algorithm. These measurements are used for the method and its heuristics.

1.5.3 Creating an Applicable Convergence Test

The convergence test that is applicable to the method being develop is an algorithm that takes in a sequence of real numbers and determine whether or not this sequence converges to a real number limit. If it does converge, it must converge to 1 to satisfy the validation. This is crucial to the method being developed because it depends on the idea of asymptotic equivalence. Asymptotically equivalent functions are functions whose ratios approach 1 as the input size approaches infinity. This test was used to verify the correctness of the results calculated by the developed algorithm.

1.5.4 Identify the Formulas For Approximating an Asymptotic Behavior

The understanding of the prerequisite concepts was used in deriving formulas that would determine the asymptotic behavior of a given sequence of frequency counts. The derivations for these formulas are shown in Chapter 5 and Appendix B.

1.5.5 Development of the New Method of Algorithm Analysis and Implementation

The formulas and convergence test were both implemented in Python. Python was used because of the way it stores numbers and do operations. It automatically switches to arbitrary-precision arithmetic when dealing with very large numbers and it also allows users to set the desired numerical precision for small numbers that require many decimal places.

1.5.6 Testing of Implemented Method

The method was tested with pre-analyzed algorithms and compared the results to their actual behavior. Apriori methods, namely, the iterative method and solving linear recurrence relations were used in the pre-analysis of the algorithms for testing.

1.5.7 Generating Graphs from the Results

The implementation includes a feature to a generate graph to compare the output asymptotic behavior with the measured frequency counts of the input. This would provide a visual and intuitive verification of the output.

1.5.8 Documentation

Documentation is written text that accompanies the development of this method. The documentation also includes the testing and results of these tests, i.e. successes, failures, and accuracy of the method.

1.5.9 Calendar of Activities

Table 1.1 shows a Gantt chart of the activities. Each asterisk represents approximately one week worth of activity.

	STRESME			Summer Break			THSST-1					THSST-2				THSST-3			
Activities (2015-2016)	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug
Study of the Prerequisite Concepts	****	****	****	****	****	****	****	****	**										
Creating an Applicable Convergence Test							****	****	**										
Deriving the Formulas to be Used						****													
Development of the New Method of Algorithm Analysis and Implementation						****	****	****	****						****	****	****		
Testing of Theoretical Framework									****	****			****	****				****	****
Generating Graphs from the Results									****	****			****	****					
Documentation	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****	****

Table 1.1: Calendar of Activities

Chapter 2

Review of Related Literature

This section discusses existing works or systems that measures the performance of algorithms.

2.1 Posteriori Analysis

Posteriori analysis refers to running experiments on an algorithm and then analyzing the data gathered from these experiments to determine the efficiency of an algorithm (Pai, 2008). The following subsections will discuss common methods of posteriori analysis.

2.1.1 Profiling

Profiling is a common posteriori technique that has many variations, but all profiling techniques follow three distinct steps known as the augment-run/execute-analyze process. Augmentation is the preprocessing of the algorithm where special code are placed into the original algorithm. These special code will measure the resource consumption an algorithm along its execution. Execution is the actual processing of the algorithm. This is the step where the algorithm is executed and the augmented code will generate the data that will be stored to a special file or temporary memory. Analysis is the processing of the generated data. It utilizes a special program called the analyzer that gathers all the data to generate a report on the performance of the algorithm. There are a lot of techniques concerning profiling, but all of them fall into two main categories: instruction counting and clock-based timing. These two main categories will be discussed in the following subsections (*Profiling of Algorithms*, n.d.).

Instruction Counting

Instruction Counting is concerned with the measurement on the number of times particular instruction(s) are executed. Only certain instruction executions are recorded in augmentation. An example could be found in Listing 2.1 and Listing 2.2 where comparisons and assignments were the specified instructions to be counted. The comparisons were counted using the *comp_count* variable while the assignments were counted using the *assign_count* variable.

Listing 2.1: Bubble Sort

```
begin
     for i <— 2 to array_size do
          for j <— array_size downto i do
               if a[j-1] > a[j]
                    then {swap}
                         temp <— a[j-1];
                         a[j-1] <— a[j];
                         a[j] <— temp;
                    end if;
               end for;
          end for;
     end program.
```

Listing 2.2: Augmented Bubble Sort with counting instructions

```
begin
     comp_count <— 0;
     assign_count <— 0;
     for i <— 2 to array_size do
          for j <— array_size downto i do
               comp_count <— comp_count + 1
               if a[j-1] > a[j]
```

```
                    then {swap}
                        temp <-- a[j-1];
                        a[j-1] <-- a[j];
                        a[j] <-- temp;
                        assign_count <-- assign_count + 3;
                    end if;
                end for;
            end for;
        end program.
```

This technique is advantageous due to its simplicity in implementation it does not introduce an error in the quantities being measured. Although, it is undeniable that instruction counts are not always definitive in measuring the actual performance of an algorithm as it will only output simple measurements (*Profiling of Algorithms*, n.d.).

Clock-Based Timing

Clock-Based Timing is concerned with measuring the time required for certain blocks of code to execute. There are two principal types of clock-based timing: Fixed position logging and Random-sample logging, and they are discussed in the following subsubsections (*Profiling of Algorithms*, n.d.).

Fixed-Position Logging

In fixed position logging, augmented instructions are put within the program to be profiled. These instructions will log the elapsed running time. Listing 2.3 shows the augmentation of the algorithm shown in Listing 2.1 (*Profiling of Algorithms*, n.d.).

Listing 2.3: Augmented Bubble Sort with clock monitoring

```
begin
    begin_time <-- clock();
    for i <-- 2 to array_size do
        for j <-- array_size downto i do
            if a[j-1] > a[j]
```

```
                then {swap}
                    temp <-- a[j-1];
                    a[j-1] <-- a[j];
                    a[j] <-- temp;
                end if;
            end for;
        end for;
        end_time <-- clock();
        elapsed_time <-- end_time - begin_time;
    end program.
```

The function clock() logs the current ”time-of-day” clock or the ”system-uptime” clock. It returns the current time of the system. The difference between the current time at the start and end of the algorithm is its running time. It could also be placed on different parts of the algorithm to determine their contributions to the running time of the entire algorithm.

Random-Sample Logging

In random-sample logging, the algorithm is executed at random times, determining which procedures or statement is executing. From the samples and knowledge of the total running time of the program, the profile of the time spent executing each procedure is constructed. The implementation of random-sample logging is centered around a compiler support. At compilation, a table that matches addresses to routines is constructed. At execution, the sampler looks at the address of the executed instruction, now being able to determine which specific instruction is currently executing. The number of times which each routine is called is also computed. Total running time could be determined using the sample rate and the number of samples collected (*Profiling of Algorithms*, n.d.).

Take for example an algorithm being profiled. The sample rate is measured at 1000 per second, and there are 10000 samples taken. Given in a recorded call data which can be found in Table 2.1.

Procedure	No. Of Calls	No. Of Sample Hits
A	300	1000
B	600	2000
C	1000	7000

Table 2.1: Sample Recorded Call Data 1

The algorithm's total running time is 10 seconds which can be computed using Table 2.2.

Procedure	Total Time	Time per Call
A	1 sec.	3.33 msec.
B	2 sec.	3.33 msec.
C	7 sec.	7.00 msec.

Table 2.2: Sample Recorded Call Data 2

2.1.2 Input-sensitive Profiling

Input-sensitive Profiling is a type of profiling that returns a function that estimates the growth of the input algorithm rather than simple measurements. A method of this type is discussed in McGeoch et al. (2002). The method maps input sizes to measurements of the performance of the input algorithm. It utilizes several rules to determine the relationship between the measurements and the input size. An example of a rule is the Guess Ratio rule which assumes that the performance follows a function of the form:

$$P(n) = \sum_i (a_i n^{b_i}),$$

where a and b are non-negative rational numbers. It attempts to identify a guess function of the form:

$$G(n) = n^b.$$

It uses the fact that if $P(n) \notin O(G(n))$, then $\dfrac{P(n)}{G(n)}$ must be increasing for high values of n. The method starts by setting b to 0 and increments b until $\dfrac{P(n)}{G(n)}$ becomes nonincreasing which is when the rule infers that $P(n) \in O(n^b)$.

The method utilizes Oracle functions which guides the execution of the rules. An example of an Oracle function is the Trend function (McGeoch et al., 2002). The Guess Ratio rule uses the Trend function to determine if $\dfrac{P(n)}{G(n)}$ is decreasing, increasing, or neither.

Although the method do give promising estimates of asymptotic behavior, it is limited to determining accuracy only up to Big O and Big Omega asymptotic notation as results. This can be seen in the Guess Ratio rule. Since the method only does guesses and checks, it is very possible for the value of b to be an overestimation. For instance, if the actual performance function follows $n^{1.8}$, the accepted guess function will be n^2 since b increments from 0 to 1 and then to 2 where Trend stops the loop which introduces a significant discrepancy and looses information contained in the performance function.

It also lacks rigorousness since some Oracle functions depend on concepts such as Pearsons correlation coefficient and linear regression models which produce reasonable results only for linear relationships.

2.2 Apriori Analysis

Apriori analysis refers to analyzing the logical flow of an algorithm and identifying its asymptotic behavior to determine its performance (Pai, 2008). Performance may have different metrics such as operation count, frequency count, or execution time, but the discussions will only focus on frequency count. Frequency count is the measurement for the number of statements being executed by an algorithm. The following subsections will discuss common methods of apriori analysis.

Iterative Method

The Iterative method is done by expanding any iterations or recursions present in the algorithm and determining the frequency count for each expansion. And then, identifying a pattern or a series that matches the behavior of frequency counts. The generalization of the pattern together with the initial condition or base case will determine the behavior of the algorithm (Cormen et al., 2001). The necessary mathematics or computations to identify a pattern or a series that matches the given input algorithm may not exist for all algorithms as discussed in Section 1.1.

Recursion Tree

The Recursion tree is used on recursive algorithms. It is useful for visualizing the expansion of recurrences. It diagrams the tree of recursive calls and the amount of work done at each call. It works under the same mechanism as the Iterative method by observing different expansions and looking for patterns (Cormen et al., 2001).

Master's Method

The Master's method is informally called the cookbook for solving recurrences. It is a theorem that allows one to deduce the behavior of an algorithm by knowing some of its particular properties. Given that a recursive algorithm has the following form:
$T(n) = a * T(\frac{n}{b}) + f(n)$, where a and b are both greater than 0.

According to Cormen et al. (2001), the Master's method has three different cases which states that:

$$\exists_E(f(n) \in O(n^{log_b(a-E)}) \wedge E > 0) \rightarrow (T(n) \in \Theta(n^{log_b(a)})) \tag{2.2.1}$$

$$(f(n) \in \Theta(n^{log_b(a)})) \rightarrow (T(n) \in \Theta(n^{log_b(a)} * \ln(n))) \tag{2.2.2}$$

$$(\exists_E(f(n) \in \Omega(n^{log_b(a+E)}) \wedge E > 0) \wedge (\exists_c(a * f(\frac{n}{b}) \leq c * f(n) \wedge c < 1))) \\ \rightarrow (T(n) \in \Theta(f(n))) \quad (2.2.3)$$

An in-depth explanation on why these three statements are true can be seen in *Proof of the Master Method* (n.d.).

Characteristic Equation

Linear recurrence relations could be evaluated by using Characteristic equations (Gossett, 2009). Given the general linear recurrence:

$$a_n = c_1 a_{n-1} + c_2 a_{n+2} + c_3 a_{n-3} + \ldots + c_k a_{n-k} \quad (2.2.4)$$

Its corresponding Characteristic equation will have the form:

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - c_3 r^{k-3} + \ldots - c_k = 0 \quad (2.2.5)$$

If the Characteristic equation has roots $r_1, r_2, r_3, \ldots, r_k$, then a_n will have the closed form:

$$a_n = C_1 {r_1}^n + C_2 {r_2}^n + C_3 {r_3}^n + \ldots + C_k {r_k}^n \quad (2.2.6)$$

All of the C's could be solved by creating a system of linear equations from the k base cases and using algorithms such as Gaussian elimination to solve for each C_i.

Chapter 3

Theoretical Framework

This chapter discusses the possible interpretations of algorithm analysis. The majority of the discussion concentrate on the type of algorithm analysis of developed method which is Time complexity. This chapter also includes the discussion on the properties of different rates of growth that will be used in the method.

3.1 Algorithm Comprehension

Algorithm analysis may pertain to code recognition or understanding the functionality of a program or what the program does. For example, given a sorting algorithm, understanding that algorithm means one can say that the algorithm does the sorting functionality, and its output would be a set conforming to a particular order, i.e. ascending order or descending order (Hantler et al., 1976).

3.2 Algorithm Correctness

Algorithm analysis may also pertain to the validity of an algorithm. It is said that an algorithm would be considered correct if and only if, for any valid input, it produces the result required by a given specification. These can be done in multiple ways, namely, testing, model-checking (states), assertions, invariants,

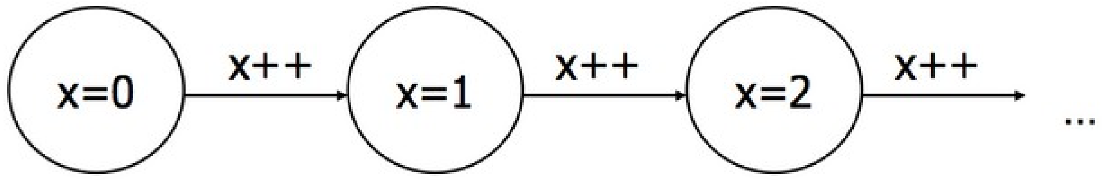


Figure 3.1: Automata assertions

and lastly, by design. Testing is running the algorithm and checking its output's correctness. Model-checking, on the other hand, represents the program to be verified as an automata where states are values of variables and transitions are atomic actions of the algorithm; as sample automata can be seen in Figure 3.1. As every automata should function, this automaton exhaustively searches the transition system for error states and then finally says the validity of the algorithm (Hantler et al., 1976).

3.3 Algorithm Performance

Algorithms have a certain optimization goal to meet, i.e. computing the minimal edit distance, or the shortest path, or the most cost-efficient combination of things, etc. In exact algorithms, the focus is more at computing the optimal solution given such a goal, and most of the time, this is very inefficient in terms of run time or memory, which makes it not viable for large inputs. Approximation algorithms focus on computing the solution which has a guaranteed factor worse than the optimal solution. Alternatively, there are heuristic algorithms that will always try to get the optimal solution but it can not guarantee that they will always do. Lastly, there are algorithms that does not focus on optimizing an objective function, Bockmayr et al. (2006) calls them operational algorithms because they chain a series of computational operations with accordance to expert knowledge but without a specific objective. In order to provide a general grading system for all these types of algorithms, the concept of run time analysis was formulated. Although, they are only applicable to deterministic algorithms, whereas the opposite of randomized algorithms which tend to have variables during execution, resulting to the order of execution or the result of the algorithm itself, undeniably different for every run. There are also different run time definitions, namely: worst case, best case, and average case analysis; which are done by considering the maximum, minimum, and average number of basic operations each must perform on any given input. In all cases, it will determine, for every function, their relationship between input size and run time, i.e. constant, grows logarithmically grows linearly, grows quadratically, grows polynomially, exponentially, etc (Bockmayr et al., 2006). There are two ways to perform algorithm analysis based on performance and these are apriori analysis and posteriori analysis which were discussed in Section 1.1.

3.3.1 Measurements of Algorithm Performance

The performance of an algorithm can be measured in terms of computational resource such as space and time usage. Algorithms are classified to certain complexity classes based on the amount of space and time it uses to execute. For instance, algorithms classified under the exponential class will generally take more time and consume more space than algorithms classified under the polynomial class. The basis for this separation is the Hierarchy Theorem which states that number of problems solvable by a machine increases as the amount of resources available increases (Hartmanis et al., 1965).

Time Complexity

In terms of time complexity, one fundamental complexity class is polynomial time (P) which is the collection of all problems that can be solved in polynomial time. Algorithms such as Path Finding, Searching and Sorting are all elements of P. Another fundamental complexity class is Non-deterministic polynomial time (NP) which is the collection of all problems that with solutions that can be verified or proven in polynomial time. An equivalent way of describing NP is the collection of all problems that can be solved in polynomial time provided that all the possible solutions are traversed simultaneously. The Clique problem where the task is to determine if a graph contains a clique larger than a given size is an example of an NP problem. Another fundamental complexity class is exponential time (Exp) which is the collection of all problems that could be solved in exponential time. An example of this is a brute-force password cracking algorithm which goes through all possible combinations of inputs. Another fundamental complexity class is finite time (R) which is the collection of all problems that could be solved in a finite amount of time (Cormen et al., 2001).

According to Donail (2006), since NP is much more powerful (contains harder problems) than P, P is a subset of NP; and an exponential growth grows faster than a polynomial growth, NP is a subset of Exp; and an exponential growth will always be finite, Exp is a subset of R. Although, it is not yet known if NP is a subset of P or P=NP. These relationships are represented in Figure 3.2.

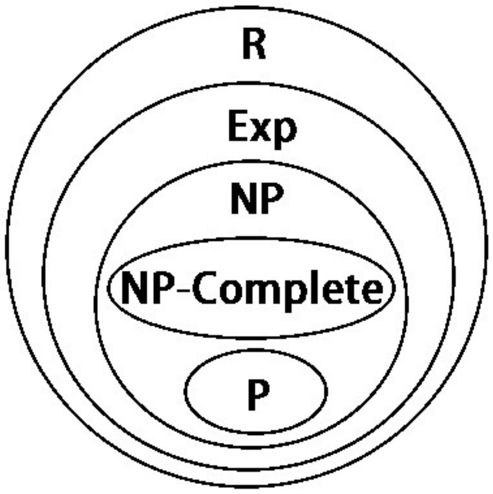


Figure 3.2: Time complexity classes

Space Complexity

Space complexity have similar classes than that of time complexity, but it has been proven that some space complexities are actually equal. This is due to Savitch's theorem which states that if a non-deterministic machine can solve a problem using $f(n)$ space, an ordinary deterministic machine can solve the same problem in $f(n)^2$ space. Therefore in space complexity, NP space actually equals P space (Sutner, n.d.).

3.4 The Properties of Asymptotic Functions

Big O and Little O describes the asymptotic upper bound of an algorithm within a constant factor; informally, how slow an algorithm must run at most. Big

Omega and Little Omega describes the asymptotic lower bound of an algorithm within a constant factor; informally, how fast an algorithm must run at most. The difference between the Big and Little notations are that the Big notations have a possibility of being an asymptotic tight bound while the Little notations do not. A tight bound means that it more or less follows the time complexity of the algorithm. Big Theta describes the asymptotic tight bound of an algorithm within a constant factor (McConnell, 2008). These could be mathematically defined using limits (Sedgewick et al., 2013) as shown below.

Given that f(n) and g(n) are functions, these five notations can be defined as:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = c, c < \infty \longleftrightarrow f(n) \in O(g(n)) \quad \text{\\Big O} \tag{3.4.1}$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = c, 0 < c < \infty \longleftrightarrow f(n) \in \Theta(g(n)) \quad \text{\\Big Theta} \tag{3.4.2}$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = c, c > 0 \longleftrightarrow f(n) \in \Omega(g(n)) \quad \text{\\Big Omega} \tag{3.4.3}$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = c, c = 0 \longleftrightarrow f(n) \in o(g(n)) \quad \text{\\Little O} \tag{3.4.4}$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = c, c = \infty \longleftrightarrow f(n) \in \omega(g(n)) \quad \text{\\Little Omega} \tag{3.4.5}$$

Note that these are not the commonly used definitions. The definitions of the five asymptotic notations are stated in McConnell (2008), but they are proven to be equivalent to the limit definitions as shown in Appendix D.

The problem with these five notations is that they are not very accurate. Big O and Little O only describe the slowest possible time the algorithm will run, while the Big Omega and Little Omega only describe the fastest possible time the algorithm will run; implying that these four notions do not describe how fast an algorithm actually runs. For instance, given $f(n) = 5n \in O(5^n)$ is a true statement and knowing just $O(5^n)$ will not give any further insight on how fast f(n) runs; and if one tries to compare it with $g(n) = n^3 \in O(n^3)$, the two completely valid Big O expressions will state that $g(n)$ is much faster than $f(n)$ since the Big O of $g(n)$ is much smaller than the Big O of $f(n)$ which is absurd.

Among the five, Big Theta is the most accurate one because it provides some description on how fast the algorithm runs. For instance, given
$f(n) = 25n \in \Theta(n)$, knowing $\Theta(n)$, one can say with certainty that $f(n)$ must

exhibit some kind of linear growth. The problem with Big Theta is that it is not accurate enough to determine the constant factor which means that knowing $\Theta(n)$ will not imply that the given was $f(n) = 25n$. This leaves open the possibility of having two algorithms with the exact same $\Theta(g(n))$ but one may be 25 times slower than the other. Constant factors are important because they will never become negligible no matter how much the input size grows. This can be shown mathematically as any factor k within a limit could be factored outside the limit (Fradkin, 2013):

$$\lim_{n\to\infty} \frac{kf(n)}{g(n)} = \lim_{n\to\infty} k\frac{f(n)}{g(n)}.$$

This flaw in the notations are due to the nature of comparison. Comparison deals with ordinal concepts like ”less than”, ”greater than”, and ”equal to”. These ideas produce well-defined answers when comparing definite values like $13 < 24$. However, they produce undefined results when comparing ranges of possible results. For instance, let a be a number less than 100 and let b be a number less than 100 that is not equal to a. Given these two definitions, a could be 10 while b could be 80 which means that $a < b$. However, a could also be 0 while b could be -123 which means that $a > b$. Notice that all values that were chosen valid based on the definitions but one could get two completely contradictory results namely, $a < b$ and $a > b$ which makes the comparison undefined. This dilemma in comparison applies to quantities that are defined with ranges; i.e., defined by inequalities. It does not apply to quantities that are defined with singular possible values; i.e., defined by equalities.

A better way of determining the asymptotic behavior of an algorithm is by looking at a function that is asymptotically equivalent to it as the input gets larger. It can be defined using limits as seen in Definition 3.4.6. Intuitively, this means that both functions in the numerator and denominator grow exactly the same way thus having a ratio of 1. Asymptotic equivalence could be thought of as Big Theta that is also accurate up to a constant factor; this can be seen by observing Definitions 3.4.2 and 3.4.6. Although asymptotic equivalence is a better approximation and it holds more information than any of the five notations, it is currently not a widely recognized asymptotic notation. Fortunately, asymptotic equivalence can be considered as Big Theta. In fact, if one chooses to ignore the constant factor present in an asymptotic equivalence, then it is exactly Big Theta.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c = 1 \longleftrightarrow f(n) \sim g(n) \text{ as } n \to \infty \quad \backslash\backslash\text{Asymptotic equivalence} \tag{3.4.6}$$

3.5 Properties of Mathematical Operations

The following properties are used in the derivations in Appendix B and Appendix C (Fradkin, 2013).

- $c(a + b) = ac + bc$
- $a^c * b^c = (ab)^c$
- $log_b(a) = log(a)/log(b)$ (logarithm base conversion)
- $log(ab) = log(a) + log(b)$
- $log(a/b) = log(a) - log(b)$
- $log(a^b) = blog(a)$
- $exp(log(a)) = a$
- If $a = b$,
  - $a + c = b + c$
  - $ac = bc$
  - $log(a) = log(b)$
  - $exp(a) = exp(b)$
- If $a = b$ and $b = c$, then $a = c$
- $|ab| = |a||b|$
- $a^2 = |a^2|$

- If $a \geq 0$. then $a = |a|$
- $(a+1)^b = \sum_{i=0}^{\infty} \frac{b! a^{b-i}}{i!(b-i)!}$ (Maclaurin expansion of the binomial series)
- $\frac{d}{da}|F(a)| = \frac{F(a)}{|F(a)|} * \frac{d}{da}F(a)$ (absolute value rule in differentiation)
- $\frac{d}{da}F(G(a)) = \frac{d}{dG(a)}(F(a)) * \frac{d}{da}(G(a))$ (chain rule in differentiation)
- $\frac{d}{da}(F(a) * G(a)) = F(a) * \frac{d}{da}(G(a)) + \frac{d}{da}(F(a)) * G(a)$ (product rule in differentiation)
- $\frac{d}{da}a^b = (b)a^{b-1}$ (power rule in differentiation)
- $\frac{d}{da}b^a = (ln(b))b^a$ (exponential rule in differentiation)
- $\frac{d}{da}log_b(a) = \frac{1}{aln(b)}$ (logarithmic rule in differentiation)

Chapter 4

The System Model, Algorithm, and Design

This study explored the concept of extracting the time complexity of an input program through its frequency count measurements. This is done by observing how the frequency count measurement changes with respect to the input size. This means the method bypassed the limitations of the apriori approach since all of the complexities of the logical structure within a program will be reduced to simple measurements. In other words, the logical progression of a given program will be completely hidden inside a black-box and the method determines its asymptotic behavior by observing what comes out of the black-box instead of what is inside it. The method also bypass the disadvantages seen in posteriori analysis since measuring frequency counts is something that could be programmed and does not depend on hardware specifications.

4.1 Architectural Design

A system has been developed written in Python. Its logical flow can be seen in Figure 4.1 and a brief description of each component is as follows and each items will be discussed in detail in the following subsections:

1. Input - Loads the input algorithm and other parameters in the system.

2. Preprocessing - Augments frequency count increments in the input algorithm.

3. Execution - Executes the augmented input algorithm and stores the frequency count measurements.

4. Calculations for e, p, c, $hasLog$ - Utilizes the frequency count measurements to yield approximations for the e, p, c, and $hasLog$ quantities. The implications of these approximation are discussed in Subsection 5.2.

5. Validation - Tests the asymptotic equivalence between the measured frequency counts and approximations.

6. Output - Presents the validated e, p, c, and $hasLog$ values and an option to graph the measured frequency counts together with the output values.

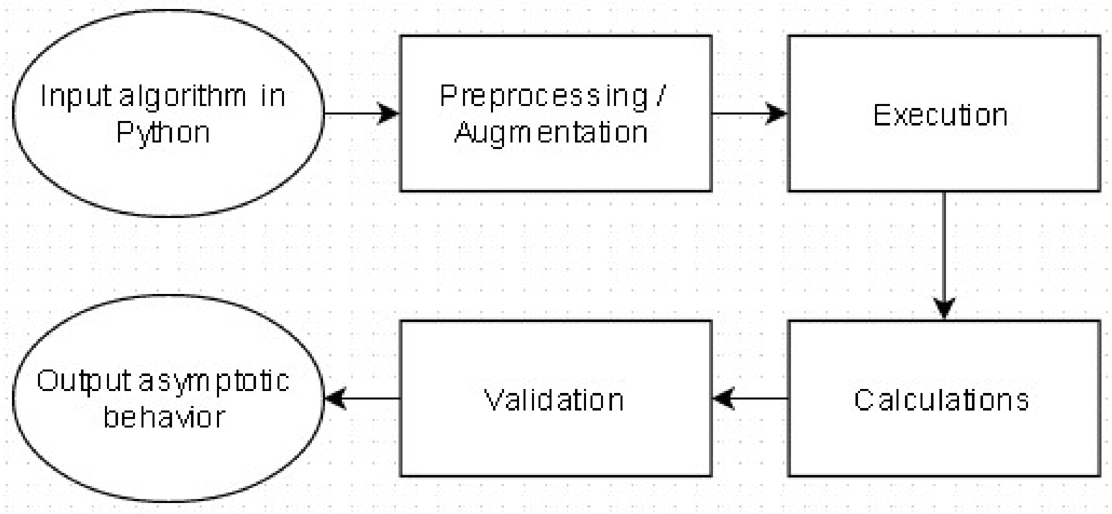

Input algorithm in Python
Preprocessing / Augmentation
Execution
Calculations
Validation
Output asymptotic behavior

Figure 4.1: Architectural Design of the Proposed System

4.1.1 Input

There are two essential parameters in the system. These are the $algoFile$ and $endTerm$. $algoFile$ is the input program written in a Python file. $endTerm$ is the number of terms or incremental input sizes to be computed. There are three nonessential parameters which are automatically set to the recommended values. These are the k, $decimalPrecision$, and $decimalTolerance$. The use of k is discussed in Section 5.4. This is set to 0 by default.

The $decimalPrecision$ is the number of significant figures to be considered in the calculations. This is set to 132 by default. The $decimalTolerance$ is the number of significant figures at the end of each number to round off. This is set to 6 by default. This is necessary because the stored values in the variables are not exactly correct. They will be mostly correct but it will have some discrepancy towards the end. For instance, 4 may be stored as 4.000000 followed by 126 zeroes and then some random nonzero numbers at the end. If the numbers at the end are not rounded off, then it may lead to comparing if 4 is equal to 4 and getting $False$ as a result.

4.1.2 Preprocessing

The system opens the input Python file and create a new Python file where it will store the augmented version of the input. The augmented version of the code has the same logical behavior as the input algorithm but it will have frequency count incrementers for every line that executes an instruction (e.g., conditional check, function call, declaration, assignment). This approach is similar to instruction counting discussed Subsubsection 2.1.1.

