UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

PROIECT DE DIPLOMĂ

Metrici de evaluare a performanțelor și complexității pentru programele de calculatoare

O noua abordare asupra modalitatilor traditionale de masurare a complexitatii algoritmilor si estimare a timpilor de rulare in raport cu arhitectura aleasa, in contextul infrastructurii dinamice

Rares Folea

Coordonator științific:

Prof. Emil-Ioan Slusanschi

BUCUREȘTI
2020

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

DIPLOMA PROJECT

Metrics for evaluating the performance and complexity of computer programs

A new approach to the traditional ways of measuring the complexity of algorithms and estimating running times given the chosen architecture, in the context of dynamic infrastructure

Rares Folea

Thesis advisor:

Prof. Emil-Ioan Slusanschi

BUCHAREST

2020

Contents

SINOPSIS

TODO

ABSTRACT

TODO

Chapter 1

Classical Computational Complexity Calculus

1.1 Introduction and Motivation

This chapter will present the traditional methodology of calculus in the field of algorithm's computational complexity. The complexity will be expressed as a function $f : \mathbb{N} \longrightarrow \mathbb{R}$, where the function is characterized by the size of the input, while the evaluated value $f(n)$, for a given input size n, represents the amount of resources needed in order to compute the result. While the most often analyzed resource is time, expressed in primitive computational operations, the mathematical model is self-reliant for others natures of resources, including space reasoning or hybrid metrics.

All the results of this chapter are well known in the literature and they represent the reference standard in calculating algorithm complexity in Computer Science.

Because of the difficulty of having a rigorous calculus of the complexity, an asymptotic computation approach for an algorithm's complexity offers a valuable insight about the real computational cost without requiring a precisely, flawless calculus.

1.2 Family of Bachmann–Landau notations

The following notations and names will be used for describing the asymptotic behavior of a algorithm's complexity characterized by a function, $f : \mathbb{N} \longrightarrow \mathbb{R}$.
We define the set of all complexity calculus $\mathcal{F} = \{f : \mathbb{N} \longrightarrow \mathbb{R}\}$
Assume that $n, n_0 \in \mathbb{N}$. Also, we will consider an arbitrary complexity function $g \in \mathcal{F}$.

Definition 1.2.1. *Assume that two continuous and derivable functions $v, w : \mathbb{R} \longrightarrow \mathbb{R}$ are*

considered to be similar in an **asymptotic analysis** *iff:*

$$\lim_{x\to\infty} \frac{v(x)}{w(x)} = c \in (-\infty, 0) \cup (0, \infty)$$

Lemma 1.2.1. *Consider that if we analyses asymptotic behavior of functions defined over* $\mathbb{N}$ *the functions* $v, w : \mathbb{N} \longrightarrow \mathbb{R}$ *are similar iff there exists a function* $r : \mathbb{N} \longrightarrow \mathbb{R}$*, with* $r(x) = \frac{v(x)}{w(x)}\ \forall\ x \in \mathbb{N}$*, such that*

$$\lim_{x\to\infty} r(x) = \lim_{x\to\infty} \frac{v(x)}{w(x)} = C \in (-\infty, 0) \cup (0, \infty)$$

The following function classes can be therefore defined for any function $g : \mathbb{N} \longrightarrow \mathbb{R}$ that describes a convergent sequence or a divergent sequence with a limit that tends to infinity:

Definition 1.2.2. Big Theta*: This set defines the group of mathematical functions similar in magnitude with* $g(n)$ *in the study of asymptotic behavior. A set-based description of this group can be expressed as:*

$$\Theta(g(n)) = \{f \in \mathcal{F} \mid \exists c_1, c_2 \in \mathbb{R}^*_+, \exists n_0 \in \mathbb{N}^*\ s.t.\ \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\ ,\ \forall n \geq n_0\}$$

Lemma 1.2.2. *Another useful method of establishing a Big-Theta acceptance, if the composed sequence defined by the ratio of the two functions exists and also* $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ *exists, is by applying the sufficient condition:*

$$f \in \Theta(g(n)) \ \Leftrightarrow\ \lim_{x\to\infty} \frac{f(x)}{g(x)} = C \in (-\infty, 0) \cup (0, \infty)$$

Remark that both f, g can define divergent sequence with limits that tend to infinity, while the sequence $r(x) = \frac{f(x)}{g(x)}\ \forall\ x \in \mathbb{N}$ can be convergent and can have a nonzero limit.
Iff $r(x) = \frac{f(x)}{g(x)}$ exists, a sufficient condition can be defined using the formal limit definition for an arbitrary sequence if the following condition holds:

$$\exists\ C \in \mathbb{R},\ \ C \neq 0\ s.t.\ \forall \epsilon > 0, \exists n_0 \in \mathbb{N}\ \ s.t.\ \forall n \geq n_0,\ \ |r(n) - C| < \epsilon$$

Thus, if $\exists\ C \in \mathbb{R}, C \neq 0\ \Rightarrow\ f \in \Theta(g(n))$

Definition 1.2.3. Big O*: This set defines the group of mathematical functions that are known to have a similar or lower asymptotic performance in comparison with* $g(n)$*. The set of such functions is defined as it follows:*

$$\mathcal{O}(g(n)) = \{f \in \mathcal{F} \mid \exists c \in \mathbb{R}^*_+, \exists n_0 \in \mathbb{N}^*\ s.t.\ f(n) \leq c \cdot g(n),\ \forall n \geq n_0\}$$

Lemma 1.2.3. *In order to establish a Big-O acceptance, the sufficient condition is that the* $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ *exists and it is finite.*

$$f \in \mathcal{O}(g(n)) \ \Leftrightarrow\ \lim_{x\to\infty} \frac{f(x)}{g(x)} = C \in (-\infty, \infty)$$

Definition 1.2.4. Big Omega: *This set defines the group of mathematical functions that are known to have a similar or higher asymptotic performance in comparison with $g(n)$. The set of all function is defined as:*

$$\Omega(g(n)) = \{f \in \mathcal{F} \mid \exists c \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^* \; s.t. \; f(n) \geq c \cdot g(n), \; \forall n \geq n_0\}$$

Lemma 1.2.4. *In order to establish a Big-Omega acceptance, a sufficient condition is one of the following:*

(i)

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = -\infty \Rightarrow f \in \Omega(g(n))$$

(ii)

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty \Rightarrow f \in \Omega(g(n))$$

(iii)

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = C \in (-\infty, 0) \cup (0, \infty) \Rightarrow f \in \Omega(g(n))$$

Definition 1.2.5. Small O: *This set defines the group of mathematical functions that are known to have a humble asymptotic performance in comparison with $g(n)$. The set of such functions is defined as it follows:*

$$o(g(n)) = \{f \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^* \; s.t. \; f(n) < c \cdot g(n), \; \forall n \geq n_0\}$$

Lemma 1.2.5. *In order to establish a Small-O acceptance, the sufficient condition is that the $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ exists and it is finite.*

$$f \in o(g(n)) \;\Leftrightarrow\; \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$$

Definition 1.2.6. Small Omega: *This set defines the group of mathematical functions that are known to have a commanding asymptotic performance in comparison with $g(n)$. The set of such functions is defined as it follows:*

$$\omega(g(n)) = \{f \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^* \; s.t. \; f(n) > c \cdot g(n), \; \forall n \geq n_0\}$$

Lemma 1.2.6. *In order to establish a Small-Omega acceptance, the sufficient condition is that the $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ exists and one of the following occurs:*

(i)

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = +\infty \Rightarrow f \in \omega(g(n))$$

(ii)

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = -\infty \Rightarrow f \in \omega(g(n))$$

1.3 Common properties

We will study few aspects [**?**]; with impact on defining equivalence relations over classes defined in *Bachmann–Landau* notations.

Theorem 1.3.1. *The reflexivity property holds for Big Theta, Big O and Big Omega notations.*
$f \in \Theta(f(n))$
$f \in \mathcal{O}(f(n))$
$f \in \Omega(f(n))$

Remark. *The reflexivity property does not hold for Small O and Small Omega notations*
$f \notin \theta(f(n))$
$f \notin \omega(f(n))$

Theorem 1.3.2. *The transitivity property holds for Bachmann–Landau notations.*
$f \in \Theta(g(n)), g \in \Theta(h(n)) \Rightarrow f \in \Theta(h(n))$
$f \in \mathcal{O}(g(n)), g \in \mathcal{O}(h(n)) \Rightarrow f \in \mathcal{O}(h(n))$
$f \in \Omega(g(n)), g \in \Omega(h(n)) \Rightarrow f \in \Omega(h(n))$
$f \in o(g(n)), g \in o(h(n)) \ \Rightarrow f \in o(h(n))$
$f \in \omega(g(n)), g \in \omega(h(n)) \Rightarrow f \in \omega(h(n))$

Theorem 1.3.3. *The symmetry property holds for Big Theta notation.*
$f \in \Theta(g(n)) \Rightarrow g \in \Theta(f(n))$

Theorem 1.3.4. *The transpose symmetry property for Bachmann–Landau notations.*
$f \in \mathcal{O}(g(n)) \Leftrightarrow g \in \Omega(f(n))$
$f \in o(g(n)) \Leftrightarrow g \in \omega(f(n))$

Theorem 1.3.5. *The projection property for Big Theta, Big O and Big Omega notations.*
$f \in \Theta(g(n)) \Leftrightarrow f \in \mathcal{O}(g(n)), f \in \Omega(g(n))$

1.4 Addition properties

Calculus in $Bachmann-Landau$ notations (including Big-O arithmetic) is extremely powerful and when it comes to addition operations, it is straightforward. The following relations hold for any correctly defined functions $f, g, h : \mathbb{N} \longrightarrow \mathbb{R}$:

Lemma 1.4.1. *Addition in* **Big Theta**

If $h(n) = f(n) + g(n)\ \forall n \in \mathbb{N}$ *and* $\exists n_0 \in \mathbb{N},\ s.t.\ \forall n \in \mathbb{N}\ \Rightarrow f(n) \leq g(n)$, *then:*

$$h \in \Theta(g)$$

Remark. *For each theorem, we will expose a corollary using a relax notation (consider that by any $Bachmann - Landau$ class notation, we denote an arbitrary function part of the class), some relations that can be settled.*

Corollary 1.4.1.1. *Addition in* **Big Theta** *using the relaxed notation:*

$$\Theta(f) + \Theta(g) = \Theta(g)$$

Lemma 1.4.2. *Addition in* **Big O***:*

If $h(n) = f(n) + g(n)\ \forall n \in \mathbb{N}$ *and* $\exists n_0 \in \mathbb{N},\ s.t.\ \forall n \in \mathbb{N}\ \Rightarrow f(n) \leq g(n)$*, then:*

$$h \in \mathcal{O}(g)$$

Corollary 1.4.2.1. *Addition in* **Big O** *using the relaxed notation:*

$$\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(g)$$

Lemma 1.4.3. *Addition in* **Big Omega***:*

If $h(n) = f(n) + g(n)\ \forall n \in \mathbb{N}$ *and* $\exists n_0 \in \mathbb{N},\ s.t.\ \forall n \in \mathbb{N}\ \Rightarrow f(n) \leq g(n)$*, then:*

$$h \in \Omega(g)$$

Corollary 1.4.3.1. *Addition in* **Big Omega** *using the relaxed notation:*

$$\Omega(f) + \Omega(g) = \Omega(g)$$

Lemma 1.4.4. *Addition in* **Small O***:*

If $h(n) = f(n) + g(n)\ \forall n \in \mathbb{N}$ *and* $\exists n_0 \in \mathbb{N},\ s.t.\ \forall n \in \mathbb{N}\ \Rightarrow f(n) \leq g(n)$*, then:*

$$h \in o(g)$$

Corollary 1.4.4.1. *Addition in* **Small O** *using the relaxed notation:*

$$o(f) + o(g) = o(g)$$

Lemma 1.4.5. *Addition in* **Small Omega***:*

If $h(n) = f(n) + g(n)\ \forall n \in \mathbb{N}$ *and* $\exists n_0 \in \mathbb{N},\ s.t.\ \forall n \in \mathbb{N}\ \Rightarrow f(n) \leq g(n)$*, then:*

$$h \in \omega(g)$$

Corollary 1.4.5.1. *Addition in* **Small Omega** *using the relaxed notation:*

$$\omega(f) + \omega(g) = \omega(g)$$

Chapter 2

A refined complexity calculus model: r-Complexity

2.1 Introduction and Motivation

This chapter will present a new approach in the field of algorithm's computational complexity. Similar to the conventional asymptotic notations proposed in the literature, *rComplexity* will be expressed as a function $f : \mathbb{N} \longrightarrow \mathbb{R}$, where the function is characterized by the size of the input, while the evaluated value $f(n)$, for a given input size n, represents the amount of resources needed in order to compute the result.

This new calculus model aims to produce new asymptotic notations that offer better complexity feedback for similar algorithms, providing subtle insights even for algorithms that are part of the same conventional complexity class $\Theta(g(n))$ denoted by an arbitrary function $g : \mathbb{N} \longrightarrow \mathbb{R}$, in the definition of *Bachmann–Landau* notations. The additional information contained by *rComplexity* classes consists of the fine-granularity obtained by the model based on a refined clustering strategy for functions that used to belong to the same *Bachmann–Landau* group in different complexity classes, established on asymptotic constant analysis.

The classical complexity calculus model is a long-established, verified metric of evaluating an algorithm's performance and it is a valuable measurement in estimating feasibility of computing a considerable algorithm.

However, the model has few shortfalls in making discrepancy between similar algorithms with two similar complexity functions, $v, w : \mathbb{N} \longrightarrow \mathbb{R}$, such that $v(n), w(n) \in \Theta(g(n))$. In order to highlight the lack of distinction, suppose two algorithms $Alg1$ and $Alg2$ that solve exactly the same problem, with the following complexity functions: $f_1, f_2 : \mathbb{N} \longrightarrow \mathbb{R}$ where $f_1 = x \cdot f_2$, with $x \in \mathbb{R}_+,\ x > 1$. If

$$f_2 \in \Theta(g(n)) \Rightarrow \exists c_1, c_2 \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^*\ s.t.\ \ c_1 \cdot g(n) \leq f_2(n) \leq c_2 \cdot g(n)\ ,\ \forall n \geq n_0$$

Therefore, $f_1 \in \Theta(g(n))$, as there exists $c_1^{'}, c_2^{'} \in \mathbb{R}_+^*, n_0^{'} \in \mathbb{N}^*$ such that $c_1^{'} = x \cdot c_1$, $c_2^{'} = x \cdot c_2$

and $n_0^{'} = n_0$ and $\forall n \geq n_0^{'} : c_1^{'} \cdot g(n) \leq f_1(n) \leq c_2^{'} \cdot g(n)$.

Remark that even if $f_1 > f_2$, both complexity functions are part of the same complexity class. This observation implies that two algorithms, whose complexity functions can differ by a constant, even as high as 2020^{2020}, are part of the same complexity class, even if the actual run-time might differ by over 6676 orders of magnitude.

Remark. *The discussion of how big numbers really are is fruitless and worthless. In this discussions the only bound can be expressed by the idiom sky is the limit, as numbers such as* $2020 \uparrow\uparrow 2020$ *defer by colossal orders of magnitude. Knuth's uparrow notation is a method of notation for very large integers, introduced by Donald Knuth in 1976. [**?**] The idea is based on the fact that multiplication can be viewed as iterated addition and exponentiation as iterated multiplication.*

For comparison, only a 30 magnitude order between the two complexity functions $f_1 = 10^{30} \cdot f_2$, signify that if for a given input n, if $Alg2$ ends execution in 1 *attosecond* (10^{-9} part of a nanosecond), then $Alg1$ is expected to end execution in about 3 *millenniums*. Despite of the colossal difference in time, classical complexity model is not perceptive between algorithms whose complexity functions differs only through constants.

rComplexity calculus aims to clarify this issue by taking into deep analysis the preeminent constants that can state major improvements in an algorithm's complexity and can have tremendous effect over total execution time.

2.2 Adjusting the Bachmann–Landau notations for rComplexity Calculus

The following notations and names will be used for describing the asymptotic behavior of a algorithm's complexity characterized by a function, $f : \mathbb{N} \longrightarrow \mathbb{R}$.
We define the set of all complexity calculus $\mathcal{F} = \{f : \mathbb{N} \longrightarrow \mathbb{R}\}$
Assume that $n, n_0 \in \mathbb{N}$. Also, we will consider an arbitrary complexity function $g \in \mathcal{F}$. Acknowledge the following notations $\forall r \neq 0$:

Definition 2.2.1. Big r-Theta*: This set defines the group of mathematical functions similar in magnitude with* $g(n)$ *in the study of asymptotic behavior. A set-based description of this group can be expressed as:*

$$\Theta_r(g(n)) = \{f \in \mathcal{F} \mid \forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < r < c_2, \exists n_0 \in \mathbb{N}^* \\ s.t. \ \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ , \ \forall n \geq n_0\}$$

Definition 2.2.2. Big r-O*: This set defines the group of mathematical functions that are known to have a similar or lower asymptotic performance in comparison with* $g(n)$*. The set*

of such functions is defined as it follows:

$$\mathcal{O}_r(g(n)) = \{f \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^* \ s.t. \ r < c, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) \leq c \cdot g(n), \ \forall n \geq n_0\}$$

Definition 2.2.3. Big r-Omega: *This set defines the group of mathematical functions that are known to have a similar or higher asymptotic performance in comparison with $g(n)$. The set of all function is defined as:*

$$\Omega_r(g(n)) = \{f \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^* \ s.t. \ c < r, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) \geq c \cdot g(n), \ \forall n \geq n_0\}$$

Definition 2.2.4. Small r-O: *This set defines the group of mathematical functions that are known to have a humble asymptotic performance in comparison with $g(n)$. The set of such functions is defined as it follows:*

$$o_r(g(n)) = \{f \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) < c \cdot g(n), \ \forall n \geq n_0\}$$

Remark. *This set is defined for symmetry of the model and it is equal with the set defined by* **Small O** *notation in Bachmann–Landau notations, as the definition is independent on r.*

Definition 2.2.5. Small r-Omega: *This set defines the group of mathematical functions that are known to have a commanding asymptotic performance in comparison with $g(n)$. The set of such functions is defined as it follows:*

$$\omega_r(g(n)) = \{f \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) > c \cdot g(n), \ \forall n \geq n_0\}$$

Remark. *This set is defined for symmetry of the model and it is equal with the set defined by* **Small Omega** *notation in Bachmann–Landau notations, as the definition is independent on r.*

2.3 Asymptotic Analysis

Calculus in $rComplexity$ can be performed either using limits of sequences or limits of functions. Consider any two complexity functions $f, g : \mathbb{N}_+ \longrightarrow \mathbb{R}_+$.

Theorem 2.3.1. *Admittance of a function f in* **Big r-Theta** *class defined by a function g:*

$$f \in \Theta_r(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = r$$

Theorem 2.3.2. *Admittance of a function f in* **Big r-O** *class defined by a function g:*

$$f \in \mathcal{O}_r(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = l, \ l \in [0, r]$$

Theorem 2.3.3. *Admittance of a function f in* **Big r-Omega** *class defined by a function g:*

$$f \in \Omega_r(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = l, \ l \in [r, \infty)$$

Theorem 2.3.4. *Admittance of a function f in* **Small r-O** *class defined by a function g:*

$$f \in o_r(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Theorem 2.3.5. *Admittance of a function f in* **Small r-Omega** *class defined by a function g:*

$$f \in \omega_r(g(n)) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

2.4 Common properties

This chapter will present new implications in terms of *reflexivity, transitivity, symmetry and projections* compared with conventional *Bachmann–Landau notations*.

