A Generalized Set Theoretic Approach for Time and Space Complexity Analysis of Algorithms and Functions

AHMED TAREK
Eastern Kentucky University
Department of Computer Science
407, Wallace Building, 521 Lancaster Avenue, Richmond, Kentucky
UNITED STATES OF AMERICA

Abstract: There exists a variety of techniques for analyzing the computational complexity of algorithms and functions. This analysis is critical in finding out the upper and the lower bounds on time and space requirements using the big-oh and the big-omega notations. Besides these, there are other complexity criteria, such as, small omega and small o complexities, which are also useful. Complexity analysis is used in selecting an appropriate algorithm for solving a given problem using computer. Unfortunately, most of the existing techniques are complex, obsolete and hard to use in practice. Besides, there is a trend to abuse notational complexities by considering them as functions. However, notational complexities are sets of functions rather than bare functions. In this paper, it has been established that the notational complexities are sets of functions that include all algorithms and functions on the given order satisfying certain constraints. Moreover, starting from the scratch, we show how to determine the time and space complexity functions for a given computational algorithm. We consider application of the proposed framework in determining the notational complexities. The proposed framework may be extended for functions involving multiple variables. Also the space-time bandwidth product has been discussed in deciding the economy of using the computational algorithms.

Key–Words: Time complexity, Space complexity, Complexity function, Complexity order, Asymptotic analysis, Performance analysis, Performance measurement, Space-time bandwidth product.

1 Introduction
Computation time and memory space requirements are the major deciding factors in computerized implementation of computational algorithms. The performance of an algorithm or a function depends on internal and external factors. The internal factors are related to the algorithm’s efficiency in terms of time and memory space usage. There are a number of external factors, which vary from platform to platform and from software to software.

Computational complexity of algorithms are dependent on internal factors. We will consider only the time and space complexity of algorithms and functions. In general, time is more crucial than space requirement. We do not determine the actual number of CPU cycles or even want to count every instruction executed in analyzing algorithms. The big-O notational complexity is introduced for this. Big-O complexity is independent of the hardware and software used. Hence, it is also called the big-O estimate. It shows the growth of time and space with the increasing problem size without taking care of the constant coefficients or smaller order terms. Sometimes, we can prove that we cannot compute something without a sufficient amount of time or space. This does not mean that we know how to compute it in this lower bound. This provides us the Omega notational complexity. Similarly, omega and small o complexities are used without tight lower and upper bounds. To determine whether a function lies at the set intersection of big-O and Omega complexities, the Theta notational complexity is used. All the notational complexities are independent of hardware and software used for the implementation.

For estimating the space-time trade-off in implementing an algorithm, we have introduced the notion of cost function, $C(n)$. The function $C(n)$ is sometimes called the space-time bandwidth product, which is expressed as the product between $g_1(n)$ and $g_2(n)$, where the big-oh time complexity is $O(g_1(n))$ and the space complexity is $O(g_2(n))$.

Section 2 introduces terminology and notation used throughout this paper. Section 3 explains the need for asymptotic complexity in complexity analysis. Different notational complexities, their definitions and examples are also considered. The relationship among the notational complexities are discussed. Sec-
tion 4 treats the set theoretic properties related to complexity analysis. Section 5 discusses complexity criteria, and introduces the complexity class hierarchy using graphs and charts. Section 6 elaborates on guidelines in determining the complexity functions. Section 7 considers a general framework in estimating the time and space complexities. The framework is extended for multi-variable functions and fractional expressions, which may also be applied to the addition, and multiplication of functions. Section 8 considers performance analysis in space complexity. We have also considered the space-time bandwidth product with examples. Section 9 discusses performance measurement in time complexity. Section 10 is application of space and time complexity in combinatorial explosion and AI.

2 Terminology and Notation

In this paper, following notations are used to discuss the proposed framework and applications.

\( n \): Input size.

\( g(n) \): Highest ordered term in modified complexity function.

\( f(n) \): Complexity function for a problem of size \( n \).

\( f(n_1, n_2, \ldots, n_m) \): Complexity function involving \( m \) different variables. Here, \( m = 1, 2, \ldots \).

\( f(n) \): Modified complexity function obtained by eliminating the constant coefficients from the complexity function \( f(n) \).

\( f(n_1, n_2, \ldots, n_m) \): The modified complexity function obtained by removing the constant coefficients from the original function \( f(n_1, n_2, \ldots, n_m) \).

\( g(n_1, n_2, \ldots, n_m) \): The highest order term in the expression for \( f(n_1, n_2, \ldots, n_m) \).

\( O(g(n)) \): Big-oh complexity for an algorithm or a function with a problem of size \( n \).

\( O(g(n_1, n_2, \ldots, n_m)) \): Big-oh complexity for an algorithm or a function involving \( m \) different parameters. Here, \( m = 1, 2, \ldots \).

\( \Omega(g(n)) \): Big-omega complexity for an algorithm or a function with a problem of size \( n \).

\( \Theta(g(n)) \): Cap-theta complexity for an algorithm or a function with input size \( n \).

\( o(g(n)) \): Small \( o \) complexity for an algorithm or a function with a problem of size \( n \).

