ISSN 2278-3091
Volume 9, No.1.4, 2020
International Journal of Advanced Trends in Computer Science and Engineering
Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse6591.42020.pdf
https://doi.org/10.30534/ijatcse/2020/6591.42020

Calculation on Euler Arithmetic Complexity using Big O Notation

Nooraida Samsudin[1], Nurhafizah Moziyana Mohd Yusop[2], Anis Shahida Niza binti Mokhtar[2], Mohd Fahmy bin Amran[2], Iliana Mohd Ali[1]
[1] University College TATI, Malaysia, nooraida@tatiuc.edu.my, iliana@tatiuc.edu.my
[2]Universiti Pertahanan Nasional Malaysia, Malaysia, nmoziana@upnm.edu.my, anis@upnm.edu.my, fahmy@upnm.edu.my

ABSTRACT

Compute the complexity of algorithm is such way to describe it efficiency. Algorithm complexity computes the total time taken by an algorithm. It computes the time of the algorithm to run as the length of the input in the function. This paper describes the efficiency of algorithm complexity of Euler Arithmetic. The efficiency is determines by analyzing the run time of the algorithm using Big O. It only compared the complexity of the algorithm using notation of O(1) and O(n). Then, the result is analyzed using best case and worst case. As the result, the algorithm can be solving using both O(1) and O(n). However worst case gives better complexity for thisalgorithm even in small or higher step size compare to best scenario. This is because time taken on the size of the data N decreases as the value of N increases in O(n).

Key words :complexity, algorithm, analysis, Big O, efficiency

1. INTRODUCTION

Euler Arithmetic is an improvement of the Euler method. Euler Arithmetic improves the performance of the algorithm's complexity compared to the Euler method. Algorithms commonly used for solving problems or mechanism via computers. A*lgorithm efficiency* is a property of an algorithm which relates to the number of computational resources is being used. Most of the algorithms try to achieve the better performance globally [1]. Calculating the efficiency of the algorithm is including time complexity, space complexity, administrative cost and faster implementation [2]. In this study, the complexity of the algorithm is focus on time efficiency. Time efficeincy will take the time complexity of an algorithm to computes the amount of time taken by an algorithm to run as a function of the length of the input [3]. It being tested using Big O notation method to obtain the efficiency of the algorithm developed.

Big O notation is one of the effective methods for studying the time efficiency of algorithms [4]. This paper analyzed the complexity of the algorithm based on the Big O notations, O(1) and O(n). Calculations are performed by running an algorithm based on algorithms using both O(n) and O(1). Run time calculations for both algorithms using O(n) and O(1) are recorded. Then, the efficiency of the algorithm is analyzed based on the worst case and best case by measuring the step sizea*N* use in the testing.

2. LITERATURE REVIEW

Big O notation consists of scientific capacities and gradual analysis. It is important to represent algorithmic complexity of of each solution. There are two types of metrics to analyze the algorithm complexity. Complexity based on the algorithmic efficiency and the other one is the complexity on the structure of the algorithm [5].

Big O notation used to identify the function base on their growth rates [6]. It can operate the identical O notation for unique functions with the identical surge rate. Recently, algorithm analysis describes the usage of computational resources that regularly used. The length of input function in Big O notation usually refers to the three aspects. There are the worst cases, average case and best case. These scenarios help algorithm developers to predict the behavior of their algorithms [7]. They also can resolve which of multiple algorithms to use.

\Algorithm complexity evaluates the obtaining of the count of operations, performed by a given algorithm as a size of input data in the function [8]. To make it as simple as it can, complexity is a rough approximation of the number of steps necessary to execute an algorithm. This paper aim is to calculate the complexity of Euler Arithmetic using Big O and analyzed using worst case and best case scenario. This is to compare the better way to perform the algorithm of Euler Arithmetic in increasing the efficiency.

Calculation of the algorithm is referring to standard measure of algorithm efficiency for calculating algorithm complexity [9] as in Table 1. The best way to understand O in general is to generate a program code. Figure 1,2,3 and 4 shows the example program code to describe the varies of O notation [10].

Table 1: Standard Measure Of Algorithm Efficiency

Efficiency	Notation O
Logarithm	O(log n)
Linear	O(n)
Logarithm Linear	O(n(log n))
Quadratik	$O(n^2)$
Polynomial	$O(n^k)$
Exponen	$O(e^n)$
Factorial	O(n!)

