On Asymptotic Notation: an Introduction to Analyses of Algorithms

Sam Atamazhori, Arsalan Hassanvand, and Sepehr Omidvar.

August 2, 2021

Abstract

In this paper we will introduce the asymptotic notation and will explore its history and usage in the modern contexts and prove some elementary properties and facts.

1 Introduction

the analysis of algorithms is the process of finding the computational complexity of algorithms the amount of time, storage, or other resources needed to execute them. Usually, this involves determining a function that relates the length of an algorithm's input to the number of steps it takes (its time complexity) or the number of storage locations it uses (its space complexity). An algorithm is said to be efficient when this function's values are small, or grow slowly compared to a growth in the size of the input. Different inputs of the same length may cause the algorithm to have different behavior, so best, worst and average case descriptions might all be of practical interest. When not otherwise specified, the function describing the performance of an algorithm is usually an upper bound, determined from the worst case inputs to the algorithm. The term "analysis of algorithms" was coined by Donald Knuth. Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

1.1 Cost models

Time efficiency estimates depend on what we define to be a step. For the analysis to correspond usefully to the actual execution time, the time required to perform a step must be guaranteed to be bounded above by a constant. One must be careful here; for instance, some analyses count an addition of two numbers as one step. This assumption may not be warranted in certain contexts. For example, if the numbers involved in a computation may be arbitrarily large, the time required by a single addition can no longer be assumed to be constant. In this paper we will use the **uniform cost model**, also called uniform-cost measurement, which assigns a constant to each machine operation.

1.2 Input size

Different input requires different amount of resources. Instead of dealing with input itself, we prefer to deal with the Size of Input. The amount of memory required for representing the input with respect to a fixed coding scheme.

Example 1.1. *The input size can be any of these depending on the data structure:*

- *For an array The size can be considered as the number of its elements.*
- *For an integer The size can be considered as the number of its bit in binary representation.*
- *For a graph The size can be considered as the number of vertices or edges.*

1.3 Time complexity

The time complexity is the computational complexity that describes the amount of computer time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken and the number of elementary operations performed by the algorithm are taken to differ by at most a constant factor.
Since an algorithm's running time may vary among different inputs of the same size, one commonly considers the worst-case time complexity, which is the maximum amount of time required for inputs of a given size. Less common, and usually specified explicitly, is the average-case complexity, which is the average of the time taken on inputs of a given size (this makes sense because there are only a finite number of possible inputs of a given size). In both cases, the time complexity is generally expressed as a function of the size of the input. Since this function is generally difficult to compute exactly, and the running time for small inputs is usually not consequential, one commonly focuses on the behavior of the complexity when the input size increases—that is, the asymptotic behavior of the complexity. Therefore, the time complexity is commonly expressed using big O notation.

2 Asymptotic notation

In mathematical analysis, asymptotic analysis, also known as asymptotics, is a method of describing limiting behavior. This idea was imported from analysis into complexity theory as a mean to describe the time efficiency of algorithms. In this paper we will define these types of notation most common in computer science, collectively called Bachmann–Landau [1] [2] notation or asymptotic notation, which were invented by Paul Bachmann, Edmund Landau and adopted into this field by Donald Knuth. The Big O notation is the most famous of these notations, and we mainly focus on that.

2.1 The Big O notation

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.

In computer science, big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows. In analytic number theory, big O notation is often used to express a bound on the difference between an arithmetical function and a better understood approximation; a famous example of such a difference is the remainder term in the prime number theorem. Big O notation is also used in many other fields to provide similar estimates.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation. The letter O is used because the growth rate of a function is also referred to as the order of the function. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function.

Definition 1 (Big O notation)**.** *Let f be a real or complex valued function and g a real valued function. Let both functions be defined on some unbounded subset of the positive real numbers, and $g(x)$ be strictly positive for all large enough values of x. One writes:*

$$f(x) = O\big(g(x)\big) \quad x \to \infty$$

If the absolute value of $f(x)$ is at most a positive constant multiple of $g(x)$ for all sufficiently large values of x. That is, $f(x) = O\big(g(x)\big)$ if there exists a positive real number c and a real number x_0 such that: [see figure 1]

$$|f(x)| \leq cg(x) \quad \forall x \geq x_0$$

In many contexts, the assumption that we are interested in the growth rate as the variable x goes to infinity is left unstated, and one writes more simply that

$$f(x) = O\big(g(x)\big)$$

2.2 The Big Theta

The big Theta was first introduced by Donald Knuth[3].This function gives an upper and a lower bound, a much tighter bound for the behavior of functions and algorithms. The Big Theta can be read as the "exact order of $f(x)$".