The algorithm file being analyzed must have a function named f which takes in only one integer parameter. f is the function that will be ran in the experiments. f may call other functions without any restrictions on the name, and number and type of parameters as long as they defined within the file. Listing 4.1 is a sample algorithm that is a valid input while Listing 4.2 is the augmented version of Listing 4.1 with additional comments.

Listing 4.1: Sample input algorithm

```python
def f(n):
    for i in range(0, n):
        for j in range(0, n):
            x=i+j
```

Listing 4.2: Sample preprocessed output for the algorithm in Listing 4.1

```python
def f(n):
```

```
2       global freqCount
3       freqCount+=1                    #count for i==n
4       for i in range(0, n):
5           freqCount+=1                #count for i++ & i<n
6           freqCount+=1                #count for j==n
7           for j in range(0, n):
8               freqCount+=1            #count for j++ & j<n
9               freqCount+=1            #count for x=i+j
10              x=i+j
```

The frequency count increment on line 3 comes from the checking of the exit condition for the *for loop* on line 4. The increment on line 5 comes from the automated increment of i from the *for loop* on line 4. The increment of freqCount on line 6 comes from the checking of the exit condition for the *for loop* on line 7. The increment on line 8 comes from the automated increment of j from the *for loop* on line 7. The increment of freqCount on line 9 comes from the instruction on line 10.

4.1.3 Execution

The resulting augmented program is then executed on various input sizes starting from 0 and increments by 1 until it reaches the user-defined number of terms. The frequency count variable is set to 0 before every execution of the augmented program and its value is stored into an array of frequency count measurements after each execution. The resulting array will then be the input to the formula discussed in Section 5.2 and derived in Appendix B to determine the asymptotic behavior of the input algorithm. Further discussions about using the system can be seen in Appendix E.

Using Listing 4.2 as an example, the augmented program will be run with incremental parameter sizes starting from 0. The array of frequency count measurements in this case will contain 1, 5, 13, 25, and so on. This is taken from the measured Frequency count is 1 when the input is 0, 5 when the input is 1, 13 when the input is 2, and so on.

4.1.4 Calculations for e, p, c, hasLog

The array of frequency count measurements will have its constants removed using the process from Section 5.4 and then be put through a series of calculations that outputs the asymptotically behavior the input algorithm. The calculations will be discussed Chapter 5 and derived in Appendix B. The e, p, and c approximation are calculated for both log and no log cases. Then it will check for discontinuities in the approximations using the method discussed in Appendix B.5. Informally, a discontinuity is an instantaneous jump in a function. Discontinuities imply the existence of noise or error terms in the measurements. If there are more than two discontinuities found, then the system will calculate another set of approximations around the discontinuous points. It will then choose between the approximations calculated continuously and around discontinuities using a method discussed in Appendix B.5. And then, the system will choose between the log and no log cases using the method discussed in Appendix B.4.

4.1.5 Validation

Once the e, p, c, and $hasLog$ values are known, the system will test if the approximations are asymptotically equivalent to the generated frequency counts using the method described in Section 5.3.

4.1.6 Output

Lastly, the system will output a summary containing the e, p, c, and $hasLog$ values together with the number of terms used to generate the approximations. The number of terms will serve as a metric for the precision of the approximations. This is because the approximations get more precise when the number of terms gets larger. The summary also includes the detected points of discontinuity and if the approximations are calculated around the discontinuous points. The system also provides an option to graph the measured frequency counts together with the approximated growth rate as this will provide a visual verification of growth rate equivalence. The number of terms also allows the results to be duplicated since the results depend on the number of terms utilized.

Chapter 5

Implementation and Analysis of the Proposed Method

This chapter tackles the software implementation of the method being developed and the mathematical basis.

5.1 Software Implementation

The prototype is implemented in Python 2.7.10 using the PyGTK, Matlibplot.pyplot, Math, Decimal, OS, and Sys libraries. The GUI is designed using Glade UI Designer. All of the software used is in their respective 32-bit Windows versions. A program called editbin which is provided by Microsoft is also used. editbin has the ability to increase the stack size that Windows allocates to a program. It will be used on Python to give users more freedom in running experiments without running out of stack.

The prototype consists of four objects, namely the Experimenter, DataProcessor, GUI, and Controller. The Experimenter is responsible for generating the experimental data to be used in the calculations. It is responsible for the Preprocessing and Execution components of the architectural design. The DataProcessor is where all the calculations of method is done. It is responsible for the Calculations and Validation components of the architectural design. The GUI holds all the GUI elements and functions which is responsible for the Input and Output components of the architectural design. The Controller manages the data flow into the Experimenter and DataProcessor and the GUI. The prototype follows the Model-View-Controller (MVC) design where the Model consists of the Experimenter and DataProcessor and the View consists of the GUI. The class diagram of the prototype can be seen Appendix G.

5.1.1 Experimenter

The Controller retrieves the file path of the input algorithm and ending term value from the GUI and passes it to the Experimenter. The experimenter will then augment frequency count incrementers in the input algorithm as stated in Section 4.1.2.

The preprocessing of the input algorithm is as follows. The Experimenter appends a declaration for the global variable $freqCount$ after all function definitions. These will give access to the variable that stores frequency count increments from within the augmented code. The Experimenter then inserts a $freqCount$ increment for each line of code excluding lines starting with *def* or *global* since it is not part of the execution, *else* since its increment has already been accounted for in the preceding *if*, and comments. These inserted lines precede the lines to be executed and follow the same indentation level to preserve the nested nature of the code. The system inserts an additional $freqCount$ increment after the declaration of a *for* or *while* loop which is indented once relative to the loop declaration line. This will serve as the $freqCount$ increment for the condition checks together with the loop counter increments.

The augmented algorithm would then be loaded inside the Experimenter where a global variable $freqCount$ is declared and initialized to 0. The variable n will be incremented inside the a loop from 0 until the user-defined number of terms. Inside this loop, the algorithm will be executed given n as the parameter and the freqCount after the execution will be stored in the $seqFreqCount$ array. After this, $freqCount$ will be set to 0 again and the loop continues.

A mathematical function can be used as an alternative to using an input algorithm to generate the experimental data. The input function must be a function of n and the valid functions are factorial, log, sqrt, floor, ceil, and all the built-in functions and operations of Python. n will be put in a loop similar to the loop in generating data from an augmented algorithm.

A sequence of numbers can also be used as an alternative to generate the experimental data. The sequence of numbers must be separated per line. In this case, the data will just simply copy the sequence of numbers per line, i.e., the number in line 1 will be stored in the 0th index, line 2 will be stored in the 1st index, line 3 will be stored in the 2nd index, and so on.

The user may preemptively stop the generation of experimental data by pressing Ctrl+C which stops the term generating executions and sets the $endTerm$ value to the current size of the $freqCount$ array.

5.1.2 DataProcessor

The Controller retrieves the experimental data and $endTerm$ from the Experimenter and the k value from the GUI and passes it to the DataProcessor. The DataProcessor will automatically do the removal of constants process discussed in Section 5.4 when given these parameters. It will then calculate the e, p, c, e_{nl}, p_{nl}, and c_{nl} approximations for each term of the experimental data with removed constants and store them in their respective arrays (see formulas in Appendix B.1 to B.3). Then, it will check for discontinuities, calculate the approximations around the discontinuities, and choose to whether to keep or ignore the mentioned approximations (see Appendix B.5). After this, it will determine the value of $hasLog$ using the process discussed in Appendix B.4. Lastly, it will validate if the calculated e, p, c, and $hasLog$ values are asymptotically equivalent to the generated data using the process discussed in Section 5.3 and then calculate the plot points for the graph.

5.1.3 GUI

The GUI retrieves the GUI related elements like labels, text fields, buttons, etc. from a glade file. It has an function named *addListener* which allows the controller to specify an action to be done when a particular event occurs in a widget. It has methods for clearing and displaying texts on widgets. It also has a function named *createGraph* which takes in the generated frequency count measurement plot points and e, p, c, and $hasLog$ approximation plot points and uses the Pyplot library in Matplotlib generate and show the graph of these two plots. Pyplot has built-in functions that allows users to move around the graph, zoom in to any part of the graph, and save the graph as an image file.

5.1.4 Controller

The Controller manages the exchange of data between the GUI, Experimenter, and DataProcessor. It is also provides a simplified interface that hides the complex structure of the execution of the developed method.

5.2 Extracting Asymptotic Equivalence from the Measurements

The method uses formulas that will output a function that is asymptotically equivalent to the measured frequency counts of the input algorithm. The accuracy through the use of asymptotic equivalence is one of the huge advantages of the developed method. Despite of the method being fully automated, it still outputs a tight approximation of the asymptotic behavior that is even tighter than Big Theta.

The method covers rates of growth from logarithmic to exponential including constant or no growth. This is because there an infinite number of growth rates and most algorithms fall within this range. And anything beyond logarithmic and exponential will grow too slow to detect and explode too fast to compute for accurately. It is possible to extend the scope further by doing the calculations under different assumptions.

In order to create a general solution for the problem, one must first identify the generalized problem. By allowing each growth within the scope to have its own independent variable (i.e. A for exponential, B for polynomial, C for linear, D for radical, E for logarithmic), a possible general form of the problem can be defined as Expression 5.2.1.

$$A^n * n^B * Cn * \sqrt[D]{n} * log_E(n) \tag{5.2.1}$$

The number of variables in Expression 5.2.1 can be reduced if it is written down as Expression 5.2.2.

$$\begin{gathered} e^n * n^p * cln(n), \text{ where} \\ e = A,\ p = B + 1 + \frac{1}{D},\ c = C * log_E(e_{Euler}) \end{gathered} \tag{5.2.2}$$

Note that e_{Euler} is the Eulers constant and not the variable e in the generalization. These simplifications are made possible because linear and radical growths can be expressed in terms of a polynomial growth with a degree of 1 or a fractional degree. The base of a logarithm can be expressed as a constant divisor through logarithm base conversion (Fradkin, 2013) which in turn can be expressed as a constant factor. Expressions 5.2.1 and 5.2.2 are constructed in such a way that takes on a special property that is - it exhibits all the rates of growths within the scope simultaneously. This is done by taking the product of rates of growths together. The special property is made possible due to the definition of an asymptotic equivalence seen in Definition 3.4.6. The asymptotic equivalence is determined by using division and division being commutative among a series of products assures that the contribution of each rate of growth exists and is independent of the other rates of growths that are present.

This justifies the possibility for e, p, and c to be used to represent rates of growth within the scope. Consider the example $(3n)ln(n)$ where variables can be set to $e = 1$, $p = 1$, $c = 3$ since:

$$1^n n^1 3ln(n) = (3n)ln(n).$$

Another example is $(2^{n+3})ln(n)$ where variables can be set to $e = 2$, $p = 0$, $c = 8$ since:

$$2^n n^0 8ln(n) = (2^{n+3})ln(n).$$

It is also possible that particular term/s does not exhibit a logarithmic growth. In this case, the expression to be used to represent its growth rate is $e^n n^p c$ instead of $e^n n^p cln(n)$. A boolean value $hasLog$ is introduced that can generalize the two cases: (a) $hasLog = False$ represents $e^n n^p c$, while (b) $hasLog = True$ represents $e^n n^p cln(n)$. An example for $hasLog = False$ is given $8n\sqrt[2]{n}$, variables can be set to $e = 1$, $p = 1.5$, $c = 8$ since:

$$1^n n^{1.5} 8 = n * n^{1/2} 8 = 8n\sqrt[2]{n}.$$

The structure of each expression allows one to generalize all possible growth rates within the scope. But, there exists the possibility of combining two functions to form a new one. Consider function A that sorts an array with n^2 complexity and function B that prints the contents of an array with n complexity where n

is the length of the array. There is a function C that sorts an array using A and then prints it using B which has $n^2 + n$ complexity. This can be done with any arbitrary pair of functions and can be repeated in an arbitrary number of times. This implies that an arbitrary function within the scope must have the form in Equation 5.2.3 where a_n is the measurement given an input size of n, m is an arbitrary positive integer which represents the number of terms; $ln(n, hasLog_i)$ returns $ln(n)$ if $hasLog_i = True$, else it returns 1.

$$a_n = \sum_{i=1}^{m}({e_i}^n * n^{p_i} * c_i * ln(n, hasLog_i)) \tag{5.2.3}$$

It is safe to assume that this series in Equation 5.2.3 has a simplest form. If it is not in its simplest form, it can be further simplified by expanding expressions and combining like terms. Once it has been simplified, there are no two terms that have an equal growth rate which implies that there exists a single term that has the largest growth rate. The next step is to rearrange the terms such that the first term contains the largest rate of growth and satisfies Equation 5.2.4. The motivation behind the rearrangement is only to simplify the remaining manipulations and it is possible because addition is commutative.

$$\lim_{n\to\infty} \frac{\sum_{i=1}^{m}({e_i}^n * n^{p_i} * c_i * ln(n, hasLog_i))}{{e_1}^n * n^{p_1} * c_1 * ln(n, hasLog_1} = 0 \tag{5.2.4}$$

By using the definition in Equation 5.2.3 and the property in Equation 5.2.4, it can be shown that ${e_1}^n * n^{p_1} * c_1 * ln(n, hasLog_1$ is asymptotically equivalent to a_n:

$$\lim_{n\to\infty} \frac{a_n}{{e_1}^n * n^{p_1} * c_1 * ln(n, hasLog_1)} \tag{5.2.5}$$

$$= \lim_{n\to\infty} \frac{{e_1}^n * n^{p_1} * c_1 * ln(n, hasLog_1)}{{e_1}^n * n^{p_1} * c_1 * ln(n, hasLog_1)} \tag{5.2.6}$$

$$+ \frac{\sum_{i=2}^{m}({e_i}^n * n^{p_i} * c_i * ln(n, hasLog_i))}{{e_1}^n * n^{p_1} * c_1 * ln(n, hasLog_1)} = \frac{1+0}{1} = 1 \tag{5.2.7}$$

$$\therefore a_n \sim {e_i}^n * n^{p_i} * c_i * ln(n, hasLog_i) \text{ as n} \to \infty \tag{5.2.8}$$

Note that in subsequent discussions, e_1, p_1, c_1, and $hasLog_1$ will be referred to as e, p, c, and $hasLog$ respectively for simplicity. This means that any given algorithm, provided that it is within the scope, a function that is asymptotic equivalent to the behavior of the given algorithm can be described by using three numbers which are e, p, c and one boolean value which is $hasLog$. The algebraic derivation for the formulas for e, p, and c depends on Asymptotic Equivalence 5.2.8. And it could be seen along with a method for determining $hasLog$ in Appendix B.

One can infer the importance of asymptotics as it enables one to ”trade-off” a complicated expression with an arbitrary number of unknowns such as $\sum_{i=1}^{m} {e_i}^n * n^{p_i} * c_i$ with a simpler expression like $e^n * n^p * c$ that is limited to only three unknowns. The simplicity of the latter expression opens up more properties and these include (a) it contains fewer unknowns and fewer number of operations; and (b) allows unknowns to be completely separated to one side of an equation in some cases. For instance, given $a_n = n^p + n^q$ where $p > q$, it is impossible to separate p to one side of the equation. But, by using asymptotics, one could do the following to completely separate p:

$$a_n \sim n^p \text{ as } n \to \infty$$
$$log(a_n) \sim plog(n) \text{ as } n \to \infty$$
$$\frac{log(a_n)}{log(n)} \sim p \text{ as } n \to \infty.$$

5.3 Validation and Convergence Test

The output must satisfy Asymptotic Equivalence 5.2.8 where a_n is the given sequence of frequency counts and $e^n * n^p * c * ln(n, hasLog)$ is the e, p, c, and $hasLog$ values in expression form that were calculated from the derived formulas. This is equivalent to stating that the limit of the sequence of ratios between a_n and $e^n * n^p * c * ln(n, hasLog)$ must converge to 1.

However, limits could not be used to test for convergence since the actual function is not given; only the function values at certain points are given. To develop the convergence test in the method, the concept of convergence and divergence must be associated with something that could be done by using only a finite number of incremental function values. An intuitive way to test for convergence is by checking if the distance between consecutive terms of a sequence F_n are not changing or decreasing, then it converges. This condition is written down in mathematical form in Condition 5.3.1.

$$(\frac{d}{dn}|\Delta F_n| \leq 0) \to (\lim_{n\to\infty} F_n = k) \wedge (\frac{d}{dn}|\Delta F_n| > 0) \to (\lim_{n\to\infty} F_n = \infty) \qquad (5.3.1)$$

In Condition 5.3.1, ΔF_n is defined to be $F_{n+1} - F_n$ and k is some real number constant. Tables 5.1 to 5.3 are the tables of all possible cases could be created by considering all of the operations being considered in this study.

The derivations, evaluations, and simplifications that were made to create the following tables can be seen on Appendix C. Note that linear operation is not included to the table because it is already considered at $n^r, r = 1$. log(-a) is simplified as log(a) since $log(-a) = log(a) + i\pi$; all the imaginary numbers are left out to simplify the tables. Limits that are undefined like -1^n as $n \to \infty$ are considered as divergent because it does not approach a limit and the limit is written down as ∞ even though it does not actually approach ∞. The latter two are unconventional and wrong simplifications, nonetheless they are convenient to use as shorthands in the study and the errors that were introduced by the simplifications are irrelevant.

F_n	Real interval	$\lim_{n\to\infty} F_n$
r^n	$r \leq -1$	∞
	$-1 < r \leq 1$	k
	$r > 1$	∞
n^r	$r \leq 0$	k
	$r > 0$	∞
$log_r(n)$	$r < 0$	∞
	$k = 0$	k
	$k > 0$	∞

Table 5.1: Convergence and Divergence of Functions (Fradkin, 2013)

F_n	ΔF_n	$\frac{d}{dn}\lvert\Delta F_n\rvert$	Real interval	sign$(\frac{d}{dn}\lvert\Delta F_n\rvert)$
r^n	$(r-1)r^n$	$log(r)\lvert(r-1)r^n\rvert$	$r \leq -1$	> 0
			$\underline{-1 < r \leq 1}$	$\underline{\leq 0}$
			$r > 1$	> 0
n^r	$(r)n^{r-1}$	$(r-1)\lvert r\rvert * \frac{1}{n^2}$	$\underline{r \leq 1}$	$\underline{\leq 0}$
			$r > 1$	> 0
$log_r(n)$	$log_r(1+\frac{1}{n})$	$\frac{-1}{\lvert log(r)\rvert} * \frac{1}{n(n+1)}$	$\underline{r < 0}$	$\underline{\leq 0}$
			$r = 0$	≤ 0
			$\underline{r > 0}$	$\underline{\leq 0}$

Table 5.2: Distances Between Consecutive Terms of Functions

The underlined cells in Table 5.2 are the cases where using the ΔF_n does not match the convergence of F_n in Table 5.1; the distance between consecutive terms are not changing nor getting smaller but the limit still goes off to infinity. There are four cases where using ΔF_n to determine the convergence of F_n give the wrong answer, but many of the cases in Table 5.2 do give the right answer. This implies that using ΔF_n is mostly correct but it must slightly be modified to better fit its purpose in Condition 5.3.1.

By observing the $F_n = n^r$ row in Table 5.2, the reason why it failed for $0 < r \leq 1$ is because of the $r-1$ factor after taking the derivative of the difference. This could be fixed by offsetting $r-1$ to r which could be achieved by multiplying n to $(r)n^{r-1}$. As an initial guess, let the modification to ΔF_n be $\Delta' F_n$ which is defined as $\Delta' F_n = n\Delta F_n$. Table 5.3 is constructed by substituting the ΔF_n in Table 5.2 by $\Delta' F_n$.

F_n	$\Delta' F_n$	$\frac{d}{dn}\lvert\Delta' F_n\rvert$	Real interval	sign$(\frac{d}{dn}\lvert\Delta' F_n\rvert)$
r^n	$n(r-1)r^n$	$(log(r) + \frac{1}{n})\lvert(r-1)r^n\rvert * n$	$r \leq -1$ $-1 < r \leq 1$ $r > 1$	> 0 ≤ 0 > 0
n^r	$(nr)n^{r-1}$	$r * \lvert r\rvert n^{r-1}$	$r \leq 0$ $r > 0$	≤ 0 > 0
$log_r(n)$	$nlog_r(1 + \frac{1}{n})$	$\frac{log(1+\frac{1}{n}) - \frac{1}{n+1}}{\lvert log(r)\rvert}$	$r < 0$ $r = 0$ $r > 0$	> 0 ≤ 0 > 0

Table 5.3: Modified Distances Between Consecutive Terms of Functions

As shown in the Table 5.3, the initial guess is correct and all the $\Delta' F_n$ matches the convergence for all the considered F_n. The same intuition about why ΔF_n should work as a convergence test is applicable to $\Delta' F_n$, but the distances between consecutive terms must not only become smaller, but also be smaller by a factor of $\frac{n+1}{n}$. This factor will help detect the divergent nature of the cases where ΔF_n fails to detect divergence while still hold true for all cases that were already correct in the initial ΔF_n tests. In other words, Table 5.3 shows that for all the considered F_n, Condition 5.3.2 must be satisfied.

$$(\frac{d}{dn}|\Delta' F_n| \leq 0) \to (\lim_{n\to\infty} F_n = k) \wedge (\frac{d}{dn}|\Delta' F_n| > 0) \to (\lim_{n\to\infty} F_n = \infty) \qquad (5.3.2)$$

Condition 5.3.2 could be used as an alternative convergence test and it can be implemented by checking if:

$$|(n+1) * (F_{n+2} - F_{n+1})| \leq |n * (F_{n+1} - F_n)|.$$

This condition will be done for all n's from 0 to the specified endTerm. If it is satisfied for the majority of the checks, then it passes the convergence test. The behavior of the formulas used to generate the approximation assures that the ratio of the approximation and the measurements approaches 1. These imply that the sequence of frequency counts is asymptotically equivalent to computed answer.

In cases where the approximations calculated around the discontinuities were chosen, the x, y, and z values around discontinuities (see Appendix B) will be used in place of $n+2$, $n+1$, and n respectively.

5.4 Improving the Accuracy of Results

Rate of growths that are slow or have decreasing derivatives have a problem. They grow so slowly that they require very large values of n to make the contributions made by any added constants negligible. For instance, given the rate of growth of $\ln(n) + 10$, n must be equal to 22,027 just to be roughly equal to the added 10. n has to be much larger than 22,027 to make the added constant 10 negligible.

The theory does hold that as $n \to \infty$ added constants do become negligible. For rate of growths with non-decreasing derivatives, the negligibility of added constants become immediately apparent. But, for rate of growths with decreasing derivatives, the negligibility of added constants require much more terms to be apparent. The problem is generating terms for high values of n takes a lot of time.

Added constants must be removed to overcome this problem. Given a sequence s_n it is possible to transform it to another sequence s'_n which is has no added constants to it. This transformation could be accomplished by defining s'_n to be:

$$s'_n = s_n - s_0 + k$$

where k is the assumed value of the 0th term of the sequence with removed constants; usually, it is assumed to be 0. 0 is the best value for k for the purposes of algorithm analysis since algorithm analysis usually deals with monotonically increasing functions. While it could be set to other values like 1 for more abstract purposes like when identifying an exponentially decreasing function where the 0th term equals 1. It is worth noting that the difference between 0 and 1 is negligible when identifying an exponentially increasing function which is why 0 could still used even if the 0th term of the exponentially increasing function is technically 1. Using this concept will enable the computations to give more accurate results without the need to generate hundreds of thousands of terms.

If every term of a given sequence is equal to each other, then the process of removing constants will be avoided as the constant term is the only term present.

5.5 Scope Based Derivations

It is worth noting that there is nothing special about introducing a factor n to use as a correction to ΔF_n. There are other modifications to ΔF_n that could be used in order to accomplish the same goal. Also, there is nothing special about the e, p, c formulas. All of these are just derived entities that will work under the scope of the study. These derived entities may change when the scope changes, but the process of looking for these entities will remain the same regardless of the scope. In other words, in a different scope, one may have five unknowns to look for instead of four or one may need a more complicated correction to ΔF_n like:

$$\Delta' F_n = (F_{n+1})^n - (F_n)^n.$$

But the way of looking for the four unknowns will still be isolating unnecessary unknowns and substitutions, and looking for $\Delta' F_n$ will still require tables of limits and derivatives of distances between consecutive terms.

Furthermore, it is impossible to create a convergence test of the form seen in Section 5.3 that will universally work for all functions. This is due to the fact that there is no limit to how slow a function could grow and therefore no limit to the amount of modifications one has to make to be able to detect the divergence of these slow growing functions. However, a convergence test of this form will have no problem detecting divergence of fast growing functions since the faster a function grows, the easier it is to detect the increasing distances between consecutive terms.

5.6 Inconsistencies Between the Theoretical Domain and Actual Domain

There is a possibility of having an inconsistent result since the theoretical framework uses concepts like limits approaching infinity and sequences with irrational terms. In reality, our proposed method could only take in a finite number of positive rational terms.

Inconsistencies may arise from having an exceedingly small number of terms. The method works by allowing fastest rate of growth to overwhelm the rest which makes it easier to observe and separate from all the others. This is why it is necessary to maximize the number of terms being observed.

Inconsistencies may also arise from the heuristics being used. The heuristic for choosing between the continuous and discontinuous approximations has a possibility of being wrong. This is difficult to determine whether or not the asymptotic behavior changes after any of the discontinuities. For instance, in the $floor(log(n))$ graph, there are large values of n like 8100 where the function will seem to be linear. This is because the next step which occurs at 8103 is not visible yet and majority of the graph is one straight line starting at x=2980 until x=8100 and thus, it is difficult to distinguish if it is a rounded down logarithmic function or a piecewise function with constant values for each piece. Fortunately, this difficulty is prevalent only in special cases.

Chapter 6

Testing, Results, and Analysis

A prototype was created where the method discussed in the Chapter 5 is implemented and tested on the following algorithms. It is tested on iterative algorithms and recursive algorithms with varying difficulties. The algorithms used for testing are listed in Appendix A as well as the manual calculations (by analysing the source code) for the expected output. Note that for *x in range(0, n)* in Python is a loop from 0 to $n-1$.

6.1 Comparison of the Expected Output and the Algorithm Output

The sample algorithms were chosen due to their complexity of their structure. The tests started from simple algorithms with very few recursions and loops to complicated algorithms with multiple recursive calls and nested loops (some of which depends on the preceding loop variable). The latter tests are on commonly used algorithms in practice like algorithms for searching and sorting. Code for each of the sample algorithms can be seen in Appendix A.

The experiments were run using the 32-bit version of Python 2.7.10. The decimal library was used in order to get high precision values. The nonessential parameters discussed in Subsubsection 4.1.1 are kept to their default values. Table 6.1 summarizes the results from the testing and it has been rounded off to 6 decimal places. The actual algorithm output refers to the output of the implementation of the algorithm analysis developed method. The expected algorithm output refers to the output from apriori analysis methods (see Appendix A). The δ Error refers to the difference between the actual output and the expected output that serves as a metric for accuracy. The e, p, c, and $hasLog$ values refer to the exponential, polynomial, and linear components of a rate of growth as discussed in Section 5.2.

