An Empirical Approach to Algorithm Analysis Resulting in Approximations to Big Theta Time Complexity

John Paul Guzman[1*], Teresita Limoanco[2]
College of Computer Studies, De La Salle University-Manila, Manila, Philippines.

* Corresponding author. Email: guzmanjps@gmail.com
Manuscript submitted July 9, 2017; accepted August 29, 2017.
doi: 10.17706/jsw.12.12.946-976

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 experiments. Apriori analysis has two main advantages over posteriori analysis: a.) it does not depend on other factors aside from the algorithm being analyzed; b.) it naturally produces measurements in terms of asymptotic notations. These advantages result in thorough and more generalizable analysis. However, apriori techniques are limited by how powerful the current methods of mathematical analysis are. This paper presents a posteriori method that outputs time complexity measured in Big Theta notation. The developed method uses a series of formulas and heuristics to extract an algorithm's asymptotic behavior solely from its frequency count measurements. The method was tested on 30 Python programs involving arithmetic operations, iterative statements, and recursive functions to establish its accuracy and correctness in determining time complexity behavior. Results have shown that the developed method outputs precise approximations of time complexity that are expected from manual apriori calculations.

Key words: Algorithm analysis, time complexity, asymptotic notations, empirical algorithmics.

1. Introduction

Algorithm analysis is used to measure the efficiency of algorithms. It can be used to compare different algorithms, balance trade-offs between memory to time consumption, and identify an optimal choice. 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 [1]. To be more specific, the 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. However, different hardware and software used in the experiments may yield varying results. The apriori approach on the other hand, analyzes the logical flow of the algorithm. It quantifies the amount of consumed resources while keeping track of the logical progression of the algorithm. Its goal is to classify an algorithm based on its asymptotic behavior which serves as the metric for its efficiency [2]. 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 in complex recurrence relation problems. Linear recurrence relations could generally be solved through characteristic equations which are polynomial equations with the same degree as the

recurrence relation [3]. However, not all polynomial equations are easily solvable. Abel's Impossibility Theorem states that polynomial equations with degree 5 or higher do not have a general solution [4]. This implies that in some cases, one must resort to using approximations from numerical root-finding algorithms [5]. Using root-finding algorithms may give rise to other problems such as inaccurate answers when the solutions converge slowly or it may not find all the roots [5]. Other difficulties arise from nonlinear recurrence relations. These are more complex than linear recurrence relations and they are not generally solvable [6]. There are also non-primitive recursive functions which are much more complex than nonlinear recurrences. One example of this is the Ackermann function [7].

Iterative method is commonly used to determine algorithm's behavior following apriori technique. This is done by expanding any iterations or recursions present in an algorithm and determining the frequency count for each expansion. And then, identifying a closed-form expression that matches the behavior of frequency counts. The generalization of the pattern together with the initial condition or base case are used to evaluate the behavior of an algorithm [8]. But, there are some cases where the mathematics necessary for identifying the closed-form expression does not exist.

Algorithm profiling is a posteriori analysis technique that follows three distinct steps known as the augment, execute, analyze. Here, a given algorithm is preprocessed by inserting special codes that correspond to an increase in running time. These special codes will generate the necessary timing information upon the execution of the algorithm. The generated timing information may vary depending on hardware and software specifications. A special program called the analyzer gathers all the timing information recorded during the algorithm execution to generate a report regarding the time consumption of the algorithm [9].

Input-sensitive profiling is a special type of profiling that tries to find a curve that is an upper bound or a lower bound to some input data [10]. This method maps input sizes to performance measurements, then utilizes several rules to generate guess curve and Oracle functions to check for useful features within the data such as Trend and Concavity. One example of a rule is the Guess Ratio that assumes the performance follows a function of the form $P=\Sigma_i(a_i\ x\text{^}b_i)$ where all a's and b's are non-negative rational numbers [10]. It attempts to identify a guess curve of the form $G=x^b$. Utilizing the fact that if P is not $O(G)$, then P/G must be increasing for large values of n. The rule starts by setting b to 0 and increments b until P/G becomes nonincreasing which is when the rule infers that P is $O(n^b)$. This method utilizes the Oracle functions which guides the execution of the rules. An example of an Oracle function is the Trend function [10]. The Guess Ratio rule uses the Trend function to determine if P/G is decreasing, increasing, or neither. Although the method gives promising estimates to asymptotic behavior, methods of this sort are limited to evaluating behavior with Big Oh and Big Omega polynomial estimates. This can be seen in the Guess Ratio rule. Since the method does guesses and checks only after each increment of b, 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 the Trend oracle signals the stop of the Guess Ratio rule. This introduces a significant discrepancy and loses information contained in the performance function.

This paper shows a method of empirical algorithm analysis that yields approximations to the asymptotic equivalent time complexity of an input algorithm. The asymptotic equivalence notation is a stricter special case of the Big Theta notation. This notation provides additional information about constant multipliers to the rate of growth. The justification for choosing this notation over Big Theta is discussed in Section 3.1.

This paper is organized as follows: Section 2 discusses the development of the method in the study. Section 3 discusses the arguments and theoretical basis that supports the method. Section 4 presents the different experiments conducted on the method and the analysis of the results follows. Lastly, Section 5 discusses further work that can be done in this area.

