Metrics for evaluating the performance and complexity of computer programs

Rares Folea * **advisor: Prof. Emil-Ioan Slusanschi** *

* *Faculty of Automatic Control and Computers, University Politehnica of Bucharest*
(e-mail: rares.folea@stud.acs.upb.ro)

Abstract
This document presents a new asymptotic notations that offer better complexity feedback for similar programs, providing subtle insights even for algorithms that are part of the same conventional complexity class

Keywords: metrics, complexity, C, Python, benchmark, **M2**, **ECM2**

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 1.1. 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 1.1. 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. **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. 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 1.3. **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 1.3. 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 1.4. **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 1.4. 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 1.5. **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 1.5. 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 1.6. **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 1.6. 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 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 1.7. 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 1. 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 1.7.1. Addition in **Big Theta** using the relaxed notation:

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

Lemma 1.8. 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 1.8.1. Addition in **Big O** using the relaxed notation:

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

Lemma 1.9. 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 1.9.1. Addition in **Big Omega** using the relaxed notation:

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

Lemma 1.10. 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 1.10.1. Addition in **Small O** using the relaxed notation:

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

Lemma 1.11. 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 1.11.1. Addition in **Small Omega** using the relaxed notation:

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

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 2. 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 2.1. **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. **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 2.3. **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 2.4. **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 3. 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 2.5. **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 4. 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 2.1. 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 2.2. 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. 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 2.4. 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 2.5. 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 2.6. Reflexivity in $rComplexity$: - Big r-Theta notation

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

Theorem 2.7. Reflexivity in $rComplexity$: - Big r-O notation

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

Remark 5. 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 6. 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 2.8. Transitivity in $rComplexity$ - Big r-Theta notation:
$f \in \Theta_r(g(n)), g \in \Theta_{r'}(h(n)) \Rightarrow f \in \Theta_{r \cdot r'}(h(n))$

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

Theorem 2.9. 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 2.10. 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 2.11. 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 2.12. 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 2.13. **Symmetry in $rComplexity$:**
$f \in \Theta_r(g(n)) \Rightarrow g \in \Theta_{\frac{1}{r}}(f(n))$

Theorem 2.14. **Transpose symmetry in $rComplexity$:**
$f \in \mathcal{O}_r(g(n)) \Leftrightarrow g \in \Omega_{\frac{1}{r}}(f(n))$

Theorem 2.15. **Transpose symmetry in $rComplexity$:** $f \in o_r(g(n)) \Leftrightarrow g \in \omega_{\frac{1}{r}}(f(n))$

Theorem 2.16. **Projection in $rComplexity$:**
$f \in \Theta_r(g(n)) \Leftrightarrow f \in \mathcal{O}_r(g(n)), f \in \Omega_r(g(n))$

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 2.17.

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 1. $f \in \Theta_r(g(n)) \Rightarrow \forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < r < c_2, \exists n_0 \in \mathbb{N}^*$
$s.t. \ \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ , \ \forall n \geq n_0$
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 2.17.1.

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

.

Theorem 2.18. **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 2. Is similar with the proof for Big r-Theta, with the inequalities becoming $f(n) \leq c \cdot g(n)$.

Corollary 2.18.1. The following conversion relationship arises:

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

Theorem 2.19. **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 3. Is similar with the proof for Big r-Theta, with the inequalities becoming $f(n) \geq c \cdot g(n)$.

Corollary 2.19.1. The following conversion relationship arises:

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

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 2.20. 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 2.21. 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 2.22. 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 2.23. **If** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow h \in \Theta_q(g)$.

  Proof 4. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow \lim_{n\to\infty} \frac{f'(n)}{g'(n)} = 0 \Rightarrow \lim_{n\to\infty} \frac{g'(n) + f'(n)}{g'(n)} = 1$ 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 2.24. **If** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow h \in \Theta_r(f)$.