Table 6.1: Expected Output and the Actual Output

Algorithm	**Expected Algorithm Output**	**Actual Algorithm Output**	**δ Error (Actual-Expected)**

Iter1(n)	e = 1
	p = 0
	c = 41
	hasLog = False
		e = 1
		p = 0
		c = 41
		hasLog = False
		(Using 20000 terms)
			δ e = 0
			δ p = 0
			δ c = 0
			δ hasLog = None

Iter2(n)	e = 1
	p = 1
	c = 3
	hasLog = False
		e= 1.000000
		p= 1.000000
		c= 3.000000
		hasLog= False
		(Using 20000 terms)
			δ e = 0
			δ p = 0
			δ c = 0
			δ hasLog = None

Iter3(n)	e = 1
	p = 1
	c = 4
	hasLog = False
		e= 1.000000
		p= 1.000000
		c= 4.000000
		hasLog= False
		(Using 20000 terms)
			δ e = 0
			δ p = 0
			δ c = 0
			δ hasLog = None

Iter4(n)	e = 1
	p = 1
	c = 15
	hasLog = False
		e= 1.000000
		p= 1.000000
		c= 15.000000
		hasLog= False
		(Using 20000 terms)
			δ e = 0
			δ p = 0
			δ c = 0
			δ hasLog = None

Iter5(n)	e = 1
	p = 2
	c = 2
	hasLog = False
		e= 1.000000
		p= 1.999600
		c= 2.006826
		hasLog= False
		(Using 5000 terms)
			δ e = 0
			δ p = -0.0004
			δ c = 0.006826
			δ hasLog = None

Iter6(n)	e = 1
	p = 2
	c = 1
	hasLog = False
		e= 1.000000
		p= 1.999200
		c= 1.006836
		hasLog= False
		(Using 5000 terms)
			δ e = 0
			δ p = -0.0008
			δ c = 0.006836
			δ hasLog = None

Iter7(n)	e = 1
	p = 3
	c = 2
	hasLog = False
		e= 1.000000
		p= 2.999600
		c= 2.006830
		hasLog= False
		(Using 5000 terms)
			δ e = 0
			δ p = -0.0004
			δ c = 0.00683
			δ hasLog = None

Iter8(n)	e = 1
	p = 3
	c = 0.333333
	hasLog = False
		e= 1.000000
		p= 2.999999
		c= 0.333337
		hasLog= False
		(Using 5000 terms)
			δ e = 0
			δ p = -0.000001
			δ c = 0.000004
			δ hasLog = None

IterFibo(n)	e = 1
	p = 1
	c = 4
	hasLog = False
		e = 1.000000
		p = 1.000100
		c = 3.996040
		hasLog = False
		(Using 20000 terms)
			δ e = 0
			δ p = 0.0001
			δ c = -0.00396
			δ hasLog = None

FactLike(n)	e = divergent
	p = divergent
	c = divergent
	hasLog = False
		Converges = False
		(Using 30 terms)
			Correct
			divergence
			detection

Log2(n)	e = 1
	p = 0
	c = 2.885390
	hasLog = True
		e = 1.000000
		p = 0.000035
		c = 2.884487
		hasLog = True
		(Using 20000 terms)
			δ e = 0
			δ p = -0.000035
			δ c = -0.000903
			δ hasLog = None

Fibo(n)	e = 1.618034
	p = 0
	c = 1.788854
	hasLog = False
		e = 1.618034
		p = 0.000000
		c = 1.788854
		hasLog = False
		(Using 50 terms)
			δ e = 0
			δ p = 0
			δ c = 0
			δ hasLog = None

Ack(0, n)	e = 1
	p = 0
	c = 3
	hasLog = False
		e = 1.000000
		p = 0.000000
		c = 3.000000
		hasLog = False
		(Using 20000 terms)
			δ e = 0
			δ p = 0
			δ c = 0
			δ hasLog = None

Ack(1, n)	e = 1
	p = 1
	c = 5
	hasLog = False
		e = 1.000000
		p = 1.000000
		c = 5.000000
		hasLog = False
		(Using 20000 terms)
			δ e = 0
			δ p = 0
			δ c = 0
			δ hasLog = None

Ack(2, n)	e = 1
	p = 2
	c = 5
	hasLog = False
		e = 1.000000
		p = 1.998561
		c = 5.061660
		hasLog = False
		(Using 5000 terms)
			δ e = 0
			δ p = -0.001439
			δ c = 0.06166
			δ hasLog = None

Ack(3, n)	e = 4
	p = 0
	c = 106.666667
	hasLog = False
		e = 4.000000
		p = 0.000000
		c = 106.666666
		hasLog = False
		(Using 40 terms)
			δ e = 0
			δ p = 0
			δ c = -0.000001
			δ hasLog = None

QuinticFibo(n)	e = 1.927562
	p = 0
	c = unknown
	hasLog = False
		e = 1.927562
		p = 0
		c = 0.183563
		hasLog = False
		(Using 40 terms)
			δ e = 0
			δ p = 0
			δ c = unknown
			δ hasLog = None

LinearSearch(n)	e = 1
	p = 1
	c = 2
	hasLog = False
		e = 1.000000
		p = 1.000000
		c = 2.000000
		hasLog = False
		(Using 20000 terms)
			δ e = 0
			δ p = 0
			δ c = 0
			δ hasLog = None

BinarySearch(n)	e = 1
	p = 0
	c = 7.2134751
	hasLog = True
		e = 1.000000
		p = -0.006035
		c = 7.959977
		hasLog = True
		(Using 20000 terms)
			δ e = 0
			δ p = -0.006035
			δ c = 0.746502
			δ hasLog = None

Bubble_sort(n)	e = 1
	p = 2
	c = 1.5
	hasLog = False
		e = 1.000000
		p = 1.999867
		c = 1.501705
		hasLog = False
		(Using 5000 terms)
			δ e = 0
			δ p = -0.000133
			δ c = 0.001705
			δ hasLog = None

Insertion_sort(n)	e = 1
	p = 2
	c = 1.5
	hasLog = False
		e = 1.000000
		p = 1.999600
		c = 1.505114
		hasLog = False
		(Using 5000 terms)
			δ e = 0
			δ p = -0.0004
			δ c = 0.005114
			δ hasLog = None

Merge_sort(n)	e = 1
	p = 1
	c = 11.541560
	hasLog = True
		e = 1.000000
		p = 0.988068
		c = 14.059309
		hasLog = True
		(Using 20000 terms)
			δ e = 0
			δ p = -0.011932
			δ c = 2.517749
			δ hasLog = None

Quick_sort(n)	e = 1
	p = 2
	c = 2
	hasLog = False
		e = 1
		p = 1.998203
		c = 2.030848
		hasLog = False
		(Using 5000 terms)
			δ e = 1
			δ p = -0.001797
			δ c = 0.030848
			δ hasLog = None

Selection_sort(n)	e = 1
	p = 2
	c = 1.5
	hasLog = False
		e = 1.000000
		p = 1.999334
		c = 1.508533
		hasLog = False
		(Using 5000 terms)
			δ e = 0
			δ p = -0.000666
			δ c = 0.008533
			δ hasLog = None

Piecewise(n)	e = 1
	p = 2
	c = 2
	hasLog = False
		e = 1.000000
		p = 2.000004
		c = 1.999942
		hasLog = False
		(Using 5000 terms)
			δ e = 0
			δ p = 0.000004
			δ c = -0.000058
			δ hasLog = None

6.2 Analysis

It is worth noting that almost all error values are small; most of them are all zeroes up to 2 decimal places. Only 2 cases out of 26 ended up having discrepancies larger than 2 decimal places. These two cases are discontinuous in nature which are more likely to converge slowly. Also, the most of the discrepancies are from c approximations and c is the least significant among the three numeric approximations. The actual algorithm output can be further improved by allowing the prototype to generate and process more terms. In addition, the results give direct insight on how the method works - which is by allowing contribution of fastest growing term overwhelm the contributions of the all negligible terms which makes the fastest growing term easier to observe and isolate. This means that if the fastest growing term grows relatively close to the growth of the rest of the terms, it will take more time to make the rest of the terms negligible. This can be seen in the results of $Ack(2, n)$ which follows the time complexity of $5n^2 + 18n + 14$. In this case, the second term is a polynomial of degree 1 while the first term is polynomial of degree 2. This combined with the second term having a larger factor of 18 results in the first term requiring a higher value for the input to make the second term more negligible.

Conversely, if the fastest growing term grows much faster than the rest, then the approximations will approach to the answer quickly. This can be seen in the results of Fibo(n) where the largest growing term is approximately $(1.788854)1.618034^n$ while the rest of the terms is growing at approximately $(1.788854)0.618034^n$ (see Appendix A.2). The first term grows exponentially faster than the rest of the terms resulting in the answer being accurate up to 6 decimal places from studying only the first 50 terms.

Results seen in Table 6.1 are determined without the evaluating the algorithms logical structure. This is useful when doing analysis on complex recursions like $Ack(m, n)$ and $QuinticFibo(n)$; their definitions can be seen in Appendix A. These two functions are notable since they are difficult to evaluate due to their complexity. $Ack(m, n)$ is the implementation of the Ackermann function which is a non-primitive recursive function (Ackermann, 1928), while $QuinticFibo(n)$ is a linear recurrence relation with a corresponding quintic (5th degree) characteristic equation. The complexity of the $Ack(m, n)$ depends on its first parameter m and the method succeeded in extracting an accurate approximation of the asymptotic equivalence for all values of m that are within the defined scope. The method is also successful in approximating the known e and p values for the $QuinticFibo(n)$ behavior. Note that the expected e value cannot be solved analytically (i.e., has no exact form). Its value was approximated using a graph. The actual algorithm does not require any analytic solutions or graphs for calculating e value and yet it produced an accurate result. The expected c value, on the other hand, was not computed for due to the difficulty in working with complex numbers and errors will be encountered when dealing with mere approximations. Despite of these problems in computing for the expected c value, the developed method did not encounter any problems in computing for the c value using the formulas. The accuracy seen from all the experiments conducted serves as a sufficient justification to conclude that the method is effective in determining the algorithms asymptotic behavior.

6.3 Conclusion and Recommendation

The prototype is successful in parsing and augmenting Python programs with the exception of a few statements. Fortunately, these exceptions could be translated into some other form that could be properly preprocessed. The study is successful in developing a theoretical basis for the method. This includes the arguments and justifications for the generalized form of the scope and the use of asymptotic equivalences. This also includes the derivations for the e, p, c formulas and the proofs that support claims made in the theory. The study is also successful in creating heuristics that detect logarithmic growth and discontinuities. However, it is not completely successful in the convergence heuristic because divergences slower than logarithmic divergence produce false positive results.

The prototype has been tested with a comprehensive collection of algorithms. The collection includes algorithms containing no loops, single loops, double loops, triple loops, if-else statements, and recursions which may be nested within each other. All except one of the resulting approximations are very close to the corresponding manual apriori calculations. The exception with the farthest result is generated from the $Merge\_sort(n)$ algorithm. Its slow pace in converging to the correct answer is due to the fact that its largest term $(11.5nlog(n))$ and second largest term $(10n)$ are growing relatively close to each other. This means that a larger value of n to turn the other terms negligible relative to the largest growing term. Although the rate at which the approximation converges to the exact answer vary, it is still true that the approximations will eventually approach to the exact answer. Generally, one could get a closer approximation to the exact answer by generating more terms, and every other term could always be computed for, assuming that one has sufficient computing resources. It is interesting to note that the research study on the analysis of algorithms using frequency counts has provided an avenue for an in-depth understanding of algorithm performance through numeric means. This consequently leads to a sound basis of determining algorithms' asymptotic behavior. More over, the developed method shows that one could extract asymptotic behaviors without any analysis on the input algorithm itself. However, the method for now only works on single parameter functions.

Despite the novelty of this approach, the current method requires some improvements. One of the improvements is the ability to analyze the performance of algorithms with multiple parameters. The process of generating the series of performance metrics can also be improved by considering other performance metrics such as Operation count or CPU cycle count.

Future research may be focused on extending the domain of the input sequence. This work may be extended for rates of growth not covered within the scope such as iterated exponential and iterated logarithmic growths. It is also interesting to extend the domain to consider sequences containing negative terms or terms with alternating signs. This implies that there is some underlying complex behavior of the sequence. This is because algorithms, despite running within real positive time, may contain negative or imaginary components. Although there is no concept of imaginary or negative time in running time, it is necessary to consider in the calculations since it will have an effect on the real positive part of an algorithm's running time.

Extending the output to deal with more than one term is also being considered, with a conjecture that a higher degree of accuracy can be foreseen. For instance, one may use the method recursively on a given input sequence, then subtracting the input sequence with the output of the method. This is done repeatedly until all the terms in the input sequence become equal to each other. In theory, one can find the largest term in a given rate of growth; then, subtract it to every term; then, find the next largest term, and so on until all of the terms have been discovered. New heuristics and the extension to the negative domain must be developed to automate this process.

References

Abel, N. H. (1824). *Mmoire sur les quations algbriques, ou l'on dmontre l'impossibilit de la rsolution de l'quation gnrale du cinquime degr.*

Ackermann, W. (1928). Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, *99*, 118-133.

Bockmayr, et al. (2006). *Discrete math for bioinformatics: Types of algorithms* (Unpublished master's thesis). MI, Berlin.

Cormen, T., et al. (2001). *Introduction to algorithms.* MIT Press.

Donail, D. A. M. (2006). *On the scalability of molecular computational solutions to np problems* (Unpublished master's thesis). Trinity College, University of Dublin.

Fradkin, L. (2013). *Elementary algebra and calculus.* Bookboon.

Gossett, E. (2009). *Discrete mathematics with proof.* Wiley Publishing.

Greene, D., et al. (1982). *Mathematics for the analysis of algorithms (second ed.).* Birkhuser.

Hantler, S. L., et al. (1976). *An introduction to proving the correctness of programs.* Computer Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

Hartmanis, J., et al. (1965). *On the computational complexity of algorithms.* American Mathematical Society.

Knuth, D. (1976). Mathematics and computer science: Coping with finiteness. *Science*, *194*(17), 1235-1242.

McConnell, J. (2008). *Analysis of algorithms.* Jones and Bartlett Publishers Inc.

McGeoch, C., et al. (2002). Using finite experiments to study asymptotic performance. In *From algorithm design to robust and efficient software.* Springer Berlin Heidelberg.

Montesinos, V., et al. (2015). *An introduction to modern analysis.* Springer.

Pai, G. (2008). *Data structures and algorithms:concepts, techniques and applica-*

tions. Tata McGraw-Hill Education Pvt. Ltd.

Profiling of Algorithms. (n.d.). Retrieved from https://www8.cs.umu.se/kurser/TDBC91/H99/Slides/profiling.pdf (Accessed on August 2015)

Proof of the Master Method. (n.d.). Retrieved from http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec19-master/mm-proof.pdf (Accessed on October 2015)

Rabinovich, S., et al. (1996). Solving nonlinear recursions. *Journal of Mathematical Physics*, *37*(11), 5828-5836.

Sedgewick, R., et al. (2013). *Introduction to the analysis of algorithms*. Addison-Wesley Professional.

Sudan, G. (1927). Sur le nombre transfini. *Bulletin Mathgmatique de la Societe Roumaine des Sciences*, 11-30.

Sutner, K. (n.d.). *Space complexity.* Retrieved from http://www.cs.cmu.edu/~flac/pdf/PSPACE-6up.pdf (Accessed on August 2015)

Wolfram alpha. (n.d.). Retrieved from http://www.wolframalpha.com/input/?i=x\%5E5-2x\%5E3-2x\%5E2-2x\%5E1-1\%3D0 (Accessed on February 2015)

Appendix A

Manual Calculations for the Frequency Counts of the Sample Algorithms

A.1 Analysis of Simple Functions

This section tackles the analysis of the multiple iterative functions and derives its the expected e, p, c, and hasLog values. To simplify the calculations, EX_x denotes the added frequency count contributed by the exiting condition check from loop x, IT_x denotes the added frequency count contributed by incrementing a variable and condition checking within loop x. Note that for x in range(0, n) in Python is a loop from 0 to n-1 and e, p, c, and hasLog can be obtained by observing the largest growing term in the equation.

Listing A.1: Iter1

```
def f(n): #41
    for i in range(0 ,20):
        print i
```

$F_{Iter1} = EX_i + 20 * (IT_i + 1) = 1 + 20 * 2 = 41$

Listing A.2: Iter2

```
def f(n): #3n+1
```

```
    for i in range(0 ,n):
        x=i+1
        print x
```

$F_{Iter2} = EX_i + n * (IT_i + 1 + 1) = 1 + n * 3 = 3n + 1$

Listing A.3: Iter3

```
def f(n): #4n+1
    for i in range(0 ,2*n):
        add_i_n=i+n
```

$F_{Iter3} = EX_i + (2 * n) * (IT_i + 1) = 1 + 2n * 2 = 4n + 1$

Listing A.4: Iter4

```
def f(n): #15n+1
    for i in range(0 ,5*n):
        a=i+n
        b=a*n
```

$F_{Iter4} = EX_i + (5 * n) * (IT_i + 1 + 1) = 1 + 5n * 3 = 15n + 1$

Listing A.5: Iter5

```
def f(n): #2n^2+2n+1
    for i in range(0 ,n):
        for j in range(0, n):
            mult = i*j
```

$F_{Iter5} = EX_i + n * (IT_i + EX_j + n * (IT_j + 1)) = 1 + n(1 + 1 + n * 2)$
$= 1 + n + n + 2n^2 = 2n^2 + 2n + 1$

Listing A.6: Iter6

```
def f(n):#n^2+2n+1
    for i in range(0 ,n):
        sum_to_i=0
        for j in range(0, i):
            sum_to_i+=i
```

$F_{Iter6} = EX_i + n*(IT_i + 1 + EX_j) + \sum_{i=0}^{n-1}(\sum_{j=0}^{i-1}(IT_j + 1))$
$= 1 + n*(1+1+1) + \sum_{i=0}^{n-1}(\sum_{j=0}^{i-1}(1+1)) = 1 + 3n + \sum_{i=0}^{n-1}(2i) = 1 + 3n + n^2 - n$
$= n^2 + 2n + 1$

Listing A.7: Iter7

```
def f(n): #2n^3+2n^2+2n+1
    for i in range(0 ,n):
        for j in range(0, n):
            for k in range(0, n):
                tuple = "(%d, %d, %d)"%(i, j, k)
```

$F_{Iter7} = EX_i + n*(IT_i + EX_j + n*(IT_j + EX_k + n*(IT_k + 1)))$
$= 1 + n*(2 + n*(2 + n*(2))) = 1 + n*(2 + n*(2+2n)) = 1 + n*(2 + 2n + 2n^2))$
$= 2n^3 + 2n^2 + 2n + 1$

Listing A.8: Iter8

```
def f(n): #n^3/3+5n/3+2
    big_nested_sum=0
    for i in range(0 ,n):
        for j in range(0, i):
            for k in range(0, j):
                big_nested_sum+=k
```

$F_{Iter8} = 1 + EX_i + \sum_{i=0}^{n-1}(IT_i + EX_j + \sum_{j=0}^{i-1}(IT_j + EX_k + \sum_{k=0}^{j-1}(IT_k + 1)))$
$= 2 + \sum_{i=0}^{n-1}(2 + \sum_{j=0}^{i-1}(2 + \sum_{k=0}^{j-1}(2))) = 2 + \sum_{i=0}^{n-1}(2 + \sum_{j=0}^{i-1}(2+2j))$
$= 2 + \sum_{i=0}^{n-1}(2 + 2i + \frac{2i(i-1)}{2}) = 2 + \sum_{i=0}^{n-1}(2 + i + i^2)$
$= 2 + 2n + \frac{n*(n-1)}{2} + \frac{n*(n-1)*(2n-1)}{6} = \frac{n^3}{3} + \frac{5n}{3} + 2$

Listing A.9: IterFibo

```
def f(n):#4n
    nm1=1
    nm2=0
    for i in range (0,n-1):
        temp=nm1+nm2
        nm2=nm1
        nm1=temp
    return nm1
```

$F_{IterFibo} = 1 + 1 + EX_i + \sum_{i=0}^{n-2}(IT_i + 1 + 1 + 1) + 1 = 4 + (n-1)(4) = 4n$

The following two algorithms are recursive. The Iterative method is used to generate the expected time complexity in terms of Frequency count.

Listing A.10: FactLike

```
def f(n):#~n^n+n!
    if n==0:
        return 1
    else:
        for i in range(0, n):
            f(n-1)
```

$F_{FactLike}(0) = 1 + 1$
$F_{FactLike}(n) = 1 + EX_i + n(IT_i + 1 + F_{FactLike}(n-1)) = 2 + 2n + nF_{FactLike}(n-1)$
$\sim 2n + nF_{FactLike}(n-1) = 2n + n(2(n-1) + (n-1)F_{FactLike}(n-2))$
$\sim 2n^2 + n(n-1)F_{FactLikeLike}(n-2) \sim ... \sim 2n^i + i!F_{FactLike}(n-i)$
To reduce the recursion to the base case, i must satisfy:
$n - i = 0 \rightarrow i = n$
By setting the value of i to n:
$F_{FactLike}(n) \sim 2n^n + 2n!$

Note that n^n is not part of the scope which means that the e, p, and c formulas must not be applicable for FactLike.

Listing A.11: Log2

```
def f(n):#log(n)/log(2) (log base 2 of n)
    if(n<=2):
        return 1
    else:
        return f(n/2.0)
```

$F_{Log2}(2) = 1 + 1$
$F_{Log2}(n) = 1 + 1 + F_{Log2}(n/2) = 2 + F_{Log2}(n/2)$
$= 4 + F_{Log2}(n/4) = 6 + F_{Log2}(n/8) = ... = 2i + F_{Log2}(n/2^i)$
To reduce the recursion to the base case, i must satisfy:
$n/2^i = 2 \rightarrow n = 2^{i+1} \rightarrow log(n) = (i+1)log(2) \rightarrow log_2(n) = i+1 \rightarrow i = log_2(n)-1$

By setting the value of i to $log_2(n) - 1$:

$$F_{Log2}(n) = 2(log_2(n) - 1) + 2 = 2log_2(n) = \frac{2log(n)}{log(2)} \approx 2.885390log(n)$$

A.2 Analysis of the Recursive Fibonacci Function

This section tackles the analysis of the Fibonacci function implemented using recursions and derives its the expected e, p, c, and hasLog values.

The Fibonacci or Fibo function is defined to be:

$$Fibo(n) = \begin{cases} 1 & \text{if } n <= 2 \\ Fibo(n-1) + Fibo(n-2) & \end{cases}$$

From the definition above, the recurrence of $Fibo$ called F_{Fibo} will behave as follows

$$F_{Fibo}(n) = \begin{cases} 2 & \text{if } n <= 2 \\ 2 + F_{Fibo}(n-1) + F_{Fibo}(n-2) & \end{cases}$$

The 2 in the base case comes from the first condition checking and the returning of 1. The recursive/default case comes from the checking of the first condition and returning a value and evaluating the 2 recursive calls.

The closed form expression for the F_{Fibo} can be solved through the Linear Recurrence Equation (Gossett, 2009). First, isolate all the recursive terms on one side of the equation.

$$F_{Fibo}(n) - F_{Fibo}(n-1) - F_{Fibo}(n-2) = 2 \tag{A.2.1}$$

Then, turn it into a Homogeneous Recurrence Equation by doing the following:

$$\begin{array}{r} [F_{Fibo}(n+1) - F_{Fibo}(n) - F_{Fibo}(n-1) = 2] \\ -[F_{Fibo}(n) - F_{Fibo}(n-1) - F_{Fibo}(n-2) = 2] \\ \hline F_{Fibo}(n+1) - 2F_{Fibo}(n) + F_{Fibo}(n-2) = 0 \end{array} \tag{A.2.2}$$

The corresponding characteristic equation for this is:

$$F_{Fibo}(n) = c_1{r_1}^n + c_2{r_2}^n + c_3{r_3}^n \tag{A.2.3}$$

Where c's are real number constants and r's are the roots of this equation:

$$r^3 - 2r^2 + 1 = 0 \tag{A.2.4}$$

The r's can be solved by using any method that finds roots or factors of polynomials:

$$r_1 = \frac{1+\sqrt{5}}{2}, r_2 = \frac{1-\sqrt{5}}{2}, r_3 = 1 \tag{A.2.5}$$

The roots for the Homogeneous Solution can be determined by inspecting the recursive part of the original recurrence equation:

$$\begin{aligned} F_{Fibo}(n) &= F_{Fibo}(n-1) + F_{Fibo}(n-2) \\ F_{Fibo}(n) - F_{Fibo}(n-1) - F_{Fibo}(n-2) &= 0 \end{aligned} \tag{A.2.6}$$

And, by solving for the characteristic equation and roots of this equation:

$${r_{homo}}^2 - r_{homo} - 1 = 0 \tag{A.2.7}$$

The r_{homo}'s can be solved by using any method that finds roots or factors of polynomials:

$$r_{homo1} = \frac{1+\sqrt{5}}{2}, r_{homo2} = \frac{1-\sqrt{5}}{2} \tag{A.2.8}$$

It is evident that r_1 and r_2 are the roots for the Homogeneous Solution. Therefore, the following is the Homogeneous Solution for the original problem.

$$homo\_F_{Fibo}(n) = c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n \tag{A.2.9}$$

Knowing the Homogeneous Solution, one can conclude that the Particular Solution is:

$$part\_F_{Fibo}(n) = c_3 * 1^n \tag{A.2.10}$$

c_3 can be solved for by using the original recurrence relation.

$$\begin{aligned} part\_F_{Fibo}(n) = part\_F_{Fibo}(n-1) + part\_F_{Fibo}(n-2) + 2 \\ c_3 = c_3 + c_3 + 2 \\ c_3 = -2 \end{aligned} \tag{A.2.11}$$

It is now possible to solve for the General Equation by plugging all the r's and c_3.

$$F_{Fibo}(n) = c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n - 2 \tag{A.2.12}$$

Solve for c_1 and c_2 by creating two equations using the two base cases.

$$F_{Fibo}(1) = 2 = c_1(\frac{1+\sqrt{5}}{2}) + c_2(\frac{1-\sqrt{5}}{2}) - 2 \tag{A.2.13}$$

$$F_{Fibo}(2) = 2 = c_1(\frac{1+\sqrt{5}}{2})^2 + c_2(\frac{1-\sqrt{5}}{2})^2 - 2 \tag{A.2.14}$$

By using any method that solves Systems of Equations with two unknowns:

$$c_1 = \frac{4}{\sqrt{5}}, c_2 = \frac{-4}{\sqrt{5}} \tag{A.2.15}$$

The explicit formula for the Frequency Count of Fibo:

$$F_{Fibo}(n) = \frac{4}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{4}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n - 2 \tag{A.2.16}$$

By inspecting the explicit formula, it is clear that $\frac{4}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n$ is the term with largest rate of growth. Therefore, it is asymptotic to $F_{Fibo}(n)$ as $n \to \infty$. The expected e, p, c, and hasLog values (rounded off to 12 decimal placed) would be the following:
$e_{Fibo} = \frac{1+\sqrt{5}}{2} \approx 1.618033988750$
$p_{Fibo} = 0$
$c_{Fibo} = \frac{4}{\sqrt{5}} \approx 1.788854382000$
$hasLog_{Fibo} = False$

A.3 Analysis of the Ackermann Function

This section tackles the analysis of the Ackermann function and derives its the expected e, p, c, and hasLog values. The second parameter 'n' will serve as the independent variable in this analysis of the Ackermann Function. This analysis will only consider integer values from 0 to 3 for the first parameter 'm'. This is because when 'm>3', the function grows beyond exponential growth which is not within the scope (see Appendix A.3.1).

The Ackermann Function is defined to be:

$$Ack(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ Ack(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ Ack(m-1, Ack(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

From the definition above, the recurrence of Ack called F_{Ack} will behave as follows:

$$F_{Ack}(m,n) = \begin{cases} 2 & \text{if } m=0 \\ 3+F_{Ack}(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ 3+F_{Ack}(m-1, Ack(m,n-1)) + F_{Ack}(m,n-1) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

The 2 in the case 'm=0' comes from the checking the first condition and returning a value. The added 3 in the case 'm>0 and n=0' comes from the checking the first and second conditions and returning a value. The added $3 + F_{Ack}(m, n-1)$ in the default case comes from the checking of the first and second conditions and returning a value and evaluating the enclosed recursive call.

To understand how F_{Ack} grows, it is necessary to understand how Ack grows.
$Ack(0,n) = n+1$

$Ack(1,n) = Ack(0, Ack(1,n-1)) = Ack(1,n-1)+1$
$= Ack(0, Ack(1,n-2))+1 = Ack(1,n-2)+2$
$= Ack(0, Ack(1,n-3))+2 = Ack(1,n-3)+3$
$= Ack(1,n-n)+n = Ack(1,0)+n$
$= Ack(0,1)+n = n+2$
$Ack(1,n) = n+2$

$Ack(2,n) = Ack(1, Ack(2,n-1)) = Ack(2,n-1)+2$
$= Ack(1, Ack(2,n-2))+2 = Ack(2,n-2)+4$
$= Ack(1, Ack(2,n-3))+4 = Ack(2,n-3)+6$
$= Ack(2,n-n)+2n = Ack(2,0)+2n$
$= Ack(1,1)+2n = 3+2n$
$Ack(2,n) = 2n+3$

$Ack(3,n) = Ack(2, Ack(3, n-1)) = 2Ack(3, n-1) + 3$
$= 2Ack(2, Ack(3, n-2) + 3 = 2(2Ack(3, n-2) + 3) + 3 = 4Ack(3, n-2) + 9$
$= 4Ack(2, Ack(3, n-3)) + 9 = 4(2Ack(3, n-3) + 3) + 9 = 8Ack(3, n-3) + 21$
$= 2^n(3, n-n) + 3 * (\sum_{i=0}^{n-1} 2^i) = 2^n * Ack(3,0) + 3 * \frac{2^n - 1}{2-1}$
$= 2^n * Ack(3,0) + 3 * 2^n - 3 = 2^n * Ack(2,1) + 3 * 2^n - 3$
$= 2^n * 5 + 3 * 2^n - 3 = 8 * 2^n - 3$
$Ack(3,n) = 8 * 2^n - 3$

The values for $Ack(m,n)$ where m is an integer value from 0 to 3 will be used in solving for $F_{Ack}(m,n)$. Note that since these functions have two parameters, they cannot be called directly from the prototype. There must be an external one parameter function that calls Ack which introduces an additional +1 to the experimental values whenever Ack is tested.