2. The Developed Method

This study explores 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 relative to the input size. This means the method bypasses all of the complexities from the logical structure of an algorithm and reduces them 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. This ensures that the method will bypass the mathematical limitations of apriori methods assuming that the closed-form solution is a rate of growth that exists within the scope defined in Section 3.2.1. Furthermore, the method removes the disadvantages seen in posteriori analysis since measuring frequency counts can be automated and does not rely on hardware specifications.

A prototype has been developed in Python and Fig. 1 illustrates its logical flow. Essentially, an input Python function taking an integer argument *n* is modified, where *n* denotes the input size of the function. The modifications are done by inserting instructions to increment the global variable denoting the frequency count. The insertions are done for every line of instruction (e.g., conditional check, function call, declaration, assignment) to be executed. This approach is similar to the instruction counting discussed in [9]. These modifications are made such that they do not interact nor 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 bound. 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 formulas and heuristics discussed in Section 3.2 that will determine the asymptotic behavior of the input algorithm.

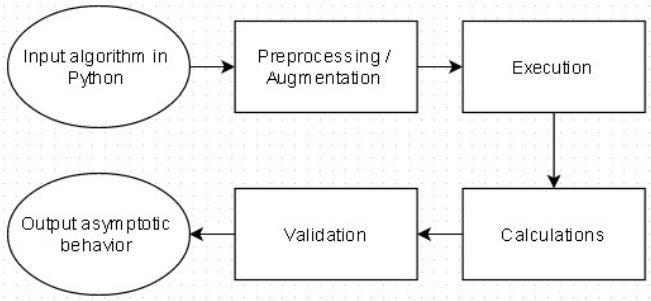

Fig. 1. Architectural design of the developed prototype.

An example of an input algorithm and its corresponding augmented code can be seen in Table 1. The augmented code has exactly the same logical structure as the original code with inserted global variable declaration and frequency count increment instructions (can be seen in line numbers 2, 3, 5, 6, 8 and 9).

Table 1. Sample Python Code and Augmentation

Input Algorithm	Augmented Program
1 def f(n): 2 for i in range(n): 3 for j in range(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(n): 5 freqCount+=1 #for manipulating var i 6 freqCount+=1 #for exiting loop j 7 for j in range(n): 8 freqCount+=1 #for manipulating var j 9 freqCount+=1 #for executing line 10

3. Theoretical Framework

3.1. Utilizing 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 Oh (O), Big Omega(Ω), and Big Theta(Θ). Big Oh acts as an asymptotic upper bound, while Big Omega acts as an asymptotic lower bound, and Big Theta acts as an asymptotic tight bound [11]. These are mathematically defined as shown in (1).

$$\begin{gathered} f(n) \in O\big(g(n)\big) \Leftrightarrow \exists_c \exists_{n_0} \forall_n \big(f(n) \leq cg(n)\big), \\ f(n) \in \Omega\big(g(n)\big) \Leftrightarrow \exists_c \exists_{n_0} \forall_n \big(f(n) \geq cg(n)\big), \\ f(n) \in \Theta\big(g(n)\big) \Leftrightarrow f(n) \in O\big(g(n)\big) \cap \Omega\big(g(n)\big), \\ \text{where } c \in \mathbb{R}^+, n_0 \in \mathbb{R}^+, n \geq n_0. \end{gathered} \tag{1}$$

The definitions in (1) are equivalent to the definitions of Big Oh, Big Theta, and Big Omega in (2) [12]. The proof for the equivalence between these two definitions can be derived by assuming the limits exists, and then introducing the convergence criterion [13].

$$\begin{aligned} f(n) \in O\big(g(n)\big) &\Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty, \\ f(n) \in \Omega\big(g(n)\big) &\Leftrightarrow 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)}, \\ f(n) \in \Theta\big(g(n)\big) &\Leftrightarrow 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty. \end{aligned} \tag{2}$$

Another asymptotic notation that is worth considering is the asymptotic equivalence as seen in (3).

$$f(n) \sim g(n) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1. \tag{3}$$

Intuitively, this means that both the functions in the numerator and denominator grow exactly at the same rate thus having a ratio of 1. Asymptotic equivalence is special case of Big Theta that is accurate up to a constant factor. It is chosen over the other notations for several reasons. First, it is a strictly stronger notion because one asymptotic equivalence can induce every possible Big Oh/Theta/Omega, but a Big Oh/Theta/Omega cannot induce an asymptotic equivalence. Second, in the limit definitions of the asymptotic notations, a constant factor *(k)* can never be negligible since it can be factored outside the limit: $\lim_{n\to\infty} \frac{kf(n)+c}{g(n)} = k \lim_{n\to\infty} \frac{f(n)+0}{g(n)}$. Third, to make growth comparisons well-defined. Comparisons are ill-defined when considering quantities that are characterized by inequalities: $\big(O(n^2) < O(2^n)\big) \not\Rightarrow (n^2 < 10\,n)$, where $n^2 \in O(n^2)$ and $10\,n \in O(2^n)$. Therefore, there is no sensible way to talk about "faster" or "slower" when only considering Big Ohs/Thetas/Omegas. Lastly, to be able to inherit algebraic manipulations from the = relation which is crucial in Section 3.2.3.

3.2. Extracting the Asymptotic Equivalence from the Measurements

A series of formulas and heuristics are used in order to generate a rate of growth that is asymptotically equivalent to the performance of an input algorithm. The heuristics guides the calculations and verifies the output, thereby enabling the method to be fully automated. The process of extracting the asymptotic equivalent function from frequency count measurements are discussed in the following subsubsections.

