International Journal of Scientific and Innovative Mathematical Research (IJSIMR)
Volume 3, Issue 10, October 2015, PP 5-13
ISSN 2347-307X (Print) & ISSN 2347-3142 (Online)
www.arcjournals.org

Complexity Study on Fibonacci's Sequence

Ruffin-Benoît M. Ngoie
Mathematics Department
Institut Supérieur Pédagogique (ISP)
Mbanza-Ngungu, DR Congo
benoitmpoy@hotmail.com

Luz N. Mpemba
Mathematics Department
Institut Supérieur Pédagogique (ISP)
Mbanza-Ngungu, DR Congo
luz_lupemba@yahoo.fr

Abstract: *This article deals with algorithmic complexity used in the determination of a Fibonacci's sequence term. While exposing three correct algorithms, we have, in the light of complexity study of each one of them, released the one that is optimal. Numbers of Fibonacci's sequence being for the majority very large, we preferred the using of computer to determine them. Thus, our readers will find in this literature as well the algorithms as their correspondents PASCAL programs.*

Keywords: *Algorithm, Complexity, Fibonacci's sequence, Matrix*

1. Introduction

If we look around us, we see that computers and computer networks are everywhere, activating a fabric of complex human activities: education, trade, entertainment, research, health, communication, and even war. One of the two technological factors causing this astonishing proliferation, is obviously the blowing speed with which the projections in micro-electronics and design of chips took us to increasingly fast hardware.

However, though becoming increasingly fast, computers do not manage to solve certain problems for which the algorithms are however known. This is due to the complexity of the latter. This article aims at indicating to the reader which algorithm to choose vis-à-vis a problem as several correct algorithms can correctly solve the same problem.

For this purpose, the paper is organized as follows: section 2 is devoted to a brief introduction to algorithms, section 3 outlines the Fibonacci's sequence and different algorithms designed to find a Fibonnacci's number. Lastly, section 4 is devoted to remarks and conclusion.

2. General Informations on Algorithms

The word "algorithm" comes from the Latin word (Algorismus) taken after the name of Arab mathematician ALKHAREZMI or Al-khwarizmi, author of a handbook of popularization on Indian positional decimal calculation (about 830 A.D.) explaining its use and, especially, the handling of various algorithms that allow to carry out traditional arithmetic operations (addition, subtraction, multiplication, division, extraction of square roots, rule of three, etc).

Definition 2.1 (Algorithm). An algorithm is a well-defined procedure of calculation, which takes in input a value, or a set of values, and produces, at output, a value or a set of values. Thus, it is a sequence of calculations stages making it possible to pass from the input value to the output value [1].

Definition 2.2 (Program). A program is the realization (implementation) of an algorithm by means of a given language (on a given architecture). It is about the implementation of the principle. For example, during the programming, one will sometimes explicitly deal with the memory management (dynamic allocation into C Language for example) which is an ignored problem of implementation at the algorithmic level [2].

Definition 2.3 (Complexity of an algorithm). The complexity of an algorithm is the measurement of the fundamental operations number which it carries out on a data file. Complexity is expressed like a data file size function.

We note D_n the set of data whose size is n and $T(d)$ the cost of the algorithm on the data d.

Definition 2.4 (Complexity in the best case). It is the smallest number of operations which the algorithm will have to carry out on a data file of fixed size, here with $\boldsymbol{n}$. It is a lower limit of algorithm complexity on a data file of size $\boldsymbol{n}$.

$$\boldsymbol{T}_{\boldsymbol{min}}(\boldsymbol{n}) = \boldsymbol{min}_{\boldsymbol{D} \in \boldsymbol{D}_{\boldsymbol{n}}} \boldsymbol{C}(\boldsymbol{d})$$

Definition 2.5 (Complexity in the worst case). It is the greatest number of operations which the algorithm will have to carry out on a data file of fixed size, here with $\boldsymbol{n}$.

$$\boldsymbol{T}_{\boldsymbol{max}}(\boldsymbol{n}) = \boldsymbol{max}_{\boldsymbol{D} \in \boldsymbol{D}_{\boldsymbol{n}}} \boldsymbol{C}(\boldsymbol{d})$$

Advantage: It is about a maximum, and thus the algorithm will always finish before having carried out $\boldsymbol{T}_{\boldsymbol{max}}(\boldsymbol{n})$ operations.