2.1 O(1)

In O(1) notation, the algorithm normally will execute in the same time (or space) regardless of the size of the input data set. Figure 1 shows the example of the coding for O(1).

```
bool IsFirstElementNull(String[] strings)
{
        if(strings[0] == null)
        {
                return true;
        }
                return false;
}
```

Figure 1: Program code for O(1)

2.2 O(n)

The performance of the algorithm in O(n) notation is it will grow linearly and in direct proportion to the size of the input data set. The example below demonstrates the relation between worst-case scenario performances towards Big O.In this notation, during any iteration of the *(for)* loop the matching string in the function would return fast. However, Big O notation will always assume the upper limit is the maximum number of iteration during execution of the algorithm. Figure 2 shows the example of the program code for O(n).

```
bool ContainsValue(String[] strings, String value)
{
        for(int i = 0; i < strings.Length; i++)
                { if(strings[i] == value)
        {
                return true;
        }
                }
                return false;
                        }
```

Figure 2: Program code for O(n)

A. 2.3 $O(n^2)$

Performance of $O(n^2)$ notation in algorithm is directly proportional to the square of the size of the input data set. This algorithm typically involves nested iterations over the data set. Deeper nested iterations will result in $O(n^3)$, $O(n^4)$ and so on. Figure 3 shows the example of the program code for O (n^2).

```
bool ContainsDuplicates(IList<string> elements)
        {
                for (var outer = 0; outer < elements.Count; outer++)
                {
                        for (var inner = 0; inner < elements.Count; inner++)
                        {
        if (elements[outer] == elements[inner]) return true;
                }
                        }
                return false;
                }
```

Figure 3: Program code for O (n^2)

B. 2.4 $O(2^n)$

For the algorithm whose growth doubles it normally refer to $O(2^n)$ notation. The each addition is increasing depend on the input of the data set. O (2^n) function represent exponential in growth curve, which is starting off very shallow, then increasing drastically. Recursive calculation of Fibonacci numbers is an example of an O (2^n) function. Figure 4 shows the example of the program code for O (2^n).

```
int Fibonacci(int number)
        {
            if (number <= 1) return number;
                return Fibonacci(number - 2) + Fibonacci(number - 1);
        }
```

Figure 4: Program code for O (2^n)

The efficiency of the algorithm either it is good or bad is referring to the three possible cases complexity [11]. Worst-case complexity is the best efficiency because in any step size *N*, the function will be iterate by the maximum of the step size [12]. Thus, at each column the loop invades the maximum point. Best-case complexity is a straight forward analysis which the function does not have the iteration. Finally, the average-case complexity of the algorithm is the function describes the mean of the number of steps taken at any size of *N*.

3. RESEARCH METHOD

The research uses order notation primary to compare the algorithms efficiency. Analysis of the algorithm efficiency has been done using Big O. In this paper, the algorithm for the Euler Arithmetic using O(1) and O(n) is produce to compare the complexity. Figure 5 show the flowchart for the calculation. Calculation is done by running time based on each row in the algorithm [13]. The result is obtain by comparing the algorithm of Euler Arithmetic using O(n) and O(1).

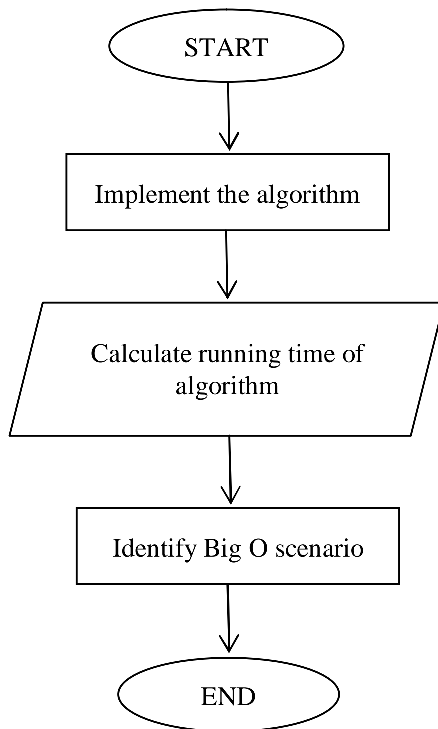

Figure 5: Flowchart to calculate Big O for the algorithm

Code for the algorithm in this paper is described in Figure 6 and 7 below. Figure 6 shows the code for thealgorithm using O(1). Meanwhile code for algorithm using O(n) is shows in Figure 7. This research follows the following steps [14] to complete the analysis of the running time of an algorithm:

i. Wholly implement the algorithm

ii. Figure out the time needed for each baseline operation

iii. Describe unnamed quantities which can be used to define the execution frequency of the basic operations

iv. Flourish a sensible model for the input to the program

v. Inspect the unnamed quantities and assumed as modelled input.

vi. Compute the total running time.