\( \omega(g(n)) \): Small omega complexity for an algorithm or a function with a problem of size \( n \).

\( C(n) \): Space-time bandwidth product involving the highest order terms in the time and space complexity functions without coefficients for an input of size \( n \).

3 Asymptotic Complexity

The need for asymptotic complexity may best be explained using an example. Consider the complexity function, \( f(n) = 3n^2 + 14n + 27 \). The constant coefficients in this function may reflect implementation details. The lower order terms become insignificant with increasing input size \( n \) as shown in Table 1.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( 3n^2 )</th>
<th>( 14n + 17 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>31</td>
</tr>
<tr>
<td>10</td>
<td>300</td>
<td>157</td>
</tr>
<tr>
<td>100</td>
<td>30,000</td>
<td>1,417</td>
</tr>
<tr>
<td>1000</td>
<td>3,000,000</td>
<td>14,017</td>
</tr>
<tr>
<td>100000</td>
<td>300,000,000</td>
<td>140,017</td>
</tr>
</tbody>
</table>

As the input size grows bigger and bigger, the algorithmic performance converges to the highest order term. Following table verifies this fact.

Table 2: As the input size becomes larger and larger, the numerator converges to the highest order term, which is same as the denominator, and the ratio converges to 1 asymptotically.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \frac{n^2}{2} )</th>
<th>( \frac{5n^2}{2} + \frac{5n}{2} - 3 )</th>
<th>( \frac{n^2 + 5n - 3}{2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-3</td>
<td>( \infty )</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4.5</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>12.5</td>
<td>22</td>
<td>1.76</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>72</td>
<td>1.44</td>
</tr>
<tr>
<td>100</td>
<td>5000</td>
<td>5247</td>
<td>1.0494</td>
</tr>
<tr>
<td>1000</td>
<td>500,000</td>
<td>502,493</td>
<td>1.00495</td>
</tr>
<tr>
<td>10,000</td>
<td>50,000,000</td>
<td>50,025,997</td>
<td>1.00052</td>
</tr>
</tbody>
</table>

Hence, the complexity function asymptotically approaches the highest order term as the problem size grows. Following are the asymptotic definitions of \( O \) and \( \Omega \) notational complexities.

A complexity function \( f(n) \) is \( O(g(n)) \) if there exist positive constants \( c \) and \( n_0 \) such that \( f(n) \leq c \times g(n) \) for all \( n \geq n_0 \). \( O(g(n)) \) is a set of functions, which can be formally represented as follows:

\[
O(g(n)) = \{ f(n) \mid \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } f(n) \leq c \times g(n) \forall n \geq n_0 \} \tag{1}
\]
A more precise and exact definition of the $O$-notational complexity follows [3]:

**Definition 1** If $f$ and $g$ are functions on the size of a given 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 there exists constants $c$, and $n_0$ such that:

\[ |f(n)| \leq c|g(n)|, \text{ whenever } n \geq n_0 \]

or \[ |f(n_1, n_2, \ldots, n_m)| \leq c|g(n_1, n_2, \ldots, n_m)|, \text{ whenever each of } n_1, n_2, \ldots, n_m \geq n_0. \]

Here, $n_0$ represents the threshold value for the big-oh notational analysis to hold true.

Similarly, $\Omega$ notation is used to define the asymptotic lower bound. A complexity function $f(n)$ is $\Omega(g(n))$ if there exists positive constants $c$ and $n_0$ such that \(0 \leq c \times g(n) \leq f(n)\) for all $n \geq n_0$. $\Omega(g(n))$ is also a set of functions, for which, formal representation follows:

\[ \Omega(g(n)) = \{f(n) \mid \exists \text{ positive constants } c, n_0 \text{ such that } f(n) \geq c \times g(n) \forall n \geq n_0.\} \tag{2} \]

The $\Theta$ notational complexity represents a set of functions that lies at the intersection of $O(g(n))$ and $\Omega(g(n))$. A complexity function $f(n) \in \Theta(g(n))$ provided that $f(n) \in \Omega(g(n))$, and also, $f(n) \in \Omega(g(n))$. Formally, the $\Theta$ notational complexity may be defined as follows:

\[ \Theta(g(n)) = \{f(n) \mid \exists \text{ positive constants } c_1, c_2, n_0 \text{ such that } c_1 \times g(n) \leq f(n) \leq c_2 \times g(n) \forall n \geq n_0.\} \tag{3} \]

Using set theoretic notation: $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

**Example:** Consider the following set of functions, which are on the order of $O(2^n)$.

\[ O(2^n) = \{ 4^n g(n) + 5, 5n + 7, 3n \log(n) + 4, 8n^2, 3n^2 + 7, 9n^2 + 6n \} \]

Next consider the following set of functions, for which, $\Omega(n^2)$ defines the lower bound.

\[ \Omega(n^2) = \{ 8n^2, 3n^2 + 7, 9n^2 + 6n 7n^4, 32n^7, 7, 9n^3 + 5n^2 \} \]

Now $\Theta(n^2)$ defines the set of functions that lies at the intersection of $O(n^2)$ and $\Omega(n^2)$. Therefore, $\Theta(n^2) = O(n^2) \cap \Omega(n^2) = \{8n^2, 3n^2 + 7, 9n^2 + 6n\}$.