  Proof 5. 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 2.25. **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 6. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = t \Rightarrow \lim_{n\to\infty} \frac{f'(n)}{g'(n)} = t \cdot \frac{r}{q} = t' \Rightarrow \lim_{n\to\infty} \frac{g'(n) + f'(n)}{g'(n)} = t' + 1$ 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 2.26. **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 2.27. **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 2.28. **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 2.29. **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 2.30. **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 2.31. **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 2.32.
  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow \Theta_r(f) + \Theta_q(g) = \Theta_q(g)$$
- **Big r-O**:
  Lemma 2.33.
  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow \mathcal{O}_r(f) + \mathcal{O}_q(g) = \mathcal{O}_q(g)$$
- **Big r-Omega**:
  Lemma 2.34.
  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow \Omega_r(f) + \Omega_q(g) = \Omega_q(g)$$

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

- **Big r-Theta**:
  Lemma 2.35.
  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow \Theta_r(f) + \Theta_q(g) = \Theta_r(f)$$
- **Big r-O**:
  Lemma 2.36.
  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow \mathcal{O}_r(f) + \mathcal{O}_q(g) = \mathcal{O}_r(f)$$
- **Big r-Omega**:
  Lemma 2.37.
  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow \Omega_r(f) + \Omega_q(g) = \Omega_r(f)$$

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

- **Big r-Theta**:
  Lemma 2.38.
  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow \Theta_r(f) + \Theta_q(g) = \Theta_r(f + \frac{r}{q} \cdot g)$$
- **Big r-O**:
  Lemma 2.39.
  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow \mathcal{O}_r(f) + \mathcal{O}_q(g) = \mathcal{O}_r(f + \frac{r}{q} \cdot g)$$
- **Big r-Omega**:
  Lemma 2.40.
  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow \Omega_r(f) + \Omega_q(g) = \Omega_r(f + \frac{r}{q} \cdot g)$$

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

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 3.1. **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. **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 3.3. **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 3.4. **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 3.5. **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 3.1. 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 3.2. 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. 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 3.4. 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 3.5. 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 Addition properties

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

Theorem 3.6. 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** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow h \in \Theta_1(g)$.
- **If** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow h \in \Theta_1(f)$.
- **If** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow h \in \Theta_1(f+g)$.

Theorem 3.7. 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** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow h \in \mathcal{O}_1(g)$.
- **If** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow h \in \mathcal{O}_1(f)$.
- **If** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow h \in \mathcal{O}_1(f+g)$.

Theorem 3.8. 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** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow h \in \Omega_1(g)$.
- **If** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow h \in \Omega_1(f)$.
- **If** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = t,\ t \in \mathbb{R}_+ \Rightarrow h \in \Omega_r(f+g)$.

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

- **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 3.10. 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 3.11. 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 7. All proofs are based on the particularizing $r = 1$ and rewrite the addition properties from rComplexity.

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

Remark 8. 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 3.7. 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 9. 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 10. 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.

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 11. 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.

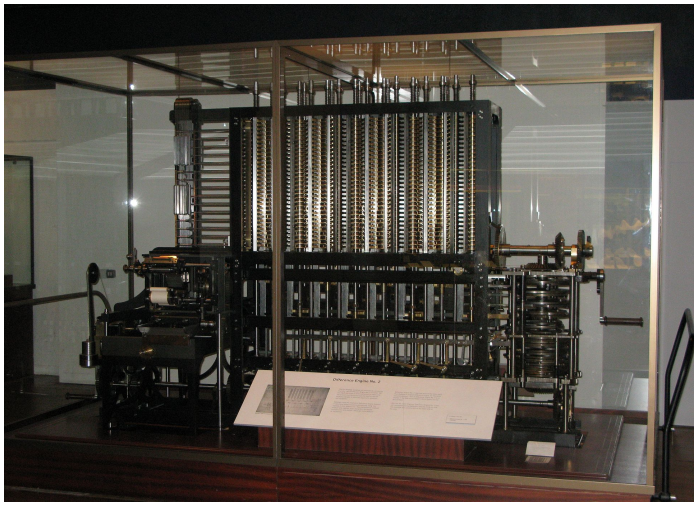

Figure 1. An implementation built from Babbage's design of a difference engine exhibited at The London Science Museum

.

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 4.1. **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 4.2. **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. **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 4.4. **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 12.