Disadvantage: This complexity might not reflect the "usual" behavior of the algorithm, the worst case can only occur very seldom, but it is not rare that the average case may be as bad as the worst case.

Definition 2.6 (Complexity on average). It is the average of algorithm complexities on data files of size n (in any rigour, it is obviously necessary to take into account each data file appearance probability).

$$T_{av}(n) = \frac{\sum_{d \in D_n} C(d)}{|D_n|}$$

Advantage: It reflects the "general" behavior of the algorithm if the extreme cases are rare or if complexity slightly varies according to data.

Disadvantage: In practice, complexity on a particular data file can be definitely more significant than complexity on average. In this case, complexity on average will not give a good indication of the algorithm behavior.

In practice, we will be interested only in complexity in the worst case and complexity on average.

Definition 2.7 (Optimality). An algorithm is known as *optimal* if its complexity is minimal complexity among its class algorithms.

We will be exclusively concerned with *complexity in time of algorithms*. It is sometimes interesting deal with others of their characteristics, like *complexity in space* (size of used memory capacity), the required busy bandwidth, etc.

Frederic Vivien [2] suggests that it is necessary to have a model of machine on which the algorithm will be implemented (in the form of program) so that the result of an algorithm analysis may be relevant. In this paper, we will take as reference a model of Random Access Machine (RAM) and single processor, where instructions are carried out one after the other, without simultaneous operations.

The principal reason which pushes to analyze an algorithm is primarily to determine its characteristics and evaluate if it is appropriate to certain applications, or to compare it with other algorithms carrying out the same task. According to Sedgewick and Flajolet [3], resources in time and space (memory), especially in time, are the principal studied characteristics. Indeed, one seeks the execution time of an algorithm running on a particular machine and the memory capacity which it uses. But, in general, one is worried to analyze an algorithm independently of his effective environment: one seeks to obtain results on the principal characteristics of the algorithm from which one can derive a precise estimate from the resources necessary on a given machine.

The algorithms analysis refers to two different concepts of the scientific study of a program performances [3].

The first concept seeks to determine the magnitude order of the algorithm performance in worst of the cases (an upper limit). The essential objective of an analysis is to determine optimal algorithms, in the direction where one can prove, for any algorithm solving a particular problem, that there is a lower limit coinciding with his performances in the worst case. One calls sometimes this type of analysis *calculation of complexity* [4], although this terminology is adapted to the general study of relations between problems, algorithms, languages and machines.

The second approach attempts to characterize in a rigorous way an algorithm performances by analyzing them in best case, the case average and worst of the cases, with methods that allow to refine at will the precision [5, 6, 7, 8, 9].

3. Fibonacci's Sequence

Although having worked much for popularization in Occident of Al Khwarizmi work, in particular by carrying out the potential of the positional system and by working hard to develop it and spread it, Leonardo Fibonacci (15th century Italian mathematician after J-C) is rather more known for his famous sequence of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55… Each one being the sum of its two predecessors [10].

In this section, we present the above-mentionned sequence and its particular properties.

Definition 3.1. (Numerical sequence). Being given A a subset of $\mathbb{R}$, one calls sequence with values in A any map noted u such as:

$$u : \mathbb{N} \rightarrow A$$
$$n \mapsto u(n) = u_n$$

Definition 3.1. (Fibonacci's sequence). More formally, the numbers of Fibonacci F_n are generated by the simple rule:

$$F_n = \begin{cases} F_n + F_{n-1} & if \quad n > 1 \\ 1 & if \quad n = 1 \\ 0 & if \quad n = 0 \end{cases}$$

No other sequence of numbers was studied or was as largely applied to so many fields as this sequence: Biology, Demography, Art, Architecture, Music, to quote only these [11].

And together with the powers of 2, they are the favorite continuations of Data processing. Indeed, Fibonacci's numbers grow almost as quickly as the powers of 2 : For example, F_{30} is worth more than one million and F_{100} is a number of 21 digits! In general $F_n \approx 2^{0.694\,n}$ [3].

But which is the precise value of F_{100}, F_{200} ? Fibonacci himself would like surely to know it. To answer this question, we need an algorithm to calculate nth of Fibonacci's numbers.

