Experimental Algorithmics Through Algebraic Manipulation of Asymptotic Equivalences

John Paul S. Guzman and Teresita C. Limoanco

College of Computer Studies, De La Salle University-Manila
2401 Taft Avenue, Metro Manila,
1004 Metro Manila, Philippines
john_paul_guzman@dlsu.edu.ph, tessie.limoanco@delasalle.ph

ABSTRACT

Literature has shown that apriori or posteriori estimates are used in order to determine efficiency of algorithms. Apriori analysis determines the efficiency following algorithm's logical structure while posteriori analysis accomplishes this by using data from experiment. The advantage of apriori analysis over posteriori is that it does not depend 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. This paper presents a posteriori method which analyzes the measured of 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 enables one to get an insight on the asymptotic behavior of an algorithm without the need to do any manual computations or mathematical analysis. The method was tested on 25 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 precise approximations of time complexity that are expected from manual apriori calculations. This is true even in cases where the performance of the algorithm contains discontinuities. However, the rate of which the approximation approaches the correct answer varies depending on the input algorithm.

Keywords

Algorithm Analysis, class complexity, time complexity, asymptotic notations, empirical algorithmics.

1. INTRODUCTION

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.

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 [5]. To be more specific, posteriori approach 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. Apriori approach on the other hand, 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 [11]. The advantage of this method is that it does not depend on any external factors like those in posteriori analysis. The method is much more thorough, but is limited by how powerful the current methods of mathematical analysis are. An example of this limitation can be seen when solving for closed-form solutions to recurrence relations. Linear recurrence relations could be solved through Characteristic equations which is a polynomial equation with the same degree as the recurrence relation. However, not all polynomial equations are solvable. Abel's Impossibility Theorem states that polynomial equations with degree 5 or higher do not have a general solution [1]. Thus, we usually settle on approximate solutions given by numerical methods for quintic equations. Nonlinear recurrence relations are much more complex than linear recurrence relations and they are also not generally solvable [13]. Example 1 is the general form of a linear recurrence relation, where all the c's are constant real numbers and k is a positive integer. Example 2 is a nonlinear recurrence relation known as the Ackermann function which is known as an example of a non-primitive recursive function [2].

$$F(n) = c_1F(n-1) + c_2F(n-2) + c_3F(n-3) + ... + c_kF(n-k) \quad \textbf{(Example 1)}$$

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

Iterative method is commonly used to determine algorithm's behavior following apriori technique. This 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 [3]. 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 previously stated.

Algorithm profiling is a posteriori analysis technique that it follows three distinct steps known as the augment, run/execute, analyze process. Here, a given algorithm is preprocessed by inserting special code or marks that correspond to an increase in running time. These special codes in turn will generate the necessary timing information upon the execution of the algorithm. A special program called as the analyzer gathers all the timing information recorded during the algorithm execution to generate a report regarding the time consumption of the algorithm [12]. Even though this approach is fully automated, the generated timing information may vary depending on hardware and software specifications.

Input-sensitive profiling is another posteriori analysis technique that is also a type of profiling [9]. It returns a function that estimates the growth of an input algorithm rather than simple time measurements. The method plots or maps input sizes to measurements of the performance then utilizes several rules to determine the relationship between the measurements and the input size. One example of a rule is the Guess Ratio [9] that assumes the performance follows a function of the form $P = \Sigma_i(a_i x^{b_i})$ where all a_i's and b_i's are non-negative rational numbers. It attempts to identify a guess function of the form $G = x^b$. Utilizing the knowledge that if P is not $O(G)$, then P/G must be increasing for high values of n. The method starts by setting b to 0 and increments b until P/G becomes nonincreasing which is when the rule infers that P is $O(n^b)$. The method utilizes the Oracle functions which guides the execution of the rules. An example of an Oracle function is the Trend function [9]. The Guess Ratio rule uses the Trend function to determine if P/G is decreasing, increasing, or neither. Although the method gives 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.

This paper shows the empirical algorithmic research conducted in performing a posteriori analysis through the use of frequency counts derived from a given algorithm or program to determined its asymptotic behavior. This paper is organized as follows: Section 2 discusses the development of the posteriori method in the study. Section 3 presents the different experiments conducted on the method and the analysis of the results follows. Lastly, Section 4 discusses further work that can be done in this area.

2. THE DEVELOPED METHOD

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 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.

