A New Paradigm for the Computational Complexity Analysis of Algorithms and Functions

Ahmed Tarek

***Abstract*—** There exists a variety of techniques for the computational complexity analysis of algorithms and functions. This analysis is foundational to the algorithmic and the functional performance. Asides from the big-oh complexity, there are other complexity notations, such as Ω, Θ, *small* o and *small* ω notational complexities. Complexity analysis is used to select an appropriate algorithm for solving a given problem using computer. Unfortunately, most of the prevailing approaches lack in simplicity, and consistency. Existing techniques are complex, and rather hard to realize in practice. There is a trend to exploit the notational complexities in the existing literature by treating those as functions instead of sets. In this paper, notational complexities and their applications paradigms are studied from the new perspectives. Simplified and consistent approaches are introduced that will make the analysis even simpler and easy to understand. Abused notational complexities are analyzed through the appropriate approach. Also, the paradigm introduced is extended to algorithms and functions involving multiple input variables.

***Keywords*—** Time Complexity, Space Complexity, Complexity Function, Notational Complexity, Complexity Paradigm, Algorithm Analysis.

I. Introduction

The time taken by an algorithm largely depends on the hardware architecture (processor model, computer organization, etc.), and also on the software configuration (the operating system, the particular compiler used, etc.) that are used to implement the algorithm. Therefore, the actual number of the CPU cycles used is not required for analyzing the algorithm. Notational complexities are independent of hardware and software architectures used for the implementation. However, the constant coefficients in the expressions for the complexity function, $f(n)$ or $f(n_1, n_2, \ldots, n_m)$ depend on the hardware and the software characteristics used for the implementation. Either $g(n)$ or $g(n_1, n_2, \ldots, n_m)$, which depends on the number of independent variables used in the notational complexities, is free from the constant coefficients; therefore, the notational complexities are independent of hardware and software characteristics used for the implementation.

With new algorithms, time and space complexity analysis are imperative to show improvements over the existing techniques and to establish their usefulness. Regardless of how fast computers become or how cheap memory gets, efficiency will always remain as an important decision factor [2]. *Big-oh* notation provides an estimate on the upper-bound of an algorithm or a function. In literature, the big-oh complexity is often confused to be a function. In fact, the big-oh complexity is not a function, and it is a *set*. Existing literature streamlines multiple techniques for analyzing the notational complexities. However, these methods are not general, and may be applied only to a limited class of algorithms and functions. As a result, a unified paradigm for the big-oh complexity becomes almost impossible. Some of these diverse techniques are rather difficult to understand and hard to realize in practice. In this paper, an extended framework for the generalized analysis of algorithms, functions, and the complexity expressions is presented. The proposed paradigm may as well be applied to functions involving multiple variables as well.

Big-oh complexity provides with an estimate of the upper bound on the computational resources required to implement the algorithm. On the other hand, Ω estimate provides with an idea of the minimum computational resources requirements. Big-oh complexity may be combined with the Ω-notational complexity to express both of these using a common Θ-notation. Θ-notational complexity provides with a general estimate of the computational resources requirements. Other notational complexities, their significance and applications are also discussed. These are *small o-notational* and ω notational complexities. The essential difference among notions for expressing the complexity are also considered. A part of the analytical tools and techniques are based on the set theoretic terms and notations.

In section II, terminology and notations are discussed briefly. Section III explores the basic steps involved in applying the paradigm. Section IV discusses application of the proposed paradigm to multi-variable functions. Section V considers the paradigm with fractional expressions. Related examples are also considered. Section VI explores application of the paradigm to addition, multiplication and division of functions. Section VII provides with a detailed guideline on the structured complexity analysis. Section $VIII$ considers an algorithm with the proposed paradigm to illustrate the paradigm. Section IX is the set theoretic approach to complexity analysis. Other notational complexities are also considered. Section X uses the paradigm in the space complexity analysis. Related examples are also considered. Section XI explores future research avenues in Numerical Algorithms and Operations Research using the paradigm.

II. Terminology and Notations

In this paper, following notations are used.

n: Input size.

$g(n)$: Highest order term in the expression for the complexity function without coefficients.

$f(n)$: Complexity function for a problem with the input size, n.

This work was supported by the California University of Pennsylvania.

Ahmed Tarek is affiliated with the Department of Math and Computer Science at California University of Pennsylvania, 250 University Avenue, California, Pennsylvania 15419, USA (phone: (724) 938-4127; fax: (724) 938-5972; e-mail: tarek@cup.edu)

$f(n_1, n_2, \ldots, n_m)$: Complexity function involving m independent variables. Here, m is an integer, and $m = 2, 3, \ldots$.
$\acute{f}(n)$: Complexity function derived by eliminating the constant coefficients from $f(n)$.
$\acute{f}(n_1, n_2, \ldots, n_m)$: Efficiency function obtained by removing the constant coefficients from $f(n_1, n_2, \ldots, n_m)$.
$g(n_1, n_2, \ldots, n_m)$: Highest order term in $\acute{f}(n_1, n_2, \ldots, n_m)$.
$O(g(n))$: Big-oh complexity with the problem size, n.
$O(g(n_1, n_2, \ldots, n_m))$: Big-oh complexity for an algorithm or a function involving m independent parameters. Here, $m = 2, 3, \ldots$.
The big-oh complexity has been defined in the literature [1] as:
If f and g are functions on the size of a problem n, or the set of parameters $n_1, n_2, \ldots, n_m$, $m = 1, 2, \ldots$ involved, then $f(n)$ is $O(g(n))$, or $f(n_1, n_2, \ldots, n_m)$ is $O(g(n_1, n_2, \ldots, n_m))$ provided that there exists constants C, and k such that:
$|f(n)| \leq C|g(n)|$, whenever $n > k$
or $|f(n_1, n_2, \ldots, n_m)| \leq C|g(n_1, n_2, \ldots, n_m)|$, whenever each of $n_1, n_2, \ldots, n_m > k$. Here, k is an integer representing the *threshold value* for the big-oh notational analysis to hold true. Following hierarchical relationship forms a basis for determining the order of complexity as outlined in [3].
$log_2 n < n < nlog_2 n < n^2 < n^3 < \ldots < n^k < 2^n < C^n < n!$