3.2.1. Construction of the General Problem

The method encompasses products of growth rates from logarithmic to exponential including no growth or constants. This is because there is an infinite number of growth rates and most of the common algorithms fall under this range. Furthermore, growth rates beyond logarithmic and exponential will grow too slow to detect or 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. A model is used to

characterize the general problem. Equation (4) attempts to model one term of the overall growth rate by allowing each growth within the scope to have its own independent variable from V_1 to V_5 which are then combined to form a product. In this construction, each growth is ensured to be independent since the division from (3) acts commutatively on products independent on which factor it acts on. This property enables our model capture general growths (e.g., $f(n)=(2)\ (x^3)\ (4^n)\ log_5(n)$). The number of unknowns in (4) can be reduced and the simplified expression is shown in (5).

$$t(n) = {V_1}^n \cdot n^{V_2} \cdot (V_3)n \cdot \sqrt[V_4]{n} \cdot log_{V_5}(n). \tag{4}$$

$$t(n) = E^n \cdot n^P \cdot (C)\, ln(n)\,, \text{where } E = V_1, P = V_2 + 1 + \frac{1}{V_4}, C = (V_3)log_{V_5}(e). \tag{5}$$

However, the logarithmic growth $ln(n)$ cannot vanish by any choice of C without going through some undefined or indeterminate expression. Thus, we introduce the boolean variable *hasLog* in (6). This allows the $ln(n)$ factor to exist or to vanish by some boolean choice.

$$t(n) = E^n \cdot n^P \cdot (C)doLog(n, hasLog), \text{ where } doLog(n, hasLog) = \begin{cases} \ln(n) & hasLog = True \\ 1 & hasLog = False \end{cases}. \tag{6}$$

3.2.2. The Generalization to an Arbitrary Number of Terms

The model in (6) is further generalized by considering growth rates that are defined with multiple terms. This implies that an arbitrary growth rate within the scope must satisfy (7) where a_n is the performance measurement of an algorithm where m is the number of terms within the growth rate.

$$a_n = \sum_{i=1}^{m} t_i(n) = \sum_{i=1}^{m} {E_i}^n \cdot n^{P_i} \cdot (C_i)doLog(n, hasLog_i). \tag{7}$$

It suffices to say that there exists a term $t_1(n)$ in (7) that grows faster than the rest of the terms. These are enough to show that the one largest growing term is asymptotically equivalent to the entire model as seen in (8). Thus, our model can capture growths with any number of terms, and asymptotics allows us to safely discard the negligible terms.

$$\left(\lim_{n\to\infty} \frac{\sum_{i=2}^{m} t_i(n)}{t_1(n)}\right) = 0 \Longrightarrow \left(\lim_{n\to\infty} \frac{a_n}{t_1(n)}\right) = \left(\lim_{n\to\infty} \frac{t_1(n) + \sum_{i=2}^{m} t_i(n)}{t_1(n)}\right) = \frac{1+0}{1} \Longrightarrow a_n \sim t_1(n). \tag{8}$$

Note that in subsequent discussions, the variable E_1, P_1, C_1, and $hasLog_1$ from t_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, has an asymptotically equivalent function that can be expressed in terms of 4 variables E, P, C, and *hasLog*. Asymptotics is a crucial part of the developed method as it enables one to trade-off a complicated equation like (7) with $4m$ unknowns for a simpler expression with only 4 unknowns. The simplicity of the latter expression greatly simplifies the remaining manipulations.

3.2.3. Determining *E, P , C, hasLog*

The analytic solutions for E, P, C is calculated using 3 equations generated by substituting 3 unique indices x, y, z in place of n in the asymptotic equivalence in (8). Note that a's in the left-hand side are known frequency count measurements. This leaves us with 3 equations and 3 unknowns and each unknown is solved through algebraic manipulations inherited from the = relation. Equations (9), (10), (11) show the derived formulas, and they become more accurate as the values of n, x, y, z grow larger. These are the solutions for E, P, C assuming *hasLog=True*. The solutions for *hasLog=False* are gained by setting every occurrence of $ln(n)$ to 1 while keeping every $log(n)$ intact. The value of *hasLog* is determined by choosing set assumes *hasLog=True* while the other set

assumes *hasLog=False*. The set closer to the measurements determines the value of hasLog. The two assumptions are compared for every instance of *n* from 0 to the user-defined upper bound. The case that has more instances closer to the generated data is chosen.

$$E \sim exp\Big(\big((log(y) - log(z))\,\big(log(a_x) - log\big(a_y\big) + log(ln(y)) - log(ln(x))\big) - (log(x) - log(y))\,\big(log\big(a_y\big) - log(a_z) + log(ln(z)) - log(ln(y))\big)\big) \div \big((log(x) - log(y))\,(z - y) - (log(y) - log(z))\,(y - x)\big)\Big). \quad (9)$$

$$P \sim \Big((y - z)\,\big(log(a_x) - log\big(a_y\big) + log(ln(y)) - log(ln(x))\big) - (x - y)\,\big(log\big(a_y\big) - log(a_z) + log(ln(z)) - log(ln(y))\big)\Big) \div \big((x - y)\,(log(z) - log(y)) - (y - z)\,(log(y) - log(x))\big). \quad (10)$$

$$C \sim exp\Big(\big((z\,log(y) - y\,log(z))\,\big(y(log(a_x) - log(ln(x))) - x\big(log\big(a_y\big) - log(ln(y))\big)\big) - (y\,log(x) - x\,log(y))\,\big(z\big(log\big(a_y\big) - log(ln(y))\big) - y(log(a_z) - log(ln(z)))\big)\big) \div \big((y\,log(x) - x\,log(y))\,(y - z) - (z\,log(y) - y\,log(z))\,(x - y)\big)\Big). \quad (11)$$