Theorem 2.4.1. *Reflexivity in $rComplexity$: - Big r-Theta notation*

$$f \in \Theta_r\left(\frac{1}{r} \cdot f(n)\right) \ \forall r \neq 0$$

Proof. Using the definition of $\Theta_r(f(n))$

$$\Theta_r(f(n)) = \{f' \in \mathcal{F} \mid \forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < r < c_2, \exists n_0 \in \mathbb{N}^* \\ s.t. \ \ c_1 \cdot f(n) \leq f'(n) \leq c_2 \cdot f(n) \ , \ \forall n \geq n_0\}$$

Using substitution $f(n) \longleftarrow \frac{1}{r} \cdot f(n)$

$$\Theta_r\left(\frac{1}{r} \cdot f(n)\right) = \{f' \in \mathcal{F} \mid \forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < r < c_2, \exists n_0 \in \mathbb{N}^* \\ s.t. \ \frac{1}{r} \cdot \ c_1 \cdot f(n) \leq f'(n) \leq \frac{1}{r} \cdot c_2 \cdot f(n) \ , \ \forall n \geq n_0\}$$

By choosing $c_1' = \frac{1}{r} \cdot c_1, c_2' = \frac{1}{r} \cdot c_2$

$$\Theta_r\left(\frac{1}{r} \cdot f(n)\right) = \{f' \in \mathcal{F} \mid \forall c_1', c_2' \in \mathbb{R}_+^* \ s.t. c_1' < 1 < c_2', \exists n_0 \in \mathbb{N}^* \\ s.t. \ \ c_1' \cdot f(n) \leq f'(n) \leq c_2' \cdot f(n) \ , \ \forall n \geq n_0\}$$

For any $x \in \mathbb{R}^*$, $\forall c_1, c_2 \ \ s.t. c_1 \leq 1 \leq c_2$, we have $x \cdot c_1 \leq x \leq x \cdot c_2$.
Thus, if $f : \mathbb{N} \longrightarrow \mathbb{R}$, $\forall c_1, c_2 \ \ s.t. c_1 \leq 1 \leq c_2$, we have $f(n) \cdot c_1 \leq f(n) \leq f(n) \cdot c_2 \ \ \forall n \in \mathbb{N}^*$ or $f(n) \cdot c_1 \geq f(n) \geq f(n) \cdot c_2 \ \ \forall n \in \mathbb{N}^*$

Therefore:

$$\forall c_1', c_2' \in \mathbb{R}_+^* \ s.t. c_1' < 1 < c_2', \exists n_0 = 1 s.t. \ \ c_1' \cdot f(n) \leq f(n) \leq c_2' \cdot f(n) \ , \ \forall n \geq n_0 = 1 \Rightarrow$$

$$f \in \Theta_r\left(\frac{1}{r} \cdot f(n)\right) \ \forall r \neq 0$$

■

Theorem 2.4.2. *Reflexivity in $rComplexity$: - Big r-O notation*

$$f \in \mathcal{O}_r\left(x \cdot f(n)\right) \ \forall x \geq \frac{1}{r}$$

Proof. : Using the definition of $\mathcal{O}_r(f(n))$

$$\mathcal{O}_r(f(n)) = \{f' \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^* \ s.t. \ r < c, \exists n_0 \in \mathbb{N}^* \ s.t. \ f'(n) \leq c \cdot f(n), \ \forall n \geq n_0\}$$

Using substitution $f(n) \longleftarrow x \cdot f(n), \forall x \geq \frac{1}{r}, \forall r \neq 0$

$$\mathcal{O}_r(x \cdot f(n)) = \{f' \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^* \ s.t. \ r < c, \exists n_0 \in \mathbb{N}^* \ s.t. \ f'(n) \leq c \cdot x \cdot f(n), \ \forall n \geq n_0\}$$

If $r < c$ and $x \geq \frac{1}{r}$, then $c \cdot x \geq 1, \forall r, c, x \quad r \neq 0$ Thus, if $f : \mathbb{N} \longrightarrow \mathbb{R}$, we have $f(n) \leq 1 \cdot f(n) \leq c \cdot x \cdot f(n) \ \ \forall n \in \mathbb{N}^*$.

Therefore:

$$\forall x \geq \frac{1}{r}, \ \forall c \in \mathbb{R}_+^* \ s.t. \ r < c, \exists n_0 = 1 s.t. \ \ f(n) \leq c \cdot x \cdot f(n) \ \ \forall n \in \mathbb{N}^*, \ \forall n \geq n_0 = 1 \Rightarrow$$

$$f \in \mathcal{O}_r\left(x \cdot f(n)\right) \ \forall x \geq \frac{1}{r}$$

■

Theorem 2.4.3. *Reflexivity in $rComplexity$: - Big r-Omega notation*

$$f \in \Omega_r\left(x \cdot f(n)\right) \ \forall x \leq \frac{1}{r}$$

Proof. Using the definition of $\Omega_r(f(n))$

$$\Omega_r(f(n)) = \{f' \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^* \ s.t. \ c < r, \exists n_0 \in \mathbb{N}^* \ s.t. \ f'(n) \geq c \cdot f(n), \ \forall n \geq n_0\}$$

Using substitution $f(n) \longleftarrow x \cdot f(n), \forall x \leq \frac{1}{r}, \forall r \neq 0$

$$\Omega_r(x \cdot f(n)) = \{f' \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^* \ s.t. \ c < r, \exists n_0 \in \mathbb{N}^* \ s.t. \ f'(n) \geq x \cdot c \cdot f(n), \ \forall n \geq n_0\}$$

If $r > c$ and $x \leq \frac{1}{r}$, then $c \cdot x \leq 1, \forall r, c, x \quad r \neq 0$

Therefore:

$$\forall x \leq \frac{1}{r}, \ \forall c \in \mathbb{R}_+^* \ s.t. \ c < r, \exists n_0 = 1 s.t. \ \ f(n) \geq c \cdot x \cdot f(n) \ \ \forall n \in \mathbb{N}^*, \ \forall n \geq n_0 = 1 \Rightarrow$$

$$f \in \Omega_r\left(x \cdot f(n)\right) \ \forall x \leq \frac{1}{r}$$

■

Remark. *Reflexivity does not hold in $rComplexity$ for small r-O notation.*

$$f \notin o_r(f(n))$$

The reflexivity does not hold for Small r-o and Small r-Omega, as these two sets are equal with the classical sets defined in Bachmann–Landau notations

Remark. *Reflexivity does not hold in $rComplexity$ for small r-Omega notation.*

$$f \notin \omega_r(f(n))$$

The reflexivity does not hold for Small r-o and Small r-Omega, as these two sets are equal with the classical sets defined in Bachmann–Landau notations

Theorem 2.4.4. *Transitivity in $rComplexity$ - Big r-Theta notation:*
$f \in \Theta_r(g(n)), g \in \Theta_{r'}(h(n)) \Rightarrow f \in \Theta_{r \cdot r'}(h(n))$

Proof. If $f \in \Theta_r(g(n)) \Rightarrow \forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < r < c_2, \exists n_0 \in \mathbb{N}^*$
$s.t. \ \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ , \ \forall n \geq n_0$
If $g \in \Theta_{r'}(h(n)) \Rightarrow \forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < r' < c_2, \exists n_0' \in \mathbb{N}^*$
$s.t. \ \ c_1 \cdot h(n) \leq g(n) \leq c_2 \cdot h(n) \ , \ \forall n \geq n_0'$
Thus, if $f \in \Theta_r(g(n))$ and $g \in \Theta_{r'}(h(n))$
$\Rightarrow \forall c_1, c_2, c_1', c_2' \in \mathbb{R}_+^* \ s.t. c_1 < r < c_2, c_1' < r' < c_2', \exists n_0'' = max(n_0, n_0') \in \mathbb{N}^*$
$s.t. \ \ c_1' \cdot c_1 \cdot h(n) \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \leq c_2' \cdot c_2 \cdot h(n) \ , \ \forall n \geq n_0''$
Let $c_1'' = c_1 \cdot c_1', c_2'' = c_2 \cdot c_2'$.
Then, $c_1'' < r \cdot r' < c_2''$
Therefore, $\forall c_1'', c_2'' \in \mathbb{R}_+^* \ s.t. c_1'' < r \cdot r' < c_2'' \ \ \exists n_0'' = max(n_0, n_0') \in \mathbb{N}^*$
$s.t. \ \ c_1'' \cdot h(n) \leq f(n) \leq c_2'' \cdot h(n) \ , \ \forall n \geq n_0'' \Rightarrow f \in \Theta_{r \cdot r'}(h(n))$

■

All other Transitivity properties hold as well in $rComplexity$ Calculus. The proof is similar with the above.

Theorem 2.4.5. *Transitivity in $rComplexity$ - Big r-O notation*

$f \in \mathcal{O}_r(g(n)), g \in \mathcal{O}_r(h(n)) \Rightarrow f \in \mathcal{O}_{r \cdot r'}(h(n))$

Theorem 2.4.6. *Transitivity in $rComplexity$ - Big r-Omega notation*

$f \in \Omega_r(g(n)), g \in \Omega_r(h(n)) \Rightarrow f \in \Omega_{r \cdot r'}(h(n))$

Theorem 2.4.7. *Transitivity in $rComplexity$ - Small r-O notation*

$f \in o_r(g(n)), g \in o_{r'}(h(n)) \Rightarrow f \in o_{r \cdot r'}(h(n))$

Theorem 2.4.8. *Transitivity in $rComplexity$ - Small r-Omega notation*

$f \in \omega_r(g(n)), g \in \omega_{r'}(h(n)) \Rightarrow f \in \omega_{r \cdot r'}(h(n))$

Theorem 2.4.9. Symmetry in $rComplexity$:
$f \in \Theta_r(g(n)) \Rightarrow g \in \Theta_{\frac{1}{r}}(f(n))$

Proof. $f \in \Theta_r(g(n)) \Rightarrow \forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < r < c_2, \exists n_0 \in \mathbb{N}^*$
$s.t. \ \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ , \ \forall n \geq n_0$
Using the substitution $c_1' = \dfrac{c_1}{r}, c_2' = \dfrac{c_2}{r} \Rightarrow \forall c_1', c_2' \in \mathbb{R}_+^* \ s.t. c_1 < 1 < c_2, \exists n_0 \in \mathbb{N}^*$
$s.t. \ \ c_1' \cdot g(n) \leq \dfrac{1}{r} \cdot f(n) \leq c_2' \cdot g(n) \ , \ \forall n \geq n_0$
The previous inequality can be re-written as:

$$\begin{cases} g(n) \leq \dfrac{1}{c_1'} \cdot \dfrac{1}{r} \cdot f(n) \\ g(n) \geq \dfrac{1}{c_2'} \cdot \dfrac{1}{r} \cdot f(n) \end{cases}$$

Using notation $c_1'' = \dfrac{1}{c_2'}, c_2'' = \dfrac{1}{c_1'}$ the inequality becomes:
$\forall c_1,'' c_2'' \in \mathbb{R}_+^* \ s.t. c_1'' < r < c_2'', \exists n_0 \in \mathbb{N}^*$

$$c_1'' \cdot \frac{1}{r} \cdot f(n) \leq g(n) \leq c_2' \cdot \frac{1}{r} \cdot f(n) \ \ \forall n \geq n_0$$

Thus, using the definition of $\Theta_r(f(n))$, $f \in \Theta_r(g(n)) \Rightarrow g \in \Theta_{\frac{1}{r}}(f(n))$. ■

Theorem 2.4.10. Transpose symmetry in $rComplexity$:
$f \in \mathcal{O}_r(g(n)) \Leftrightarrow g \in \Omega_{\frac{1}{r}}(f(n))$

Proof. Using the property of $\mathcal{O}_r(f(n))$ and $f \in \mathcal{O}_r(g(n))$:

(i)

$$\mathcal{O}_r(g(n)) \supseteq \{f\}$$

(ii)

$$\forall c \in \mathbb{R}_+^* \ s.t. \ r < c, \exists n_0 \in \mathbb{N}^* \ s.t. \ g(n) \leq c \cdot f(n), \ \forall n \geq n_0$$

We can rewrite the previous relation as following:

$$\forall c \in \mathbb{R}_+^* \ s.t. \ r < c, \exists n_0 \in \mathbb{N}^* \ s.t. \ \frac{1}{c} \cdot g(n) \leq f(n), \ \forall n \geq n_0$$

Substituting the constant $c' = \dfrac{1}{c}$:

$$\forall c' \in \mathbb{R}_+^* \ s.t. \ c' < \frac{1}{r}, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) \geq c' \cdot g(n), \ \forall n \geq n_0$$

Therefore:

$$\Omega_{\frac{1}{r}}(f(n)) \supseteq \{g\}$$

■

Theorem 2.4.11. Transpose symmetry in $rComplexity$: $f \in o_r(g(n)) \Leftrightarrow g \in \omega_{\frac{1}{r}}(f(n))$

Proof. The Transpose symmetry hold for Small *r*-o and Small *r*-Omega, as these two sets are equal with the classical sets defined in *Bachmann–Landau notations* ■

Theorem 2.4.12. Projection in $rComplexity$:
$f \in \Theta_r(g(n)) \Leftrightarrow f \in \mathcal{O}_r(g(n)), f \in \Omega_r(g(n))$

Proof. The proof is straightforward, usind the definitions of **Big r-Theta**, **Big r-O** and **Big r-Omega**.
" $\Rightarrow$ " $f \in \Theta_r(g(n)) \Rightarrow f \in \mathcal{O}_r(g(n)), f \in \Omega_r(g(n))$ Using the definition of $\Theta_r(g(n))$, $f \in \Theta_r(g(n)) \Rightarrow$

$$\forall c_1, c_2 \in \mathbb{R}_+^* \; s.t. c_1 < r < c_2, \exists n_0 \in \mathbb{N}^* s.t. \;\; c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \;,\; \forall n \geq n_0$$

By splitting the inequality:

$$\begin{cases} \forall c_1 \in \mathbb{R}_+^* \; s.t. c_1 < r, \exists n_0 \in \mathbb{N}^* \; s.t. \;\; c_1 \cdot g(n) \leq f(n), \; \forall n \geq n_0 \\ \forall c_2 \in \mathbb{R}_+^* \; s.t. r < c_2, \exists n_0 \in \mathbb{N}^* \; s.t. \; f(n) \leq c_2 \cdot g(n) \;,\; \forall n \geq n_0 \end{cases}$$

Therefore:

$$\begin{cases} f \in \mathcal{O}_r(g(n)) \\ f \in \Omega_r(g(n)) \end{cases}$$

" $\Leftarrow$ " $f \in \mathcal{O}_r(g(n)), f \in \Omega_r(g(n)) \Rightarrow f \in \Theta_r(g(n))$ Usind the definitions of **Big r-O** and **Big r-Omega**:

$$\begin{cases} \forall c_1 \in \mathbb{R}_+^* \; s.t. c_1 < r, \exists n_0 \in \mathbb{N}^* \; s.t. \;\; c_1 \cdot g(n) \leq f(n), \; \forall n \geq n_0 \\ \forall c_2 \in \mathbb{R}_+^* \; s.t. r < c_2, \exists n_0' \in \mathbb{N}^* \; s.t. \; f(n) \leq c_2 \cdot g(n) \;,\; \forall n \geq n_0' \end{cases}$$

By choosing $n_0'' = max(n_0, n_0')$:

$$\begin{cases} \forall c_1 \in \mathbb{R}_+^* \; s.t. c_1 < r, \Rightarrow \;\; c_1 \cdot g(n) \leq f(n), \; \forall n \geq n_0'' \\ \forall c_2 \in \mathbb{R}_+^* \; s.t. r < c_2, \Rightarrow \; f(n) \leq c_2 \cdot g(n) \;,\; \forall n \geq n_0'' \end{cases}$$

By merging the inequalities, we have:

$$\forall c_1, c_2 \in \mathbb{R}_+^* \; s.t. c_1 < r < c_2, \exists n_0'' = max(n_0, n_0' \in \mathbb{N}^* s.t. \; c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \;, \forall n \geq n_0''$$

Therefore $f \in \Theta_r(g(n))$ ■

2.5 Interdependence properties

An interesting property of the **Big r-Theta**, **Big r-O** and **Big r-Omega** classes is the simple technique of conversion between various values fot the rs parameters. The following results arise:

Theorem 2.5.1. Big r-Theta conversion:

$$f \in \Theta_r(g) \Rightarrow f \in \Theta_q\left(\frac{q}{r} \cdot g\right) \ \forall r, q \in \mathbb{R}_+$$

Proof. $f \in \Theta_r(g(n)) \Rightarrow \forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < r < c_2, \exists n_0 \in \mathbb{N}^*$
$s.t. \ \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ , \ \forall n \geq n_0$
Therefore, $\forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < q < c_2, \exists n_0 \in \mathbb{N}^*$
$s.t. \ \ \frac{r}{q} \cdot c_1 \cdot g(n) \leq f(n) \leq \frac{r}{q} \cdot c_2 \cdot g(n) \ , \ \forall n \geq n_0.$
Thus, $f \in \Theta_q\left(\frac{q}{r} \cdot g\right)$. ■

Another interesting result is obtained by multiplying the last equation by $\frac{q}{r}$:
$\forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < q < c_2, \exists n_0' = n_0 \in \mathbb{N}^*$
$s.t. \ \ c_1 \cdot g(n) \leq \frac{q}{r} \cdot f(n) \leq \frac{r}{q} \cdot g(n) \ , \ \forall n \geq n_0$

Corollary 2.5.1.1.

$$\frac{q}{r} \cdot f \in \Theta_q(g)$$

.

Theorem 2.5.2. Big r-O Conversion:

$$f \in \mathcal{O}_r(g) \Rightarrow f \in \mathcal{O}_q\left(\frac{q}{r} \cdot g\right) \ \forall r, q \in \mathbb{R}_+$$

Proof. Is similar with the proof for Big r-Theta, with the inequalities becoming $f(n) \leq c \cdot g(n)$. ■

Corollary 2.5.2.1. *The following conversion relationship arises:*

$$f \in \mathcal{O}_r(g) \Rightarrow \frac{q}{r} \cdot f \in \mathcal{O}_q(g) \ \forall r, q \in \mathbb{R}_+$$

Theorem 2.5.3. Big r-Omega Conversion:

$$f \in \Omega_r(g) \Rightarrow f \in \Omega_q\left(\frac{q}{r} \cdot g\right) \ \forall r, q \in \mathbb{R}_+$$

Proof. Is similar with the proof for Big r-Theta, with the inequalities becoming $f(n) \geq c \cdot g(n)$. ■

Corollary 2.5.3.1. *The following conversion relationship arises:*

$$f \in \Omega_r(g) \Rightarrow \frac{q}{r} \cdot f \in \Omega_q(g) \ \forall r, q \in \mathbb{R}_+$$

Furthermore, we present some interesting results regarding the connection between $rComplexity$ functions and the correspondent class in the $Bachmann - Landau$ notations. For Small notations, we already presented the connection in definition section.

For Big notations $(\Theta, \mathcal{O}, \Omega)$, we present further some results:

Theorem 2.5.4. *Relationship between Big r-Theta and Big Theta:*

$$f \in \Theta_r(g) \Rightarrow f \in \Theta(g)$$

$$f \in \Theta(g) \Rightarrow \exists r \in \mathbb{R}_+ \ f \in \Theta_r(g)$$

Theorem 2.5.5. *Relationship between Big r-O and Big O:*

$$f \in \mathcal{O}_r(g) \Rightarrow f \in \mathcal{O}(g)$$

$$f \in \mathcal{O}(g) \Rightarrow \exists r \in \mathbb{R}_+ \ f \in \mathcal{O}r(g)$$

Theorem 2.5.6. *Relationship between Big r-Omega and Big Omega:*

$$f \in \Omega_r(g) \Rightarrow f \in \Omega(g)$$

$$f \in \Omega(g) \Rightarrow \exists r \in \mathbb{R}_+ \ f \in \Omega_r(g)$$

2.6 Addition properties

Similar to the calculus in $Bachmann - Landau$ notations (including Big-O arithmetic), a useful technique can be obtained by analyzing the behavior of $r-$Complexity classes in regards to addition exercises.

- Addition in **Big r-Theta**:
  The following relations hold for any correctly defined functions $f, g, f', g', h : \mathbb{N} \longrightarrow \mathbb{R}$, where $h(n) = f'(n) + g'(n) \ \forall n \in \mathbb{N}$, where f', g' are two arbitrary functions such that $f' \in \Theta_r(f), g' \in \Theta_q(g)$ and $r, q \in \mathbb{R}_+$:

  Theorem 2.6.1. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow h \in \Theta_q(g)$.

  Proof. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow \lim_{n\to\infty} \frac{f'(n)}{g'(n)} = 0 \Rightarrow \lim_{n\to\infty} \frac{g'(n) + f'(n)}{g'(n)} = 1$ and using the result from asymptotic analysis section, we have $g'(n) + f'(n) = h(n) \in \Theta_1(g')$.
  Using reflexivity property, $h(n) \in \Theta_q(g)$. ■

  Theorem 2.6.2. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow h \in \Theta_r(f)$.

Proof. Using the last result, by swapping $f \leftarrow g, g \leftarrow f$ and considering the commutativity of addition, we obtain the proof for this statement. ■

Theorem 2.6.3. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow h \in \Theta_r\left(f + \frac{r}{q} \cdot g\right)$.

Proof. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = t \Rightarrow \lim_{n\to\infty} \frac{f'(n)}{g'(n)} = t \cdot \frac{r}{q} = t' \Rightarrow \lim_{n\to\infty} \frac{g'(n) + f'(n)}{g'(n)} = t' + 1$ and using the result from asymptotic analysis section, we have $g'(n) + f'(n) = h(n) \in \Theta_{t'+1}(g')$.
Using reflexivity property, $h(n) \in \Theta_{t'+1}(q \cdot g)$.
Using the conversion technique, we have $h(n) \in \Theta_r(\frac{1}{r} \cdot t \cdot \frac{r}{q} + 1) \cdot q \cdot g)$.
By swapping back t, asymptotically, we can establish: $h(n) \in \Theta_r\left(\frac{1}{r} \cdot (\frac{f(n)}{g(n)} \cdot \frac{r}{q} + 1) \cdot q \cdot g\right)$
Therefore $h \in \Theta_r\left(f + \frac{r}{q} \cdot g\right)$. ■

- Addition in **Big r-O**:
  The following relations hold for any correctly defined functions $f, g, f', g', h : \mathbb{N} \longrightarrow \mathbb{R}$, where $h(n) = f'(n) + g'(n)\ \forall n \in \mathbb{N}$, where f', g' are two arbitrary functions such that $f' \in \mathcal{O}_r(f), g' \in \mathcal{O}_q(g)$ and $r, q \in \mathbb{R}_+$:

  Theorem 2.6.4. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow h \in \mathcal{O}_q(g)$.