III. An Extended Paradigm for Big-Oh

Following steps applies to determining the *big-oh complexity*.
Step 1**:** Find out the complexity function $f(n)$ on the input size n.
Step 2**:** Remove all coefficients of the terms in $f(n)$ and obtain the *modified complexity function,* $\acute{f}(n)$. If necessary, also expand and consider the *nested functions* (if any) within the expression for $f(n)$.
Step 3**:** Find out the highest order term in $\acute{f}(n)$, and express it as $g(n)$. If there are nested functions in $\acute{f}(n)$, the final expression for $g(n)$ is obtained by expanding the nested functions, eliminating the constant coefficients, and then finally refining the initial expression for $g(n)$.
Step 4**:** Big-oh notational complexity is, $O(g(n))$.
The above steps are applied to the general polynomial function, $f(n) = a_k n^k + a_{k-1} n^{(k-1)} + \ldots + a_1 n + a_0$; here $a_k \neq 0$, $k = 0, 1, 2, \ldots$ and $a_k, a_{k-1}, \ldots, a_0$ are constant coefficients. Applying the 2nd step and removing all constant coefficients from $f(n)$, the modified complexity function, $\acute{f}(n) = n^k + n^{(k-1)} + \ldots + 1$, where k is a nonnegative integer, and $k = 0, 1, 2, \ldots$. Using step 3, $g(n) = n^k$, $k = 0, 1, 2, \ldots$. Therefore, the complexity order of $f(n)$ is, $O(n^k)$.

Example 1: Applying the above procedural steps to the arithmetic progression: $1^2 + 2^2 + ... + m^2$ (here m replaces n in the general model). Now, $f(m) = 1^2 + 2^2 + ... + m^2 = \frac{m(m+1)(2m+1)}{6} = \frac{1}{3} \times m^3 + \frac{1}{3} \times m^2 + \frac{1}{6} \times m^2 + \frac{1}{6} \times m = \frac{1}{3} \times m^3 + \frac{1}{2} \times m^2 + \frac{1}{6} \times m$. Removing the constant coefficients, the modified complexity function, $\acute{f}(m) = m^3 + m^2 + m$. The highest order term in $\acute{f}(m)$ is, m^3. Therefore, $g(m) = m^3$ and $f(m) \in O(m^3)$.

Example 2: Consider a logarithmic function with a nested structure within the expression for $f(x)$:
$f(x) = (x^3 + 2x)log_2(x^4+2x) = x^3 log_2(x^4+2x) + 2x log_2(x^4+2x)$. Removing the constant coefficients, the *modified complexity function,* $\acute{f}(x)$ is, $x^3 log_2(x^4 + 2x) + x log_2(x^4 + 2x)$. The highest order term in this expression is, $x^3 log_2(x^4+2x)$. Now, $log_2(x^4 + 2x)$ is a function of, $h(x) = (x^4 + 2x)$. Removing constant coefficients, the modified function, $\acute{h}(x)$ is, $(x^4 + x)$. The highest-ordered term in $\acute{h}(x)$ is x^4. Therefore, $g(x) = x^3 log_2(x^4) = 4x^3 log_2(x)$. Removing the constant coefficient, 4, the refined $g(x)$ is, $g_r(x) = x^3 log_2(x)$. Hence, the complexity order of $f(x)$ is, $O(g_r(x))$, which is $O(x^3 log_2 x)$.

IV. Big-Oh Complexity for Multi-Variable Functions and Algorithms

With some computational algorithms, input depends on two or more variables. For example, when the input to an algorithm is a graph, we often measure the size of the input in terms of both the number of vertices, V and the number of edges, E using a two dimensional matrix. Here, the input size considers both the parameters V and E. For determining the complexity of multi-variable functions, use the following steps.
Step 1**:** Determine the complexity function $f(n_1, n_2, \ldots, n_m)$, $m = 1, 2, \cdots$ involving multiple variables.
Step 2**:** Remove constant coefficients to obtain the *modified complexity function,* $\acute{f}(n_1, n_2, \ldots, n_m)$, $m = 1, 2, \cdots$. This includes any nested function inside $f(n_1, n_2, \ldots, n_m)$, $m = 2, 3, \cdots$.
Step 3**:** Find out the highest order term in $\acute{f}(n_1, n_2, \ldots, n_m)$, $m = 2, 3, \cdots$, and express it as, $g(n_1, n_2, \ldots, n_m)$, $m = 2, 3, \cdots$. The highest order term in $\acute{f}(n_1, n_2, \ldots, n_m)$, $m = 2, 3, \cdots$ is the term with the cumulative highest total power for all the variables involved. If there are nested functions in $\acute{f}(n_1, n_2, \ldots, n_m)$, the final expression for $g(n_1, n_2, \ldots, n_m)$, $m = 1, 2, \cdots$ is obtained by eliminating the constant coefficients and considering also the nested expression in $g(n_1, n_2, \ldots, n_m)$, $m = 2, 3, \cdots$. Therefore, the initial $g(n_1, n_2, \ldots, n_m)$, $m = 2, 3, \cdots$ needs to be completely expanded, and refined in obtaining the final expression.
Step 4**:** The big-oh notational complexity is, $O(g(n_1, n_2, \ldots, n_m))$.

Next the above steps are applied to a polynomial function involving k variables. The polynomial function, $f(n_1, n_2, \ldots, n_k)$ with k different variables is expressed by, $f(n_1, n_2, \ldots, n_k) = a_j n_1^{j_1} n_2^{j_2} \ldots n_k^{j_k} + a_{j-1} n_1^{j_1-1} n_2^{j_2} \ldots n_k^{j_k} + a_{j-2} n_1^{j_1} n_2^{j_2-1} \ldots n_k^{j_k} + \ldots + a_1 n_1^0 n_2^0 \ldots n_k + a_0$, here $a_j \neq 0$, $j, j_1, j_2, \ldots, j_m = 0, 1, 2, \ldots$, and $a_j, a_{j-1}, \ldots, a_0$ are constant coefficients. In the total power $(j_1 + j_2 + \ldots + j_k - 1)$, there are k possible terms; for the power $(j_1 + j_2 + \ldots + j_k - 2)$, there are $(k + C(k, 2))$ possible terms; for $(j_1 + j_2 + \ldots + j_k - 3)$, there are $(k + C(k, 2) + C(k, 3))$ different terms; ..., for the total power 1, there are $(k + C(k, 2) + C(k, 3) + C(k, 4) + \ldots + C(k, k-1))$ possible terms. Here, $C(k, j)$ is the number of ways to select j different terms out of k given terms. This is true, since both $n_1^{j_1-1} n_2^{j_2-1} \ldots n_k^{j_k}$, and $n_1^{j_1-2} n_2^{j_2} \ldots n_k^{j_k}$ provides with a total power of, $(j_1 + j_2 + \ldots + j_k - 2)$. But for the total power, $(j_1 + j_2 + \ldots + j_k)$, and 0, there is only one term that corresponds to each one of these two different

