Discussion-3: Introduction to Algorithm Efficiency

Discussion Topic:

Define the concepts of logarithm, linear, and quadratic when analyzing algorithms. What are some of the pros and cons associated with utilizing Big-Oh notation to analyze an algorithm? How can analyzing the efficiency of an algorithm lead to improvements in the initial design?
In your answer, specifically, think of and give a real-life scenario where:

- Algorithm analysis is important
- Algorithm efficiency is paramount

My Post:

In computer science, Big-Oh notation is used to describe the time complexity or space complexity of algorithms (Geeks for Geeks, 2024). Mathematically, it defines the upper bound of an algorithm's growth rate, known as the asymptotic upper bound, and is denoted as $f(n)$ is $O(g(n))$ or $f(n) \in O(g(n))$, pronounced $f(n)$ is Big-Oh of $g(n)$. The term "asymptotic" refers to the behavior of the function as its input size n approaches infinity. In the context of computer science, it describes the worst-case scenario for time complexity or space complexity. For example, an algorithm with $O(n^2)$ time complexity will grow much faster than one with $O(n)$ as the input size increases, with n representing the number of primitive operations. Primitive operations are low-level instructions with a constant execution time, such as assigning a value to a variable, performing an arithmetic operation, comparing two values, or accessing an element in an array by its index.

Definition of Big-Oh:

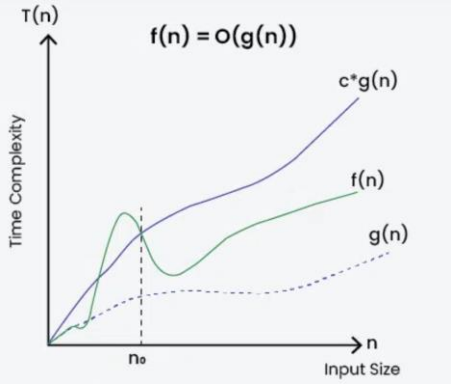
Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers.

We say that $f(n) \in O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n) \quad , \quad for\ n \geq n_0$$ (Carrano & Henry, 2018)

The Graphical representation of the relationship:

Figure 1:
Big-Oh Time complexity



Note: From Big O notation tutorial – A guide to Big O by Geeks for Geeks (2024)

Here are some of the properties of the Big-Oh notation:

- ***f(n) = a*** **is** ***O(1)***, where *a* is a constant. Ex: *a = 4, f(5) = 4 and f(7) = 4*
  justification:
  a ≤ a, hence *a = a => a = 1a = ca*, for *c = 1,* when $n \geq n_0 = 1$

- ***f(n) = 7n + 2*** **is** ***O(n)***
  justification:
  7n + 2 ≤ (7+2)n = cn, for *c = 9, when* $n \geq n_0 = 1$

- ***f(n) =*** $6n^4 + 2n^3 + n^2 - 3n$ **is** $O(n^4)$
  justification:
  $6n^4 + 2n^3 + n^2 - 3n \leq (6+2+1-3)\, n^4 = cn^4$, for *c = 6,* when $n \geq n_0 = 1$

- ***f(n) =*** $7n^3 + 5n \log n + 9n$ **is** $O(n^2)$
  justification:
  $7n^3 + 5n \log n + 9n \leq (7+5+9)\, n^3 = cn^3$, for *c = 6,* when $n \geq n_0 = 1$

- ***f(n) = 5 log n + 3*** **is** ***O(log n)***
  justification:
  5n log n + 3 ≤ (5+3) log n = c log n, for *c = 8* and $n \geq 2$, when $n \geq n_0 = 2$.
  Note that *log n* is *0* for *n = 1*.

- ***f(n) =*** 3^{n+2} **is** $O(3^n)$
  justification:
  $3^{n+2} \leq 3^2 + 3^n$, hence $3^{n+2} = 9 \cdot 3^n = c3^n$, *for c = 9* when $n_0 = 1$

- ***f(n) = 7n + 75 log n*** **is** ***O(n)***
  justification:
  7n + 75 log n ≤ (7+75)n = cn, for *c = 82,* when $n \geq n_0 = 1$

(Carrano & Henry, 2018)

Not that the Big-Oh notation eliminates constant terms and lower factors.

The major function types used to analyze time complexity are Logarithm, Linear, Quadratic, and Constant Time Complexities, below is the definition of these functions:

- **Constant Time Complexity -** ***O(1)***
  O(g(n)) where *g(n) = c*
  For some fixed constant c, such as *c = 2*, *c = 33*, or *c = 300*.
  The time complexity remains constant, regardless of the input size. The algorithm takes the same amount of time to complete, no matter how large *n* gets. This is the most efficient time complexity.