  Proof: Is similar with the proof for addition in Big Theta, using conversion presented for Big r-O class.

  Theorem 2.6.5. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow h \in \mathcal{O}_r(f)$.

  Proof: Is similar with the proof for addition in Big Theta, using conversion presented for Big r-O class.

  Theorem 2.6.6. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow h \in \mathcal{O}_r\left(f + \frac{r}{q} \cdot g\right)$.

  Proof: Is similar with the proof for addition in Big Theta, using conversion presented for Big r-O class.

- Addition in **Big r-Omega**: The following relations hold for any correctly defined functions $f, g, f', g', h : \mathbb{N} \longrightarrow \mathbb{R}$, where $h(n) = f'(n) + g'(n)\ \forall n \in \mathbb{N}$, where f', g' are two arbitrary functions such that $f' \in \Omega_r(f), g' \in \Omega_q(g)$ and $r, q \in \mathbb{R}_+$:

Theorem 2.6.7. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow h \in \Omega_q(g)$.

Proof: Is similar with the proof for addition in Big Theta, using conversion presented for Big r-O class.

Theorem 2.6.8. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow h \in \Omega_r(f)$.

Proof: Is similar with the proof for addition in Big Theta, using conversion presented for Big r-Omega class.

Theorem 2.6.9. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow h \in \Omega_r\left(f + \frac{r}{q} \cdot g\right)$.

Proof: Is similar with the proof for addition in Big Theta, using conversion presented for Big r-Omega class.

- Addition in **Small r-O**: Same property as described in $Bachmann - Landau$ notations.
- Addition in **Small r-Omega**: Same property as described in $Bachmann - Landau$ notations.

In a relax notation (consider that by any $r-$Complexity class notation, we denote an arbitrary function part of the class), the following relations can be settled:

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

- **Big r-Theta**:

  Lemma 2.6.10.
  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow \Theta_r(f) + \Theta_q(g) = \Theta_q(g)$$

- **Big r-O**:

  Lemma 2.6.11.
  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow \mathcal{O}_r(f) + \mathcal{O}_q(g) = \mathcal{O}_q(g)$$

- **Big r-Omega**:

  Lemma 2.6.12.
  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow \Omega_r(f) + \Omega_q(g) = \Omega_q(g)$$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$:

- **Big r-Theta**:

  Lemma 2.6.13.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow \Theta_r(f) + \Theta_q(g) = \Theta_r(f)$$

- **Big r-O**:

  Lemma 2.6.14.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow \mathcal{O}_r(f) + \mathcal{O}_q(g) = \mathcal{O}_r(f)$$

- **Big r-Omega**:

  Lemma 2.6.15.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow \Omega_r(f) + \Omega_q(g) = \Omega_r(f)$$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+$:

- **Big r-Theta**:

  Lemma 2.6.16.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow \Theta_r(f) + \Theta_q(g) = \Theta_r(f + \frac{r}{q} \cdot g)$$

- **Big r-O**:

  Lemma 2.6.17.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow \mathcal{O}_r(f) + \mathcal{O}_q(g) = \mathcal{O}_r(f + \frac{r}{q} \cdot g)$$

- **Big r-Omega**:

  Lemma 2.6.18.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow \Omega_r(f) + \Omega_q(g) = \Omega_r(f + \frac{r}{q} \cdot g)$$

The proofs of these Lemmas are imminent from the corresponding theorems.

Chapter 3

Calculus in 1-Complexity

3.1 Introduction and Motivation

This chapter will present a specific set of $rComplexity$ classes, highlighting **Big r-Theta**, **Big r-O** and **Big r-Omega** with unitary parameter (i.e. $r = 1$) by providing useful properties for a more straightforward calculation. The last part of the chapter introduce a new concept for *monotonic, continuous function*: **normal form representation** and defines a new workflow in $rComplexity$: **normalized rComplexity calculus**.

Calculus in $rComplexity$ is dependent on the parameter $r \in \mathbb{R}_+$, and as a result a large number of operations may require rudimentary *conversions* using relations described in *Common properties* and *Notable properties* chapters. Working with unitary $rComplexity$ classes comes effortless and brings an agile manner of operating ample calculus using this Complexity Model.

3.2 Main notations in 1-Complexity Calculus

The following notations and names will be used for describing the asymptotic behavior of a algorithm's complexity characterized by a function, $f : \mathbb{N} \longrightarrow \mathbb{R}$ in $1 - Complexity$.
We define the set of all complexity calculus $\mathcal{F} = \{f : \mathbb{N} \longrightarrow \mathbb{R}\}$
Assume that $n, n_0 \in \mathbb{N}$. Also, we will consider an arbitrary complexity function $g \in \mathcal{F}$. The following notations are particularization ($r = 1$) of notations provided in $rComplexity$:

Definition 3.2.1. Big 1-Theta: *This set defines the group of mathematical functions similar in magnitude with $g(n)$ in the study of asymptotic behavior. A set-based description of this group can be expressed as:*

$$\Theta_1(g(n)) = \{f \in \mathcal{F} \mid \forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < 1 < c_2, \exists n_0 \in \mathbb{N}^* \\ s.t. \ \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ , \ \forall n \geq n_0\}$$

Definition 3.2.2. Big 1-O: *This set defines the group of mathematical functions that are*

known to have a similar or lower asymptotic performance in comparison with $g(n)$. The set of such functions is defined as it follows:

$$\mathcal{O}_1(g(n)) = \{f \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^* \ s.t. \ 1 < c, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) \leq c \cdot g(n), \ \forall n \geq n_0\}$$

Definition 3.2.3. Big 1-Omega: *This set defines the group of mathematical functions that are known to have a similar or higher asymptotic performance in comparison with $g(n)$. The set of all function is defined as:*

$$\Omega_1(g(n)) = \{f \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^* \ s.t. \ c < 1, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) \geq c \cdot g(n), \ \forall n \geq n_0\}$$

Definition 3.2.4. Small 1-O: *This set defines the group of mathematical functions that are known to have a humble asymptotic performance in comparison with $g(n)$. The set of such functions is defined as it follows:*

$$o_1(g(n)) = \{f \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) < c \cdot g(n), \ \forall n \geq n_0\}$$

Note that $o_r(g(n) = o_1(g(n) \forall r \in \mathbb{R}_+$, as Small 1-O notation is r-independent.

Definition 3.2.5. Small 1-Omega: *This set defines the group of mathematical functions that are known to have a commanding asymptotic performance in comparison with $g(n)$. The set of such functions is defined as it follows:*

$$\omega_1(g(n)) = \{f \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) > c \cdot g(n), \ \forall n \geq n_0\}$$

Note that $\omega_r(g(n) = \omega_1(g(n) \forall r \in \mathbb{R}_+$, as Small 1-Omega notation is r-independent.

3.3 Asymptotic Analysis

Calculus in $1 - Complexity$ can be performed as well using either limits of sequences or limits of functions. Consider any two complexity functions $f, g : \mathbb{N}_+ \longrightarrow \mathbb{R}_+$.

Theorem 3.3.1. *Admittance of a function f in* **Big 1-Theta** *class defined by a function g:*

$$f \in \Theta_r(g(n)) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$$

Theorem 3.3.2. *Admittance of a function f in* **Big 1-O** *class defined by a function g:*

$$f \in \mathcal{O}_1(g(n)) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = l, \ l \in [0, 1]$$

Theorem 3.3.3. *Admittance of a function f in* **Big 1-Omega** *class defined by a function g:*

$$f \in \Omega_1(g(n)) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = l, \ l \in [1, \infty)$$

Theorem 3.3.4. *Admittance of a function f in* **Small 1-O** *class defined by a function g:*

$$f \in o_r(g(n)) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Theorem 3.3.5. *Admittance of a function f in* **Small 1-Omega** *class defined by a function g:*

$$f \in \omega_r(g(n)) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

3.4 Common properties

This chapter will present new implications in terms of reflexivity, transitivity, symmetry and projections properties of calculus in $1 - Complexity$.

Theorem 3.4.1. *Reflexivity in* $1 - Complexity$ *- Big 1-Omega notation:*

$$f \in \Theta_1 (f(n))$$

Proof. Let $r = 1$ and use Reflexivity property described in Common properties sections for classes in $rComplexity$ calculus.

$$f \in \Theta_r \left(\frac{1}{r} \cdot f(n) \right) \ \forall r \neq 0$$

Therefore for $r = 1$:

$$f \in \Theta_1 (f(n))$$

■

Theorem 3.4.2. *Reflexivity in* $1 - Complexity$ *- Big 1-O notation:*

$$f \in \mathcal{O}_1 (x \cdot f(n)) \ \forall x \geq 1$$

Proof. Let $r = 1$ and use Reflexivity property for Big *r*-O class in $rComplexity$ calculus. ■

Theorem 3.4.3. *Reflexivity in* $1 - Complexity$ *- Big 1-Omega notation:*

$$f \in \Omega_1 (x \cdot f(n)) \ \forall x \leq 1$$

Proof. Let $r = 1$ and use Reflexivity property for Big *r*-Omega class in $rComplexity$ calculus. ■

The reflexivity does not hold **either** for Small *1*-o and Small *1*-Omega, as these two sets are equal with the classical sets defined in *Bachmann–Landau notations*.

Remark. *Reflexivity in* $1 - Complexity$ *- Small 1-o notation:* $f \notin o_1(f(n))$

Remark. *Reflexivity in* $1 - Complexity$ *- Small 1-Omega notation:* $f \notin \omega_1(f(n))$

Theorem 3.4.4. *Transitivity in* $1 - Complexity$ *- Big 1-Theta notation:*
$f \in \Theta_1(g(n)), g \in \Theta_1(h(n)) \Rightarrow f \in \Theta_1(h(n))$

Proof. Let $r = 1$ and use Transitivity property for Big *r*-Theta class in $rComplexity$ calculus. ■

Theorem 3.4.5. *Transitivity in* $1 - Complexity$ *- Big 1-O notation:* $f \in \mathcal{O}_1(g(n)), g \in \mathcal{O}_1(h(n)) \Rightarrow f \in \mathcal{O}_1(h(n))$

Proof. Let $r = 1$ and use Transitivity property for Big *r*-O class in $rComplexity$ calculus. ■

Theorem 3.4.6. *Transitivity in* $1-Complexity$ *- Big 1-Omega notation:* $f \in \Omega_1(g(n)), g \in \Omega_1(h(n)) \Rightarrow f \in \Omega_1(h(n))$

Proof. Let $r = 1$ and use Transitivity property for Big *r*-Omega class in $rComplexity$ calculus. ■

Lemma 3.4.7. *Transitivity in* $1 - Complexity$ *- small notations:*

$$f \in o_1(g(n)), g \in o_1(h(n)) \Rightarrow f \in o_1(h(n))$$

$$f \in \omega_1(g(n)), g \in \omega_1(h(n)) \Rightarrow f \in \omega_1(h(n))$$

Proof. Small *1*-o and Small *1*-Omega are equal with the classical sets defined in *Bachmann–Landau notations* and thus they conserve transitivity properties ■

Theorem 3.4.8. Symmetry in $1 - Complexity$**:**
$f \in \Theta_1(g(n)) \Rightarrow g \in \Theta_1(f(n))$

Proof. Let $r = 1$ and use Symmetry property for Big *r*-Theta class in $rComplexity$ calculus. ■

Theorem 3.4.9. Transpose symmetry in $1 - Complexity$**:**
$f \in \mathcal{O}_1(g(n)) \Leftrightarrow g \in \Omega_1(f(n))$

Proof. Let $r = 1$ and use Transpose symmetry property described in $rComplexity$ calculus. ■

Theorem 3.4.10. *Transpose symmetry in* $1-Complexity$ *in small notations:* $f \in o_1(g(n)) \Leftrightarrow g \in \omega_1(f(n))$

Proof. Small *1*-o and Small *1*-Omega are equal with the classical sets defined in *Bachmann–Landau notations* and thus they conserve the Transpose symmetry property. ■

Theorem 3.4.11. Projection in $1 - Complexity$**:**
$f \in \Theta_1(g(n)) \Leftrightarrow f \in \mathcal{O}_1(g(n)), f \in \Omega_1(g(n))$

Proof. Let $r = 1$ and use Projection property described in $rComplexity$ calculus. ■

3.5 Addition properties

Addition properties are obtained by assuming $r = 1$ in the rComplexity model.

Theorem 3.5.1. *Addition properties in* **Big 1-Theta***:*
The following relations hold for any correctly defined functions $f, g, f', g', h : \mathbb{N} \longrightarrow \mathbb{R}$*, where* $h(n) = f'(n) + g'(n)\ \forall n \in \mathbb{N}$*, where* f', g' *are two arbitrary functions such that* $f' \in \Theta_1(f), g' \in \Theta_1(g)$*:*

- **If** $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = 0 \Rightarrow h \in \Theta_1(g)$.
- **If** $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty \Rightarrow h \in \Theta_1(f)$.
- **If** $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow h \in \Theta_1(f+g)$.

Theorem 3.5.2. *Addition properties* **Big 1-O***:*
The following relations hold for any correctly defined functions $f, g, f', g', h : \mathbb{N} \longrightarrow \mathbb{R}$*, where* $h(n) = f'(n) + g'(n)\ \forall n \in \mathbb{N}$*, where* f', g' *are two arbitrary functions such that* $f' \in \mathcal{O}_1(f), g' \in \mathcal{O}_1(g)$*:*

- **If** $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = 0 \Rightarrow h \in \mathcal{O}_1(g)$.
- **If** $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty \Rightarrow h \in \mathcal{O}_1(f)$.
- **If** $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow h \in \mathcal{O}_1(f+g)$.

Theorem 3.5.3. *Addition properties* **Big 1-Omega***:*
The following relations hold for any correctly defined functions $f, g, f', g', h : \mathbb{N} \longrightarrow \mathbb{R}$*, where* $h(n) = f'(n) + g'(n)\ \forall n \in \mathbb{N}$*, where* f', g' *are two arbitrary functions such that* $f' \in \Omega_1(f), g' \in \Omega_1(g)$*:*

- **If** $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = 0 \Rightarrow h \in \Omega_1(g)$.
- **If** $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty \Rightarrow h \in \Omega_1(f)$.
- **If** $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow h \in \Omega_r(f+g)$.

In a relax notation (consider that by any $1-$Complexity class notation, we denote an arbitrary function part of the class), the following relations can be settled:

Lemma 3.5.4. *If* $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$*:*

- **Big 1-Theta***:*

$$\Theta_1(f) + \Theta_1(g) = \Theta_1(g)$$

- **Big 1-O**:
$$\mathcal{O}_1(f) + \mathcal{O}_1(g) = \mathcal{O}_1(g)$$
- **Big 1-Omega**:
$$\Omega_1(f) + \Omega_1(g) = \Omega_1(g)$$

Lemma 3.5.5. *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$*:*

- **Big 1-Theta**:
$$\Theta_1(f) + \Theta_1(g) = \Theta_1(f)$$
- **Big 1-O**:
$$\mathcal{O}_1(f) + \mathcal{O}_1(g) = \mathcal{O}_1(f)$$
- **Big 1-Omega**:
$$\Omega_1(f) + \Omega_1(g) = \Omega_1(f)$$

Lemma 3.5.6. *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+$*:*

- **Big 1-Theta**:
$$\Theta_1(f) + \Theta_1(g) = \Theta_1(f+g)$$
- **Big 1-O**:
$$\mathcal{O}_1(f) + \mathcal{O}_1(g) = \mathcal{O}_1(f+g)$$
- **Big 1-Omega**:
$$\Omega_1(f) + \Omega_1(g) = \Omega_1(f+g)$$

Proof. All proofs are based on the particularizing $r = 1$ and rewrite the addition properties from rComplexity. ■

3.6 Normal form functions

In this section, we will take further steps in simplifying calculus by introducing a new concept: the normal form of a monotonic, continuous function.

Definition 3.6.1. *Let* $g : \mathbb{N}^* \longrightarrow \mathbb{R}_+$ *be a monotonic, continuous function. Let the following:*

$$g(n) = \sum_{i=1}^{p} g_i(n)$$

be a decomposition, with correctly defined function $g_i : \mathbb{N}^* \longrightarrow \mathbb{R}_+ \forall i \in [1, p]$*, with the properties:*

- $\exists\ j, \lim_{n\to\infty} \frac{g_j(n)}{g(n)} = 1$

- $\forall i \neq j \ lim_{n\to\infty} \frac{g_j(n)}{g_i(n)} = \infty$
- *there is no another decomposition for* $g_j(n) = \sum_{k=1}^{p'} g_{j_k}(n)$ *such that* $\lim_{n\to\infty} \frac{g_j(n)}{g_{j_k}(n)} = \infty, \forall k \in [1, p']$.

Then, we call g_j *to be a function in* **normal form** *or in* **atomic form**.

Remark. *We will refer to a monotonic, continuous function* g *that is in normal form using the notation* $g_1(n)$.

Remark. *Working in* 1*-Complexity with functions in normal form will involve converting an arbitrary monotonic, continuous function into an atomic representation, given by the normal form* $g_1(n)$*, by applying the decomposition presented in the definition above.This step involve few additional overhead in calculus, but overall the conversions are accessible and simplifies a lot the progress of complexity detection for various algorithms. Nonetheless, it provides powerful benchmarkings solutions for comparing algorithms' efficiency.*

An useful subset of rComplexity Calculus is defined by Calculus using Normalized rComplexity class functions.

Definition 3.6.2. *We will denote by:*

$$f(n) \in \Theta_1(g_1(n))$$

a function f that is in the Big 1-Theta $1-$*Complexity class defined by the function* g_1 *and* g_1 *is in normal form. Working with normal form functions for the class characterization in* $1-$*Complexity Calculus will be named* **normalized rComplexity calculus**.

Remark. *Working in* r*-Complexity with the required conversions such that* $r = 1$ *and the functions defining complexity classes is in normal form* $(g = g_1)$ *is the most simple calculus model while working with* r*-Complexity classes.*

Remark. *A practical use case of this notation will be distinguishable when discussing the N Queens problems analysis for asymptotic time metric while working with* r*-Complexity classes.*

Chapter 4

Relationship between Algorithms and Complexity

4.1 Problems and Algorithms

The concept of a "problem" colligates a set of questions making reference to the dynamic or structural particularities of an entity- processes or objects- that acquires pinpointed, defined answers, whose accuracy can be rigorous demonstrated. The units create the universe of the problem. That segment of the problem, which is formerly familiar (known) represents the facts of the problem and the answers to the question shape the solutions. [**?**];
Making use of a conventional inscription in order to specify an effective approach, taking into account the computational individuality of a calculus machine, it's called an **algorithm**.

An algorithm that can be accepted by a calculus machine consists of a limited amount of instructions with accurate signification, that can be implemented by the machine in a restricted period of time in order to determine the answer of a problem.

The definition of the algorithm reveals the crucial distinction between a function and an algorithm, which is, to be more specifically, the distinction between a functional representation of a problem and its effective solution. [**?**]

In this chapter, we will define various metrics for estimating an algorithm's performances in regard to a specific metric, such as: **RM1**, **RM2** with enhanced variations **ERM1**, **ERM2**.

4.2 From algorithm to Normalized rComplexity

Algorithms' Complexity Theory studies the static and the dynamic performances of mechanical-solvable problems. Static performances aim the clarity of the evaluated algorithm or the abstraction level imposed by used operations, while, more interesting and comparable, the

dynamic performances imposes a calculus regarding the amount of resources (e.g. total execution time, peak memory usage) required for executing the algorithm. [**?**]; Theoretical, such metrics can be exactly calculated for any given algorithm.

Remark. *Mechanical-solvable problems inevitably makes us think to the related field in physics. An intuitive approach on the mechanical-solvable process is accessible to understanding when considering massive motion elements rather than transistors in technologies as advanced as 7nm.*

An example of an early mechanical solution (exhibited in 1862 at The International Exhibition in South Kensington, UK) to a mechanical-solvable problems is the Babbage's design for the difference engine, which is an automatic mechanical calculator designed to tabulate polynomial functions.

This was a solution for approximation some mathematical functions frequently used, including logarithmic and trigonometric functions, which could be approximated by polynomials.

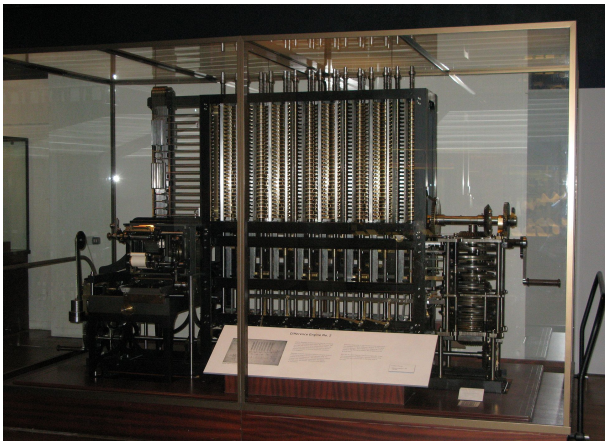


Figure 1: An implementation built from Babbage's design of a difference engine exhibited at The London Science Museum

.