powers. Therefore, index, j = [1 + 1 + k + ($k+\frac{k(k-1)}{2!}$) + ... + ($k+\frac{k(k-1)}{2!}+\frac{k(k-1)(k-2)}{3!}+\ldots+\frac{k(k-1)(k-2)\ldots(k-k+1)}{(k-1)!}$)]. Hence, the index j needs to have a value, which is [2 + $(k-1)\times C(k,1)$ + $(k-2)\times C(k,2)$ + ... + $(k-k+1)\times C(k,k-1)$], or [2 + $k(k-1)$ + $\frac{k(k-1)(k-2)}{2!}$ + ... + 1 × $\frac{k(k-1)(k-2)\ldots(k-k+1+1)}{(k-1)!}$]. This is true, since $C(k,j)$ = $\frac{k(k-1)(k-2)\ldots(k-j+1)}{j!}$, where $j!$ is the *factorial of* j. Applying Step 2 from the above and eliminating the constant coefficients, the modified complexity function is given by, $\acute{f}(n_1,n_2,\ldots,n_k) = n_1^{j_1}n_2^{j_2}\ldots n_k^{j_k} + n_1^{k_1-1}n_2^{k_2}\ldots n_k^{j_k} + \ldots + n_1^{k_1-1}n_2^{k_2-1}\ldots n_k^{j_k-1} + \ldots + n_1n_2\ldots n_k + \ldots + n_1^0n_2^0\ldots n_k$ + 1, $j, j_1, j_2, \ldots, j_k = 0,1,2,\ldots$. Therefore, the highest order term is, $g(n_1,n_2,\ldots,n_k) = n_1^{j_1}n_2^{j_2}\ldots n_k^{j_k}$. The complexity order of, $f(n_1,n_2,\ldots,n_k)$ involving k different variables is, $O(n_1^{j_1}n_2^{j_2}\ldots n_k^{j_k})$.

Example 3: Consider the function $f(x, y) = (x^2 + xy + xlog(y))^3$ involving 2 variables x and y. Now $f(x, y) = (x^2 + xy + xlog(y))^3 = (x^2 + xy + xlog(y))\times(x^2 + xy + xlog(y))\times(x^2+xy+xlog(y))$. The highest order term in each of the 3 expressions in the multiplication is, xy. Therefore, $g_1(x,y) = g_2(x,y) = g_3(x,y) = xy$. The highest ordered term in the expression for $\acute{f}(x, y)$ that contains both x and y (because in analyzing the big-oh notational complexity, if possible, it is more meaningful to show the effects of all the variables involved) is, $xy \times xy \times xy = x^3y^3$. As, $log(y) < y$ and the total combined power of x and y in x^3y^3 is, $(3+3) = 6$. This 6 is the highest combined power possible in the expression for $\acute{f}(x, y)$. Hence, the big-oh notational complexity is, $O(x^3y^3)$.

V. Big-Oh Complexity for Fractional Expressions

For the fractional expressions, following are the required steps.

Step 1**:** Obtain expression for the fractional time complexity function, $f(n)$. Suppose the numerator in $f(n)$ is, $f_1(n)$, and the denominator is, $f_2(n)$.

Step 2**:** Remove the coefficients from $f_1(n)$ and obtain, $\acute{f}_1(n)$. This may include any nested function in $f_1(n)$. Similarly, obtain $\acute{f}_2(n)$ from $f_2(n)$ by removing the constant coefficients.

Step 3**:** Find out the highest order term in $\acute{f}_1(n)$, and express it as, $g_1(n)$. If there are nested functions in, $\acute{f}_1(n)$, the final expression for $g_1(n)$ is obtained by eliminating any constant coefficient after expanding the nested function and through a complete refinement from the initial expression for $g_1(n)$. Similarly, find out the highest order term in $\acute{f}_2(n)$, and express it as, $g_2(n)$. Finally, find out the ratio, $\frac{g_1(n)}{g_2(n)}$, and denote it by, $g(n)$.

Step 4: The big-oh time complexity is, $O(g(n))$.

Example 4: Consider $f(n) = \frac{5-7n^5-4n^3-2n}{3n^2-9n+4}$. Here, $f_1(n) = 5 - 7n^5 - 4n^3 - 2n$, and $f_2(n) = 3n^2 - 9n + 4$. Therefore, $\acute{f}_1(n) = 1+n^5+n^3+n$, and $\acute{f}_2(n) = n^2+n+1$. The highest ordered term in $\acute{f}_1(n)$ is, $g_1(n) = n^5$, and that in $\acute{f}_2(n)$ is, $g_2(n) = n^2$. Hence, $g(n) = \frac{n^5}{n^2} = n^3$. The big-oh complexity is $O(n^3)$.

Example 5: Consider $f(n) = \frac{(7+9n^5+4n^3+2n)(nlog(n)+n+2)}{(9n^4-6n^3+4n)(n^2log(n)+4)}$. Here, $f_1(n) = (7 + 9n^5 + 4n^3 + 2n)(nlog(n) + n + 2)$, and $f_2(n) = (9n^4 - 6n^3 + 4n)(n^2log(n) + 4)$. Therefore, $\acute{f}_1(n) = \acute{g}_{11}(n)\times\ \acute{g}_{12}(n) = n^5 \times nlog(n) = n^6log(n)$, and $\acute{f}_2(n) = \acute{g}_{21}(n)\times\ \acute{g}_{22}(n) = n^4 \times n^2log(n) = n^6log(n)$. Hence, $g(n) = \frac{n^6log(n)}{n^6log(n)} = 1$. Therefore, the big-oh complexity is, $O(1)$, which is a constant complexity.