We may define $o$ and $\omega$ asymptotic complexities for those functions for which the upper and lower bounds are not tight. A function $f(n)$ is said to be $o(g(n))$, provided there exists positive constants $c$ and $n_0$ such that for any positive constant $c$ and for all input sizes $n \geq n_0$, the $f(n) < c \times g(n)$ relationship holds.

Following defines the set $o(g(n))$ formally using the set theoretic notations.

\[ o(g(n)) = \{f(n) \mid \exists \text{ positive constants } c, n_0 \text{ such that } f(n) < c \times g(n), \forall n \geq n_0 \text{ and } \forall c.\} \tag{4} \]

Since the function $f(n)$ strictly satisfies $f(n) < c \times g(n)$, with the increasing input size $n$, $g(n)$ grows bigger and bigger compared to $f(n)$, and eventually, for large input values, the asymptotic limit of the ratio between $f(n)$ and $g(n)$ approaches 0. Stated mathematically,

\[ \lim (n \to \infty) \frac{f(n)}{g(n)} = 0 \tag{5} \]

A function $f(n)$ is $\omega(g(n))$, if there exists positive constants $c$ and $n_0$ such that for positive constant $c$ and for all input sizes $n \geq n_0$, the relationship $f(n) > c \times g(n)$ holds true. Following provides the formal definition of $\omega(g(n))$.

\[ \omega(g(n)) = \{f(n) \mid \exists \text{ positive constants } c, n_0 \text{ such that } f(n) > c \times g(n), \forall n \geq n_0 \text{ and } \forall c.\} \tag{6} \]

To be a member of the set $\omega(g(n))$, $f(n)$ has to be strictly greater than the product $c \times g(n)$. Therefore, with increasing values of $n$, $f(n)$ grows much bigger compared to the function $g(n)$. Eventually, for very large values of $n$, the ratio $f(n)/g(n)$ becomes negligible for the asymptotic case. Following condition holds true.

\[ \lim (n \to \infty) \frac{f(n)}{g(n)} = 0 \tag{7} \]

**Examples:** The function $f(n) = n^2$ is $O(n^3)$, since $n^2 \leq 1 \times n^3 + 1 \times n^2$, and the highest order function becomes $g(n) = n^3$. But there is no positive real constant $c$ such that: $n^2 \leq c \times n^3$. Therefore, $n^2 \notin \Omega(n^3)$. Hence, $n^2 \in (O(n^3) - \Omega(n^3))$.

Consider another function $f(n) = 3n + 4$. Since, $3n + 4 <= 7n$, hence for all $n \geq 1$, $f(n) \in O(n)$. Again, $f(n) = 3n + 4 \geq 3n$, for all $n \geq 1$. Hence, $f(n) \in \Omega(n)$. Using the definition of $\Theta(n)$, $f(n) \in \Theta(n)$. Now, $3n + 4 \leq 3n^2 + 4n^2$ for all $n \geq 1$. Therefore, $f(n) \in O(n^2)$. Similarly, $f(n) \in O(n^3)$, $f(n) \in O(n^4)$, and so. Therefore, $O(n^2)$, $O(n^3)$, … may always weaken the asymptotic upper bound. Similarly, $3n + 4 \leq 3\log(n)$ for all $n \geq 1$, and $f(n) \in O(\log(n))$. Also, $3n + 4 \geq 3 + 3n + 4 \geq 7\log(n)$ for all $n \geq 1$, and $f(n) \in \Omega(1)$. Therefore, $\Omega(\log(n))$, $\Omega(1)$ may always weaken the asymptotic lower bound. Given a positive constant $c$, $3^n \geq c \times n$ for all but small integers $n$. Therefore, $3^n \in \Omega(n)$. But it is impossible to find a positive constant $c$ such that for higher values of $n$, $3^n \leq c \times n$. Hence, $3^n \notin O(n)$.

Given a positive $c$, $\log(n) \geq c \times 7$ for sufficiently
large values of \( n \). Therefore, \( n^{\log(n)} \in O(n^7) \). However, \( n^{\log(n)} \not\in O(n^7) \). Therefore, \( n^{\log(n)} \in (O(n^7) - O(n^7)) \).

Next consider \( f(n) = 3n + 16 \) for \( o(n) \) and \( \omega(n) \). Now, \( 3n + 16 \not\in o(n) \), and also \( 3n + 16 \not\in \omega(n) \). For \( 3n + 16 \) to be \( o(n) \), the relationship \( 3n + 16 < c \times n \) must hold for all positive constants \( c \) and any input size \( n \geq n_0 \). However, \( 3n + 16 < c \times n \) fails for many positive constants \( c \). Therefore, \( 3n+16 \not\in o(n) \).

Similarly, \( 3n + 16 > c \times n \) fails for many positive constants \( c \). Hence, \( 3n + 16 \not\in \omega(n) \). However, \( 3n + 16 < c \times n \log(n) \) is always true beyond the cut-off input size \( n_0 \), and for all real positive constants \( c \).