2.1 Design of the Developed Method

A prototype has been developed written in Python. Figure 1 illustrates its logical flow. Essentially, a given Python function taking one integer argument is modified by adding instructions to increment a global counter denoting the frequency count for every line of instruction (e.g., conditional check, function call, declaration, assignment) executed. This approach is similar to the instruction counting discussed in [12]. The modifications made does not interact or affect the logical structure of the program. The resulting augmented program is then executed on various input sizes starting from 0 and increments by 1 until a user-defined upper value that specifies the end of program's execution. 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 Subsection 2.3 and derived in Appendix B to determine the asymptotic behavior of the input algorithm.

Figure 1: Architectural Design of the Developed System

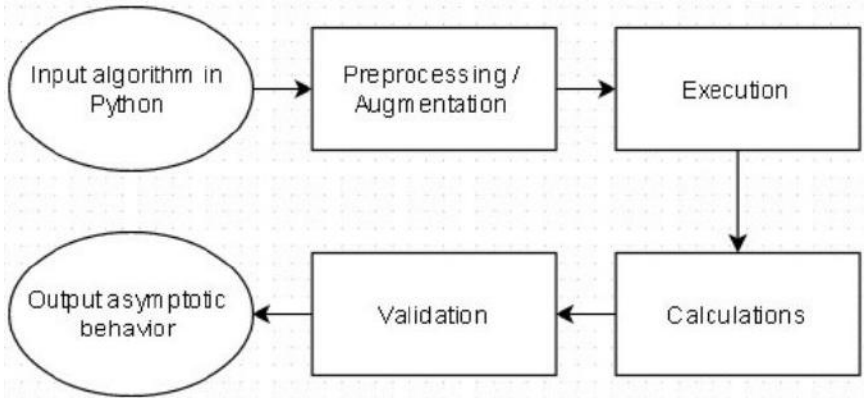

Table 1 shows an example of Python program and its corresponding augmented code. The augmented code has exactly the same logical structure as the original code with inserted frequency count increment instructions (can be seen in line numbers 3,5, 6, 8 and 9).

Table 1: Sample Python code and its corresponding augmented code

Input/Original Program	Augmented Program
1 def f(n): 2 for i in range(0, n): 3 for j in range(0, n): 4 x=i+j	1 def f(n): 2 global freqCount 3 freqCount+=1 #count for exit i loop condition 4 for i in range(0, n): 5 freqCount+=1 #count for manipulating i variable 6 freqCount+=1 #count exit j loop condition 7 for j in range(0, n): 8 freqCount+=1 #count for manipulating j variable 9 freqCount+=1 #count for the execution of line 10 10 x=i+j

Using the program in Table 1 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 taken from the execution when 0 input is 1, given 1 input is 5, given 2 is 25, etc. The array of frequency count measurements will be passed through a series of calculations that outputs the asymptotically behavior the input algorithm. The calculations will be discussed in the following subsections.

2.2 The Use of Asymptotic Notations

Asymptotic notations are generalizations of the behavior of a function given an arbitrarily large input size. The three most commonly used notations are Big O (O), Big Omega (Ω), and Big Theta(Θ). Big O acts as an asymptotic upper bound, while Big Omega acts as an asymptotic lower bound, and Big Theta acts as an asymptotic tight bound. These three are mathematically defined using limits [14] as shown below. Equivalence of these definitions with definitions presented in [3, 8] can be found in Appendix D.

$$(\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,\ c < \infty) \leftrightarrow (f(n) \in O(g(n))) \quad \textbf{(Definition 1)}$$

$$(\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,\ 0 < c < \infty) \leftrightarrow (f(n) \in \Theta(g(n))) \quad \textbf{(Definition 2)}$$

$$(\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,\ c > 0) \leftrightarrow (f(n) \in \Omega(g(n))) \quad \textbf{(Definition 3)}$$

Another asymptotic notation that is worth considering is the asymptotic equivalence. It can be defined using limits as seen in Definition 4. 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 Definition 2 and 4. Although asymptotic equivalence is a better approximation and it holds more information than Big O, Big Omega, and Big Theta, 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.

$$(\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1) \leftrightarrow (f(n) \sim g(n)) \quad \textbf{(Definition 4)}$$

2.3 Extracting the Asymptotic Equivalence from the Measurements

The formulas will output a function that is asymptotically equivalent to the performance (measured in Frequency count) 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 and even tighter than Big Theta.