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

Definition 4.5. 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 4.1. 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 8. $\lim_{n\to\infty} \frac{g_1(n)}{g_1'(n)} = 0 \Rightarrow g_1(n) < g_1'(n)\ \forall n \geq n_0 \Rightarrow f(n) < f'(n)\ \forall n \geq n_0'$

Definition 4.6. 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:

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

Lemma 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 9. $\lim_{n\to\infty} \frac{g_1(n)}{g_1'(n)} < 1 \Rightarrow g_1(n) < g_1'(n)\ \forall n \geq n_0 \Rightarrow f(n) < f'(n)\ \forall n \geq n_0'$

Lemma 4.3. 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 10. $\lim_{n\to\infty} \frac{g_1(n)}{g_1'(n)} > 1 \Rightarrow g_1(n) > g_1'(n)\ \forall n \geq n_0 \Rightarrow f(n) > f'(n)\ \forall n \geq n_0'$

Definition 4.7. 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. 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 11. $\lim_{n\to\infty} \frac{g_1(n)}{g_1'(n)} = \infty \Rightarrow g_1(n) > g_1'(n)\ \forall n \geq n_0 \Rightarrow f(n) > f'(n)\ \forall n \geq n_0'$

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. 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 4.6. 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 13. 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 4.7. 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 4.7.1. 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 14. 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 15. 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.

s

Remark 16. 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 4.8. 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 4.8.1. 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 17. 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 18. 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 $n_0 = n_{min}, n_f = n_{max}$.

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.

5.2 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.

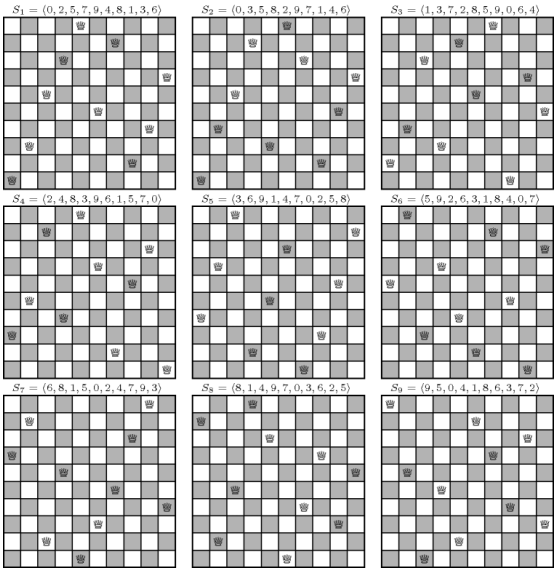

Figure 2. Possible solutions for $n = 10$

This subsubsubsection 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 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] ← 1
9:             QueenProblem(currentMatrix, row + 1, 1)
10:            currentMatrix[row][i] ← 0 //BackTracking Step
11:        end if
12:    end for
13: end procedure
14: procedure Main(n)
15:    read n
16:    currentMatrix.initiate()
17:    currentMatrix.setrows(n)
18:    currentMatrix.setcolumns(n)
19:    currentMatrix ← [0 0 ... 0 0
                        0 0 ... 0 0
                            ...
                        0 0 ... 0 0
                        0 0 ... 0 0]
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 $c_{readMemory} = O_1(100)$.

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
```

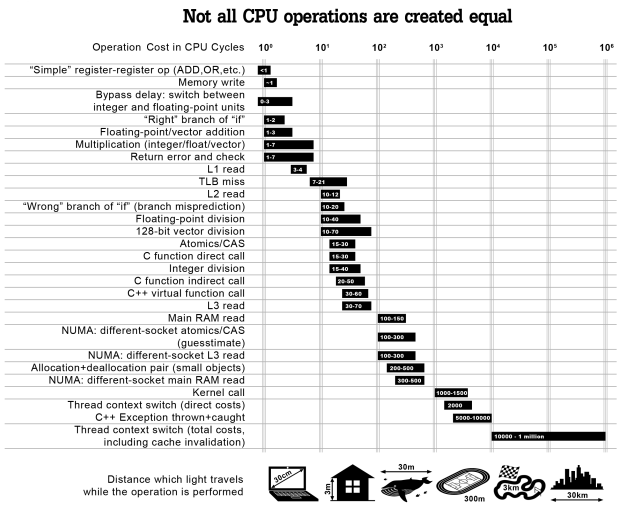

Figure 3. Averages of the Costs (in CPU Clock Cycles) for the fundamental assembly instructions **?**

```
.L5:
mov eax, DWORD PTR [ebp-4]
imul edx, eax, 400 # hardcoded 20x20 matrix
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** $i \leq 0$ **then**
6: break
7: **end if**
8: $i \leftarrow i - 1$
9: **end for**

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

1: **for** j=column..0 : increment -1 **do**
2: **if** currentMatrix[i][j] $= 1$ **then**
3: **return** FALSE
4: **end if**
5: **if** $i \leq 0$ **then**
6: break
7: **end if**
8: $i \leftarrow i - 1$
9: **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:

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

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.