Therefore, \( f(n) = 3n + 16 \in o(n\log(n)) \). Similarly, \( 3n+16 \in o(n^2) \), and so. Again, \( 3n+16 > c \times \log(n) \) for all real positive constants \( c \) beyond a certain input size \( n_0 \). Hence, \( 3n + 16 \in \omega(\log(n)) \). Similarly, \( 3n + 16 \in \omega(1) \).

If we are posed with the problem of verifying whether \( \frac{1}{\log n} \in o(1) \), we may solve it using the limiting approach. In this case, \( f(n) = \frac{1}{\log n} \), and \( g(n) = 1 \).

Thus, \( \frac{f(n)}{g(n)} = \frac{1}{\log n} \). As \( n \to \infty \), \( \log n \to \infty \) and \( \frac{1}{\log n} \to 0 \). By definition, \( \lim n \to \infty \frac{f(n)}{g(n)} \) needs to be \( 0 \) for \( f(n) \in o(g(n)) \). Hence, \( \frac{1}{\log n} \in o(1) \).

4 Set Theory in Complexity Analysis

**Lemma 2** If \( \exists \) positive constants \( c \) and \( n_0 \), such that \( \forall n \geq n_0 \), then a set of functions \( \{ f(n) \} \) maps on to a set \( S(g(n)) \) of functions \( g(n) \) using the set membership \( \in \), where \( S \in \{ O, \Omega, o, \omega, \Theta \} \), and \( S \) denotes a notational complexity the notation of which depends on the relation \( R \) in \( f(n) \) \( R \in \times g(n) \). Here, \( R \) is a relation from the set \( \{ f(n) \} \) to the set \( \{ c \times g(n) \} \), where \( R \in \{ \geq, >, \leq, <, = \} \), and is a subset of the cartesian product between the sets \( \{ f(n) \} \) and \( \{ c \times g(n) \} \).

**Proof:** We prove this lemma using proof by cases.

**R is \( \geq \):** Here, \( f(n) \geq c \times g(n) \), and \( f(n) \in \Omega(g(n)) \).

**R is \( \leq \):** In this case \( f(n) \leq c \times g(n) \), and \( f(n) \in \omega(g(n)) \).

**R is \( > \):** In this case \( f(n) > c \times g(n) \), and \( f(n) \in O(g(n)) \).

**R is \( < \):** In this case \( f(n) < c \times g(n) \), and \( f(n) \in o(g(n)) \).

**R is \( = \):** In this case \( f(n) = c \times g(n) \), and \( f(n) \in \Theta(g(n)) \).

In all five of the cases, \( f(n) \in S(g(n)) \), where the exact notion of \( S \) will depend on the relation \( R \) between \( \{ f(n) \} \) and \( \{ c \times g(n) \} \).

We already know that \( O, \Omega, o, \omega \) and \( \Theta \) notational complexities are sets rather than functions. However, it is a common practice to abuse complexity notations, such as writing \( f(n) = O(g(n)) \) instead of \( f(n) \in O(g(n)) \). Also, the expressions such as \( f(n) = h(n) + O(g(n)) \) are quite common. But a function cannot be added to a set like this, and in no way a function could be equal to a set. Therefore, it is more appropriate to write: \( f(n) \in O(h(n) + g(n)) \).

![Figure 1: Notational complexities bear complimentary characteristics.](image)

From Figure 1, function \( g(n) \) defines the upper bound for \( f(n) \) whenever \( n \geq 0 \). Therefore, \( f(n) \in O(g(n)) \). Stated in another way, the function \( f(n) \) defines the lower bound for the function \( g(n) \). Hence, \( g(n) \in \Omega(f(n)) \). The figure also shows that the function \( h(n) \) defines the lower bound of \( f(n) \), and \( f(n) \in \Omega(h(n)) \). Therefore, the function \( f(n) \) must define the upper bound to the function \( h(n) \) and \( h(n) \in O(f(n)) \). Therefore, the notational complexity relationship between a given function and its bounding function is complimentary and preserves reciprocity. Following result is true.

**Theorem 3** If for a given function \( f(n) \), if \( f(n) \in O(g(n)) \), then \( g(n) \in \Omega(f(n)) \), and vice-versa.

**Proof:** Since \( f(n) \in O(g(n)) \), therefore, there exists positive constants \( c \) and \( n_0 \) such that for all \( n \geq n_0 \), \( f(n) \leq c \times g(n) \). This means, \( \frac{1}{c} \times f(n) \leq g(n) \). Denote the positive constant \( \frac{1}{c} \) by \( c_1 \). Therefore, \( g(n) \geq c_1 \times f(n) \). Hence, \( g(n) \in \Omega(f(n)) \). □

5 Complexity Criteria

Following hierarchical relationship forms a basis for determining the notational complexity order for functions and expressions [3].

\[ 1 < \log_2 n < n < n\log_2 n < n^2 < n^3 < \ldots < n^k < 2^n < c^n < n! \]
Here, \( k \) and \( c \) are positive constants, and \( b \) is a positive integer defining the base of the logarithmic functions. This hierarchical relationship is known as the *complexity class*. In terms of set theory:

\[
O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \\
\subseteq O(n^2) \subseteq O(n^3) \subseteq \ldots \subseteq O(n^k) \subseteq O(2^n) \\
\subseteq O(c^n) \subseteq O(nl) \tag{8}
\]

