Polynomial computer algorithm performance complexity analysis of $O(n \cdot log(n))$. Sub-zero field of positive infinities.

Dmytro Brazhnyk
Master Degree in Computer Science

August 2022

Abstract

The objective of this study is to provide a mathematically accurate notation for algorithm performance complexity comparison. The first objective is to show that computation complexity between $O(n \cdot log(n))$ and $O(n)$ is insignificant [1]. Computer science articles quite often differentiate $O(n)$ and $O(log(n))$ computation complexity, however when it comes to practical application, software engineers intuitively quite often ignore that difference, so the objective of this article is to show science background that the difference can be ignored, so it will simplify the decision process for engineering process. Another objective is to propose a polynomial table of complexities that better represents algorithmic complexity and how to compare them. Also, an addendum to the main part of the theory is the sub-zero field of positive infinities.

1 About author

Dmytro Brazhnyk is solution architect with strong software engineering background in computer science. Was participating in computer contest awarded as winner among other contestant and has skills in computer algorithm and analysis.

2 Introduction

Thomas H. Cormen spent at least 200 pages in his great Book about algorithm[4] to prove that some algorithms are better than others, because they are running in $O(n)$ instead of $O(n \cdot log(n))$, and it is repeated in many other computer science articles related to computation performance during the last 60 years, but usually intuitively ignored by engineers. The goal of this paper is to show the difference $O(n)$ and $O(n \cdot log(n))$, and show it's insignificant. It also proposes for another notation to use polynomial power for algorithm computation comparison which more accurately represents difference.

3 Overview

The following article provides proof of the subject of this matter of study from three points of view:

- the First viewpoint will be from a more practice standpoint, where we consider that in the real world there is quite a limitation for computer resources, and considering the current existent known architecture we can quite easily go from $O(n \cdot log(n))$ into $O(n)$.
- the Second viewpoint is explaining the dynamics, of how the computer hardware capability may evolve, and how it affects the computational complexity, and the purpose to show that approximation of $O(n \cdot log(n))$ to $O(n)$ is still valid.
- And the last, in this article, will give a mathematical proof, that the given equation is correct: $lim_{n \to \infty} O(n \cdot log(n)) = O(n)$, for any $n \to \infty$

The conclusion of this article will show how polynomial factor notation can represent computation complexities in a better way.

4 Real world examples

Let's assume: $f(n)$ is complexity of some $n \cdot log(n)$ function.

$\boxed{f(n) = O(n \cdot log(n))}$ - formula (1)

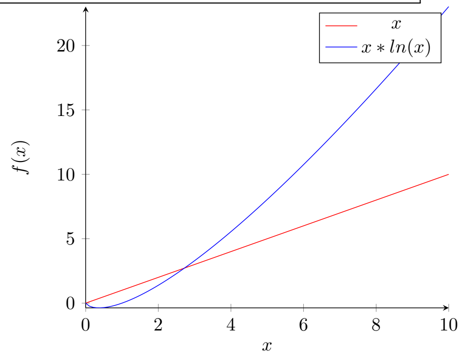
Considering that majority of computer algorithms are bound to computer architecture as in the table below, the maximum possible value of n is defined by the computer register bit size of the given architecture[2], [3]

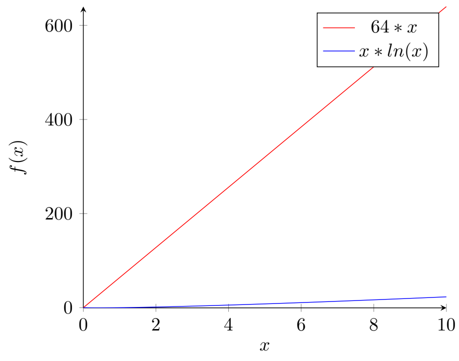
Computer Architecture	Register bit size	Max computer integer size
8080	8	2^8
x86	16	2^{16}
i386	32	2^{32}
x64	64	2^{64}