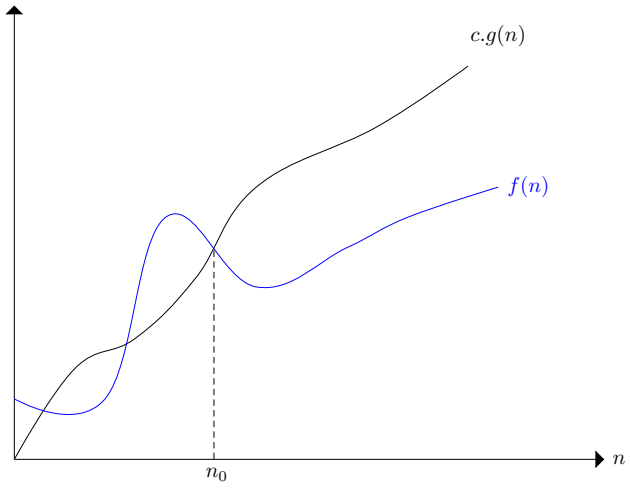

Figure 1: $g(n)$ is the smallest upper bound.

Definition 2 (Big Theta)**.** *Let $f(x)$ and $g(x)$ be functions with values in the set of real numbers. We write $f(x) = \Theta(g(x))$, if there exists c_1, c_2, n_0 such that for every $n \geq n_0$ we have:*

$$\exists c_1, c_2, n_0 \quad \forall n \geq n_0 \quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

[see figure 2]

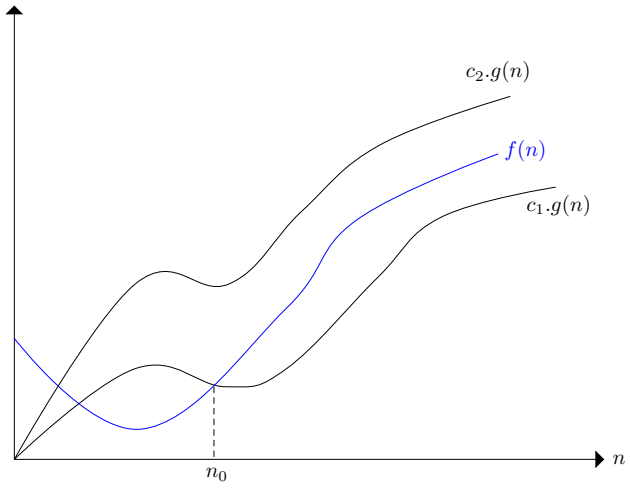

Figure 2: $g(n)$ is the smallest upper bound.

2.3 The Big Omega

There are two widespread and incompatible definitions of this notation, one is introduced by Hardy–Littlewood [4] and the other by Knuth. In this paper and in the field of complexity theory we use the definition by Knuth. This function gives us a lower bound. It can be read as "order at least $f(n)$"

Definition 3 (Big Omega)**.** *Let $f(x)$ and $g(x)$ be functions with values in the set of real numbers. We write $f(x) = \Omega(g(x))$, if there exists c, n_0 such that for every $n \geq n_0$ we have:[see figure 3]*

$$\exists c, n_0 \quad \forall n \geq n_0 \quad 0 \leq cg(n) \leq f(n)$$

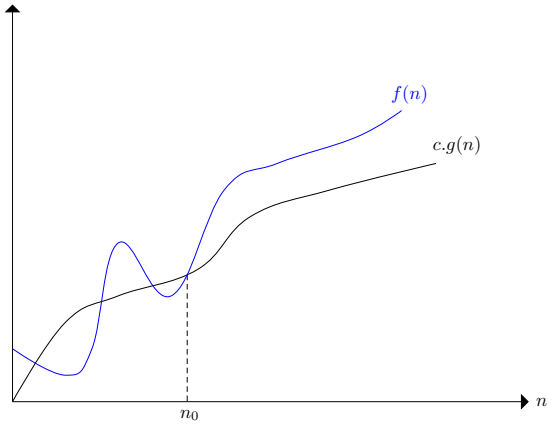

Figure 3: $g(n)$ is the largest lower bound.

2.4 Summary and further reading

We have defined the Omicron, Omega and Theta notation. For a summary of the whole asymptotic notation see the table below. In the next section we will discuss basic properties and theorems related to these notations. For a more detailed look at this topic the authors recommend the reader to examine Knuth's paper [3].

Notation	Definition	Meaning
$f(n) = \mathcal{O}(g(n))$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$	f grows much more slowly than g.
$f(n) = \Omega(g(n))$	$g(n) = \mathcal{O}(f(n))$	f grows at least as fast as g
$f(n) = \Theta(g(n))$	$f(n) = \mathcal{O}(g(n))$ $f(n) = \Omega(g(n))$	f and g have about the same order of magnitude.
$f(n) \sim g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$	$f(n)$ and $g(n)$ are almost the same.

3 Theorems and properties

In this section we will prove a basic theorem and provide a list of properties that these notations poses. Some of the properties are quit natural and follow immediately form the definitions so we have left these proofs to the reader and will not dwell on them too much. We first begin by proving the theorem below.