The accuracy up to a constant factor, as implied by using asymptotic equivalence instead of Big Theta, is considered as constant factors will never become negligible no matter how much the input size grows. Consider two functions, *A* and *B*, both working on an input n: *A* runs for a time *100n* while *B* runs for a time *n*. No matter how large *n* grows, it is still a known fact that *A* will run 100 times slower than *B*. Thus, it could never truly be negligible. This can be shown mathematically as any factor *k* within a limit could be factored outside the limit (Equivalence 1) [4]. The asymptotically equivalent function will be extracted from the array of frequency count measurements. The process of extracting the asymptotic equivalent function and the reasoning behind it are discussed in the following paragraphs.

$$\lim_{n\to\infty} k(\frac{f(n)}{g(n)}) = k(\lim_{n\to\infty} \frac{f(n)}{g(n)}) \quad \textbf{(Equivalence 1)}$$

The study on the developed method covers rates of growth from logarithmic to exponential including 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. 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 general form of the problem can be defined as:

$$A^n \cdot n^B \cdot Cn \cdot \sqrt[D]{n} \cdot log_E(n) \quad \textbf{(Expression 1)}$$

The number of variables in Expression 1 could be reduced if it is written down as:

$$e^n \cdot n^p \cdot c\ ln(n) \quad \textbf{(Expression 2)}$$

where $e = A$, $p = B + 1 + \frac{1}{D}$, and $c = C\ log_E(e_{Euler})$. e_{Euler} is the Euler's 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 [4] which in turn can be expressed as a constant factor. Expressions 1 and 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 4. 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 $1^n n^1 3\ ln(n)=(3n)ln(n)$. Another example is $(2^{n+3})ln(n)$ where variables can be set to *e=2*, *p=0*, *c=8* since $2^n n^0 8\ ln(n)=(2^{n+3})ln(n)$. 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 $e^n \cdot n^p \cdot c$ instead of $e^n \cdot n^p \cdot c\ ln(n)$. A boolean value *hasLog* is introduced that can generalize the two cases: (a) *hasLog=False* represents $e^n \cdot n^p \cdot c$, while (b) *hasLog=True* represents $e^n \cdot n^p \cdot c\ ln(n)$. An example for *hasLog=False* is given in $8n\sqrt{n}$, variables can be set to *e=1*, *p=1.5*, *c=8* since $1^n n^{1.5} 8 = n \cdot n^{1/2} \cdot 8 = 8n\sqrt{n}$.

The structure of each term allows one to generalize all possible growth rates within the scope. But, this could further be generalized because 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. In this case, *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 yields to 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 1 where a_n is the measurement given an input size of *n*; *m* is an arbitrary positive integer which represents the number of terms; $ln(n, hasLog_i)$ function returns *ln(n)* if $hasLog_i=True$, else it returns *1*.

$$a_n = \sum_{i=1}^{m}(e_i^n \cdot n^{p_i} \cdot c_i\ ln(n,\ hasLog_i)) \quad \textbf{(Equation 1)}$$

It is safe to assume that this series in Equation 1 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 equal growth rate which means 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 2. The motivation behind the rearrangement is only to simplify the remaining manipulations and it is possible because addition is commutative.

$$\lim_{n\to\infty} \frac{\sum_{i=2}^{m}(e_i^n \cdot n^{p_i} \cdot c_i\ ln(n, hasLog_i))}{e_1^n \cdot n^{p_1} \cdot c_1\ ln(n, hasLog_1)} = 0 \quad \textbf{(Equation 2)}$$

By the definition in Equation 1 and the property in Equation 2, it can be shown that $e_1^n * n^{p_1} * c_1\ ln(n,\ hasLog_1)$ is asymptotically equivalent to a_n:

$$\lim_{n\to\infty} \frac{an}{e_1^n n^{p_1} c_1\ ln(n, hasLog_1)} = \lim_{n\to\infty} \frac{e_1^n n^1 c_1\ ln(n, hasLog_1) + \sum_{i=2}^{m}(e_i^n n^{p_i} c_i\ ln(n, hasLog_i))}{e_1^n n^{p_1} c_1\ ln(n, hasLog_1)} = \frac{1+0}{1} = 1 \quad \textbf{(Lemma 1)}$$

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*. 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 $e^n * n^p * c$ 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 can deduce that $a_n \sim n^p$ as $n \to \infty$ and then $log_n(a_n) \sim p$ where p is completely isolated.

The algebraic derivation for the formulas for e, p, and c could be seen along with a method for determining *hasLog* and detecting discontinuities in Appendix B. The detection of discontinuities allow the program to adjust the calculations accordingly. A further in-depth explanation is also included in Appendix B.