$F_{Ack}(0,n) = 2$

$F_{Ack}(1,n) = 3 + F_{Ack}(0, Ack(1, n-1)) + F_{Ack}(1, n-1)$
$= 6 + F_{Ack}(0, Ack(1, n-1)) + F_{Ack}(0, Ack(1, n-2)) + F_{Ack}(1, n-2)$
$= 9 + F_{Ack}(0, Ack(1, n-1)) + F_{Ack}(0, Ack(1, n-2)) + F_{Ack}(0, Ack(1, n-3)) + F_{Ack}(1, n-3)$
$= 3n + \sum_{i=1}^{n}(F_{Ack}(0, Ack(1, n-i))) + F_{Ack}(1,0)$
$= 3n + \sum_{i=1}^{n}(F_{Ack}(0, Ack(1, n-i))) + 3 + F_{Ack}(0,0)$
$= 3n + 5 + \sum_{i=1}^{n}(F_{Ack}(0, Ack(1, n-i)))$
$= 3n + 5 + \sum_{i=1}^{n}(F_{Ack}(0, n-i+2))$
$= 3n + 5 + \sum_{i=1}^{n}(2)$
$= 3n + 5 + 2n$
$F_{Ack}(1,n) = 5n + 5$

$F_{Ack}(2,n) = 3 + F_{Ack}(1, Ack(2, n-1)) + F_{Ack}(2, n-1)$
$= 6 + F_{Ack}(1, Ack(2, n-1)) + F_{Ack}(1, Ack(2, n-2)) + F_{Ack}(2, n-2)$
$= 9 + F_{Ack}(1, Ack(2, n-1)) + F_{Ack}(1, Ack(2, n-2)) + F_{Ack}(1, Ack(2, n-3)) + F_{Ack}(2, n-3)$
$= 3n + \sum_{i=1}^{n}(F_{Ack}(1, Ack(2, n-i))) + F_{Ack}(2,0)$
$= 3n + \sum_{i=1}^{n}(F_{Ack}(1, Ack(2, n-i))) + 3 + F_{Ack}(1,1)$
$= 3n + 13 + \sum_{i=1}^{n}(F_{Ack}(1, Ack(2, n-i))) = 3n + 13 + \sum_{i=1}^{n}(F_{Ack}(1, 2n-2i+3))$
$= 3n + 13 + 5\sum_{i=1}^{n}(2n - 2i + 4)$
$= 3n + 13 + 5(2n^2 + 4 * n - n^2 - n) = 3n + 13 + 5n^2 + 15n$
$F_{Ack}(2,n) = 5n^2 + 18n + 13$

$F_{Ack}(3,n) = 3 + F_{Ack}(2, Ack(3, n-1)) + F_{Ack}(3, n-1)$
$= 6 + F_{Ack}(2, Ack(3, n-1)) + F_{Ack}(2, Ack(3, n-2)) + F_{Ack}(3, n-2)$
$= 9 + F_{Ack}(2, Ack(3, n-1)) + F_{Ack}(2, Ack(3, n-2)) + F_{Ack}(2, Ack(3, n-3)) + F_{Ack}(3, n-3)$
$= 3n + \sum_{i=1}^{n}(F_{Ack}(2, Ack(3, n-i))) + F_{Ack}(3, 0)$
$= 3n + \sum_{i=1}^{n}(F_{Ack}(2, Ack(3, n-i))) + 3 + F_{Ack}(2, 1)$
$= 3n + 39 + \sum_{i=1}^{n}(F_{Ack}(2, Ack(3, n-i)))$
$= 3n + 39 + \sum_{i=1}^{n}(F_{Ack}(2, 8 * 2^{n-i} - 3))$
$= 3n + 39 + \sum_{i=1}^{n}(5(8 * 2^{n-i} - 3)^2 + 18(8 * 2^{n-i} - 3) + 13)$
$= 3n + 39 + \sum_{i=1}^{n}(5(64 * 4^{n-i} - 48 * 2^{n-i} + 9) + 18(8 * 2^{n-i} - 3) + 13)$
$= 3n + 39 + \sum_{i=1}^{n}(320 * 4^{n-1} - 240 * 2^{n-i} + 45 + 144 * 2^{n-i} - 54 + 13)$
$= 3n + 39 + 4n + \sum_{i=1}^{n}(320 * 4^{n-1} - 96 * 2^{n-i})$
$= 7n + 39 + 320 * \dfrac{4^n - 1}{4 - 1} - 96 * \dfrac{2^n - 1}{2 - 1}$
$= 7n + 39 + \dfrac{320}{3} * (4^n - 1) - 96 * (2^n - 1)$
$= 7n + 39 + \dfrac{320}{3} * 4^n - \dfrac{320}{3} - 96 * 2^n + 96$
$= \dfrac{320}{3} * 4^n - 96 * 2^n + 7n + 39 - \dfrac{320}{3} + 96$
$F_{Ack}(3,n) = \dfrac{320}{3} * 4^n - 96 * 2^n + 7n + \dfrac{85}{3}$

By inspecting the largest terms from all the $F_{Ack}(m,n)$'s the expected e, p, c, and hasLog values are:
$e_{Ack0} = 1$, $p_{Ack0} = 0$, $c_{Ack0} = 2$, $hasLog_{Ack0} = False$

$e_{Ack1} = 1$, $p_{Ack1} = 1$, $c_{Ack1} = 5$, $hasLog_{Ack1} = False$

$e_{Ack2} = 1$, $p_{Ack2} = 2$, $c_{Ack2} = 5$, $hasLog_{Ack2} = False$

$e_{Ack3} = 4$, $p_{Ack3} = 0$, $c_{Ack3} = \dfrac{320}{3} = 106.\bar{6}$, $hasLog_{Ack3} = False$

A.3.1 Behavior of Ackermann(4, n)

Understanding the behavior of Ack(4, n) is necessary for understanding the asymptotic behavior of $F_{Ack}(4, n)$. The following is the derivation for the explicit formula of Ack(4, n); the derived explicit form of Ack(3, n) is $8 * 2^n - 3$ or $2^{n+3} - 3$ as shown in A.3.

$$Ack(4,n) = Ack(3, Ack(4, n-1)) = 2^{Ack(4,n-1)+3} - 3$$
$$= 2^{Ack(3,Ack(4,n-2))+3} - 3 = 2^{2^{Ack(4,n-2)+3}-3+3} - 3$$
$$= 2^{2^{Ack(3,Ack(4,n-3))+3}} - 3 = 2^{2^{2^{Ack(4,n-3)+3}-3+3}} - 3$$
$$= 2^{2^{2^{\cdot^{\cdot^{Ack(4,0)+3}}}}} - 3 = 2^{2^{2^{\cdot^{\cdot^{8-3+3}}}}} - 3$$
$$= 2^{2^{2^{\cdot^{\cdot^{2^{2^{2}}}}}}} - 3 = 2 \uparrow (n+3) - 3$$

The $\uparrow$ denotes Knuth's up-arrow notation introduced in Knuth (1976) where $a \uparrow b$ is equal to a exponentiated with itself b many times forming a right-associative tower of exponents.

Using the same F_{Ack} expansions in the derivation of the explicit forms of various F_{Ack}'s in A.3. $F_{Ack}(4,n)$ will have the following form:

$$F_{Ack}(4,n) = 3n + \sum_{i=1}^{n}(F_{Ack}(3, Ack(4, n-i))) + F_{Ack}(4,0)$$
$$= 3n + 273 + \sum_{i=1}^{n}(F_{Ack}(3, Ack(4, n-i)))$$
$$= 3n + 273 + \sum_{i=1}^{n}(\frac{320}{3} * 4^{Ack(4,n-i)} - 96 * 2^{Ack(4,n-i)} + 7Ack(4, n-i) + \frac{85}{3}))$$

By observing the third term for each iteration of the summation ($7Ack(4, n\text{-}i)$), it is clear that a formula for $2 + 2^2 + 2^{2^2} + ...$ or $\sum_{i=1}^{n}(2 \uparrow i)$ is needed to identify the explicit form of $F_{Ack}(4,n)$. This also implies that the Ack(m, n) for any value of m greater than or equal to 4 are not included within the scope.

A.4 Analysis of QuinticFibo

This section tackles the analysis of the QuinticFibo function (see Listing A.12) and derives its the expected e, p, and hasLog values. The expected values for c is not solved for due to the difficulty in solving for it.

Listing A.12: QuinticFibo

```
def f(n):
    if(n<=5):
        return 1
    else:
        return f(n−1)+f(n−2)+f(n−3)+f(n−4)
```

The recurrence relation for QuinticFibo has the form:

$$F_{QF}(n) = F_{QF}(n-1) + F_{QF}(n-2) + F_{QF}(n-3) + F_{QF}(n-4) + 2 \tag{A.4.1}$$

The closed form expression for the F_{QF} can be solved through the Linear Recurrence Equation. First, isolate all the recursive terms on one side of the equation.

$$F_{QF}(n) - F_{QF}(n-1) - F_{QF}(n-2) - F_{QF}(n-3) - F_{QF}(n-4) = 2 \tag{A.4.2}$$

Then, turn it into a Homogeneous Recurrence Equation by doing the following:

$$\begin{array}{r} [F_{QF}(n+1) - F_{QF}(n) - F_{QF}(n-1) - F_{QF}(n-2) - F_{QF}(n-3) = 2] \\ -[F_{QF}(n) - F_{QF}(n-1) - F_{QF}(n-2) - F_{QF}(n-3) - F_{QF}(n-4) = 2] \\ \hline \end{array}$$

$$F_{QF}(n+1) - 2F_{QF}(n-1) - 2F_{QF}(n-2) - 2F_{QF}(n-3) - 2F_{QF}(n-4) = 0 \tag{A.4.3}$$

The corresponding characteristic equation for this is:

$$F_{Fibo}(n) = c_1{r_1}^n + c_2{r_2}^n + c_3{r_3}^n + c_4{r_4}^n + c_5{r_5}^n \tag{A.4.4}$$

Where c's are real number constants and r's are the roots of this equation:

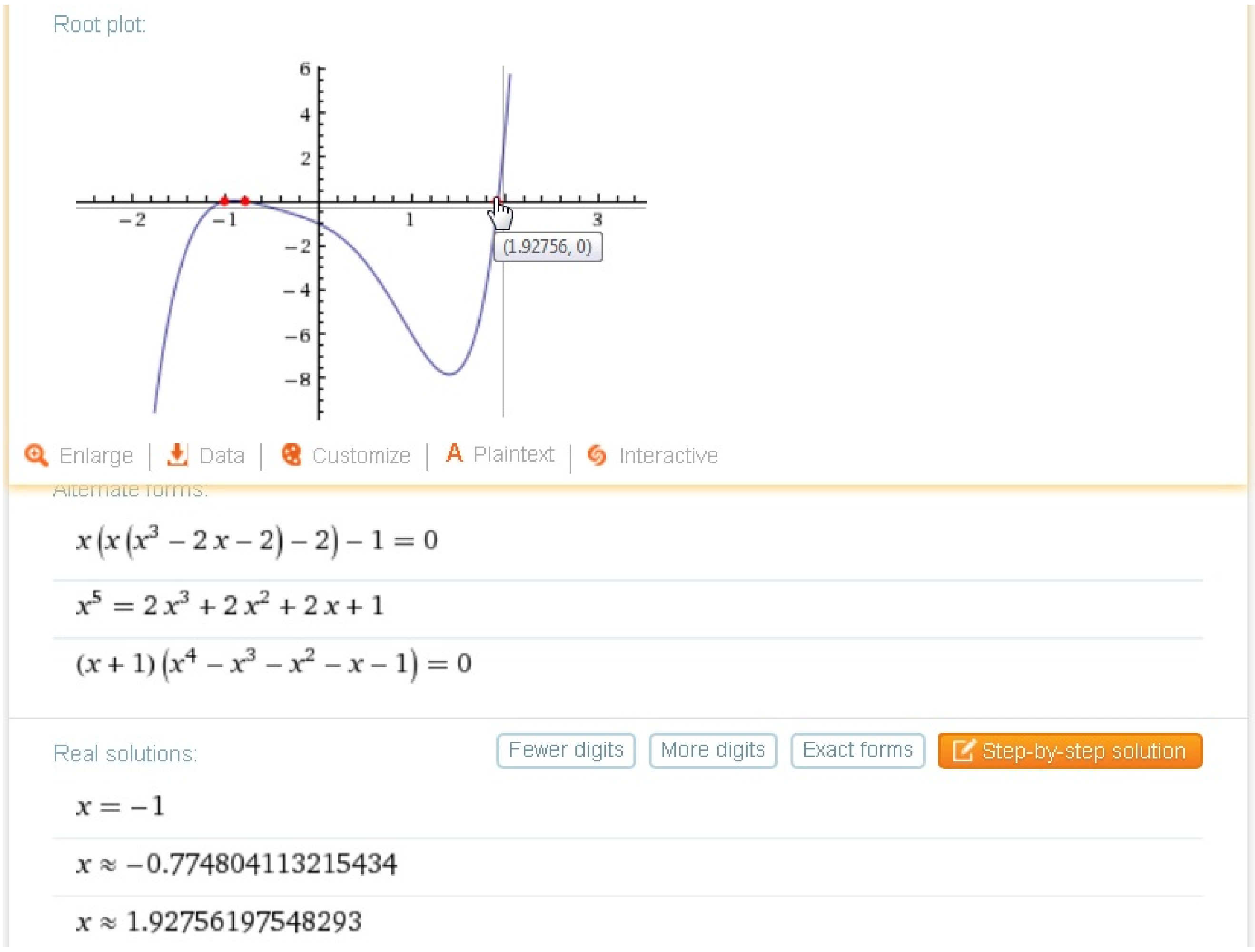
$$r^5 - 2r^3 - 2r^2 - 2r - 1 = 0 \tag{A.4.5}$$

Since the characteristic equation is a quintic equation, there is no general solution to solve for the r's. A graph will be used to approximate values of r. The following function has been graphed and equated to zero using Wolfram Alpha (*Wolfram Alpha*, n.d.):

$$x^5 - 2x^3 - 2x^2 - 2x^1 - 1 = 0 \tag{A.4.6}$$

The Figure A.1 is the graph that shows that the largest root is approximately 1.927652 (rounded off to 6 decimal places). This implies that the largest growing term must have the approximate form $(c_1)1.927562^n$.

With these information, the expected values are 1.927562 for e, 0 for p, and False for hasLog. The expected value for c is not computed because the exact forms are not given and two out of the five of the roots have imaginary parts. Another problem is that the errors will stack up when computing for c. This is because all the terms and coefficients in the succeeding equations are all approximations. The errors will grow larger and larger as the number of manipulations that are done on the equations increases in the pursuit of solving for c.


Root plot:

Alternate forms:

$x(x(x^3 - 2x - 2) - 2) - 1 = 0$

$x^5 = 2x^3 + 2x^2 + 2x + 1$

$(x + 1)(x^4 - x^3 - x^2 - x - 1) = 0$

Real solutions:

$x = -1$

$x \approx -0.774804113215434$

$x \approx 1.92756197548293$

Figure A.1: QuinticFibo Characteristic Equation Roots (Wolfram Alpha, n.d.)

A.5 Analysis of Searching and Sorting Algorithms

This section tackes the analysis of two searching algortihms and five sorting algorithms. The two searching algorithms are Linear and Binary search. The five sorting algorithms are Bubble sort, Insertion sort, Merge sort, Quicksort, and Selection sort. The experiments and calculations are set up such that they measure the worst case asymptotic behavior of these algorithms.

A.5.1 Analysis of Linear Search

The implementation of the Linear search algorithm in Python is seen in Listing A.13.

Listing A.13: Linear search

```python
def linearSearch(alist, item):
    for i in range(0, len(alist)):
        if alist[i] == item:
            return i
    return -1
```

The setup of the experiment could be seen in Listing A.14 which is the worst case scenario. The array to be searched is an array containing incremetal values from 0 to $n-1$. The item to be search is $n-1$ which is at the last index of the array. This means that it has to check all the elements of the array.

Listing A.14: Worst case Linear search

```python
def f(n):
    x = range(0,n,1)
    linearSearch(x, n-1)
```

$$F_{Linear} = EX_i + n * (IT_i + 1) + 1 = 2n + 2$$

A.5.2 Analysis of Binary Search

The implementation of the Binary search algorithm in Python is seen in Listing A.15.

Listing A.15: Binary search

```python
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False
    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            return midpoint
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1
    return -1
```

The setup of the experiment could be seen in Listing A.16 which is the worst case scenario similar to Listing A.14. It requires the algorithm to keep on dividing the array up until it reaches the last index.

Listing A.16: Worst case Binary search

```python
def f(n):
    x = range(0,n,1)
    linearSearch(x, n-1)
```

$F_{Binary} = 3 + EX + i * (IT + 4) - 1$, where $i = log_2(n)$

$$= 5log_2(n) + 3 = \frac{5}{log(2)} log(n) + 3 \simeq 7.213475log(n) + 3$$

A.5.3 Analysis of Bubble sort

The implementation of the Bubble sort algorithm in Python is seen in Listing A.17.

Listing A.17: Bubble sort

```
def bubble_sort(items):
    for i in range(len(items)):
        for j in range(len(items)-1-i):
            if items[j] > items[j+1]:
                items[j],items[j+1]=items[j+1],items[j]
```

The setup of the experiment could be seen in Listing A.18 which is the worst case scenario. The Bubble sort algorithm arranges the array in ascending order while the input array is sorted in descending order. This means that it must compare and swap each element of the array for every step.

Listing A.18: Worst case Bubble sort

```
def f(n):
    x = range(n,0,-1)
    bubble_sort(x)
```

$$F_{Bubble} = EX_i + \sum_{i=0}^{n-1}(IT_i + EX_j + \sum_{j=0}^{n-i-1}(IT_j + 2)) = 1 + \sum_{i=0}^{n-1}(2 + (n-i)*3)$$
$$= 1 + 2n + 3n^2 - 3(n)(n-1)/2 = 1 + 2n + 3n^2 - \frac{3}{2}n^2 + \frac{3}{2}n = \frac{3}{2}n^2 + \frac{7}{2}n + 1$$

A.5.4 Analysis of Insertion sort

The implementation of the Insertion sort algorithm in Python is seen in Listing A.19.

Listing A.19: Insertion sort

```
def insertion_sort(items):
    for i in range(1, len(items)):
        j = i
        while j > 0 and items[j] < items[j-1]:
            items[j],items[j-1] = items[j-1],items[j]
            j -= 1
```

The setup of the experiment could be seen in Listing A.20 which is the worst case scenario. The Insertion sort algorithm arranges the array in ascending order while the input array is sorted in descending order. This means that it must also compare and swap at each step like the worst case of Bubble sort.

Listing A.20: Worst case Inserion sort

```
def f(n):
    x = range(n,0,-1)
    insertion_sort(x)
```

$F_{Insert} = EX_i + \sum_{i=0}^{n-1}(IT_i + EX + (n-i)*(IT+2)) = 1 + \sum_{i=0}^{n-1}(2+(n-i)*3)$
$= 1 + 2n + 3n^2 - 3(n)(n-1)/2 = 1 + 2n + 3n^2 - \frac{3}{2}n^2 + \frac{3}{2}n = \frac{3}{2}n^2 + \frac{7}{2}n + 1$

A.5.5 Analysis of Merge sort

The implementation of the Merge sort algorithm in Python is seen in Listing A.21.

Listing A.21: Merge sort

```
def merge_sort(items):
    if len(items) > 1:
        mid = len(items) // 2          # Determine the
            midpoint and split
        left = items[0:mid]
        right = items[mid:]
        merge_sort(left)               # Sort left list in-
            place
        merge_sort(right)              # Sort right list in-
            place
        l = 0
        r = 0
        for i in range(len(items)):        # Merging the left
             and right list
            if l < len(left):
                lval = left[l]
            else:
                lval = None
            if r < len(right):
                rval = right[r]
            else:
                rval = None
            if (lval is not None and rval is not None and
                lval < rval) or rval is None:
                items[i] = lval
                l += 1
            else:
                items[i] = rval
                r += 1
```

The setup of the experiment could be seen in Listing A.22 which is the worst case scenario. The Merge sort algorithm arranges the array in ascending order while the input array is sorted in ascending order. This means that it must compare and swap each element of the array for every step. This is the worst case because the number of executed instructions do not change depending on the arrangement of the elements in the array when using Merge sort.

Listing A.22: Worst case Merge sort

```
def f(n):
    x = range(0,n,1)
    merge_sort(x)
```

$F_{Merge} = R(n); R(0) = R(1) = 1$

$R(n) = 4 + R(\frac{n}{2}) + R(1 - \frac{n}{2}) + 5 + EX_i + n * (IT_i + 7) = 2R(\frac{n}{2}) + 8n + 9$

$= 2(2R(\frac{n}{4}) + 4n + 9) + 8n + 9 = 4R(\frac{n}{4}) + 8n + 18 + 8n + 9 = 4R(\frac{n}{4}) + 16n + 27$

$= 4(2R(\frac{n}{8}) + 2n + 9) + 16n + 27 = 8R(\frac{n}{8}) + 8n + 36 + 16n + 27 = 8R(\frac{n}{8}) + 24n + 63$

$= 8(2R(\frac{n}{16}) + n + 9) + 24n + 63 = 16R(\frac{n}{16}) + 8n + 72 + 24n + 63 = 16R(\frac{n}{16}) + 32n + 135$

$= ... = 2^i R(\frac{n}{2^i}) + 8in + 9 * (2^i - 1) = 2^i R(\frac{n}{2^i}) + 8in + 9 * 2^i - 9$

To reduce the recursion to the base case, i must satisfy:

$\frac{n}{2^i} = 1 \rightarrow n = 2^i \rightarrow i = \frac{log(n)}{log(2)}$

By setting the value of i to $\frac{log(n)}{log(2)}$:

$R(n) = nR(1) + 8in + 9 * n - 9 = n * 1 + 8\frac{log(n)}{log(2)} + 9 * n - 9 = \frac{8}{log(2)} nlog(n) + 10n - 9$

$R(n) = F_{Merge} \simeq 11.541560nlog(n) + 10n - 9$

A.5.6 Analysis of Quicksort

The implementation of the Quicksort algorithm in Python is seen in Listing A.23. The pivot is always set to the first index in this case. The experiment is setup by passing an array sorted in decreasing order with a size of n.

Listing A.23: Merge sort (Pivot is first index)

```python
def quick_sort(items):
    if len(items) > 1:
        pivot_index = 0
        smaller_items = []
        larger_items = []
        for i, val in enumerate(items):
            if i != pivot_index:
                if val < items[pivot_index]:
                    smaller_items.append(val)
                else:
                    larger_items.append(val)
        quick_sort(smaller_items)
        quick_sort(larger_items)
        items[:] = smaller_items + [items[pivot_index]] +
            larger_items
```

$F_{Quick} = R(n); R(0) = R(1) = 1$
$R(n) = 4 + EX_i + n(IT_i + 3) + 3 + R(n-1) + 1 = R(n-1) + 4n + 9$
$= R(n-2) + 4(n-1) + 9 + 4n + 9 = R(n-2) + 2(4n+9) - 4$
$= R(n-3) + 4(n-2) + 9 + 2(4n+9) - 4 = R(n-2) + 2(4n+9) - 4(1+2)$
$= ... = R(n-i) + i(4n+9) - 4\sum_{j=0}^{i-1} j = R(n-i) + i(4n+9) - 4\dfrac{i(i-1)}{2}$
$= R(n-i) + i(4n+9) - 2i(i-1)$
To reduce the recursion to the base case, i must satisfy:
$n - i = 1 \rightarrow i = n - 1$
By setting the value of i to $n-1$:
$R(n) = R(1) + (n-1)(4n+9) - 2(n-1)(n-2) = 1 + 4n^2 + 9n - 4n - 9 - 2n^2 + 4n + 2n - 4$
$R(n) = F_{Quick} = 2n^2 + 11n - 12$

A.5.7 Analysis of Selection sort

The implementation of the Selection sort algorithm in Python is seen in Listing A.24.

Listing A.24: Selection sort

```
def selection_sort(alist):
    for fillslot in range(len(alist)-1,0,-1):
        positionOfMax=0
        for location in range(1,fillslot+1):
            if alist[location]>alist[positionOfMax]:
                positionOfMax = location
        alist[fillslot], alist[positionOfMax] = alist[
            positionOfMax], alist[fillslot]
```

The setup of the experiment could be seen in Listing A.25 which is the worst case scenario. The Selection sort algorithm arranges the array in ascending order while the input array is sorted in ascending order. This is the worst case scenario because Selection sort looks for the largest elements and places it in the furthest end of the array. It has to keep saving the index of the max element at each step since the element at the succeeding step is always incrementally larger.

Listing A.25: Worst case Selection sort

```
def f(n):
    x = range(0,n,1)
    selection_sort(x)
```

$$F_{Selection} = EX_F + \sum_{F=1}^{n-1}(IT_F + 1 + EX_L + \sum_{L=1}^{F}(IT_L + 2) + 1)$$
$$= 1+\sum_{F=1}^{n-1}(\sum_{L=1}^{F}(3)+4) = 1+\sum_{F=1}^{n-1}(3F+4) = 1+(n-1)*4+3*(\frac{n(n-1)}{2}-1)$$
$$= 1 + 4n - 4 + \frac{3}{2}n^2 - \frac{3}{2}n - 3 = \frac{3}{2}n^2 - \frac{11}{2}n - 6$$

A.6 Analysis of Piecewise Algorithm

The Piecewise algorithm is an algorithm which has a performance that is piecewise in nature as seen in Listing A.26. It exhibits a constant behavior for inputs less than 100, linear behavior for inputs between 99 and 200, linear behavior with multiples of 4 for inputs between 199 and 300, and a quadratic behavior for inputs that are greater than 300.

Listing A.26: Piecewise Algorithm

```
def f(x):
    loops = 0
    if x<100:
        loops = 16
    else:
        if x<200:
            loops = x
        else:
            if x<300:
                loops = 4*x
            else:
                loops = x**2
    for i in xrange(loops):
        t=0
```

$$F_{Piecewise} = \begin{cases} 3 + EX_i + 16(IT_i + 1) = 36 & x < 100 \\ 4 + EX_i + n(IT_i + 1) = 2n + 5 & 100 \geq x < 200 \\ 5 + EX_i + 4n(IT_i + 1) = 8n + 6 & 200 \geq x < 300 \\ 6 + EX_i + n^2(IT_i + 1) = 2n^2 + 7 & x \geq 300 \end{cases}$$

$\therefore F_{Piecewise} \sim 2n^2 + 7$ as $n \to \infty$.

Appendix B

Derivation of the Formulas for e, p, and c and Determining hasLog

The array of frequency count measurements be the parameter of the formulas that determine asymptotic behavior. The method iterates n over each term of the array and creates an approximation of the asymptotic behavior for each term. The approximations get better when more terms are considered. x, y, and z values are determined for each iteration of n. These three values will be used in the formulas. For the continuous e, p, and c calculations, x, y, z will take the values n, $n-1$, $n-2$ respectively. For the e, p, and c calculations around discontinuities, x will be set to the largest discontinuous point less than n, y will be set to the second largest discontinuous point less than n, and z will be set to the third largest discontinuous point less than n. A method for detecting discontinuities can be seen in Appendix B.5.

The following 3 sections would be the derivation of the formulas for e, p, and c respectively. These derivations assume the case where there is a logarithmic growth involved. To get the formula for the case without logarithmic growth, set all occurrences of ln(x) to 1. To simplify the expressions, "=" is used in place of "$\sim$" and "as $x \to \infty$". The log and ln functions used both pertain to the natural log; ln is used to denote the log factor from the asymptotic behavior of the assumption while log is used when using logarithms in the manipulation of equations.

