Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 15, SEPTEMBER 2023

DOI: 10.37943/15DNLB5877

Zhanar Bimurat
Ph.D. in Information systems, Associate Professor, Department of Cybersecurity
zh.bimurat@aues.kz orcid.org/0000-0001-8283-898X
Energo University, Kazakhstan

Yekaterina Kim
Ph.D. in Engineering Science, Associate Professor, Department of Information Technology
e.kim@turan-edu.kz orcid.org/0000-0001-7441-524X
Turan University, Kazakhstan

Rauza Ismailova
Ph.D. in Engineering Science, Associate Professor, Department of Information Technology
r.ismailova@turan-edu.kz orcid.org/0000-0002-8488-0855
Turan University, Kazakhstan

Bimurat Sagindykov
Ph.D. in Physics and Mathematics, Associate Professor, Department of Higher Mathematics and Modeling
b.sagindykov@satbayev.university orcid.org/0000-0002-5349-1961
Satbayev University, Kazakhstan

METHODS OF NAVIGATING ALGORITHMIC COMPLEXITY: BIG-OH AND SMALL-OH NOTATIONS

Abstract: This article provides an in-depth exploration of Big-Oh and small-oh notations, shedding light on their practical implications in the analysis of algorithm complexity. Big-Oh notation offers a valuable tool for estimating an upper bound on the growth rate of an algorithm's running time, whereas small-oh notation delineates a lower limit on this growth rate. The piece delves into a comprehensive examination of various complexity classes that emerge through the application of Big-Oh notation, underscoring the significance of small-oh notation as it complements and enriches complexity analysis.

In the realm of programming and computer science, the employment of these notations holds paramount importance. They empower developers and researchers to make informed decisions regarding algorithm selection and optimization. It is crucial to recognize that while complexity analysis is a vital facet of effective programming, ongoing research endeavors may yield more refined methodologies and approaches within this domain.

By understanding and harnessing the power of Big-Oh and small-oh notations, professionals can effectively evaluate algorithm efficiency and scalability. This knowledge equips them with the ability to design and implement algorithms that meet specific performance criteria, which is pivotal in the ever-evolving landscape of technology and computation. As pushing the boundaries of what is possible in the field of algorithm design is being continued, these notations remain invaluable tools for navigating the complex terrain of algorithmic analysis and optimization.

By embracing Big-Oh and small-oh notations, professionals can finely assess algorithmic efficiency, ensuring they meet performance criteria in the evolving technological landscape. These notations remain indispensable for algorithmic analysis.

Received: 04.09.2023 Accepted: 23.09.2023 Published: 30.09.2023

DOI: 10.37943/15DNLB5877

Keywords: Big-Oh notation, small-oh notation, algorithmic analysis, asymptotic analysis

Introduction

The effectiveness of a program is primarily determined by the resources it consumes, namely time and memory. In particular, the time complexity of a program serves as a metric to evaluate the duration of its execution, typically quantified by the number of elementary operations required for its solution. The efficiency of a program executed on a computer is inherently contingent upon the algorithm it employs. Additionally, it can be mathematically established, based on plausible assumptions regarding coding conventions, programming languages, and realistic computer models, that the program's efficiency is independent of the chosen language, coding methodology, or computer performance. Instead, it is an intrinsic property of the algorithm itself. Thus, assessing the efficiency of a program effectively reduces to evaluating the efficiency of its underlying algorithm. One key parameter used to gauge an algorithm's efficiency is its running time.

Given the intricacy involved in determining the precise running time of an algorithm, it is common practice to rely on an approximate estimation. To achieve this, the technique of asymptotic analysis is frequently employed [1].

Asymptotic analysis is a systematic approach utilized in the assessment of algorithmic effectiveness, providing insights into their behavior as the input size grows. This method is rooted in the principle that for sufficiently large input sizes, the prominent attributes of an algorithm, including running time and memory consumption, exhibit discernible patterns in relation to the input size. By estimating the growth rate of these attributes as the input size approaches infinity, it becomes possible to compare different algorithms and draw conclusions regarding their efficiency in worst-case or average-case scenarios [2].

Asymptotic analysis of algorithms employs asymptotic notations, namely "O" (Big-Oh), "Ω" (Big-Omega), and "Θ" (Big-Theta), as fundamental tools to establish upper, lower, and tight bounds, respectively, on the time or space complexity of an algorithm [3, 4]. These notations provide concise representations of how the algorithm's resource requirements scale with input size and facilitate a clear understanding of its efficiency characteristics. By utilizing these asymptotic notations, analysts can compare algorithms, make informed decisions about algorithm selection, and gain insights into the best- and worst-case scenarios for algorithm performance [5, 6].

Asymptotic analysis of algorithms empowers engineers and software developers to make judicious choices regarding algorithm selection by considering factors such as efficiency requirements and resource utilization. This analytical approach further enables the prediction of an algorithm's scalability with respect to increasing data sizes, a crucial consideration when dealing with substantial datasets. By leveraging asymptotic analysis, professionals in these domains can optimize algorithmic solutions, ensuring optimal performance and resource allocation in scenarios where speed and efficiency are paramount.

The concepts of Big-Oh and small-oh (or little-oh) constitute pivotal components within the realm of asymptotic notation. These concepts hold substantial significance in the analysis of algorithms and data structures, serving as fundamental instruments in assessing the temporal intricacy of algorithms. This article focuses on comprehensively exploring these concepts and their practical application in analyzing algorithmic complexity.

Methods and Materials

Big-Oh. The symbol "O" commonly referred to as Big-Oh notation, assumes the role of denoting the upper bound on the growth rate of an algorithm's running time with respect to the size of the input data. In essence, Big-Oh elucidates the rate at which the algorithm's

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 15, SEPTEMBER 2023

performance escalates in the worst-case, where the input data size tends towards infinity. This notation provides a concise and meaningful representation of the algorithm's scalability characteristics, facilitating a comprehensive understanding of its temporal efficiency [8, 9].

Definition 1. A function $f(n)$ is said to belong to the class $O(g(n))$, denoted as $f(n) \in O(g(n))$, if $f(n)$ is bounded above by a constant multiple of $g(n)$ for sufficiently large values if n. In other words, there exist a positive constant c and a non-negative integer n_0 such that $f(n) \leq cg(n)$ holds for all $n \geq n_0$.

For instance, if the running time of an algorithm $f(n)$ is constrained by the function $g(n)$, we can express this as $g(n)$, or $f(n) \in O(g(n))$ (Fig. 1). This implies that in the worst-case, the algorithm will not execute more quickly than the growth rate of $g(n)$.

When we express $f(n) = O(g(n))$, it signifies that $g(n)$ serves as an upper bound on $f(n)$, specifically an asymptotic upper bound, without considering constant factors. In other words, $g(n)$ provides a mathematical representation that encapsulates the growth rate of $f(n)$ while disregarding the influence of multiplicative constants.

The primary criterion for assessing algorithmic performance lies in quantifying the number of elementary operations involved in processing input data of a given size. The terms "basic operations" and "size" entail some degree of ambiguity and are contingent upon the specific algorithm under analysis. The determination of size frequently hinges upon the volume of input being processed. For instance, when comparing sorting algorithms, the size of the problem is commonly measured by the number of records necessitating sorting. It is crucial that the basic operation exhibits the attribute of execution time independence from the specific operand values. Typical examples of basic operations in most programming languages encompass arithmetic operations or comparisons involving integer variables. However, operations such as summing the contents of an array consisting of n integers do not qualify as basic operations, as their cost is influenced by the value of n (i.e., the size of the input data).

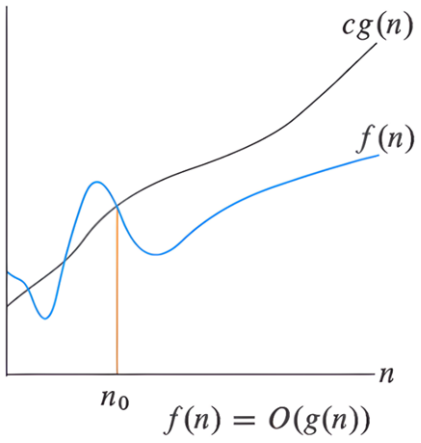

Figure. 1. Big-Oh notation gives an upper bound for a function to within a constant factor.