3.1. An exponential algorithm

An idea is the blind application of the recursive definition of. F_n. Here the resulting algorithm in "pseudocode ", a notation used in this paper:

```
Function Fib1(n)
If n=0 : Return 0
If n=1 : Return 1
Return Fib(n-1)+fib(n-2)
```

The corresponding PASCAL program is:

```
PROGRAM Fonction_Fib1(INPUT,OUTPUT) ;
USES CRT;
VAR
                          K : INTEGER ;
FUNCTION Fib1(n : INTEGER) : REAL ;
BEGIN
      IF n=0 THEN Fib1 := 0 ;
      IF n=1 THEN Fib2 := 1 ;
      Fib(n) := Fib1(n-1) + Fib1(n-2)
END ;

BEGIN
      CLRSCR;
      WRITE('Saisir le rang du terme de la suite de Fibonacci ');
      READLN(K) ;
      WRITELN('Le terme équivalent vaut :', Fib1(K))
END.
```

Even though we have the algorithm, there are three questions that we ask on this subject:

1. Is this algorithm correct?
2. How much time is it taken like function of n?
3. Can we better do?

The first question is debatable (it might not be asked) more especially as the algorithm, as well as the corresponding program, is precisely the Fibonacci's definition of F_n. But the second question deserves an answer.

Let $T(n)$ be the number of operations which the computer needs to carry out to calculate Fib1(n). What can we say about this function? For not informed readers, if n is lower than 2, the procedure stops almost immediately after two operations. Thus, $T(n) \leq 2, \forall\, n \leq 1$

For great values of n, there are two recursive invocations of Fib1 repeated $T(n-1)$ and $T(n-$ 2 times respectively, plus three operations of the computer (2 checks on the value of n and the final addition).

Then $T(n) = T(n-1) + T(n-2) + 3\ \forall\, n > 1$

Let us compare this result with the recursive relation of F_n : We see immediately that. $T(n) \geq F_n$. This is a bad news: The execution time of the algorithm grows as quickly as Fibonacci's numbers! $T(n)$ is exponential out of n, which implies that the algorithm is practically slow except for very small values of n.

Let us be a little more concrete on the disadvantage of an exponential growth. To calculate F_{200}, the Fib1 algorithm carries out $T(n) \geq F_{200} \geq 2^{138}$ elementary operations of the computer. Time that that takes depends of course on the computer used. Until 2006, Dasgupta et al. [12, 13] affirm that the fastest computer in the world was the NEC Earth Similator whose speed was 40. 10 12 elementary operations per second. Even on this computer, Fib1(200) would take at least 2^{92}seconds ($1{,}51 \times 10^{20}$ years). This means that, if we begin calculation today, it would continue even after the sun is transformed into an enormous red star.

But technology develops quickly: Speed of computers doubles every 18 months, a phenomenon sometimes called "Moore's law". With this extraordinary growth, perhaps Fib1 will be carried out more quickly by the future years machines. Cottet-Emard and Goetgheluck [14] remain pessimistic on this subject: Indeed, the execution time of Fib1(n) is proportional to $2^{0.694\,n} \approx (1.6)^n$, thus one will need 1,6 times more time to calculate F_{n+1} than F_n. And under the Moore's law, computers become 1,6 times faster each year. Thus, if we can reasonably calculate F_{100} with this year technology, then the next year we will calculate F_{101}. And the following year F_{102}, and so on: just a Fibonacci's number moreover each year! such is the misfortune of exponential time.

In short, our naive and recursive algorithm is correct but hopelessly ineffective. We will need, to solve this problem, to resort to another algorithm.

3.2. A polynomial algorithm

Let us try to understand why Fib1 is so slow. The figure below shows the cascade of recursive invocations started by a simple call to Fib1(n). Let us notice how several calculations are repeated!

A more significant arrangement would hold the intermediate results (values $F_0, F_1, \cdots, F_{n-1}$) since they are known.

```
Function Fib2(n)
If n=0 Return 0
Create Table F[0…n]
F[0]=0, F[1]=1
For i=2…n :
F[i]=F[i-1]+F[i-2]
Return F[n]
```

The corresponding PASCAL program is:

```
PROGRAM Fonction_Fib2(INPUT, OUTPUT) ;
USES CRT ;
VAR
      I, n : INTEGER ;
      Fib2 : ARRAY[0..n] OF REAL ;
BEGIN
  CLRSCR ;
  WRITE('Saisir le rang du terme de la suite de Fibonacci ') ;
  READLN(n) ;
  IF n=0 THEN Fib2 :=0 ;
  Fib2[0] := 0 ;
  Fib2[1] := 1 ;
  FOR i := 2 TO n DO
    Fib2[i] := Fib2[i-1]+Fib2[i-2]
  WRITELN('Le terme equivalent vaut : ', Fib2[i])
END.
```