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

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

5.3 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.

5.4 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:

Remark 20. 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) $n = 4$.

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

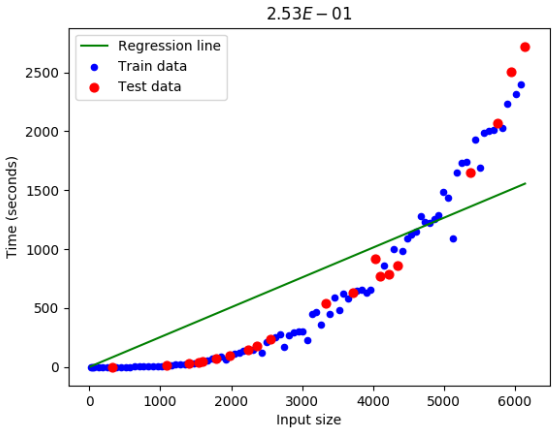

Figure 4. Linear Regression Model trained with features obtained using the transformation $g(n) = n^1$

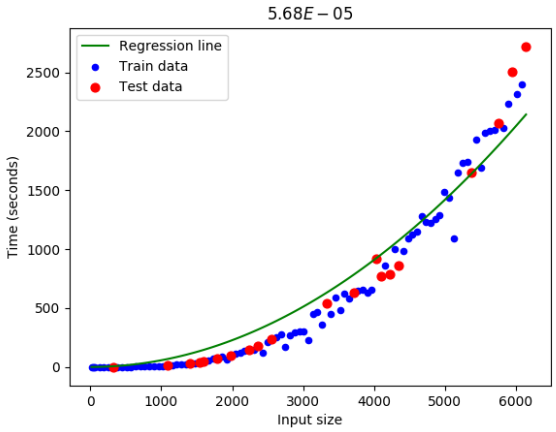

Figure 5. Linear Regression Model trained with features obtained using the transformation $g(n) = n^2$

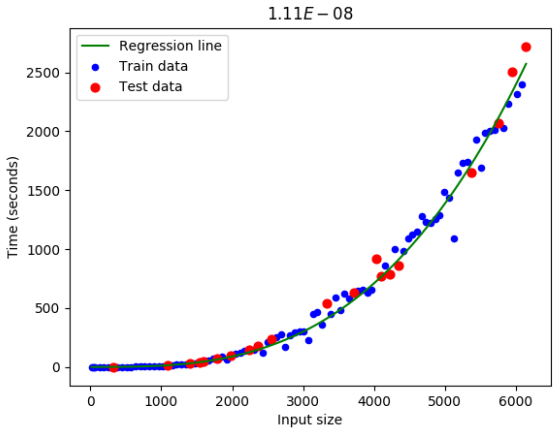

Figure 6. Linear Regression Model trained with features obtained using the transformation $g(n) = n^3$

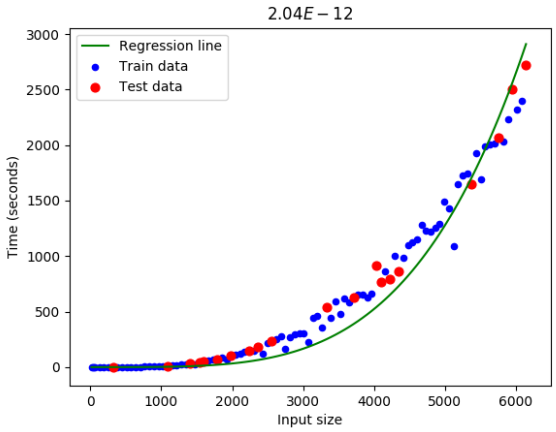

Figure 7. Linear Regression Model trained with features obtained using the transformation $g(n) = n^4$