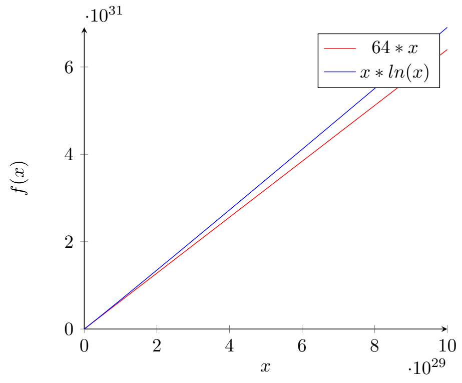
So we can say that n is one value of bellow:
$n \in [2^8, 2^{16}, 2^{32}, 2^{64}]$

If we'll take practical maximum for x64, for any $n \leq 2^{64}$ we can rewrite the formula (1) in a next way:
$f(n) = O(n \cdot log(n)) = O(n \cdot log(2^{64})) = O(64 \cdot n \cdot log(2)) = O(n)$

So,
$\boxed{\text{O}(n \cdot log(n)) = O(n), \text{ for any } n \leq 2^{64}}$ - formula (2)







$2^{64} = 10^{19.26} \ll 10^{31}$, 10^{31} - is scale on the chart.

5 Geometric growth

Let's extend this approach for future computer generation growth, let's assume that:
g - is the computer generation index, and every new generation doubles the number of bits b in computer architecture, so:
$b = 8$, for $g = 1$
$b = 16$, for $g = 2$
$b = 32$, for $g = 3$
$b = 64$, for $g = 4$

so there is a generic formula between the number of bits and the computer generation index,

$b = 8 \cdot 2^g = 2^{g+3}$, while $g+3$ can be substituted into variable $k = g+3, b = 2^k$

And we can assume that computation complexity argument n be limited by computer architecture, so:
$\boxed{n \leq 2^b = 2^{2^k};}$ - formula (3)

Now taking formula (1) and maximum possible value (3), considering that formula (1) is a monotonically grown function, we can say that:
$n \cdot log(n) \leq 2^{2^k} \cdot log(2^{2^k}) = 2^k \cdot 2^{2^k} * log(2) = 2^{2^k+k} \cdot log(2)$
$O(2^{2^k+k} \cdot log(2)) = O(2^{2^k+k})$
$\boxed{O(n \cdot log(n)) \leq O(2^{2^k+k})}$ - formula (4)

For smaller computer architerure generation index, where $g \in [1, 2, 3, 4]$ in formula (2) we've already proved that $O(n \cdot log(n)) = O(n)$

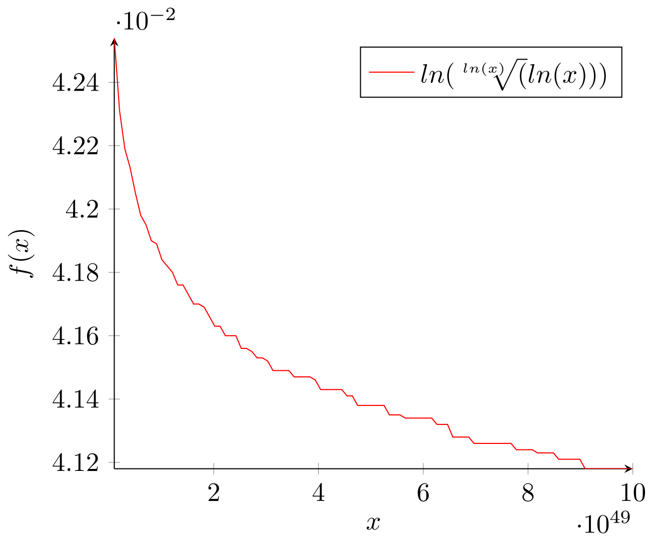
Now let's see what happening for any $g \to \infty$, so substitution above
$k = g+3$, considering that $g \to \infty$, then $k \to \infty + 3$, so $k \to \infty$, now:
Considering $lim_{k\to\infty} O(2^k + k) = O(2^k)$,
$\boxed{lim_{k\to\infty} O(2^{2^k+k}) \text{ is something similar to } lim_{k\to\infty} O(2^{2^k}) = O(n)}$ - formula (5)
formula (5) prove in the next section

Considering formula (4) and formula (5),
$lim_{n\to\infty} O(n \cdot log(n)) \leq lim_{k\to\infty} O(2^{2^k+k}) \leq O(n)$, therefore $\boxed{lim_{n\to\infty} O(n \cdot log(n)) \leq O(n)}$, while n - is defined by a final number of bits of computers of next generations.