While the dynamic performances of an algorithm are precisely calculable, due to the high complexity (By high complexity it is denoted that it is extremely difficult to exactly compute it) of most computer programs, it is often more suitable an approximation. Such resemblance is provided by the classical model, with drastic compromise compared to the real behavior in a non-asymptotic analysis. A in-between solution for approximating the dynamic performances of an algorithm is provided by Normalized rComplexity Calculus Model, which aim to close the gap between the architecture of the computing machine and a generic-written algorithm for solving a problem.

4.3 Estimating computational time based on Normalized rComplexity

Let an arbitrary algorithm Alg characterized by the complexity function f with a variable input dimension $n \in \mathcal{N}^*$. Consider that the input size is bounded such that $n \in [n_{min}, n_{max}]$. We aim to define various metrics for approximation an average computational time required based on the size of the input and the algorithm's complexity function $T(n_{min}, n_{max})$.

Definition 4.3.1. RM1

Defined as a metric for time estimation (capable of generalization to any other estimators) based on arithmetic mean in Normalized rComplexity model is defined as follows:

$$T(n_{min}, n_{max}) = \frac{\sum_{n=n_{min}}^{n_{max}} g_1(n)}{n_{max} - n_{min} + 1}$$

Definition 4.3.2. RM2

Defined as a metric for time estimation (capable of generalization to any other estimators) based on Mean-Value Theorem(Lagrange) using integrals in Normalized rComplexity model is defined as follows:

$$T(n_{min}, n_{max}) = \frac{\int_{n_{min}}^{n_{max}} g_1(n)dn}{n_{max} - n_{min}}$$

The previous two metrics are tailored for systems where the input size is bounded but there is no additional knowledge regarding the weights and probabilities of occurrence. If this information is available, we can redefine the previous metrics using the acquisition data.

Definition 4.3.3. ERM1, *an enhanced metric for time estimation based on arithmetic mean in Normalized rComplexity model is defined as follows:*

$$T(n_{min}, n_{max}) = \sum_{n=0}^{f} p_n \cdot g_1(n + n_{min})$$

where:

- p_0 *is the weight associated with* $n_0 = n_{min}$
- p_1 *is the weight associated with* $n_1 = n_{min+1}$
- p_f *is the weight associated with* $n_f = n_{max}$

and $f = max - min + 1$.

Definition 4.3.4. ERM2, *an enhanced metric for time estimation based on Mean-Value Theorem(Lagrange) using integrals in Normalized rComplexity model is defined as follows:*

$$T(n_{min}, n_{max}) = \sum_{k=0}^{f-1} p_k \cdot \int_{n_k}^{n_{k+1}} g_1(n)dn$$

where:

- p_0 *is the weight associated with the probability of the input to be bounded in the interval* $[n_0, n_1]$
- p_1 *is the weight associated with the probability of the input to be bounded in the interval* $[n_1, n_2]$
- p_{f-1} *is the weight associated with the probability of the input to be bounded in the interval* $[n_{f-1}, n_f]$

and $f = max - min + 1$, *and* $n_0 = n_{min}, n_f = n_{max}$.

4.4 Comparing algorithms asymptotic performances

This section aims to compare two generic Algorithms ($Alg1$, $Alg2$) time-based performances based on associated Big r-Theta class from Normalized rComplexity model, by asymptotically correlating the characterized complexity functions f, f'.

For specific application of this theoretical work, please refer to the following chapter.

Let $f \in \Theta_1(g_1(n))$ and $f' \in \Theta_1(g_1'(n))$. We can therefore compare the time-based performances of the two algorithms by evaluating:

$$\lim_{n\to\infty} \frac{g_1(n)}{g_1'(n)}$$

Remark.

$$\lim_{n\to\infty} \frac{g_1(n)}{g_1'(n)} = \lim_{n\to\infty} \frac{f(n)}{f'(n)}$$

Definition 4.4.1. *Let* $f \in \Theta_1(g_1(n))$ *and* $f' \in \Theta_1(g_1'(n))$. f *is part of a smaller class than* f' *iff* $\lim_{n\to\infty} \frac{g_1(n)}{g_1'(n)} = 0$.

Lemma 4.4.1. *If* $\lim_{n\to\infty} \frac{g_1(n)}{g_1'(n)} = 0$, *then* $Alg1$ *will terminate faster than* $Alg2$ *for any input size* $n \geq n_0$, *with the possibility of computing* $n_0 \in \mathcal{N}^*$.

Proof. $\lim_{n\to\infty} \frac{g_1(n)}{g_1'(n)} = 0 \Rightarrow g_1(n) < g_1'(n)\ \forall n \geq n_0 \Rightarrow f(n) < f'(n)\ \forall n \geq n_0'$ ■

Definition 4.4.2. *Let $f \in \Theta_1(g_1(n))$ and $f' \in \Theta_1(g'_1(n))$. f is part of the same class as f' iff $\lim_{n\to\infty} \frac{g_1(n)}{g'_1(n)} = r$ and $r \in \mathcal{R}_+$. If f is part of the same class as f', we can distinguish the following cases:*

$$\lim_{n\to\infty} \frac{g_1(n)}{g'_1(n)} = \begin{cases} x \in (0,1), f \text{ has a smaller constant than } f' \\ 1, f \text{ has the same constant as } f' \\ y \in (1,\infty), f \text{ has a bigger constant than } f' \end{cases}$$

Lemma 4.4.2. *If $\lim_{n\to\infty} \frac{g_1(n)}{g'_1(n)} = r \in (0,1)$, then $Alg1$ will terminate faster than $Alg2$ for any input size $n \geq n_0$, with the possibility of computing $n_0 \in \mathcal{N}^*$.*

Proof. $\lim_{n\to\infty} \frac{g_1(n)}{g'_1(n)} < 1 \Rightarrow g_1(n) < g'_1(n)\ \forall n \geq n_0 \Rightarrow f(n) < f'(n)\ \forall n \geq n'_0$ ■

Lemma 4.4.3. *If $\lim_{n\to\infty} \frac{g_1(n)}{g'_1(n)} = r \in (1,\infty)$, then $Alg1$ will terminate slower than $Alg2$ for any input size $n \geq n_0$, with the possibility of computing $n_0 \in \mathcal{N}^*$.*

Proof. $\lim_{n\to\infty} \frac{g_1(n)}{g'_1(n)} > 1 \Rightarrow g_1(n) > g'_1(n)\ \forall n \geq n_0 \Rightarrow f(n) > f'(n)\ \forall n \geq n'_0$ ■

Definition 4.4.3. *Let $f \in \Theta_1(g_1(n))$ and $f' \in \Theta_1(g'_1(n))$. f is part of a bigger class than f' iff $\lim_{n\to\infty} \frac{g_1(n)}{g'_1(n)} = \infty$.*

Lemma 4.4.4. *If $\lim_{n\to\infty} \frac{g_1(n)}{g'_1(n)} = \infty$, then $Alg1$ will terminate slower than $Alg2$ for any input size $n \geq n_0$, with the possibility of computing $n_0 \in \mathcal{N}^*$.*

Proof. $\lim_{n\to\infty} \frac{g_1(n)}{g'_1(n)} = \infty \Rightarrow g_1(n) > g'_1(n)\ \forall n \geq n_0 \Rightarrow f(n) > f'(n)\ \forall n \geq n'_0$ ■

4.5 Comparing algorithms interval-based performances

Comparing algorithms asymptotic performances based on Normalized rComplexity was a generic way of comparing two algorithms' asymptotic performances for unbounded large input. However, asymptotic performance is not always relevant for computer programs that have a settled *(or a interval-based approximation, with or without weights on sub-specific intervals)* range of input size in order of solving a specific task.

Consider an application responsible of scheduling a football league agenda for the next competitive season, avoiding conflicts and following specific objectives. This problem can be modeled and solved as a *constraint satisfaction problem*(CSP) with different flavors. An asymptotic performance analyzer would simply pick the lowest complexity function in consideration to asymptotic behavior. However, the application is not designed to run on

extremely large input size, as the cardinal of the set of all teams part of a football league is a bounded well-known small integer (*most leagues have between 14 and 20 teams*). Therefore, it may be a wise choice to have another method of comparing different algorithms with respect to finite upper bounded input.

For a fixed unique input size $n_0 \in \mathcal{N}^*$, we can use the following natural comparison:

Lemma 4.5.1. *If* $\lim_{n \to n_0} \frac{g_1(n)}{g_1'(n)} = \frac{g_1(n_0)}{g_1'(n_0)} = r \in [0, 1)$ *, then* $Alg1$ *will terminate faster than* $Alg2$ *for the input size* n_0*.*

Symmetrical:

Lemma 4.5.2. *If* $\lim_{n \to n_0} \frac{g_1(n)}{g_1'(n)} = \frac{g_1(n_0)}{g_1'(n_0)} = r \in (1, \infty)$ *, then* $Alg2$ *will terminate faster than* $Alg1$ *for the input size* n_0*.*

Remark. *If* $\lim_{n \to n_0} \frac{g_1(n)}{g_1'(n)} = \frac{g_1(n_0)}{g_1'(n_0)} = 1$*, rComplexity calculus considers* $Alg1$ *and* $Alg2$ *equivalent from a computational cost-based perspective.*

For a bounded interval-based input size $n \in [n_{min}, n_{max}]$, we can use the following comparison based on the metrics determined in the previous sections:

Theorem 4.5.3. *If* $\frac{T_1(n_{min}, n_{max})}{T_2(n_{min}, n_{max})} = r \in [0, 1)$ *, then* $Alg1$ *will terminate faster (in average) than* $Alg2$ *for a randomly distribution of input size* $n \in [n_{min}, n_{max}]$*, assuming*

$$T_1(n_{min}, n_{max}) = \frac{\sum_{n=n_{min}}^{n_{max}} g_1(n)}{n_{max} - n_{min} + 1}$$

and

$$T_2(n_{min}, n_{max}) = \frac{\sum_{n=n_{min}}^{n_{max}} g_1'(n)}{n_{max} - n_{min} + 1}$$

where $Alg1$ *has a complexity function associated to the normalized rComplexity Big 1-Theta defined by* g_1 *and* $Alg2$ *has a complexity function associated to the normalized rComplexity Big 1-Theta defined by* g_1'*.*

Corollary 4.5.3.1. *If* $\frac{T_1(n_{min}, n_{max})}{T_2(n_{min}, n_{max})} = r \in (1, \infty)$ *, then* $Alg2$ *will terminate faster (in average) than* $Alg1$ *for a randomly distribution of input size* $n \in [n_{min}, n_{max}]$*.*

Remark. *If* $\frac{T_1(n_{min}, n_{max})}{T_2(n_{min}, n_{max})} = 1$*, rComplexity calculus considers* $Alg1$ *and* $Alg2$ *equivalent from a computational cost-based perspective for a randomly distribution of input size* $n \in [n_{min}, n_{max}]$*.*

Remark. *If* $\frac{T_1(n_{min}, n_{max})}{T_2(n_{min}, n_{max})} = 1$*, a tiebreak can be done by a deep analysis of the less dominant features as described in the Enhanced rComplexity model feature extraction in the next chapter.*

Remark. *Theorem stands likewise using Mean-Value Theorem(Lagrange), with*

$$T_1(n_{min}, n_{max}) = \frac{\int_{n_{min}}^{n_{max}} g_1(n)dn}{n_{max} - n_{min}}$$

$$T_2(n_{min}, n_{max}) = \frac{\int_{n_{min}}^{n_{max}} g_1'(n)dn}{n_{max} - n_{min}}$$

for a randomly distribution of input size $n \in [n_{min}, n_{max}]$.

Currently, we assumed that the input size is randomly distributed in a bounded interval. Further information about the probabilistic distribution of input size $n \in [n_{min}, n_{max}]$ can offer more insights to this complexity model and a refined comparison model can be established.

Theorem 4.5.4. *If* $\frac{T_1(n_{min}, n_{max})}{T_2(n_{min}, n_{max})} = r \in [0, 1)$ *, then* $Alg1$ *will terminate faster (in average) than* $Alg2$ *for a probabilistic distribution of input size* $n \in [n_{min}, n_{max}]$*, assuming*

$$T_1(n_{min}, n_{max}) = \sum_{n=0}^{f} p_n \cdot g_1(n + n_{min})$$

and

$$T_2(n_{min}, n_{max}) = \sum_{n=0}^{f} p_n \cdot g_1'(n + n_{min})$$

where $Alg1$ *has a complexity function associated to the normalized rComplexity Big 1-Theta defined by* g_1 *and* $Alg2$ *has a complexity function associated to the normalized rComplexity Big 1-Theta defined by* g_1'*, and*

- p_0 *is the weight associated with* $n_0 = n_{min}$
- p_1 *is the weight associated with* $n_1 = n_{min+1}$
- p_f *is the weight associated with* $n_f = n_{max}$

with $f = max - min + 1$.

Corollary 4.5.4.1. *If* $\frac{T_1(n_{min}, n_{max})}{T_2(n_{min}, n_{max})} = r \in (1, \infty)$ *, then* $Alg2$ *will terminate faster (in average) than* $Alg1$ *for a probabilistic distribution of input size* $n \in [n_{min}, n_{max}]$.

Remark. *If* $\frac{T_1(n_{min}, n_{max})}{T_2(n_{min}, n_{max})} = 1$*, rComplexity calculus considers* $Alg1$ *and* $Alg2$ *equivalent from a computational cost-based perspective for probabilistic distribution of input size* $n \in [n_{min}, n_{max}]$.

Remark. *Theorem stands likewise using Mean-Value Theorem(Lagrange), with :*

$$T_1(n_{min}, n_{max}) = \sum_{k=0}^{f-1} p_k \cdot \int_{n_k}^{n_{k+1}} g_1(n)dn$$

$$T_2(n_{min}, n_{max}) = \sum_{k=0}^{f-1} p_k \cdot \int_{n_k}^{n_{k+1}} g_1'(n)dn$$

for a probabilistic distribution of input size $n \in [n_{min}, n_{max}]$*, where:*

- p_0 *is the weight associated with the probability of the input to be bounded in the interval* $[n_0, n_1]$
- p_1 *is the weight associated with the probability of the input to be bounded in the interval* $[n_1, n_2]$
- p_{f-1} *is the weight associated with the probability of the input to be bounded in the interval* $[n_{f-1}, n_f]$

and $f = max - min + 1$*, and* $n_0 = n_{min}, n_f = n_{max}$*.*

Chapter 5

rComplexity in Practice

5.1 Human-driven calculus of rComplexity

This section aims to present a simple methodology of calculating by hand the associated rComplexity class for any given algorithm, provided a predefined instruction set architecture and the correspondence between generic instructions and required time for execution as well as enhanced hardware designs details related to the total execution time (no. of stages of pipeline, scalability degree, etc.)[**?**]. Even if the process of calculating an exact rComplexity class associated to a real algorithm is unpractical, the method provided can be applied with colossal endeavor. Later on, we will present an automated method of estimating the rComplexity of a given algorithm.

5.1.1 N-Queens' Problem

As a fundamental process-key in testing the capacity of the AI in solving various problems together with that of creating low-complexity algorithms, N queens counts as one of the most applicable and explored programmes in the field of mathematics and programming. Being initially designed as a solution of Max Bezzel's 8 queens puzzle (proposed in 1848), that by Franz Nauck created the generalized puzzle for the known N-queens problem.

Due to movement complexity of queens, the increased number of required computations as the size of the input increases(time complexity grows fast), and the vast number of possible board states, N-queens problem has been explored by mathematicians and programmers quite extensively.

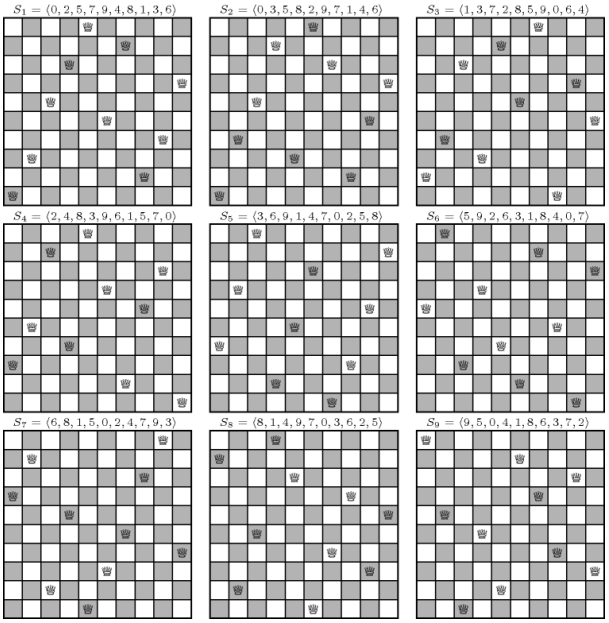


Figure 2: Possible solutions for $n = 10$

This subsection proposes a naive solution. In order to begin to solve this problem,an observation about choosing the N Queens on the board with the dimensions n x n is needed. In order not to have attacks between two Queens, it must not have two Queens on the same line or column.

We define a Queen Q by the pair on points (x, y), where $x, y \leq n$, are the coordinates corresponding to the position on the chessboard.

The problem can be solved by checking all possible permutations:

$Q(1) = (1, y_1)$, $Q(2) = (2, y_2)$ ... , $Q(n) = (n, y_n)$
where $y_1, y_2, ..., y_n \; in \; 1, 2, ..., n$ and $y_1, y_2, ..., y_n$ distinct.

In these conditions, the only action that still needs to be verified is that of the *"diagonal attack"* of any two Queens.

Algorithm 1 Exhaustive N-Queens' Problem Pseudocode

1: **procedure** $\text{QUEENPROBLEM}(currentMatrix, row, column)$
2: **if** $column = 1$ and $row = n + 1$ **then**
3: print currentMatrix
4: Return
5: **end if**
6: **for** i=1..n **do**
7: **if** noConflict(currentMatrix,row,column) = TRUE **then**
8: $currentMatrix[row][i] \leftarrow 1$
9: $QueenProblem(currentMatrix, row + 1, 1)$
10: $currentMatrix[row][i] \leftarrow 0$ //BackTracking Step
11: **end if**
12: **end for**
13: **end procedure**
14: **procedure** $\text{MAIN}(n)$
15: $read\ n$
16: $currentMatrix.initiate()$
17: $currentMatrix.setrows(n)$
18: $currentMatrix.setcolumns(n)$
19: $currentMatrix \leftarrow \begin{bmatrix} 0 & 0 & ... & 0 & 0 \\ 0 & 0 & ... & 0 & 0 \\ & & ... & & \\ 0 & 0 & ... & 0 & 0 \\ 0 & 0 & ... & 0 & 0 \end{bmatrix}$
20: $QueenProblem(currentMatrix, 1, 1)$
21: **end procedure**

To begin with, a method of calculating traditional complexity using the classical model is presented hereby. To calculate the complexity of the QueenProblem algorithm, we will individually analyze the complexities of the operations in the QueenProblem function. We will note with n the number of Queens to be completed at a time t of the algorithm.

The matrix display operation occupies $O(n^2)$ (it represents a simple lookup in a $n\ x\ n$ matrix)

Each Queen Problem function calls other n instances, through recursive calls, of a smaller order with one unit complexity index $(n - 1)$.