The efficiency of the algorithm is analyzed based on the worst case and best case. The best way to understand O in general is to generate a program code as shows in Figure 6 and 7 below. This study provides only two examples of program codes, namely O(1) and O(n) because only two of these notations are applicable to the generated algorithm.

1\. Start
2\. Set x_0, y_0, h, x_n
3\. Count n = $\frac{x_n - x_0}{h}$
4\. Record time processing
5\. Count $x_{n+1} = x_{n+}h$
6\. Equation
7\. for (j=0; j <= n; j++) {
7.1 Count, a(j)
7.2 Count y(j)
7.3 Calculate error, abs [a(j) – y(j)]
}
1\. Record maximum error
2\. End time processing
3\. Record time processing
4\. End

Figure 6: Algorithm develop using O(n)

1\. Start
2\. Set x_0, y_0, h, x_n
3\. Count n = $\frac{x_n - x_0}{h}$
4\. Record time processing
5\. Count $x_{n+1} = x_{n+}h$
6\. Equation
7\. if (j <= n) {
7.1 Count, a(j)
7.2 Count y(j)
7.3 Calculate error, abs [a(j) – y(j)]
}
8\. Record maximum error
9\. End time processing
10\. Record time processing
11\. End

Figure 7: Algorithm develop using O(1)

4. RESULT AND ANALYSIS

Calculation on running time for the algorithm using O(1) and O(n) is shown in Table 2 and Table 3 below. Analysis on total running time is being analyzed in this section to determine types of O notation. Total run time for O(n) is O(n) + 16 while total run time for O(1) is 17. According to the notation O, comparing the O(n) and O(1) shows that O(n) has the largest value. After calculating the run time, this research has the largest value which is O(n). It have been showed in Table 2 that the total running time is O(n) + 16 comparing to Table 3 the total running time gives value 17. It shows O(n) emphasizes the speed of time growth when algorithms are given different data inputs.

Table 2:. Calulation using O(n)

Line	Running Time (rt)
2	4
3	3
5	1
7	O(n)
7.1	1
7.2	1
7.3	1
8	1
9	1
10	3
Total Running Time	4+3+1+O(n) + 1+1+1+1+1+3=O(n) + 16

Table 3: Calulation using O(1)

Line	Running Time (rt)
2	4
3	3
5	1
7	1
7.1	1
7.2	1
7.3	1
8	1
9	1
10	3
Total Running Time	4+3+1+1+ 1+1+1+1+1+3= 17

The analysis performed the algorithm can be solved using big O notation, O(1) where the algorithm is run without the loop (*for*) as in line 7 of the algorithm in Figure 6. The use of O(1) is seen as a best case because the algorithm does not have to loop (*for*) and there is no value transmission in the sort. It give the shortest running time for any input of size *N*.The use of O(1) indicates that the run time is constant and does not depend on the problem of size *N*. In the worst case scenario, the use of O(n) is seen as significant to the algorithm as it determine the maximum amount of time that an algorithm requires to solve problems. This is based on the use of loop (*for*) like line 7 in the algorithm. The use of O(n) shows that the run time increases linearly with increasing data size. Based on the comparison of these worst-case and best-case scenarios, the algorithm can be solved using both O(n) and O(1) notations. However, the use of O(n) is more efficient in this study.

To prove that O(n) is efficient in this study, Table 4 shows how the run time for an algorithm differs for different types of complexity over the number of *N* data sizes [15]. Types of complexity is reflect to amount of data [16]. For this study, the data size of *N* is 10,100 and 1000. The size of this data *N* refers to the step size used during this study which is 0.1, 0.01 and 0.001. For different types of complexity, only O(1) and O(n) are compared. The efficiency analysis refers to the time taken by solving the algorithm [17]. Refer to table below, for O(1) the time taken is constant across all data sizes. However, when viewed in O(n), the time taken on the size of the data *N* decreases as the value of *N* increases.

Table 4: Running Time for Types of Complexity Towards Amount of Data *N*

Complexity	N = 10	N = 100	N = 1,000
O(1)	$1x10^{-7}$ seconds	$1x10^{-7}$ seconds	$1x10^{-7}$ seconds
$O(\log_2 N)$	$3.3x10^{-7}$ seconds	$6.6x10^{-7}$ seconds	$10x10^{-7}$ seconds
O(N)	$1x10^{-7}$seconds	$1x10^{-6}$ seconds	$1x10^{-5}$ seconds
$O(N\log_2 N)$	$3.3x10^{-7}$ seconds	$6.6x10^{-6}$ seconds	$10x10^{-5}$ seconds
$O(N^2)$	$1x10^{-6}$ seconds	$1x10^{-4}$ seconds	$1x10^{-2}$ seconds
$O(N^3)$	$1x10^{-5}$ seconds	$1x10^{-2}$ seconds	10 seconds
$O(2^N)$	$1x10^{-5}$ seconds	$4x10^{21}$ seconds	∞