6 Generic mathematical proof of formula (5)

Starting from formula(1) again:
$f(n) = O(n \cdot log(n)) = O(n \cdot n^{log_n(log(n))}) = O(n^{1+\frac{log(log(n))}{log(n)}}) = O(n^{1+log(\sqrt[log(n)]{log(n)})})$,
- formula (6)

considering that $lim_{x\to\infty} \sqrt[x]{x} = 1 + \epsilon, \epsilon \to +0$, so and
$\boxed{lim_{n\to\infty} \sqrt[log(n)]{log(n)} = 1 + \epsilon, \epsilon \to +0}$, because $log(n)$ is monotonically increasing function iself, here the plot bellow



As an example on the chart, we can observe that $\epsilon = 4 \cdot 10^{-2} = 0.04$, for $n = 10^{49}$ using natural logarithm and it is moving to zero, so
$n \cdot log(n) = 10^{49} \cdot (10^{49})^{0.04} = 10^{49} \cdot 10^{49 \cdot 0.04} = 10^{49} \cdot 10^{1.96} = 10^{49+1.96} > 10^{49}$
while comparing it with x64 $max(integer)$, so: $2^{64} = 10^{19.26} \ll 10^{49}$
And similar computation for x64 $max(integer)$, $2^{64} \cdot ln(2^{64}) = 10^{19.26} \cdot 44.4 = 10^{19.26} \cdot 10^{1.64} = 10^{19.26+1.64} > 10^{19.26}$

However looking at the $log(n)$ part of $n \cdot log(n)$, the value 1.64 from the x64 calculation is very close to 1.96 from the chart scale, while the n part of $n \cdot log(n)$ is significantly different: $10^{19.26} \ll 10^{49}$, it means that with larger n value $log(n)$ part behave more like a constant, and we can simplify that of formula as constant in Big O notation.

Or simply saying, let's say we have some supercomputer that can calculate all possible permutations of $10^{19.26}$ in one hour, and because of the logarithm part same supercomputer will take a few hours more to calculate 10^{49}, however, due to n part, it will take all time of our solar system to calculate 10^{49} permutations, or even maybe all time of the universe.

so back to formula (6)

$$lim_{n \to \infty} O(n^{1+log(\sqrt[log(n)]{log(n)})}) = lim_{n \to \infty} O(n^{1+log(1+\epsilon')}) = lim_{n \to \infty} O(n^{1+\epsilon}),$$

to be fair it should be noted that: $lim_{n \to \infty} n^{\epsilon} = \infty$ and $n \ll n^{1+\epsilon}$ even though $\epsilon \to +0$, however $n^{\epsilon} \to const(n)$, for $n \to \infty$, and because it is inside Big O that constant part can be simplified, proving in next section.

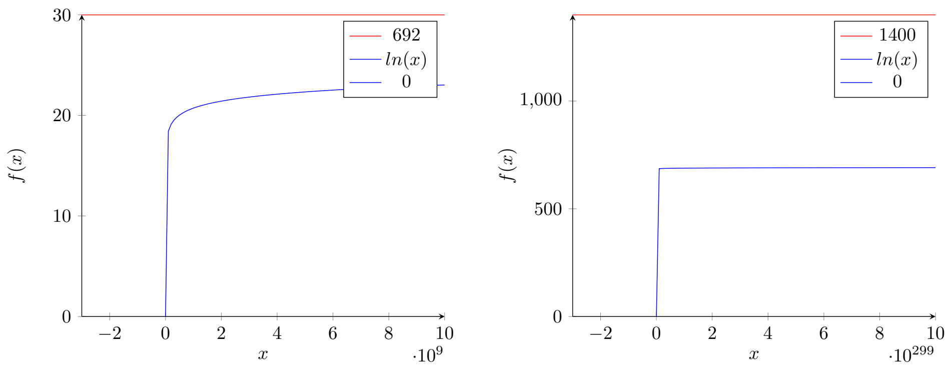
so we can consider that:

$\boxed{lim_{n\to\infty}O(n\cdot log(n)) = lim_{n\to\infty}O(const(n)\cdot n) = O(n), \text{ for any } n\to\infty}$ - formula (7)

7 Proving $n^{\epsilon} = log(n) \to const$, for $n \to \infty$

Let's checking what's happening with derivative of $log(n)$, for $n \to \infty$
$lim_{n\to\infty}\frac{d(log(n))}{dn} = lim_{n\to\infty}\frac{\frac{1}{n}\cdot dn}{dn} = lim_{n\to\infty}\frac{1}{n} = 0$

derivative equal to 0 means that the tangent line is almost horizontal, so the function itself becomes a constant, which we can observe on the plot, with a different scale:



By definition of integral: $\int d(f(x)+c_1) = f(x)+c_2$, so let's repeat same operation with $log(n)$ function, while $n\to\infty$

$lim_{n\to\infty}(log(n)+c_1) = lim_{n\to\infty}\int d(log(n)+c_1) = lim_{n\to\infty}\int\frac{1}{n}dn = \int 0dn = c_2$, therefore
$lim_{n\to\infty}(log(n)+c_1) = c_2$, therefore $lim_{n\to\infty}log(n) = c_2 - c_1$, so

$\boxed{lim_{n\to\infty}log(n) = const(n) \ll n}$

8 Conclusions

The conclusion that can be made from formulas (2) and formula (7) is that, while we consider small values or scaling complexity to infinite, it is either
small values: $O(n\cdot log(n)) = O(c\cdot n) = O(n), c = const$,
or
scaling to infinity: $lim_{n\to\infty}O(n\cdot log(n)) = lim_{n\to\infty}O(n^{1+\epsilon}) = O(const\cdot n) =$

$O(n), \epsilon \to +0$

Also considering that $O(f(n))$, is defined for constantly growing argument $n \to \infty$, using a limit on top of this function, like $lim_{n\to\infty}O(f(n))$ will still provide a precise approximation, however, it simplifies computation complexity analysis. Also, this article shows an example, of how it can be applied for future growth of computation power scale

It is more representable to show some subset of widely used complexity using polynomial power because the difference between $n \cdot log(n)$ and n which is widely considered, however with ϵ can show its insignificance and reasonably be excluded from complexity analysis for most scenarios.

Considering $\epsilon \to +0$, table bellow

Mathematically accurate Big O notation $O(f(n))$	Big O notation with limit $lim_{n\to\infty}O(f(n))$	Polynomial notation $O(f(n))$
$O(1)$	$O(1)$	$O(n^0)$
$O(log(n))$	$O(1)$	$O(n^{0+\epsilon})$
$O(n)$	$O(n)$	$O(n^1)$
$O(n \cdot log(n))$	$O(n)$	$O(n^{1+\epsilon})$
$O(n^2)$	$O(n^2)$	$O(n^2)$
$O(n^3)$	$O(n^3)$	$O(n^3)$
$O(n^k)$	$O(n^k)$	$O(n^k)$
$O(2^n)$	$O(n^n)$	$O(n^{n^{1-\epsilon}})$
$O(n^n)$	$O(n^n)$	$O(n^n)$

$O(2^n) = O(n^{log_n 2^n}) = O(n^{n \cdot log_n 2}) = O(n^{n \cdot log^{-1}(n)}) = O(n^{n^{1-\epsilon}})$, where $\epsilon \to +0$

$log^{-1}(n) = n^{log_n(log^{-1}(n))} = n^{-log_n(log(n))} = n^{-\frac{log(log(n))}{log(n)}} = n^{-log(\sqrt[log(n)]{log(n)})} = n^{-log(1+\epsilon)} = n^{-\epsilon}$, while $n \to \infty, \epsilon \to +0$

9 Addendum: Sub-zero field

The remarkable observation of the polynomial table above is a sub-zero field.