3.2.4. Choosing indices *x*, *y*, *z*

The indices *x*, *y*, *z* chosen such that it minimizes the error within the model which depends on the behavior of the input. For input that behaves continuously, the error is minimized by choosing the 3 largest indices; *x=n-2*, *y=n-1*, *z=n*. This is due to the model's asymptotic nature. And for input that behaves discontinuously, the error is minimized by choosing the 3 largest indices following discontinuities. This is from the assumption that the discontinuity acts like a floor function discontinuity where the error term is usually smallest after right the discontinuities. Behavior of this type usually arise from recursive algorithms that terminate depending on some minimum value (e.g., if (n<=minimum_value): return 1+prev_answer).

For the discontinuous case, every discontinuity must be found in order to determine the indices. One way to detect discontinuities is to use the *C* approximations of consecutive points since they become ill-defined around discontinuities. This is because the formula attempts to fit a curve through a sequence of points with an instantaneous jump in between them. This results in steep singularities in the approximations. Fig. 2 graphs the consecutive *C* approximations of the function *f(x) = floor(log(x))*, and as expected, singularities occur at powers of 2. Having random noise in data would introduce these instantaneous jumps and result in nonsensible answers.

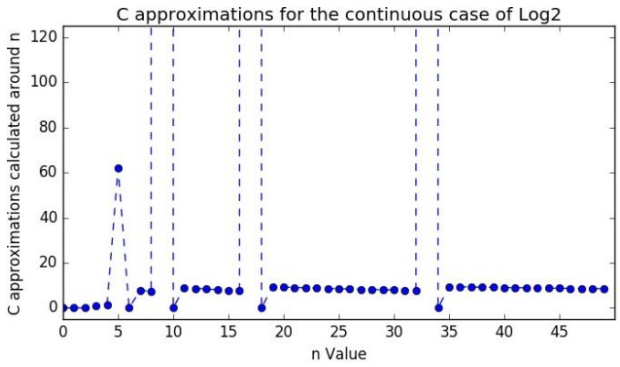

Fig. 2. Visual presentation of discontinuous behavior.

The heuristic in (12) detects discontinuities by checking if an intermediate approximation C_n lie outside the range spanned by its neighboring points. For this particular discussion, let $C_A=min(C_{n-1}, C_{n+1})$ and $C_B=max(C_{n-1}, C_{n+1})$. This heuristic tags the point on *n-1* as discontinuous if its corresponding C_n is not within the interval

enclosed by $C_A \pm C_B$. A caveat to using this heuristic is that it detects the same discontinuity twice. As seen in Fig. 2, every discontinuity has both a rising part and a falling part. The heuristic detects both them due to the symmetry of the mathematics involved. The checking for the following 2 points must be skipped in order to compensate for this problem. Lastly, if there are 3 or more discontinuities found, the method has to choose between the continuous and discontinuous case similar to how *hasLog* was determined in Section 3.2.3.

$$\neg(-|C_{n-1} - C_{n+1}| \leq C_n \leq C_{n-1} + C_{n+1}) \Longrightarrow discontinuity(n-1). \tag{12}$$

3.3. Validating Asymptotic Equivalence

The generated data a_n and calculated approximations t_1 must satisfy (3) which states that the ratio 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. The concept of convergence and divergence must be associated with some operation that can be done using only a finite number function values. An intuitive way to check the convergence of a sequence F is as follows: if the distances D's between consecutive terms are not changing or decreasing ($dD/dn=\dot{D}$, $\dot{D} \leq 0$) for the majority of the terms, then it is tagged as convergent; otherwise, it is tagged as divergent. Table 2 shows the expected results for a D whose associated condition agrees with the convergence of the various rate of growths within the scope. Note that the $sgn(\dot{D})$ values are taken as $n \to \infty$.

Table 2. Expected Results from D for Various Growth Rates

F_n	Real Interval	Convergence	$sgn(\dot{D})$
r^n	$r \leq -1$ $-1 < r \leq 1$ $r > 1$	Divergent Convergent Divergent	+1 −1, 0 +1
n^r	$r \leq 0$ $r > 0$	Convergent Divergent	−1, 0 +1
$log_r(n)$	$r < 0$ $r = 0$ $r > 0$	Divergent Convergent Divergent	+1 −1, 0 +1

Based on intuition, let the initial guess for $D=|\Delta F_n|=|F_{n+1}-F_n|$. Table 3 shows the effectiveness of this assumption. Factors that do not affect $sgn(\dot{D})$ are isolated to the right side of the center dot symbol (·). Any failures will be marked with the asterisk symbol (*).