2.3 Validating Asymptotic Equivalence

The calculated e, p, c, and *hasLog* values must satisfy Definition 4 which states that the ratios as n grows 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. A concept of convergence and divergence must be associated with something that can be done by using only a finite number of discrete function values. An intuitive way to test for convergence for a sequence F is as follows: if the distance between consecutive terms are not changing or decreasing ($\frac{d}{dn}|\Delta F| \leq 0$), $\Delta F = F_{n+1} - F_n$ for majority of the terms, then it converges else it diverges. Note that $log(-r) = log(r) + i\pi$ which does not affect convergence and a derivative with a positive sign is associated with divergence, while a derivative with a negative sign or 0 is associated with convergence. Factors that do not affect the sign are isolated to the right side of the explicit multiplication symbol (*).

Table 2.1: Distances Between Consecutive Terms of Functions

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

As seen in Table 2.1, the intuitive way gives correct results in most cases, but fails when considering radical and logarithmic growth. Thus the conditions must be modified to give correct results for all growths within the scope. By observing the $F_n = n^r$ row in Table 2.1, the reason why it failed for ($0 < r \leq 1$) is due to the (r - 1) factor after taking the derivative. 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 be $\Delta' F$ which is defined as $\Delta' F = n\, \Delta F$. Table 2.2 is constructed by substituting the ΔF in Table 2.1 by $\Delta' F$.

Table 2.2: Modified Distances Between Consecutive Terms of Functions

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

As shown in the Table 2.2, the initial guess is correct and all the $\Delta'F$ matches the convergence for all the considered *Fn*. The same intuition about why ΔF should work as a convergence test is applicable to $\Delta'F$, 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 fails to detect divergence while still hold true for all cases that were already correct in the initial ΔF tests.

It is worth noting that this is not an actual convergence test. It detects when the ratio of performance measurement growth (a_n) and approximated growth ($e^n\ n^p\ c\ log(n, hasLog)$) move too fast to approach 1. If it does move too fast, then it is flagged as not converging. But, this results in a false positive convergence in cases that considers rates of growth that diverges very slowly like *log(log(n))*.

3. TESTING, RESULTS AND ANALYSIS

A prototype was created where the method discussed in the Section 2 is implemented and tested on both iterative and recursive algorithms written in Python 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*. 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 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 default context precision (e.g. the number of significant digits used in computations) was set to 132. Lastly, the generated array of Frequency counts were normalized before the computations were done. Normalization was done by subtracting each element by the original value of the *0th* element in the array to improve accuracy by removing unnecessary constants.

Table 3 summarizes the results from the testing and it has been rounded off to 6 decimal places. The actual output refers to the output of the implementation of the algorithm analysis method being developed and each algorithm is run for 5000 input sizes except for *Fibo(n)*, *Ack(3, n)*, *FactLike(n)*, and *QuinticFibo(n)* due to their astronomical growth rates. The expected output refers to the output from apriori analysis methods (see manual calculations in 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 2.

Table 3: Sample test data and the corresponding results

Algorithm	**Expected Algorithm Output**	**Actual Algorithm Output**	**δError (Actual-Expected)**
Iter1(n)	e= 1 p= 0 c= 41	e= 1 p= 0 c= 41	δe= 0 δp= 0 δc= 0

	hasLog= False	hasLog= False (Using 20000 terms)	δhasLog= 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)	δe= 0 δp= 0 δc= 0 δhasLog= 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)	δe= 0 δp= 0 δc= 0 δhasLog= 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)	δe= 0 δp= 0 δc= 0 δhasLog= 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)	δe= 0 δp= -0.0004 δc= 0.006826 δhasLog= 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)	δe= 0 δp= -0.0008 δc= 0.006836 δhasLog= 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)	δe= 0 δp= -0.0004 δc= 0.00683 δhasLog= None
Iter8(n)	e= 1 p= 3 c= 0.333333 hasLog= False	e= 1.000000 p= 2.999999 c= 0.33333 hasLog= False (Using 5000 terms)	δe= 0 δp= -0.000001 δc= 0.000004 δhasLog= 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)	δe= 0 δp= 0.0001 δc= -0.00396 δhasLog= None
FactLike(n)	e= divergent p= divergent	Converges= False (Using 30 terms)	Correct divergence

	c= divergent hasLog= n/a		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)	δe= 0 δp= -0.000035 δc= -0.000903 δhasLog= 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)	δe= 0 δp= 0 δc= 0 δhasLog= 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)	δe= 0 δp= 0 δc= 0 δhasLog= 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)	δe= 0 δp= 0 δc= 0 δhasLog= 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)	δe= 0 δp= -0.001439 δc= 0.06166 δhasLog= 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)	δe= 0 δp= 0 δc= -0.000001 δhasLog= None
QuinticFibo(n)	e= 1.927562 p= 0 c= unknown hasLog= False	e= 1.927562 p= 0.000000 c= 0.183563 hasLog= False (Using 40 terms)	δe= 0 δp= 0 δc= unknown δhasLog= 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)	δe= 0 δp= 0 δc= 0 δhasLog= None
BinarySearch(n)	e= 1 p= 0	e= 1.000000 p= -0.006035	δe= 0 δp= -0.006035