B.1 Derivation of the Formula For e

$$a_x = e^x x^p c \ln(x) \tag{B.1.1}$$

By dividing both sides of Equation B.1.1 by $e^x x^p ln(x)$ and introducing an equivalent expression:

$$\frac{a_x}{e^x x^p \ln(x)} = c = \frac{a_y}{e^y y^p \ln(y)} \tag{B.1.2}$$

By cross multiplying Equation B.1.2 to isolate variables with p:

$$\frac{a_x}{a_y} e^{y-x} \frac{ln(y)}{ln(x)} = (\frac{x}{y})^p \tag{B.1.3}$$

By taking logarithms both sides of B.1.3:

$$(y-x)log(e) + log(a_x) - log(a_y) + log(ln(y)) - log(ln(x)) = p(log(x) - log(y)) \tag{B.1.4}$$

By dividing both sides of Equation B.1.4 by $log(x) - log(y)$ and introducing an equivalent expression:

$$\begin{aligned} &\frac{(y-x)log(e) + log(a_x) - log(a_y) + log(ln(y)) - log(ln(x))}{log(x) - log(y)} = p \\ &= \frac{(z-y)log(e) + log(a_y) - log(a_z) + log(ln(z)) - log(ln(y))}{log(y) - log(z)} \end{aligned} \tag{B.1.5}$$

By cross multiplying the denominators of Equation B.1.5:

$$\begin{aligned} &(log(y) - log(z))((y-x)log(e) + log(a_x) - log(a_y) + log(ln(y)) - log(ln(x))) \\ &= (log(x) - log(y))((z-y)log(e) + log(a_y) - log(a_z) + log(ln(z)) - log(ln(y))) \end{aligned} \tag{B.1.6}$$

By isolating terms with log(e) in Equation B.1.6:

$$\begin{aligned} &(log(y) - log(z))(log(a_x) - log(a_y) + log(ln(y)) - log(ln(x))) \\ &-(log(x) - log(y))(log(a_y) - log(a_z) + log(ln(z)) - log(ln(y))) \\ &= log(e)((log(x) - log(y))(z-y) - (log(y) - log(z))(y-x)) \end{aligned} \tag{B.1.7}$$

By dividing both sides of Equation B.1.7 by ((log(x)-log(y))(z-y)-(log(y)-log(z))(y-x)):

$$\begin{aligned}&((log(y)-log(z))(log(a_x)-log(a_y)+log(ln(y))-log(ln(x)))\\&-(log(x)-log(y))(log(a_y)-log(a_z)+log(ln(z))-log(ln(y))))\\&\div((log(x)-log(y))(z-y)-(log(y)-log(z))(y-x))=log(e)\end{aligned}\tag{B.1.8}$$

By exponentiating both sides of Equation B.1.8:

$$\begin{aligned}e=exp(((log(y)-log(z))(log(a_x)-log(a_y)+log(ln(y))-log(ln(x)))\\-(log(x)-log(y))(log(a_y)-log(a_z)+log(ln(z))-log(ln(y))))\\\div((log(x)-log(y))(z-y)-(log(y)-log(z))(y-x)))\end{aligned}\tag{B.1.9}$$

By setting all ln(x) values of Equation B.1.9 to 1, formula for e without the presence of an ln(x) factor can be obtained:

$$\begin{aligned}e_{nl}=exp(((log(y)-log(z))(log(a_x)-log(a_y))\\-(log(x)-log(y))(log(a_y)-log(a_z)))\\\div((log(x)-log(y))(z-y)-(log(y)-log(z))(y-x)))\end{aligned}\tag{B.1.10}$$

B.2 Derivation of the Formula For p

$$a_x=e^x x^p c\ln(x)\tag{B.2.1}$$

By dividing both sides of Equation B.2.1 by $e^x x^p ln(x)$ and introducing an equivalent expression:

$$\frac{a_x}{e^x x^p\ln(x)}=c=\frac{a_y}{e^y y^p\ln(y)}\tag{B.2.2}$$

By cross multiplying Equation B.2.2 to isolate variables with e:

$$\frac{a_x}{a_y}(\frac{y}{x})^p\frac{ln(y)}{ln(x)}=e^{x-y}\tag{B.2.3}$$

By taking logarithms both sides of B.2.3:

$$p(log(y)-log(x))+log(a_x)-log(a_y)+log(ln(y))-log(ln(x))=(x-y)log(e)\tag{B.2.4}$$

By dividing both sides of Equation B.2.4 by $x - y$ and introducing an equivalent expression:

$$\begin{aligned}\frac{p(log(y) - log(x)) + log(a_x) - log(a_y) + log(ln(y)) - log(ln(x))}{(x-y)} &= log(e) \\ = \frac{p(log(z) - log(y)) + log(a_y) - log(a_z) + log(ln(z)) - log(ln(y))}{(y-z)}\end{aligned} \tag{B.2.5}$$

By cross multiplying the denominators of Equation B.2.5:

$$\begin{aligned}&(y-z)(p(log(y) - log(x)) + log(a_x) - log(a_y) + log(ln(y)) - log(ln(x))) \\ &= (x-y)(p(log(z) - log(y)) + log(a_y) - log(a_z) + log(ln(z)) - log(ln(y)))\end{aligned} \tag{B.2.6}$$

By isolating terms with p in Equation B.2.6:

$$\begin{aligned}&(y-z)(log(a_x) - log(a_y) + log(ln(y)) - log(ln(x))) \\ &-(x-y)(log(a_y) - log(a_z) + log(ln(z)) - log(ln(y))) \\ &= p((x-y)(log(z) - log(y)) - (y-z)(log(y) - log(x)))\end{aligned} \tag{B.2.7}$$

By dividing both sides of Equation B.2.7 by
$(x-y)(log(z) - log(y)) - (y-z)(log(y) - log(x))$:

$$\begin{aligned}p = &((y-z)(log(a_x) - log(a_y) + log(ln(y)) - log(ln(x))) \\ &-(x-y)(log(a_y) - log(a_z) + log(ln(z)) - log(ln(y)))) \\ &\div((x-y)(log(z) - log(y)) - (y-z)(log(y) - log(x)))\end{aligned} \tag{B.2.8}$$

By setting all ln(x) values of Equation B.2.8 to 1, formula for p without the presence of an ln(x) factor can be obtained:

$$\begin{aligned}p_{nl} = &((y-z)(log(a_x) - log(a_y)) - (x-y)(log(a_y) - log(a_z))) \\ &\div((x-y)(log(z) - log(y)) - (y-z)(log(y) - log(x)))\end{aligned} \tag{B.2.9}$$

B.3 Derivation of the Formula For c

$$a_x = e^x x^p c \ln(x) \tag{B.3.1}$$

By dividing both sides of Equation B.3.1 by $x^p cln(x)$:

$$\frac{a_x}{x^p c \ln(x)} = e^x \tag{B.3.2}$$

By taking logarithms both sides of B.3.2:

$$log(a_x) - plog(x) - log(c) - log(ln(x)) = xlog(e) \tag{B.3.3}$$

By dividing both sides of Equation B.3.3 by x and introducing an equivalent expression:

$$\begin{aligned}\frac{log(a_x) - plog(x) - log(ln(x)) - log(c)}{x} &= log(e) \\ = \frac{log(a_y) - plog(y) - log(ln(y)) - log(c)}{y}\end{aligned} \tag{B.3.4}$$

By cross multiplying the denominators of Equation B.3.4:

$$\begin{aligned}&y(log(a_x) - plog(x) - log(ln(x))) - ylog(c) \\ &= x(log(a_y) - plog(y) - log(ln(y))) - xlog(c)\end{aligned} \tag{B.3.5}$$

By isolating terms with p in Equation B.3.5:

$$\begin{aligned}y(log(a_x) - log(ln(x))) + log(c)(x-y) - x(log(a_y) - log(ln(y))) \\ = p(ylog(x) - xlog(y))\end{aligned} \tag{B.3.6}$$

By dividing both sides of Equation B.3.6 by
$ylog(x) - xlog(y)$ and introducing an equivalent expression:

$$\begin{aligned}\frac{log(c)(x-y) + y(log(a_x) - log(ln(x))) - x(log(a_y) - log(ln(y)))}{(ylog(x) - xlog(y))} &= p \\ = \frac{log(c)(y-z) + z(log(a_y) - log(ln(y))) - y(log(a_z) - log(ln(z)))}{(z\ log(y) - ylog(z))}\end{aligned} \tag{B.3.7}$$

By cross multiplying the denominators of Equation B.3.7:

$$\begin{aligned}&(zlog(y) - ylog(z))(log(c)(x-y) + y(log(a_x) - log(ln(x))) - x(log(a_y) \\ &-log(ln(y)))) = (ylog(x) - xlog(y))(log(c)(y-z) + z(log(a_y) - log(ln(y))) \\ &-y(log(a_z) - log(ln(z))))\end{aligned} \tag{B.3.8}$$

By isolating terms with c in Equation B.3.8:

$$\begin{aligned}
&(zlog(y) - ylog(z))(y(log(a_x) - log(ln(x))) - x(log(a_y) - log(ln(y)))) \\
&-(ylog(x) - xlog(y))(z(log(a_y) - log(ln(y))) - y(log(a_z) - log(ln(z)))) \\
&\quad = log(c)((ylog(x) - xlog(y))(y - z) - (z\ log(y) - ylog(z))(x - y))
\end{aligned} \tag{B.3.9}$$

By dividing both sides of Equation B.3.9 by
$((ylog(x) - xlog(y)(y - z) - (z\ log(y) - ylog(z))(x - y))$ and exponentiating:

$$\begin{aligned}
c = exp(&((z\ log(y) - ylog(z))(y(log(a_x) - log(ln(x))) - x(log(a_y) - log(ln(y)))) \\
&-(ylog(x) - xlog(y))(z(log(a_y) - log(ln(y))) - y(log(a_z) - log(ln(z))))) \\
&\div((ylog(x) - xlog(y))(y - z) - (z\ log(y) - ylog(z))(x - y)))
\end{aligned} \tag{B.3.10}$$

By setting all ln(x) values of Equation B.3.10 to 1, formula for c without the presence of an ln(x) factor can be obtained:

$$\begin{aligned}
c_{nl} = exp(&((z\ log(y) - ylog(z))(y(log(a_x)) - x(log(a_y))) \\
&-(ylog(x) - xlog(y))(z(log(a_y)) - y(log(a_z)))) \\
\div((ylog(x) &- xlog(y))(y - z) - (z\ log(y) - ylog(z))(x - y)))
\end{aligned} \tag{B.3.11}$$

B.4 Determining hasLog

Two counters will be used in determining which of the two cases, log or no log, is closer to the observed measurements. If $|a_n - {e_{nl}}^n * n^{p_{nl}} * c_{nl}| > |a_n - e^n * n^p * c \ln(n)|$ then the counter for log case will increment; else the counter for the no log case will increment. If the continuous case is chosen, then this condition will be checked for all terms. If the discontinuous case is chosen, then this condition will only be checked on the discontinuity points. The following section contains all the discussions about discontinuity. The log case is chosen if the counter for the log case is greater than the no log case.

B.5 Evaluating Discontinuities

Discontinuities are places where functions instantaneously jump from one place to another. In other words, the value of a function approaching a discontinuity from the left is different from the value approaching from its right. Figure B.1 is an example of a function, $floor(ln(n))$, which contains discontinuities (highlighted by green ovals).

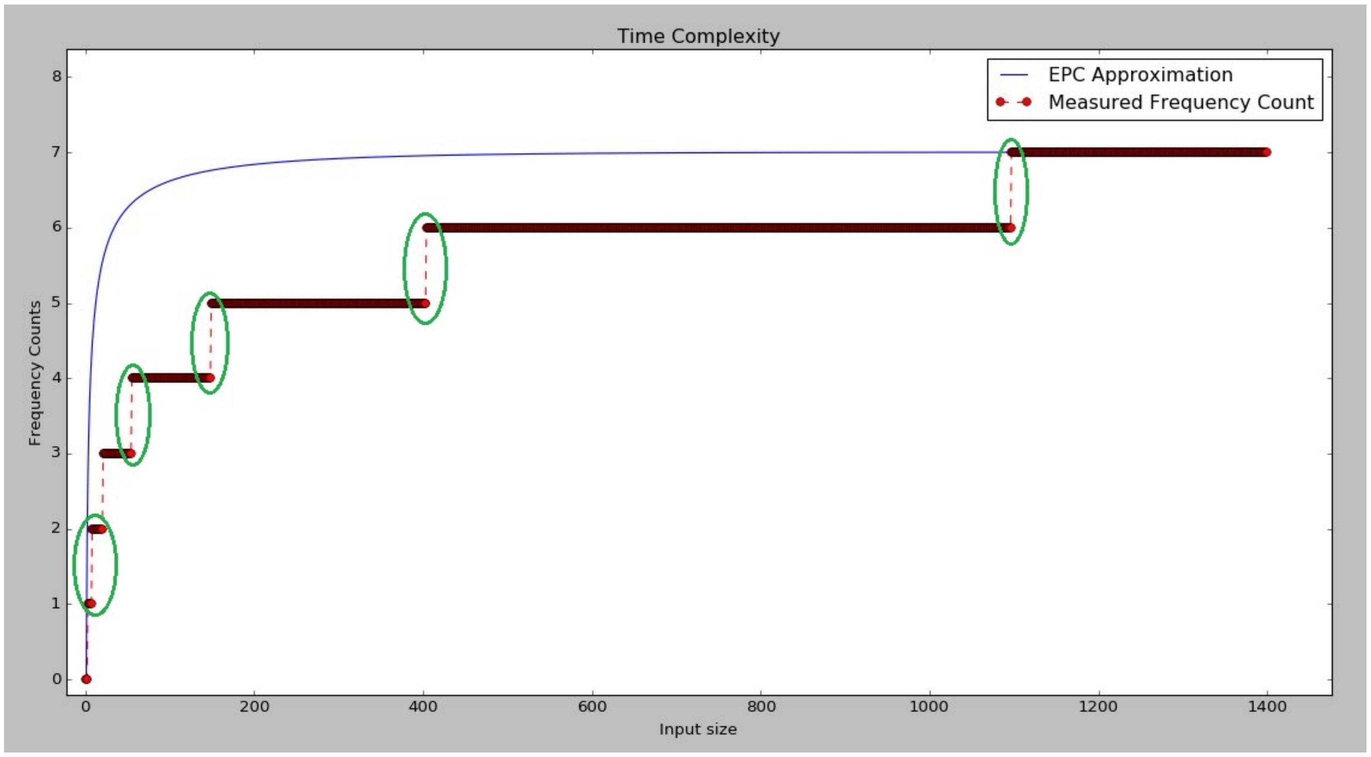
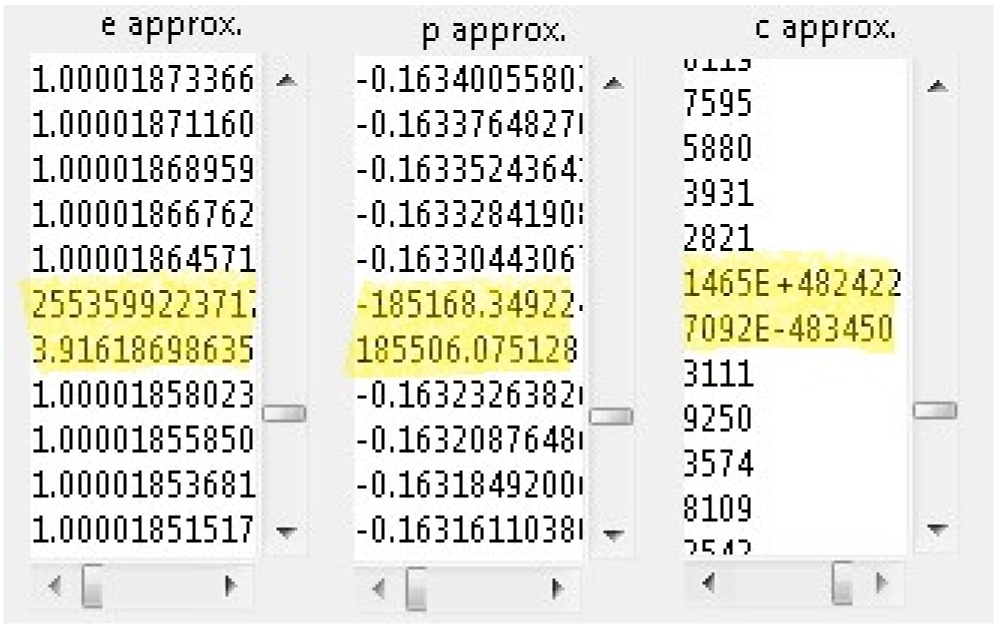


Figure B.1: Sample of Discontinuous Function

Discontinuities could be detected by looking for sudden fluctuations within the e, p, and c approximations. An example of a fluctuation could be seen highlighted in Figure B.2. The approximations are calculated around three points. When at least one of these three points vary in behavior relative to the rest of the points, then the curve that will "fit" these three points will be distorted. If one calculates the e, p, and c approximations within the green region in the example given in Figure B.1, then the resulting approximation yield a curve that seems to skyrocket straight upwards and then fall straight down or vice versa.

The e, p, and c approximations tend to converge on a particular value as the parameter size n increases. This means that the differences between the approximation should get smaller and smaller. This concept could be converted



```
e approx.          p approx.          c approx.
1.00001873366      -0.1634005580
1.00001871160      -0.1633764827      7595
1.00001868959      -0.1633524364      5880
1.00001866762      -0.1633284190      3931
1.00001864571      -0.1633044306      2821
255359922371       -185168.34922      1465E+482422
3.91618698635      185506.075128      7092E-483450
1.00001858023      -0.1632326382      3111
1.00001855850      -0.1632087648      9250
1.00001853681      -0.1631849200      3574
1.00001851517      -0.1631611038      8109
```

Figure B.2: Sample of Fluctuation due to a Discontinuity

to a heuristic that detects discontinuities which has the form:
$(|e_i - e_{i-1}| < |e_{i+1} - e_i| \wedge |p_i - p_{i-1}| < |p_{i+1} - p_i| \wedge |c_i - c_{i-1}| < |c_{i+1} - c_i|)$

Another condition is used in conjunction with the previous condition to avoid false positive detections. The previous condition is susceptible to tiny amounts of noise. The other condition uses the property that the approximations tend to skyrocket at discontinuities which will filter the otherwise false positive detections. It has the form: $|e_{i+1} - e_i| + |p_{i+1} - p_i| + |c_{i+1} - c_i|) > 1$.

These conditions are checked for each value of index i less than $n - 1$ and greater than 3. The checking starts at index 4 because the number of terms required to yield reasonable approximations are lacking at this point. If both are satisfied, then it means that there is a discontinuity between i and $i + 1$. $i + 1$ will be considered a point to calculate with and i will be incremented by 2 to avoid detecting the same discontinuity twice. The point at $i + 1$ is chosen because discontinuity due to round down or floor function is assumed. This is because commonly used algorithms contain base cases that use a condition with a minimum threshold or a less than sign. The terms with the least error when rounded down are the terms immediately right after a discontinuity.

If there are more than two discontinuities found, then new approximations will be calculated around these discontinuities. The next step is to choose between approximations calculated continuously and around discontinuities. This is done using two counters, $contCount$ and $discontCount$. The distance between the approximations and the actual measurements will serve as the error metric. If the error for the continuous case is less than the error for the discontinuous case, then $discontCount$ is incremented, else $contCount$ is incremented. This condition done for each of the log and no log case. The discontinuous case will be chosen if $discontCount$ is greater than $contCount$.

Appendix C

Validation Table Derivations

The following calculations are done in the construction of the tables in Section 5.2. Note that n is a positive number approaching infinity which is useful fact in the simplification of expressions.

C.1 Calculations in Table 5.2

The values in column ΔF_n use the definition $\Delta F_n = F_{n+1} - Fn$, and the values in column $\frac{d}{dn}|\Delta F_n|$ use the derivative of the absolute value of ΔF_n. The following are the evaluations of these quanities.

C.1.1 ΔF_n Column

Row $F_n = r^n$:

$$r^{n+1} - r^n = r^n(r-1) \tag{C.1.1}$$

Row $F_n = n^r$: (Using the Maclaurin expansion of the binomial series)

$$(n+1)^r - n^r = -n^r + \frac{n^r}{0!} + \frac{(r)n^{r-1}}{1!} + \frac{r(r-1)n^{r-2}}{2!} + ... \sim rn^{r-1} \text{ as } n \to \infty \tag{C.1.2}$$

Row $F_n = log_r(n)$**:**

$$log_r(n+1) - log_r(n) = log_r(\frac{n+1}{n}) = log_r(1 + \frac{1}{n}) \tag{C.1.3}$$

C.1.2 $\frac{d}{dn}|\Delta F_n|$ Column

Row $F_n = r^n$**:**

$$\frac{d}{dn}|(r-1)r^n| \tag{C.1.4}$$

By using the absolute value rule in differentiation on C.1.4:

$$\frac{(r-1)r^n}{|(r-1)r^n|} * \frac{d}{dn}(r-1)r^n \tag{C.1.5}$$

By using the exponential rule in differentiation on C.1.5:

$$\frac{(r-1)r^n}{|(r-1)r^n|} * (r-1)log(r)r^n = log(r) * \frac{(r-1)^2 r^{2n}}{|(r-1)r^n|} = log(r)|(r-1)r^n| \tag{C.1.6}$$

Row $F_n = n^r$**:**

$$\frac{d}{dn}|(r)n^{r-1}| \tag{C.1.7}$$

By using the absolute value rule in differentiation on C.1.7:

$$\frac{(r)n^{r-1}}{|(r)n^{r-1}|} * \frac{d}{dn}(r)n^{r-1} \tag{C.1.8}$$

By using the power rule in differentiation on C.1.8:

$$\frac{(r)n^{r-1}}{|(r)n^{r-1}|} * r(r-1)n^{r-2} = (r-1) * \frac{r^2 n^{r-3}}{|(r)n^{r-1}|} = (r-1)|r| * \frac{1}{n^2} \tag{C.1.9}$$

Row $F_n = log_r(n)$**:**

$$\frac{d}{dn}|log_r(1 + \frac{1}{n})| \tag{C.1.10}$$

By using the absolute value rule in differentiation on C.1.10 and converting the logarithms from base k to natural base:

$$\frac{log(1+\frac{1}{n})|log(r)|}{|log(1+\frac{1}{n})|log(r)} * \frac{d}{dn}(log_r(1+\frac{1}{n})) \tag{C.1.11}$$

By using the logarithmic rule in differentiation and the chain rule on C.1.11:

$$\frac{log(1+\frac{1}{n})|log(r)|}{|log_r(1+\frac{1}{n})|log(r)} * \frac{1}{(1+\frac{1}{n})log(r)} * \frac{d}{dn}(1+\frac{1}{n}) \tag{C.1.12}$$

By using the power rule in differentiation on C.1.12:

$$\frac{log(1+\frac{1}{n})|log(r)|}{|log_r(1+\frac{1}{n})|log(r)} * \frac{1}{(1+\frac{1}{n})log(r)} * \frac{-1}{n^2} = \frac{-|log(r)|}{(log(r))^2} * \frac{1}{n^2+n} = \frac{-1}{|log(r)|} * \frac{1}{n(n+1)} \tag{C.1.13}$$

C.2 Calculations in Table 5.3

C.2.1 $\Delta' F_n$ and $\frac{d}{dn}|\Delta' F_n|$ Columns

The calculations for the $\Delta' F_n$ Column in Table 5.3 is equal to the calculations for the ΔF_n Column in Table 5.2 but multiplied with an additional factor of n. While the calculations for the $\frac{d}{dn}|\Delta' F_n|$ Column are the following:
Row $F_n = r^n$:

$$\frac{d}{dn}|n(r-1)r^n| \tag{C.2.1}$$

By using the absolute value rule in differentiation on C.2.1:

$$\frac{n(r-1)r^n}{|n(r-1)r^n|} * \frac{d}{dn}(n(r-1)r^n) \tag{C.2.2}$$

By using the product rule in differentiation on C.2.2:

$$\begin{aligned}\frac{n(r-1)r^n}{|n(r-1)r^n|} * (r-1)(n * log(r)r^n + r^n * 1)\\ = \frac{n(r-1)r^n}{|n(r-1)r^n|} * (r-1)r^n(n * log(r) + 1)\\ = (log(r) + \frac{1}{n})|(r-1)r^n| * n\end{aligned} \tag{C.2.3}$$

Row $F_n = n^r$**:**

$$\frac{d}{dn}|(nr)n^{r-1}| \tag{C.2.4}$$

By using the absolute value rule in differentiation on C.2.4:

$$\frac{(nr)n^{r-1}}{|(nr)n^{r-1}|} * \frac{d}{dn}((nr)n^{r-1}) \tag{C.2.5}$$

By using the product rule in differentiation on C.2.5:

$$\begin{aligned}\frac{(nr)n^{r-1}}{|(nr)n^{r-1}|} * r(n * (r-1)n^{r-2} + n^{r-1} * 1)\\ = \frac{(nr)n^{r-1}}{|(nr)n^{r-1}|} * (r)n^{r-1}(r-1+1)\\ = r * |r|n^{r-1}\end{aligned} \tag{C.2.6}$$

Row $F_n = log_r(n)$**:**

$$\frac{d}{dn}|nlog_r(1 + \frac{1}{n})| \tag{C.2.7}$$

By using the absolute value rule in differentiation on C.2.7 and converting the logarithms from base k to natural base:

$$\frac{nlog(1+\frac{1}{n})|log(r)|}{|nlog(1+\frac{1}{n})|log(r)} * \frac{d}{dn}(\frac{nlog(1+\frac{1}{n})}{log(r)}) \tag{C.2.8}$$

By using the logarithmic rule and chain rule in differentiation on C.2.8 and converting the logarithms from base r to natural base:

$$\frac{nlog(1+\frac{1}{n})|log(r)|}{|nlog(1+\frac{1}{n})|log(r)} * \frac{1}{log(r)}(n * \frac{1}{(1+\frac{1}{n})} * \frac{-1}{n^2} + log(1+\frac{1}{n}) * 1) = \frac{log(1+\frac{1}{n}) - \frac{1}{n+1}}{|log(r)|} \tag{C.2.9}$$

C.2.2 Proof of $log(1+\frac{1}{n}) > \frac{1}{n+1}$

This subsection contains the proof of $log(1+\frac{1}{n}) > \frac{1}{n+1}$ when $n > 0$. This is necessary in showing that $\frac{d}{dn}|nlog_r(1+\frac{1}{n})|$ is positive everywhere except when $r = 0$ where it is 0.
Notice that $log(1+\frac{1}{n})$ could be rewritten as a definite integral of the form:

$$log(1+\frac{1}{n}) = \int_0^{\frac{1}{n+1}} \frac{1}{1-x}dx \tag{C.2.10}$$

While $\frac{1}{n+1}$ could also be rewritten as a definite integral of the form:

$$\frac{1}{n+1} = \int_0^{\frac{1}{n+1}} 1dx \tag{C.2.11}$$

Since integration could be thought of as summing up the area of infinitesimally small strips of a function between an interval, showing that all the small strips of Function C.2.10 is greater than all the corresponding small strips of Function C.2.11 within the interval is enough to show that $log(1+\frac{1}{n}) > \frac{1}{n+1}$.

The strip at x=0 covers the same area in both of the integral forms:

$$(x = 0) \rightarrow (\frac{1}{1-0} = 1) \rightarrow (1 = 1) \tag{C.2.12}$$

While the the C.2.10 strips covers more area than the C.2.11 strips at everywhere else in the interval could be shown by contradiction:

$$\begin{aligned}(0 < x \leq \frac{1}{n+1} \wedge \frac{1}{1-x} \leq 1) \to (1 \leq 1-x) \to (x \leq 0)(\text{contradiction}) \\ \to (\frac{1}{1-x} > 1)\end{aligned} \tag{C.2.13}$$

$$\therefore (n > 0) \to (\int_0^{\frac{1}{n+1}} \frac{1}{1-x} dx > \int_0^{\frac{1}{n+1}} 1 dx) \to (log(1+\frac{1}{n}) > \frac{1}{n+1}) \ \blacksquare \tag{C.2.14}$$

Appendix D

Limit Definition Equivalences of Asymptotic Notations

The following are the proofs that show the definitions of Big O, Big Omega, Big Theta, Little O, and Little Omega are equivalent to the limit definitions in Section 3.4. The proofs operate under the assumption that limit of $\lim_{n\to\infty} \dfrac{f(n)}{g(n)}$ exists and f(n) and g(n) are always positivD. Note that the convergence criterion is defined in Montesinos et al. (2015).

D.1 Big O $\leftrightarrow$ Limit Big O

The proof being shown has been separated into two parts since it is not easily reversible from one way to another.

D.1.1 Big O $\rightarrow$ Limit Big O

The condition for Big O is:

$$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(f(n) \leq cg(n)) \tag{D.1.1}$$

By dividing $g(n)$ on both sides of the inequality in D.1.1:

$$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{f(n)}{g(n)} \leq c) \tag{D.1.2}$$

By universal instantiation on D.1.2, n could be chosen arbitrarily larger than n_0 by letting n be a positive real number that approaches infinity:

$$\exists(c \in \mathbb{R}^+) \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c \tag{D.1.3}$$

By using the fact that infinity is larger than any positive real number:

$$\exists(c \in \mathbb{R}^+) \to (c < \infty) \tag{D.1.4}$$

By combining D.1.3 and D.1.4:

$$\exists(c \in \mathbb{R}^+) \lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c < \infty \tag{D.1.5}$$

By simplifying D.1.5:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \blacksquare \tag{D.1.6}$$

D.1.2 Limit Big O $\to$ Big O

The Limit condition for Big O is:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \tag{D.1.7}$$

By introducing the convergence criterion:

$$\forall(d \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(|\frac{f(n)}{g(n)} - \lim_{n\to\infty} \frac{f(n)}{g(n)}| < d) \tag{D.1.8}$$

D.1.8 could be proven by separating it into two cases:
The case when $\frac{f(n)}{g(n)} > \lim_{n\to\infty} \frac{f(n)}{g(n)}$ and the case when $\frac{f(n)}{g(n)} < \lim_{n\to\infty} \frac{f(n)}{g(n)}$.