The above equation (8) holds true also for the \( o \) notational complexity class hierarchy. Similarly, the complexity class hierarchy for the \( \Omega \) notational complexity becomes:

\[
\Omega(1) \supseteq \Omega(\log n) \supseteq \Omega(n) \supseteq \Omega(n \log n) \\
\supseteq \Omega(n^2) \supseteq \Omega(n^3) \supseteq \ldots \supseteq \Omega(n^k) \supseteq \Omega(2^n) \\
\supseteq \Omega(c^n) \supseteq \Omega(nl) \tag{9}
\]

Equation (9) is also applicable for the \( \omega \) notational complexity class hierarchy. From these hierarchical relationships, following result follows.

**Lemma 4** For \( \Theta \) notational complexity, following results hold true.

\[
\Theta(1) \cap \Theta(\log n) \cap \Theta(n) \cap \Theta(n \log n) \\
\cap \Theta(n^2) \cap \Theta(n^3) \cap \ldots \cap \Theta(n^k) \cap \Theta(2^n) \\
\cap \Theta(c^n) \cap \Theta(nl) = \emptyset \\
\Theta(\log n) \cap \Theta(n) \cap \Theta(n \log n) \\
\cap \Theta(n^2) \cap \Theta(n^3) \cap \ldots \cap \Theta(n^k) \cap \Theta(2^n) \\
\cap \Theta(c^n) \cap \Theta(nl) = \emptyset \\
\vdots \\
\Theta(c^n) \cap \Theta(nl) = \emptyset \tag{10}
\]

Here, \( \emptyset \) denotes an empty set.

**Proof:** Consider a function \( f_1(n) \) with input size \( n \). Suppose that \( f(n) \in \Theta(1) \). In this case, \( f_1(n) \in O(1) \) and also, \( f_1(n) \in \Omega(1) \). Using the definition of \( O \)-complexity, \( f_1(n) \in O(\log n), f_1(n) \in O(n), \) and so on. However, \( f_1(n) \notin \Omega(1) \) and \( f_1(n) \notin \Omega(\log n), f_1(n) \notin \Omega(n), \) and so on. These two relationships hold true for any such function \( f(n) \). Therefore,

\[
\Theta(1) \cap \Theta(\log n) \cap \Theta(n) \cap \Theta(n \log n) \cap \Theta(n^2) \cap \Theta(n^3) \cap \ldots \cap \Theta(n^k) \cap \Theta(2^n) \\
\cap \Theta(c^n) \cap \Theta(nl) = \emptyset 
\]

Next consider another function \( f_2(n) \) such that: \( f_2(n) \in \Theta(\log n) \). Therefore, \( f_2(n) \in O(n), f_2(n) \in O(\log n), f_2(n) \in O(n), \) and so on. But \( f_2(n) \notin \Omega(n), f_2(n) \notin \Omega(\log n), \) and so on. These two relationships hold true for any such function \( f(n) \). Hence, \( \Theta(\log n) \cap \Theta(n) \cap \Theta(n \log n) \cap \Theta(n^2) \cap \Theta(n^3) \cap \ldots \cap \Theta(n^k) \cap \Theta(2^n) \cap \Theta(c^n) \cap \Theta(nl) = \emptyset \). Proceeding this way, it is possible to show that:

\[
\Theta(c^n) \cap \Theta(nl) = \emptyset 
\]

Figure 2 illustrates complexity class hierarchy. Before \( n_0 \), the hierarchy does not hold. We discard input values smaller than \( n_0 \). Beyond the input size \( n_0 \), the functions preserve the complexity class hierarchy.

\[\text{Figure 2: Graphical representation of functions in complexity class hierarchy.}\]

In literature, we find three types of complexities. These are best case, worst case, and average case complexities. Big-oh complexity defines an algorithm’s upper-bound in computation time and memory space requirements. The algorithm consumes time or space in the order of big-oh in the worst possible case. \( \Omega \)-notational complexity defines the order of the minimum time and space required to execute the algorithm. This is possible in the best case. The average case complexity considers the effects of all possible cases including the best and the worst cases as well.

### 6 Guidelines for Functions

There are five common guidelines in finding out the complexity function corresponding to an algorithm. We discuss each one of them in sequence.

1. **Loops**: Loops play a major role in most coding structure. 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 loop iterations. For
example, consider the following loop:

```plaintext
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 for \(n\) different times. Therefore, the time complexity function, \(f(n) = c \times n\). We need one memory unit for storing \(i\), one for \(n\) and one for \(m\). Therefore, altogether, we need 3 memory units or 6 bytes (fixed), and \(f_s(n) = 6\).

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

```plaintext
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
```

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

3. **Consecutive Statements**: For consecutive statements, we 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 gives us the time complexity function. For example, consider the following statements.