- **Logarithmic Complexity - *O(log n)***
  O(g(n)) where *g(n) = log n*
  Note that in computer science *log n = log_2 n*
  And *x = log_2 n iff $2^x = n$*
  The time complexity grows logarithmically with input size, meaning it increases slowly.

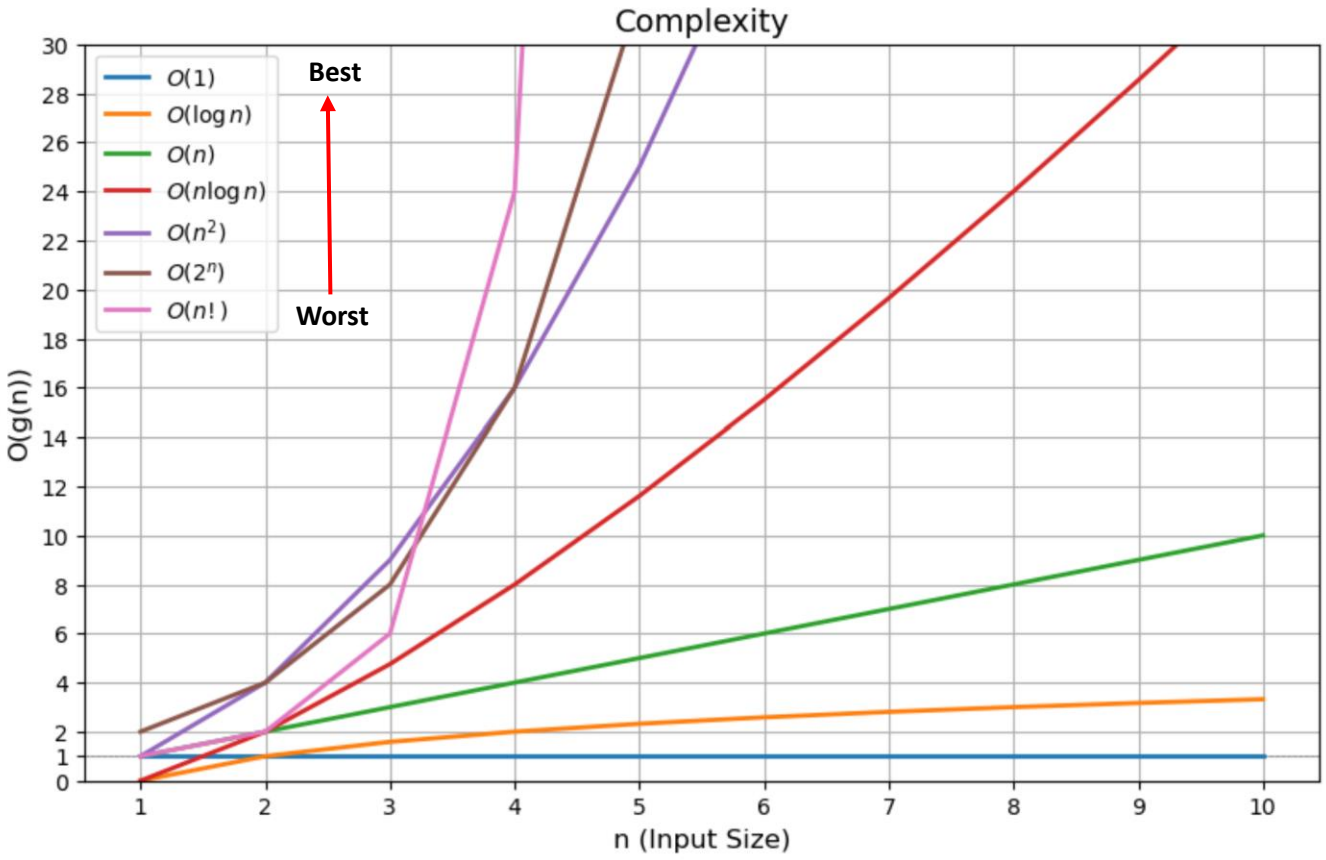
- **Linear Complexity - *O(n)***
  O(n) where *g(n) = n*
  Given an input value *n*, the linear function *g* assigns the value *n* itself.
  The time complexity grows linearly with input size, meaning the time taken increases directly in proportion to the input size.