We write $f(n) = O(g(n))$ if there are positive constants n_0 n_0 and c such that at and to the right of n_0, the $f(n)$ value of always lies on or below $cg(n)$ [7]

Let's examine the insertion sort algorithm and its associated pseudocode and block-scheme (see Fig. 2 and Fig. 3).

DOI: 10.37943/15DNLB5877


```
InsertionSort(A)
   for i = 1 to length(A) - 1 do
      current_element = A[i]
      position = i

      while position > 0 and A[position - 1] > current_element do
         A[position] = A[position - 1]
         position = position - 1
      end while

      A[position] = current_element
   end for
End of the Algorithm
```

Figure 2. Insertion sort algorithm

The efficiency of the algorithm is contingent upon the characteristics of the input data, particularly when dealing with larger datasets, as it requires more time to complete the sorting process. The runtime of the algorithm, for inputs of the same size, can be determined by the enumeration of basic operations or steps that need to be executed. These basic operations serve as fundamental units for measuring the algorithm's computational complexity and provide insights into its performance characteristics.

Assuming that each line of pseudocode requires a constant amount of time to execute, denoted as c_k, where c_k represents a fixed constant time cost associated with the k^{th} line, we can observe that the actual execution time may vary for individual lines. Nevertheless, for practical purposes and to align with the implementation on most real computers (as depicted in Table 1), it is reasonable to consider these constant time costs.

The running time of an algorithm can be computed by summing up the execution time of each individual statement. If statement requires c_k steps and is executed m times, it contributes $c_k \times m$ to the overall running time. This relationship allows for the estimation and analysis of the total running time of an algorithm based on the execution counts and the time complexity of its constituent statements [7].

Table 1. Analysis of the insertion sort algorithm

Pseudocode	Cost	Times
for i = 1 to length(A) – 1 do	c_1	$n-1$
current_element = A[i]	c_2	$n-1$
position = i	c_3	$n-1$
while position > 0 and A[position – 1] > current_element do	c_4	$\sum_{i=1}^{n-1} t_i$
A[position] = A[position – 1]	c_5	$\sum_{i=1}^{n-1} (t_i - 1)$
position = position - 1	c_6	$\sum_{i=1}^{n-1} (t_i - 1)$
A[position] = current_element	c_7	$n-1$

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 15, SEPTEMBER 2023

In order to compute the running time, $f(n)$, of the insertion sort algorithm when applied to an input containing n values, we evaluate the sum of the products derived from the "Cost" and "Times" columns of Table 1. Thus, the expression for $f(n)$ can be expressed as:

$$f(n)=c_1(n-1)+c_2(n-1)+c_3(n-1)+c_4\sum_{i=1}^{n-1}t_i+c_5\sum_{i=1}^{n-1}(t_i-1)+c_6\sum_{i=1}^{n-1}(t_i-1)+c_7(n-1). \quad (1)$$

Here, the variable t_i represents a variable number of repetitions based on the iterations executed within the inner loop. By substituting the appropriate values into this equation, we can obtain a quantitative measure of the running time for the given insertion sort algorithm.

The number of iterations executed in the inner loop of the insertion sort algorithm varies based on the initial order of the elements if the array. Generally, three cases are considered: worst, average, and best. In our scenario, the best-case occurs when the array is already sorted. In this case, each iteration (t_i) within the inner loop is equal to 1 for all i. Consequently, the running time of the algorithm can be expressed as:

$$\begin{aligned} f(n)&=c_1(n-1)+c_2(n-1)+c_3(n-1)+c_4(n-1)+c_5(n-2)+c_6(n-2)+c_7(n-1)= \\ &=(c_1+c_2+c_3+c_4+c_5+c_6+c_7)n-(c_1+c_2+c_3+c_4+2c_5+2c_6+c_7). \end{aligned} \quad (2)$$

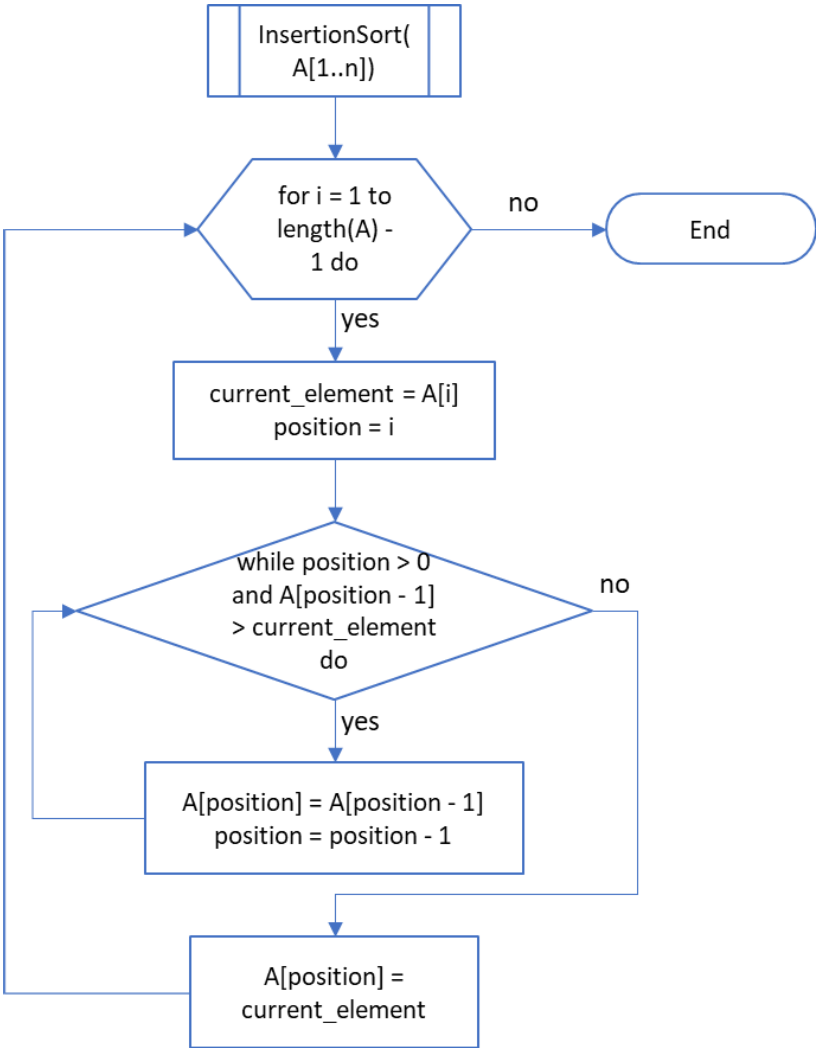

Figure 3. Insertion sort flowchart

DOI: 10.37943/15DNLB5877

The worst-case arises when the array is sorted in reverse order. In this instance, the insertion sort procedure necessitates comparing each element $A[i]$ of the array with every element in the fully sorted subarray $A[1: i-1]$. Consequently, for $i=2, 3, ..., n$, the number of repetitions t_i can be expressed as $t_i = i$. Based on this information, we can calculate the sums of repetitions as follows:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2},$$

$$\sum_{i=1}^{n-1} (i-1) = \frac{(n-1)(n-2)}{2}.$$

These equations provide the summation formulas for the sum of integers from 1 to $(n-1)$ and the sum of $(i-1)$ from 1 to $(n-1)$ respectively.

Substituting the obtained expressions into equation (1), we obtain:

$$f(n) = c_1(n-1) + c_2(n-1) + c_3(n-1) + c_4\frac{(n-1)n}{2} + c_5\frac{(n-1)(n-2)}{2} + c_6\frac{(n-1)(n-2)}{2} + \\ + c_7(n-1) = \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 - \frac{1}{2}c_4 - \frac{3}{2}c_5 - \frac{3}{2}c_6 + c_7\right)n - \\ - \left(c_1 + c_2 + c_3 - \frac{3}{2}c_5 - \frac{3}{2}c_6 + c_7\right). \quad (3)$$

To conduct an average case analysis, it is imperative to compute the average number of comparisons needed to ascertain the correct position for each newly added element. Even if the element is already in its proper place, at least one comparison is still necessary. When a new element is added to the array, it can occupy any of the $(i+1)$ positions, where i denotes the number of previously inserted elements. Assuming a random input, each new element possesses an equal probability of being positioned in any of the available locations.

In order to compute the average number of comparisons required for inserting the i-th element into an array of length n, assuming a random distribution of elements and equiprobability of their positions, the expectation method can be applied.