```plaintext
\(p = p + 1;\)
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
```

\(c_0\) is the time required by the assignment statement: \(p = p + 1\), \(c_1\) is the time required by the statement: \(m = m + 2\), and \(c_2\) is the time consumed by the statement: \(k = k + 1\). However, we need only 1 memory unit for storing \(p\), 1 for \(i\), 1 for \(n\), 1 for \(m\), 1 for \(j\) and finally, 1 for storing \(k\). Therefore, altogether we need 6 integer memory units or 12 bytes. Here, \(f_s(n) = 12\).

4. **If-then-else statements**: With If-then-else statements, we are interested in the worst-case time complexity function. 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:

```plaintext
if \((x \text{ is equal to } y)\) then
    return false
else
    \{ 
    for \((m = 0\) to \(m < n\) in step 1\) do
        if \((m \text{ is equal to } y)\) 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 of the for loop takes \(c_1\) and the if condition check takes \(c_2\) times, then the time complexity function \(f(n) = c_0 + n \times (c_1 + c_2)\). 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 \(m\), and finally, 2 bytes for the return address from the second if statement. Altogether, we need \((4 + 2 + 2 + 2 + 2) = 12\) bytes of memory for storing this code segment. In this case, \(f_s(n) = 12\).

5. **Logarithmic complexity**: An algorithm is of logarithmic complexity if it takes a constant time to cut down the current problem size by a constant fraction (usually by \(\frac{1}{2}\)). An example is the binary search algorithm. Quite often we use the binary search for finding a word inside a dictionary of \(n\) pages.

7. **A Framework for Analysis**

Once we have determined the time or the space complexity function using the guidelines described previ-
ously, we may use the following steps in determining the big-oH notational complexity.

**Step 1:** Expand the complexity function \( f(n) \) completely for an input of size \( n \).

**Step 2:** Remove all constant coefficients from the terms within \( f(n) \), and obtain the modified complexity function \( \hat{f}(n) \). If necessary, expand any nested function within the expression for \( f(n) \).

**Step 3:** Find out the highest ordered term in the expression for \( \hat{f}(n) \) and express it as \( g(n) \). If there are nested functions within \( \hat{f}(n) \), the final expression for \( g(n) \) is obtained by eliminating the constant coefficients and after refinement of the initial expression for \( g(n) \).

**Step 4:** The big-oH notational complexity is \( O(g(n)) \).

**Example:** We apply the above steps to \( f(n) = a_k n^k + a_{k-1} n^{(k-1)} + \ldots + a_1 n + a_0 \), here \( a_k \neq 0, k = 0, 1, 2, \ldots \) and \( a_k, a_{k-1}, \ldots, a_0 \) are constant coefficients. Applying the second procedural step and removing all constant coefficients from \( f(n) \), the modified complexity function, \( \hat{f}(n) = n^k + n^{(k-1)} + \ldots + 1 \), 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, \( f(n) \) is \( O(n^k) \).

The proposed framework is general, and may be applied to complexity functions involving multiple variables. An application of this in AI has been considered in section 11.

Quite often, complexity functions for algorithms are expressed as fractions. For fractional expressions, the steps described earlier need to be applied both to the numerator and to the denominator. Using the procedural steps, we find out the highest order term in the numerator as \( g_1(n) \), and that in the denominator as \( g_2(n) \). Finally, we calculate the ratio \( g_1(n)/g_2(n) \), and denote it by \( g(n) \). The big-oH complexity for the fractional expression is \( O(g(n)) \). An application of this appears at section 11.

The steps may be applied in determining the big-oH complexity for functions involving fractional expressions with multiple variables.

### 7.1 Addition and Multiplication of functions

Here, we extend our proposed framework in finding out the big-oH complexity for the addition and multiplication of functions. Suppose that \( f_1(n) = O(g_1(n)) \) and \( f_2(n) = O(g_2(n)) \). It is necessary to find out the big-oH complexity for \( (f_1 + f_2)(n) \) and \( (f_1 \times f_2)(n) \). Now, \( f_1(n) = O(g_1(n)) \). Therefore, \( f_1(n) = K_1 \times g_1(n) + \text{Lower Order Terms} \). Similarly, \( f_2(n) = K_2 \times g_2(n) + \text{Lower Order Terms} \).

Therefore, \( (f_1 + f_2)(n) = f_1(n) + f_2(n) = (K_1 \times g_1(n) + K_2 \times g_2(n)) + \text{Other Lower Ordered Terms from } f_1(n) \text{ and } f_2(n) \). Three cases are possible:

1. \( g_1(n) > g_2(n) \): \((f_1 + f_2)(n) \in O(g_1(n))\);
2. \( g_1(n) < g_2(n) \): \((f_1 + f_2)(n) \in O(g_2(n))\);
3. \( g_1(n) = g_2(n) \): \( (f_1 + f_2)(n) = f_1(n) + f_2(n) = (K_1 + K_2)(g_1(n)) + \text{Other Lower Ordered Terms, and } (f_1 + f_2)(n) \in O(g(n)) \).