Table 3. Expected Results from $D = |\Delta F_n|$

F_n	ΔF_n	$\frac{d}{dn}\lvert\Delta F_n\rvert$	Real Interval	$sgn(\dot{D})$
r^n	$(r-1)r^n$	$log(r)\lvert(r-1)r^n\rvert$	$r \leq -1$ $-1 < r \leq 1$ $r > 1$	+1 −1, 0 +1
n^r	$(r)n^{r-1}$	$(r-1)\lvert r\rvert \cdot \frac{1}{n^2}$	$r \leq 1$ $r > 1$	−1, 0* +1
$log_r(n)$	$log_r\left(1+\frac{1}{n}\right)$	$\frac{-1}{\lvert log(r)\rvert} \cdot \frac{1}{n(n+1)}$	$r < 0$ $r = 0$ $r > 0$	−1, 0* −1, 0 −1, 0*

As seen in Table 3, the intuitive condition gives correct results in most cases. However, it fails when considering some interval for radical and logarithmic growth. The conditions must be modified in order yield correct results. By observing the $F_n=n^r$ row, the reason why it fails 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 ΔF_n. Let the modified guess for $D=|n\,\Delta F_n|=|n\,(F_{n+1}-F_n)|$. Table 4 shows the effectiveness of this modified assumption.

Table 4. Expected Results from $D = |n\Delta F_n|$

F_n	$n\,\Delta F_n$	$\frac{d}{dn}\|n\,\Delta F_n\|$	*Real Interval*	$sgn(\dot{D})$
r^n	$n(r-1)r^n$	$\left(log(r)+\frac{1}{n}\right)\|(r-1)r^n\|\cdot n$	$r \le -1$ $-1 < r \le 1$ $r > 1$	$+1$ $-1, 0$ $+1$
n^r	$(nr)n^{r-1}$	$r \cdot \|r\|n^{r-1}$	$r \le 0$ $r > 0$	$-1, 0$ $+1$
$log_r(n)$	$n\,log_r\left(1+\frac{1}{n}\right)$	$\frac{log\left(1+\frac{1}{n}\right)-\frac{1}{n+1}}{\|log(r)\|}$	$r < 0$ $r = 0$ $r > 0$	$+1$ $-1, 0$ $+1$

As shown in the Table 4, the modified guess fits the convergence for each of the considered F_n. The same intuition about why ΔF_n should work is applicable to $n\Delta F_n$, but the distances between consecutive terms must not only become smaller, but also be smaller by an added scaling factor. This factor will help detect the divergence of slowly progressing sequences. This condition is used to determine if the ratio of the measured growth and approximated growth progresses too fast to converge to 1. This condition is written terms of discrete indices in (13) . The heuristic checks this condition for all choices of *x*, *y*, *z* from *n=0* to the user-defined upper bound. Then, it compares the number of instances that are convergent and divergent. If there are more instances that are divergent, then the limit of the ratio of a_n and t_1 is said to be divergent.

$$\neg\left(y\left|\frac{a_x}{t_1(x)}-\frac{a_y}{t_1(y)}\right| \le z\left|\frac{a_y}{t_1(y)}-\frac{a_z}{t_1(z)}\right|\right) \Rightarrow divergent(\{x,y,z\}). \tag{13}$$

It is worth noting that this is merely a heuristic and not an actual convergence test. This condition is merely a byproduct of the scope. Using this condition on very slowly diverging functions such as *log(log(n))* will result in a false positive convergence. In other scopes, one may have to resort to more aggressive modifications to D to bring about its expected behavior.

4. Testing, Results, and Analysis

A prototype was created where the method discussed in the Sections 2 and 3 was implemented and tested on both iterative and recursive algorithms, as well as mathematical functions. The algorithms used for testing are listed in Appendix I. The prototype runs the functions defined as *f(n)* for various values of *n*. Note that "for x in range(n)" in Python is a loop from 0 to *n−1*. The sample algorithms were chosen due to the complexity of their logical structure. The tests started from simple algorithms with very few recursions and loops. The latter tests were on 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 (i.e., 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. This improves accuracy by removing unnecessary constants.

Tables 5 summarizes the results from the testing and it has been rounded off to 6 decimal places. The Actual Algorithm Output refers to the output of the prototype while the Expected Algorithm Output refers to the output from manual apriori calculations done on the algorithms in Appendix I. Each algorithm is run for varying input sizes depending on their execution time. The δ Error refers to the difference between the actual output and the expected output that serves as a metric for accuracy.

It is worth noting that almost all error values are small. 28 cases out of 30 have approximations that are accurate up to at least 2 decimal places. Furthermore, the accuracy of the approximations can be 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 the

fastest growing term to overwhelm the contributions of the negligible terms. This makes the fastest growing term easier to observe and isolate. This also implies that if the fastest growing term grows relatively close to the rest of the terms, it will require more time to make the rest of the terms negligible. This can be seen in the results of *MergeSort(n)* which roughly follows the growth *(11.5n) log(n)+10n.* In this case, the first order term (*11.5n log(n)*) and the second order term (*10n*) grows relatively close to each other. This 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 of the terms, then the approximations will approach to the answer much more rapidly. This can be seen in the results of *Fibo(n)* and *Catalan(n)* where the first term grows exponentially faster than the rest of the terms. This results in answers that are accurate up to 6 decimal places while only using the first 50 and 25 terms respectively.