Using the case when $\dfrac{f(n)}{g(n)} > \lim_{n\to\infty} \dfrac{f(n)}{g(n)}$, the absolute value function can be removed:

$$\forall(d \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{f(n)}{g(n)} - \lim_{n\to\infty} \frac{f(n)}{g(n)} < d) \tag{D.1.9}$$

1. By adding $\lim_{n\to\infty} \dfrac{f(n)}{g(n)}$ to both sides of the inequality in D.1.9:
$$\forall(d \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{f(n)}{g(n)} < d + \lim_{n\to\infty} \frac{f(n)}{g(n)}) \tag{D.1.10}$$
2. Using the assumption that $f(n)$ and $g(n)$ are always positive:
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq 0 \tag{D.1.11}$$
3. By using D.1.11, the right hand side of the inequality in D.1.10 can be shown to always be a positive real number:
$$d + \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \tag{D.1.12}$$
4. By instantiating d to be any positive real number and using D.1.12, it can be shown that there exists a positive real number $c = d + \lim_{n\to\infty} \dfrac{f(n)}{g(n)}$ such that:
$$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{f(n)}{g(n)} < c) \tag{D.1.13}$$
5. By multiplying both sides of D.1.13 by g(n) and disjunctive addition:
$$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(f(n) \leq cg(n)) \tag{D.1.14}$$

Using the case when $\dfrac{f(n)}{g(n)} < \lim_{n\to\infty} \dfrac{f(n)}{g(n)}$, the absolute value function can be removed:

$$\forall(d \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\lim_{n\to\infty} \frac{f(n)}{g(n)} - \frac{f(n)}{g(n)} < d) \tag{D.1.15}$$

1. By adding $\dfrac{2f(n)}{g(n)} - \lim_{n\to\infty}\dfrac{f(n)}{g(n)}$ to both sides of the inequality in D.1.15:

$$\forall(d \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{f(n)}{g(n)} < \frac{f(n)}{g(n)} + \frac{f(n)}{g(n)} + d - \lim_{n\to\infty}\frac{f(n)}{g(n)}) \tag{D.1.16}$$

2. By rearranging the right hand side of the inequality in D.1.16:

$$\forall(d \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{f(n)}{g(n)} < \frac{f(n)}{g(n)} + d - (\lim_{n\to\infty}\frac{f(n)}{g(n)} - \frac{f(n)}{g(n)}) \tag{D.1.17}$$

3. By using D.1.15, the right hand side of the inequality in D.1.17 can be shown to always be a positive real number:

$$\frac{f(n)}{g(n)} + d - (\lim_{n\to\infty}\frac{f(n)}{g(n)} - \frac{f(n)}{g(n)}) > 0 \tag{D.1.18}$$

4. By instantiating d to be any positive real number and using D.1.18, it can be shown that there exists a positive real number $c = \dfrac{f(n)}{g(n)} + d - (\lim_{n\to\infty}\dfrac{f(n)}{g(n)} - \dfrac{f(n)}{g(n)})$ such that:

$$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{f(n)}{g(n)} < c) \tag{D.1.19}$$

5. By multiplying both sides of D.1.19 by g(n) and disjunctive addition:

$$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(f(n) \leq cg(n)) \blacksquare \tag{D.1.20}$$

D.2 Big Omega $\leftrightarrow$ Limit Big Omega

The proof being shown has been separated into two parts since it is not easily reversible from one way to another.

D.2.1 Big O $\rightarrow$ Limit Big Omega

The condition for Big Omega is:

$$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(f(n) \geq cg(n)) \tag{D.2.1}$$

By dividing $g(n)$ on both sides of the inequality in D.2.1:

$$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{f(n)}{g(n)} \geq c) \tag{D.2.2}$$

By universal instantiation on D.2.2, n could be chosen arbitrarily larger than n_0 by letting n be a positive real number that approaches infinity:

$$\exists(c \in \mathbb{R}^+) \lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c \tag{D.2.3}$$

By using the fact that 0 is smaller than any positive real number:

$$\exists(c \in \mathbb{R}^+) \rightarrow (c > 0) \tag{D.2.4}$$

By combining D.2.3 and D.2.4:

$$\exists(c \in \mathbb{R}^+) \lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c > 0 \tag{D.2.5}$$

By simplifying D.2.5:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \ \blacksquare \tag{D.2.6}$$

D.2.2 Limit Big Omega $\rightarrow$ Big Omega

The Limit condition for Big Omega is:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \tag{D.2.7}$$

By introducing the convergence criterion:

$$\forall(d \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(|\frac{f(n)}{g(n)} - \lim_{n\to\infty} \frac{f(n)}{g(n)}| < d) \tag{D.2.8}$$

D.2.8 could be proven by separating it into two cases:
The case when $\dfrac{f(n)}{g(n)} > \lim_{n\to\infty} \dfrac{f(n)}{g(n)}$ and the case when $\dfrac{f(n)}{g(n)} < \lim_{n\to\infty} \dfrac{f(n)}{g(n)}$.

Using the case when $\dfrac{f(n)}{g(n)} > \lim_{n\to\infty} \dfrac{f(n)}{g(n)}$, the absolute value function can be removed:

$$\forall(d \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{f(n)}{g(n)} - \lim_{n\to\infty}\frac{f(n)}{g(n)} < d) \tag{D.2.9}$$

1. By adding $\dfrac{f(n)}{g(n)} - d$ to both sides of the inequality in D.2.9:

   $$\forall(d \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{2f(n)}{g(n)} - \lim_{n\to\infty}\frac{f(n)}{g(n)} - d < \frac{f(n)}{g(n)}) \tag{D.2.10}$$

2. By letting the left hand side of D.2.10 be positive:

   $$\frac{2f(n)}{g(n)} - \lim_{n\to\infty}\frac{f(n)}{g(n)} - d > 0 \tag{D.2.11}$$

3. By using D.2.9, the following can be shown to be positive:

   $$0 < \frac{f(n)}{g(n)} - \lim_{n\to\infty}\frac{f(n)}{g(n)} \tag{D.2.12}$$

4. By choosing d to be a real number that satisfies D.2.10 and D.2.11 (which is also positive because of D.2.12), it can be shown that there exists a positive real number $\dfrac{f(n)}{g(n)} - \lim_{n\to\infty}\dfrac{f(n)}{g(n)} < c = d < \dfrac{2f(n)}{g(n)} - \lim_{n\to\infty}\dfrac{f(n)}{g(n)}$ such that:

   $$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{f(n)}{g(n)} > c) \tag{D.2.13}$$

5. By multiplying both sides of D.1.13 by g(n) and disjunctive addition:

   $$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(f(n) \geq cg(n)) \tag{D.2.14}$$

Using the case when $\frac{f(n)}{g(n)} < \lim_{n\to\infty} \frac{f(n)}{g(n)}$, the absolute value function can be removed:

$$\forall(d \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\lim_{n\to\infty} \frac{f(n)}{g(n)} - \frac{f(n)}{g(n)} < d) \tag{D.2.15}$$

1. By adding $\frac{f(n)}{g(n)} - d$ to both sides of the inequality in D.2.15:

   $$\forall(d \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\lim_{n\to\infty} \frac{f(n)}{g(n)} - d < \frac{f(n)}{g(n)}) \tag{D.2.16}$$

2. By letting the left hand side of D.2.16 be positive:

   $$\lim_{n\to\infty} \frac{f(n)}{g(n)} - d > 0 \tag{D.2.17}$$

3. By using D.2.15, $\lim_{n\to\infty} \frac{f(n)}{g(n)} - \frac{f(n)}{g(n)}$ can be shown to be a positive number:

   $$\lim_{n\to\infty} \frac{f(n)}{g(n)} - \frac{f(n)}{g(n)} > 0 \tag{D.2.18}$$

4. By choosing d to be a real number that satisfies D.2.16 and D.2.17 (which is also positive because of D.2.18), it can be shown that there exists a positive real number $\lim_{n\to\infty} \frac{f(n)}{g(n)} - \frac{f(n)}{g(n)} < c = d < \lim_{n\to\infty} \frac{f(n)}{g(n)}$ such that:

   $$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{f(n)}{g(n)} < c) \tag{D.2.19}$$

5. By multiplying both sides of D.2.19 by g(n) and disjunctive addition:

   $$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(f(n) \leq cg(n)) \blacksquare \tag{D.2.20}$$

D.3 Big Theta $\leftrightarrow$ Limit Big Theta

The condition for Big Theta is:

$$O(n) \wedge \Omega(n) \tag{D.3.1}$$

By using D.1 and D.2:

$$(\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty) \wedge (\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0) \tag{D.3.2}$$

By simplifying D.3.2:

$$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \ \blacksquare \tag{D.3.3}$$

D.4 Little O $\leftrightarrow$ Limit Little O

The condition for Little O is:

$$O(n) \wedge \neg\Theta(n) \tag{D.4.1}$$

By using D.1 and D.3:

$$(\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty) \wedge \neg(0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty) \tag{D.4.2}$$

By expanding the D.4.2:

$$(\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty) \wedge \neg(0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} \wedge \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty) \tag{D.4.3}$$

By distributing the negation operator in D.4.3:

$$(\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty) \wedge (0 \geq \lim_{n\to\infty} \frac{f(n)}{g(n)} \vee \neg \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty) \tag{D.4.4}$$

By distributing the conjunction operation over the disjunctions in D.4.4:

$$((\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty) \wedge (0 \geq \lim_{n\to\infty} \frac{f(n)}{g(n)})) \vee ((\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty) \wedge (\neg \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty)) \tag{D.4.5}$$

By removing the contradictory disjunction in D.4.5:

$$(\lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty) \wedge (0 \geq \lim_{n\to\infty}\frac{f(n)}{g(n)}) \tag{D.4.6}$$

Using the assumption that $f(n)$ and $g(n)$ are always positive in D.4.6:

$$(\lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty) \wedge (0 = \lim_{n\to\infty}\frac{f(n)}{g(n)}) \tag{D.4.7}$$

By simplifying D.4.7:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0 \ \blacksquare \tag{D.4.8}$$

D.5 Little Omega ↔ Limit Little Omega

The condition for Little Omega is:

$$\Omega(n) \wedge \neg\Theta(n) \tag{D.5.1}$$

By using D.2 and D.3:

$$(\lim_{n\to\infty}\frac{f(n)}{g(n)} > 0) \wedge \neg(0 < \lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty) \tag{D.5.2}$$

By expanding the D.5.2:

$$(\lim_{n\to\infty}\frac{f(n)}{g(n)} > 0) \wedge \neg(0 < \lim_{n\to\infty}\frac{f(n)}{g(n)} \wedge \lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty) \tag{D.5.3}$$

By distributing the negation operator in D.5.3:

$$(\lim_{n\to\infty}\frac{f(n)}{g(n)} > 0) \wedge (\neg 0 < \lim_{n\to\infty}\frac{f(n)}{g(n)} \vee \lim_{n\to\infty}\frac{f(n)}{g(n)} \geq \infty) \tag{D.5.4}$$

By distributing the conjunction operation over the disjunctions in D.5.4:

$$((\lim_{n\to\infty}\frac{f(n)}{g(n)} > 0) \wedge \neg(0 < \lim_{n\to\infty}\frac{f(n)}{g(n)})) \vee ((\lim_{n\to\infty}\frac{f(n)}{g(n)} > 0) \wedge (\lim_{n\to\infty}\frac{f(n)}{g(n)} \geq \infty)) \tag{D.5.5}$$

By removing the contradictory disjunction in D.5.5:

$$(\lim_{n\to\infty}\frac{f(n)}{g(n)} > 0) \wedge (\lim_{n\to\infty}\frac{f(n)}{g(n)} \geq \infty) \tag{D.5.6}$$

By utilizing the property of infinity where nothing is greater than itself in D.5.6:

$$(\lim_{n\to\infty}\frac{f(n)}{g(n)} > 0) \wedge (\lim_{n\to\infty}\frac{f(n)}{g(n)} = \infty) \tag{D.5.7}$$

By simplifying D.5.7:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \infty \blacksquare \tag{D.5.8}$$

Appendix E

User's Manual

E.1 Installation

The folder containing the dependencies consists of three parts. One is the WinPython 2.7.10.3 32-bit installer which contains Python 2.7.10 along with libraries necessary for generating the graphs. Another is the PyGTK 2.24.2 for 32-bit Windows and Python 2.7 (all-in-one version). This contains the GUI libraries used in the prototype. Lastly, it contains a copy of the editbin program provided by Microsoft which is used to increase the stack allocation of Windows to Python. FULL_INSTALLATION.bat could be used to automate some parts of the installation process. One must follow the instructions being displayed in the console carefully for each step to avoid any problems.

After is has successfully installed the dependencies, it will create a file named path.txt outside the dependency folder. This text file will be used by the RUN_PROTOTYPE.bat. This batch file must be in the folder containing the source files to work. This allows users to run the prototype without having to add the Python folder in the system's environmental variables and manually inputting commands the command line.

E.2 Input

The input of our prototype is a Python file with the implemented algorithm. The algorithm must follow these rules:

- The main function to be called in the algorithm should be defined as f.
- The algorithm must only have 1 variable as parameter.
- The algorithm may call other algorithms provided that they are defined within the same file.
- Do-while loops are not accepted since there are no do-while loops in Python. However do-while loops can be simulated using while loops with additional condition checking.
- The ++ keyword is not accepted since it is not accepted by in Python. However, ++ can be simulated by using +=1.
- The elif keyword is not accepted due to a difficulty in parsing. It could be replaced by nesting an if statement under an existing else statement.
- The """ keyword is not accepted due to a difficulty in parsing. It could be replaced by using the keyword # for each line of comment.

A sample algorithm could be observed in Listing E.1.

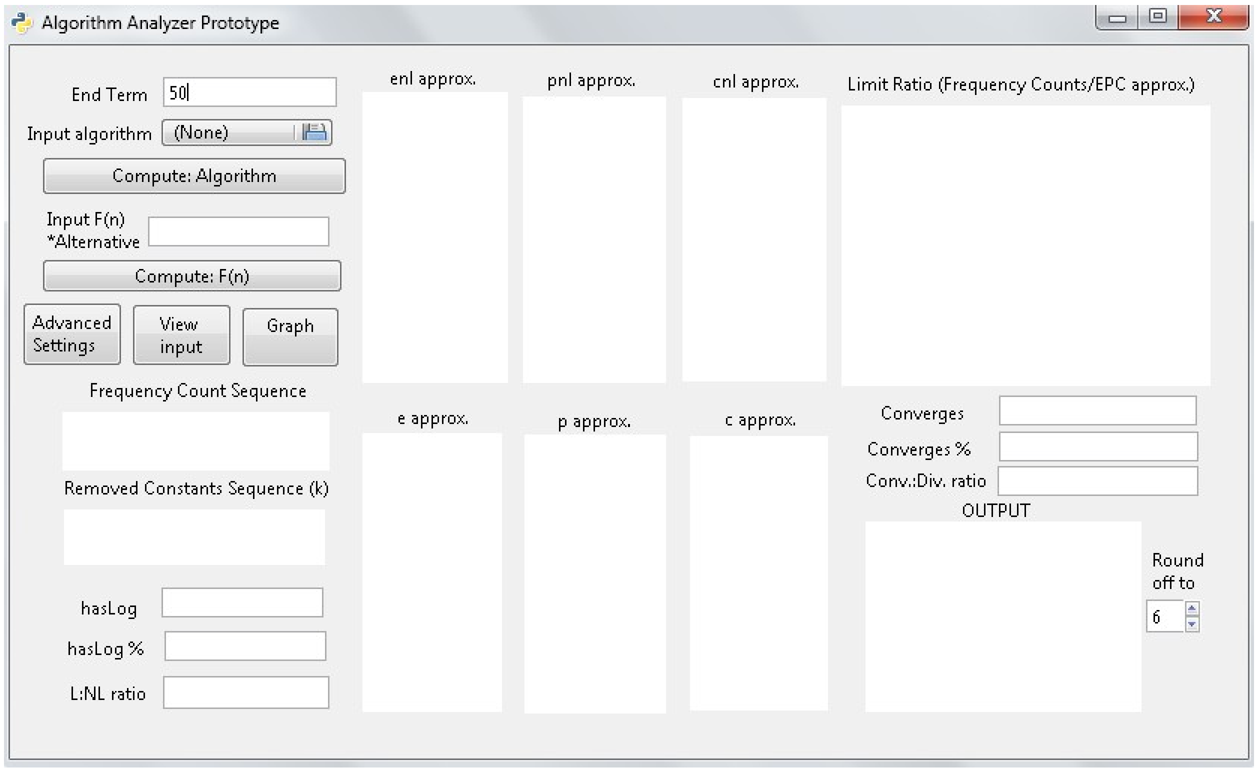
Listing E.1: Sample input ack1.py

```
def f(n)
    ack(1,n)

def ack(m,n):
    if m == 0
        return n+1
    if n == 0
        return ack(m-1, 1)
    return ack(m-1, ack(m, n-1))
```

E.3 User Interface

The prototype could be ran through a batch file which opens the command line, ensures that the Python folder is included in the system path, and runs the main program.

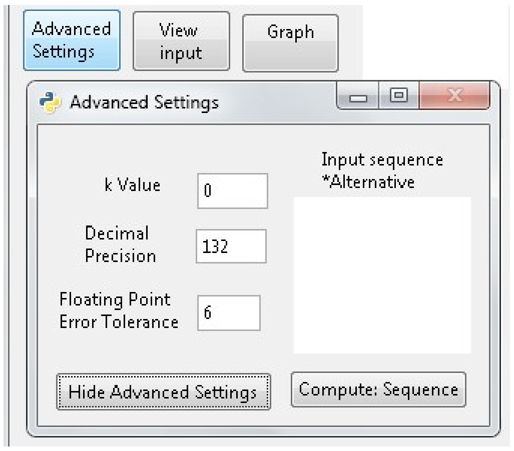


End Term: 50

Round off to: 6

E.3.1 Preprocessing and Frequency Counts Generation

After all the necessary parameters have been set, the user must click the "Compute" button to initiate the computations. The unessential parameters could be accessed by clicking the Advanced Settings button.

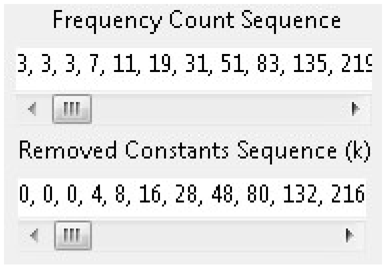


k Value: 0
Decimal Precision: 132
Floating Point Error Tolerance: 6

The command line will serve as an indicator of the current status of the prototype and it will also be used in case the user chooses to preemptively stop the term generation process.

```
Augmenting code...
Generating 0th Frequency Count measurement; use Ctrl+C to stop.
Generating 1th Frequency Count measurement; use Ctrl+C to stop.
Generating 2th Frequency Count measurement; use Ctrl+C to stop.
Generating 3th Frequency Count measurement; use Ctrl+C to stop.
Generating 4th Frequency Count measurement; use Ctrl+C to stop.
Generating 5th Frequency Count measurement; use Ctrl+C to stop.
Generating 6th Frequency Count measurement; use Ctrl+C to stop.
Generating 7th Frequency Count measurement; use Ctrl+C to stop.
Generating 8th Frequency Count measurement; use Ctrl+C to stop.
Generating 9th Frequency Count measurement; use Ctrl+C to stop.
Generating 10th Frequency Count measurement; use Ctrl+C to stop.
Generating 11th Frequency Count measurement; use Ctrl+C to stop.
Generating 12th Frequency Count measurement; use Ctrl+C to stop.
Generating 13th Frequency Count measurement; use Ctrl+C to stop.
Generating 14th Frequency Count measurement; use Ctrl+C to stop.
Generating 15th Frequency Count measurement; use Ctrl+C to stop.
```

After all the terms have been generated, the sequence of frequency counts from running the input algorithm on various values will be displayed after the generation of terms have been finished, alongside with the sequence of frequency counts that have been modified by the removal of constants described in Section 5.4.

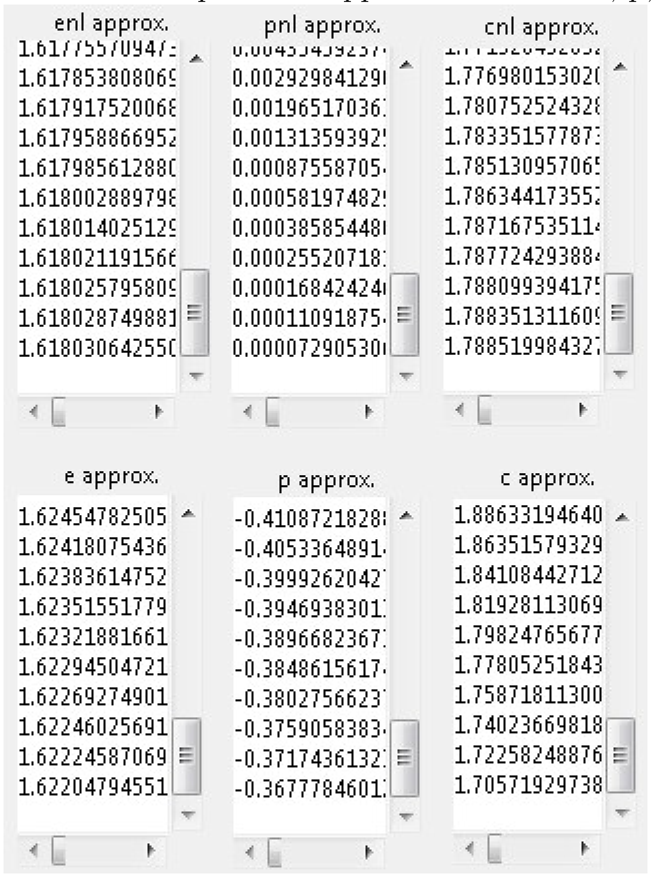


```
Frequency Count Sequence
3, 3, 3, 7, 11, 19, 31, 51, 83, 135, 219

Removed Constants Sequence (k)
0, 0, 0, 4, 8, 16, 28, 48, 80, 132, 216
```

E.3.2 Calculation of the Asymptotic Behavior

The output will contain the sequence of approximations for e, p, and c values.

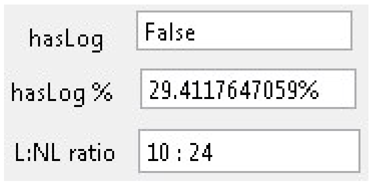


```
enl approx.        pnl approx.        cnl approx.
[illegible]        [illegible]        [illegible]
1.617853808069     0.002929841290     1.776980153020
1.617917520068     0.001965170361     1.780752524328
1.617958866952     0.001313593925     1.783351577873
1.617985612880     0.000875587054     1.785130957065
1.618002889798     0.000581974829     1.786344173552
1.618014025129     0.000385854480     1.787167535114
1.618021191566     0.000255207181     1.787724293884
1.618025795809     0.000168424240     1.788099394175
1.618028749881     0.000110918754     1.788351311609
1.618030642550     0.000072905300     1.788519984327

e approx.          p approx.          c approx.
1.62454782505      -0.41087218288     1.88633194640
1.62418075436      -0.40533648914     1.86351579329
1.62383614752      -0.39992620427     1.84108442712
1.62351551779      -0.39469383011     1.81928113069
1.62321881661      -0.38966823671     1.79824765677
1.62294504721      -0.38486156174     1.77805251843
1.62269274901      -0.38027566237     1.75871811300
1.62246025691      -0.37590583834     1.74023669818
1.62224587069      -0.37174361321     1.72258248876
1.62204794551      -0.36777846011     1.70571929738
```

E.3.3 Verifying the Presence of an ln(n) Factor

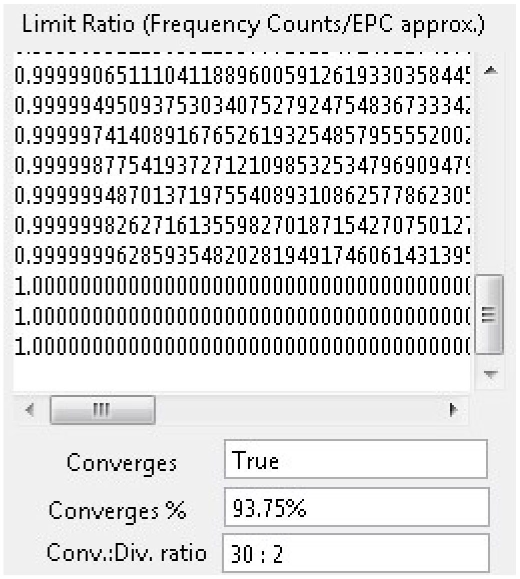
The program will now determine the presence of an ln(n) factor by checking which case is closer to the actual frequency count for each term of the input sequence. The interface will show the number of terms where the log case is a closer approximation than no log case and vice versa. It displays the percentage of the terms where the log case is a closer approximation out of the total number of terms.



hasLog	False
hasLog %	29.4117647059%
L:NL ratio	10 : 24

E.3.4 Verifying the Asymptotic Equivalence

The program determines if the computed e, p, c, and hasLog values is asymptotic to the sequence of frequency counts. In this case, it will display the ratio of frequency count over the computed asymptotic function for all terms. It will display the percentage of the terms where the ratio converges out of all the total number of terms.



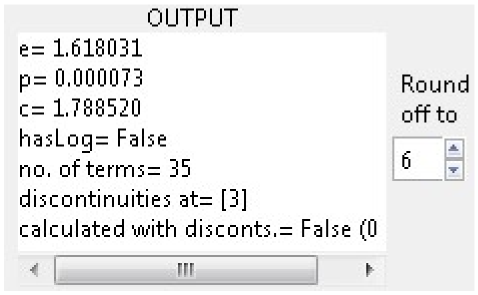
Limit Ratio (Frequency Counts/EPC approx.)

```
0.9999906511104118896005912619330358445
0.9999949509375303407527924754836733342
0.9999974140891676526193254857955552002
0.9999987754193727121098532534796909479
0.9999994870137197554089310862577862305
0.9999998262716135598270187154270750127
0.9999999628593548202819491746061431395
1.0000000000000000000000000000000000000
1.0000000000000000000000000000000000000
1.0000000000000000000000000000000000000
```

Converges	True
Converges %	93.75%
Conv.:Div. ratio	30 : 2

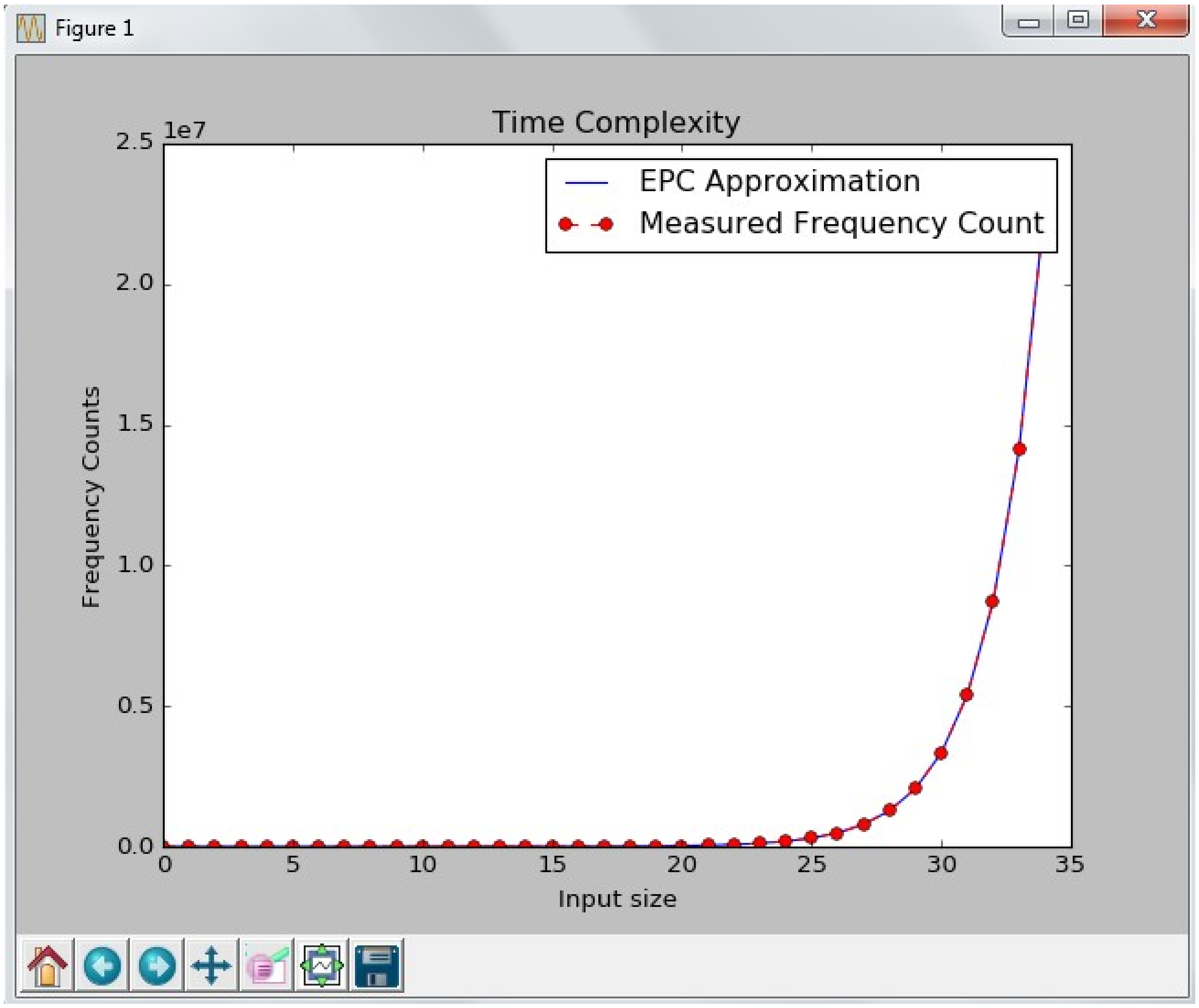
E.3.5 Final Output

The program will display the summary of all computations that have been done; the computed e, p, c, and hasLog values. It will also display the number of terms used in the computations, detected discontinuities, and if the approximations are calculated around discontinuities. It also has a feature that allows the user to tweak the number of decimal places to round off in the output.