Next consider fractional expressions with multiple variables. Suppose that the fractional time complexity function, $f(n_1,n_2,\ldots,n_k)$, $k = 2,3,\cdots$ contains the numerator as $f_{11}(n_1,n_2,\ldots,n_k)f_{12}(n_1,n_2,\ldots,n_k)\times\ldots\ f_{1r}(n_1,n_2,\ldots,n_k)$, $k = 2,3,\cdots$. Let the denominator be: $f_{21}(n_1,n_2,\ldots,n_k)f_{22}(n_1,n_2,\ldots,n_k)\times\ldots f_{2s}(n_1,n_2,\ldots,n_k)$, $k = 2,3,\cdots$, $s = 1,2,\cdots$. Determine the modified complexity functions for both the numerator, and the denominator separately as, $\acute{f}_{11}(n_1,n_2,\ldots,n_k)\acute{f}_{12}(n_1,n_2,\ldots,n_k)\times\ldots\ \acute{f}_{1r}(n_1,n_2,\ldots,n_k)$, $k = 2,3,\cdots$, and $\acute{f}_{21}(n_1,n_2,\ldots,n_k)\acute{f}_{22}(n_1,n_2,\ldots,n_k)\times\ldots\ \acute{f}_{2s}(n_1,n_2,\ldots,n_k)$, $k = 2,3,\cdots$, $s = 1,2,\cdots$, respectively. From the modified complexity functions, the highest order terms in the numerator and also in the denominator are, $g_1(n_1,n_2,\ldots,n_k)$=$g_{11}(n_1,n_2,\ldots,n_k)$ $g_{12}(n_1,n_2,\ldots,n_k)\times\ \ldots\ g_{1r}(n_1,n_2,\ldots,n_k)$, and $g_2(n_1,n_2,\ldots,n_m)$=$g_{21}(n_1,n_2,\ldots,n_k)$ $g_{22}(n_1,n_2,\ldots,n_k)\times\ldots\ g_{2s}(n_1,n_2,\ldots,n_k)$, $k = 2,3,\cdots$, respectively. Therefore, $g(n_1,n_2,\ldots,n_k) = \frac{g_{11}(n_1,n_2,\ldots,n_k)\times\ldots g_{1r}(n_1,n_2,\ldots,n_k)}{g_{22}(n_1,n_2,\ldots,n_k)\times\ldots g_{2s}(n_1,n_2,\ldots,n_k)}$, $k = 2,3,\cdots$. The complexity order is, $O(g(n_1,n_2,\ldots,n_m))$.

Example 6: Consider $f(x, y, z)$ involving three different variables as, $f(x, y, z) = \frac{(x^4+x^2y+y^3+y^3z^2)(x^2+y+z^3)}{(x^2+xy+yz^2)(x+y+z^2)}$. Therefore, $f_1(x, y, z) = (x^4 + x^2y + y^3 + y^3z^2)(x^2 + y + z^3)$, and $f_2(x, y, z) = (x^2 + xy + yz^2)(x + y + z^2)$. Eliminating the constant coefficients both in $f_1(x, y, z)$, and in $f_2(x, y, z)$, the modified complexity functions are, $\acute{f}_1(x, y, z) = (x^4 + x^2y + y^3 + y^3z^2)(x^2 + y + z^3)$, and $\acute{f}_2(x, y, z) = (x^2 + xy + yz^2)(x + y + z^2)$, respectively. Therefore, $g_{11}(x, y, z) = y^3z^2$, and $g_{12}(x, y, z) = z^3$ Also, $g_{21}(x, y, z) = x^0y^1z^2$, $g_{22}(x, y, z) = z^2$. Finally, $g(x, y, z) = \frac{(y^3z^2)(z^3)}{(x^0y^1z^2)(z^2)} = \frac{y^3z^5}{y^1z^4} = y^2z$. Hence, the big-oh notational complexity, $O(y^2z)$ is independent of x. The computational resources requirement will be independent of x.

From the above analysis, following result is true.

Proposition 7: If $f(n)$ is $O(g_{11}(n)g_{12}(n)\ldots g_{1r}(n))$, and $h(n)$ is, $O(g_{21}(n)g_{22}(n)\ldots g_{2s}(n)))$, then $\frac{f(n)}{h(n)}$ is, $O(\frac{g_{11}(n)g_{12}(n)\ldots g_{1r}(n)}{g_{21}(n)g_{22}(n)\ldots g_{2s}(n))})$.

Proof: Follows directly from the four procedural steps in determining the big-oh notational complexity for fractional expressions as described before under this section. □

VI. Big-Oh Complexity for Addition and Multiplication

In this section, the proposed framework has been extended to the addition and the multiplication of functions. Suppose that $f_1(n)$ is $O(g_1(n))$ or $O(g_{11}(n)g_{12}(n)\ldots g_{1r}(n))$, and $f_2(n)$ is $O(g_2(n))$ or $O(g_{21}(n)g_{22}(n)\ldots g_{2s}(n)))$. It is required to find out the big-oh complexity of, $(f_1 + f_2)(n)$, and also of $(f_1f_2)(n)$. Now, $f_1(n)$ is $O(g_1(n))$. Therefore, $f_1(n) = K_1 \times g_1(n)$ + *Lower Order Terms* = $K_1 \times g_{11}(n)g_{12}(n)\ldots g_{1r}(n)$ + *Lower Order Terms*. Similarly, $f_2(n) = K_2 \times g_2(n)$ + *Lower Order Terms* = $K_2 \times g_{21}(n)g_{22}(n)\ldots g_{2s}(n)$. Therefore,

$(f_1 + f_2)(n) = f_1(n) + f_2(n) = K_1 \times g_1(n) + K_2 \times g_2(n)$ + *Other Lower Ordered Terms from* $f_1(n)$ *and* $f_2(n)$. Three cases may be possible:
(1) $g_1(n) > g_2(n)$: In this case, $(f_1 + f_2)(n)$ is $O(g_1(n))$ or $O(g_{11}(n)g_{12}(n)...g_{1r}(n))$.
(2) $g_1(n) < g_2(n)$: In this case, $(f_1 + f_2)(n)$ is, $O(g_2(n))$ or $O(g_{21}(n)g_{22}(n)...g_{2s}(n))$.
(3) $g_1(n) = g_2(n) = g(n)$: In this case, $g(n) = g_{11}(n)g_{12}(n)...g_{1r}(n) = g_{21}(n)g_{22}(n)...g_{2s}(n))$. $(f_1 + f_2)(n) = f_1(n) + f_2(n) = (K_1 + K_2)(g(n))$ + *Other Lower Ordered Terms from* $f_1(n)$ *and* $f_2(n)$, and $(f_1 + f_2)(n)$ is $O(g(n))$ or $O(g_{11}(n)g_{12}(n)...g_{1r}(n))$ or $O(g_{21}(n)g_{22}(n)...g_{2s}(n))$. This analysis may be extended to functions with multiple variables as well.

Next consider the multiplication of functions. Now, $(f_1f_2)(n) = f_1(n)f_2(n) = (K_1 \times g_1(n)$ + *Lower Order Terms*$)\times(K_2 \times g_2(n)$ + *Lower Order Terms*$) = K_1K_2g_1(n)g_2(n)$ + *Other Lower Order Terms in the product*. The modified complexity function is obtained by eliminating the constant coefficients from $(f_1f_2)(n)$. Since, $(\acute{f_1}\acute{f_2})(n) = g_1(n)g_2(n)$ + *Remaining Lower Order Terms without Coefficients*, therefore, $(f_1f_2)(n)$ is $O(g_1(n)g_2(n))$.