	c= 7.213475 hasLog= True	c= 7.959977 hasLog= True (Using 20000 terms)	δc= 0.746502 δhasLog= 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)	δe= 0 δp= -0.000133 δc= 0.001705 δhasLog= 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)	δe= 0 δp= -0.0004 δc= 0.005114 δhasLog= 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)	δe= 0 δp= -0.011932 δc= 2.517749 δhasLog= None
Quick sort(n)	e= 1 p= 2 c= 2 hasLog= False	e= 1.000000 p= 1.998203 c= 2.030848 hasLog= False (Using 5000 terms)	δe= 0 δp= -0.001797 δc= 0.030848 δhasLog= 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)	δe= 0 δp= -0.000666 δc= 0.008533 δhasLog= 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)	δe= 0 δp= 0.000004 δc= -0.000058 δhasLog= None

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 25 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 parameters. 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 the Actual Output column of Table 3 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). 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 [2], 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 closed form). It was solved through numerical approximations. The expected *c* value, on the other hand, was not computed for due to the difficulty in working with complex numbers and the errors that 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.

4. CONCLUSION AND RECOMMENDATION

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 by the Merge_sort(n) algorithm. Its slow pace in converging to the correct answer is due to the fact that its largest term ($11.5n\log(n)$) 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 numerical means. This consequently leads to a sound basis of determining algorithms' asymptotic behavior. Moreover, 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.

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.

5. REFERENCES

[1] 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.

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

[3] Cormen, T., et al. (2009). Introduction to Algorithms. 3rd edition. The MIT Press.

[4] Fradkin, L. (2013). Elementary Algebra and Calculus: The Whys and Hows. Bookboon.

[5] Gossett, E. (2009). Discrete mathematics with Proof. 2nd edition. A John Wiley and Sons, Inc. Publication

[6] Greene, D., et al. (1982). Mathematics for the analysis of algorithms. 2nd edition. Birkhuser.

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

[8] McConnell, J. (2008). Analysis of algorithms. Jones and Bartlett Publishers Inc.

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

[10] Montesinos, V., et al. (2015). An introduction to modern analysis. Springer.

[11] Pai, G. (2008). Data structures and algorithms:concepts, techniques and applications. Tata McGraw-Hill Education Pvt. Ltd.

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

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

[14] Sedgewick, R., etal (2013). Introduction to the Analysis of Algorithms, 2nd edition. Addison-Wesley Professional.

[15] 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

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

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

Listing A.2: Iter2

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

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

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

Listing A.3: Iter3

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

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

Listing A.4: Iter4

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

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

Listing A.5: Iter5

```
1 def f(n): #2n^2+2n+1
2       for i in range(0  ,n):
3           for j in range(0, n):
4               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

```
1 def f(n):#n^2+2n+1
2       for i in range(0  ,n):
3           sum_to_i=0
4           for j in range(0, i):
5               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:

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

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.

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

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

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

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

By inspecting the largest terms from all the $F_{Ack}(m,n)$'s the expected e, p, c, and hasLog values are:

$e_{Ack0} = 1$, $p_{Ack0} = 0$, $c_{Ack0} = 2$, $hasLog_{Ack0} = False$

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

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

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

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.

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

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:

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

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

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

```
5           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:

$$\begin{aligned} &[F_{QF}(n+1) - F_{QF}(n) - F_{QF}(n-1) - F_{QF}(n-2) - F_{QF}(n-3) = 2] \\ -&[F_{QF}(n) - F_{QF}(n-1) - F_{QF}(n-2) - F_{QF}(n-3) - F_{QF}(n-4) = 2] \\ \hline \\ &F_{QF}(n+1) - 2F_{QF}(n-1) - 2F_{QF}(n-2) - 2F_{QF}(n-3) - 2F_{QF}(n-4) = 0 \end{aligned} \tag{A.4.3}$$

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.

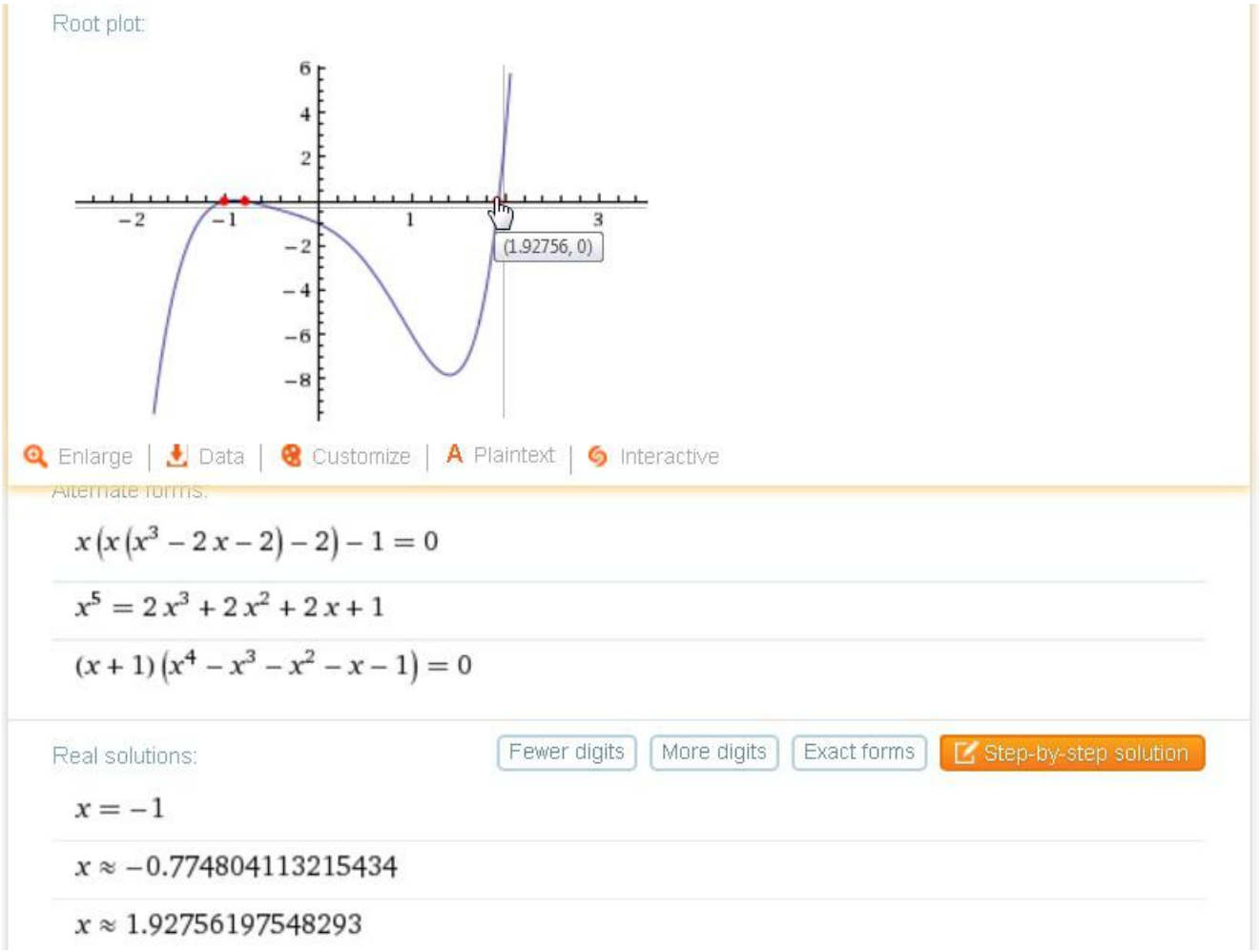

Figure A.1: QuinticFibo Characteristic Equation Roots (Wolfram Alpha, n.d.)

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

```
1 def linearSearch(alist, item):
2     for i in range(0, len(alist)):
3         if alist[i] == item:
4             return i
5     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

```
1 def f(n):
2     x = range(0,n,1)
3     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

```
1  def binarySearch(alist, item):
2      first = 0
3      last = len(alist)-1
4      found = False
5      while first<=last and not found:
6          midpoint = (first + last)//2
7          if alist[midpoint] == item:
8              return midpoint
9          else:
10             if item < alist[midpoint]:
11                 last = midpoint-1
12             else:
13                 first = midpoint+1
14     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

```
1  def f(n):
2      x = range(0,n,1)
3      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