Mathematically, let a_i represent the number of comparisons needed to insert the i-th element. The average number of comparisons, denoted as $E(a_i)$, can be computed using the expectation formula:

$$E(a_i) = \sum ap(a_i = a),$$

where a represents the number of comparisons and $p(a_i = a)$ denotes the probability that a_i takes the value a.

In this case, assuming a random element distribution and equiprobability of their positions, each new element has an equal chance of being inserted at any of the $(i+1)$ available positions. Hence, the probability of a_i taking the value a is given by $p(a_i = a) = \frac{1}{i+1}$.

By substituting this probability distribution into the expectation formula, we can calculate the average number of comparisons required to insert the i-th element into an array of length n.

Then the probability of the i-th element being inserted as position j, where $1 \le j \le i-1$, is determined to be $\frac{1}{i-1}$ since there are $(i-1)$ possible positions for the i-th element. Consequently, the average number of comparisons required for inserting the i-th element can be computed by summing the products of the number of comparisons at each position (j) and

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 15, SEPTEMBER 2023

their corresponding probabilities $\left(\frac{1}{i-1}\right)$. Specifically, for the *i*-th element, this average number of comparisons is expressed as:

$$t_i = \sum_{j=1}^{i-1} \frac{1}{i-1} \cdot j = \frac{1}{i-1} \sum_{j=1}^{i-1} j = \frac{1}{i-1} \frac{(i-1)i}{2} = \frac{i}{2}.$$

Hence

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{(n-1)n}{4},$$

$$\sum_{i=1}^{n-1} \left(\frac{i}{2} - 1\right) = \frac{(n-1)n}{4} - (n-1) = \frac{n^2 - 5n}{4} + 1.$$

By substituting the obtained expression into equation (1), we obtain:

$$f(n) = c_1(n-1) + c_2(n-1) + c_3(n-1) + c_4 \frac{(n-1)n}{4} + c_5\left(\frac{n^2-5n}{4} + 1\right) + c_6\left(\frac{n^2-5n}{4} + 1\right) +$$

$$+ c_7(n-1) = \left(\frac{c_4}{4} + \frac{c_5}{4} + \frac{c_6}{4}\right)n^2 + \left(c_1 + c_2 + c_3 - \frac{1}{4}c_4 - \frac{5}{4}c_5 - \frac{5}{4}c_6 + c_7\right)n - \quad (4)$$

$$-(c_1 + c_2 + c_3 - c_5 - c_6 + c_7).$$

Prior to defining the function $g(n)$ as an upper bound for the function $f(n)$, it is necessary to employ simplification rules outlined by Shaffer in "Data structures and algorithm analysis":

1. If $f(n)$ belongs to $O(g(n))$, and $g(n)$ belongs to $O(h(n))$, then $f(n)$ belongs to $O(h(n))$.
2. If $f(n)$ belongs to $O(kg(n))$ for any constant $k > 0$, then $f(n)$ belongs to $O(g(n))$
3. If $f_1(n)$ belongs to $O(g_1(n))$ and $f_2(n)$ belongs $O(g_2(n))$, then $f_1(n) + f_2(n)$ belongs to $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ belongs to $O(g_1(n))$ and $f_2(n)$ belongs to $O(g_2(n))$, then $f_1(n) \cdot f_2(n)$ belongs to $O(f_1(n) \cdot f_2(n))$.

The first rule asserts that if function $g(n)$ serves as an upper bound for the cost function, which calculates the algorithm's running time, any upper bound on $g(n)$ will also be an upper bound on the cost function. This rule establishes the transitive nature of upper bounds.

The second rule holds significant value as it allows for the omission of multiplicative constants when utilizing Big-Oh notation in equations. This simplification permits a focus on the essential growth rate behavior of the algorithm, disregarding constant factors.

Rule (3) emphasizes that when executing sequential parts of a program, be it statements or code blocks, only the more computationally expensive part necessitates consideration in determining the overall complexity. This rule allows for an efficient analysis by focusing on the dominant factors impacting the program's efficiency.

The fourth rule finds application in the analysis of repetitive cycles within programs. When an action is iteratively performed a fixed number of times, with each iteration incurring the same cost, the total cost of the action is determined by multiplying the cost of a single iteration by the number of repetitions. This rule enables a straightforward analysis of repetitive operations within a program.

By combining the first three rules, it becomes possible to disregard all constants and lower-order terms when determining the asymptotic growth rate of any cost function. This approach is justifiable in asymptotic analysis, as the higher-order terms eventually surpass the

DOI: 10.37943/15DNLB5877

lower-order terms in their contribution to the total cost as *n* increases. Hence, for the best-case (2) of the insertion sort algorithm, the asymptotic upper bound for the cost function can be expressed as:

$$f(n)=(c_1+c_2+c_3+c_4+c_5+c_6+c_7)n-(c_1+c_2+c_3+c_4+2c_5+2c_6+c_7)=O(n),$$

Indicating that $g(n)=n$, and consequently, the function $f(n)$ belongs to $O(n)$.

The asymptotic upper bound for the cost function of the insertion sort algorithm in the worst-case (3) is given by:

$$f(n)=\left(\frac{c_4}{2}+\frac{c_5}{2}+\frac{c_6}{2}\right)n^2+\left(c_1+c_2+c_3-\frac{1}{2}c_4-\frac{3}{2}c_5-\frac{3}{2}c_6+c_7\right)n-\left(c_1+c_2+c_3-\frac{3}{2}c_5-\frac{3}{2}c_6+c_7\right)=O(n^2).$$

Similarly, the asymptotic upper bound for the cost function of the insertion sort algorithm in the average-case (4) is given by:

$$f(n)=\left(\frac{c_4}{4}+\frac{c_5}{4}+\frac{c_6}{4}\right)n^2+\left(c_1+c_2+c_3-\frac{1}{4}c_4-\frac{5}{4}c_5-\frac{5}{4}c_6+c_7\right)n-(c_1+c_2+c_3-c_5-c_6+c_7)=O(n^2).$$

Typically, the primary focus in algorithm analysis is to determine the worst-case execution time, which refers to the longest possible time an algorithm takes to complete for input data of any size. This emphasis on worst-case analysis is driven by several compelling reasons:

1. Upper bound guarantee: the worst-case execution time provides an upper bound on the execution time for input data of any size. By knowing this worst-case time complexity, one can ensure that the algorithm will never exceed this upper limit. It eliminates the need to rely on assumptions or hope for the best-case scenario. This feature is particularly critical in real-time computing applications, where operations must be completed within specific time constraints.

2. Frequent occurrence: for certain algorithms, the worst-case scenario arises quite frequently in practice. For instance, in information retrieval from a database, the worst case of the search algorithm often occurs when the desired information is not present. In various applications, the need to search for missing information can occur frequently, making worst-case analysis relevant.

3. Similarity to average case: in many cases, the average-case time complexity of an algorithm is comparable to the worst-case scenario. This implies that the algorithm's performance is unlikely to significantly improve on average compared to the worst case. Hence, by analyzing the worst-case behavior, one gains insights into the algorithm's average-case efficiency.

Overall, the focus on worst-case analysis provides a solid foundation for understanding an algorithm's performance and ensures robustness, particularly in real-time and critical applications.

Efficiency analysis encompasses a broad range of functions that exhibit different growth orders (Fig. 4) but are considered equivalent within a constant factor. Despite this vast array of functions, it is noteworthy that the time complexities of numerous algorithms can be classified into a limited number of efficiency classes. Table 2 provides a comprehensive enumeration of these classes, arranged in ascending order according to their growth orders. The existence of a small set of efficiency classes to describe the time characteristics of diverse algorithms is indeed remarkable, considering the infinite possibilities within the realm of efficiency analysis.