Example 8: Consider $f(n) = 5n^2log_2(\sum_{j=1}^{n} j) + (4n^2 + 7)log_2(n)$. Denote $n^2log_2(\sum_{j=1}^{n} j)$ by $f_1(n)$, and $(4n^2 + 7)log_2n$ by $f_2(n)$. Therefore, $f(n) = 5f_1(n) + f_2(n)$, which is the summation of two individual functions. Now, $f_1(n) = n^2log_2(\sum_{j=1}^{n} j) = n^2log_2(\frac{n(n+1)}{2}) = n^2log_2(\frac{n^2}{2} + \frac{n}{2})$. Again, $f_2(n) = (4n^2 + 7)log_2n$. Hence, $f_2(n)$ is a product of two different functions. In this case, the product of the highest ordered terms without any coefficients is, n^2log_2n. In $f_1(n)$, the product of the highest ordered terms without any coefficients is, $n^2log_2(\frac{n^2}{2} + \frac{n}{2})$. Therefore, $g_1(n) = n^2log_2(\frac{n^2}{2} + \frac{n}{2})$, and $g_2(n) = n^2log_2n$. Now, $\acute{f}(n) = 5n^2log_2(\frac{n^2}{2} + \frac{n}{2}) + n^2log_2n$ + *Lower order terms*. The order of $log_2(\frac{n^2}{2} + \frac{n}{2})$ is higher than that of log_2n. Hence, $g_1(n) > g_2(n)$. In $log_2(\frac{n^2}{2} + \frac{n}{2})$, the highest ordered term is, n^2, and $f(n)$ is $O(g_1(n))$. Now, $log_2(\frac{n^2}{2} + \frac{n}{2})$ is a function of, $h(n) = \frac{n^2}{2} + \frac{n}{2}$. Removing the constant coefficients in $h(n)$, the modified function is, $\acute{h}(n) = n^2 + n$. The highest ordered term in $\acute{h}(n)$ is, n^2. Using **case** 1, $f(n)$ is, $O(n^2\ log_2(n^2))$, which is, $O(n^2 \times 2log_2(n))$. Removing the constant 2, $f(n)$ is, $O(n^2log_2(n))$.

Analysis for the multiplication of functions can be extended to the functions involving multiple variables as well as to the multiplication of more than two functions, as discussed in the following subsection.

A. Addition And Multiplication Of Multi-variable Functions

Here, complexity of the addition and the multiplication of j different functions f_1, f_2, ..., f_j, each involving k different variables n_1, n_2, ..., n_k are considered. Here $k = 2, 3, ...$ Suppose that the highest ordered terms in f_1, f_2, ..., f_j are g_1, g_2, ..., g_j, respectively. For the addition of functions, $f(n_1, n_2, ..., n_k) = f_1(n_1, n_2, ..., n_k) + f_2(n_1, n_2, ..., n_k) + ... + f_j(n_1, n_2, ..., n_k) = K_1g_1(n_1, n_2, ..., n_k) + K_2g_2(n_1, n_2, ..., n_k) + ... + K_jg_j(n_1, n_2, ..., n_k)$ + *Lower Order Terms from all of the* j *different functions*. There are two possible cases.

Case 1**:** At least two functions out of the j different functions, g_1, g_2, ..., g_j are distinct. In that case, let $g(n_1, n_2, ..., n_k) = \max\{g_1(n_1, n_2, ..., n_k), g_2(n_1, n_2, ..., n_k), ..., g_j(n_1, n_2, ..., n_k)\}$. Therefore, $f(n_1, n_2, ..., n_k)$ is $O(g(n_1, n_2, ..., n_k))$.
Case 2**:** All functions g_1, g_2, ..., g_j are the same. In that event, let $g(n_1, n_2, ..., n_k) = g_1(n_1, n_2, ..., n_k) = g_2(n_1, n_2, ..., n_k) = ... = g_j(n_1, n_2, ..., n_k)$. Therefore, $f(n_1, n_2, ..., n_k) = (K_1 + K_2 + ... + K_j)g(n_1, n_2, ..., n_k)$ + *Lower Order Terms* $= Kg(n_1, n_2, ..., n_k)$ + *Lower Order Terms*, where $K = K_1, +K_2 + ... + K_k$. Hence, $f(n_1, n_2, ..., n_k) \in O(g(n_1, n_2, ..., n_k))$.

For multiplication, $f(n_1, n_2, ..., n_k) = f_1(n_1, n_2, ..., n_k) \times f_2(n_1, n_2, ..., n_k) \times ... \times f_j(n_1, n_2, ..., n_k) = (K_1g_1(n_1, n_2, ..., n_k)$ + *Lower Order Terms*$) \times (K_2g_2(n_1, n_2, ..., n_k)$ + *Lower Order Terms*$) \times ... \times (K_jg_j(n_1, n_2, ..., n_k)$ + *Lower Order Terms*$) = (K_1K_2...K_j)(g_1(n_1, n_2, ..., n_k) \times g_2(n_1, n_2, ..., n_k) \times ... g_j(n_1, n_2, ..., n_k))$ + *Other Lower Order Terms resulting from the multiplication*. The modified complexity function is, $\acute{f}(n_1, n_2, ..., n_k) = (g_1(n_1, n_2, ..., n_k)g_2(n_1, n_2, ..., n_k)...g_j(n_1, n_2, ..., n_k))$ + *Lower Order Terms* $= (g_1g_2...g_j)(n_1, n_2, ..., n_k)$ + *Lower Order Terms*. Hence, $g(n_1, n_2, ..., n_k) = (g_1g_2...g_j)(n_1, n_2, ..., n_k)$, and $f(n_1, n_2, ..., n_k)$ is $O(g(n_1, n_2, ..., n_k))$ or $O((g_1g_2...g_j)(n_1, n_2, ..., n_k))$.