- **Quadratic Complexity - *$O(n^2)$***
  $O(n^2)$ where *$g(n) = n^2$*
  The time complexity grows quadratically with input size, meaning the time taken increases by the square of the input size.

As shown above, the time or space complexity directly relates to the Big-Oh bounding function *g(n)* type. For example, the Big-Oh quadratic function $O(n^2)$ has a greater growth rate than the logarithmic function *O(log n)*. In other words, $O(n^2)$ has greater complexity than *O(log n)* making the algorithm associated with $O(n^2)$ worse than the one related to *O(log n).* See Figure 2 for different Big-Oh functions' growth rates.

Figure 2

Big-OH - O(g(n)) Growth Rates



The table below illustrates various O(g(n)) related to common computer science algorithms:

Table 1

Big-Oh Notation Summary

Notation	Type	Examples	Description
O(1)	Constant	Hash table access	Remains constant regardless of the size of the set
O(log n)	Logarithmic	Binary search of a sorted table	Increases by a constant. If n doubles, the time to perform increases by a constant, smaller than n amount
O(<n)	Sublinear	Search using parallel processing	Performs at less than linear and more than logarithmic levels
O(n)	Linear	Finding an item in an unsorted list	Increases in proportion to n. If n doubles, the time to perform doubles
O(n log(n))	n log(n)	Quicksort, Merge Sort	Increases at a multiple of a constant
O(n^2)	Quadratic	Bubble sort	Increases in proportion to the product of n*n
O(c^n)	Exponential	Traveling salesman problem solved using dynamic programming	Increases based on the exponent n of a constant c
O(n!)	Factorial	Traveling salesman problem solved using brute force	Increases in proportion to the product of all numbers included (e.g., 1*2*3*4...)

Note: From Big O Notation by Cowan (n.d.).

Big-Oh Pros and Cos

The Big-Oh notation has both pros and cons. One of the main advantages is that it provides a clear way to express algorithm efficiency by eliminating constant factors and lower-order terms, allowing the focus to be on how the algorithm's performance scales as the input size grows, rather than being tied to specific computing system specifications or the type of compiler used. In other words, it is platform independent and programming language agnostic. Although both will have a significant impact on the final running time of the algorithms; however, that impact will most likely be a constant multiple (Educative, n.d.).

The major disadvantage of Big-Oh notation is that it loses important information by eliminating constant factors and lower-order terms, and by ignoring computing system specifications and the type of compiler used. Additionally, it only focuses on the worst-case scenario, the upper bound, leaving out potential insights into how the algorithm might perform under typical conditions or with smaller input sizes. In

other words, the Big-Oh notation gives the worst big picture of algorithm performance and ignores the rest of the nuances that could be crucial for better understanding the algorithm's behavior.

The Importance of Big-Oh

Big-Oh analysis, or any other algorithm analysis, can help uncover performance bottlenecks, identify opportunities for optimization, and influence software design by selecting more appropriate algorithms for specific tasks. Making Big-Oh and any other algorithm analysis crucial for developing efficient, and reliable software systems.

One real-world application where Big-Oh analysis is essential for efficiency is in e-commerce platforms like Amazon, where millions of products need to be searched quickly and accurately. An inefficient search algorithm can directly impact user experience by slowing down searches, frustrating potential customers, and leading to abandoned carts and missed sales. Thus, optimizing or selecting the right search algorithms using Big-Oh or another algorithm analysis is paramount for the efficiency of the application and user experience.

In conclusion, Big-Oh notation is a powerful tool for evaluating and comparing the efficiency of algorithms. While it has limitations in ignoring constant factors and average-case scenarios, its ability to highlight worst-case performance and guide algorithm selection makes it indispensable in developing efficient and reliable software systems, particularly in high-demand applications like e-commerce platforms.

-Alex

References:

Carrano, F. M., & Henry, T. M. (2018, January 31). Algorithms: Algorithm Analysis. *Data structures and abstractions with Java (5th Edition). Pearson.*

Cowan, D. (n.d.). Big O Notation. *Down Cowan Blog.* https://www.donkcowan.com/blog/2013/5/11/big-o-notation

Educative (n.d.). *Advantages and disadvantages of the Big-O notation*. Educative. https://www.educative.io/courses/algorithmic-problem-solving-preparing-for-a-coding-interview/advantages-and-disadvantages-of-the-big-o-notation

Geeks for Geeks (March 29, 2024). Big O notation tutorial – A guide to Big O analysis. https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/Links to an external site.