The Actual Algorithm Output was determined without evaluating logical structures. 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 [7], while QuinticFibo(n) is a linear recurrence relation with a corresponding fifth 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 expected *E* and *P* values for *QuinticFibo(n)*. The expected *C* value, on the other hand, was not evaluated (NE) due to the difficulty in working with complex numbers and the discrepancies that will be encountered when dealing with mere approximations. Despite these problems in computing for the expected *C* value, the developed method did not encounter any problem in computing for the actual *C* value. The accuracy seen from all the experiments conducted serves as a sufficient justification to conclude that the method is effective in determining asymptotic behaviors.

5. 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 formulas 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 growth 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. The prototype was also tested on mathematical functions of *n*.

Although the rate at which the approximation converges does vary, it is still true that the approximations will eventually approach to the exact answer. Generally, one could gain closer approximations to the exact answer by generating more and more terms. The developed method shows that one could extract asymptotic behaviors without any analysis on the logical structure of an algorithm. However, the current method 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, iterated logarithmic growths, or complex function compositions such as $2^{\log(n)}$. 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 that governs the progression of a sequence. This is because algorithms, despite running within real positive time, may contain negative or imaginary components that eventually vanish or cancel out. Although there is no concept of imaginary or negative time in running time, it is necessary to consider it in the calculations since it may have an effect on the real positive part of an algorithms running time.

Extending the output to deal with more than one term is should also be considered. This may lead to a method that outputs exact time complexities. One may use this method to get the largest growing term. Then, subtracting

it to the original sequence and apply the method once more to get the next term, and so on. In theory, this could allow one to discover all the terms of the time complexity of an algorithm.

Lastly, the method often returns inaccurate approximations in cases where the input data is contaminated by noise or depends on some measure of randomness which was discussed in Section 3.2.4. This problem stems from the use small localized regions to make inferences. More robust methods may be developed by introducing redundancies through utilizing several other points for calculating the approximations.

Table 5. Test Results

Algorithm	Expected Algorithm Output	Actual Algorithm Output	δ Error (Actual – Expected)
Iter1(n) # of terms: 20000	E=1 P=0 C=41 hasLog=False	E=1 P=0 C=41 hasLog=False	δE=0 δP=0 δC=0 δhasLog=None
Iter2(n) # of terms: 20000	E=1 P=1 C=3 hasLog=False	E=1.000000 P=1.000000 C=3.000000 hasLog=False	δE=0 δP=0 δC=0 δhasLog=None
Iter3(n) # of terms: 20000	E=1 P=1 C=4 hasLog=False	E=1.000000 P=1.000000 C=4.000000 hasLog=False	δE=0 δP=0 δC=0 δhasLog=None
Iter4(n) # of terms: 20000	E=1 P=1 C=15 hasLog=False	E=1.000000 P=1.000000 C=15.000000 hasLog=False	δE=0 δP=0 δC=0 δhasLog=None