Theorem 3.1. *let $f(n) and g(n)$ be real valued function. $f(n) = \Theta(g(n))$ if and only if $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$.*

Proof. we use the definition given in the previous section and re-write the terms.

$$f(n) = \mathcal{O}(g(n)) \iff \exists c, n_0 > 0 \mid \forall n \geq n_0, 0 < f(n) \leq c_0.g(n)$$
$$f(n) = \Omega(g(n)) \iff \exists c, n_1 > 0 \mid \forall n \geq n_0, 0 < c_1.g(n) \leq f(n)$$

now by taking $n_2 = max(n_0, n_1)$, we have:

$$f(n) = \Theta(g(n)) \iff \exists c_1, c_2, n_2 > 0 \mid \forall n \geq n_2, 0 < c_1.g(n) \leq f(n) \leq c_2.g(n)$$

□

Now we list some basic properties:

- Reflexivity: $f(n) = \Theta(f(n))$, $f(n) = \mathcal{O}(f(n))$, $f(n) = \Omega(f(n))$
- Symmetry: $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$
- Transitivity:
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n))$
  - $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n)) \implies f(n) = \mathcal{O}(h(n))$
  - $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \implies f(n) = \Omega(h(n))$
- Transpose symmetry: $f(n) = \mathcal{O}(g(n)) \iff g(n) = \Omega(f(n))$

3.1 The Big $\mathcal{O}$ arithmetic

We list some facts about The Big $\mathcal{O}$ notation that are useful for calculating the order of functions. Assume $f_1 = \mathcal{O}(g_1) and f_2 = \mathcal{O}(g_2)$:

addition: $f_1 + f_2 = \mathcal{O}(max(g_1, g_2))$

multiplication: $f_1 f_2 = \mathcal{O}(g_1 g_2)$
$f.\mathcal{O}(g) = \mathcal{O}(fg)$
let k be a non-zero constant, $\mathcal{O}(|k|g) = \mathcal{O}(g)$

We encourage the reader to prove these facts to better understand the concept and get familiar with these kinds of proofs involving asymptotic notation. It should be stated that the facts given above are necessary for calculating orders of functions.

4 Classifying Functions

We have established a rigorous mathematical foundation for analysis of algorithms. Using facts and theorems introduced in the previous section we can tackle any problem involving the asymptotic notation. There are infinitely many functions. In order to compare them with each other, we prefer to classify them as simple functions without loss of their properties. In fact we want to choose one simple function for each class of functions. Below, we have provided a list of the most used and famous classes with an algorithmic example for each entry.

Notation	Name	Example
$\mathcal{O}(1)$	constant time	Determining if a binary number is even or odd
$\mathcal{O}(\log(n))$	logarithmic	Finding an item in a sorted array with a Binary search algorithm.
$\mathcal{O}(n)$	linear	Finding an item in an unsorted list or in an unsorted array.
$\mathcal{O}(n\log(n))$	quasilinear, or "n log n"	Performing a fast Fourier transform.
$\mathcal{O}(n^2)$	quadratic time	Multiplying two "n"-digit numbers by a simple algorithm.
$\mathcal{O}(n^c)$	polynomial or algebraic	maximum Matching (graph theory) for bipartite graphs.
$\mathcal{O}(c^n)$	exponential	Finding the (exact) solution to the travelling salesman problem
$\mathcal{O}(n!)$	factorial	Solving the travelling salesman problem via brute-force search.

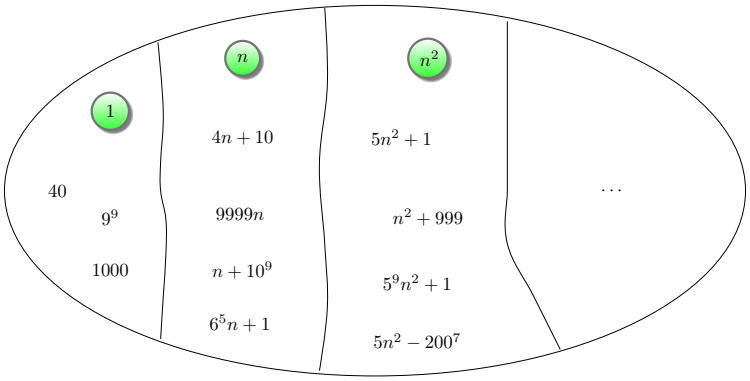

Figure 4: Classifying functions

References

[1] Paul Bachmann. *Analytische Zahlentheorie [Analytic Number Theory] (in German).* Leipzig: Teubner, 1894.

[2] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen [Handbook on the theory of the distribution of the primes] (in German).* Leipzig: B. G. Teubner, 1909.

[3] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, 1976.

[4] G. H. Hardy and J. E. Littlewood. Some problems of diophantine approximation: Part II. the trigonometrical series associated with the elliptic theta-functions. *Acta Mathematica*, 37(0):193–239, 1914.

[5] Jiri Matousek and Jaroslav Nesetril. *Invitation to Discrete Mathematics*. Oxford University Press, Inc., USA, 1998.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.