```
1 def f(n):
2     x = range(n,0,-1)
3     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

```
1  def merge_sort(items):
2      if len(items) > 1:
3          mid = len(items) // 2          # Determine the
               midpoint and split
4          left = items[0:mid]
5          right = items[mid:]
6          merge_sort(left)               # Sort left list in-
               place
7          merge_sort(right)              # Sort right list in-
               place
8          l = 0
9          r = 0
10         for i in range(len(items)):      # Merging the left
               and right list
11             if l < len(left):
12                 lval = left[l]
13             else:
14                 lval = None
15             if r < len(right):
16                 rval = right[r]
17             else:
18                 rval = None
```

```
19                if (lval is not None and rval is not None and
                      lval < rval) or rval is None:
20                    items[i] = lval
21                    l += 1
22                else:
23                    items[i] = rval
24                    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

```
1  def f(n):
2      x = range(0,n,1)
3      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)

```
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
```

$F_{Quick} = R(n); R(0) = R(1) = 1$
$R(n) = 4 + EX_i + n(IT_i + 3) + 3 + R(n-1) + 1 = R(n-1) + 4n + 9$
$= R(n-2) + 4(n-1) + 9 + 4n + 9 = R(n-2) + 2(4n+9) - 4$
$= R(n-3) + 4(n-2) + 9 + 2(4n+9) - 4 = R(n-2) + 2(4n+9) - 4(1+2)$
$= ... = R(n-i) + i(4n+9) - 4\sum_{j=0}^{i-1} j = R(n-i) + i(4n+9) - 4\frac{i(i-1)}{2}$
$= R(n-i) + i(4n+9) - 2i(i-1)$
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

```
1  def f(x):
2      loops = 0
3      if x<100:
4          loops = 16
5      else:
6          if x<200:
7              loops = x
8          else:
9              if x<300:
10                 loops = 4*x
11             else:
12                 loops = x**2
13     for i in xrange(loops):
14         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:

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

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).

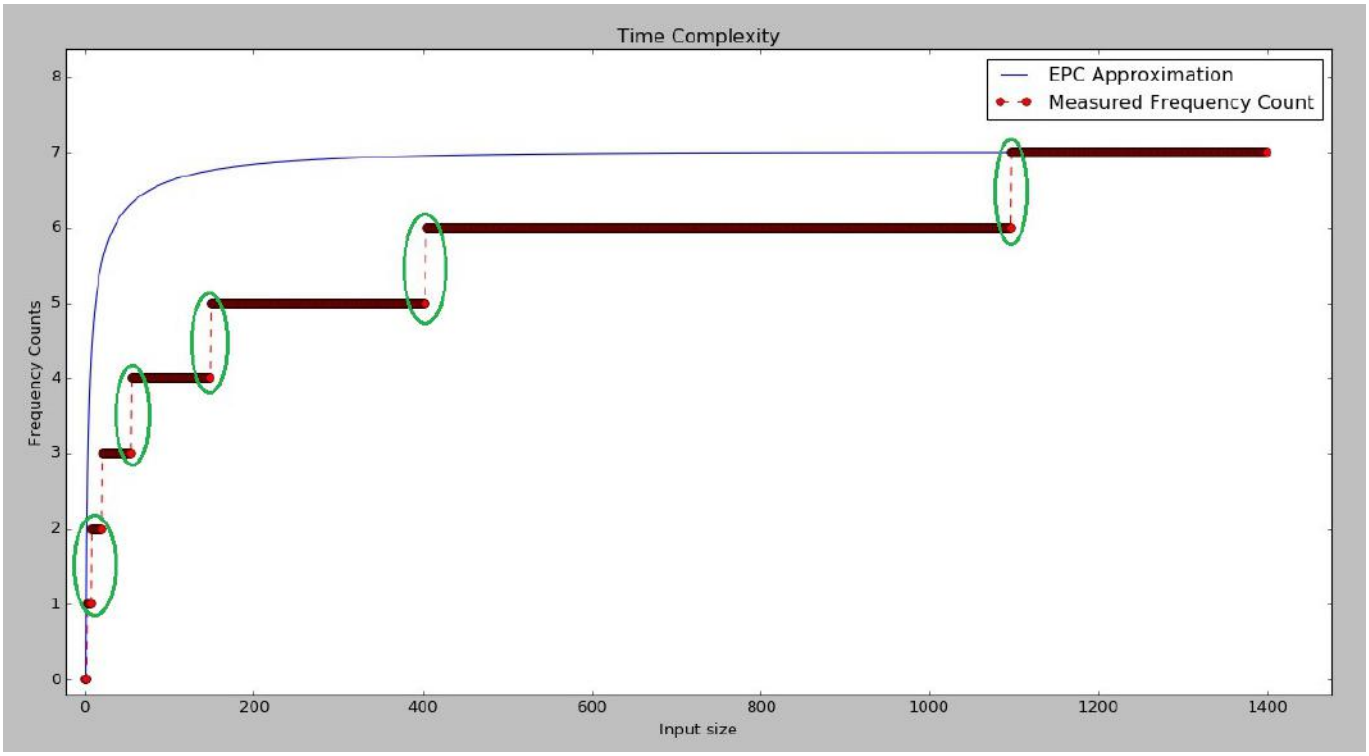

Figure B.1: Sample of Discontinuous Function

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

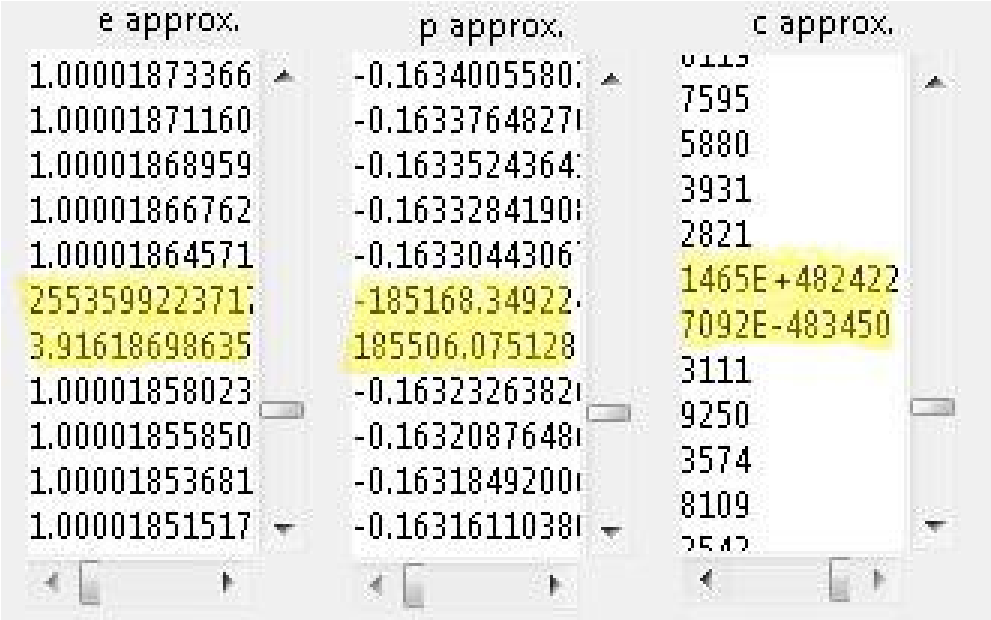

Figure B.2: Sample of Fluctuation due to a Discontinuity

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 2.3. Note that n is a positive number approaching infinity which is useful fact in the simplification of expressions.