5.5 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. **?**

5.6 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. **?**

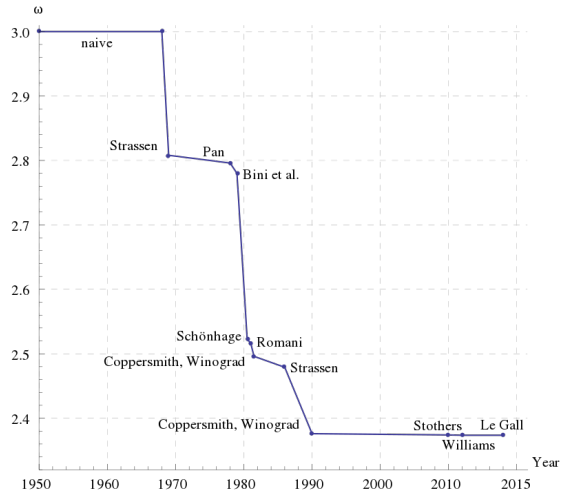

Figure 8. Before 1969, all known matrix multiplication algorithm were in $O(n^3)$. After Volker Strassen first published this algorithm and proved that the naive algorithm wasn't optimal, various attempts to narrow down the exponent ω appeared. One big breakthrough was brought by Coppersmith–Winograd algorithm, with complexity $O(n^{2.375})$.

5.7 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.

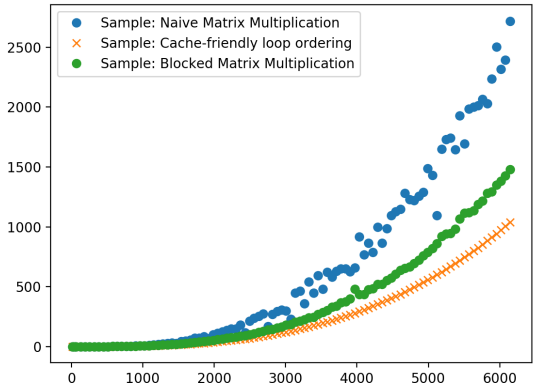

Figure 9. The results above have been reported on an i5 3.2GHz, x86_64 Architecture with L1d cache: 32K, L1i cache: 32K, L2 cache: 256K, L3 cache: 6144K. We do not postulate that the methods above cannot be enhanced or the efficiency of the optimizations are in a specific order. We aim to provide various 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 $f(n) = c \cdot n^3$.

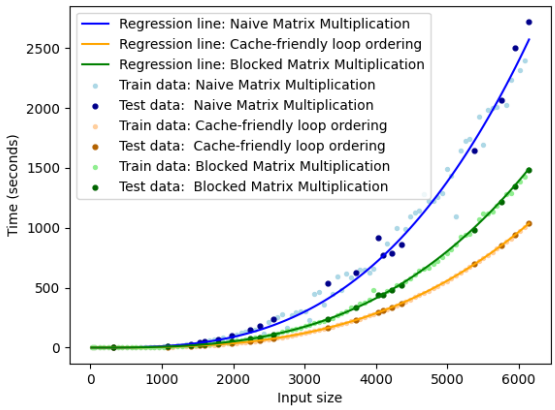

Figure 10. Regression lines corresponding to each of the matrix-multiplication algorithms

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

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:

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	$O_1(6.441 \cdot 10^{-9} \cdot n^3 \cdot HZ)$

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

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:

Algorithm	c	*[Training] RMSQ*	*[Training] R2 score*
Naive Matrix Multiplication	*1.109e-08*	5740.95	0.9886
Cache-friendly loop ordering	*4.472e-09*	0.2517	0.9999969
Blocked Matrix Multiplication	*6.441e-09*	185.70	0.9989

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

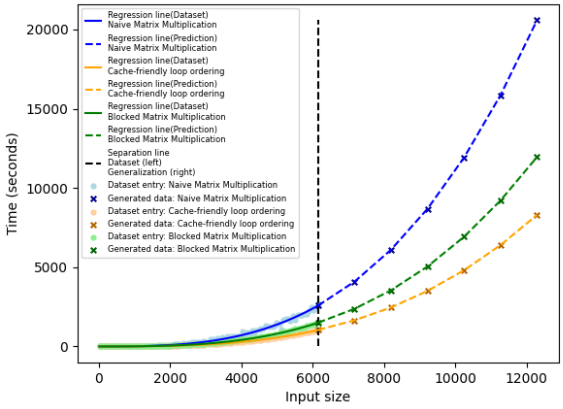

Figure 11. Estimations based on the generalizations of the model tailored on the aquired dataset

5.8 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 5.0.1. 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 5.0.2. 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 $Alg2$.