Algorithm 2 noConflict Helper function

```
1: procedure noConflict(currentMatrix, row, column)
2:     for i=1..row-1 do
3:         if currentMatrix[i][column] = 1 then
4:             return FALSE
5:         end if
6:     end for
       //Check on "diagonal" parallel with second diagonal
7:     i ← row
8:     for j=column..n do
9:         if currentMatrix[i][j] = 1 then
10:            return FALSE
11:        end if
12:        if i ≤ 0 then
13:            break
14:        end if
15:        i ← i − 1
16:    end for
       //Check on "diagonal" parallel with main diagonal
17:    i ← row
18:    for j=column..0 : increment -1 do
19:        if currentMatrix[i][j] = 1 then
20:            return FALSE
21:        end if
22:        if i ≤ 0 then
23:            break
24:        end if
25:        i ← i − 1
26:    end for
27: end procedure
```

The forward and backtracking operations respectively are performed in $O(1)$. The time consuming operation is the conflict checking operation (refer to the noConflict function defined below) that is performed in linear time $O(n)$, because this function iterates, individually (only to the left of the created matrix), after the lines of the matrix (in $O(n)$), after a diagonal parallel to the secondary diagonal of the matrix (in $O(n)$) and after a diagonal parallel to the main diagonal of the matrix (in $O(n)$). Therefore:

$$QueenProblem(n) = n \cdot QueenProblem(n-1) + n \cdot O(n),\ O(1) = O(n^2)$$

Proceeding to solving this equation:

$QueenProblem(n) = n \cdot QueenProblem(n-1) + n \cdot O(n) | \cdot\ 1$
$QueenProblem(n-1) = (n-1) \cdot QueenProblem(n-2) + (n-1) \cdot O(n-1) | \cdot\ n-0$
$QueenProblem(n-2) = (n-2) \cdot QueenProblem(n-3) + (n-2) \cdot O(n-2) | \cdot\ n \cdot (n-1)$
...
$QueenProblem(3) = 3 \cdot QueenProblem(2) + 3 \cdot O(3) | \cdot\ n \cdot (n-1) \cdot ... \cdot 3$
$QueenProblem(2) = 2 \cdot QueenProblem(1) + 2 \cdot O(2) | \cdot\ n \cdot (n-1) \cdot ... \cdot 2$

After computing the additions of the previously weighted equation, in a compact representation, the following result occurs:

$$QueenProblem(n) = n! \cdot QueenProblem(1) + \sum_{i=2}^{n} i \cdot O(i)$$

$QueenProblem(n) = n! \cdot O(n^2) + \sum_{i=2}^{n} O(i^2)$
$QueenProblem(n) = n! \cdot O(n^2) + O(\sum_{i=2}^{n} i^2)$
$QueenProblem(n) = n! \cdot O(n^2) + O(\frac{n \cdot (n+1) \cdot (2n+1)}{6} - 1)$
$QueenProblem(n) = n! \cdot O(n^2) + O(n^3)$
$QueenProblem(n) = O(n! \cdot n^2 + n^3)$

Then, the Bachmann–Landau associated Big O-complexity class for the given algorithm is:

$$QueenProblem(n) = O(n^2 \cdot n!)$$

---

rComplexity calculus is similar, but it requires higher precision of class approximation and additional architectural aspects. First change is that the noConflict function has it's performance dependent on the architecture, as not all CPU operations takes the same amount of CPU cycles.

For instance, for an x86 CPU, the associated rComplexity class would be $O_1(c_{noConflict} * n)$, where $c_{noConflict}$ is a constant corresponding to the no. of cycles required to perform the operations inner the for-loop.

For the pseudocode:

```
1: for i=1..row-1 do
2:     if currentMatrix[i][column] = 1 then
3:         return FALSE
4:     end if
5: end for
```

In a broad manner, we can estimate that the cost for the inner snippet is $c_{computeOffset} + c_{readMemory} + c_{check=1}$, where in order to compute offset, we need to perform $i * ROWS + column$, so $c_{computeOffset} = O_1(c_{addition} + c_{multiplication} + 3 * c_{readMemory})$.

Seeking in a x86 Cycle cost table for various operations, we can estimate the primitives $c_{addition} = O_1(4)$ and $c_{multiplication} = O_1(4)$. For memory access, the time varies, on reasons based on cache mechanics, prefetch and other runtime mechanism for improvement in speed of read operations. A satisfying margin would be $c_{readMemory} = O_1(100)$.

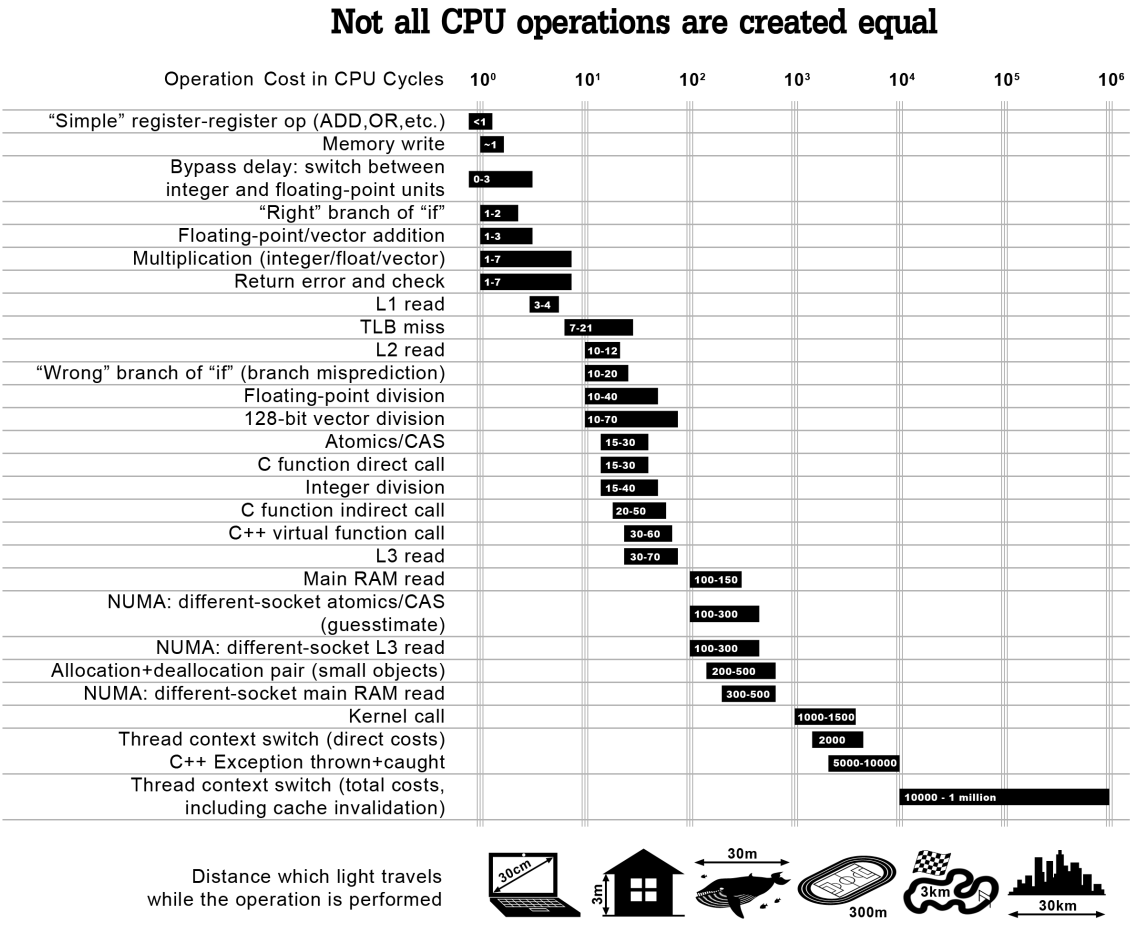


Figure 3: Averages of the Costs (in CPU Clock Cycles) for the fundamental assembly instructions [**?**]

Suppose for a complete check and jump at an pre-computed address $c_{check=1} = 10$.

Therefore, for the inner snippet using the above estimations, the rComplexity would be $O_1(4 * 100 + 4 + 4 + 10) = O_1(418)$. Now, the whole snippet will have the complexity $O_1((418 + c_{for\_checks}) \cdot n)$, where $c_{for\_checks} = c_{check=row} + c_{addition}$ is the associated cost for incrementing the index and other checks.

Thus, the rComplexity of the snippet is $O_1((418 + 14) \cdot n) = O_1(432 \cdot n)$ using the above estimations.

For an exact value, we need to check the associated generated assembly code for the architecture (example x86-32bit):

```
f:
```

```
    push ebp
    mov ebp, esp
    sub esp, 16
    mov DWORD PTR [ebp-4], 0
    jmp .L2
.L5:
    mov eax, DWORD PTR [ebp-4]
    imul edx, eax, 400 # for a 20x20 board
    mov eax, DWORD PTR [ebp+16]
    add edx, eax
    mov eax, DWORD PTR [ebp+12]
    mov eax, DWORD PTR [edx+eax*4]
    cmp eax, 1
    je .L6
    add DWORD PTR [ebp-4], 1
.L2:
    mov eax, DWORD PTR [ebp-4]
    cmp eax, DWORD PTR [ebp+8]
    jl .L5
    jmp .L1
.L6:
    nop
.L1:
    leave
    ret
```

The inner loop calculus are provided inner $.L5$ label. Having this code and the x86 cycles/instructions table, we can calculate the ideal rComplexity $\Theta_1(n \cdot (c_{.L2} + c_{.L5}))$.

Also, $c_{.L2} = c_{mov} + c_{cmp} + c_{li}$ and $c_{.L5} = c_{mov} + c_{imul} + c_{mov} + c_{add} + c_{mov} + c_{mov} + c_{cmp} + c_{je} + c_{add}$.

Using reflexivity, we conclude the rComplexity of the snippet 1 is

$$f_{snippet_1} = \Theta_1(n \cdot (5 * c_{mov} + c_{imul} + 2 * c_{add} + 2 * c_{cmp} + c_{je} + c_{li}))$$

Having this snippet's associated complexity calculated, we can proceed to calculate the other independent for-loops in the noConflict function:

```
1: for j=column..n do
2:     if currentMatrix[i][j] = 1 then
3:         return FALSE
4:     end if
5:     if i ≤ 0 then
6:         break
```

```
    end if
    i ← i − 1
end for
```

We will associate the complexity function of this snippet with $f_{snippet_2}$. Similarly, for the snippet:

```
for j=column..0 : increment -1 do
    if currentMatrix[i][j] = 1 then
        return FALSE
    end if
    if i ≤ 0 then
        break
    end if
    i ← i − 1
end for
```

a complexity function will be obtained described by $f_{snippet_3}$.

Using addition properties, the complexity function for the procedute noConflict will be $f_{snippet_1} + f_{snippet_2} + f_{snippet_3} = O_1(c_{noConflict} \cdot n)$, where $c_{noConflict} \cdot n$ is the architecture-dependant constant.

For instance, for x86, we can approximate $c_{noConflict \cdot n} = 432 * 3 = 1296$ and $f_{snippet_1} + f_{snippet_2} + f_{snippet_3} = O_1(1296 \cdot n)$.

Going deeper, tracing the flow, we can evaluate the complexity of N-Queens Problem procedure by checking all elements involved in the procedure.

$$QueenProblem(n) = O_1(recursiveCall) + O_1(noConflict) + O_1(overhead) + O_1(basecase)$$

Therefore:

$$\begin{cases} QueenProblem(n) = n \cdot QueenProblem(n-1) + O_1(1296 \cdot n)(c_{overheadFor} + c_{setBitsMatrix}) \cdot O_1(n) \\ QueenProblem_1(1) = O_1(c_{printMatrix}) \end{cases}$$

where $O_1(1296 \cdot n)$ is the rComplexity for the noConflict check.

Using the previous estimation, we assume $c_{overheadFor} = 14$ and $c_{setBitsMatrix} = 108$. Thus, the recurrent equation looks as follows:

$$QueenProblem(n) = n \cdot QueenProblem(n-1) + O_1(1418 \cdot n)$$

For the base case, we need to compute the rComplexity of the $printMatrix$ method.

$$\begin{cases} QueenProblem(1) = O_1(n^2 \cdot (c_{computeOffset} + c_{readMatrixElement})) \\ QueenProblem(1) = O_1(n^2 \cdot 408) \end{cases}$$

For simplicity, we can rewrite:

$$\begin{cases} QueenProblem(n) = n \cdot QueenProblem(n-1) + 1418 \cdot O_1(n) \\ QueenProblem(1) = 408 \cdot O_1(n^2) \end{cases}$$

We can model the system as an recurrence equation:

$$g(n) = n \cdot g(n-1) + 1418 \cdot n; g(0) = 408 \cdot n^2$$

with the solution:

$$408 \cdot n^3 \cdot \Gamma(n) + 1418 \cdot e \cdot n \cdot \Gamma(n, 1)$$

where Γ represents the *gamma function* that satisfies:

$$\Gamma(x) = \int_0^\infty s^{x-1} e^{-s} ds$$

and $\Gamma(n, x)$ represents the *incomplete gamma function* that satisfies:

$$\Gamma(x, n) = \int_n^\infty s^{x-1} e^{-s} ds$$

The solution of this recurrence equation in rComplexity calculus (with $n \in \mathbb{N}$) is:

$$QueenProblem(n) = O_1(408 \cdot n^2 \cdot n!) + O_1(1418 \cdot n \cdot n!)$$

Using the O_1 addition properties, we conclude:

$$QueenProblem(n) = O_1(408 \cdot n^2 \cdot n!)$$

Remark. *The rComplexity function associated with the algorithm* $(408 \cdot n^2 \cdot n!)$ *is from the same tradition complexity class as calculated before* $O(n^2 \cdot n!)$.

5.2 Automatic estimation of rComplexity

This section aims to present a solution for automation for calculating an approximate of the associated rComplexity class for any given algorithm. The prerequisites for this method implies a technique for obtaining relevant metric-specific details for diversified input dimensions. For instance, if time is the monitored metric, there must exist a collection of pertinent data linking the correspondence between input size and the total execution time for the designated input size.

5.2.1 Estimation for algorithms with known Bachmann–Landau Complexity

Reckoning an associated rComplexity class (f) for an algorithm with established Bachmann–Landau Complexity (g) consists in the process of tailoring an suitable constant c, such that $f \approx \Theta_1(c \cdot g)$ or in Big-O calculus, $f \leq \mathbb{O}_1(c \cdot g)$. The approach presented below is a particularized version of linear regression, which attempts to model the relationship between various variables by fitting a linear equation to observed data. Even if the model generally follows the classical pattern of a Machine Learning Process (training, predicting, etc.), where a training example consists of a pair ($inputSize$, $metricValue$).

A trick (frequently used in data science) is used to adjust the entry values if the Bachmann–Landau relationship between the $inputSize$ and the metric is known. In order to adjust the learning set to a more knowledgeable set, we can extract new features and replace all the ($inputSize$, $metricValue$) pairs with ($g(inputSize)$, $metricValue$), where g is the known Bachmann–Landau Complexity function converted into Normal form.

The importance of this trick can be emphasized comparing the classical linear regression model with various learning datasets. For the matrix multiplication problem, a naive algorithm (with Bachmann–Landau Complexity $\mathbb{O}(n^3)$) has been implemented. After testing, the algorithm has been deployed and executed matrix multiplications for various sizes of the matrixes. The execution has been audited and the results have been summarized in the following table, which represents the correspondance between matrix dimension and running time:

input Size	Time (s)	input Size	Time (s)	input Size	Time (s)	input Size	Time (s)	input Size	Time (s)	input Size	Time (s)
16	0.000018	1024	8.52	2112	123.73	3200	462.71	4288	999.25	5376	1647.23
32	0.000339	1088	10.96	2176	135.21	3264	358.01	4352	864.15	5440	1929.46
48	0.001123	1152	13.76	2240	149.43	3328	540.64	4416	987.71	5504	1693.75
64	0.001468	1216	18.03	2304	148.82	3392	446.29	4480	1095.01	5568	1986.99
128	0.015482	1280	21.55	2368	180.07	3456	592.91	4544	1126.39	5632	2002.14
256	0.084991	1344	25.59	2432	117.83	3520	480.98	4608	1147.28	5696	2013.22
320	0.172411	1408	30.66	2496	213.51	3584	621.36	4672	1280.23	5760	2066.87
384	0.296398	1472	28.27	2560	236.41	3648	582.46	4736	1228.73	5824	2029.52
448	0.535461	1536	41.91	2624	255.39	3712	628.36	4800	1222.14	5888	2236.73
512	0.764842	1600	48.90	2688	275.05	3776	650.76	4864	1256.73	5952	2503.13
576	1.308485	1664	56.34	2752	167.51	3840	651.43	4928	1290.35	6016	2317.91
640	1.803608	1728	70.47	2816	268.64	3904	626.79	4992	1488.43	6080	2395.12
704	2.516613	1792	71.02	2880	296.00	3968	659.29	5056	1432.41	6144	2718.02
768	3.334002	1856	84.41	2944	304.91	4032	915.86	5120	1093.62		
832	4.315076	1920	67.05	3008	300.06	4096	769.01	5184	1649.00		
896	5.571474	1984	99.74	3072	228.34	4160	864.89	5248	1729.42		
960	7.076612	2048	110.90	3136	446.72	4224	789.13	5312	1743.26		

Table 5.1: Reported timings for different input size for a naive matrix multiplication algorithm in $\mathbb{O}(n^3)$. Results have been obtained on an i5 3.2GHz, x86_64 Architecture with L1d cache: 32K, L1i cache: 32K, L2 cache: 256K, L3 cache: 6144K

Training a linear regression model on this dataset would result in model with a linear equation to observed data, similar with the one below(a). Changes in the original dataset can enhance fitting results for the regression.

Examples for ($inputSize$, $metricValue$) $\Rightarrow$ ($inputSize^n$, $metricValue$) are presented in (b)

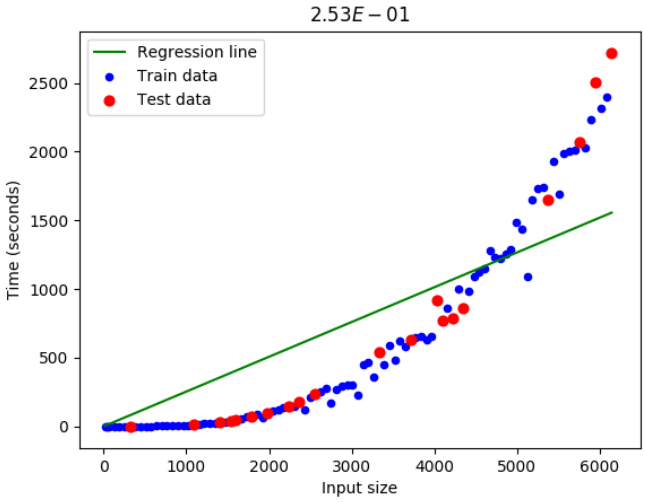
$n = 2$, (c) $n = 3$ and (d) $n = 4$.

input Size^3	Time (s)	input Size^3	Time (s)	input Size^3	Time (s)	input Size^3	Time (s)	input Size^3	Time (s)	input Size^3	Time (s)
4.10E+03	0.000018	1.07E+09	8.52	9.42E+09	123.73	3.28E+10	462.71	7.88E+10	999.25	1.55E+11	1647.23
3.28E+04	0.000339	1.29E+09	10.96	1.03E+10	135.21	3.48E+10	358.01	8.24E+10	864.15	1.61E+11	1929.46
1.11E+05	0.001123	1.53E+09	13.76	1.12E+10	149.43	3.69E+10	540.64	8.61E+10	987.71	1.67E+11	1693.75
2.62E+05	0.001468	1.80E+09	18.03	1.22E+10	148.82	3.90E+10	446.29	8.99E+10	1095.01	1.73E+11	1986.99
2.10E+06	0.015482	2.10E+09	21.55	1.33E+10	180.07	4.13E+10	592.91	9.38E+10	1126.39	1.79E+11	2002.14
1.68E+07	0.084991	2.43E+09	25.59	1.44E+10	117.83	4.36E+10	480.98	9.78E+10	1147.28	1.85E+11	2013.22
3.28E+07	0.172411	2.79E+09	30.66	1.56E+10	213.51	4.60E+10	621.36	1.02E+11	1280.23	1.91E+11	2066.87
5.66E+07	0.296398	3.19E+09	28.27	1.68E+10	236.41	4.85E+10	582.46	1.06E+11	1228.73	1.98E+11	2029.52
8.99E+07	0.535461	3.62E+09	41.91	1.81E+10	255.39	5.11E+10	628.36	1.11E+11	1222.14	2.04E+11	2236.73
1.34E+08	0.764842	4.10E+09	48.90	1.94E+10	275.05	5.38E+10	650.76	1.15E+11	1256.73	2.11E+11	2503.13
1.91E+08	1.308485	4.61E+09	56.34	2.08E+10	167.51	5.66E+10	651.43	1.20E+11	1290.35	2.18E+11	2317.91
2.62E+08	1.803608	5.16E+09	70.47	2.23E+10	268.64	5.95E+10	626.79	1.24E+11	1488.43	2.25E+11	2395.12
3.49E+08	2.516613	5.75E+09	71.02	2.39E+10	296.00	6.25E+10	659.29	1.29E+11	1432.41	2.32E+11	2718.02
4.53E+08	3.334002	6.39E+09	84.41	2.55E+10	304.91	6.55E+10	915.86	1.34E+11	1093.62		
5.76E+08	4.315076	7.08E+09	67.05	2.72E+10	300.06	6.87E+10	769.01	1.39E+11	1649.00		
7.19E+08	5.571474	7.81E+09	99.74	2.90E+10	228.34	7.20E+10	864.89	1.45E+11	1729.42		
8.85E+08	7.076612	8.59E+09	110.90	3.08E+10	446.72	7.54E+10	789.13	1.50E+11	1743.26		

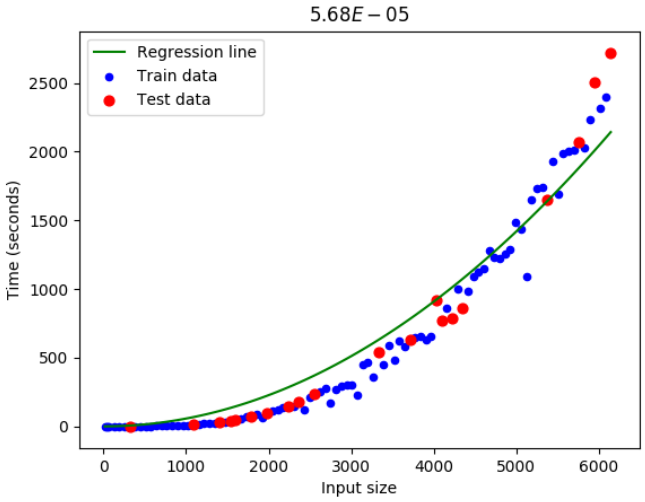
Table 5.2: Training data for the linear regression model after mapping ($inputSize$, $metricValue$) pairs with ($inputSize^3$, $metricValue$)

As an intuition (due to the associated complexity function $O(n^3)$ in the Bachmann–Landau Complexity model), the natural fit was obtained when using $g(n) = n^3$ with consideration to generalization. If we choose much much bigger degree polynomial transformations, we may obtain better results on this datasets, but the models are becoming subject to overfit.