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 15, SEPTEMBER 2023

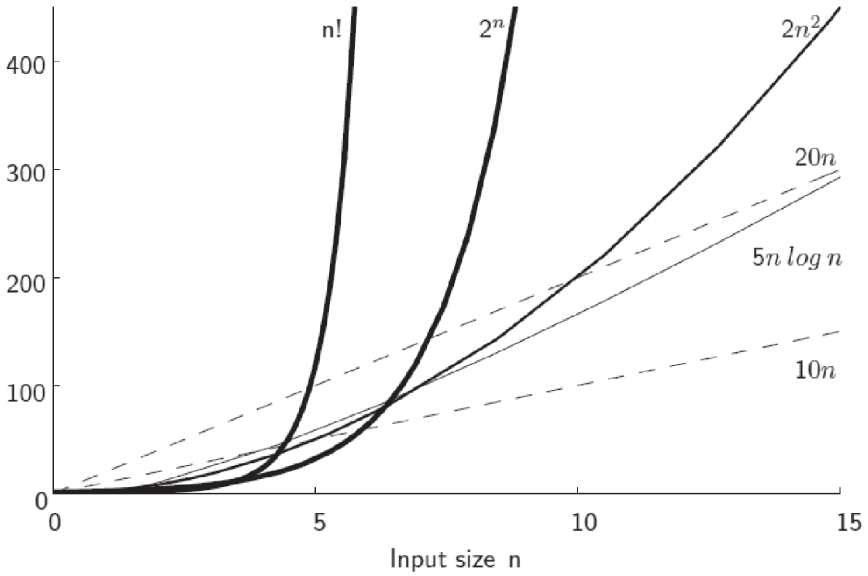

Figure 4. The growth rate for six equations. This figure was taken from Shaffer's "Data structures and algorithm analysis"

Table 2. Basic asymptotic efficiency classes

Class	Name	Comment
1	Constant	Outside of optimal conditions, there exist minimal practical instances where efficiency maintains a steady state. Generally, an algorithm's processing time edges toward infinity with the continual growth of input data size.
$\log n$	Logarithmic	Algorithms within this category exhibit growth velocities that are less rapid than any given power function related to n. This is frequently due to the consistent reduction in problem dimensions with each cycle of the algorithm. It's important to recognize that a logarithmic algorithm doesn't process every piece of input data, nor a stable fraction thereof – accomplishing such a comprehensive review would necessitate a minimum of a linear runtime.
n	Linear	Algorithms that go through a list of n elements, such as linear search, fall into this category.
$n \log n$	Linearithmic	Numerous divide-and-conquer strategies, such as merge sort and, in the average case, quicksort, are classified within this group.
n^2	Quadratic	This classification is commonly associated with algorithms that employ two nested loops. Basic sorting methods and specific matrix operations serve as standard illustrations of this category.
n^3	Cubic	This description typically applies to algorithms that utilize three nested loops. Various complex linear algebra algorithms are prominent examples of this category.
n^a	Polynomial	An algorithm is polynomial if $g(n)$ grows no faster than some polynomial (generally a power function) of n.
$n!$	Factorial	This is characteristic of algorithms tasked with generating all possible permutations of a set consisting of n elements.

The complexity class $O(1)$ denotes constant complexity of an algorithm, signifying that the execution time remains unaffected by the input data size.

DOI: 10.37943/15DNLB5877

Consider the algorithm designed to retrieve an element from an array based on its index (see Fig. 5 and Fig. 6). Presented below is the pseudocode outlining the procedure for obtaining an element from an array by index:

```
Algorithm GetElement(array, index):
    if index >= 0 and index < length(array) then
        return array[index]
    else
        print "Error: Invalid index access"
        // or return a special flag indicating an index access error
    end if
End of the Algorithm
```

Figure 5. Algorithm of the procedure for retrieving an element from an array by index

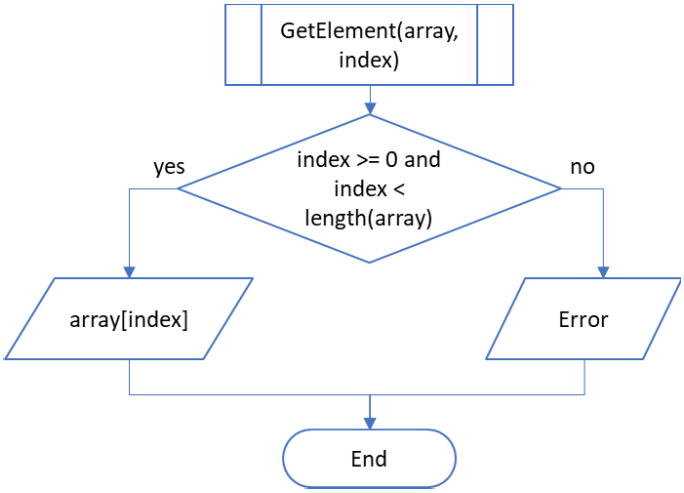

Figure 6. Flowchart of the procedure for retrieving an element from an array by index

Table 3. Analysis of the algorithm for retrieving an element from an array by index

Pseudocode	Cost	Times
if index >= 0 and index < length(array) then	c_1	1
return array[index]	c_2	1
Else	0	1
print "Error: Invalid index access"	c_3	1
end if	0	1

To analyze the complexity of the algorithm for retrieving an element from an array by index, we consider the execution time by summing the products of the values from the “Cost” and “Times” columns in Table 3. The resulting expression is

$$f(n) = c_1 + c_2 + c_3$$

As c_1, c_2 and c_3 are constants, it follows that $f(n) = O(1)$, indicating a constant-time complexity.

The bubble sort algorithm typically exhibits quadratic complexity. However, a specific modification of the algorithm, such as optimization using a flag, can be employed to achieve linear complexity. Here, we present an optimized bubble sort algorithm in pseudocode (see Fig. 7 and Fig. 8):

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 15, SEPTEMBER 2023

```
Algorithm BubbleSortLinear(array):
    length = length(array)
    swapped = true
    while swapped:
        swapped = false
        lastSwapIndex = length - 1
        for i from 0 to lastSwapIndex:
            if array[i] > array[i+1] then
                swap(array[i], array[i+1])
                swapped = true
                lastSwapIndex = i
            end if
        end for
        length = lastSwapIndex + 1
    end while
End of Algorithm
```

Figure 7. Algorithm of the Bubble Sort procedure

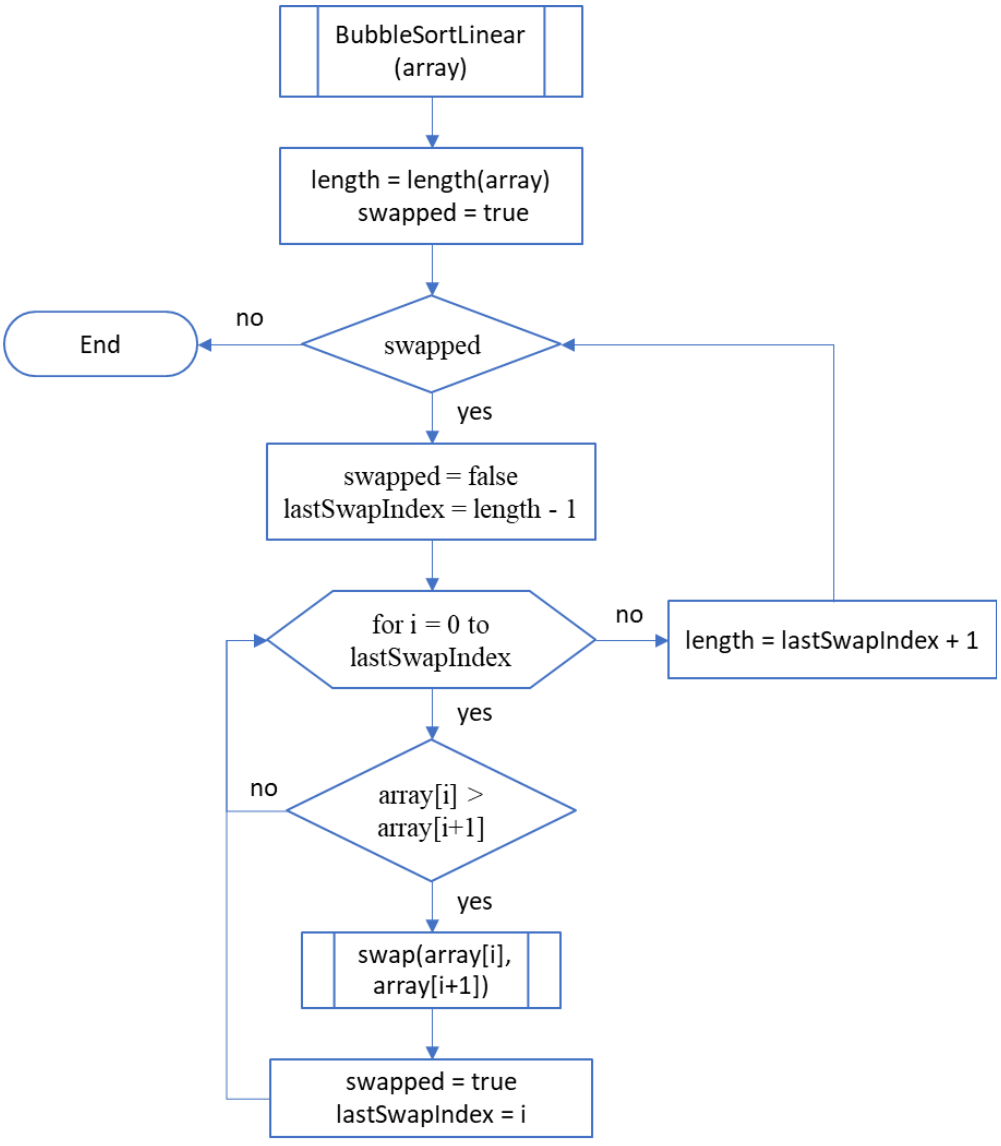

Figure 8. Flowchart of the Bubble Sort procedure

DOI: 10.37943/15DNLB5877

The bubble sort is a sorting algorithm that operates by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. This process is repeated until no more swaps are needed, indicating that the list is sorted. Let's break down the algorithm (Fig. 7 and Fig. 8) in more detail:

1. *Initialization*:
- the algorithm starts by getting the length of the input array, which is the number of elements that need to be sorted;
- it also initializes a Boolean variable called **swapped** to True. This variable will be used to check if any swaps are made during a pass through the array.

2. *Main Loop*:
- the algorithm enters a while-loop that continues as long as **swapped** is True. This loop will continue until no more swaps are needed, which means the array is sorted.

3. *Pass through the array*:
- inside the while-loop, the algorithm sets **swapped** to False at the beginning of each pass through the array. This is done to initially assume that no swaps will be needed;
- the algorithm also initializes **lastSwapIndex** to the last index of the array.

4. *Comparing and swapping*:
- the algorithm enters a for-loop that iterates through the array from the first element to **lastSwapIndex**.
- for each pair of adjacent elements (indexed as $\boldsymbol{i}$ and $\boldsymbol{i} + \mathbf{1}$), it compares them. If the element at index $\boldsymbol{i}$ is greater than the element at index $\boldsymbol{i} + \mathbf{1}$, it swaps them. This is the key step of the Bubble Sort algorithm, as it pushes the larger elements toward the end of the array.
- after each successful swap, the **swapped** flag is set to True, indicating that a swap has occurred, and **lastSwapIndex** is updated to the index of the last swap.

5. *Updating length*:
- after completing a pass through the array, the algorithm updates the **length** variable to **lastSwapIndex + 1**. This is because the largest element has moved to its correct position at the end of the array after each pass, so there is no need to consider it in the next pass.

6. *Repeating the loop*:
- the while-loop continues as long as **swapped** remains True, meaning that swaps are still occurring in the array. If **swapped** becomes False after a pass, it means that the array is sorted, and the loop exits.

7. *End of algorithm*:
- once the while-loop terminates, the algorithm has completed sorting the array, and it ends.

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 15, SEPTEMBER 2023

Table 4. Analysis of the Bubble Sort algorithm for the worst-case scenario

Pseudocode	Cost	Times
length = length(array)	c_1	1
swapped = true	c_2	1
while swapped:	c_3	min (n, $n-1$)
swapped = false	c_4	min (n, $n-1$)
lastSwapIndex = length – 1	c_5	min (n, $n-1$)
for i from 0 to lastSwapIndex:	c_6	min (n, $n-1$, $n-2$, ...)
if array[i] > array[i+1] then	c_7	min (n, $n-1$, $n-2$, ...)
swap(array[i], array[i+1])	c_8	min (n, $n-1$, $n-2$, ...)
swapped = true	c_9	min (n, $n-1$, $n-2$, ...)
lastSwapIndex = i	c_{10}	min (n, $n-1$, $n-2$, ...)
end if	0	min (n, $n-1$, $n-2$, ...)
end for	0	min (n, $n-1$)
length = lastSwapIndex + 1	c_{11}	min (n, $n-1$)
end while	0	1

To calculate the complexity of the algorithm, the running time of the Bubble Sort algorithm, we sum the products of the values from the "Cost" and "Times" columns of Table 4, resulting in

$$f(n) = c_1 + c_2 + (c_3 + c_4 + c_5 + c_{11})\min(n, n-1) + (c_6 + c_7 + c_8 + c_{10})\min(n, n-1, n-2, ...) = O(n).$$

In this pseudocode, the algorithm iterates through the array, comparing adjacent elements and swapping them if necessary. By updating the length variable to lastSwapIndex + 1, the algorithm ensures that it only considers the unsorted portion of the array in subsequent iterations. With this modification, the worst-case time complexity of the Bubble Sort algorithm becomes $O(n)$. In the worst-case scenario, where the input array is sorted in reverse order, the algorithm will make a single pass through the array, performing the necessary swaps to sort it.

Note that the pseudocode assumes the existence of a swap function that swaps the values of two elements in the array. Although this modified version improves the worst-case time complexity, it still has an average case complexity of $O(n^2)$.

The complexity class $O(\log n)$ signifies logarithmic complexity of an algorithm, where the execution time of the algorithm grows logarithmically with the size of the input data.

Let's consider the binary search algorithm (Fig. 9 and Fig. 10). Pseudocode for the binary search procedure shown in Fig 9.

DOI: 10.37943/15DNLB5877


```
Algorithm BinarySearch(array, targe_element):
    leftBoundary = 0
    rightBoundary = length(array) - 1
    while leftBoundary <= rightBoundary:
        midIndex = (leftBoundary + rightBoundary) / 2
        if array[midIndex] is equal to target_element then
            return midIndex
        else if array[midIndex] is greater than target_element then
            rightBoundary = midIndex - 1
        else
            leftBoundary = midIndex + 1
        end if
    end while

    return -1  // Element not found
End of the Algorithm
```

Figure 9. Algorithm of the binary search procedure

Let's break down how the algorithm of binary search (Fig. 9 and Fig. 10) works step by step:

Input:

- **array**: a sorted array in which you want to find the target_element;
- **target_element**: the element you want to find within the array.

Output:

- if the **target_element** is found in the array, the algorithm returns the index at which it is located;
- if the **target_element** is not found in the array, the algorithm returns -1.

Algorithm Steps:

1. Initialize two variables, **leftBoundary** and **rightBoundary**, which represent the search boundaries within the array. **leftBoundary** is initially set to 0 (the first element of the array), and **rightBoundary** is initially set to the index of the last element of the array (i.e., length(array) - 1).

2. Enter a while-loop that continues as long as **leftBoundary** is less than or equal to **rightBoundary**. This loop is the core of the Binary Search algorithm.

3. Calculate the **midIndex** by taking the average of **leftBoundary** and **rightBoundary**. This index represents the middle element of the current search range.

4. Compare the element at **array[midIndex]** with the **target_element**:

- if they are equal, the algorithm has found the **target_element** and returns **midIndex**;
- if **array[midIndex]** is greater than **target_element**, it means that the **target_element** should be located to the left of **midIndex**, so **rightBoundary** is updated to **midIndex - 1** to search in the left half of the current range;
- if **array[midIndex]** is less than **target_element**, it means that the **target_element** should be located to the right of **midIndex**, so **leftBoundary** is updated to **midIndex + 1** to search in the right half of the current range.

5. Repeat steps 3 and 4 until either the **target_element** is found (returning its index) or the **leftBoundary** surpasses the **rightBoundary**, indicating that the element is not in the array.

6. If the while-loop exits without finding the **target_element**, return -1 to signify that the element was not found in the array.