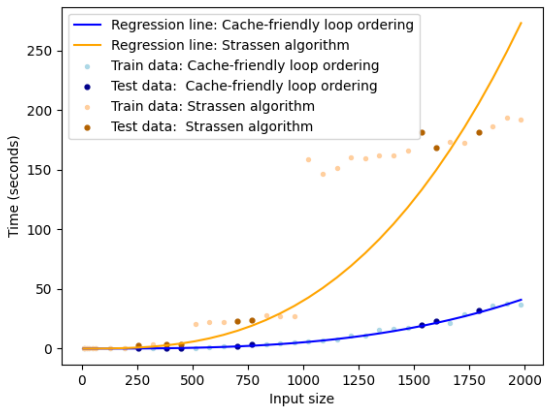

Figure 12. The regression line corresponding to the Cache-friendly loop ordering matrix multiplication algorithm is described by $f(n) = 5.23 \cdot 10^{-9} \cdot n^3$, while the regression line corresponding to the the Strassen algorithm is described by $g(n) = 1.59 \cdot 10^{-7} \cdot n^{2.80}$. Even 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 5.0.1. 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 5.0.2. 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 $n_0 \approx 25970312 \approx 2.5 \cdot 10^7$.

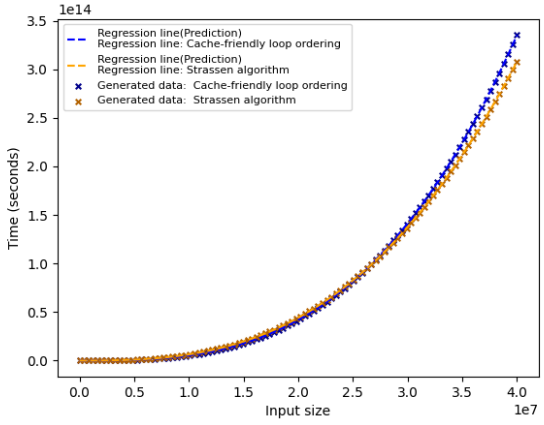

Figure 13. Predictions for the Cache-friendly loop ordering matrix multiplication algorithm and Strassen algorithm based on acquired data. Remark the *"catch up point"* between the two polynomial function for input size $\approx 2.5 \cdot 10^7$.

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.

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

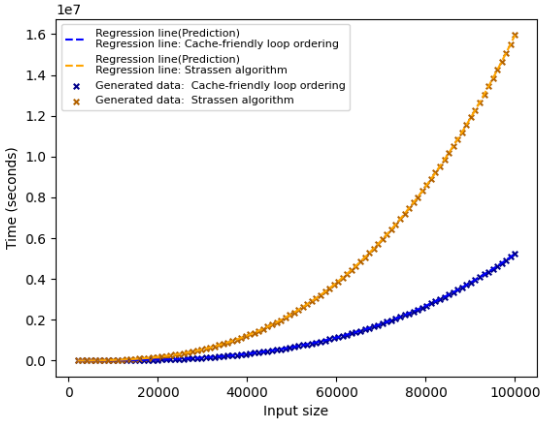

Figure 14. Comparison between the two algorithms. Input size magnitude $\approx 10^6$.

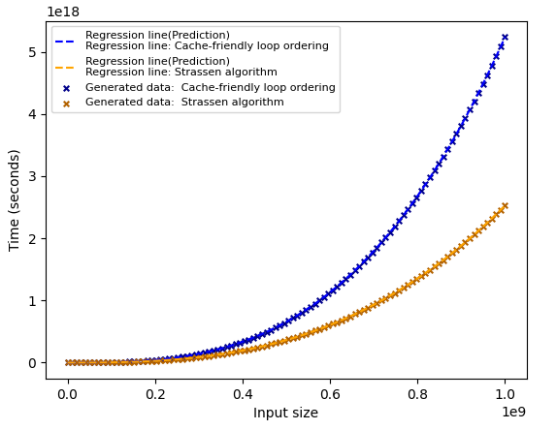

Figure 15. Comparison between the two algorithms. Input size magnitude $\approx 10^9$.