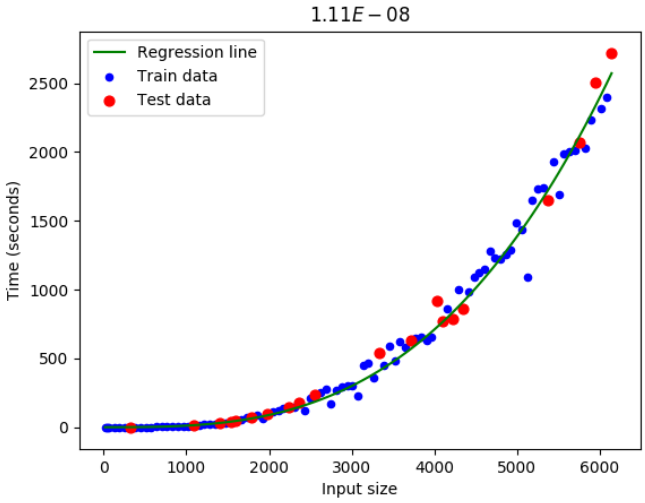
The subject of Matrix multiplication is furthered discussed in a distinct section.



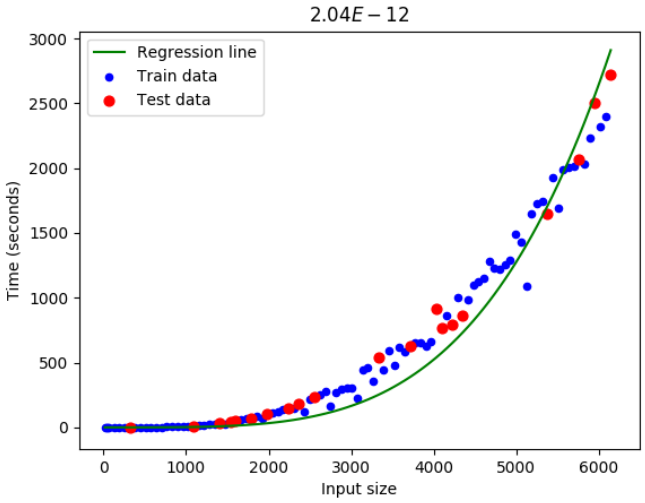
(a) Linear Regression Model trained with features obtained using the transformation $g(n) = n^1$



(b) Linear Regression Model trained with features obtained using the transformation $g(n) = n^2$



(c) Linear Regression Model trained with features obtained using the transformation $g(n) = n^3$



(d) Linear Regression Model trained with features obtained using the transformation $g(n) = n^4$

Figure 4: Various prediction boundaries based on accommodated training dataset using multiple relations g. Training data are obtained for different input size for a naive matrix multiplication algorithm in $\mathbb{O}(n^3)$

5.2.2 Estimation for algorithms with unknown Bachmann–Landau Complexity

Estimation for algorithms with unknown Bachmann–Landau Complexity becomes a lot more difficult as there are numerous possible candidates for a matching complexity function.

A general polynomial Performance model normal form is presented in Chapter 2.1 Automatic Empirical Performance Modeling of Parallel Programs [**?**]. An enhance model for complexity

functions should contain also an exponential behavior, which is often seen as a synergy between NP-Hard problems. Thus, we propose the following general expression:

$$f(n) = \sum_{t=1}^{y} \sum_{k=1}^{x} c_k \cdot n^{p_k} \cdot log_{l_k}^{j_k}(n) \cdot e_t^n \cdot \Gamma(n)^{g_k}$$

This representation is, of course, not exhaustive, but it works in most practical schemes. An intuitive motivation is a consequence of how most computer algorithms are designed. [**?**]

5.3 Matrix multiplication

In this section, we aim to provide an example of estimation for matrix multiplication algorithms with known Bachmann–Landau Complexity.

Matrix multiplication plays very important role in many scientific disciplines because of fact that it is considered as the main tool for many other computations in different areas, like those in seismic analysis, different simulations (like galactic simulations), aerodynamic computations, signal and images processing. [**?**]

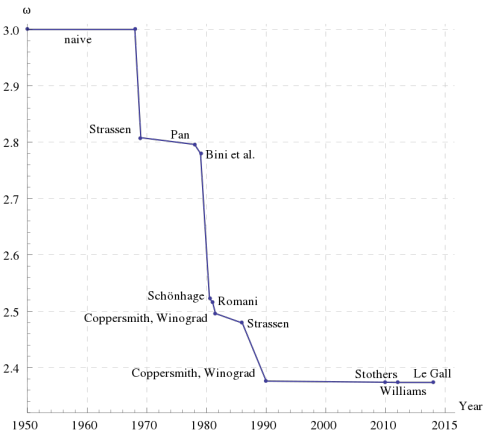


Figure 5: Before 1969, all known matrix multiplication algorithm were in $O(n^3)$. After Volker Strassen first published this algorithm and proved that the naive algorithm wasn't optimal, various attempts to narrow down the exponent ω appeared. One big breakthrough was brought by Coppersmith–Winograd algorithm, with complexity $O(n^{2.375})$.

5.3.1 Naive and optimized implementations

We analyzed various naive matrix multiplication algorithms $O(n^3)$ with memory-access improvements (cache-locality of loops, Blocked Matrix Multiplication) and an efficient implementation of Strassen algorithm.

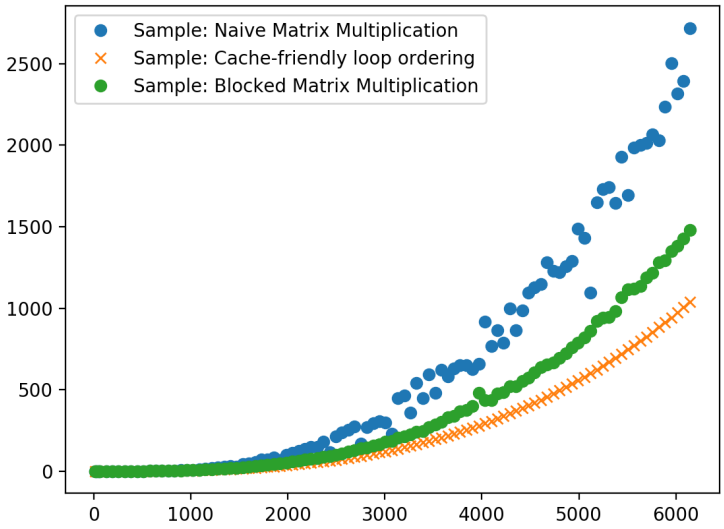


Figure 6: The results above have been reported on an i5 3.2GHz, x86_64 Architecture with L1d cache: 32K, L1i cache: 32K, L2 cache: 256K, L3 cache: 6144K. We do not postulate that the methods above cannot be enhanced or the efficiency of the optimizations are in a specific order. We aim to provide various estimation for these implementations of matrix multiplication algorithms with known $O(n^3)$ Complexity. Please remark the natural distribution of the two cache-friendly algorithms presented on larger datasets vs the naive algorithm, susceptible to outliers.

Using the method described in estimating section, we can tailor an architecture-specific complexity function $f(n) = c \cdot n^3$.

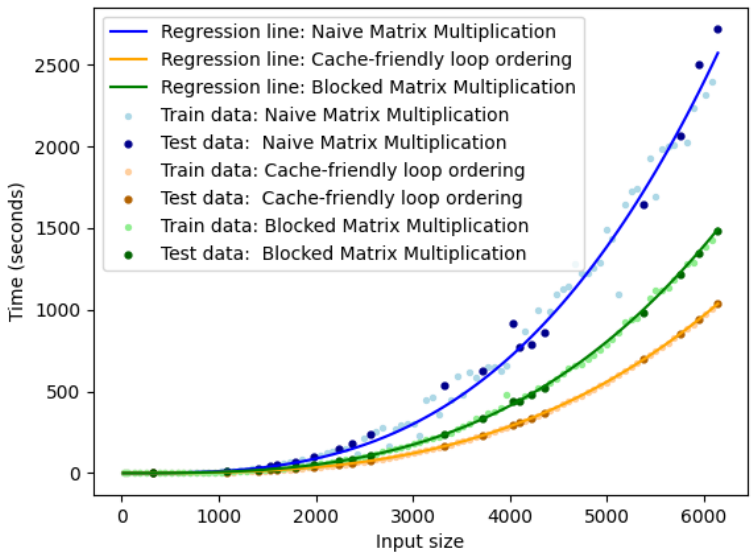


Figure 7: Regression lines corresponding to each of the matrix-multiplication algorithms

After training the regression models for each algorithm, we obtained the coefficients c, that defines the complexity function $f(n) = c \cdot n^3$.

Algorithm	Complexity Function
Naive Matrix Multiplication	$O_1(1.109 \cdot 10^{-8} \cdot n^3 \cdot HZ)$
Cache-friendly loop ordering	$O_1(4.472 \cdot 10^{-9} \cdot n^3 \cdot HZ)$
Blocked Matrix Multiplication	$O_1(6.441 \cdot 10^{-9} \cdot n^3 \cdot HZ)$

Table 5.3: In this representation, the complexity function is scaled to produce output in seconds. In order to obtain rComplexity function, multiplying with processor frequency is mandatory $HZ \approx 3.2 \cdot 10^9$. The regression model was evaluating using the above metrics: Root Mean Square Error (RMSE) and R^2 (coefficient of determination) regression score function

Furthermore, the model was evaluated adopting classical regression evaluation metrics, independently settled on training and testing data.

Algorithm	**c**	*[Training] RMSQ*	*[Training] R2 score*	[Test] RMSQ:	*[Test] R2 score*
Naive Matrix Multiplication	*1.109e-08*	5740.95	0.9886	6136.10	0.9912
Cache-friendly loop ordering	*4.472e-09*	0.2517	0.9999969	0.2917	0.999997
Blocked Matrix Multiplication	*6.441e-09*	185.70	0.9989	63.64	0.99971

Table 5.4: The regression model was evaluated using the above metrics: Root Mean Square Error (RMSE) and R^2 (coefficient of determination) regression score function. These high scores indicates that the regression line produces accurate estimations.

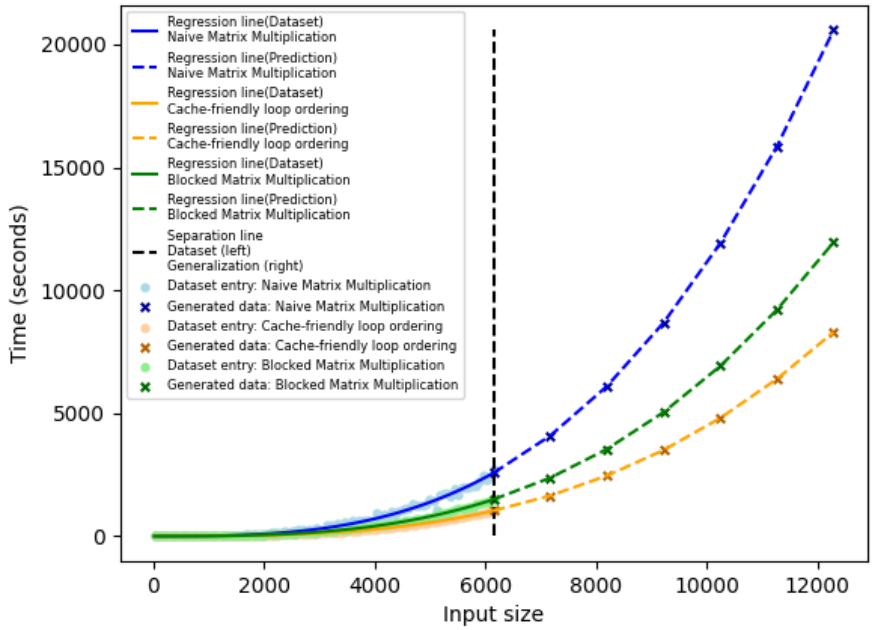


Figure 8: Estimations based on the generalizations of the model tailored on the aquired dataset

5.3.2 Strassen implementation

For a while, we will leave the $O(n^3)$ matrix multiplication algorithms and focus on a new approach. As mentioned before, Strassen proposed a matrix multiplication algorithm with complexity $O(n^{log_2(7)}) \approx O(n^{2.80735})$. We aim at comparing this algorithm with the Cache-friendly loop ordering solution presented in the previous section.

In the traditional approach, without rComplexity analysis, we could not distinguish cases in which Strassen Algorithm could perform worse than any optimized $O(n^3)$ matrix multiplication solution.

Fallacy 5.3.0.1. *Let $Alg1$ an algorithm with the complexity function $f_1 \in \Theta(g_1(n))$ and $Alg2$ an algorithm with the complexity function $f_2 \in \Theta(g_2(n))$. $Alg2$ must perform better than $Alg1$ for any size of input in regard with the specified metric if $\lim_{n\to\infty} \frac{g_2(n)}{g_1(n)} = 0$.*

Fallacy 5.3.0.2. *Adapted version for matrix multiplication:*

Let $Alg1$ an algorithm with the complexity function $f_1 \in \Theta(n^3)$ and $Alg2$ an algorithm with the complexity function $f_2 \in \Theta(n^{2.80735})$. $Alg2$ must perform better than $Alg1$ for any size of input in regard with the specified metric.

Working with traditional complexity do not imply an universal increase in performance for $Alg2$, but an asymptotic comparison, while the fallacies presented in the previous statements assumed an universal behavior.

The correct manifest would be: $Alg2$ must perform better than $Alg1$ *for sufficient large size of input* in regard with the specified metric if $\lim_{n\to\infty} \frac{g_2(n)}{g_1(n)} = 0$.

The collocation *"Sufficient large size of input"* is essential and it means that starting from a range of input size n_0 finite, better performances are obtained using $Alg2$.

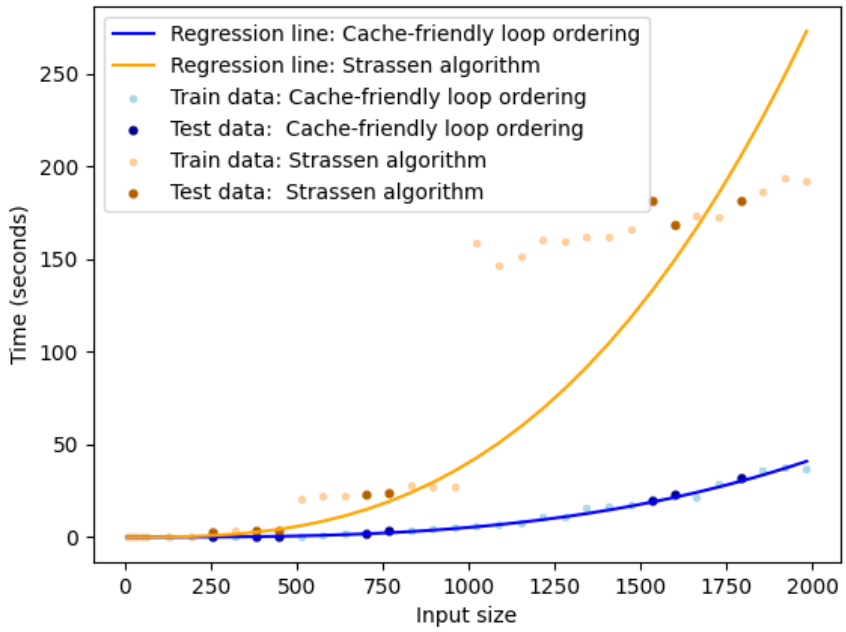


Figure 9: The regression line corresponding to the Cache-friendly loop ordering matrix multiplication algorithm is described by $f(n) = 5.23 \cdot 10^{-9} \cdot n^3$, while the regression line corresponding to the the Strassen algorithm is described by $g(n) = 1.59 \cdot 10^{-7} \cdot n^{2.80}$.
Even if the asymptotic behavior for the Strassen algorithm is desired in comparison with the cubic performance, for finite input size the Cache-friendly loop ordering matrix multiplication algorithm can perform better, despite $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.
The nature of the non-polynomial local behaviour of the Strassen algorithm is based on architecture considerations, such as the overhead introduced by specific function calls, stack manipulations and memory allocation and management computational cost.

Results recorded on a x86_64 Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz with L1d cache: 32K, L1i cache: 32K, L2 cache: 1024K, L3 cache: 22528K, CPU max frequency: 3.9GHz.

Pitfall 5.3.0.1. *Let $Alg1$ an algorithm with the complexity function $f_1 \in \Theta(g_1(n))$ and $Alg2$ an algorithm with the complexity function $f_2 \in \Theta(g_2(n))$ and $\lim_{n\to\infty} \frac{g_2(n)}{g_1(n)} = 0$.*

Even if the $Alg1$ may perform better than $Alg2$ for some cases, the nature of this behavior is superficial and, in general, for regular routines, $Alg2$ will still perform better.

Pitfall 5.3.0.2. *Adapted version for matrix multiplication:*

Even if the Cache-friendly loop ordering matrix multiplication algorithm may perform better than the Strassen algorithm for some cases, the nature of this behavior is superficial and, in general, for regular routines, the Strassen algorithm will still perform better.

The key of the previous pitfalls is the meaning of *"in general, for regular routines"*, because this phrase is extremely context-dependent. We will further calculate the separation point where the performances of Strassen algorithm catch up with (and overtake) the Cache-friendly loop ordering algorithm.

The point is provided by equalizing the two complexity functions:

$f(n) = 5.23 \cdot 10^{-9} \cdot n^3$ and $g(n) = 1.59 \cdot 10^{-7} \cdot n^{2.80}$

The nontrivial solution n_0 is obtained by solving $159 * n_0^{2.8} = 5.23 * n_0^3$, where $n_0 \neq 0$. The solution is $n_0 \approx 25970312 \approx 2.5 \cdot 10^7$.

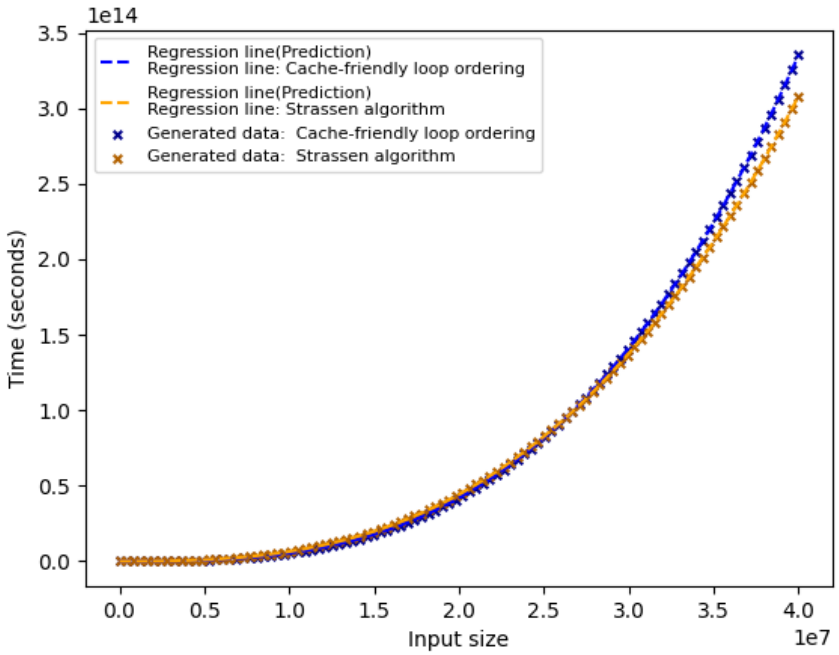


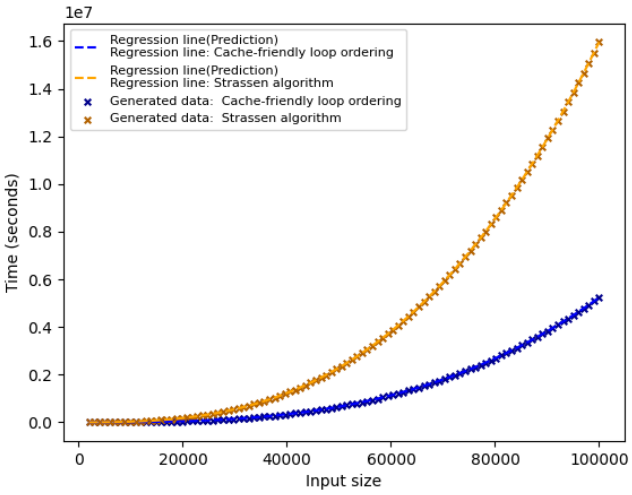
Figure 10: Predictions for the Cache-friendly loop ordering matrix multiplication algorithm and Strassen algorithm based on acquired data. Remark the *"catch up point"* between the two polynomial function for input size $\approx 2.5 \cdot 10^7$.