Example 9: Consider the following function.
$f(n_1, n_2, n_3) = (n_1^3 + n_1n_2^4 + n_2^2log_2(n_1^2 + n_3^2))(n_1^23^{n_2} + n_2^2n_3)(n_1n_3^5 + n_1log_2(n_2))$. In $f(n_1, n_2, n_3)$, there are multiplications of three different functions. These are,
$f_1(n_1, n_2, n_3) = (n_1^3 + n_1n_2^4 + n_2^2log_2(n_1^2 + n_3^2))$
$f_2(n_1, n_2, n_3) = (n_1^23^{n_2} + n_2^2n_3)$
$f_3(n_1, n_2, n_3) = (n_1n_3^5 + n_1log_2(n_2))$
The highest order term in f_1 without any co-efficient is, $g_1(n_1, n_2, n_3) = n_1n_2^4$ (using the hierarchy in Section 2 on Terminology and Notations). Similarly, the highest order term in $f_2(n_1, n_2, n_3)$ is, $g_2(n_1, n_2, n_3) = n_1^23^{n_2}$, and that in $f_3(n_1, n_2, n_3)$ is, $g_3(n_1, n_2, n_3) = n_1n_3^5$. Complexity of $f(n_1, n_2, n_3)$ is $O((g_1g_2g_3)(n_1, n_2, ..., n_m))$. But $(g_1g_2g_3)(n_1, n_2, ..., n_m) = g_1(n_1, n_2, n_3) \times g_2(n_1, n_2, n_3) \times g_3(n_1, n_2, n_3) = (n_1n_2^4) \times (n_1^23^{n_2}) \times (n_1n_3^5) = (n_1^3n_2^43^{n_2}) \times (n_1n_3^5) = n_1^4n_2^4n_3^53^{n_2}$. Hence, $f(n_1, n_2, n_3)$ is, $O(n_1^4n_2^4n_3^53^{n_2})$.

In literature, there are three types of complexities. These are *best case*, *worst case*, and the *average case complexities*. Big-oh complexity defines an algorithm's upper-bound in time and memory space requirements. However, Ω-notational complexity defines the order of the minimum time and space requirements to execute the algorithm. The average case complexity considers the effects of all possible cases including the best and the worst cases as well.

VII. Prescribed Guidelines for the Complexity Analysis

Following are five common guidelines in determining the complexity function for a given algorithm or a piece of code.

1) **Loops:** The maximum running time of a loop is the single running time of the statements within the loop

including loop tests multiplied by the total number of iterations. Consider the following:

for $i = 1$ to n in step 1 **do**
  m = $m + 2$
end for

Each execution of the loop takes a constant c amount of time. The loop executes n different times. Therefore, the time complexity function is, $f(n)$ = $c \times n$. A unit storage is required to store i, a unit to store n, and finally a unit to store m. Therefore, 3 memory units or 6 bytes of fixed storage space is required, and $f_s(n)$ = 6.

2) **Nested Loops:** Loops within the loops are common in practice. For nested loops, start at the innermost loop and then analyze inside out. Total running time is the product of the sizes for all the loops. As an example, consider the following:

for $i = 1$ to n in step 1 **do**
  for $j = 1$ to n in step 1 **do**
    k = $k + 1$
  end for
end for

Here, an inner *for loop* is nested within an outer *for loop*. For each execution of the outer for loop, the inner loop executes n times. Outer loop also executes n times in total. Suppose that the assignment statement, k = $k + 1$ takes a constant time c for its execution. Therefore, the time complexity function, $f(n)$ = $c \times n \times n$ = cn^2. We need 1 memory unit to store i, one to store j, one for n, and finally one to store k. Altogether, we need 4 memory units or 8 bytes, and $f_s(n)$ = 8 = constant.

3) **Consecutive Statements:** For consecutive statements, find out expression for the total time in terms of the input parameters for executing each individual statement, each loop and each nested loop constructs. This provides the time complexity function. For example, consider the following statements.

p = p + 1;
for $i = 1$ to n in step 1 **do**
  m = $m + 2$
end for
for $i = 1$ to n in step 1 **do**
  for $j = 1$ to n in step 1 **do**
    k = $k + 1$
  end for
end for

In this example, the complexity function, $f(n)$ = c_0 + $c_1 \times n$ + $c_2 \times n \times n$ = c_0 + $c_1 n$ + $c_2 n^2$. Here, c_0 is the time required by the assignment statement: $p = p+1$, c_1 is the time required by: $m = m + 2$, and c_2 is the time consumed by the statement: $k = k + 1$. Only 1 memory unit is required to store each one of p, i, n, m, j and k. Therefore, altogether, it requires 6 integer memory units or 12 bytes. Here, $f_s(n)$ = 12.

4) **If-then-else statements:** With the *If-then-else statements*, the worst-case time complexity function is of paramount importance. The worst-case total time is the time required by the test, plus either the then part or the else part time, whichever is the larger. As an example, consider the following code:

if (x is equal to y) **then**
  return false
else
  {
  for ($m = 0$ to $m < n$ in step 1) **do**
    if (m is equal to y) **then**
      return false
    end if
  end for
  }
end if

In this example, in the worst-case, both the *if* and the *else* parts in the outer *if-else structure* will be executed. Let the time for the *if test* is c_0. Within the *else structure*, the *for loop* will be executed n different times. If each test condition within the *for loop* takes c_1 and the *if condition check* takes c_2 amount of time, then the time complexity function is, $f(n)$ = c_0 + $n \times (c_1 + c_2)$. Here, we need 2 memory units or 4 bytes for storing x and y, 2 bytes for keeping the return address from the first *if*, 2 bytes for storing n, 2 bytes for storing m, and finally, 2 bytes for saving the return address from the second *if* statement. Altogether, we will need $(4 + 2 + 2 + 2 + 2)$ = 12 bytes of memory for storing this code segment, which is a constant. Hence, in this case, $f_s(n)$ = 12.

5) **Logarithmic complexity:** An algorithm is of logarithmic complexity if it takes a constant amount of time to cut down the current problem size by a constant fraction (usually by a fraction of $\frac{1}{2}$). For instance, if it takes a constant amount of time to cut down the current problem size by a fraction of $\frac{1}{k}$, then the time complexity of the algorithm is logarithmic, which is, $O(log_k\ (n))$. From the properties of logarithm, $log_k\ (n)$ = $log_2\ (n) \times\ log_k(2)$. However, for a particular k, $log_k(2)$ is a constant. Hence, whatever is the value of the constant k, the complexity order remains the same, which is, $O(log_2\ (n))$. An example of such an algorithm is the *binary search algorithm*. Quite often, the binary search is used to find a word inside a dictionary containing n pages.

VIII. Time Complexity Analysis

Following trivial algorithm computes the power of a real number b.

procedure $Power\_R1$ (b:real; n:positive integer)
begin
y := 1.0
for j := 1 **to** n **do**
  y := $y * b$
end

Consider the complexity function for the above trivial function. Suppose that the first statement takes c_1 amount of time to execute each time. If each execution of the for loop takes c_2 time, then the time complexity function is, $f(n)$ = c_1 +

nc_2. Therefore, $g(n) = n$, and the big-oh notational complexity of the algorithm is, $O(n)$.

procedure $Power\_R2$ (b:real; n:positive integer)
begin
$y := 1.0$
$j := n$
while $j > 0$ **do**
  begin
    if $j \neq 2$***int**$(j/2)$ **then** $\{j$ is **odd**$\}$
      $y := y * b$
    $j :=$ **int**$(j/2)$
    if $j > 0$ **then**
      $b := b * b$
  end
end

With the improved version, the number of times the *while loop* executes determines the *time-complexity function,* $f_2(n)$. It is same as the number of times the statement, $j :=$ **int**$(\frac{j}{2})$ executes. Assume that the first statement consumes c_1 amount of time, the second statement consumes c_2 amount of time, and each execution of the while loop takes c_3 amount of time on the average. If the while loop executes k different times, then $2^k = n$ (assuming n is an even power of 2), or $k = log_2(n)$. Therefore, $f_2(n) = c_1 + c_2 + k \times c_3 = c_1 + c_2 + c_3 log_2(n)$. Hence, $g(n) = log_2 n$, and the complexity is, $O(log_2 n)$.