5.9 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 $n^{log_2(7)}$.

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:

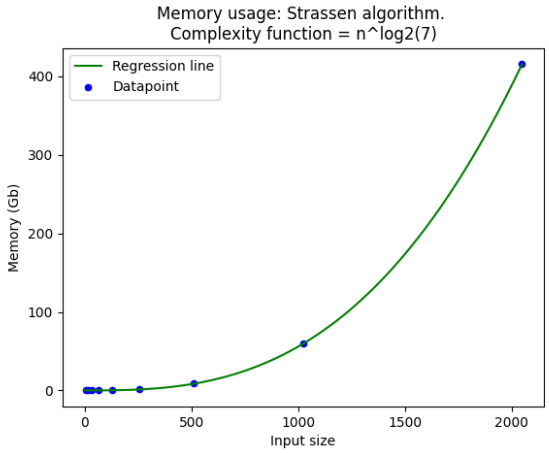

Figure 16. The total peak allocated memory usage during the execution Strassen algorithm for power-of-2 based input data

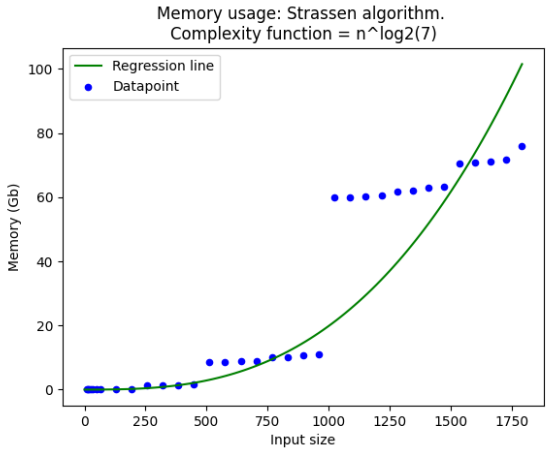

Figure 17. The total peak allocated memory usage during the execution Strassen algorithm for all recorded scenarios

Pitfall 5.0.3. 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 5.0.4. 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).

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 $f_{strassen} = 7.89 \cdot 10^{-8} \cdot n^{2.7}$:

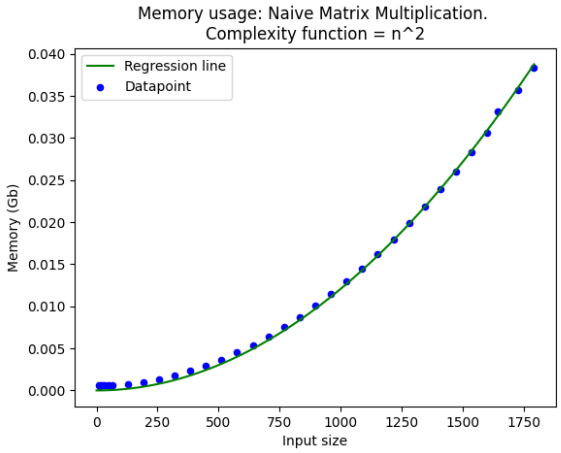

Figure 18. The total peak allocated memory usage during the execution of the Cache-friendly loop ordering matrix multiplication algorithm

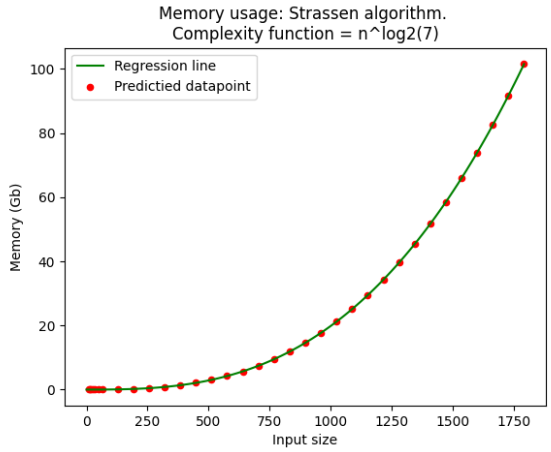

Figure 19. The total peak allocated memory usage (estimated) for the execution of the Strassen algorithm

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 5.0.5. 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.