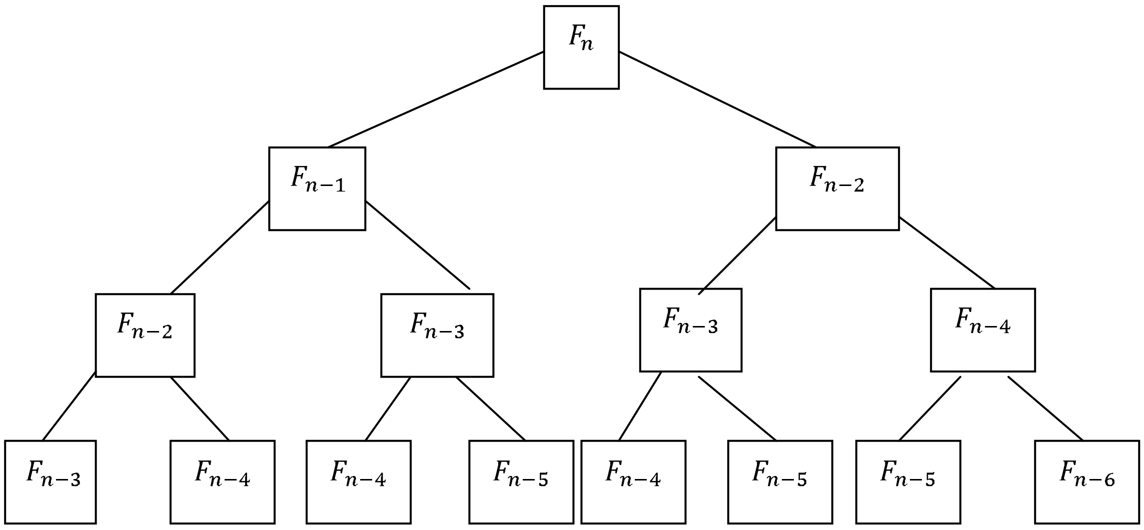

Figure 1. *Determination of F_n by algorithm Fib2*

As with Fib1, the exactitude of this algorithm is evident by itself because it directly uses the Fibonacci's definition of F_n. How much time does it take? The internal loop consists of a single operation of the computer and is carried out $(n-1)$ times. Thus, the number of computer operations used by Fib2 is linear out of n.

From the exponential one, we fall down to the polynomial one. A great rupture in the execution time. It is perfectly reasonable to calculate F_{200} or even F_{200000}.

As we will see it in the continuation of this paper, the best algorithm makes all the difference of it.

3.3. A thorough analysis

In this paper, we will count the number of basic elementary operations carried out by each algorithm while keeping of sight that these basic operations require a constant number of occurences. This is a significant simplification. After all, a set of processor instructions has a variety of basic primitives (branch, storage in memory, comparison of the numbers, simple arithmetic, etc.) [15]. And rather than to make a distinction between these elementary operations, it is by far more suitable to classify them in the same category.

But looking behind towards our processing of Fibonacci's algorithms, we were less rigorous with what we had regarded as elementary operation. It is reasonable to treat the addition as a simple elementary operation if small numbers are added, it is to say 32 bits numbers. But nth number of

Fibonacci has approximately a length of $0{,}694 \times n$ bits and this can exceed 32 bits by far when n grows. The arithmetic operations on the arbitrarily large numbers cannot possibly be at constant time. We thus need to revisit our preceding estimates of execution time and to make them more honest.

Fib1 which calculates on the additions of F_n now uses a number of basic operations brutally proportional to $n \times F_n$. In the same way, the number of operations required by Fib2 is proportional to n^2, always polynomial out of n and in fact exponentially higher than Fib1. This correction with the analysis of the execution time does not decrease the rupture. But it is necessary to think of an algorithm much faster.

3.4. A thorough analysis simple and fast algorithm

Let us note F the Fibonacci's sequence. Here a table giving the few first values:

n	0	1	2	3	4	5	6	7	8
F(n)	0	1	1	2	3	5	8	13	21

Each term is thus equal to the sum of two preceding terms. In a general way, Fibonacci's sequence can be written as follows :

$$F_n = \begin{cases} F_n + F_{n-1} & if \quad n > 1 \\ 1 & if \quad n = 1 \\ 0 & if \quad n = 0 \end{cases}$$

It is the fundamental relation which characterizes the Fibonacci's sequence. By replacing n by (n+1), one obtains: $F(n+1) = F(n) + F(n-1)$.