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 15, SEPTEMBER 2023

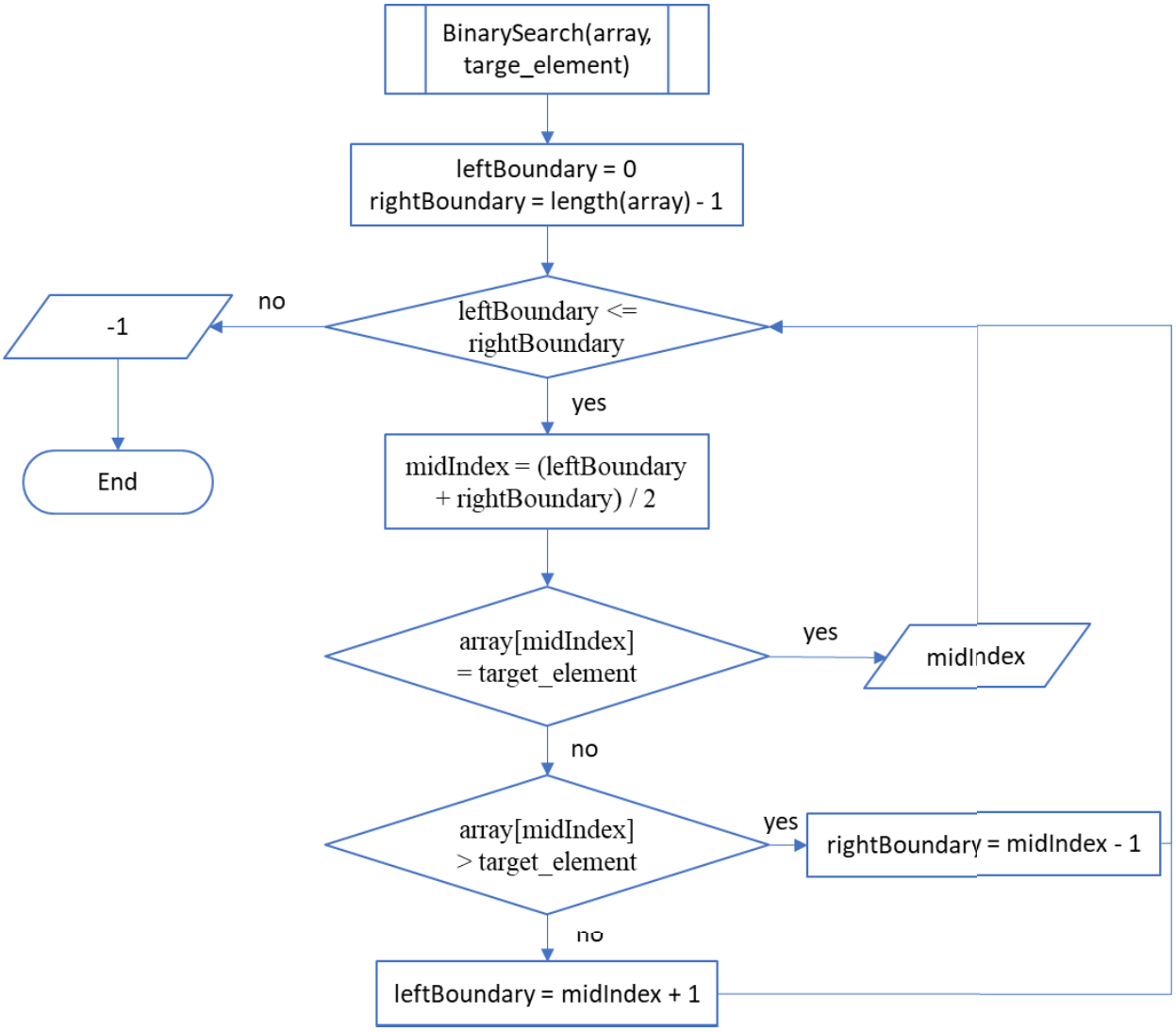

Figure 10. Flowchart of the binary search procedure

To analyze the complexity of the binary search algorithm, we consider the execution time by summing the products of the values from the "Cost" and "Times" columns in Table 4. The resulting expression is given by:

$$f(n) = c_1 + c_2 + (c_3 + c_{11})\log_2 n + (c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + c_{10})(\log_2 n - 1) = O(\log_2 n).$$

By applying mathematical simplifications, we can deduce that the complexity of the binary search algorithm is $O(\log_2 n)$, indicating a logarithmic growth rate with respect to the size of the input data. Binary search works efficiently on sorted arrays because it halves the search space in each iteration. This makes it significantly faster than linear search for large datasets.

The analysis of the best case for binary search is quite straightforward. In the best case, binary search completes in one step. The analysis of the worst-case is also simple. To complete binary search, it requires no more than $(\lfloor \log_2 n \rfloor + 1)$ steps. The proof of this statement can be found in [9].

DOI: 10.37943/15DNLB5877

Table 4. Analysis of the Binary Search Algorithm

Pseudocode	Cost	Times
leftBoundary = 0	c_1	1
rightBoundary = length(array) - 1	c_2	1
while leftBoundary <= rightBoundary	c_2	$\log_2 n$
midIndex = (leftBoundary + rightBoundary) / 2	c_4	$\log_2 n - 1$
if array[midIndex] is equal to target_element then	c_5	$\log_2 n - 1$
return midIndex	c_6	$\log_2 n - 1$
else if array[midIndex] is greater than target_element	c_7	$\log_2 n - 1$
rightBoundary = midIndex - 1	c_8	$\log_2 n - 1$
else	c_9	$\log_2 n - 1$
leftBoundary = midIndex + 1	c_{10}	$\log_2 n - 1$
end if	0	$\log_2 n - 1$
end while	0	$\log_2 n$
return -1	c_{11}	1

Linearithmic algorithms, also known as $O(n \log n)$ algorithms, represent a combination of linear and logarithmic complexity. They are commonly observed in sorting algorithms, such as merge sort and heap sort, as well as in certain matrix operations. These algorithms demonstrate a growth rate that falls between linear and logarithmic, resulting in efficient processing for large data sets.

On the other hand, certain problems, like the traveling salesman problem, demand an exhaustive exploration of all possible routes, resulting in exponential complexity. Such problems often pose significant computational challenges due to the rapidly increasing number of computations required as the input size grows.

To address these exponential complexity problems, researchers and practitioners strive to optimize algorithms and find polynomial-time solutions. By reducing the problem to polynomial complexity, the computational burden becomes more manageable, enabling efficient solutions within reasonable time frames.

Small-oh. The symbol "o" or small-oh (little-oh) also denotes the upper bound of the algorithm's runtime growth, but it is stricter than "O". "o" indicates that the algorithm's runtime growth is bounded less tightly than a certain function of the input size. For example, $o(n)$ means that the algorithm's runtime is slower than linear time, but the specific bounds depend on the context and analysis of the algorithm [7, 10].

Small-oh is a concept used to describe the lower bound of the algorithm's runtime growth. Small-oh notation is used to describe the time complexity of an algorithm that grows slower than another algorithm. Unlike Big-Oh, small-oh considers only strictly faster-growing functions [10-11]. It is defined as a bounding function that tends to zero as n approaches infinity. For example, if the runtime of an algorithm $f(n)$ is described by the function $g(n)$, we can write it as $f(n) = o(g(n))$.

Definition 2 [7]. Let $g(n)$ be a function. We define $o(g(n))$ as set of functions $f(n)$ that satisfy the following condition: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n_0 \geq 0$.

In other words, $o(g(n))$ is the collection of functions that have a runtime growth rate slower than that of $g(n)$, allowing for a looser upper bound. To illustrate this definition, consider the example: $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 15, SEPTEMBER 2023

The definitions of Big-Oh and small-oh notations are quite similar, with one key difference. In $f(n) = O(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for some constant $c > 0$. However, in $f(n) = o(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for all constants $c > 0$. Essentially, in small-oh notation, the function $f(n)$ becomes negligible compared to $g(n)$ as n grows larger (see Fig. 11). We can express this intuitively as:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Small-oh notation is often used in theoretical computer science and mathematics to describe the asymptotic behavior of functions and algorithms. However, it is less commonly used in real-world applications compared to Big-Oh notation.

The complexity of a small-oh algorithm means that the algorithm's execution time is strictly smaller than the specified growth function. In other words, the algorithm performs better than the specified function, at least for sufficiently large input sizes. Small-oh complexity is used in the analysis of algorithm complexity to provide a more precise estimate of their performance. It indicates that the algorithm's execution time is strictly smaller than the specified growth function for sufficiently large input sizes. This allows for a more accurate comparison of the performance of different algorithms and identifies more efficient solutions for specific tasks.