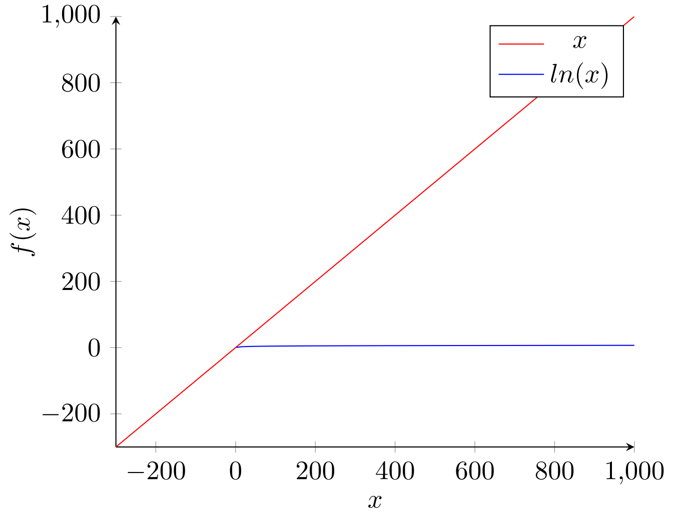


Figure 1: Linear function with logarithmic

If we review n and $log(n)$ functions separately, both of them monotonically grow to infinity separately:
$lim_{n\to\infty} n = \infty$
$lim_{n\to\infty} log(n) = \infty$

which is fare, however, if we put them on the same scale defined by a linear function, like on the chart Figure 1, will see that $log(n)$ is not just a constant, but it somewhere around zero most of the time on that scale. From Cantor's theory about Aleph number[5], we could know that some ordinals representing a number of elements of infinity sets can be somehow different. And on that scale, we can say that while $lim_{n\to\infty} n$ is infinity, the $lim_{n\to\infty} log(n)$ is also another infinity, however, logarithmic infinity is some sort of zero infinity comparing to n infinity.

So we can observe the same thing here, that linear x is some sort of zero infinity compared to x^2 infinity. However from the previous figure 1 already concluded, that logarithms infinity is smaller than linear infinity, while linear infinity is zero on that square infinity scale, so on that square scale logarithmics' infinity is even smaller than zero but not less, and it exists somewhere in the sub-zero field.

There is exist a subset of all possible infinities different from each other, however, they usually do not differentiate, Cantor's theorem is the closest approach