Iter5(n) # of terms: 5000	E=1 P=2 C=2 hasLog=False	E=1.000000 P=1.999600 C=2.006826 hasLog=False	δE=0 δP=-0.0004 δC=0.006826 δhasLog=None
Iter6(n) # of terms: 5000	E=1 P=2 C=1 hasLog=False	E=1.000000 P=1.999200 C=1.006836 hasLog=False	δE=0 δP=-0.0008 δC=0.006836 δhasLog=None
Iter7(n) # of terms: 5000	E=1 P=3 C=2 hasLog=False	E=1.000000 P=2.999600 C=2.006830 hasLog=False	δE=0 δP=-0.0004 δC=0.00683 δhasLog=None
Iter8(n) # of terms: 5000	E=1 P=3 C=0.333333 hasLog=False	E=1.000000 P=2.999999 C=0.333337 hasLog=False	δE=0 δP=-0.000001 δC=0.000004 δhasLog=None
IterFibo(n) # of terms: 20000	E=1 P=1 C=4 hasLog=False	E=1.000000 P=1.000100 C=3.996040 hasLog=False	δE=0 δP=0.0001 δC=-0.00396 δhasLog=None
FactLike(n) # of terms: 30	E=divergent P=divergent C=divergent hasLog=n/a	Converges= False	Correct divergence detection
Log2(n) # of terms: 20000	E=1 P=0 C=2.885390 hasLog=True	E=1.000000 P=0.000035 C=2.884487 hasLog=True	δE=0 δP=-0.000035 δC=-0.000903 hasLog=None
Fibo(n) # of terms: 50	E=1.618034 P=0 C=1.788854 hasLog=False	E=1.618034 P=0.000000 C=1.788854 hasLog=False	δE=0 δP=0 δC=0 δhasLog=None
Ack(0, n) # of terms: 20000	E=1 P=0 C=3 hasLog=False	E=1.000000 P=0.000000 C=3.000000 hasLog=False	δE=0 δP=0 δC=0 δhasLog=None
Ack(1, n) # of terms: 20000	E=1 P=1 C=5 hasLog=False	E=1.000000 P=1.000000 C=5.000000 hasLog=False	δE=0 δP=0 δC=0 δhasLog=None
Ack(2, n) # of terms: 5000	E=1 P=2 C=5 hasLog=False	E=1.000000 P=1.998561 C=5.061660 hasLog=False	δE=0 δP=-0.001439 δC=0.06166 δhasLog=None
Ack(3, n) # of terms: 40	E=4 P=0 C=106.666667 hasLog=False	E=4.000000 P=0.000000 C=106.666666 hasLog=False	δE=0 δP=0 δC=-0.000001 hasLog=None
Quintic Fibo(n) # of terms: 40	E=1.927562 P=0 C=NE hasLog=False	E=1.927562 P=0.000000 C=0.183563 hasLog=False	δE=0 δP=0 δC=NE δhasLog=None
Linear Search(n) # of terms: 20000	E=1 P=1 C=2 hasLog=False	E=1.000000 P=1.000000 C=2.000000 hasLog=False	δE=0 δP=0 δC=0 δhasLog=None
Binary Search(n) # of terms: 20000	E=1 P=0 C=7.213475 hasLog=True	E=1.000000 P=-0.006035 C=7.959977 hasLog=True	δE=0 δP=-0.006035 δC=0.746502 δhasLog=None
Bubble Sort(n) # of terms: 5000	E=1 P=2 C=1.5 hasLog=False	E=1.000000 P=1.999867 C=1.501705 hasLog=False	δE=0 δP=-0.000133 δC=0.001705 δhasLog=None
Insertion Sort(n) # of terms: 5000	E=1 P=2 C=1.5 hasLog=False	E=1.000000 P=1.999600 C=1.505114 hasLog=False	δE=0 δP=-0.0004 δC=0.005114 δhasLog=None
Merge Sort(n) # of terms: 20000	E=1 P=1 C=11.541560 hasLog=True	E=1.000000 P=0.988068 C=14.059309 hasLog=True	δE=0 δP=-0.011932 δC=2.517749 δhasLog=None
Quick Sort(n) # of terms: 5000	E=1 P=2 C=2 hasLog=False	E=1.000000 P=1.998203 C=2.030848 hasLog=False	δE=0 δP=-0.001797 δC=0.030848 δhasLog=None
Selection Sort(n) # of terms: 5000	E=1 P=2 C=1.5 hasLog=False	E=1.000000 P=1.999334 C=1.508533 hasLog=False	δE=0 δP=-0.000666 δC=0.008533 δhasLog=None
Piece wise(n) # of terms: 5000	E=1 P=2 C=2 hasLog=False	E=1.000000 P=2.000004 C=1.999942 hasLog=False	δE=0 δP=0.000004 δC=-0.000058 δhasLog=None
Catalan(n) # of terms: 25	E=3 P=0 C=3.666667 hasLog=False	E=3 P=0 C=3.666667 hasLog=False	δE=0 δP=0 δC=0 δhasLog=None
Tetration(n) # of terms: 40	E=divergent P=divergent C=divergent hasLog=n/a	Converges= False	Correct divergence detection
$\sqrt{n}!$ # of terms: 50	E=divergent P=divergent C=divergent hasLog=n/a	Converges= False	Correct divergence detection
$(n) \cdot floor(ln(n))$ # of terms: 10000	E=1 P=1 C=1 hasLog=True	E=1.000000 P=1.000079 C=0.999418 hasLog=True	δE=0 δP=0.000079 δC=-0.000582 δhasLog=None
$(n^{-10/27}) \cdot 1.2017^n$ # of terms: 1000	E=1.2017 P=-0.370370 C=1 hasLog=False	E=1.201700 P=-0.370370 C=1.000000 hasLog=False	δE=0 δP=0 δC=0 δhasLog=None