One can notice that $F(n-1) = F(n+1) - F(n)$(Each term is equal to the difference of the two following).

Matrix algebra is very useful to study properties of this sequence:

$$\begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix} = \begin{pmatrix} F(n) + F(n-1) \\ F(n) \end{pmatrix} = \begin{pmatrix} 1.F(n) + 1.F(n-1) \\ 1.F(n) + 0.F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} \tag{3.4.1}$$

Or $$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} F(n-1) + F(n-2) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F(n-1) \\ F(n-2) \end{pmatrix} \tag{3.4.2}$$

Replacing (2) in (1), one finds:

$$\begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F(n-1) \\ F(n-2) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} F(n-1) \\ F(n-2) \end{pmatrix}$$

While developing successively, one obtains:

$$\begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F(1) \\ F(0) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Let us pose $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, one has then $\begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix} = A^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

To study the powers of the matrix A, one uses the traditional method of the diagonalisation. Let us start by seeking its eigenvalues [16]. They are roots of the equation:

$$\begin{vmatrix} 1-\lambda & 1 \\ 1 & -\lambda \end{vmatrix} = 0 \Leftrightarrow \lambda^2 - \lambda - 1 = 0$$

Its roots are $t_1 = \frac{1+\sqrt{5}}{2}$ and $t_2 = \frac{1-\sqrt{5}}{2}$. They are called golden numbers.

Let us indicate some significant properties of these two numbers:

(a) $t_1.t_2 = -1$ (b) $t_1 + t_2 = 1$ (c) $t_1 - t_2 = \sqrt{5}$ (d) $t_1^2 = t_1 + 1$

(e) $t_2^2 = t_2 + 1$ (f) $\frac{1}{t_1} = t_1 - 1$ (g) $\frac{1}{t_2} = t_2 - 1$

Let us multiply successively the two members of the equality (d) by t_1, t_1^2, t_1^3, ..., one obtains :

$t_1^3 = t_1^2 + t_1 = 2t_1 + 1;$

$t_1^4 = t_1^3 + t_1^2 = 3t_1 + 2;$

$t_1^5 = t_1^4 + t_1^3 = 5t_1 + 3;$

$t_1^6 = t_1^5 + t_1^4 = 8t_1 + 5;$

…

In a general way, one has :

$t_1^n = F(n).t_1 + F(n-1)$ (3.4.3)

By a similar reasoning, one has : $t_2^n = F(n).t_2 + F(n-1)$ (3.4.4)

[10]

By making substraction member by member with (3.4.3) and (3.4.4), one has :

$$t_1^n - t_2^n = F(n)(t_1 - t_2) = \sqrt{5}\,.F(n)$$

Then $F_n = \frac{t_1^n - t_2^n}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$ (3.4.5)

This equality is greatly important importance: it makes it possible to replace, at will, calculations on the Fibonacci's continuation by calculations on golden sections.

It is clear that from this last equality comes out a very fast and non-recursive algorithm.

Indeed, for great values of n, it is enough to calculate just

$$F_n = \frac{t_1^n - t_2^n}{\sqrt{5}}$$

with $t_1 \approx 1{,}618$ and. $t_2 \approx -0{,}618$

The corresponding PASCAL program is:

```
PROGRAM Fonction_Fib3(INPUT, OUTPUT) ;
USES CRT ;
VAR
      n : INTEGER ;
      Fib3, K, L : REAL ;
BEGIN
  CLRSCR ;
  WRITE('Saisir le rang du terme de la suite de Fibonacci ') ;
  READLN(n) ;
  K := (1+SQRT(5))/2 ;
  L := (1-SQRT(5))/2 ;
  Fib3 := (EXP(n*LN(K)) - EXP(n*LN(L))) / SQRT(5) ;
  WRITELN('Le terme équivalent vaut : ', Fib3)
END.
```

4. Concluding Remarks

Throughout this paper, we showed that several algorithms can be applied to solve the same problem. However, an algorithm, though correct, can be difficult to implement even with a computer!

Towards several algorithms relating to the same problem, the reader will have to study the complexity of each one of them and carry out which is optimal. It is then advisable to remember that the less complex one algorithm is, the more it is efficient and fast.