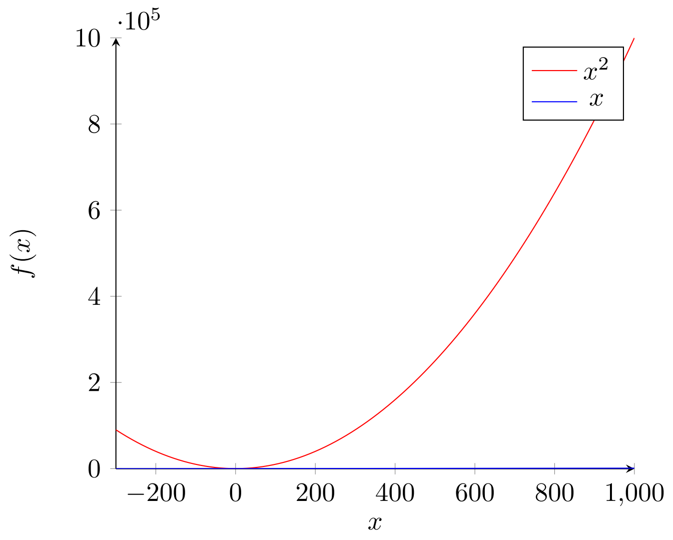


Figure 2: Square function with linear function

to differentiating infinities since it defines cardinality for infinite sets, but still, it is not applicable to compare different possible infinities [5]. An addendum to the main part of the article will show that Polynomial power is also useful to compare infinities. There is a certain similarity between Cantor's theorem and polynomial notation to compare different kinds of infinity and computation complexity, Cantor's shows the most significant difference between sets when it has a different variation of the power function $\aleph_1 = 2^{\aleph_0}$. And this example of comparing natural and integer sets $c_1 \cdot \aleph_0 + c_2 = \aleph_0$ is a perfect example of Big O transformation [6]. However, the granularity of possible infinities is denser than possible values of $\aleph$, since $\aleph$ is used as a power argument for exponential growth, and infinities are just different power arguments.

Returning to polynomial notation, this sub-zero field can be defined more formally. Let's say assume that the n function is some conditional zero thresholds like in Figure 2, transforming from n infinity into $log(n)$ requires applying a negative infinity, which is a sub-zero field. Doing transformation
from $n \sim n^1$
into $log(n) \sim n^\epsilon, \epsilon \to 0$, for any $n \to \infty$

So let's define our transformation, from n into log as $t(x)$
$t(n^1) = n^\epsilon, \epsilon \to 0$
$t(n^1) = n^\epsilon = n^{1-(1-\epsilon)} = n \cdot n^{-(1-\epsilon)} = n^1 \cdot n^{-k}$, considering that $\epsilon < 1$, the $k > 0$ so,

So to transform from n into $log(n)$ needs to multiply n to n^{-k}, which is techni-

cally just the sum of negative polynomial power, and particularly n^{-k} so that's a place where sub-zero fields of positive infinities exist.
If $lim_{n\to+\infty,\epsilon\to+0}(n^{-\epsilon}) = lim_{n\to+\infty}log^{-1}(n) = \frac{1}{const(n)} = +0$, and can assume that it is identical to plus zero, the assumption is that any function greater than log^{-1} will be no more the plus zero and will be just some very small constant, there is certain dualism in logarithm function on large scales, it is infinity and constant at the same, taking 1 and dividing to constant it will make a constant, that not a plus zero anymore, however logarithm still not a constant, it is constant and infinity at the same time, so this where comes the assumption about identity to plus zero. And logarithm is the only known function with such properties, however, the tricky thing is that $lim_{n\to+\infty}log^{-1}(log(n))$ is even closer to plus zero identity, going further $log^{-1}(log(...log(n)...))$ - is there something closer to +0 than the logarithm of logarithms, that's a question.

So comparing function $lim_{n\to\infty}(n^{-1})$ is less than $lim_{n\to+\infty}log^{-1}(n)$, so it is somewhere in the sub-zero field, but not less than the zero,

Practical outcome - if we try to multiply $\infty \cdot 0$ - the result is undefined, however, if we could define the power factor of zero and infinity, which is simply polynomial power, this expression became computable, so example:

$(lim_{n\to\infty}50n^2 + 3n) \cdot (lim_{n\to\infty}\frac{10}{n} + \frac{30}{n^2}) = (5 \cdot \infty^2) \cdot (10 \cdot \infty^{-1}) = 50 \cdot \infty^1$
$lim_{n\to\infty}log(n) = \infty^\epsilon, \epsilon \sim +0$

References

[1] *https://en.wikipedia.org/wiki/Time_complexity*, Wikipedia

[2] *https://www.doc.ic.ac.uk/~eedwards/compsys/memory/index.html*, Computer Memory

[3] *https://en.wikipedia.org/wiki/Processor_register*, Computer Memory

[4] Thomas H. Cormen(2009) *Introduction to Algorithms 3rd ed*, The MIT Press

[5] *https://en.wikipedia.org/wiki/Aleph_number*, Wikipedia

[6] *http://www.cwladis.com/math100/Lecture5Sets.htm*, The Cardinality of the Set of Integers