Following table shows the relative improvement encountered using the improved second version over the trivial first version, expressed in the form of a ratio with the increasing problem size.

TABLE I
RELATIVE IMPROVEMENT IN PERFORMANCE FOR USING THE IMPROVED ALGORITHMIC APPROACH.

n	V_{t_1}	V_{t_2}	$I=\frac{V_{t_1}}{V_{t_2}}$
2^{10}	$1,024$	40	25.6
2^{12}	$4,096$	48	85.3
2^{14}	$16,384$	56	292.6
2^{16}	$65,536$	64	$1,024.0$
2^{18}	$2,62,144$	72	$3,640.9$
2^{20}	$10,48,576$	80	$13,107.2$
2^{24}	$1,67,77,216$	96	$1,74,762.7$

Following depicts the improvement using a plotted curve encountered with the improved version over the trivial version for the increasing problem sizes.

From Fig. 1, with the increasing problem sizes, the improvement factor, I increases at an extremely fast rate, dictating the supremacy of the second algorithm over the first one with larger problem instances. This characteristic is one of the major deciding factors behind designing a new algorithm to solve a problem.

IX. Set Theoretic Approach to Complexity

The explanation for Ω, Θ, ω, and the small o notational complexities becomes quite simple and straight-forward using the set theoretic notions and notations. For instance, $\Omega(g(n))$ is the set of functions that includes all such functions $f(n)$ such that, $f(n) \geq c \times g(n)$ for $n > n_0$. Expressed using the set theoretic notation, this fact becomes, $\Omega(g(n))$ = $\{$ $f(n)$ | $f(n) \geq c \times g(n)\}$, $n > n_0$. Here, c is a constant. This means, $\Omega(g(n))$ is the set of all such functions $f(n)$, such that $f(n) \geq c \times g(n)$ for constant c. Similarly, $O(g(n))$ = $\{$ $f(n)$ | $f(n) \leq c \times g(n)\}$, $n > n_0$. This implies, $O(g(n))$ is the set of all such functions such that $f(n) \leq c \times g(n)$ holds true for a constant c, and beyond $n \geq n_0$.

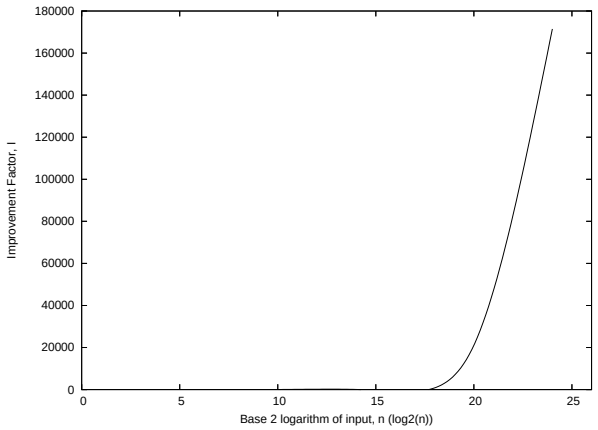

Fig. 1. The improvement factor, I is plotted against $log_2(n)$.

If there is a function $f(n)$ such that for any positive constant, c, $f(n) < c \times g(n)$, then $f(n) \in o(g(n))$. This means the function $g(n)$ defines the upper bound to the set of functions $o(g(n))$ for any positive constant, c. This is true whenever $g(n)$ is a function with a higher complexity order compared to $O(g_1(n))$, such that $f(n) \in O(g_1(n))$. Therefore, $g_1(n) < g(n)$. If there is a function $h(n)$ such that for any positive constant c, $h(n) > c \times g(n)$, then $h(n) \in \omega(g(n))$. Thus, $g(n)$ will define the lower bound to the set of functions, $\omega(g(n))$ for any positive constant, c. This is only possible, if $h(n) \in O(k(n))$, and $g(n) < k(n)$. Hence, $g(n) < k(n)$ is true for any lower order function $g(n)$ in the complexity hierarchy. If $f(n)$ is a function of n, and if there exists a positive constant c such that $f(n) \geq c \times g(n)$, then $f(n) \in \Omega(g(n))$. Therefore, $g(n)$ is the highest order function that defines the lower bound for $f(n)$. Similarly, if $f(n) \in O(g(n))$, then $g(n)$ is the lowest order function in the complexity hierarchy that defines an upper bound for $f(n)$. If there are two different constants, c_1 and c_2, such that, $c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$, then $g(n)$ defines both the lowest order upper bound and the highest order lower bound. In that event, $f(n) \in \Theta(g(n))$. Since $c_1 \times g(n) \leq f(n)$, therefore, $f(n) \in \Omega(g(n))$. Also $f(n) \leq c_2 \times g(n)$. Hence, $f(n) \in O(g(n))$. From the definition of the set intersection, $f(n) \in O(g(n)) \cap \Omega(g(n))$. Finally, $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$. This means that the set of functions, $\Theta(g(n))$ contains all such functions that lie at the set intersection of the set of functions, $O(g(n))$ and $\Omega(g(n))$.

X. Space Complexity Analysis

Storing data and programs occupy storage units or the main memory space. Processing information also needs memory. This is not the hard disk space, but the computational memory

of the computer, which is the computer's RAM. Since the computational memory is expensive, it is necessary that the programs consume as little memory as is possible, and consumes the optimum amount of it. Therefore, it is imperative to do the big-oh notational complexity analysis to find out the upper bound on memory space consumption, which also provides with an estimate on the required computational memory overhead.

Consider an example. Suppose we want to write a program to compute the sum of a given list of numbers. The program goes into a loop for a predetermined number of iterations, which is equal to the size of the list, for example 100 or 200, ... etc., asking the user to enter a number, and keeps adding these numbers. We would like to find the space complexity of this algorithm.

There are three simple steps for determining the space complexity that one should follow:

Step 1: Identify the parameter(s) that determine the problem size.