For any matrix multiplication task with input size greater than $\approx 2.5 \cdot 10^7$ (25 million element matrices), better results will be obtained using Strassen algorithm.

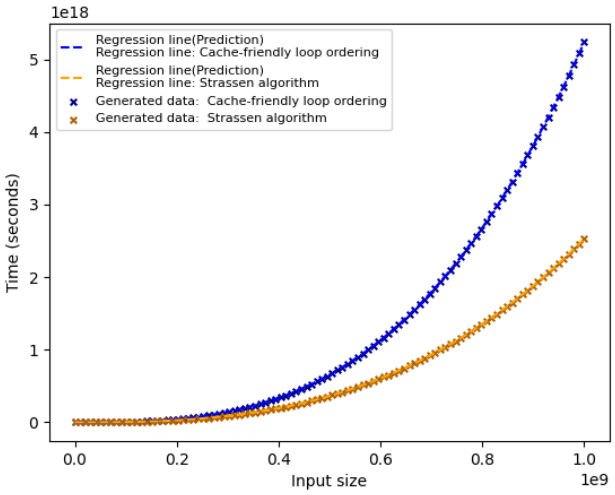
Remark that the total execution time (in seconds, on the tested computing machine) for 25 million element matrices, can be estimated to $g(2.5 \cdot 10^7) \approx f(2.5 \cdot 10^7) = 5.23 \cdot 10^{-9} \cdot (2.5 \cdot$

$10^7)^3 \approx 8.13 \cdot 10^{13}$ seconds. This number $8.13 \cdot 10^{13}$ seconds is the equivalent of around 25 762 centuries.

If by "*regular routines*" was meant multiplying *25 million element matrices* and having the resources to await for 25 762 centuries for the result, than Strassen algorithm is the perfect solution for your problem *(check also memory contraints, discussed in a short time)*. Otherwise, you should refer to a traditional approach.



(a) Comparison between the two algorithms. Input size magnitude $\approx 10^6$



(b) Comparison between the two algorithms. Input size magnitude $\approx 10^9$

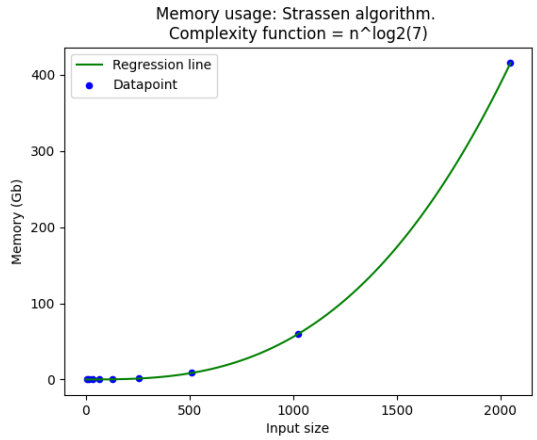
Figure 11: Behaviour of Cache-friendly loop ordering matrix multiplication algorithm and Strassen algorithm based on predictions for different ranges of input.

5.3.3 Memory considerations

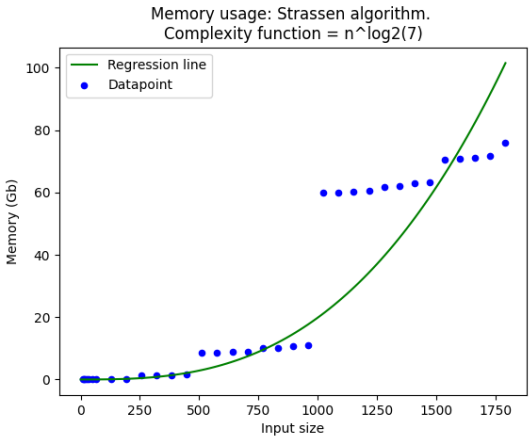
Up to this point, the only metric described in the prediction process of tailoring a complexity function was the **time** complexity. However, this is not the only resource that is important when designing algorithms and computer programs. A close match is represented by the total memory usage or peak memory usage of a computer program during the execution. Even if nowadays, memory is generally large enough to accommodate most of the possible algorithms, there are special situations in which memory management is critical.

The problem with the Strassen memory algorithm is that memory usage does not have a smooth improvement in growth. It varies in steps based on the powers of 2 (the cause is due to the recursion nature of the algorithm and at each iteration dividing in half).

Locally in intervals $[2^k, 2^{k+1} - 1]$, the memory increase in not huge, as the algorithm mentain a n^2 memory increase. However, generally the algorithm perform in a space complexity of $n^{log_2(7)}$.



(a) The total peak allocated memory usage during the execution Strassen algorithm for power-of-2 based input data



(b) The total peak allocated memory usage during the execution Strassen algorithm for all recorded scenarios

Figure 12: Peak memory usage - Strassen. Every double in input size produces a $7x$ increase in peak memory usage. Tailoring an complexity function of type $c \cdot n^{log_2(7)}$ provides a good evaluation. Using this appoximation, the maximum error at prediction is $7x$, while the average error is below $3x$. Power-of-2 based input data provides exact approximation.

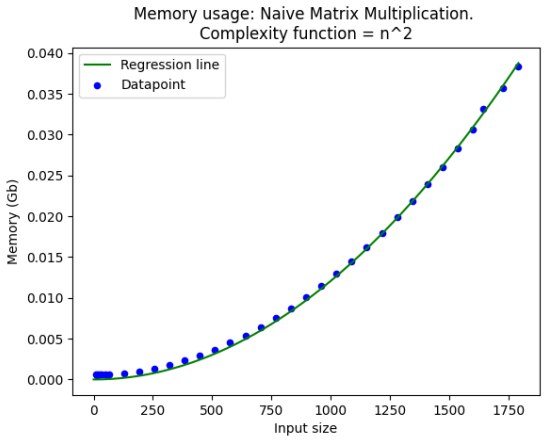
Pitfall 5.3.0.3. *Let $Alg1$ an algorithm with the complexity function $f_1 \in \Theta(g_1(n))$ and $Alg2$ an algorithm with the complexity function $f_2 \in \Theta(g_2(n))$ and $\lim_{n\to\infty} \frac{g_2(n)}{g_1(n)} = 0$.*

Even if the $Alg1$ may perform better than $Alg2$ for some cases, for large input size, $Alg2$ will perform better and we shall thereby **always** *use it.*

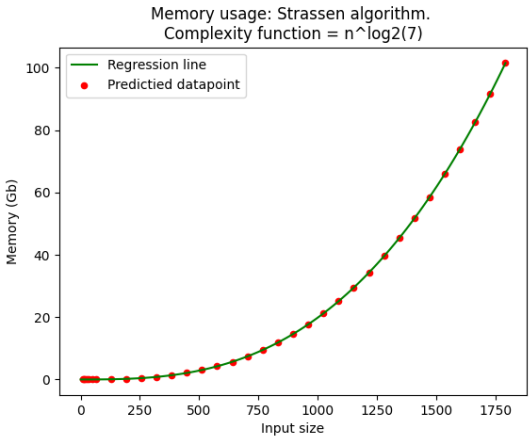
Pitfall 5.3.0.4. *Adapted version for matrix multiplication:*

For regular routines, the Strassen algorithm will perform better than any $O(n^3)$ naive matrix-multiplication algorithm for extremely large input size and we shall thereby use it.

In theory, a smaller time-complexity always produce better asymptotically performances. The problems arise when we address other architectural aspects. Consider that an various algorithms uses memory usage differentiated. In order to perform precisely, a required condition is that the peak memory usage during the execution of the algorithm is at most equal with the total storage capacity of the physical RAM (ignore additional issues such as operating system memory overhead or translation concerns as well as pagination).



(a) The total peak allocated memory usage during the execution of the Cache-friendly loop ordering matrix multiplication algorithm



(b) The total peak allocated memory usage (estimated) for the execution of the Strassen algorithm

Figure 13: Consider the peak memory usage for the last two algorithms analyzed. The behavior can be tailored by the individual associated memory complexity function: $f_{cache\ friendly} = 1.20 \cdot 10^{-8} \cdot n^2$ and $f_{strassen} = 7.89 \cdot 10^{-8} \cdot n^{2.7}$

Recall that the critical point in order to make Strassen algorithm perform better was estimated at around 25 million element matrices. For this value, the peak memory usage can be estimated at **7.43 ZB**. ($7.43 \cdot 10^{12}$ GB) This amount of memory storage can be used to store **5 589 898** centuries of **1080p** digital video content *(at a rate of 1.5GB/hour).* All this, in RAM memory.

Further extensions of this, using sophisticated group theory, are the Coppersmith–Winograd algorithm it is stressed nevertheless that such improvements are only of theoretical interest, since the huge constants involved in the complexity of fast matrix multiplication usually make these algorithms impractical. [**?**].

Pitfall 5.3.0.5. *Never ever use Strassen algorithm.*

All the analyzed data was obtained by analyzing a specific implementation of the Strassen's Matrix Multiplication Algorithm. Not all algorithmic implementation performs the same. There may be optimizing techniques to overpass some issues, especially the deep recursion problems that raised. Also, there exists memory management tricks that substantially decrease the peak memory usage.

In fact, this algorithm is not **galactic** and is used in practice. A galactic algorithm has the property that it is faster than other algorithm for inputs that are sufficiently large, but where sufficiently large is enormous such that that the algorithm is never used in practice. [**?**].

All things summarized, the rComplexity metric provides powerful insights in comparison of different complexity algorithms and such example was observed in this chapter.

5.4 Further reading

The paper prepared additional resources for estimating rComplexity for algorithms with known Bachmann–Landau Complexity at the online codebase resource.

`https://github.com/raresraf/rafMetrics/tree/master/rComplexity/samples`

This resource include analysis for the following algorithms:

- Naive Matrix Multiplication
- Naive Matrix Multiplication (parallel)
- Naive Matrix Multiplication (OpenMP optimized)
- Cache-friendly loop ordering
- Blocked Matrix Multiplication
- Strassen Matrix Multiplication
- NQueens (C)
- NQueens (python)
- Bubble Sort
- Insertion Sort
- Selection Sort
- Heap Sort
- Merge Sort
- Quick Sort
- SENPAI (serial): Simplified Evolutive N-body Processing and Analytics for Integration
- SENPAI (parallel): Simplified Evolutive N-body Processing and Analytics for Integration
- Chess Game
- Web Crawler

Chapter 6

The web of complexities

6.1 Modern means of computing

The world of computing has been vastly changed by modern means of inter-components communication, allowing computing systems to geographically extend at an affordable overhead. This allowed the emergence of new software services as well as new means of programming.

Undoubtedly, the internet has become a massive source of power and with this new technology the information is able to spread really quickly around the world.

While there are many notable achievements in software services that were created over networks, including e-commerce platforms, social media services or e-Gov apps, the focus of this chapter is to embrace the new paradigm of distributed computing and web development on complexity calculus and present means of computing an estimated complexity for complex applications that require new features such as mechanisms for networking handling which have not been analyzed yet.

6.2 HTTP requests and algorithm complexity

There are numerous way of inter-system communication, provided by powerful protocols. Undoubtable, the most used communication protocol between software systems is **HTTP** (Hypertext Transfer Protocol), an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP operates as a request – response protocol in the client – server computing model.

The scenario of development when the client sends a request to the server and the server send an associated response is frequently encountered in web applications development. We will analyze how to integrate this overhead in the complexity model presented.

Consider the following scenario: The results of this paper appears to be interesting and you

may want to re-use them and further analyze them. You built your custom computer program and you want to acquire all practical results provided so far. Note that the development might still be in progress, so you would like that every time you run the software, an up-to-date version of all results should be available. Luckily, these can be achieved via HTTP-requests, as all of the results are stored publicly on GitHub. Therefore, you can use GitHub Developer API, to view the latest published full release for the repository.

```
GET /repos/:owner/:repo/releases/
```

In Python, the code required to achieve this would reduce to an one-liner, using requests, an elegant and simple HTTP library for Python.

```
r = requests.get('https://api.github.com/repos/raresraf/rafmetrics/releases')
```

The response of the server will contain the required information, formatted as a JSON:

```
[{
    "url":
    "https://api.github.com/repos/raresraf/rafMetrics/
       releases/26933384",
    "assets_url":
    "https://api.github.com/repos/raresraf/rafMetrics/
       releases/26933384/assets",
    "upload_url":
    "https://uploads.github.com/repos/raresraf/rafMetrics/
       releases/26933384/assets{?name,label}",
    "html_url":
    "https://github.com/raresraf/rafMetrics/releases/tag/v0.1
       -alpha",
    "id": 26933384,
    "node_id": "MDc6UmVsZWFzZTI2OTMzMzg0",
    "tag_name": "v0.1-alpha",
    "target_commitish": "master",
    "name": "Alpha Baby rafMetrics",
    "draft": false,
    "author": {
        "login": "raresraf",
        "id": 30867783,
        "node_id": "MDQ6VXNlcjMwODY3Nzgz",
        "avatar_url": "https://avatars0.githubusercontent.com
           /u/30867783?v=4",
        "gravatar_id": "",
```

```
        "url": "https://api.github.com/users/raresraf",
        "html_url": "https://github.com/raresraf",
        "followers_url": "https://api.github.com/users/
          raresraf/followers",
        "following_url":
        "https://api.github.com/users/raresraf/following{/
          other_user}",
        "gists_url": "https://api.github.com/users/raresraf/
          gists{/gist_id}",
        "starred_url":
        "https://api.github.com/users/raresraf/starred{/owner
          }{/repo}",
        "subscriptions_url":
        "https://api.github.com/users/raresraf/subscriptions"
          ,
        "organizations_url": "https://api.github.com/users/
          raresraf/orgs",
        "repos_url": "https://api.github.com/users/raresraf/
          repos",
        "events_url": "https://api.github.com/users/raresraf/
          events{/privacy}",
        "received_events_url":
        "https://api.github.com/users/raresraf/
          received_events",
        "type": "User",
        "site_admin": false
    },
    "prerelease": true,
    "created_at": "2020-05-27T08:43:19Z",
    "published_at": "2020-05-27T08:48:24Z",
    "assets": [],
    "tarball_url":
    "https://api.github.com/repos/raresraf/rafMetrics/tarball
      /v0.1-alpha",
    "zipball_url":
    "https://api.github.com/repos/raresraf/rafMetrics/zipball
      /v0.1-alpha",
    "body": "`rafMetrics` v0.1-alpha release"
}]
```

We aim to provide a solution to estimate the time-complexity for the Python-snippet that makes an HTTP request and theoretically, in the traditional complexity model, the operation

should take constant time (computing well-defined finite header, simple system call to networking drivers, constant number of steps): $O(1)$. However, it is intuitive that this operation will take much more time to completely execute rather than an instruction such as $xor\ eax,\ eax$. And the r-Complexity excels at this kind of comparisons w.r.t. discrete analysis.

Therefore, we aim to create a mapping between any call $r = requests.get(url)$ to an $O_1(X)$, where X is the estimated overhead introduced by this procedure call. In Python, this can be achieved by $r.elapsed.total\_seconds()$. For instance, the previous example has a total elapsed time equal with 0.467 seconds, i.e. the associated r-Complexity class would therefore be $O_1(0.467 \cdot CPS)$ (CPS = cycle per second).

Of course, this value will not be unique, as each run may provide different outcomes. Therefore, for better estimations we created WebMonitoring tool, an easy-to-use empiric estimator for computing projections for the associated r-Complexity classes of HTTP requests. The platform is described in-detail in the next chapter.

6.3 Empirical Big r-Complexity estimations for web resources

The current work propose the following associations between Big r-Complexity estimations for web resources and recorded entries:

1. **Big r-O**notation for a given request with $O_1(highest)$
2. **Big r-Omega** notation for a given request with $\Omega_1(lowest))$
3. **Big r-Theta** notation for a given request with $\Theta_1(avg)$ or $\Theta_1(median)$ or $\Theta_1(max(avg,\ median))$

Note that these results are empirical and might significantly differ.

Resouce (GET Requests)	**Avg (s)**	**Lowest (s)**	**Median (s)**	**Highest (s)**
https://github.com/raresraf/rafMetrics	0.68	0.43	0.49	2.99
https://google.com	0.07	0.06	0.06	0.29
E-commerce website 1 (AliExpress)	0.3	0.2	0.23	0.85
E-commerce website 2 (Amazon)	0.12	0.1	0.12	0.14
E-commerce website 3 (eMag)	0.42	0.37	0.41	0.5
https://www.piday.org/	0.87	0.78	0.87	0.98

Table 6.1: Different metrics for computing Big r-Complexity estimations for the presented web resources. Results acquired during May 2020.

All the results are obtained using the rafMetrics platform described in the following chapter.

Website Total Loading Time	**Avg (s)**	**Lowest (s)**	**Median (s)**	**Highest (s)**
https://github.com/raresraf/rafMetrics	3.12	2.11	2.89	5.88
https://google.com	2.41	1.12	2.5	4.12
E-commerce website 1 (AliExpress)	6.12	4.3	6.05	11.02
E-commerce website 2 (Amazon)	3.4	2.13	3.36	5.49
E-commerce website 3 (eMag)	6.3	4.34	5.99	11.2
https://www.piday.org/	8.32	2.33	6.51	12.4

Table 6.2: Different metrics for computing Big r-Complexity estimations for the presented webpages. Results acquired during May 2020.

The analysis can be done using other metrics such as Response Size of the request, custom defined efficiency or throughput.

Chapter 7

The Platform

7.1 Introduction to rafMetrics platform

This theoretical work includes a proof-of-concept software stack including various metrics evaluation for classic and modern computer programs. The software stack is further referred as the "Platform".

The metrics that require data acquisition are based on web crawlers (interrogations as described in WebMonitoring tool), experimental data (gathered by computing and estimating running time computer algorithms for various input size in cloud-based systems) or symbolic calculus(for computing rComplexity calculus and calculating partial derivatives).

Furthermore, a ML-based system is used for estimating rComplexity in the case when the theoretical mapping between the algorithm's function and the rComplexity Class is not known. For better performances, it should be inputted with the classic Big-Theta(for asymptotic behavior) Class or an acceptable classic Big-O Class approximation. However, this platform can approximate a good fitting for algorithms with unknown classic asymptotic behavior.

The process of automatically tailoring a suitable rComplexity Class is provided on-demand, using rafComputing Tool.

7.2 Codebase

All the code can be accessed and used via GitHub at https://github.com/raresraf/rafMetrics.

rafMetrics is licensed under the MIT License: a short and simple permissive license with conditions only requiring preservation of copyright and license notices.

Licensed works, modifications, and larger works may be distributed under different terms and without source code.

An easy-to-use, dockerized implementation, can be found at DockerHub within raresraf/ repositories.

7.3 Technologies

7.3.1 Backend

The main Platform engine is written in Python3, an interpreted, high-level, general-purpose programming language.

The WebMonitoring tool and the Login service have been created using Flask framework for a quick and easy API definition. The connection between the Platform and the mySQL database is provided by the official mySQL connectors provided in the mysql-connector-python packages.

For WebMonitoring tool we use thind-party browsers and software applications, including Chrome Driver, a Selenium server or Browsermob-proxy.

The repositories contains few easy deployment scripts that are written in Python or Shell Scripting.

Some mySQL functions and procedures are generated using the build-in rendering Python options.

rafComputing accommodate both an API definition under Flask technology, while the actual Engines for computing and estimating rComplexity are developed using NumPy library.

For time bug-free code a third-party library is used, arrow. This is a Python library that offers a sensible and human-friendly approach to creating, manipulating, formatting and converting dates, times and timestamps.

For monitoring BE an open source analytic & monitoring solution for the database is desired. Such an implementation is Grafana.

7.3.2 Frontend

The frontend is implemented using React – a JavaScript library for building user interfaces. For for managing application state Redux library is used.

The platform UI is based on Flatlogic Template: React Material Admin — Material-UI Dashboard. The application can be built using default Node package manager.

7.3.3 Deployment

Each component is containerized using Docker technologies which assures a secure build and unbounded sharing options.

In order to perform a deployment of the Platform as an assembly, few orchestration systems can be used, including Docker Swarm or Kubernetes.

A Docker Swarm is a group of either physical or virtual machines that are running the Docker application and that have been configured to join together in a cluster.

Kubernetes is a vendor-agnostic cluster and container management tool, open-sourced by Google in 2014.

7.4 Networking

Networking is performed inside the orchestrator using an overlay network managed by Docker Swarm or Kubernetes services with static ClusterIPs and NodePorts.