Next consider multiplication of functions. Now, \((f_1 f_2)(n) = f_1(n) f_2(n) = (K_1 \times g_1(n) + \text{Lower Order Terms}) \times (K_2 \times g_2(n) + \text{Lower Order Terms}) = K_1 K_2 g_1(n) g_2(n) + \text{Other Lower Order Terms in the product} \). The modified complexity function is obtained by eliminating the constant coefficients from \((f_1 f_2)(n) \). Since, \((f_1 f_2)(n) = g_1(n) g_2(n) + \text{remaining lower order terms without coefficients} \). Therefore, \((f_1 f_2)(n) = O(g(n) g_2(n)) \).

This analysis may be extended to functions involving multiple variables as well as to the multiplication of more than two functions.

### 8 Space Complexity Analysis

The space complexity in implementing an algorithm is the amount of memory that it needs to run the algorithm to completion. Complexity function for the computational memory space requirement is denoted by \( S(P) \). Using the set theory, \( S(P) \in O(g(n)) \), where \( g(n) \in \{1, \log(n), n, n^2, n^3, \ldots, 2^n, n!\} \). Thus, \( g(n) \) is a member of the set containing all possible complexity class functions. The memory space required to implement an algorithm may be divided into two parts. These are:

1. **Fixed Space Requirements**, \( C \): This computational memory requirement is independent of the characteristics of inputs and outputs. This space includes instruction space, space for storing simple variables, fixed-size structured variable, constants, etc. This is also known as the static storage requirement.

2. **Variable storage requirement**, \( S_p(I) \): This memory space is required for the creation of dynamic data entities. This requirement depends on the instance characteristic \( I \). The number, size, values of inputs, and outputs associated with a particular instance, \( I \), recursive stack space, space for storing the formal parameters, local variables, and return addresses.

Therefore, the total computational memory space requirement, \( S(P) = C + S_p(I) \).

### 8.1 Space Complexity of Algorithms

We consider two examples of determining the space complexity function in recursive and iterative algorithms.
Example 1: At first, consider the following iterative algorithm. Function Sum adds elements inside a given list of numbers, a[0:n-1].

```cpp
template < class T >
T Sum (T a[ ], int n)
{
    T tsum = 0;
    for (i = 0; i < (n-1) in step 1 do
        tsum += a[i];
    end for
    return tsum;
}
```

The memory space required for storing the pointer to the array a[ ], integer n, index i, and the parameter tsum, each is 2 bytes or 1 integer memory unit long. Therefore, C = 42 bytes = 8 bytes. Also, S_p (I) = 0. Hence, S (P) = C + S_p (I) = 8, and S (P) ∈ O (1). Similarly, S (P) ∈ O (1). Suppose that the initial assignment statement takes c_1 time. The for loop executes n different times. If each execution of the iterative loop takes c_2 units of time, then the time complexity function is, f (n) = c_1 + c_2 × n. Therefore, f (n) ∈ O (n). The space-time bandwidth product for the function is, C (n) = 1 × n = n. Therefore, C (n) ∈ O (n).

Example 2: Following is a recursive version of the algorithm in Example 1 to add elements inside a given list of numbers, a[0:n-1].

```cpp
template < class T >
T RSUM (T a[ ], int n)
{
    if (n > 0) then
        return RSUM (a, n - 1) + a[n - 1];
    end if
    return 0;
}
```

For the given recursive program, the recursive calls are generated in the following sequence: RSUM(a, n), RSUM(a, n - 1), RSUM(a, n - 2), …, RSUM(a, 1), RSUM(a, 0). Therefore, the depth of recursion is (n + 1). Each call to RSUM needs 2 × 3 = 6 bytes. Hence, S_p (I) = 6 × (n + 1), with I = n, with n representing the instance characteristics, and C = 0. Now S (P) = C + S_p (I) = 6(n + 1). Finally, S (P) ∈ O (n). Also S (P) ∈ Ω (n). For any n > 0, the function RSUM executes n different times. If each execution of the function takes c units of time, then the time complexity function, f (n) = c × n. Hence, f (n) ∈ O (n). Therefore, the space-time bandwidth product for the function is, C (n) = n^2, and C (n) ∈ O (n^2).

9 Performance Measurement

Laboratory-based techniques play a key role in deciding the performance of algorithms. For example, from theoretical analysis, we know that the selection sort algorithm is O (n^2), and merge sort algorithm is O (n log n). But we are further interested in learning the effect of this order difference on computer-based implementation of the sorting algorithms. For selection sort, the highest order term g_s (n) in the complexity expression is n^2, and that in the merge sort is, g_m (n) = n log n. Now, the ratio between g_m (n) and g_s (n) is, r = g_m (n) / g_s (n) = n log n / n^2 = log(n) / n. As input size grows to a very large value, this ratio approaches 0. This is evident from the mathematical fact that: Lim n → ∞ (log(n) / n) = 0.

We verify this with performance measurement. The algorithms are implemented using Microsoft’s Visual C++.NET compiler under Visual Studio.NET IDE. A PC with an Intel Pentium 4, 2.80 GHz CPU and 512 MB of RAM space has been used. The result is listed below. With an input size, n = 10, the time consumed by both the algorithm is negligible (0). As the problem size grows, selection sort consumes more and more time compared to the merge sort. At n = 10000, the selection sort requires 52 seconds, whereas the merge sort still requires negligible time (0 second). Therefore, the ratio becomes 0. With increasing problem sizes, this ratio becomes smaller and smaller with the smallest ratio being 0.001671 at n = 60000. This means that for this particular input, the time required by the merge sort is only 0.1671% of the selection sort. Hence, as Lim n → ∞ (log(n) / n) = 0 holds true.