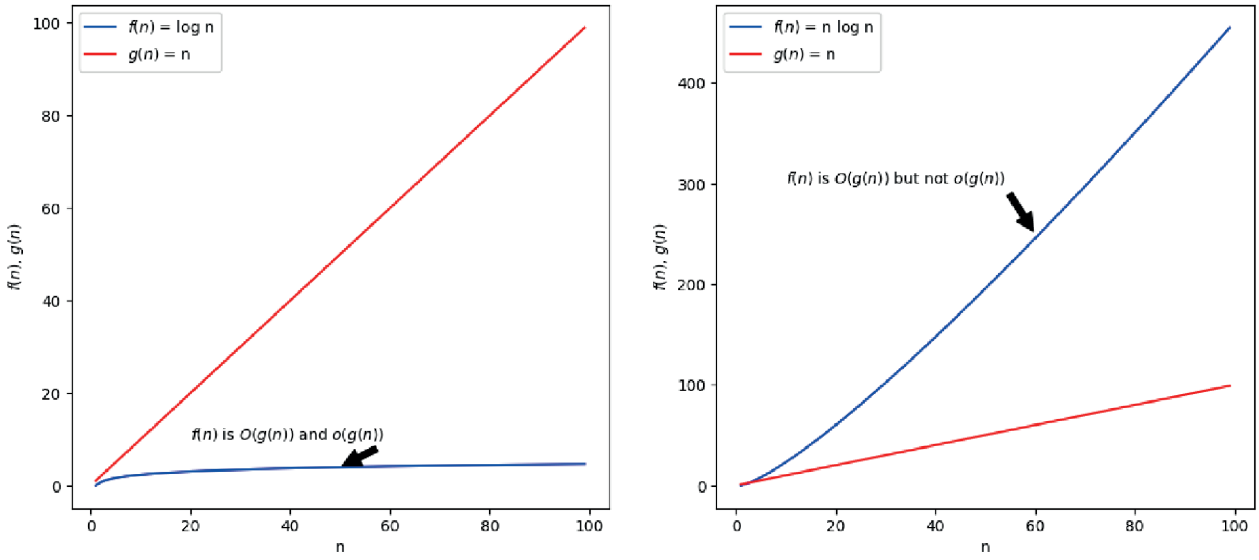

Figure 11. Illustration of Big-Oh and small-oh notations

The relationship between Big-Oh and small-oh notations lies in their common use for estimating algorithm complexity and execution time based on the size of input data. However, they have a small but important distinction.

Big-Oh notation describes the asymptotic upper bound on the growth of an algorithm's execution time. It provides an upper-bound estimate of how much time the algorithm will take as the input size increases. For example, an algorithm with a time complexity of $O(n^2)$ means that the execution time of the algorithm will not grow faster than the square of the input size. On the other hand, small-oh notation also describes the growth estimate of an algorithm's execution time but in a more precise manner. It indicates a strictly faster growth than the specified function for sufficiently large input sizes. An algorithm with a time complexity of $o(n^2)$ will have an execution time that grows faster than the square of the input size.

DOI: 10.37943/15DNLB5877

Using a combination of Big-Oh and small-oh notations helps to classify algorithms more precisely and determine how effectively they perform for different input sizes. For example, if an algorithm has a complexity of $O(n^2)$ and its execution time is strictly smaller than (n^2), we can use the notation $o(n^2)$ to indicate that the algorithm performs better than quadratic complexity. This allows for a more accurate comparison and assessment of algorithm performance, especially in cases where the algorithm's complexity does not match the exact boundary specified by the Big-Oh notation. The combination of Big-Oh and small-oh provides a more flexible and precise way of estimating the complexity and performance of algorithms based on the input size.

Application of Big-Oh and Small-oh in algorithm complexity analysis. The Big-Oh and small-oh notations are widely used in practice for analyzing the complexity of algorithms to evaluate and compare their performance. Here are a few ways these notations are used in practice:

1. Algorithm comparison: Big-Oh and small-oh notations allow for comparing different algorithms and evaluating their performance relative to the size of the input data. For example, if we have two algorithms with time complexities of $O(n)$ and $o(n^2)$, we can conclude that the first algorithm will be more efficient for larger input sizes.

2. Determining upper and lower complexity bounds: Big-Oh notation is used to indicate the asymptotic upper bound on the growth of an algorithm's execution time. This sets a limit on how quickly the execution time can increase as the input size grows. On the other hand, small-oh notation is used to indicate a strictly faster growth rate of execution time. This helps in identifying when an algorithm significantly outperforms other algorithms with the same or slower complexity [13].

3. Scalability estimation: Big-Oh and small-oh notations allow for predicting how the performance of an algorithm will change with the growth of the input size. Big-Oh provides an upper bound on the growth rate, while small-oh provides more precise estimates when the algorithm exhibits significantly better performance. This helps in determining how effectively an algorithm scales and performs with larger volumes of data.

4. Algorithm optimization: analyzing the complexity of an algorithm using Big-Oh and small-oh notations can help identify bottlenecks and determine parts of the algorithm that require optimization. If the algorithm's execution time has a complexity of $O(n^2)$, but in practice, it grows slower than n^2, it may signal the need for optimization and improving the algorithm's performance.

Limitations. The Big-Oh and small-oh notations are powerful tools for analyzing algorithm complexity, but they also have some limitations and boundaries of use. Here are some of them:

1. Asymptotic estimation: Big-Oh and small-oh notations provide information about the asymptotic behavior of an algorithm as the input size approaches infinity. They do not consider constants and lower-order terms, which can have a significant impact on the actual execution time of the algorithm for small input sizes. Therefore, they may not accurately reflect the real performance of the algorithm for specific cases.

2. Variability of hardware and software: the performance of an algorithm can depend on various factors, such as hardware, operating system, compiler, etc. Big-Oh and small-oh notations do not take these factors into account and provide only a general estimate of algorithm complexity. Therefore, the actual performance may differ from the theoretical estimation.

3. Ignoring best and worst cases: Big-Oh and small-oh notations usually provide information about the worst-case complexity of an algorithm. However, algorithms can have different time characteristics for different cases (best, average, worst). Therefore, evaluating only the worst-case may be insufficient for a complete understanding of algorithm performance.

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 15, SEPTEMBER 2023

4. Limited complexity classes: Big-Oh and small-oh notations represent complexity estimates of algorithms within certain classes (e.g., $O(n^2)$, $O(n \log n)$, etc.). They do not consider the possibility of more efficient algorithms outside these classes that can solve problems with lower complexity.

To better understand algorithm complexity and its real-world performance under specific conditions, it is essential to acknowledge its limitations and complement the use of Big-Oh and small-oh notations with other analytical methods and performance assessments. This comprehensive approach enables a more thorough evaluation of algorithm complexity and its practical efficiency.

Authors of [14] mentions that the Big-Oh problem is undecidable. While the article does not delve into a detailed discussion of the limitations of Big-Oh and small-oh notations, it does highlight some key points and discusses various variants of the threshold problem and approximation tasks, all of which are shown to be undecidable or recursively unsolvable. These findings imply that there may be constraints on the applicability of Big-Oh notation in specific scenarios. However, the article does not provide specific limitations or critiques of the notation itself.

Results

Big-Oh notation, which provides an upper bound on the growth rate of the algorithm's running time as a function of the input size. However, this method can sometimes be too imprecise or not capture the true behaviour of the algorithm. To create an algorithm that offers a more fine-grained analysis compared to the Big-Oh notation, let's consider a process that scrutinizes not just the upper bound (as Big-Oh does), but also integrates the specific computational steps and respective resources used by an algorithm. This method will still incorporate time complexity but will also yield insights into the operational nuances of the algorithm. Note that Big-Oh focuses on worst-case scenarios, while this approach will delve into the micro-level operations to provide a more comprehensive view of the algorithm's behavior.

Algorithm:

Input:

- $\mathbf{A}$: A target algorithm to analyze
- $\mathbf{S}$: A set of sample inputs for testing the algorithm

Output:

- Detailed report of the algorithm's complexity, capturing each operation, its frequency, and cost.

Step 1. Operation identification:

- For each distinct operation in algorithm **A**, identify and label it (e.g., addition, subtraction, loop initiation, etc.).
- Assign a symbolic cost to each operation, C_i, representing its fundamental computational cost.

Step 2. Dynamic profiling. For a set of input samples $\mathbf{S}$, run algorithm $\mathbf{A}$, keeping track of:

- The frequency, F_i, with which each operation is executed.
- The actual resource usage (time, memory, etc.) for each operation.

Step 3. Formulate comprehensive cost:

- Compute the total cost, T, using the formula:

$$T = \sum_{i=1}^{n} C_i \times F_i$$

where:

n – total number of distinct operations;

DOI: 10.37943/15DNLB5877

C_i – cost of the i^{th} operation;
F_i – frequency of the i^{th} operation.