Docker's overlay networks implementations connect multiple Docker machines and allow swarm services to communicate with each other, facilitating communication between two containers running on different docker swarm nodes.

In Kubernetes, there are four distinct networking problems to address [**?**]:

- Highly-coupled container-to-container communications: this is solved by pods and localhost communications.
- Pod-to-Pod communications: every Pod gets its own IP address
- Pod-to-Service communications: this is covered by Kubernetes services.
- External-to-Service communications: this is covered by Kubernetes services.

The default node port range for Kubernetes is 30000 - 32767.

7.5 Architecture

7.5.1 Architecture overview

The platform was built on a microservice architecture, which is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

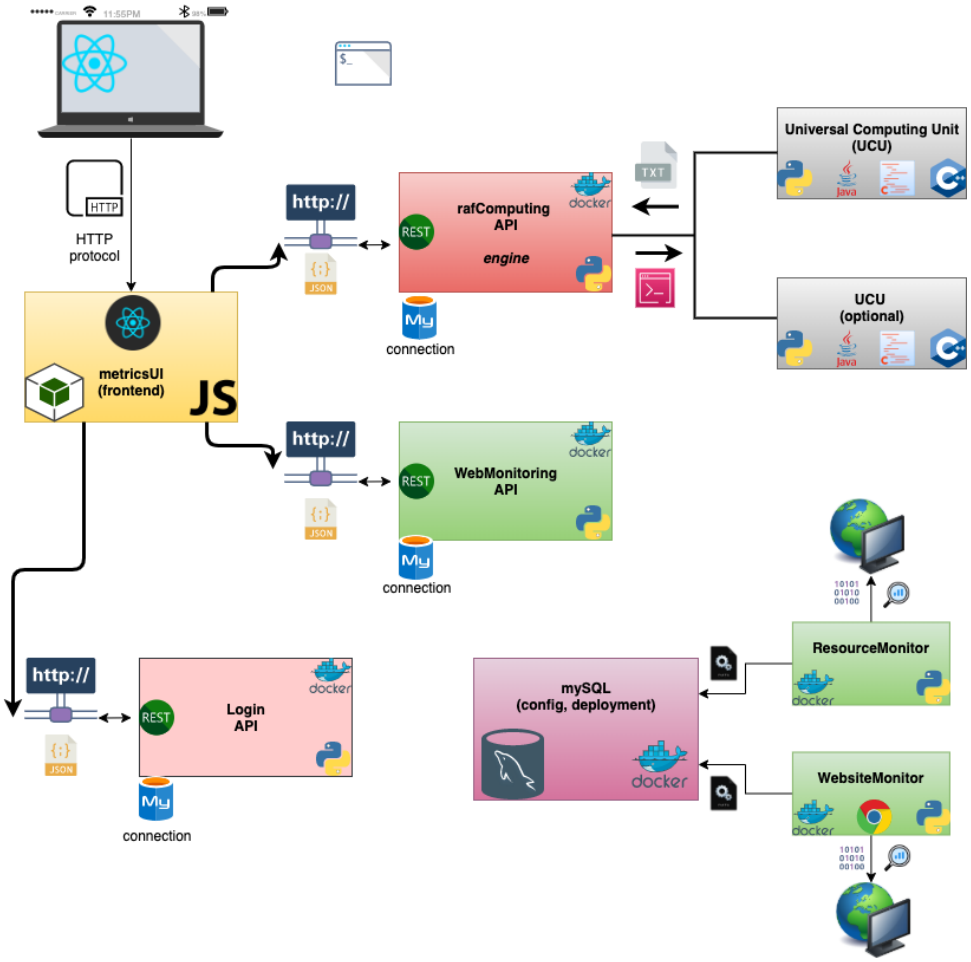


Figure 14: An overview of the architecture of the rafMetrics' platform

7.5.2 Components

The platform is made up of the following components:

rafComputing

The ML-based tool implementing the process of automatically tailoring a suitable rComplexity Class for an algorithm.

A suitable rComplexity Class set is defined by the group of mathematical functions similar in magnitude with $g(n)$ in the study of asymptotic behavior. A set-based description of this group can be expressed as:

$$\Theta_1(g(n)) = \{f \in \mathcal{F} \mid \forall c_1, c_2 \in \mathbb{R}_+^* \; s.t. c_1 < 1 < c_2, \exists n_0 \in \mathbb{N}^* \\ s.t. \;\; c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \;\; , \;\; \forall n \geq n_0\}$$

WebMonitoring

A tool for monitoring multiple network resources and websites. Gather data by periodically monitoring specific resources and websites and stores results in database.

The following metrics are defined inner WebMonitoring Tool:

- Resource Crunch (known as Resource, part of WebMonitioring Tool): Average Response Time (daily, weekly, monthly, custom), Average Response Size, Lowest, Medium, Median, Highest metric time, Efficiency Metric
- Website Crunch (known as Website, part of WebMonitioring Tool): Average Response Time (daily, weekly, monthly, custom), Average Response Size (daily, weekly, monthly, custom)
- Statistics Resource Manager: Total time, Total number of Requests, Average Time, Average Size, Standard deviation acquired for last 24 hours or all time.
- Statistics Website Manager: Total time, Total number of Requests, Average Time, Average Size, Standard deviation acquired for last 24 hours or all time.

ResourceManager

Monitors all resources by periodically (timer set default at 1 hour interval) sending requests to existing resources. Store simple metrics like total time or total requests answer as entries in DB.

WebsiteManager

Monitors all websites by periodically (timer set default at 1 hour interval) generating a HAR (HTTP-Archieve data performance file) for loading metrics corresponding to a website, with Chrome using Browsermob-Proxy. Also parse and store valuable insights resulted from the HAR file into DB. The service uses [speedprofile](https://github.com/parasdahal/speedprofile) engine.

WebMonitoring API

Provide an API for interrogating useful metrics from DB.

The routes defined can be checked in the documentation existing in the codebase.

Login

Backend implementation to provide a simple authentication, registration and management for users inside rafMetrics platform.

The routes defined can be checked in the documentation existing in the codebase.

KubernetesConfig

Keeps track of all k8s settings.

DockerConfig

Keeps track of all Docker Compose/Docker Swarm settings and configurations.

MySQL

Database used to store persistent data required by Login and WebMonitoring. All relations are kept in **Boyce-Codd Normal Form**.

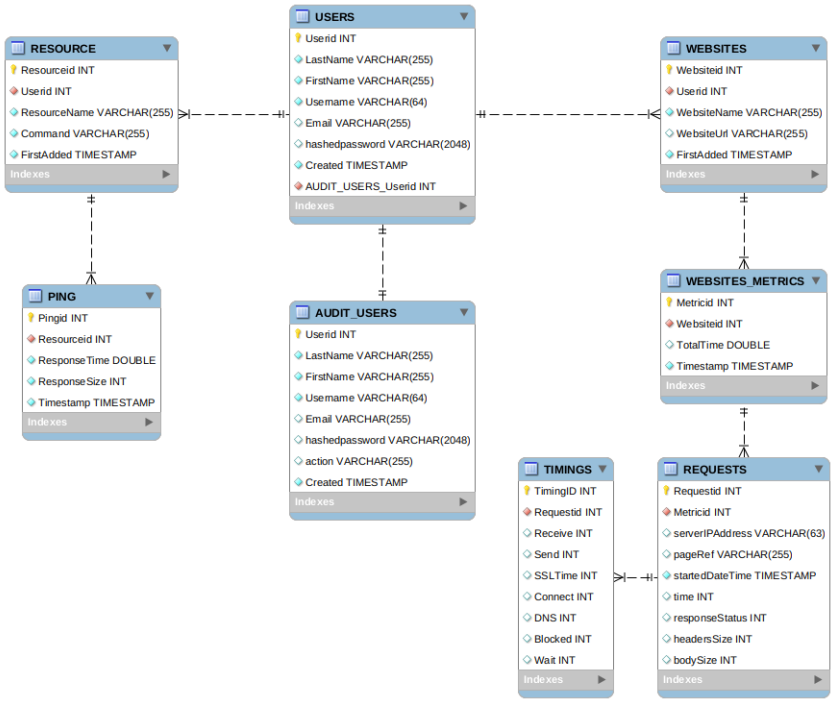


Figure 15: SQL table hierarchy used in WebMonitoring Tool

The table semantics can be checked in the documentation existing in the codebase.

deploy_repo.sh

Simple script to ensure dockerize and deployment in Kubernetes for all backend components.

metricsUI

Frontend implementation of rafMetrics platform based on Flatlogic Template: React Material Admin — Material-UI Dashboard. This component is responsable for rendering UI part of WebMonitoring Tool.

Grafana

Grafana ships with a built-in MySQL data source plugin that allow to query any visualize data from a MySQL compatible database.

Chapter 8

Chess-game analysis

8.1 The game of chess

Chess is a two-player turn-based strategy game played on a checkered board with bounded dimensions: 64 squares arranged in an 8x8 grid, where each player has a set of initial pieces arranged in a specific order aiming to conquer an specific opponent piece: The King.

In game-theory, Chess is classified in the cluster of games with perfect information in which the players move alternately and the game is not subject to randomization.

The history of chess was firstly documented around 6th Century, although the earliest origins are ambiguous. It is believed that the game's predecessor was originated in India, from where the game spread to Persia. After the Arabic occupation, the game was spread through the Muslim world and to Southern Europe, where it evolved into the current form in the 15th Century [**?**].

Figure 16: The Lewis chessmen are a group of distinctive 12th-century chess pieces, along with other game pieces, most of which are carved from walrus ivory. Present location: British Museum, London. Image courtesy of Wikipedia Commons.

8.2 Computing chess

8.2.1 Zermelo's theorem

Theorem 8.2.1. *Zermelo's theorem is a game-theory theorem regarding finite two-person games of perfect information in which the players move alternately and the game is not subject to randomization. It suggests that if the game cannot end in a draw, then one of the two players must have a winning strategy (i.e. force a win).*

*An alternate statement is that for a game meeting all of these conditions except the condition that a draw is not possible, then either the first-player can force a win, or the second-player can force a win, or both players can force a draw. [**?**].*

Remark. *As the chess is a finite two-person games of perfect information in which the players move alternately and the game is not subject to randomization, Zermelo's Theorem can be applied for this game. It states that either White can force a win, or Black can force a win, or both sides can force at least a draw.*

With this background of theoretical work, we can state that there must be a perfect algorithm for chess, at least for one of the two players.

Remark. *All endgames of 6 pieces or less have been perfectly solved.*

Remark. *Previous results show that the game of Losing Chess is a win for White. [**?**]. The "losing-winning" move is* **e3** *for White.*

8.2.2 Chess complexity using Bachmann-Landau notations

This subsection will present another example of inappropriate use of Bachmann-Landau asymptotic notations for algorithms that do not generalize well. As previously stated, a perfect algorithm for chess exists, even though the state space that it would have to search would be huge.

Due to finite-bounds of the game and the existence of the 50-move rule, which allows either player to claim a draw if 50 or more moves take place without movement of a pawn or a piece-capture, based on FIDE-game rules, the longest chess game could be up to 4851 moves. A wide estimation would be counting the total number of possibilities for a the player to move one of (at most) 16 pieces: 8 pawns, each with at most 3 moves, 2 rocks each with at most 14 moves, 2 knights 8 each with at most, 2 bishops, each with at most 14 moves, 28 for a queen and 8 for a king, for a total of 132 possible moves. Definitely these are just hypothetical situation analyzing the worst-case scenario, as in real games the possibilities are far smaller.

Hence, the total number of chess games would be at most 132^{4851}. (a more realistic estimation would be 10^{500})

As a finite game can be simulated in constant time, the above estimation, translated in Bachmann-Landau notations for complexity classes, this means that the perfect algorithm for chess should perform in $O(132^{4851})$. Using the properties of Big-O complexity class, we can state that this algorithm will perform in constant time, with $O(1)$ complexity as this number of total number of chess, regardless how big it is, is still a constant.

In this situation, the Bachmann-Landau asymptotic notations did not provide useful information and the reason is simple: these notations were developed for asymptotic-scaling problems and algorithms, w/o awareness of discrete values. Even though in most cases these notations were helpful, this is probably not the case in this scenario.

8.2.3 Previous work

Claude Shannon had studied the implications of a brute force solution for solving chess back in the 1950 [**?**], when he introduced the **Shannon number**, a conservative lower bound of the game-tree complexity of chess. The purpose was to validate that any perfect chess algorithms based on brute-force are impractical.

The Shannon number proposed back in 1950 was 10^{120}, taking into consideration a typical game of 80 moves at a rate of 10^3 possibilities for each pair of white-black moves.

Previous work of Claude Shannon was building the foundation of information theory with a landmark paper (1948), *"A Mathematical Theory of Communication"*.

Further work showed that, based on an average branching factor of 35 and an average game length of 80, the lower bound for the chess game-tree is around 10^{123}. Victor Allis: [**?**]

8.3 Chess in r-Complexity

As we previously stated, the perfect algorithm for chess is part of the $O(1)$ complexity class, as its input values are finite-bounded. Thus, the associated r-Complexity class would be $O_1(c)$, where c in a real finite constant. A human-driven calculus of r-Complexity is not feasible, as there are various run-time aspects that are difficult to be taken into consideration and an exact calculus would imply a even greater effort than solving straightforward the chess problem. Thus, we propose an automatic estimation for this algorithm, that has its Bachmann–Landau Complexity known.

The first step was acquiring data on few game-simulation. Using gym framework, we tracked the time-complexity for various number of episodes with different number of steps per each episodes.

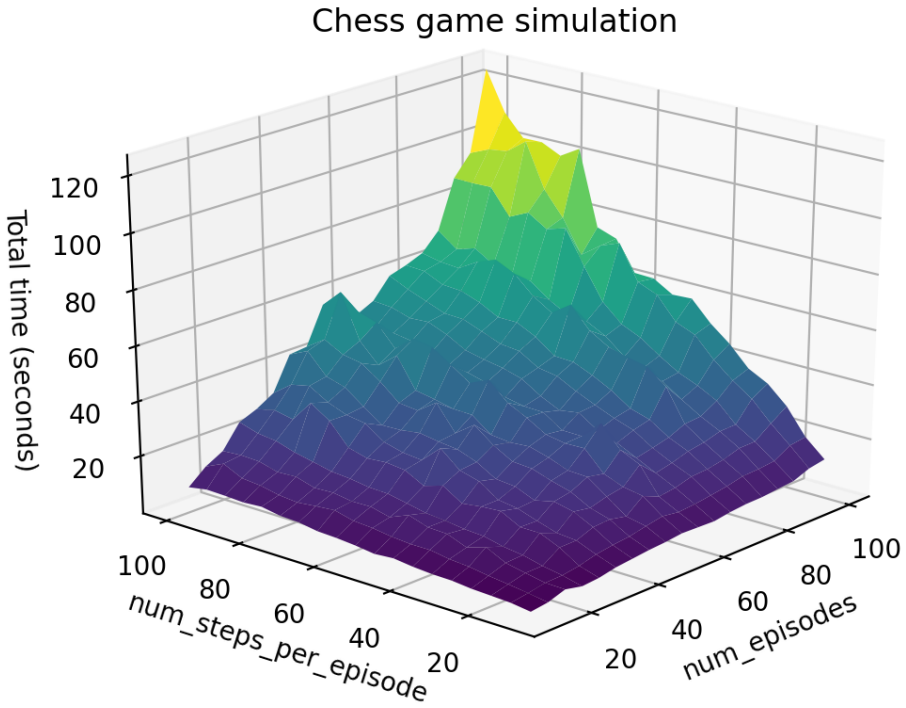


Figure 17: The plot shows the total computing time based on the variation of parameters corresponding to the total number of games and the average length of a game. Results gathered on a 2.3 GHz Intel Core i5 processor using a serial implementation in Python and Gym framework.

Using an average game length of 80, the chess-solving problem becomes a one-parameter problem that involves the total number of episodes to be generated. This value is lower-bounded by the value of 10^{123}. A brute-force solution for this algorithm would act almost linearly in terms of number of episodes to be generated.

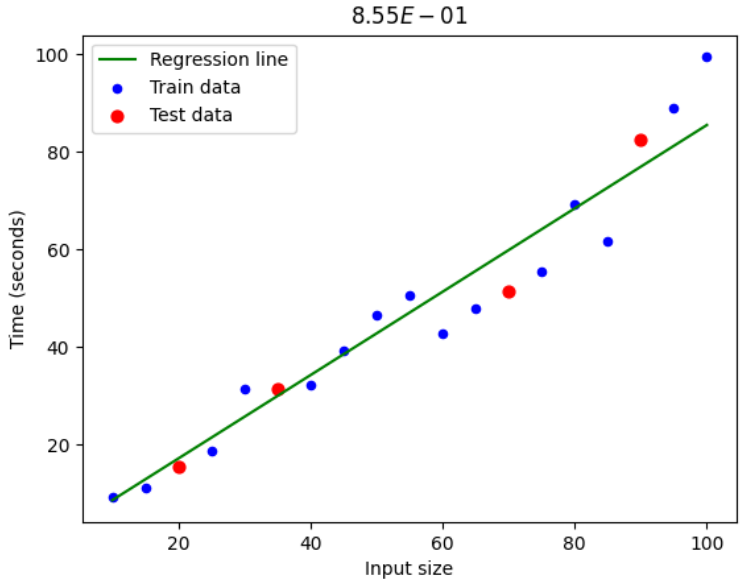


Figure 18: The relationship between the total time required to generate a specific number of games appears to be linear (actual results obtained for an average game length of 80)

Of course, many optimization can reduce the total time by even two orders of magnitude. Regardless of the optimization process, for an input of 10^{123}, the total estimated time using this algorithm would be $0.855 \cdot 10^{123}$ seconds and thus the r-Complexity would be $O_1(0.855 \cdot 10^{123} \cdot HZ)$, where $HZ \approx 2.3 \cdot 10^9$.

Such a huge time limit is a result of the greatness of the lower bound for chess. Recall that this is by over 40-orders of magnitude greater that the total estimated number of atoms in the universe.

8.3.1 The dream of the perfect algorithm

All means of computing in 2020 are at an enormous gap from what it would be needed in order to find the perfect algorithm of chess using brute-force solutions.

Based on the estimated r-Complexity (i.e. $O_1(0.855 \cdot 10^{123} \cdot HZ)$, where $HZ \approx 2.3 \cdot 10^9$) we present few scenarios describing what it would take to compute:

i **Scenario 1**
Assume that each atom in the universe is the state-of-the-art computing core of a modern processor, that operates at a frequency of 5GHz. Assume that the perfect algorithm of chess defies Amdahl's Law by assuming a theoretical unbounded speedup in the latency of the execution. Assume we have 0 latency between inter-process communication. So far, we built an system consisting of 10^{80} (this is, the estimate the total number of fundamental particles in the observable universe) computing units

operating at $HZ_0 \approx 5 \cdot 10^9$, with zero latency that needs to solve an algorithm with associated complexity of $O_1(0.855 \cdot 10^{123} \cdot 2.3 \cdot 10^9)$. That is, assuming perfect distribution and zero overhead, each unit would require approx 10^{42} seconds. This is far greater than any estimation of the universe lifespan.

ii **Scenario 2**

Moore's law might come to rescue the hope. Ignoring any physical consideration of the minimum MOSFET scaling process, one might assume that the law might scale indefinitely (even though even Gordon Moore himself foresaw that the rate of progress would eventually reach saturation).

Assume that, the number of transistors in an integrated circuit doubles about every two years and this results in doubling of the transistors implies a doubling in the computing power as well. (ignore the total amount of energy spent and all other considerations). Say that, if today a computing unit would solve the problem in $0.855 \cdot 10^{123}$ seconds, in two-year times, the new computing unit would solve the problem in $\frac{0.855 \cdot 10^{123}}{2}$. Similarly, in four-year times, the new unit would solve the problem in $\frac{0.855 \cdot 10^{123}}{2^2}$ and in six-year times, it would solve the problem in $\frac{0.855 \cdot 10^{123}}{2^3}$. This decrease in time is logarithmic and, theoretical, it reaches after $2 \cdot log_2(0.855 \cdot 10^{123})$ years to a point in which the total time for solving chess would be just under 1 second.

That is, if Moore's law would scale indefinitely, after 816 years, we would be able to solve by brute-force, the game of chess. However, this would also mean that the system would require 2^{408} more transistors than the current number of transistors in modern computers (tens of billions), reaching a peak number of transistors of 10^{132}.

Unless each fundamental particle in the observable universe could act as 10^{52} MOSFET-transistors, this scenario is impracticable.

iii **Scenario 3**

A new, groundbreaking technology appeared and computing is no longer an issue. The algorithm would still require to store at least 10^{123} games. Assume the most efficient compression, even though unfeasible, each unique position can be stored by only one brand new memory unit. This would mean that each atom in the universe should be able to store at least 10^{43} storing units. This appears to be pretty difficult.

We could further extend our philosophical discussion with many more scenarios, but the point is clear. Perfectly solving the game of chess is a problem far to complicated. And yet, we, as humans, with limited computing power, can naturally play the game of chess charmingly well.