In our concern, we studied one of the problems which made spout out ink and saliva of our time : The Fibonacci's sequence. More applications, in fields which we will not be able to enumerate all, for this sequence and its famous golden number. The characteristic of this sequence is that it is of an quasi-exponential growth and that an algorithm obeying scrupulously its definition becomes quickly ineffective and excessively slow as for the determination of the terms of sufficiently large row (from 100). We presented three completely correct algorithms of decreasing complexities; that is to say the first is more complex than the second and the latter more complex than the third. The first algorithm named Fib1 is of an exponential complexity and is too slow so that only Providence knows if the world will still exist to calculate the 200th term of the sequence here studied. The second is of a polynomial complexity and calculates 200th term in a time much more reasonable than its predecessor.

The object of our study is to present a much better algorithm than the two above-mentioned ones. It is obvious that the numbers of Fibonacci's sequence being for the majority too large, it is advisable to determine them using a computer. With this intention, we presented not only these algorithms but also the correspondents programs PASCAL. The Fib3 algorithm is not only fast but its programming is of an astonishing facility. With the opposite of its predecessors which are all recursive, it has a linear and very simple structure thus by far the least complex, better and indicated to solve our problem.

REFERENCES

[1]. Cormen T., Leiserson C. and Rivest R. *Introduction à l'algorithmique*, Dunod, Paris, 1994.

[2]. Vivien, F. *Algorithmique avancée*, IUP2, Paris, 2002.

[3]. Sedgewick R. and Flajolet P. Introduction à l'analyse des algorithmes, International Thomson Publishing France, Paris, 1996.

[4]. Aho A., Hopcroft J. E. et Ullman J. D. *The design and analysis of Algorithms*, Addison-wesley, Reading, MA, 1975.

[5]. Knuth D. E. *The art of Computer programming*. Volume 1: Fundamental Algorithms, Addison-Wesley, Reading, MA, 1968.

[6]. Knuth D. E. *The art of Computer programming*. Volume 2: Seminumerical Algorithms, Addison-Wesley, Reading, MA, 1969.

[7]. Knuth D. E. Mathematical Analysis of algorithms, Information Processing 71, *Proceedings of the IFIP Congress* : 19-27, 1971.

[8]. Knuth D. E. *The art of Computer programming*. Volume 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.

[9]. Knuth D. E. Big Omicron and big Omega and big Theta, SIGACT News : 18 – 24, Avril-juin 1976, 1976.

[10]. Berglund N. *La suite de Fibonacci*, Recréations mathématique, Université du Sud Toulon-Var, 2005.

[11]. Sciences et Techniques, « La suite de Fibonacci », in Routes n°83, 2003.

[12]. Dasgupta S., Papadimitriou, C.H., Vazirani U.V. Algorithms, Technical report, 2006.

[13]. Carey M.R. and Johnson D.S., *Computers & Interactability: a guide to the Theory of the NP-Completeness*, W.H. Freeman and Company, 1979.

[14]. Cottet-Emard F. and Goetgheluck F. *Mathématiques sur ordinateur*, Ed. DeBoeck-Wesmael, Bruxelles, 1993.

[15]. Thiel, E. *Algorithmes et programmation en PASCAL*, DEUG1, Cours, Luminy, 2004.

[16]. Brezinski C. and Redivo-Zaglia, M. *Méthodes numériques directes de l'Algèbre matricielle*, Ed. Ellipses, Paris, 2005.

AUTHORS' BIOGRAPHY

Ruffin-Benoît Ngoie Mpoy, is Senior Lecturer at Institut Supérieur Pédagogique de Mbanza-Ngungu (Teaching College of Mbanza-Ngungu), Democratic Republic of the Congo. He teaches Computer Programming, Numerical Analysis and Algorithms Theory. His main fields are Computational Social Choice Theory, Multiple Criteria Decision Aiding, and Games Theory. He proposed, in collaboration with Berthold Ulungu, the MMCM to slice between median and mean values.

Luz Mpemba Ngoma, is junior lecturer at Institut Supérieur Pédagogique de Mbanza-Ngungu (Teaching College of Mbanza-Ngungu), Democratic Republic of the Congo. He teaches Operating Systems Theory, Computer Programming and Computers Architecture since 2013. He is also a visitor lecturer at Université Pédagogique Nationale (National Teaching University), Kinshasa, Democratic Republic of the Congo.