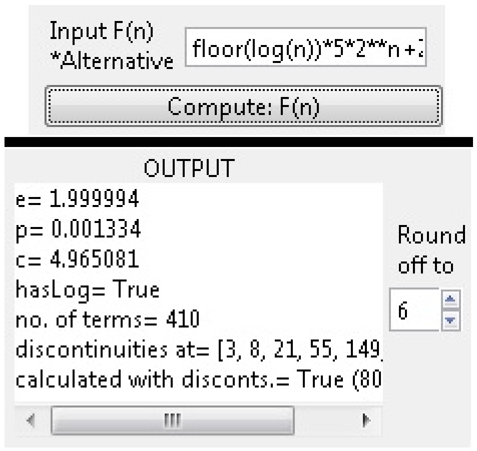
```
e= 1.618031
p= 0.000073
c= 1.788520
hasLog= False
no. of terms= 35
discontinuities at= [3]
calculated with disconts.= False (0
```

The option to graph the measured frequency counts to the calculated answer could be accessed by clicking the ”Graph” button. It will create a new window showing the graph together with several buttons at the bottom of the window. These buttons will give users the ability to zoom in/out, move to different parts of the graph, and other options that will have an aesthetic effect on the graph.

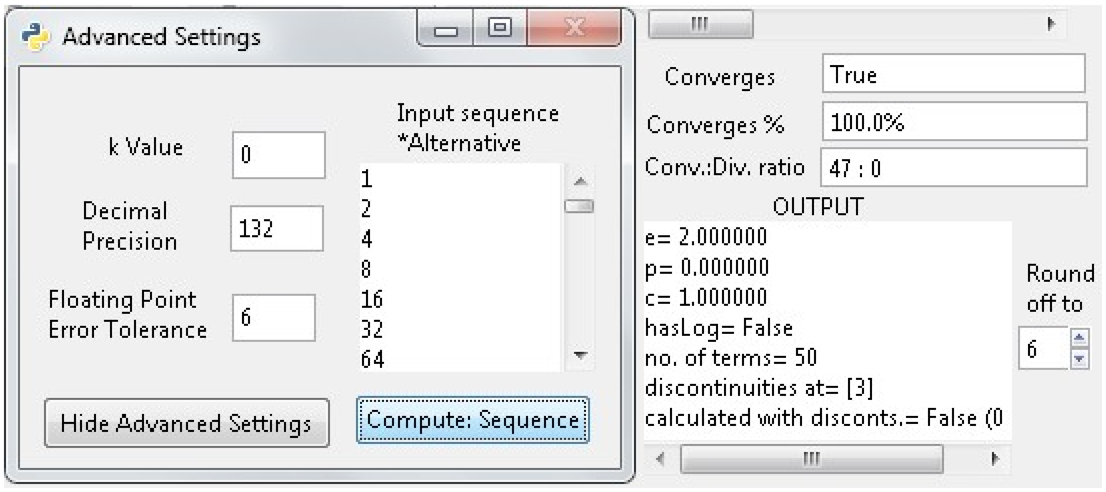


E.3.6 Alternative Data Generating Features

The functions or sequences could be used as alternatives to using algorithm files to generate terms. Users could input functions in terms of i that are valid in the Standard and Math libraries of Python like "log(i)" or "pow(i,0.5)+5" or "i**2+10*i/3.0" (** is exponentiation in Python). Users could also input a sequence of real numbers.



```
e= 1.999994
p= 0.001334
c= 4.965081
hasLog= True
no. of terms= 410
discontinuities at= [3, 8, 21, 55, 149
calculated with disconts.= True (80
```



k Value: 0
Decimal Precision: 132
Floating Point Error Tolerance: 6

Input sequence *Alternative

```
1
2
4
8
16
32
64
```

Converges: True
Converges %: 100.0%
Conv.:Div. ratio: 47 : 0

```
e= 2.000000
p= 0.000000
c= 1.000000
hasLog= False
no. of terms= 50
discontinuities at= [3]
calculated with disconts.= False (0
```

Appendix F

Output Graphs

This section contains the compilation of the generated output graphs of the experiments done in Chapter 6 and the arrangement follows the order in Table 6.1

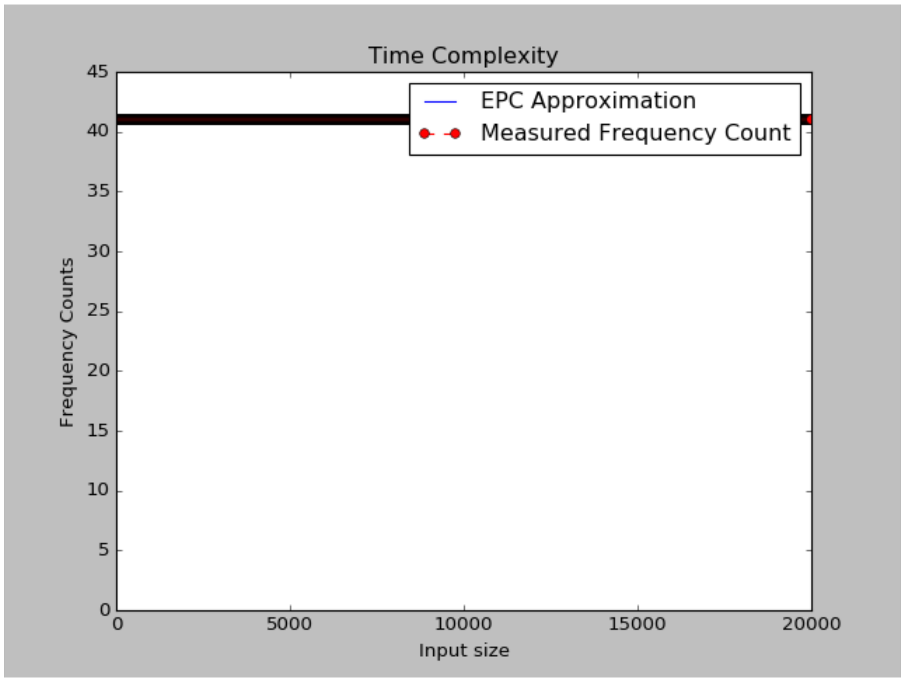


Figure F.1: Graph of Iter1(n)

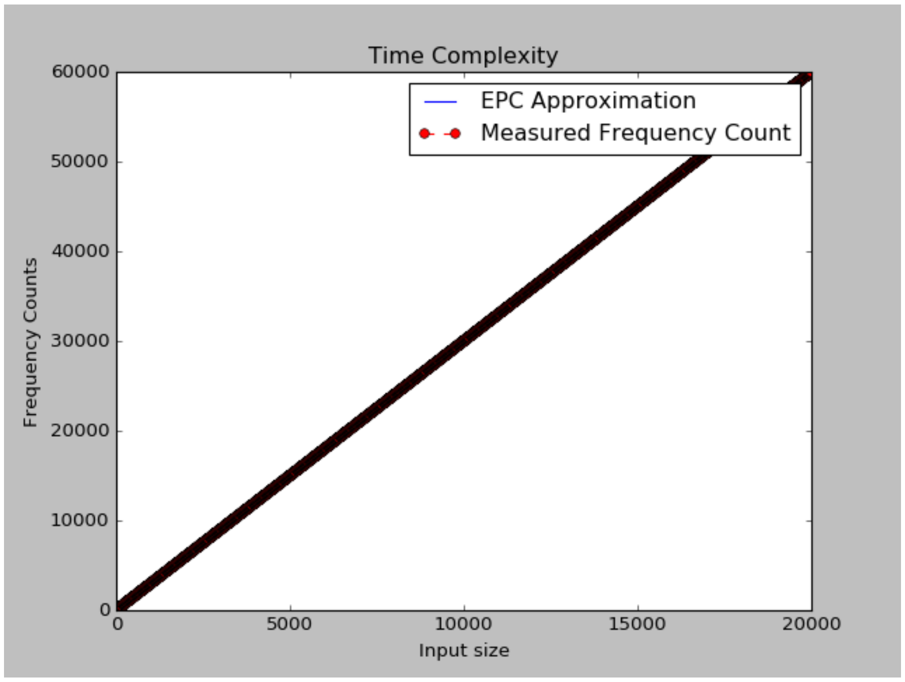


Figure F.2: Graph of Iter2(n)

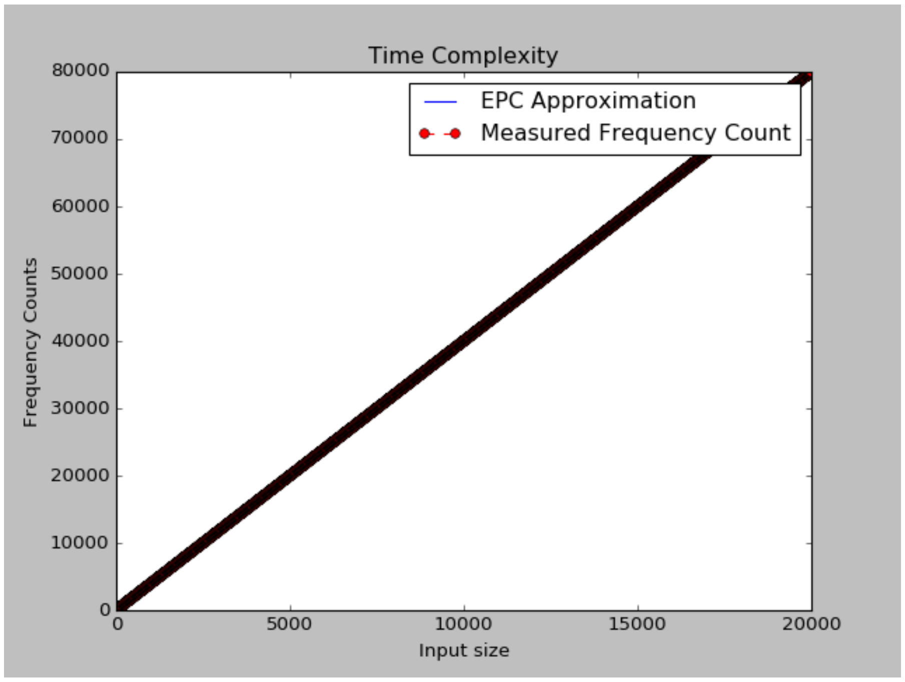


Figure F.3: Graph of Iter3(n)

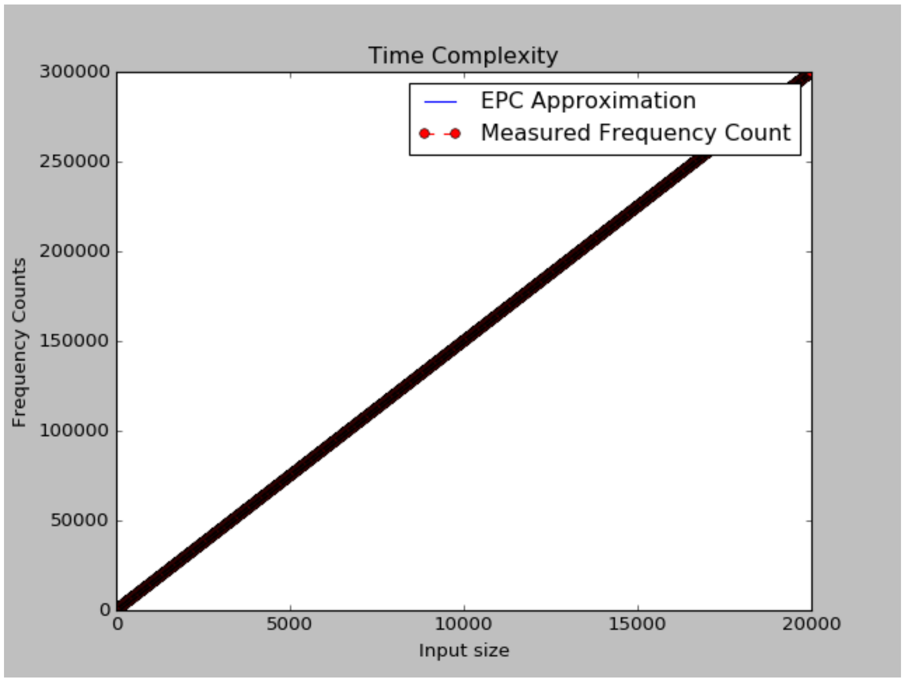


Figure F.4: Graph of Iter4(n)

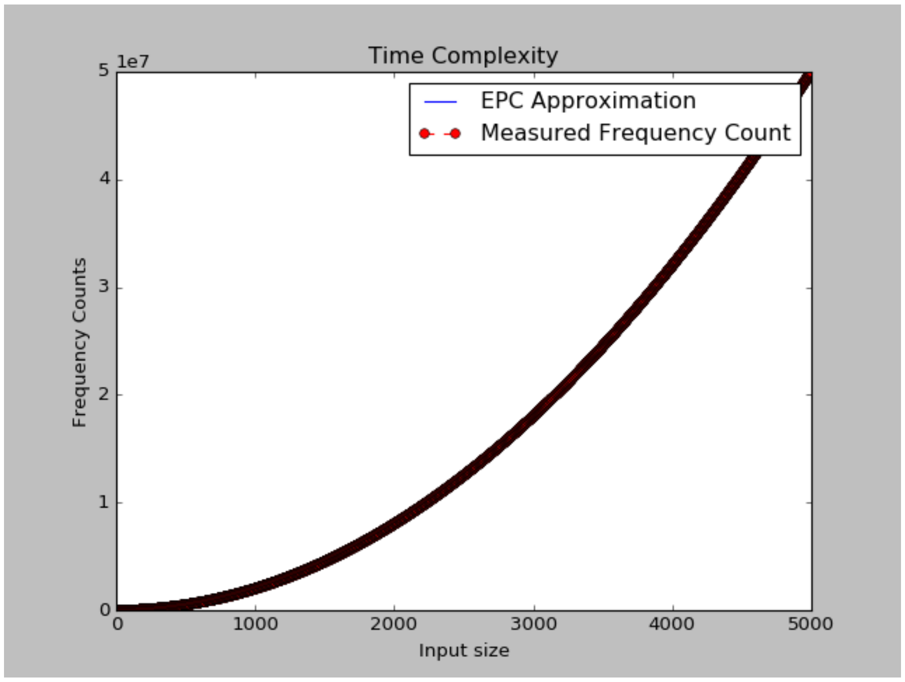


Figure F.5: Graph of Iter5(n)

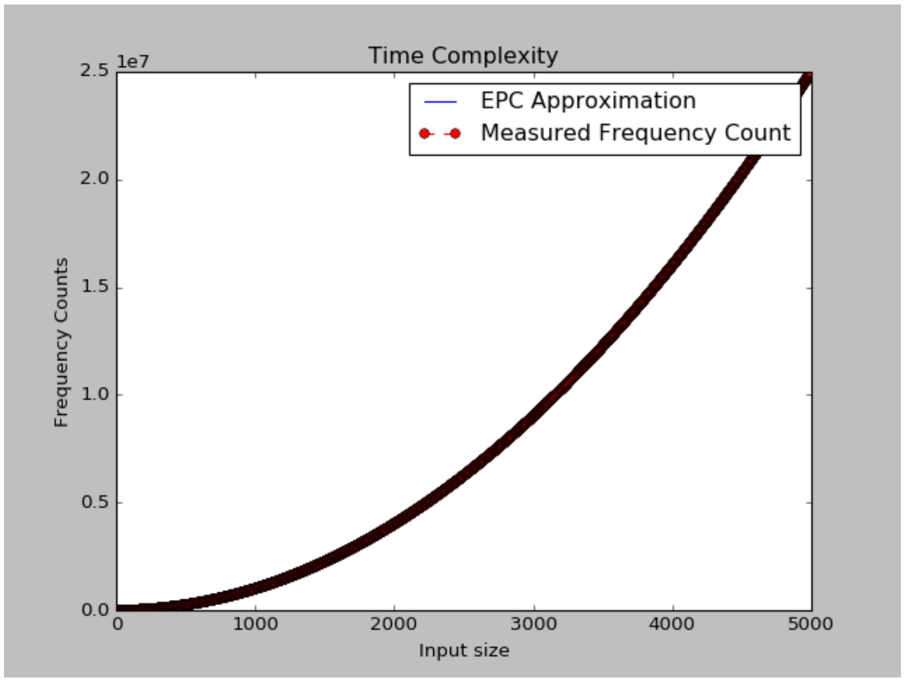


Figure F.6: Graph of Iter6(n)

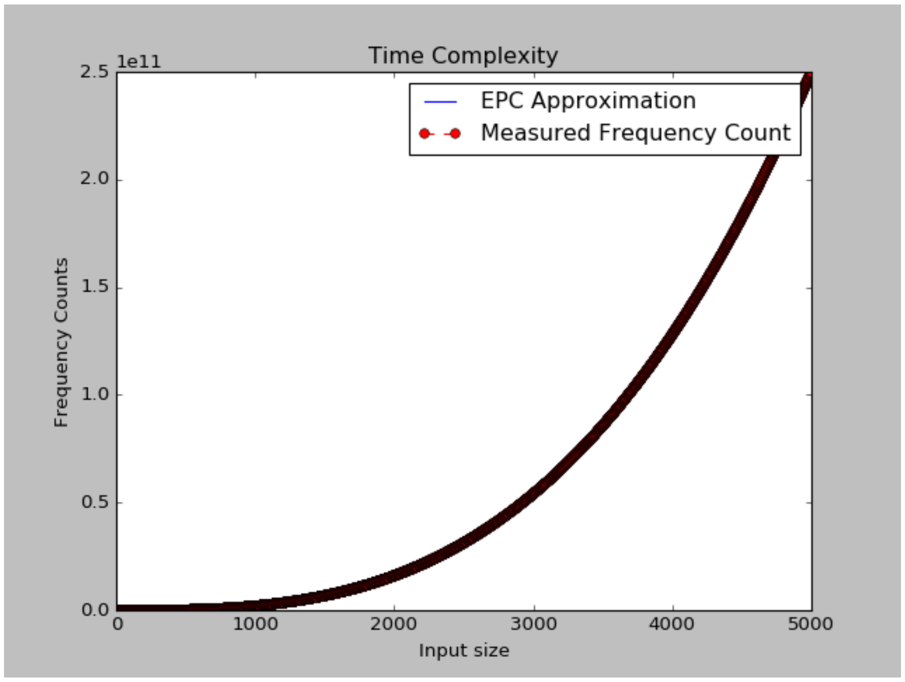


Figure F.7: Graph of Iter7(n)

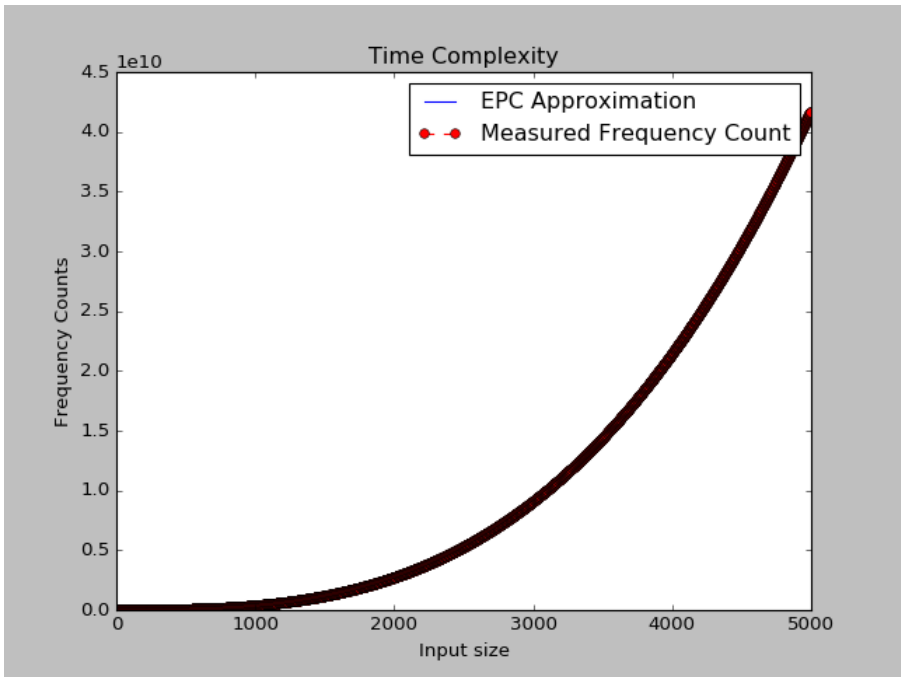


Figure F.8: Graph of Iter8(n)

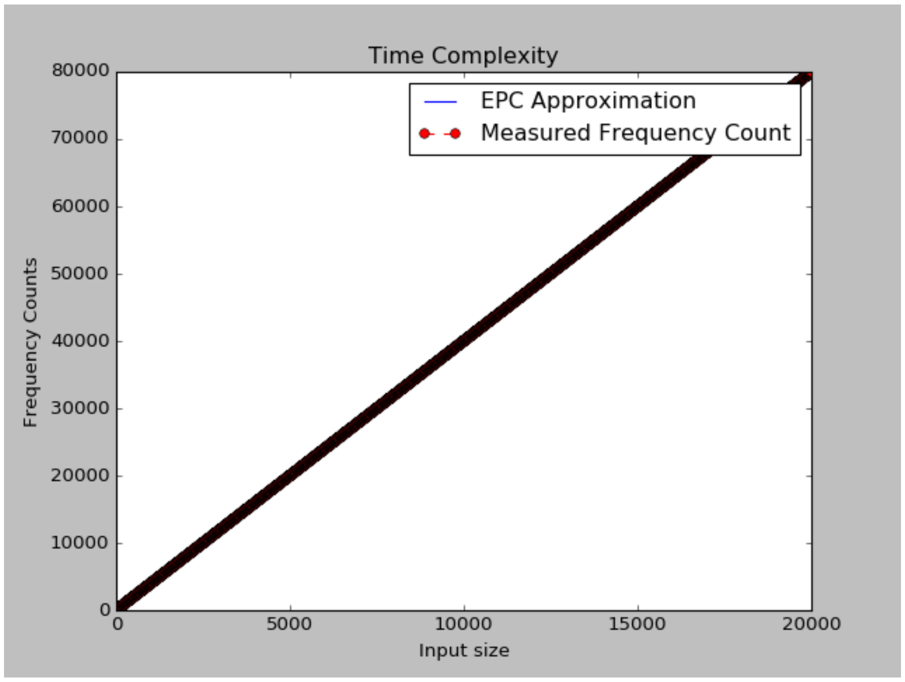


Figure F.9: Graph of IterFibo(n)

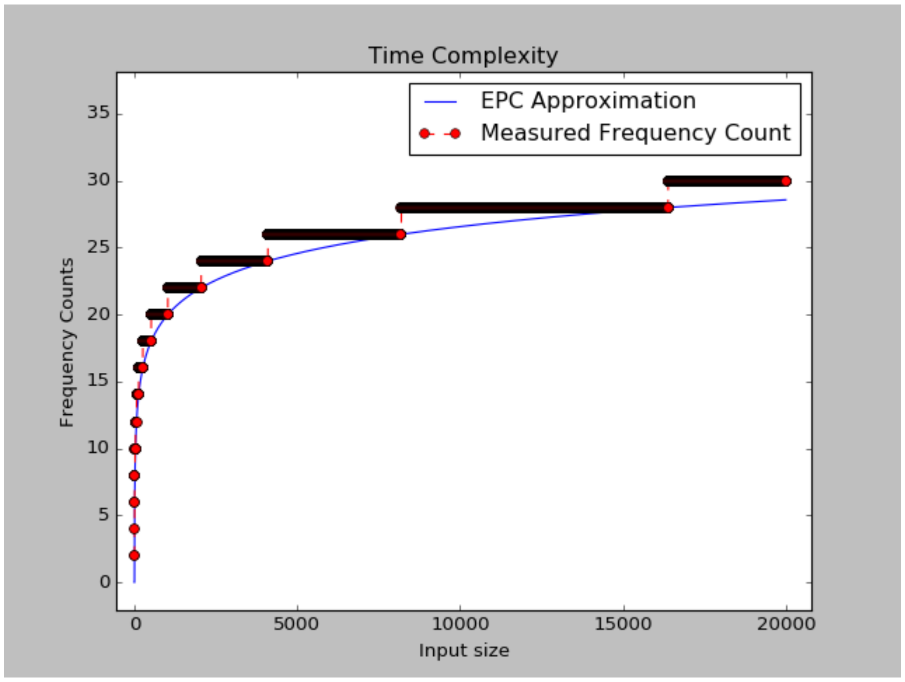


Figure F.10: Graph of Log2(n)

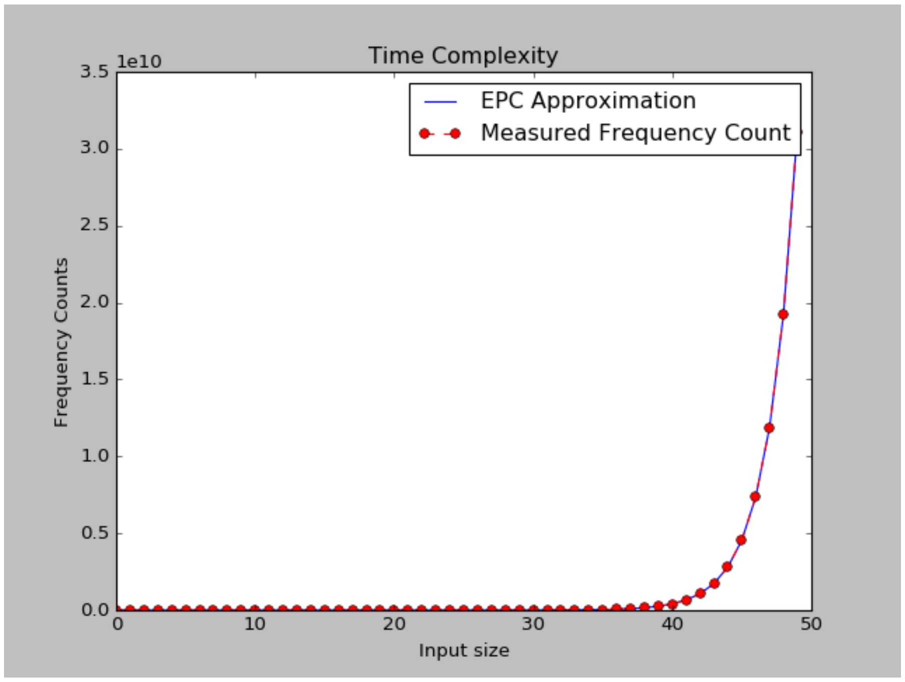


Figure F.11: Graph of Fibo(n)

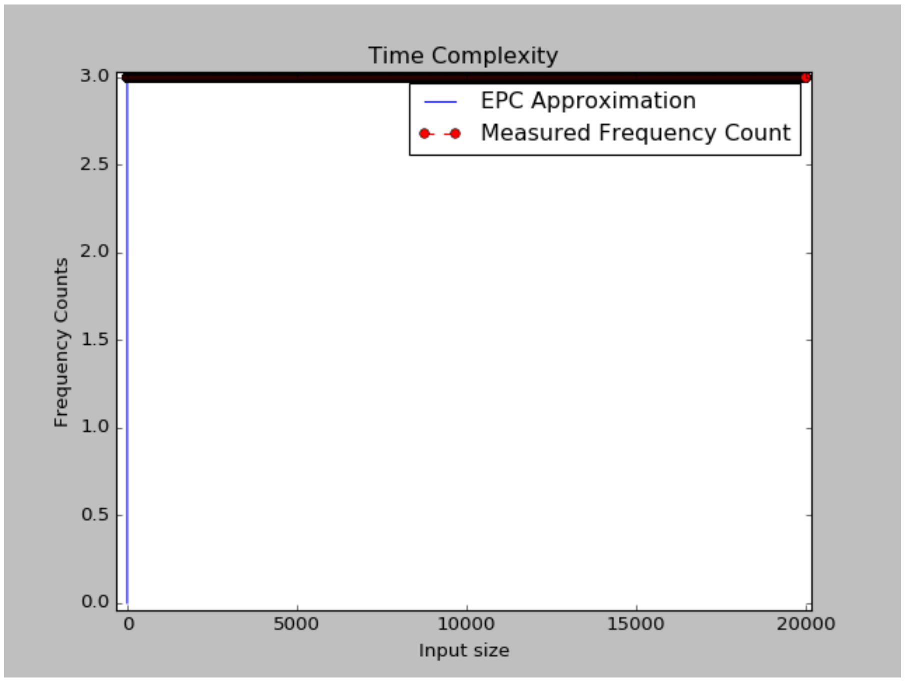


Figure F.12: Graph of Ack(0, n)