Appendix

Appendix I. Listings for the Test Algorithms

```
################### Ack_0_n_.py ####################
def f(n):
    return ack(0, n)

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

################### Ack_1_n_.py ####################
def f(n):
    return ack(1, n)

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

################### Ack_2_n_.py ####################
def f(n):
    return ack(2, n)

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

################### Ack_3_n_.py ####################
def f(n):
    return ack(3, n)

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

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

def f(n):
    binarySearch(range(n), n-1)

################### BubbleSort.py ##################
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]
def f(n): #bubble sort worst case
    bubble_sort(range(n,0,-1))
```

```
################### Catalan.py #####################
def f(n):
    if n <=0:
        return 1
    res = 0
    for i in range(n):
        res += f(i) * f(n-i-1)
    return res

################### FactLike.py ####################
def f(n):
    if n==0:
        return 1
    else:
        for i in range(n):
            f(n-1)

#################### Fibo.py #######################
def f(n):
    fibo(n)

def fibo(n):
    if(n<=2):
        return 1
    else:
        return fibo(n-1) + fibo(n-2)

################ InsertionSort.py ##################
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

def f(n): #insertion sort worst case
    insertion_sort(range(n,0,-1))

#################### Iter1.py ######################
def f(n):
    for i in range(20):
        print i

#################### Iter2.py ######################
def f(n):
    for i in range(n):
        x=i+1
        print x

#################### Iter3.py ######################
def f(n):
    for i in range(2*n):
        add_i_n=i+n

#################### Iter4.py ######################
def f(n):
    for i in range(5*n):
        a=i+n
        b=a*n



#################### Iter5.py ######################
def f(n):
    for i in range(n):
        for j in range(n):
            mult = i*j

#################### Iter6.py ######################
def f(n):
    for i in range(n):
        sum_to_i=0
```

```
        for j in range(i):
            sum_to_i+=i

##################### Iter7.py #####################
def f(n):
    for i in range(n):
        for j in range(n):
            for k in range(n):
                tuple = (i, j, k)

##################### Iter8.py #####################
def f(n):
    big_nested_sum=0
    for i in range(n):
        for j in range(i):
            for k in range(j):
                big_nested_sum+=k

################## IterFibo.py ###################
def f(n):
    nm1=1
    nm2=0
    for i in range(n-1):
        temp=nm1+nm2
        nm2=nm1
        nm1=temp
    return nm1

################ LinearSearch.py ################
def linearSearch(alist, item):
    for i in range(0, len(alist)):
        if alist[i] == item:
            return i
    return -1

def f(n):
    linearSearch(range(n), n-1)

##################### Log2.py ####################
def f(n):
    if(n<=2):
        return 1
    else:
        return f(n/2.0)

################## MergeSort.py ##################
def merge_sort(items):
    if len(items) > 1:
        mid = len(items) // 2
        left = items[0:mid]
        right = items[mid:]
        merge_sort(left)
        merge_sort(right)
        l = 0
        r = 0
        for i in range(len(items)):
            if l < len(left):
                lval = left[l]
            else:
                lval = None
            if r < len(right):
                rval = right[r]
            else:
                rval = None
            if ((lval is not None and
                 rval is not None and lval < rval)
                 or rval is None):
                items[i] = lval
                l += 1
            else:
                items[i] = rval
                r += 1

def f(n): #merge sort worst case
    merge_sort(range(n,0,-1))

################## Piecewise.py ##################
def f(x):
    loops = 0
    if x<100:
        loops = x*5
    else:
        if x<200:
            loops = x*10
        else:
            if x<300:
                loops = x*15
            else:
                loops = x*20
    for i in xrange(loops):
        t=0

################## QuickSort.py ##################
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)

def f(n): #quick_sort sort worst case
    quick_sort(range(n,0,-1))

################## QuinticFibo.py ################
def f(n):
    if(n<=5):
        return 1
    else:
        return f(n-1)+f(n-2)+f(n-3)+f(n-4)

################ SelectionSort.py ################
def selection_sort(lst):
    for slot in range(len(lst)-1,0,-1):
        max_idx=0
        for idx in range(1,slot+1):
            if lst[idx]>lst[max_idx]:
                max_idx = idx
        lst[slot], lst[max_idx] = lst[max_idx],
lst[slot]

def f(n): #selection sort worst case
    selection_sort(range(n))

################## Tetration.py ##################
def f(n):
    ubound = n**n
    count = 0
    while count < ubound:
        count += 1
```

References

[1] Greene, D. H., & Knuth, D. E. (2008). *Mathematics for the analysis of algorithms*. Boston: Birkhäuser.

[2] Pai, G. A. (2008). Data structures and algorithms: concepts, techniques and applications. New Delhi: Tata McGraw-Hill.
[3] Gossett, E. (2009). *Discrete mathematics with Proof*. Hoboken: Wiley.
[4] Abel, N. (1826). Beweis der Unmöglichkeit, algebraische Gleichungen von höheren Graden als dem vierten allgemein aufzulösen. *Journal für die reine und angewandte Mathematik (Crelles Journal),1826*(1), 65-84. doi:10.1515/crll.1826.1.65
[5] McNamee, J., McNamee, J., & Pan, V. (2013). *Numerical Methods for Roots of Polynomials - Part II*. Oxford: Elsevier Science.
[6] Rabinovich, S., Berkolaiko, G., & Havlin, S. (1996). Solving nonlinear recursions. *Journal of Mathematical Physics, 37*(11), 5828-5836. doi:10.1063/1.531702
[7] Ackermann, W. (1928). Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen, 99*(1), 118-133. doi:10.1007/bf01459088
[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2014). *Introduction to algorithms*. Cambridge, MA: The MIT Press.
[9] Graham, S. L., Kessler, P. B., & Mckusick, M. K. (1982). Gprof. *ACM SIGPLAN Notices, 17*(6), 120-126. doi:10.1145/872726.806987
[10] McGeoch, C., Sanders, P., Fleischer, R., Cohen, P. R., & Precup, D. (2002). Using Finite Experiments to Study Asymptotic Performance. *Experimental Algorithmics Lecture Notes in Computer Science,* 93-126. doi:10.1007/3-540-36383-1_5
[11] McConnell, J. J. (2008). *Analysis of algorithms: an active learning approach*. Sudbury: Jones and Bartlett.
[12] Sedgewick, R., & Flajolet, P. (2013). *An introduction to the analysis of algorithms*. Upper Saddle River, NJ: Addison-Wesley.
[13] Montesinos Santalucía, V., Zizler, P., & Zizler, V. (2015). *An Introduction to Modern Analysis*. Springer International Publishing. doi:10.1007/978-3-319-12481-0

John Paul S. Guzman received his Bachelor's Degree in Computer Science from De La Salle University-Manila on August, 2016. His current research interests are machine learning and quantum computing. He aims to develop the rigorous theoretical foundations for deep learning techniques.

Ms. Teresita C. Limoanco is an Assistant Professor of the College of Computer Studies (CCS) of De La Salle University-Manila. A PhD candidate in CS whose thesis investigates the use of learner agents in facilitating and supporting learners' scholastic activity to learn how to program using C language. She hopes that learner agents can also be effective to support social interaction in cooperative learning approach. In recent years, she is also studying different pedagogical approaches that would deem suited to students who are first time learners in programming at the University level. Other research interests include Analysis of Algorithms, Data Structures and Algorithms, Agency and Automata Theory.