- Compute the total actual resource usage, R, by summing the recorded resource usage for each operation from step 2.

Step 4. Empirical validation:

- Evaluate the algorithm $\mathbf{A}$ using different input sizes and compare the theoretical cost T with actual resource usage R.
- Use statistical methods to validate the correlation between T and R.

Step 5. Analyze and classify complexity:

- Analyze the total cost expression T to determine the theoretical complexity.
- Classify the complexity based on the actual resource usage trends obtained in step 4.

Let's create a hypothetical example using the algorithm for fine-grained analysis provided above. We'll consider a simple sorting algorithm for analysis (see Fig. 12).

```
Function BubbleSort(A)
    Input: An array A[1...n] of n elements
    Output: Array A sorted in non-decreasing order

    n ← length[A]

    for i ← 1 to n-1 do
        for j ← 1 to n-i do
            // Compare adjacent elements
            if A[j] > A[j+1] then
                // Swap A[j] and A[j+1]
                temp ← A[j]
                A[j] ← A[j+1]
                A[j+1] ← temp
            end if
        end for
    end for
    return A
End Function
```

Figure 12. Bubble sorting algorithm

Step 1: Operation identification

1. Compare operation: comparing A[j] and A[j+1]
2. Swap operation: swapping A[j] and A[j+1] if the condition is true
3. Loop operation: running the two loops

Assuming symbolic costs:

- C_1 – cost of compare operation;
- C_2 – cost of swap operation;
- C_3 – cost of loop operation.

Step 2: Dynamic profiling

Let's say we run the bubble sort algorithm for a sample set $\mathbf{S}$ of $A = [5, 1, 4, 2, 8]$:

- Compare operation frequency $F_1 = 10$ (since every pair is compared once per pass, and there are 4 +3+2+1=10 compares in total for our sample).
- Swap operation frequency $F_2 = 5$ (for our sample input).

Scientific Journal of Astana IT University
ISSN (P): 2707-9031 ISSN (E): 2707-904X
VOLUME 15, SEPTEMBER 2023

- Loop operation frequency $F_3 = 10$ (outer loop 4 times, inner loop 6 times for our sample input).

Step 3: Formulate comprehensive cost.

Assuming:

- $C_1 = 1$ unit time;
- $C_1 = 2$ unit times (as swapping involves three assignments);
- $C_3 = 0.5$ unit times.

$$T = C_1 \times F_1 + C_2 \times F_2 + C_3 \times F_3$$

$$T = 1 \times 10 + 2 \times 5 + 0.5 \times 10$$

$$T = 25$$

Step 4: Empirical validation.

We run the algorithm on multiple input sizes and compare the theoretically computed total time T to the actual time taken R to check if they are proportional. We would also examine the conditions (or types of data: random, almost sorted, reverse) under which the algorithm was tested.

Step 5: Analyze and classify complexity.

The obtained total cost expression T can be further analyzed for larger input sizes and different scenarios. We'll observe how T behaves as we alter the input size and validate against actual measurements. The generic time complexity of Bubble Sort is $O(n^2)$. The fine-grained analysis might reveal precise resource usage and variances in expected versus actual performance.

Discussion

This method not only provides a theoretical complexity (akin to Big-Oh) but also maps it to empirical data, offering a robust framework to analyze, validate, and optimize algorithms. It is particularly beneficial for critical applications where understanding and minimizing every bit of resource usage is paramount.

While it adds layers of complexity and may demand additional computational resources for analysis, the derived insights can be pivotal for optimizing algorithms, especially in scenarios where performance is crucial, such as real-time systems, high-frequency trading, or scientific computations.

Conclusion

In this article, the profound world of Big-Oh and small-oh notations, elucidating their pivotal roles in the analysis of algorithm complexity have been studied. Big-Oh notation serves as a valuable tool for estimating the upper bound on the growth rate of an algorithm's running time, while small-oh notation delineates a lower limit on this growth rate. A comprehensive look at various complexity classes defined by Big-Oh notation and provided the algorithm for more fine-grained analysis of algorithm complexity have been taken.

The analysis of algorithm complexity using these notations holds paramount importance in the fields of programming and computer science. It equips developers and researchers with the means to make informed decisions when it comes to selecting and optimizing algorithms [12]. However, it's crucial to acknowledge that while complexity analysis is a vital facet of effective programming, ongoing research endeavours may yield more refined methodologies and approaches within this domain.

Looking ahead, it's evident that the world of algorithmic analysis and optimization is dynamic and ever evolving. Future research may focus on developing even more precise methods

for analyzing algorithm complexity, allowing us to assess algorithms with greater granularity. This can lead to the optimization of algorithms for specific use cases and data sizes. With the increasing use of machine learning (ML) and artificial intelligence (AI), there is an opportunity to explore how these notations can be integrated into the analysis of complex ML algorithms. This can lead to a deeper understanding of the efficiency and scalability of AI systems [15, 16].

Applying these notations to real-world, large-scale applications such as distributed systems, cloud computing, and big data analytics will be an exciting area of exploration. Understanding how these notations apply in practical, complex scenarios is essential.

In conclusion, the world of algorithm design and analysis will continue to push boundaries and explore new horizons. Big-Oh and small-oh notations remain invaluable tools for navigating the complex terrain of algorithmic analysis and optimization. As researchers and practitioners, embracing these notations and staying attuned to emerging methodologies will be crucial in shaping the future of efficient algorithm design.

References

[1] Sipser, M. (2013). *Introduction to the theory of computation (3rd ed.)*. Cengage Learning.
[2] Shaffer, C.A. (2013). *A practical introduction to data structures and algorithm analysis* (3rd ed.). Prentice Hall.
[3] Ericson, F. (2019). *Algorithms*. Pearson Education.
[4] Sedgewick, R., & Flajolet, P. (2013). *An introduction to the analysis of algorithms (2nd ed.)*. Addison-Wesley Professional.
[5] Kleinberg, J., & Tardos, É. (2022). *Algorithm design (2nd ed.)*. Pearson Education.
[6] Chakraborty, P., Khatoon, R., & Dutta, S. (2019). *A guide to design and analysis of algorithms*. CRC Press.
[7] Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2022). *Introduction to algorithms (4th ed.)*. MIT Press.
[8] Muller, S., & Massaron, L. (2017). *Algorithms for dummies*. John Wiley & Sons.
[9] Lee, C.-H., Tseng, H.-Y., Chang, Y.-H., & Tsai, W.-T. (2017). *Introduction to the design and analysis of algorithms: A strategic approach*. CRC Press.
[10] Skiena, S.S. (2020). *The algorithm design manual, 3rd edition*. Springer.
[11] Lafore, R. (2014). *Data structures and algorithms in Java, 6th edition*. Pearson.
[12] Chistikov, D., Kiefer, S., Murawski, A. S., & Purser, D. (2020). *The Big-O Problem for Labelled Markov Chains and Weighted Automata*. Leibniz International Proceedings in Informatics, 166, 1-19. https://doi.org/10.4230/LIPIcs.CONCUR.2020.41.
[13] Bouyer, P., Markey, N., & Ouaknine, J. (2019). *The Big-O problem*. Logical Methods in Computer Science, 15(3), 1-44. https://doi.org/10.23638/LMCS-15(3:1)2019.
[14] Agarwal, P. K., Har-Peled, S., & Varadarajan, K. R. (2014). *Approximate nearest neighbor for polygonal curves under Fréchet distance*. Leibniz International Proceedings in Informatics (LIPIcs), 29-40. https://doi.org/10.4230/LIPIcs.STACS.2014.29
[15] Pugliese, R., Regongdi, S., & Marini, R. (2021). *Machine learning-based approach: global trends, research directions, and regulatory standpoints*. Data Science and Management, 4, 19-29. https://doi.org/10.1016/j.dsm.2021.12.002.
[16] Xu, Y., Liu, X., Cao, X., Huang, C., Liu, E., Qian, S., Liu, X., Wu, Y., Dong, F., Qiu, C.-W., Qiu, J., Hua, K., Su, W., Wu, J., Xu, H., Han, Y., Fu, C., Yin, Z., Liu, M., Roepman, R., & Zhang, J. (2021). *Artificial intelligence: A powerful paradigm for scientific research*. The Innovation, 2(4), 100179. https://doi.org/10.1016/j.xinn.2021.100179.