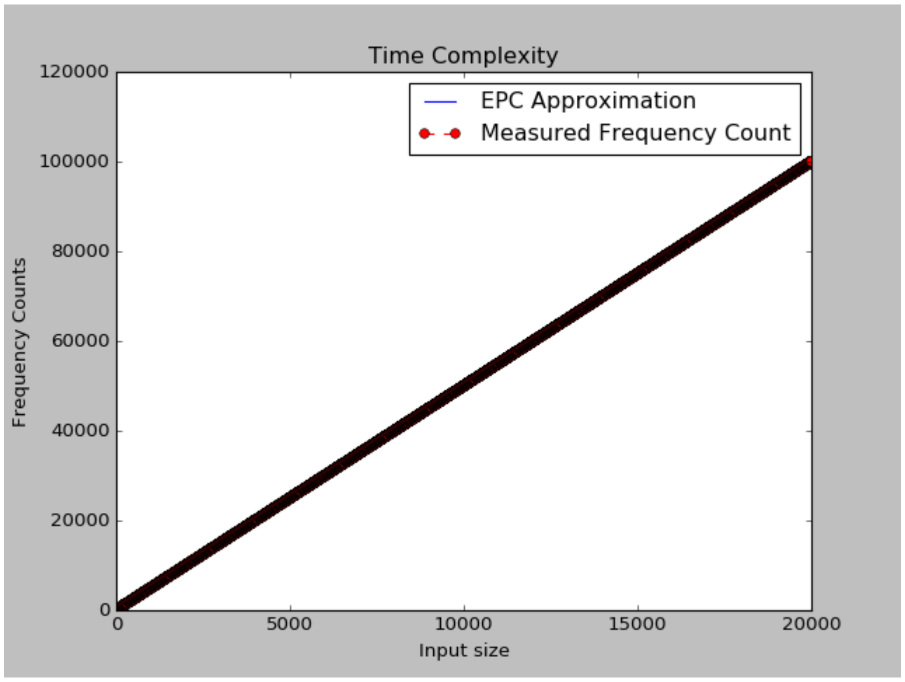


Figure F.13: Graph of Ack(1, n)

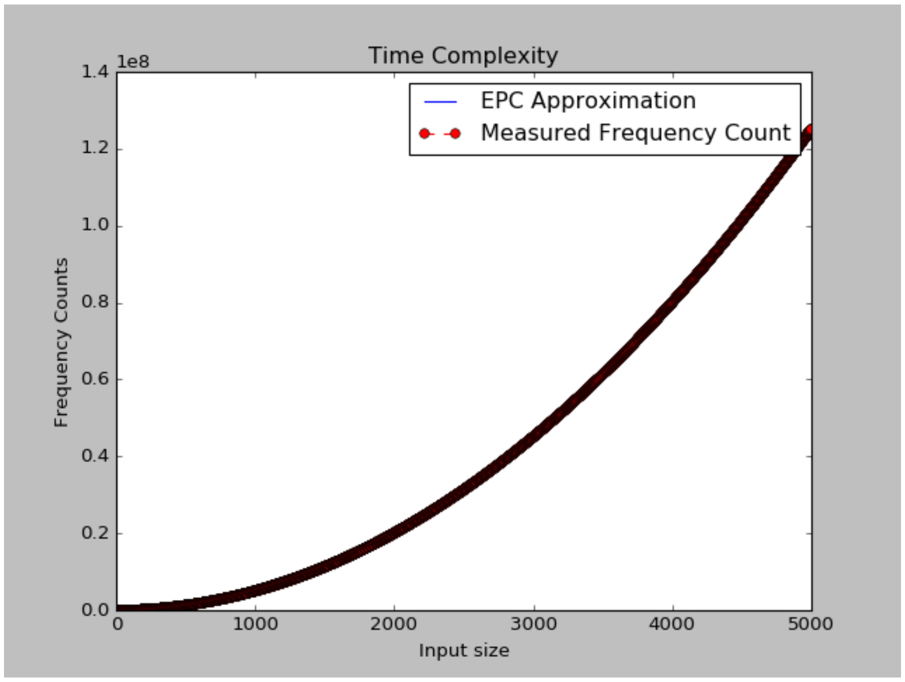


Figure F.14: Graph of Ack(2, n)

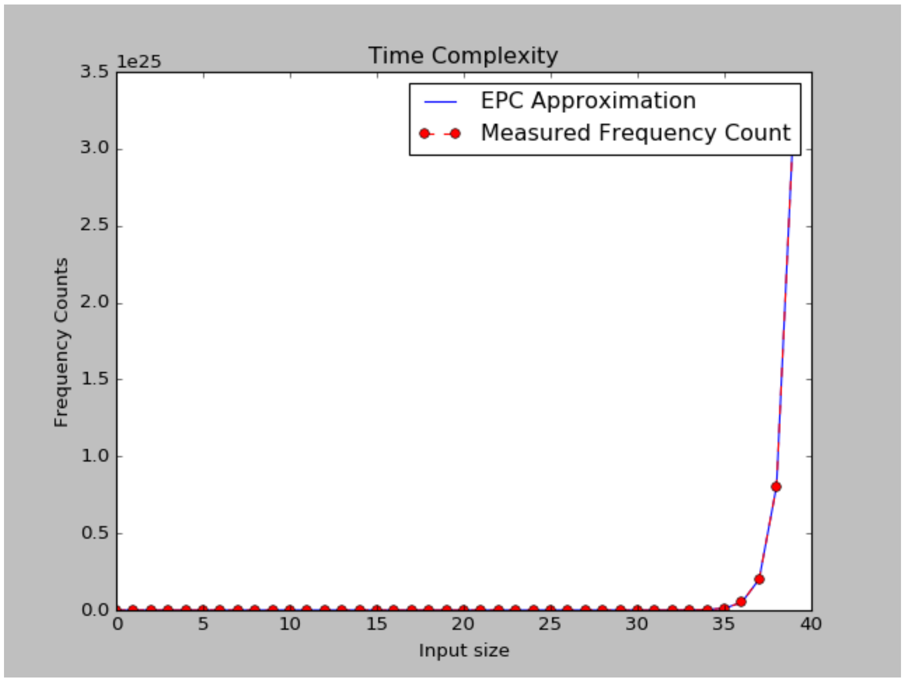


Figure F.15: Graph of Ack(3, n)

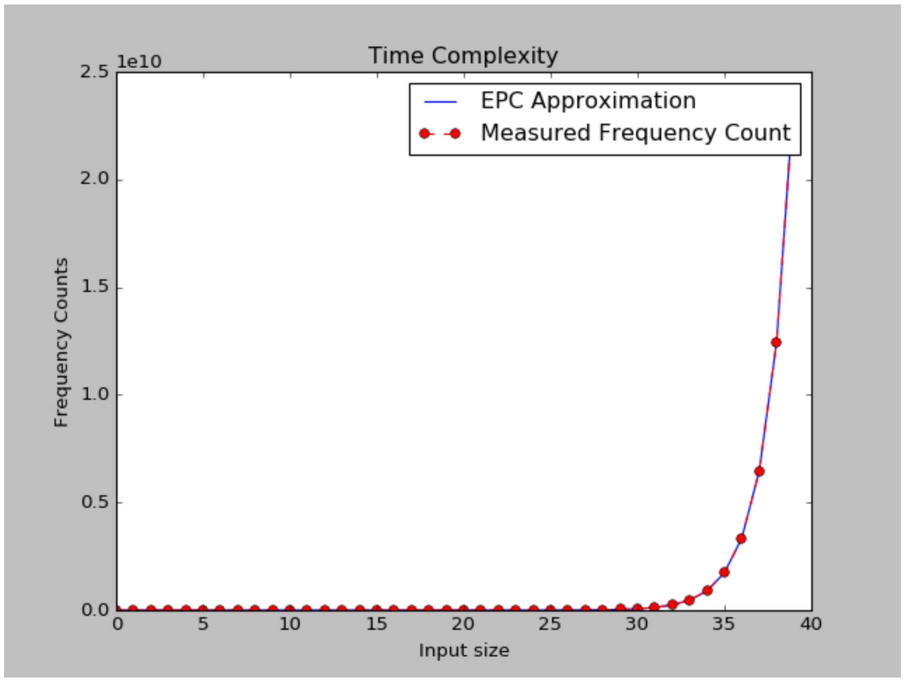


Figure F.16: Graph of QuinticFibo(n)

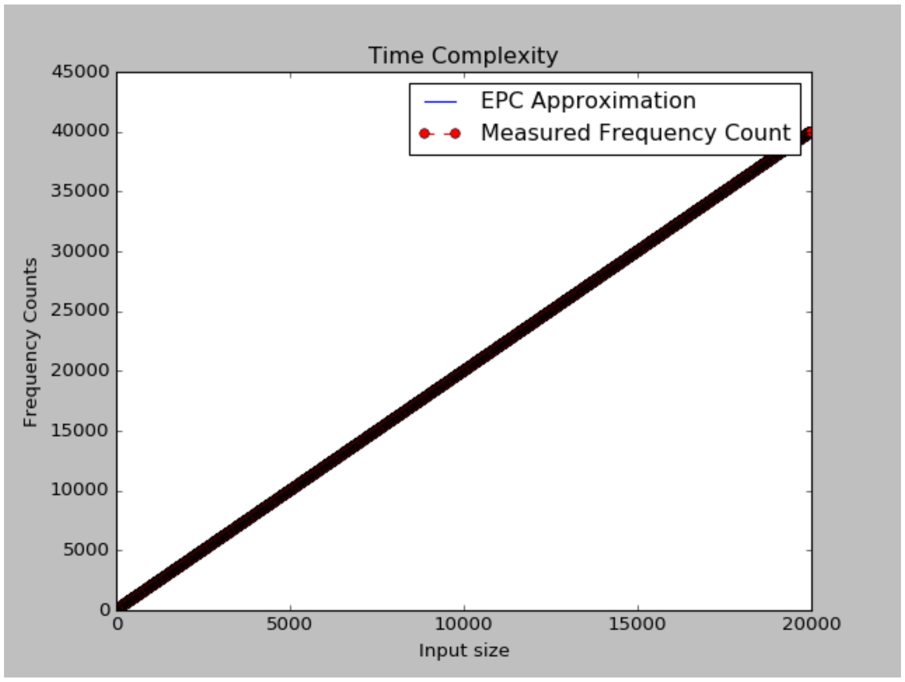


Figure F.17: Graph of LinearSearch(n)

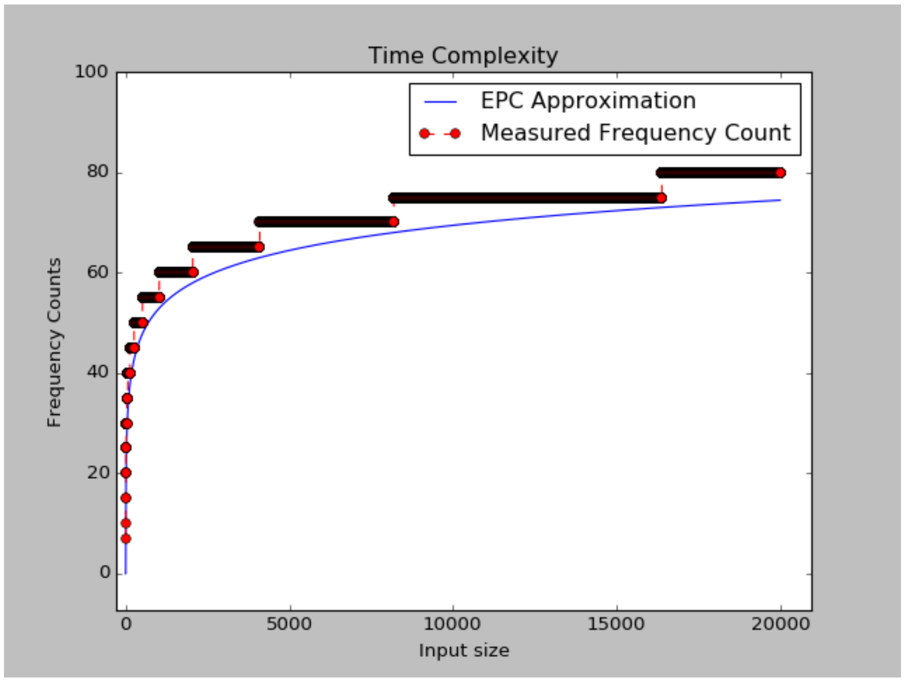


Figure F.18: Graph of BinarySearch(n)

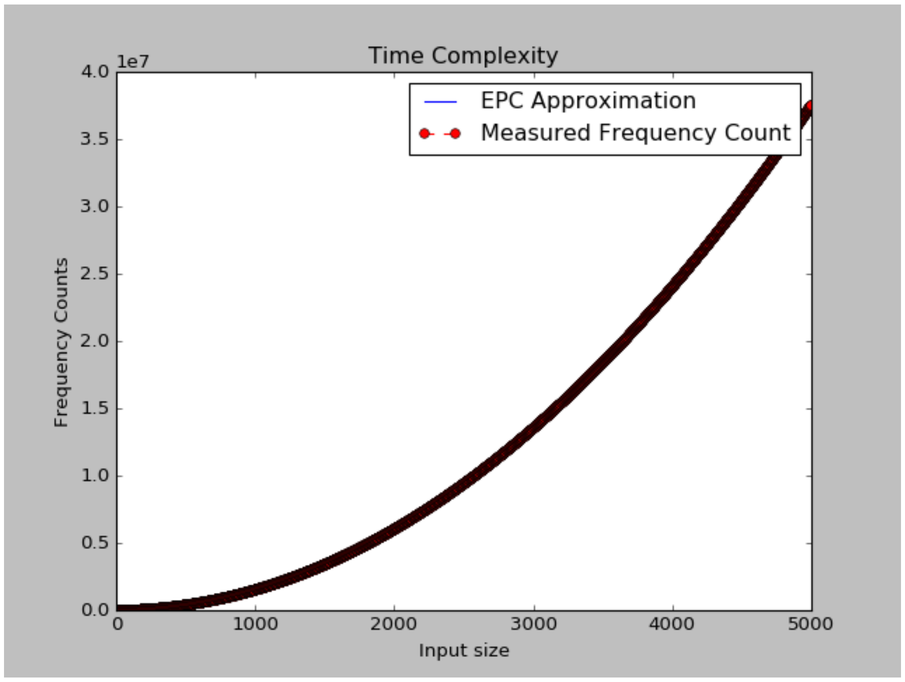


Figure F.19: Graph of Bubble_sort(n)

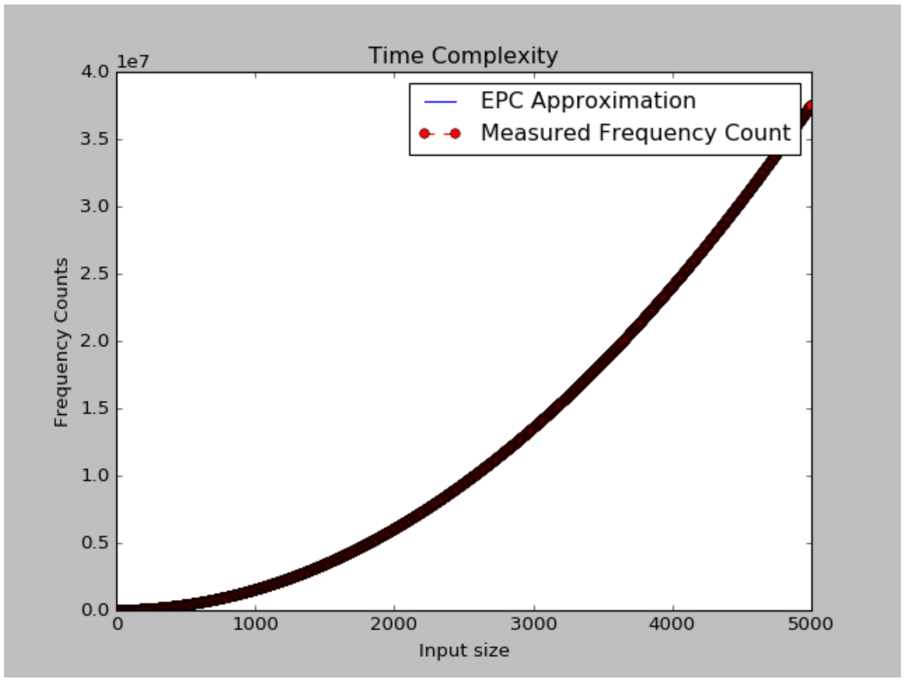


Figure F.20: Graph of Insertion_sort(n)

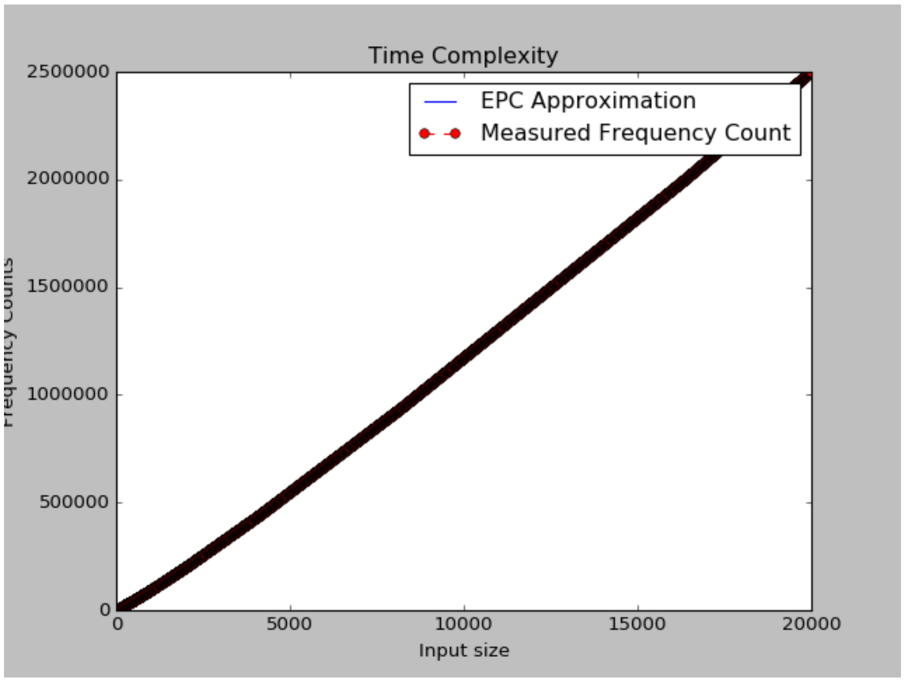


Figure F.21: Graph of Merge_sort(n)

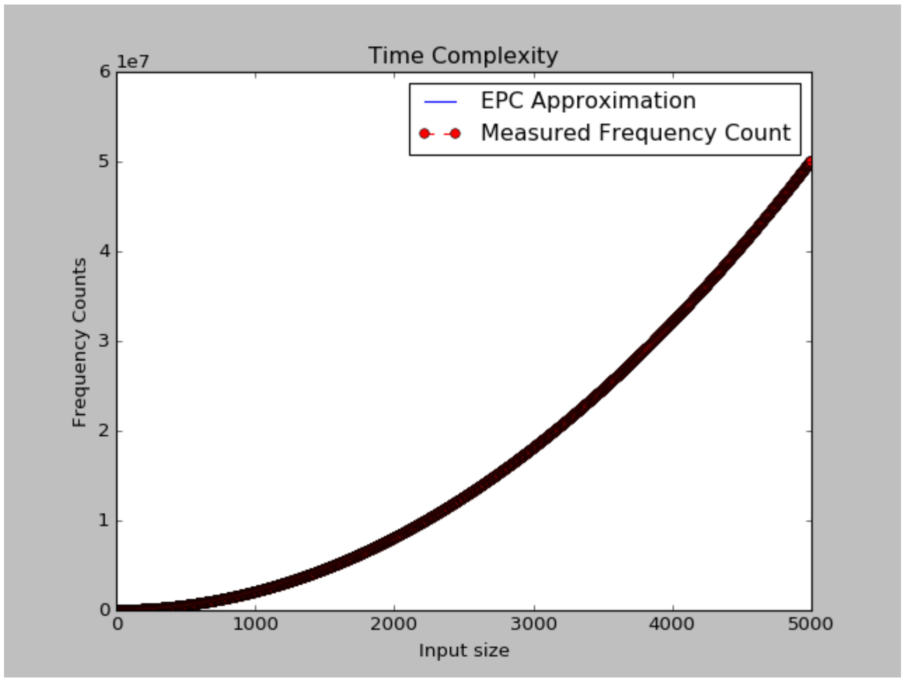


Figure F.22: Graph of Quick_sort(n)

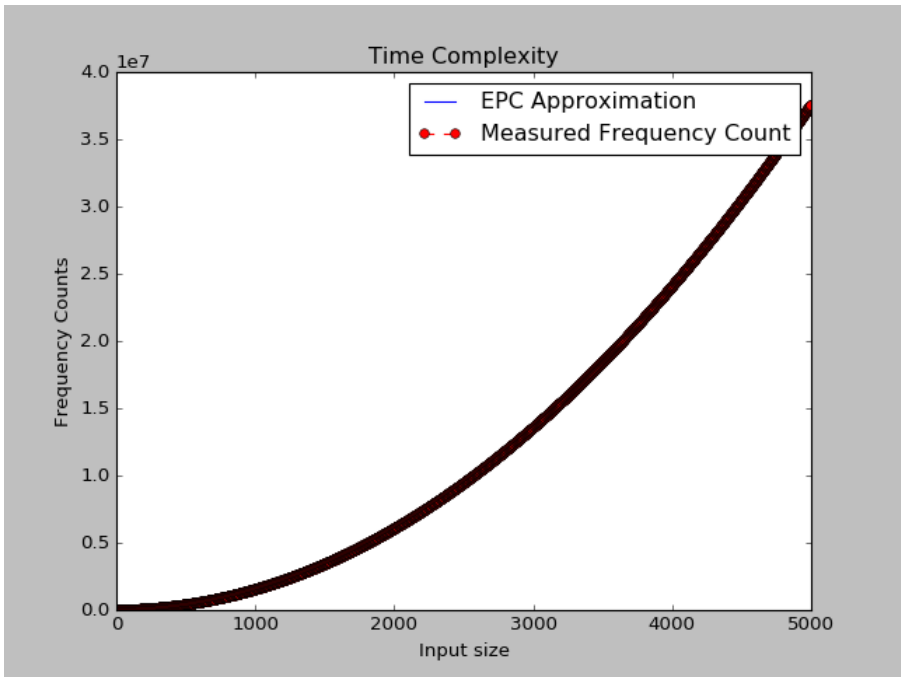


Figure F.23: Graph of Selection_sort(n)

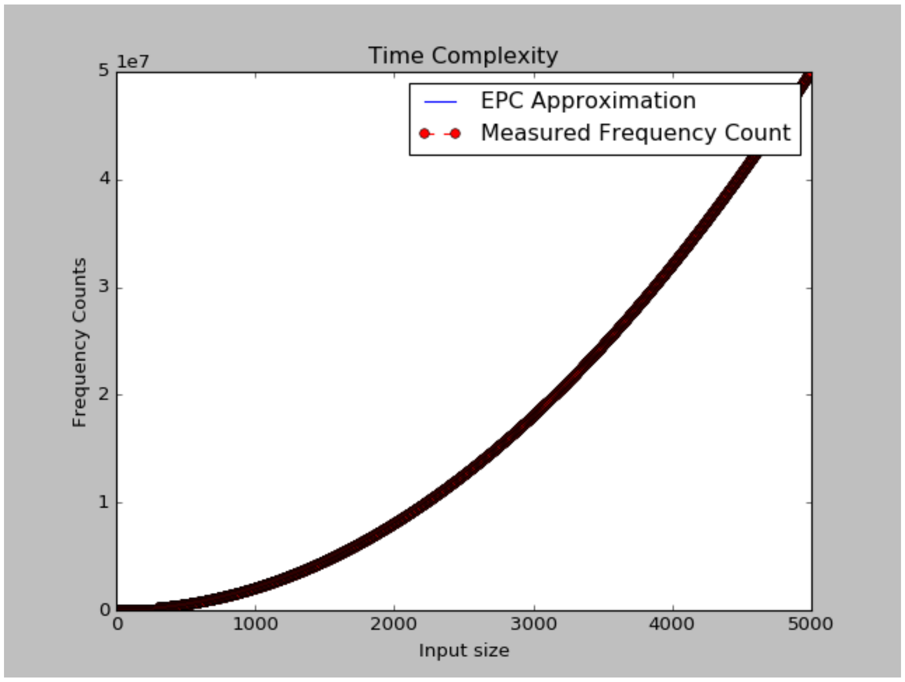


Figure F.24: Graph of Piecewise(n)

Appendix G

Prototype Class Diagram

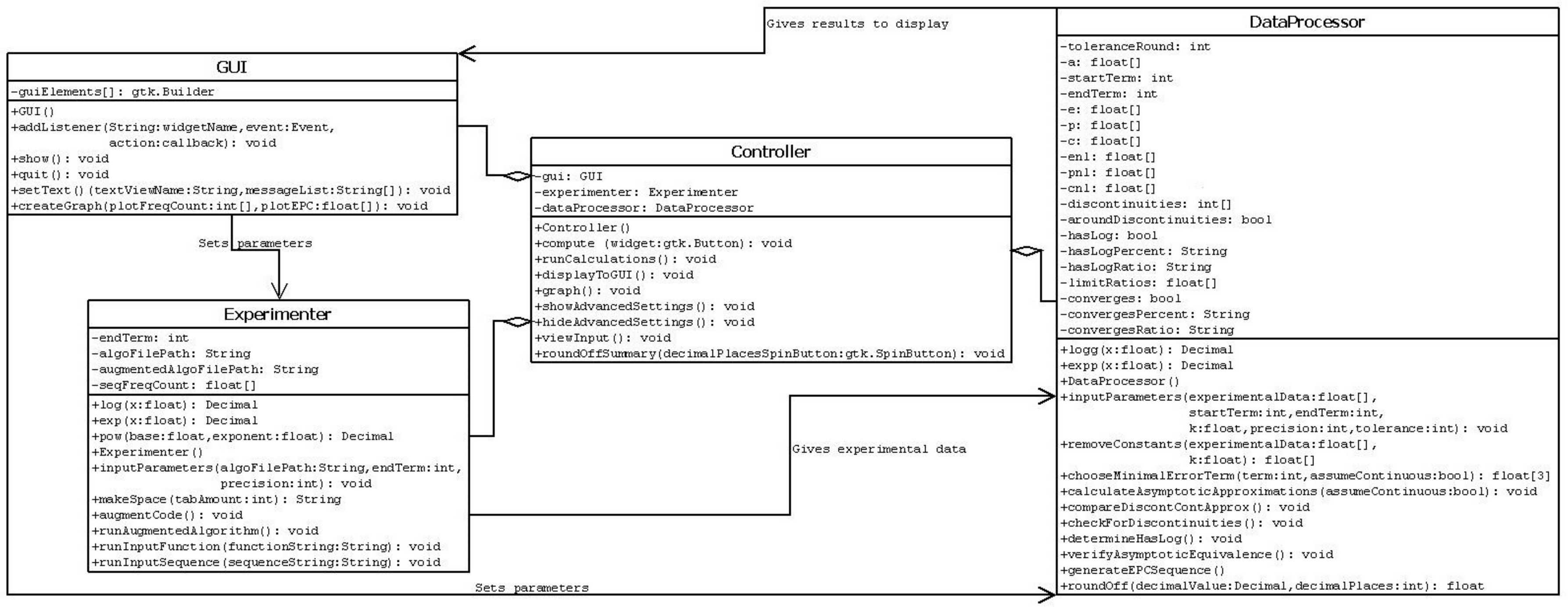

Gives results to display
GUI
-guiElements[]: gtk.Builder
+GUI()
+addListener(String:widgetName,event:Event,
action:callback): void
+show(): void
+quit(): void
+setText()(textViewName:String,messageList:String[]): void
+createGraph(plotFreqCount:int[],plotEPC:float[]): void
Sets parameters
Experimenter
-endTerm: int
-algoFilePath: String
-augmentedAlgoFilePath: String
-seqFreqCount: float[]
+log(x:float): Decimal
+exp(x:float): Decimal
+pow(base:float,exponent:float): Decimal
+Experimenter()
+inputParameters(algoFilePath:String,endTerm:int,
precision:int): void
+makeSpace(tabAmount:int): String
+augmentCode(): void
+runAugmentedAlgorithm(): void
+runInputFunction(functionString:String): void
+runInputSequence(sequenceString:String): void
Controller
-gui: GUI
-experimenter: Experimenter
-dataProcessor: DataProcessor
+Controller()
+compute (widget:gtk.Button): void
+runCalculations(): void
+displayToGUI(): void
+graph(): void
+showAdvancedSettings(): void
+hideAdvancedSettings(): void
+viewInput(): void
+roundOffSummary(decimalPlacesSpinButton:gtk.SpinButton): void
Gives experimental data
Sets parameters
DataProcessor
-toleranceRound: int
-a: float[]
-startTerm: int
-endTerm: int
-e: float[]
-p: float[]
-c: float[]
-enl: float[]
-pnl: float[]
-cnl: float[]
-discontinuities: int[]
-aroundDiscontinuities: bool
-hasLog: bool
-hasLogPercent: String
-hasLogRatio: String
-limitRatios: float[]
-converges: bool
-convergesPercent: String
-convergesRatio: String
+logg(x:float): Decimal
+expp(x:float): Decimal
+DataProcessor()
+inputParameters(experimentalData:float[],
startTerm:int,endTerm:int,
k:float,precision:int,tolerance:int): void
+removeConstants(experimentalData:float[],
k:float): float[]
+chooseMinimalErrorTerm(term:int,assumeContinuous:bool): float[3]
+calculateAsymptoticApproximations(assumeContinuous:bool): void
+compareDiscontContApprox(): void
+checkForDiscontinuities(): void
+determineHasLog(): void
+verifyAsymptoticEquivalence(): void
+generateEPCSequence()
+roundOff(decimalValue:Decimal,decimalPlaces:int): float

Figure G.1: Class Diagram of the Proposed System