Step 2: Find out how much space (i.e. memory in bytes) is needed for a particular size.

Step 3: Find out how much space (i.e. memory in bytes) is needed for twice the size considered earlier.

Repeat step 3 many times until you establish a relationship between the size of the problem, and the corresponding space requirements. This relationship provides us with the space complexity to that program. Now, apply the above steps to the example stated above.

The problem size is obviously the size, n of the list of numbers to be added. Assume that there is a list of 100 numbers. Obviously, a variable is required where the numbers are to be entered (one at a time, may be from the keyboard), and a variable (initialized to 0) where the current running sum is to be kept. Thus, two distinct variables are required to compute the sum. Next consider a list of 200 numbers (twice the size as before). Obviously, still only two variables are required. Next consider a list having 400 numbers (twice as before). Still only two variables are required to store the results. Hence, it is possible to conclude that no matter how many numbers are added, always a constant number of variables is required for the operations, thus the relationship between the input problem size, and the space consumed (i.e. the **space complexity** of the program) is a constant. Hence, the space complexity of the given algorithm is, $S(n) \in O(1)$. Consider the following segment of code. It is required to find out the space complexity for the following code segment:

int n; $cin >> n$;
$int\ A[][]$ = $new\ int[n][n]$; $int\ B[][]$ = $new\ int[n][n]$; $int\ C[][]$ = $new\ int[n][n]$; i, j;
for ($i = 0$; $i < n$; $i++$)
    for ($j = 0$; $j < n$; $j++$)
        $cin >> C[i][j] >> B[i][j]$;
for ($i = 0$; $i < n$; $i++$)
    for ($j = 0$; $j < n$; $j++$)
        $A[i][j] = B[i][j] + C[i][j]$;

There are 3 $n \times n$ matrix required to implement the above code. The size of each matrix depends on the input size n, which is a variable. The fixed space requirement for storing i, j, and n is, $S_f = 2 + 2 + 2 = 6$.

The variable space required (depending upon the supplied value of n) is, $S_v(n) = 3 \times n \times n = 3n^2$. Hence, the total space requirement is, $S = S_f + S_v(n) = 6 + 3n^2$. Therefore, $S \in O(n^2)$. Recursive data structures have significant effect on space complexity. This is due to the fact that a recursive data structure consumes a considerable amount of dynamic storage area inside the computer's recursive stack space, which is counted as a component of $S_v(n)$. For allocating dynamic memory using the keyword *new* inside the dynamic memory area, it contributes directly to $S_v(n)$. Consider the following program sample. The first version is iterative:

template <class T>
T Sum (T a[ ], int n)
{ // Return sum of numbers a[0:$n-1$].
T tsum = 0;
for (int $i = 0$; $i < n$; $i++$)
    tsum $+=$ $a[i]$;
return tsum; }

For this algorithm, suppose that the first statement takes c_1 amount of time, each execution of the for loop consumes c_2 amount of time, and the last return statement requires c_3 amount of time. Therefore, $T(n) = c_1 + n \times c_2 + c_3$. Hence, $T(n) \in O(n)$. For the space complexity, S_s = 2+2+2+2+2 = 10, and $S_v(n) = 0$. Therefore, $S(n)$ = 10, and $S(n) \in O(1)$. The space-time bandwidth product is, $C(n) \in O(n)$, which is linear. Next consider the following recursive version of the same program.

template <class T>
T Rsum (T a[ ], int n)
{ // Return sum of the numbers a[0: $n-1$].
if ($n > 0$) return Rsum (a, $n-1$) + $a[n-1]$;
return 0; }

For the recursive version, $T(n) = c_1 \times (n+1) + c_2 \times n + c_3$. This yields, $T(n) \in O(n)$. Now the space complexity function is, $S(n) = S_s + S_v(n) = 2 + x \times n + 2 \times n$. Therefore, $S(n) \in O(n)$. Hence, the space-time bandwidth product is, $C(n) \in O(n^2)$. Thus, the computer requires more computational resources with the recursive implementation compared to the corresponding iterative version. This difference is mainly due to the space complexity of the recursive data structure.

The principal factors contributing to the dynamic memory space consumptions are due to the building up of the recursive stack space and space consumptions within the dynamic memory area. There is an interesting relationship due to Knuth that relates the time complexity to the lower bound in space complexity (the minimum space requirement).

Theorem 10: Any algorithm that takes $f(n)$ time must use at least $log(f(n))$ space.

Example 11: Suppose that the time complexity of an algorithm is, $O(n^3(log(n))^2)$. Therefore, $f(n) = n^3(log(n))^2$. Hence, the algorithm requires at least $log(n^3(log(n))^2)$ amount of memory space or more than that. However, $log(n^3(log(n))^2) = log(n^3) + log(log(n))^2) = 3log(n) + 2log(log(n))$. Now, $log(n) > log(log(n))$. Therefore, $log(n)$ provides a lower-bound on the space requirement for this algorithm. Stated in another way, $S(n) \in \Omega(log(n))$.

XI. Conclusion

Complexity helps predict whether an algorithm will take up prohibitive amount of computation time with the larger input sizes for the problems. This is specifically true with the exponential algorithms.

In this paper, a new paradigm for analyzing the computational complexity of algorithms and functions is presented. A problem may have more than one solution with each one expressed as a different algorithm. Therefore, it is necessary to compare among the performance of algorithms. After comparison, the user may select the one, that best fits his computational needs. There are several factors that are required to be considered in comparing among the algorithms. For instance, algorithms may be compared based on the computational elegance, clarity, ease of understanding, and the computational resources requirements. But the computational complexity particularly concentrates on the computational resources requirements. The cost of a computation depends on two major factors. One is determined by the technology that has been used, and the other one is the actual technique involved during the computation. The actual technique refers to the manner in which the computation is carried out. Due to the rapid technological advancements, it makes more sense to compare algorithms on the basis of the costs associated with their internal characteristics rather than the external factors. With this objective on mind, discarding the constant coefficients from the complexity function provides with the modified complexity function.

In future, applications of the proposed paradigm in Numerical Algorithms as well as in Operations Research will considered. Corresponding avenues of research will be explored to identify further the unexplored new results.

References

[1] Kenneth H. Rosen, *Discrete Mathematics and Its Applications, Fifth Edition*, McGraw-Hill, 2003

[2] Richard E. Neapolitan and Kumarss Naimipour, *Foundations of Algorithms Using C++ Pseudocode, Third Edition*, Jones and Bartlett Publishers, 2004

[3] Richard F. Gilberg and Behrouz A. Forouzan, *Data Structures - A Pseudocode Approach with C++*, Brooks/Cole, Thomson Learning, 2001