5. CONCLUSION

This paper presents the complexity time of Euler Arithmetic algorithm uses Big O notations. Euler Arithmetic algorithm manage to use both O(1) and O(n) in analyze the efficiency. As a result, the algorithm can be solved using both O(1) and O(n). Therefore, worst case gives better complexity for this algorithm in this research. This is because the function specified by the ultimate number of size *N* steps taken. Total running time of the algorithm using O(n)decreases as the value of *N* increases. Solving this algorithm using O(1) contribute to the best case. This is because it gives the shortest running time for any input of size *N*. O(1) demonstrate that the time taken is constant across all data sizes. As a conclusion, O(n) is most suitable to solve Euler Arithmetic complexity time rather than O(1).

ACKNOWLEDGEMENT

The authors gratefully acknowledge the Faculty of Science and Technology Defense, Universiti Pertahanan Nasional Malaysia. The author Nooraida Samsudin also would like to thank TATI University College for the Short Term Grant (STG 1/2018).

REFERENCES

1. D. Dhanalakshmi and A.S. Vijendran. **Adaptive Data Structure Based Oversampling Algorithm for Ordinal Classification**, *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 12, pp. 1063-1070, 2018.

2. I. Chivers and J.Sleightholme. **An Introduction to Algorithms and the Big O Notation. In: Introduction to Programming with Fortran**,*Springer*, Cham, 2015.
3. C.P. Milliken.**Advanced Topics II: Complexity. In: Python Projects for Beginners**, *Apress*, 2020.
4. D.S. Ruhela. **Comparative Study of Complexity of Algorithms For Ordinary Differential Equations** I, *International Journal of Advanced Research in Computer Science & Technology,* vol. 2, pp. 329-334, 2014.
5. N. Singh and R.G. Tiwari.**Basics of Algorithm Selection: A Review**,*International Journal of Computer Science Trends and Technology*, vol. 3, pp. 139-142, 2015.
6. J. Burnim, S. Juvekar and K. Sen.**WISE: Automated test generation for worst-case complexity**, *International Conference on Software Engineering*, pp. 463-473, 2009.
7. Y. Han and M. Thorup.**Integer Sorting in O(n √(log log n) Time and Linear Space**,*Proceedings of the 43rd Symposium on Foundations of Computer Science*, pp. 135-144, 2002.
8. S.O. Kuznetsov and S. A. Obiedkov. **Comparing performance of algorithms for generating concept lattices**, *Journal of Experimental & Theoretical Artificial Intelligence,* vol. 14*,* pp.189-216, 2010. https://doi.org/10.5325/gestaltreview.14.2.0189
9. S. Bae.**Big-O Notation. In: JavaScript Data Structures and Algorithms**,*Apress*, 2019.
10. S. Gayathri Devi, K. Selvam and S. P. Rajagopalan.**An Abstract to Calculate Big O Factors of Time and Space Complexity of Machine Code**, *International Conference on Sustainable Energy and Intelligent System,* pp. 20-22, 2011.
11. R.Sedgewick and P. Flajolet.**An Introduction to the Analysis of Algorithms, 2nd edition**, *Amazon*, 2015.
12. L. Plaskotaa, G. W.Wasilkowskib and H. Woźniakowskiac.**A New Algorithm and Worst Case Complexity for Feynman–Kac Path Integration**,*Journal of Computational Physics,* vol. 164, pp. 335-353, 2000.
13. N.Bahiah et al.**Struktur Data & Algoritma menggunakan C++**,*Universiti Teknologi Malaysia*, 2005.
14. S.K. Gill, V.P. Singh, P. Sharma, D. Kumar.**A Comparative Study of Various Sorting Algorithms**,*International Journal of Advanced Studies of Scientific Research,* vol. 2, 2019.
15. P. Danziger, *Big O Notation,* 2009. Retrieved from http://www.scs.ryerson.ca/~mth110/Handouts/PD/bigO.pdf.
16. M.Hardick, **TCP with Machine Learning – Advanced and Opportunities**, *International Journal of Advanced Trends in Computer Science and Engineering,*vol.8, pp.3526-3534, 2019. https://doi.org/10.30534/ijatcse/2019/132862019
17. L.N Yasnitsky, **Algorithm for Searching and Analyzing Abnormal Observations of Statistical Information Based on The Arnold – Kolmogorov – Hecht-Nielsen Theorem,***International Journal of Advanced Trends in Computer Science and Engineering,* vol. 9, pp. 1814-1819, 2020. https://doi.org/10.30534/ijatcse/2020/139922020