<table>
<thead>
<tr>
<th>n</th>
<th>T_selection</th>
<th>T_merge</th>
<th>T_merge/T_selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10000</td>
<td>52</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20000</td>
<td>208</td>
<td>1</td>
<td>0.00481</td>
</tr>
<tr>
<td>30000</td>
<td>472</td>
<td>1</td>
<td>0.00212</td>
</tr>
<tr>
<td>40000</td>
<td>2</td>
<td>835</td>
<td>0.0024</td>
</tr>
<tr>
<td>50000</td>
<td>1313</td>
<td>3</td>
<td>0.002285</td>
</tr>
<tr>
<td>60000</td>
<td>1795</td>
<td>3</td>
<td>0.001671</td>
</tr>
</tbody>
</table>
10 Application

In this section, we consider a few well-known algorithms in Artificial Intelligence (AI).

Example 1: AI Problems that consume memory space in exponential order are prone to combinatorial exposition. Techniques that rely on storing all possibilities in memory, or even generating all possibilities, are out of the question except for the smallest of these problems. As an example, following table lists the total number of visited nodes by a set of AI Gaming Algorithms during computation, which are in the order of their computational memory requirements. Algorithms listed above are all exponential or factorial complexity in memory space requirements.

Table 4: Memory space requirements for some AI gaming algorithms.

<table>
<thead>
<tr>
<th>Game</th>
<th>Maxim. # of visited nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-Puzzle</td>
<td>9!</td>
</tr>
<tr>
<td>15-Puzzle</td>
<td>16!</td>
</tr>
<tr>
<td>n-city TSP</td>
<td>n!</td>
</tr>
<tr>
<td>Checkers</td>
<td>$10^{20}$</td>
</tr>
<tr>
<td>Chess</td>
<td>$10^{40}$</td>
</tr>
</tbody>
</table>

Example 2: Breadth-First-Search (BFS) is a frequently used algorithm in AI. We analyze the algorithm for time and space complexity. Each node using BFS may be visited in a constant time. The time complexity function $f_B$ is a function of the branching factor $b$, and the solution depth $d$ (a multi-variable function). In the the worst case, we will need to generate all the nodes at level $d$.

Now, $f_B(b, d) = 1 + b + b^2 + \ldots + b^d = \frac{(b^{d+1})-1}{b-1}$. Next we apply our framework for the fractional expressions. The highest order term in the numerator, $g_1(b, d) = b^{d+1}$. Similarly, the highest order term in the denominator, $g_2(b, d) = b$. Hence, $g(b, d) = \frac{b^{d+1}}{b} = b^d$ and $f_B(b, d) \in O(b^d)$. To generate the solution, we need to store all of the visited nodes. Hence, $S_B(b, d) = \frac{(b^{d+1})-1}{(b-1)}$, and $S_B(b, d) \in O(b^d)$. In the best possible case, we will find the solution at the first visited node. Hence, $f_B(b, d) \in \Omega(1)$, and also $S_B(b, d) \in \Omega(1)$. The space-time bandwidth product, $C_B(b, d) = (b^d)^2 = b^d2d$.

Suppose that we want to implement the algorithm on a typical machine with a speed of 100 MHz. If it visits a new state (node) in every 100 instructions, and if each instruction takes 1 machine cycle, then a total of $100 \times 10^6 = 10^6$ nodes will be visited in each second. Suppose, each node occupies 4 bytes of memory.

With a RAM size of 1 GB = $10^9$ byte, it's node keeping capacity is, $\frac{10^9}{250 \times 10^6} = 250$ seconds or 4 minutes and 10 seconds of visiting (generating) nodes, all memory space will be exhausted! This phenomenon is state-space explosion in AI.

11 Conclusion

Complexity analysis is significant in justifying whether the algorithm will take up prohibitive amount of computation time or computational memory space with considerable input size. This is particularly true with exponential algorithms.

In this paper, a generalized framework for the time and space complexity analysis of algorithms and functions has been proposed and it’s applications are discussed. Also, the significance of asymptotic complexity is established. Set theoretic properties relating to complexity are considered. It has been established that the notational complexities are sets of functions. AI algorithms are prone to both time and the space complexity analysis. For these algorithms, it is important to use the notation called Space-Time Bandwidth Product, $C$.

Performance is not everything. There can be a tradeoff between ease of understanding, writing and debugging a code corresponding to an algorithm with the efficient use of time and space. However, it is still useful in comparing the performance of different algorithms, even if the optimal algorithm may not be adopted. A problem may have more than one solution each one of which can be expressed as a different algorithm. Therefore, we need to compare among the performance of algorithms. After comparing different, possible algorithms for solving a problem, the user may select the one that best fits his computational needs.

In future, the relationship between the time and the space complexity order of algorithms and functions will be investigated. Also the proposed framework will be applied to critical AI problems for performance evaluation.

References: