National Polytechnic Institute
Superior School of Computer Sciences

Algorithm Analysis.

Practice 1 - Experimental Determination of the Temporal Complexity of an Algorithm.

Hernandez Martinez Carlos David.
davestring@outlook.com.
Group: 3cv2.

August 26, 2017

Contents

1 Introduction:

To say that a problem is solvable algorithmically means, informally, that a computer program can be written that will produce the correct answer for any input if we let it run long enough and allow it as much storage space as it needs. In the 1930s, before the advent of computers, mathematicians worked very actively to formalize and study the notion of an algorithm, which was then interpreted informally to mean a clearly specified set of simple instructions to be followed to solve a problem or compute a function. Various formal models of computation were devised and investigated. Much of the emphasis in the early work in this field, called ***computability theory***, was on describing or characterizing those problems that could be solved algorithmically and on exhibiting some problems that could not be. One of the important negative results, established by Alan Turing, was the proof of the unsolvability of the halting problem. The halting problem is to determine whether an arbitrary given algorithm (or computer program) will eventually halt (rather than, say, get into an infinite loop) while working on a given input. There cannot exist a computer program that solves this problem. [1]

Although computability theory has obvious and fundamental implications for computer science, the knowledge that a problem can theoretically be solved on a computer is not sufficient to tell us whether it is practical to do so. For example, a perfect chess-playing program could be written. This would not be a very difficult task; there are only a finite number of ways to arrange the chess pieces on the board, and under certain rules a game must terminate after a finite number of moves. The program could consider each of the computers possible moves, each of its opponents possible responses, each of its possible responses to those moves, and so on until each sequence of possible moves reaches an end. Then since it knows the ultimate result of each move, the computer can choose the best one. The number of distinct arrangements of pieces on the board that it is reasonable to consider (much less the number of sequences of moves) is roughly 1050 by some estimates. A program that examined them all would take several thousand years to run. Thus such a program has not been run. [1]

Numerous problems with practical applications can be solved that is, programs can be written for them but the time and storage requirements are much too great for these programs to be of practical use. Clearly the time and space requirements of a program are of practical importance. They have become, therefore, the subject of theoretical study in the area of computer science called ***computational complexity***. [1]

2 Basic Concepts:

- θ Notation:

A θ(g(n)) is defined as:

θ (g(n)) = { f(n) | $\exists$ C_1, $C_2 > 0$ and $n_0 > 0$ then $0 \leq C_1$g (n) $\leq$ f(n) $\leq C_2$g(n) $\forall$ n $\geq n_0$ }

- O Notation:

A O(g(n)) is defined as:

O (g(n)) = { f(n) | $\exists$ C, $n_0 > 0$ then $0 \leq$ f(n) $\leq$ Cg(n) $\forall$ n $\geq n_0$ }

- Ω Notation:

A Ω(g(n)) is defined as:

Ω (g(n)) = { f(n) | $\exists$ C, $n_0 > 0$ then $0 \leq$ g(n) $\leq$ f(n) $\forall$ n $\geq n_0$ }

3 Development:

3.1 Binary Sum:

The program is divided in three python modules for have a better control of the code.

- main.py: Contain the method "generate", "display" and "main".
- graph.py: Evaluate ***time*** against ***r=n*** where 'n' is the length of the binary number list.
- binarysum.py: Contains the algorithm that implements the sum.

3.1.1 Main.py

In main.py there is a method named "generate" and will initialize and fill 2 lists of binary numbers generated randomly, of course, after having define the size of both lists. This sizes are of order ***two to the power of 'n'*** *(* 2^n *)*. Note that can be $\boldsymbol{n} > \boldsymbol{m}$ or $\boldsymbol{n} = \boldsymbol{m}$ but never $\boldsymbol{n} < \boldsymbol{m}$. This because the ***for*** in the line 10 it's from ***i = 0*** to ***'n'***, this means that the size of the lists ***'a'*** and ***'b'*** are both of size ***'n'***. In case that $\boldsymbol{n} > \boldsymbol{m}$ the rest of the list ***'b'*** will be filled with ***0's***.

```
 1 def generate ( ):
 2     a, b, n, m = [ ], [ ], 0, 1
 3     # Define the size of a-b lists ( n-m respectively ).
 4     while ( n < m ):
 5         rnd = ( int ( random ( ) * 5 ) + 1 )
 6         n = int ( pow ( 2, rnd ) )
 7         rnd = ( int ( random ( ) * 5 ) + 1 )
 8         m = int ( pow ( 2, rnd ) )
 9     # Generate both binary numbers as integer lists.
10     for i in range ( n ):
11         a.insert ( 0, ( ( int ( random ( ) * 2 ) + 0 ) ) )
12         if ( i >= m ):
13             b.insert ( 0, 0 )
14         else:
15             b.insert ( 0, ( ( int ( random ( ) * 2 ) + 0 ) ) )
16     # Return statement.
17     return a, b
```

The method "display" it's more simple, its only function it's to display in console the binary numbers (lists) ***a [n]*** and ***b [n]***, the result of the sum stored in a list ***'c'*** and the order of the resulting list (2^p) as the figure 4.1.0 shows.

```
def display ( a, b, c ):
    DATAFORMAT = ""
    if ( len ( c ) > len ( a ) ): DATAFORMAT = "    "
    print ( "\n\tA: " + DATAFORMAT, a )
    print ( "\tB: " + DATAFORMAT, b )
    print ( "\tC: ", c )
    print ( "\n\tOrder of the sum: ", int ( pow ( 2, len ( c ) - 1 ) ), "\n" )
```

Finally, the method "main", aside of being the principal, calls the others methods and controls the sequence of execution.

```
def main ( ):
    a, b = generate ( )
    c = binarysum ( a, b )
    display ( a, b, c )
main ( )
```

3.1.2 Binarysum.py

Principal module, contains the algorithm that we are determining the temporal complexity. As the binary sum works, the code takes the values of the ***n - i*** positions, and evaluates:

- With carry = 0:
  1. If **0** + 0 + 0 = 00 then ***sum = 0*** and ***carry = 0***.
  2. If **0** + 0 + 1 = 01 then ***sum = 1*** and ***carry = 0***.
  3. If **0** + 1 + 0 = 01 then ***sum = 1*** and ***carry = 0***.
  4. If **0** + 1 + 1 = 10 then ***sum = 0*** and ***carry = 1***.
- With carry = 1:
  1. If **1** + 0 + 0 = 01 then ***sum = 1*** and ***carry = 0***.
  2. If **1** + 1 + 0 = 10 then ***sum = 0*** and ***carry = 1***.
  3. If **1** + 0 + 1 = 10 then ***sum = 0*** and ***carry = 1***.
  4. If **1** + 1 + 1 = 11 then ***sum = 1*** and ***carry = 1***.

... So on, until reaching the numbers of the 0 position. In case of ***carry = 1***, a '1' is added to the position ***c [0]***. of the list, as the line 20 shows.

```
1  def binarysum ( a, b ):
2      i, carry, c = ( len ( a ) - 1 ), 0, [ ]
3      # Evaluate both binary lists ( 'a' and 'b' ) and scroll through from ( n - 1 ) to 0
4      # and sum the numbers at 'i' position, the result is stored in 'c'.
5      while ( i >= 0 ):
6          if ( ( a [ i ] + b [ i ] + carry ) == 1 ):
7              c.insert ( 0, 1 )
8              carry = 0
9          elif ( ( a [ i ] + b [ i ] + carry ) == 2 ):
10             c.insert ( 0, 0 )
11             carry = 1
12         elif ( ( a [ i ] + b [ i ] + carry ) == 3 ):
13             c.insert ( 0, 1 )
14             carry = 1
15         else:
16             c.insert ( 0, 0 )
17             carry = 0
18         i = i - 1
19     # After finish, evaluates if there is a carry value.
20     if ( carry == 1 ):
21         c.insert ( 0, carry )
22     # Return statement.
23     return c
```

Observation: *Because it is a list, elements can easily be inserted or removed without reallocation or reorganization of the entire structure.*

To analyze the temporal complexity of the algorithm it's necessary to put a counter in each line of the code, this line will help to plot ***time (t)*** against ***size (n)***.

```python
def binarysum ( a, b ):
    i , carry , c, count = ( len ( a ) − 1 ), 0, [ ], 1
    # Evaluate both binary lists ( 'a' and 'b' ) and scroll through from ( n − 1 ) to 0
    # and sum the numbers at 'i' position , the result is stored in 'c'.
    while ( i >= 0 ):
        count += 1
        if ( ( a [ i ] + b [ i ] + carry ) == 1 ):
            count += 1
            c.insert ( 0, 1 )
            count += 1
            carry = 0
            count += 1
        elif ( ( a [ i ] + b [ i ] + carry ) == 2 ):
            count += 1
            c.insert ( 0, 0 )
            count += 1
            carry = 1
            count += 1
        elif ( ( a [ i ] + b [ i ] + carry ) == 3 ):
            count += 1
            c.insert ( 0, 1 )
            count += 1
            carry = 1
            count += 1
        else:
            count += 1
            c.insert ( 0, 0 )
            count += 1
            carry = 0
            count += 1
        i = i − 1
        count += 1
    count += 1
    # After finish , evaluates if there is a carry value.
    count += 1
    if ( carry == 1 ):
        c.insert ( 0, carry )
        count += 1
    # Return statement.
    return c, count
```

Observation: *Note that the return statement it's the counter and the result of the sum stored in a list, were the counter will be the vertical axis and the length of 'c' the horizontal axis.*

3.1.3 Graph.py

For plotting the result I'm using two python tools named ***matplotlib.pyplot*** and ***numpy***. The module ***graph.py*** has two methods. The first one is ***parameters ( )***. This method create the points where the curve will be plotted according to the parameters that ***main*** sends.

```python
def parameters ( size , time ):
    # time vs size graph points.
    t , n = [ ] , [ 0 ]
    # div: Auxiliar variable that help to plot the graph.
    div = float ( "{0:.2f}".format ( 1 / round ( time / size ) ) )
    # Time ( t ) parameters.
    for i in range ( time ):
        t.append ( i )
    # Size ( n ) parameters.
    for i in range ( time ):
        if ( i != 0 ):
            n.append ( float ( "{0:.2f}".format ( n [ i - 1 ] + div ) ) )
    # Return statement.
    return t , n
```

Finally the last method is ***graph ( )***, as the name says, plot the graphic of the temporal complexity for this algorithm comparing ***time (t)*** against ***size (n)***.

```python
def graph ( size , time ):
    # Window title.
    plt.figure ( "Temporal Complexity of Binary Sum Algorithm" )
    # Graph title.
    plt.title ( "Binary Sum:" , color = ( 0.3 , 0.4 , 0.6 ) , weight = "bold" )
    # Construct the parameters of the graph.
    t , n = parameters ( size , time )
    # Define the limits of the graph.
    plt.xlim ( 0, size )
    plt.ylim ( 0, time )
    # Proposed function: g ( n ) = ( 3/2 )n.
    _t = list ( map ( ( lambda x: x * 3/2 ), t ) )
    # Names of the axes.
    plt.xlabel ( "Size ( n )" , color = ( 0.3 , 0.4 , 0.6 ) ,
        family = "cursive" , size = "large" )
    plt.ylabel ( "Time ( t )" , color = ( 0.3 , 0.4 , 0.6 ) ,
        family = "cursive" , size = "large" )
    # Plot.
    plt.plot ( n, _t , "#800000" , linestyle = "--" , label = "g( n ) = ( 3/2 )n" )
    plt.plot ( n, t , "#778899" , linewidth = 3, label = "E( n ) = n" )
    plt.legend ( loc = "lower right" )
    plt.show ( )
```

3.2 Euclidean Algorithm:

The program is divided in four python modules for have a better control of the code.

- main.py: Contain the method "generate", "display", "menu" and "main".
- graph.py: Evaluate ***time*** against ***gcd*** where gcd it's the gratest common divisor of 'n' and 'm'.
- euclidean.py: Contains the Euclidean Algorithm.
- fibonacci.py: Return a list of 'n' Fibonacci numbers.

3.2.1 Main.py

In main.py there is a ***menu ( )*** method, the user will have two options, generate random numbers or use the Fibonacci numbers to calculate their GCD using the Euclidean algorithm. In case that the user chose for random numbers, ***main ( )*** will call the method generate and will return two numbers, ***'n' and 'm'***.

```python
def generate ( ):
    n, m = 0, 1
    while ( n < m ):
        n = ( int ( random ( ) * 1000 ) + 1 )
        m = ( int ( random ( ) * 1000 ) + 1 )
    # Return statement.
    return n, m
```

... In other case, ***main ( )*** will call ***fibonacci ( )*** (see section 3.2.2). As we can see, the principal work of this method is to control the execution sequence of the program.

```python
def main ( ):
    ans = menu ( )
    if ( ans == 2 ):
        print ( "\n\tFirst 'n' Fibonacci's to calculate: " )
        limit = int ( input ( "\tAnswer: " ) )
        # Return a Fibonacci numbers list.
        fibo = fibolist ( limit )
        # Assign the last two Fibonacci's in the list.
        n, m = fibo [ len ( fibo ) - 1 ], fibo [ len ( fibo ) - 2 ]
    else:
        # Generate random numbers.
        n, m = generate ( )
    # Return: GCD, algorithm counter for temporal complexity and gcd process list.
    result, count, gcd = algorithm ( n, m )
    # Display on screen the result.
    display ( n, m, result )
    # Graph of the Temporal Complexity of the Euclidean Algorithm.
    graph ( n, m, count, gcd )

main ( )
```

3.2.2 Fibonacci.py

The module has two methods ***fibonacci ( )*** and ***fibolist ( )***. The first method create a generator of Fibonacci numbers, and the second method returns a 'list' of the firsts 'n' Fibonacci numbers.

```
1  # Fibonacci Generator.
2  def fibonacci ( ):
3      a, b = 1, 1
4      while ( True ):
5          yield a
6          a, b = b, a + b
```

The program call ***fibonacci ( )*** in line '5' inside a ***for*** loop, this because the method it's a generator.

```
1  # Fibonacci list.
2  def fibolist ( limit ):
3      n, fibo = 0, [ ]
4      # Create a Fibonacci numbers list using a generator.
5      for i in fibonacci ( ):
6          if ( n >= limit ): break
7          fibo.append ( i )
8          n += 1
9      # Return statement.
10     return fibo
11         a, b = b, a + b
```

Observation: *Note that fibolist ( ) receive as parameter a 'limit', this integer will define to which number we want the generator to calculate.*

3.2.3 Euclidean.py

The principal module, contains the algorithm that we are determining the temporal complexity. The Euclidean algorithm proceeds in a series of steps such that the output of each step is used as an input for the next one. Let $\boldsymbol{k}$ be an integer that counts the steps of the algorithm, starting with zero. Thus, the initial step corresponds to $\boldsymbol{k = 0}$, the next step corresponds to $\boldsymbol{k = 1}$, and so on.

Each step begins with two non-negative remainders $\boldsymbol{r_{k-1}}$ and $\boldsymbol{r_{k-2}}$ Since the algorithm ensures that the remainders decrease steadily with every step, $\boldsymbol{r_{k-1}}$ is less than its predecessor $\boldsymbol{r_{k-2}}$. The goal of the $\boldsymbol{k}$th step is to find a quotient $\boldsymbol{q_k}$ and remainder $\boldsymbol{r_k}$ that satisfy the equation

$$r_{k-1} = q_k r_{k-1} + r_k \tag{1}$$

and that have $\boldsymbol{r_k} < \boldsymbol{r_{k-1}}$. In other words, multiples of the smaller number $\boldsymbol{r_{k-1}}$ are subtracted from the larger number $\boldsymbol{r_{k-2}}$ until the remainder$\boldsymbol{r_k}$ is smaller than $\boldsymbol{r_{k-1}}$. The implementation can be the following one.

```python
def algorithm ( n, m ):
    # Euclidean algorithm: Find the Greatest Common Divisor ( gcd ).
    while ( m != 0 ):
        r = n % m
        n = m
        m = r
    # return statement.
    return n
```

The program return the Greatest Common Divisor, but to determinate the time that the algorithm takes to find the GCD we need to use a counter in each line of the code, and later compare it with the parameter ***'n'***.

Observation: *Parameter 'n' always will be bigger than 'm'..*

```python
def algorithm ( n, m ):
    # Euclidean algorithm: Find the Greatest Common Divisor ( gcd ).
    count, gcd = 1, [ ]
    while ( m != 0 ):
        gcd.append ( n )
        count += 1
        r = n % m
        count += 1
        n = m
        count += 1
        m = r
        count += 1
    count += 1
    # return statement.
    return n, count, gcd
```

Observation: *Now apart of returning the GCD, returns the counter, and the list of the Euclidean process.*

3.2.4 Graph.py

For plotting the result I'm using ***matplotlib*** and ***numpy*** and comparing the ***time (t)*** determined by the counter and the 'list' of the GCD Euclidean process created in the module ***euclidean.py***. The program is able to plot the curve of the temporal complexity of the algorithm.

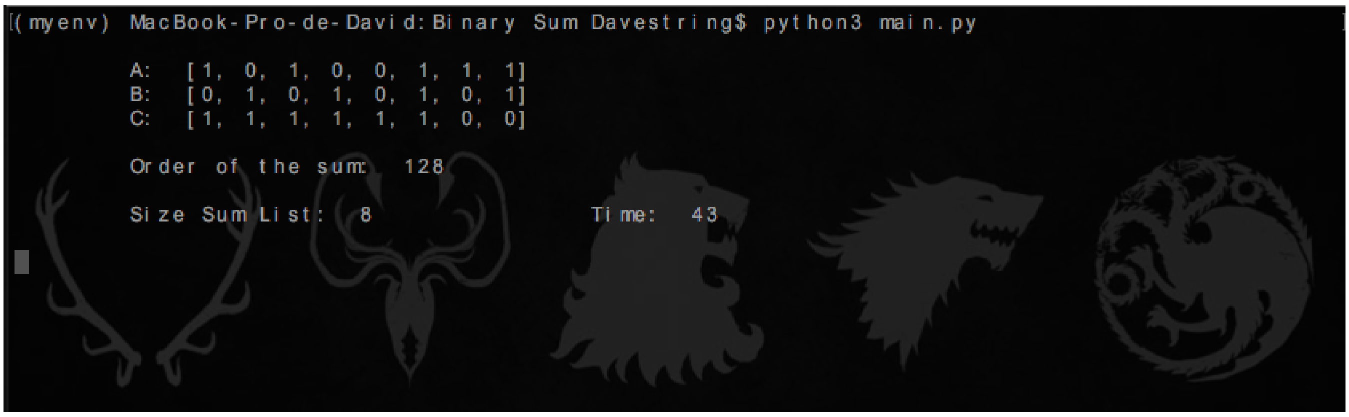
```
def graph ( n, m, count , gcd ):
    # Window title.
    plt.figure ( "Temporal Complexity of Euclidean Algorithm" )
    # Graph title.
    plt.title ( "Euclidean ( " + str ( n ) + ", " + str ( m ) + " ):",
        color = ( 0.3, 0.4, 0.6 ), weight = "bold" )
    # Parameter Time ( t ) of the graph.
    t = np.arange ( 0, count, ( count / len ( gcd ) ) )
    # Parameter Euclidean ( n ) of the list
    gcd.reverse ( )
    # Names of the axes.
    plt.xlabel ( "Euclidean ( e )", color = ( 0.3, 0.4, 0.6 ),
        family = "cursive", size = "large" )
    plt.ylabel ( "Time ( t )", color = ( 0.3, 0.4, 0.6 ),
        family = "cursive", size = "large" )
    # Proposed function: g ( n ) = ( 3/2 ) log ( n ).
    _t = list ( map ( ( lambda x: x * ( 3 / 2 ) ), t ) )
    # Plot.
    plt.plot ( gcd, _t, "#800000", linestyle = "--",
        label = "g( n ) = ( 3/2 ) log ( n )" )
    plt.plot ( gcd, t, "#778899", linewidth = 3,
        label = "E( n ) = log ( n )" )
    plt.legend ( loc = "lower right" )
    plt.show ( )
```

4 Results:

All the code shown above doesn't have significance if its operation is not shown. This section will show you the ***console*** output and the graphic of the temporal complexity of the algorithms previously mentioned.

4.1 Binary Sum Algorithm:

First test of the binary sum between 'A' and 'B', the result is stored in 'C'.



```
[(myenv) MacBook-Pro-de-David:Binary Sum Davestring$ python3 main.py

        A:   [1, 0, 1, 0, 0, 1, 1, 1]
        B:   [0, 1, 0, 1, 0, 1, 0, 1]
        C:   [1, 1, 1, 1, 1, 1, 0, 0]

        Order of the sum:   128

        Size Sum List:   8              Time:   43
```

Figure 4.1.0: Binary sum result.

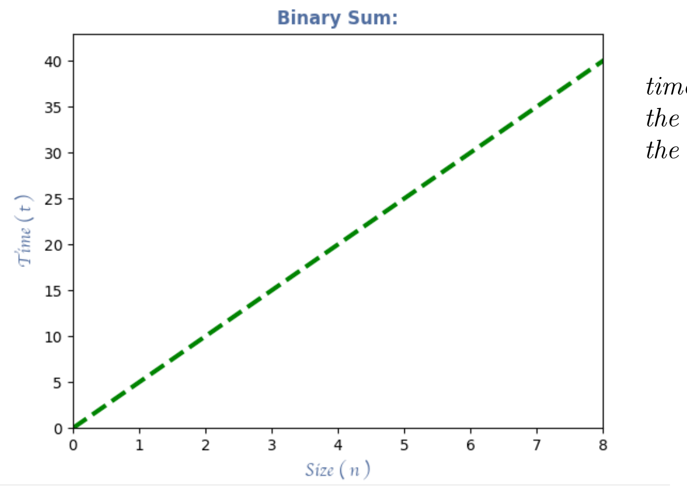
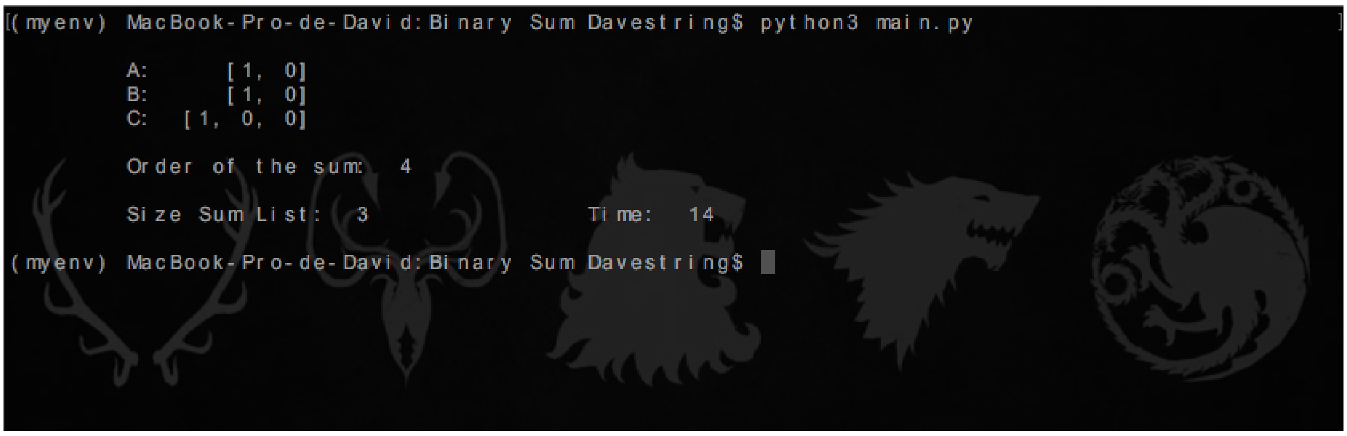


Figure 4.1.1: Plot of the Binary sum of figure 4.1.0.

Observation: *In figure 4.1.0 the shell display the time that the algorithm take to finish and the size of the resulting list, in the graph (figure 4.1.1) those are the range of the axes.*

Second test of the binary sum between 'A' and 'B', the result is stored in 'C'.



```
[(myenv) MacBook-Pro-de-David:Binary Sum Davestring$ python3 main.py

        A:      [1, 0]
        B:      [1, 0]
        C:   [1, 0, 0]

        Order of the sum:   4

        Size Sum List:   3            Time:   14

(myenv) MacBook-Pro-de-David:Binary Sum Davestring$
```

Figure 4.1.2: Binary sum result.

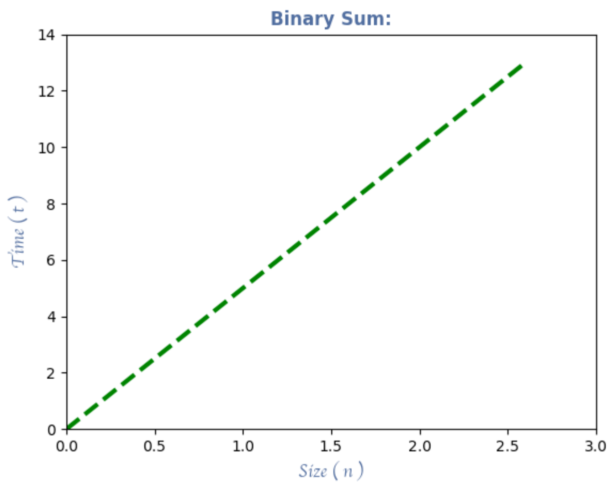


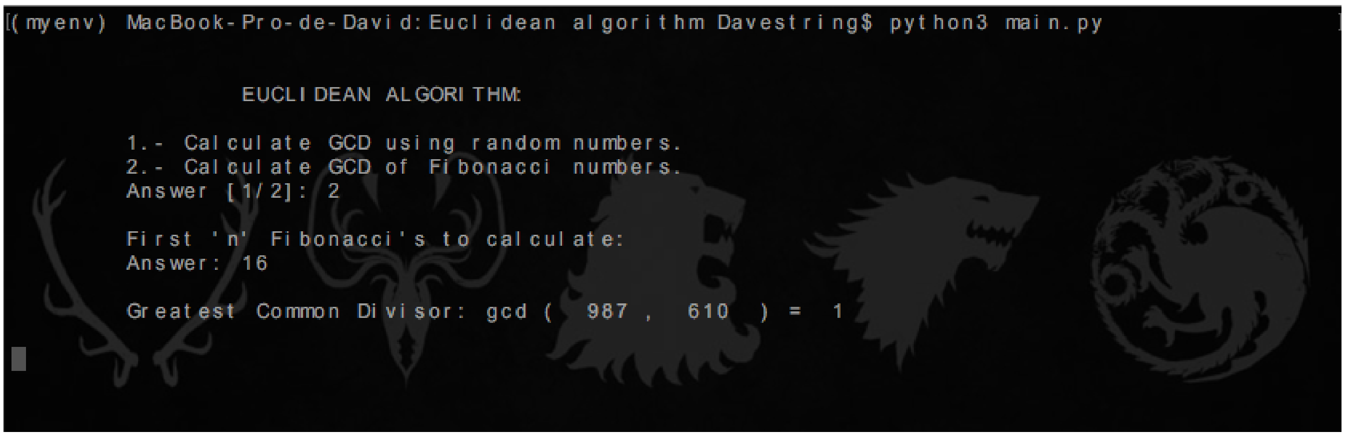
Figure 4.1.3: Plot of the Binary sum of figure 4.1.2.

time (t)	size (n)
0	0
1	0.2
2	0.4
3	0.6
4	0.8
5	1.0
6	1.2
7	1.4
8	1.6
9	1.8
10	2.0
11	2.2
12	2.4
13	2.6

Table 4.1.0: Plot points of figure 4.1.3.

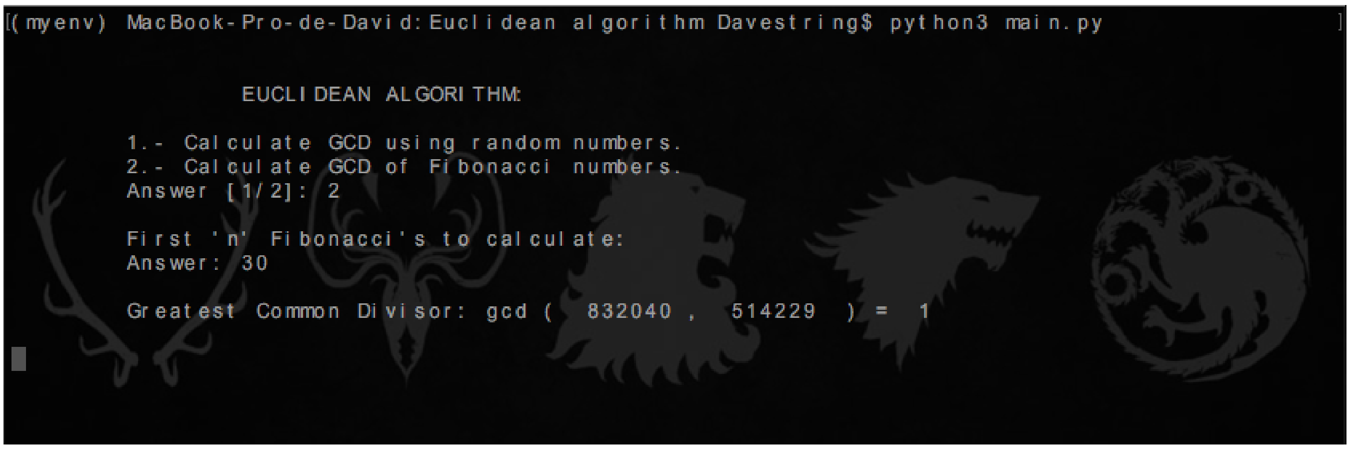
4.2 Euclidean Algorithm:

First we are going to analyze the worst case for this algorithm, and it's when the entry are Fibonacci numbers. The result will be of ***logarithmic*** order.



```
[(myenv) MacBook-Pro-de-David:Euclidean algorithm Davestring$ python3 main.py

                EUCLIDEAN ALGORITHM:

        1.- Calculate GCD using random numbers.
        2.- Calculate GCD of Fibonacci numbers.
        Answer [1/2]: 2

        First 'n' Fibonacci's to calculate:
        Answer: 16

        Greatest Common Divisor: gcd ( 987 , 610 ) = 1
```

Figure 4.2.0: Greatest Common Divider of 2 Fibonacci Numbers.



```
[(myenv) MacBook-Pro-de-David:Euclidean algorithm Davestring$ python3 main.py

                EUCLIDEAN ALGORITHM:

        1.- Calculate GCD using random numbers.
        2.- Calculate GCD of Fibonacci numbers.
        Answer [1/2]: 2

        First 'n' Fibonacci's to calculate:
        Answer: 30

        Greatest Common Divisor: gcd ( 832040 , 514229 ) = 1
```

Figure 4.2.1: Greatest Common Divider of 2 Fibonacci Numbers.

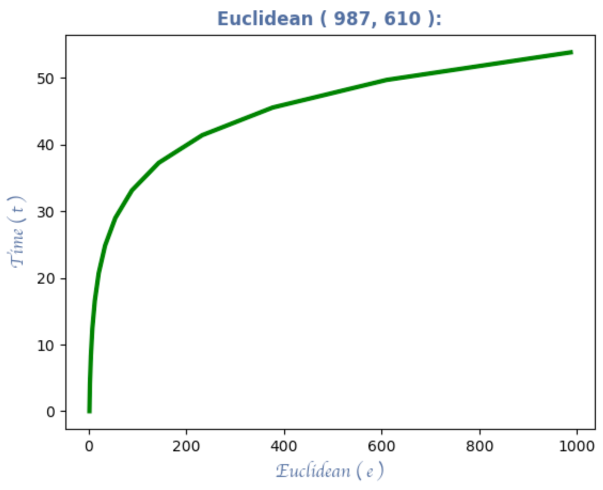


Figure 4.2.2: Plot of time (t) against euclidean (e) of Figure 4.2.0.

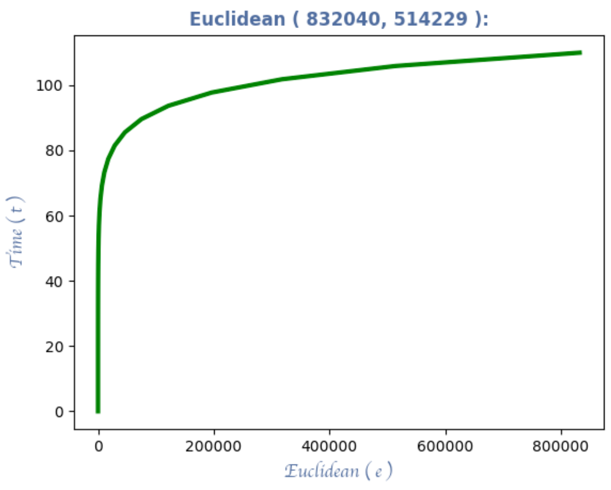
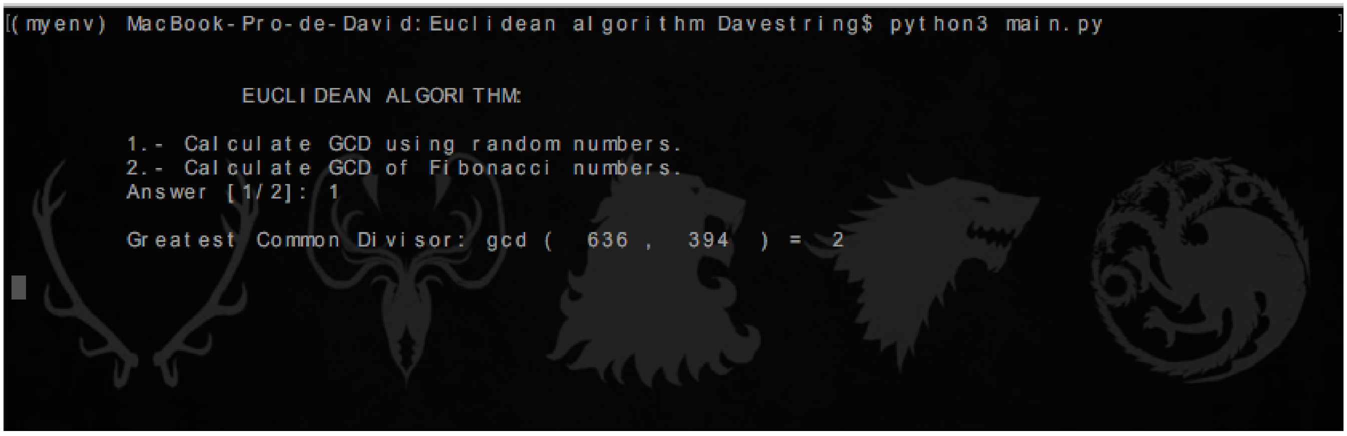


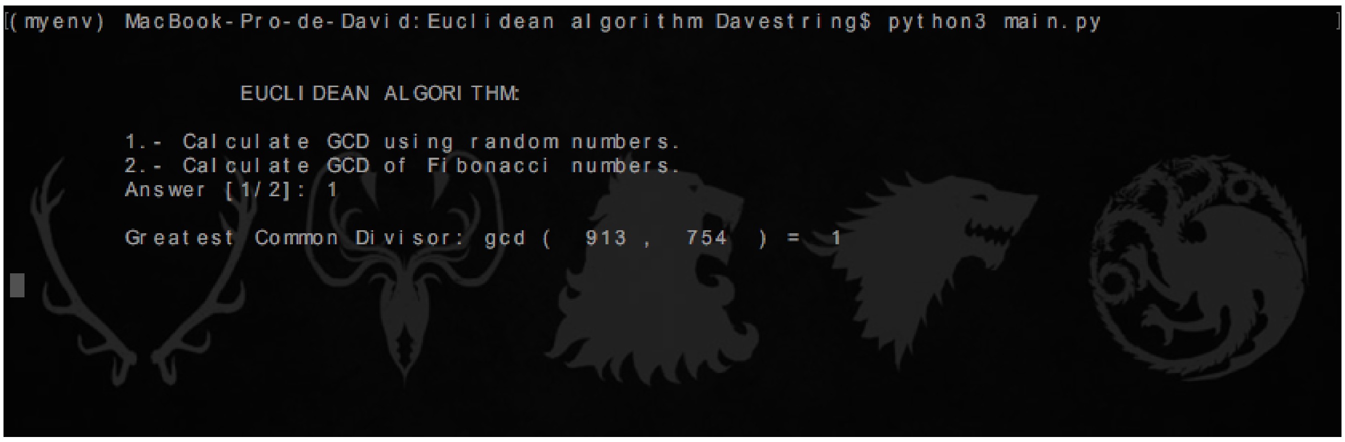
Figure 4.2.3: Plot of time (t) against euclidean (e) of Figure 4.2.1.

This other cases are basically two numbers generated randomly. They may can take a ***logarithmic*** form or not because of the plotting points, but still of ***logarithmic*** order.



```
[(myenv) MacBook-Pro-de-David:Euclidean algorithm Davestring$ python3 main.py

              EUCLIDEAN ALGORITHM:

        1.- Calculate GCD using random numbers.
        2.- Calculate GCD of Fibonacci numbers.
        Answer [1/2]: 1

        Greatest Common Divisor: gcd (  636 ,  394  ) =  2
```

Figure 4.2.4: Greatest Common Divider of 2 Random Numbers.



```
[(myenv) MacBook-Pro-de-David:Euclidean algorithm Davestring$ python3 main.py

              EUCLIDEAN ALGORITHM:

        1.- Calculate GCD using random numbers.
        2.- Calculate GCD of Fibonacci numbers.
        Answer [1/2]: 1

        Greatest Common Divisor: gcd (  913 ,  754  ) =  1
```

Figure 4.2.5: Greatest Common Divider of 2 Random Numbers.

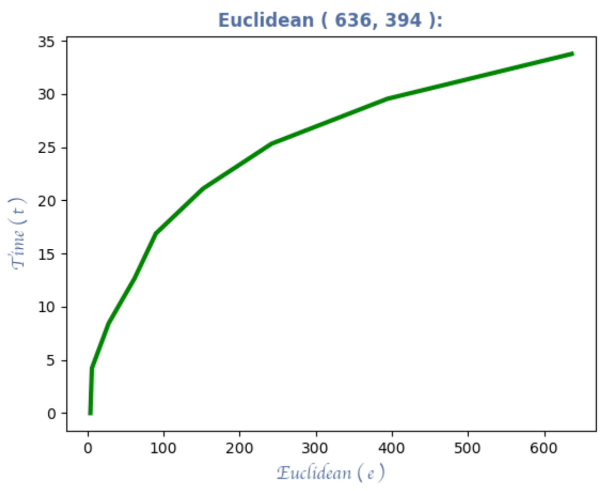


Figure 4.2.6: Plot of time (t) against euclidean (e) of Figure 4.2.4.

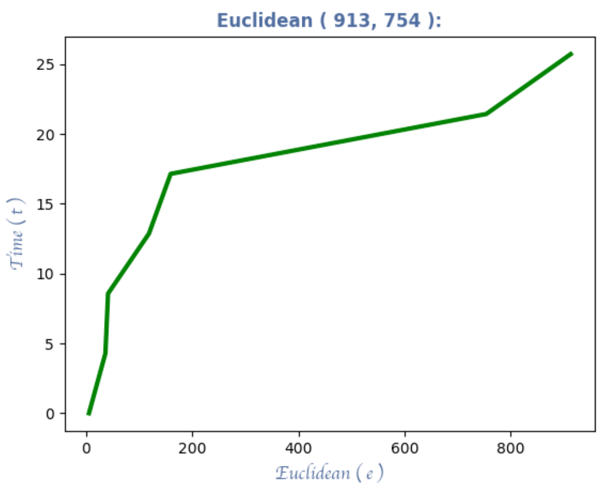


Figure 4.2.7: Plot of time (t) against euclidean (e) of Figure 4.2.5.

4.2.1 Plot Tables:

The following tables are the points with which the graphs of the Euclidean Algorithm were plotted. I didn't put the other tables because there were too many points.

- Euclidean (987, 610):

euclidean (e)	time (t)
2	0
3	4.14
5	8.28
8	12.42
13	16.57
21	20.71
34	24.85
55	29
89	33.14
144	37.28
233	41.42
377	45.57
610	49.71
987	53.85

Table 4.2.1.0: Plot points of figure 4.2.2.

- Euclidean (913, 754):

euclidean (e)	time (t)
5	0
36	4.28
41	8.57
118	12.85
159	17.14
754	21.42
913	25.71

Table 4.2.1.1: Plot points of figure 4.2.7.

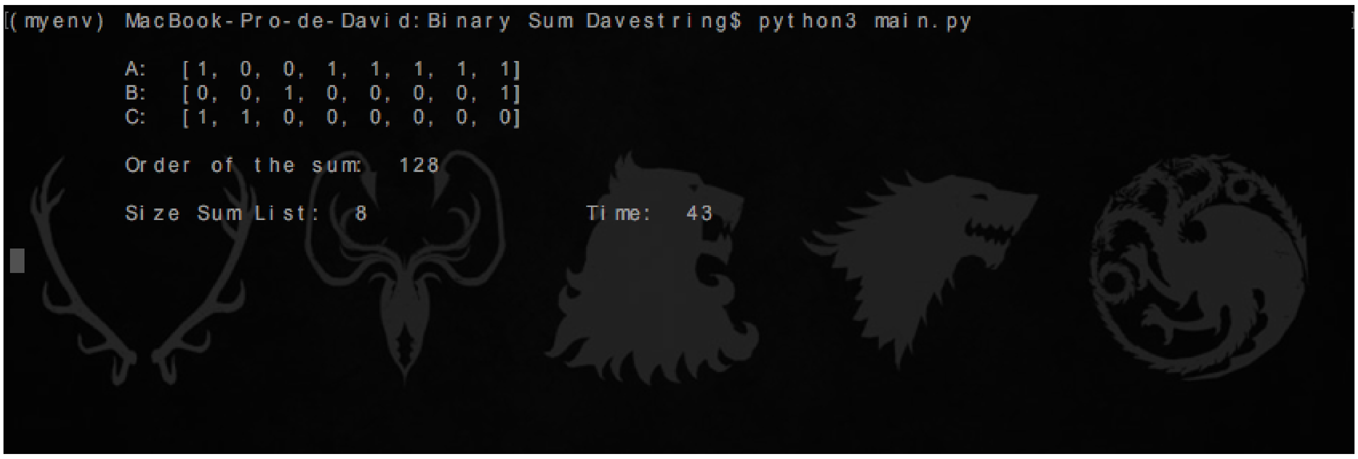
4.3 Proposed Functions:

For each Algorithm I propose a function ***g(n)*** that satisfy the following requirements:

- ***g(n)*** and ***h(n)*** are different for each algorithm.
- ***g(n)*** such that ***(binarysum, euclidean)*** $\in$ ***O(g(n))***.
- if ***(binarysum, euclidean)*** $\in$ ***O(h(n))*** then ***g(n)*** $\in$ ***O(h(n))***

4.3.1 Binary Sum g(n):

The binary sum algorithm is of ***linear*** order. The function that I propose is ***g(n)*** $= \frac{3}{2}$ ***n***.



```
(myenv) MacBook-Pro-de-David:Binary Sum Davestring$ python3 main.py

        A:   [1,  0,  0,  1,  1,  1,  1,  1]
        B:   [0,  0,  1,  0,  0,  0,  0,  1]
        C:   [1,  1,  0,  0,  0,  0,  0,  0]

        Order  of  the  sum:   128

        Size  Sum List:   8               Time:   43
```

Figure 4.3.1.0: Binary sum output.

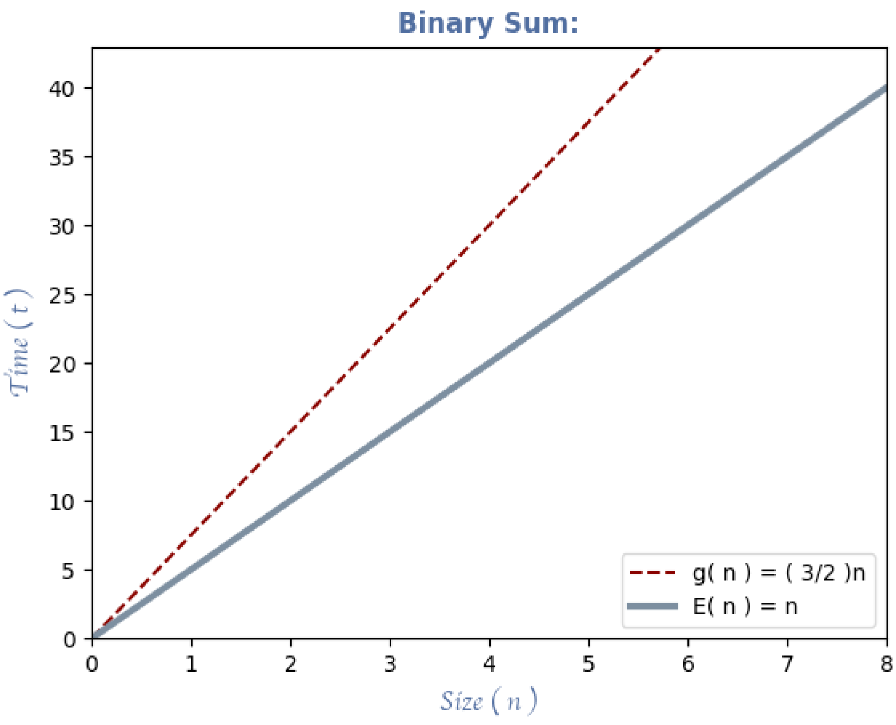
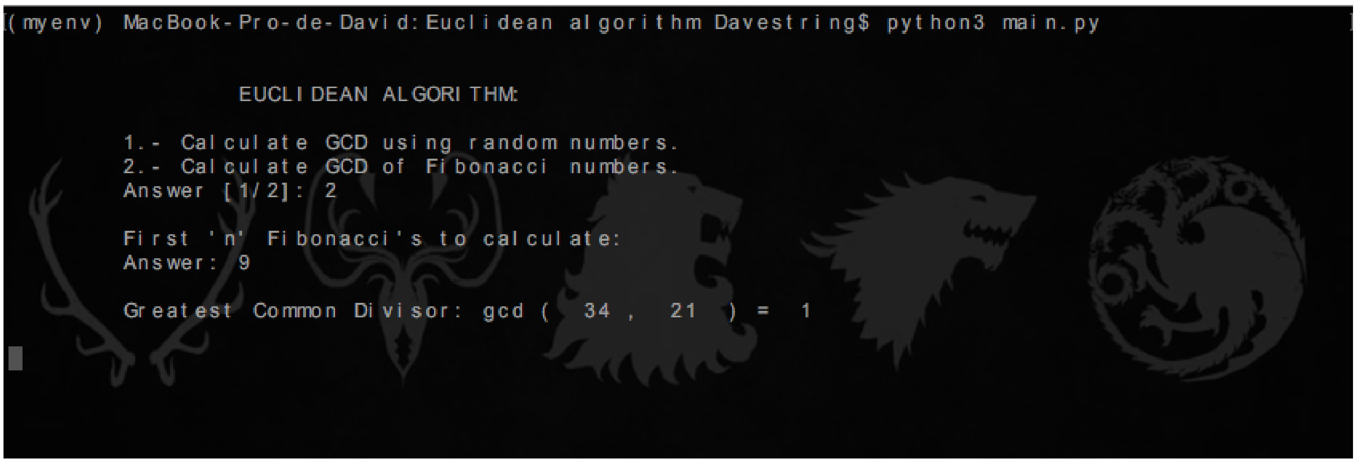


Figure 4.3.1.1: Binary sum proposed function plot.

4.3.2 Euclidean Algorithm g(n):

The Euclidean algorithm is of ***logarithmic*** order. The function that I propose is $g(\ n\ ) = \frac{3}{2}\ log(\ n\ )$.



```
(myenv) MacBook-Pro-de-David:Euclidean algorithm Davestring$ python3 main.py

            EUCLIDEAN ALGORITHM:

      1.- Calculate GCD using random numbers.
      2.- Calculate GCD of Fibonacci numbers.
      Answer [1/2]: 2

      First 'n' Fibonacci's to calculate:
      Answer: 9

      Greatest Common Divisor: gcd (  34 ,  21  ) =  1
```

Figure 4.3.2.0: Euclidean Algorithm output.

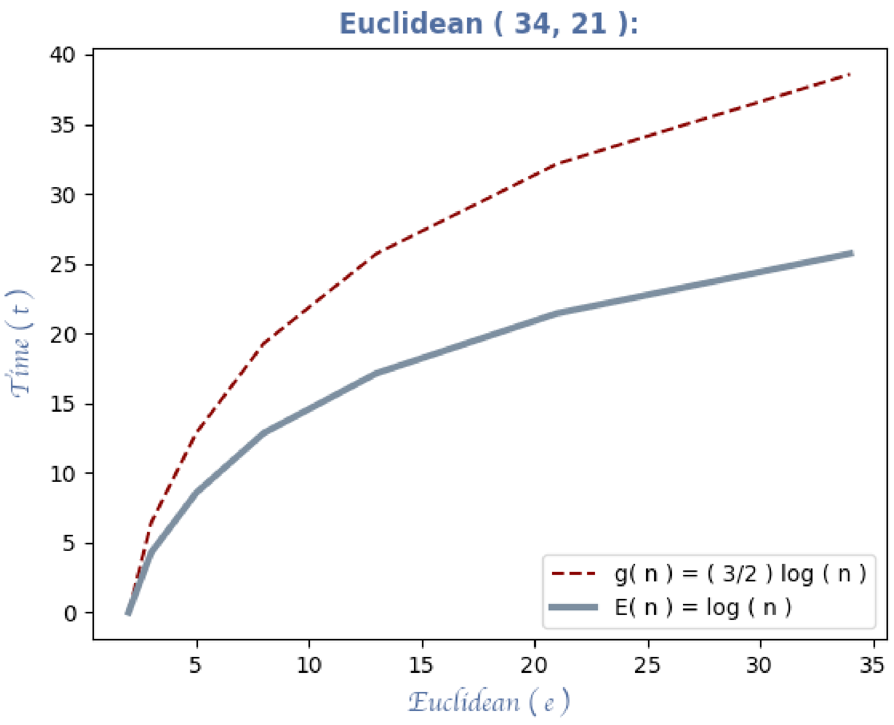


Figure 4.3.2.1: Euclidean proposed function plot.

5 Conclusion:

Always it's very interesting lead the theory to the practice, compute an algorithm and calculate the temporal complexity it's very different than doing it using all the theorems and definitions, but still very important to know what we are doing. Also, plotting result for me an interesting part, most with the ***Euclidean Algorithm***, I didn't expected that the GCD of a Fibonacci number where a sequence of the previous Fibonacci numbers, maybe I didn't stand to think on it a long time and it was obvious, but still, were a surprise for me. I actually have fun using ***matplotlib*** and ***numpy***, python never stops surprising me with all the implemented stuff that has.

6 Annexes:

Calculate the temporal complexity of the following algorithm:

```
for ( i = 0 ) to i <= ( n − 2 ) do
    k = i
    for ( j = i + 1 ) to j >= ( n − 1 ) do
        if ( A [ j ] < A [ k ] ) then
            k = j
    exchange ( A [ i ], A [ k ] )
```

Solution:

$$T(n) = C_1(n) + C_2(n-1) + C_3(\sum_{j=0}^{n-2}(t_i) + C_4(\sum_{j=0}^{n-2}(t_{i-1}) + C_5(\sum_{j=0}^{n-2}(t_{i-1}) + C_6 \tag{2}$$

- t_i calculated table:

i	t_i	range
0	n	1 < j < n
1	n - 1	2 < j < n
2	n - 2	3 < j < n
-	then	-
i	n - i	i < j < n

$$\begin{aligned}
T(n) &= C_1(n) + C_2(n-1) + C_3[\sum_{j=0}^{n-2}(n-i)] + C_4[\sum_{j=0}^{n-2}(n-i-1)] + C_5[\sum_{j=0}^{n-2}(n-i-1)] + C_6 \\
&= C_1(n) + C_2(n-1) + C_3[\sum_{j=0}^{n-2}(n)] - C_3[\sum_{j=0}^{n-2}(i)] + C_4[\sum_{j=0}^{n-2}(n)] - C_4[\sum_{j=0}^{n-2}(i)] - C_4[\sum_{j=0}^{n-2}(1)] + C_5[\sum_{j=0}^{n-2}(n)] - C_5[\sum_{j=0}^{n-2}(i)] - C_5[\sum_{j=0}^{n-2}(1)] + C_6 \\
&= C_1(n) + C_2(n-1) + C_3(n)(n-1) - C_3\frac{(n-1)(n-2)}{2} + C_4(n)(n-1) - C_4\frac{(n-1)(n-2)}{2} - C_4(n-1) + C_5(n)(n-1) - C_5\frac{(n-1)(n-2)}{2} - C_5(n-1) + C_6 \\
&= C_1(n) + C_2(n) + -C_2 + C_3(n^2) - C_3(n) - C_3(\frac{n^2-3n+2}{2}) + C_4(n^2) - C_4(n) - C_4(\frac{n^2-3n+2}{2}) - C_4(n) + C_4 + C_5(n^2) - C_5(n) - C_5(\frac{n^2-3n+2}{2}) \\
&- C_5(n) + C_5 + C_6 \\
&= (n^2)[C_3 + C_4 + C_5 - C_3(\frac{1}{2}) - C_4(\frac{1}{2}) - C_5(\frac{1}{2})] + (n)[C_1 + C_2 - C_3 - 2C_4 - 2C_5 + C_3(\frac{3}{2}) + C_4(\frac{3}{2}) + C_5(\frac{3}{2})] + [-C_2 + C_4 + C_5 + C_6 - C_3 - C_4 - C_5] \\
&= (n^2)[C_3(\frac{1}{2}) + C_4(\frac{1}{2}) + C_5(\frac{1}{2})] + (n)[C_1 + C_2 + C_3(\frac{1}{2}) - C_4(\frac{1}{2}) - C_5(\frac{1}{2})] + [-C_2 + C_6 - C_3]
\end{aligned} \tag{3}$$

Finally:

$$T(n) = an^2 + b(n) + c \Longrightarrow T(n) \in \theta(n^2) \tag{4}$$

7 Bibliography References:

[1] Baase and Van Gelder. ”Computer Algorithms: Introduction to Design and Analysis”. Addison-Wesley.