C.1 Calculations in Table 2.1

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 2.2

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

The calculations for the $\Delta' F_n$ Column in Table 2.2 is equal to the calculations for the ΔF_n Column in Table 2.1 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:

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

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:

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

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

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 2.2. The proofs operate under the assumption that limit of $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ 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:

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

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 $\rightarrow$ 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 $\frac{f(n)}{g(n)} > \lim_{n\to\infty} \frac{f(n)}{g(n)}$, the absolute value function can be removed:

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

1. By adding $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ 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 $c = d + \lim_{n\to\infty} \frac{f(n)}{g(n)}$ such that:

$$\exists(c \in \mathbb{R}^+)\exists(n_0 \in \mathbb{R}^+)\forall(n \geq n_0)(\frac{f(n)}{g(n)} < c) \tag{D.1.13}$$

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

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

1. By adding $\frac{2f(n)}{g(n)} - \lim_{n\to\infty} \frac{f(n)}{g(n)}$ 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 $c = \frac{f(n)}{g(n)} + d - (\lim_{n\to\infty} \frac{f(n)}{g(n)} - \frac{f(n)}{g(n)})$ such that:

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

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 $\frac{f(n)}{g(n)} > \lim_{n\to\infty} \frac{f(n)}{g(n)}$ and the case when $\frac{f(n)}{g(n)} < \lim_{n\to\infty} \frac{f(n)}{g(n)}$.

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

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

1. By adding $\frac{f(n)}{g(n)} - d$ 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 $\frac{f(n)}{g(n)} - \lim_{n\to\infty} \frac{f(n)}{g(n)} < c = d < \frac{2f(n)}{g(n)} - \lim_{n\to\infty} \frac{f(n)}{g(n)}$ such that:

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

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 $\leftrightarrow$ 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:

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