

NATIONAL POLYTECHNIC INSTITUTE
SUPERIOR SCHOOL OF COMPUTER SCIENCES

ALGORITHM ANALYSIS.

Practice 2 - Iterative vs Recursive Functions.

Hernandez Martinez Carlos David.
Burciaga Ornelas Rodrigo Andres.

davestring@outlook.com.
andii_burciaga@live.com.

Group: 3cv2.

August 31, 2017

Contents

1	Introduction:	2
2	Basic Concepts:	3
2.1	Divide-and-Conquer:	3
2.2	The Master Theorem:	3
3	Development:	4
3.1	Fibonacci Algorithm:	4
3.1.1	Main.py	4
3.1.2	Graph.py	4
3.1.3	Fibonacci Iterative:	5
3.1.4	Fibonacci Recursive:	6
3.2	Cube-Sum Algorithm:	7
3.2.1	Main.py:	7
3.2.2	Graph.py:	7
3.2.3	Cube-Sum Iterative:	8
3.2.4	Cube-Sum Recursive:	9
4	Results:	10
4.1	Fibonacci Iterative:	10
4.2	Fibonacci Recursive:	12
4.3	Cube-Sum Iterative:	14
4.4	Cube-Sum Recursive:	15
5	Annexes:	16
5.1	Bubble-Sort Formal Demonstration:	16
5.2	Iterative Fibonacci Formal Demonstration:	17
5.3	Iterative Cube-Sum Formal Demonstration:	18
5.4	Recursive Cube-Sum Formal Demonstration:	19
6	Conclusion:	20
7	Bibliography References:	21

1 Introduction:

Recurrences go hand in hand with the divide-and-conquer paradigm, because they give us a natural way to characterize the running times of divide-and-conquer algorithms. A **recurrence** is an equation or inequality that describes a function in terms of its value on smaller inputs. For example, the worst-case running time $T(n)$ of the MERGE-SORT procedure by the recurrence:

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T(\frac{n}{2}) + \theta(n) & \text{if } n > 1 \end{cases} \quad (1)$$

whose solution claimed to be $T(n) = \theta(n \log(n))$.

Recurrences can take many forms. For example, a recursive algorithm might divide subproblems into unequal sizes, such as a $\frac{2}{3}$ - to - $\frac{1}{3}$ split. If the divide and combine steps take linear time, such an algorithm would give rise to the recurrence: $T(n) = T(\frac{2n}{3}) + T(\frac{n}{3}) + \theta(n)$.

Sub-problems are not necessarily constrained to being a constant fraction of the original problem size. For example, a recursive version of linear search would create just one sub-problem containing only one element fewer than the original problem. Each recursive call would take constant time plus the time for the recursive calls it makes, yielding the recurrence: $T(n) = T(n-1) + \theta(1)$.

For solving recurrences the following methods are used:

- **Substitution method:** We guess a bound and then use mathematical induction to prove our guess correct.
- **Recursion-tree method:** Converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
- **Master method:** Provides bounds for recurrences of the form:

$$T(n) = aT(\frac{n}{b}) + f(n) \text{ where } a \geq 1 \text{ and } b > 1. \quad (2)$$

Such recurrences arise frequently. A recurrence of the form in equation (2) characterizes a divide-and-conquer algorithm that creates a sub-problems, each of which is $\frac{1}{b}$ the size of the original problem, and in which the divide and combine steps together take $f(n)$ time.

2 Basic Concepts:

When the sub-problems are large enough to solve recursively, we call that the **recursive case**. Once the sub-problems become small enough that we no longer recurse, we say that the recursion bottoms out and that we have gotten down to the **base case**. Sometimes, in addition to sub-problems that are smaller instances of the same problem, we have to solve sub-problems that are not quite the same as the original problem. We consider solving such sub-problems as part of the combine step.

2.1 Divide-and-Conquer:

The divide-and-conquer paradigm solve a problem recursively, applying three steps at each level of the recursion:

- **Divide:** The problem into a number of sub-problems that are smaller instances of the same problem.
- **Conquer:** The sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve The sub-problems in a straightforward manner.
- **Combine:** The solutions to the sub-problems into the solution for the original problem.

2.2 The Master Theorem:

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad (3)$$

Then $T(n)$ has the following asymptotic bounds:

- If $f(n) = O(n^{\log_b(a)-\epsilon})$ for some constant $\epsilon > 0$, then $\theta(n^{\log_b(a)})$.
- If $f(n) = \theta(n^{\log_b(a)})$, then $T(n) = \theta(n^{\log_b(a)} \lg(n))$.
- if $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some constant $\epsilon > 0$ and if $f(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then, $T(n) = \theta(f(n))$.

3 Development:

3.1 Fibonacci Algorithm:

The Fibonacci program it's divided in 3 different modules:

- main.py: Control the sequence of execution.
- fibonacci.py: Calculate and return a Fibonacci number $F(n)$.
- graph.py: Plot t against $F(n)$, where t it's the time that takes to find the Fibonacci number $F(n)$.

*Observation: Fibonacci where programmed **Recursive** and **Iterative**, both algorithms are totally different, but **main.py** and **graph.py** are basically the same.*

3.1.1 Main.py

In main.py, the program will ask the user to enter a number ' n ', this number will be the parameter of the function `fibonacci (...)`. After storing the returned values of `fibonacci (...)`, the program will call `graph (...)` to plot t against $F(n) = fibo$.

```
1 def main ( ):  
2     n = -1  
3     while ( n <= 0 ):  
4         n = int ( input ( "\n\tFibonacci Number to Calculate: " ) )  
5     # fibonacci ( n ): Return the fibonacci number, the counter that takes to  
6     # find that number, and a list of the fibonacci numbers before 'fibo'.  
7     fibo, count, f = fibonacci ( n )  
8     print ( "\n\tFibonacci ( ", n, " ): ", fibo, "\n" )  
9     graph ( count, fibo, f, n )  
10 main ( )
```

3.1.2 Graph.py

For plotting the result I'm using `matplotlib` and `numpy` and comparing the t determined by the counter returned in `fibonacci (...)` and the list of the previous Fibonacci numbers after reaching $F(n) = fibo$ the program is able to plot the curve of the temporal complexity of the algorithm.

```
1 def graph ( count, fibo, f, n ):  
2     # Window title.  
3     plt.figure ( "Fibonacci Iterative Algorithm" )  
4     # Graph title.  
5     plt.title ( "Fibonacci ( " + str ( n ) + " ): " + str ( fibo ) )  
6     # Parameter Time ( t ) and Fibonacci ( n ) of the graph.  
7     t = np.arange ( 0, count, ( count / ( len ( f ) + 1 ) ) )  
8     _t = list ( map ( ( lambda x: x * ( 3 / 2 ) ), t ) )  
9     _f = np.arange ( 0, len ( f ) + 1 )  
10    # Names of the axes.  
11    plt.xlabel ( "Time ( t )", color = ( 0.3, 0.4, 0.6 ), family = "cursive",  
12                size = "large" )  
13    plt.ylabel ( "Fibonacci ( f )", color = ( 0.3, 0.4, 0.6 ), family = "cursive",  
14                size = "large" )  
15    # Plot.  
16    plt.plot ( _f, t, "#778899", linewidth = 3, label = "T( n ) = ( ( n )" )  
17    plt.plot ( _f, _t, "#800000", linestyle = "-", label = "g( n ) = ( 3/2 )( n )" )  
18    plt.legend ( loc = "lower right" )  
19    plt.show ( )
```

3.1.3 Fibonacci Iterative:

The Fibonacci numbers are the numbers in the following integer sequence, called the Fibonacci sequence, and characterized by the fact that every number after the first two is the sum of the two preceding ones:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \dots \quad (4)$$

The code bellow, calculate a Fibonacci number where $F(n) = \text{fibonacci}$ it's the Fibonacci number to find, $F(n-1) = a$ and $F(n-2) = b$ are the two previous ones.

```
1 def fibonacci ( n ):
2     fibo , a , b = 0 , 1 , 0
3     for i in range ( 1 , n ):
4         fibo = a + b
5         b = a
6         a = fibo
7     return fibo
```

If we want to calculate the computational time $T(n)$ that the algorithm takes to find a number, it's necessary to put a counter in each line of our code.

```
1 def fibonacci ( n ):
2     count , fibo , a , b , f = 1 , 1 , 1 , 0 , [ ]
3     for i in range ( 1 , n ):
4         count += 1
5         fibo = a + b
6         count += 1
7         f.append ( fibo )
8         count += 1
9         b = a
10        count += 1
11        a = fibo
12        count += 1
13    count += 1
14    return fibo , count , f
```

Observation: Now, apart of return the Fibonacci number result, the code will also return the counter and a list of all the previous Fibonacci numbers after 'n'.

Observation: The counter and the list are necessary to plot t against n where t it's the computation time.

3.1.4 Fibonacci Recursive:

The sequence F_n of Fibonacci numbers is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2} \quad (5)$$

with seed values:

$$F_n = 1, F_n = 2 \quad (6)$$

```

1 def fibonacci ( n ):
2     if ( n == 1 or n == 2 ):
3         return 1
4     else:
5         return fibonacci ( n - 1 ) + fibonacci ( n - 2 )

```

For calculate the computation time of the algorithm it's necessary to modify some lines of the code:

```

1 def fibonacci ( n, count ):
2     count += 1
3     if ( n == 1 or n == 2 ):
4         return 1, count
5     else:
6         a, count = fibonacci ( n - 1, count )
7         b, count = fibonacci ( n - 2, count )
8         return a + b, count

```

As we can see, now the Fibonacci function receive two parameters, the 'n' and the counter to calculate the computational time, also in the *else* sentence, the return of the recursions are stored in 3 variables *a*, *b* and *counter* where *a* and *b* are the two previous Fibonacci's.

The final return statement of the algorithm will be stored in main, were the Fibonacci function were initially called, and the values will be stored in a *list of tuples*, where the values stored are ($F(n)$, *time*) where $F(n)$ it's the Fibonacci number and *time* it's the computational time that the algorithm takes to find that number. For Example: $n = 10$.

```
[(1, 1), (1, 1), (2, 3), (3, 5), (5, 9), (8, 15), (13, 25), (21, 41), (34, 67)]
```

Figure 3.1.4.0: Return statement of Fibonacci recursive.

According to the Figure 3.1.4.0:

Fibonacci	Time
1	1
1	1
2	3
3	5
5	9
8	15
21	41
34	67

Table 1: Fibonacci number against Computation Time.

3.2 Cube-Sum Algorithm:

The Cube-Sum program it's divided in 3 different modules:

- main.py: Control the sequence of execution.
- cube.py: Calculate and return a the sum of the nth numbers to the power '3': $C(n)$.
- graph.py: Plot t against $C(n)$, where t it's the time that takes to find the sum of the first nth numbers to the power '3' $C(n) = sum$.

*Observation: Cube-Sum where programmed **Recursive** and **Iterative**, both algorithms are totally different, but **main.py** and **graph.py** are basically the same.*

3.2.1 Main.py:

In main.py, the program will ask the user to enter a number ' n ', this number will be the parameter of the function `cube (...)`. After storing the returned values of `cube (...)`, the program will call `graph (...)` to plot $time$ against $C(n) = sum$.

```
1 def main ( ):  
2     n = int ( input ( "\n\tNumber to calculate firts n-Cubes: " ) )  
3     # cube ( n ): Return the _sum of the first 'n' cubes, the computational  
4     # time of the algorithm and a list of the sum of nth cubes.  
5     _sum, count, cubelist = cube ( n )  
6     print ( "\n\tThe sum of the first ", n, " cubes is: C ( ", n, " ) = ", _sum, "\n" )  
7     graph ( _sum, count, cubelist, n )  
8 main ( )
```

3.2.2 Graph.py:

For plotting the result Im using *matplotlib* and *numpy* and comparing the $time (t)$ determined by the counter returned in `cube (...)` and the list of the previous sum of nth cubes after reaching $C(n) = sum$ the program is able to plot the curve of the temporal complexity of the algorithm. Also, in line 11, I propose a function $g(n)$ that $T(n) \in O(g(n))$ where $T(n)$ it's the computational time of the algorithm.

```
1 def graph ( _sum, count, cubelist, n ):  
2     # Window title.  
3     plt.figure ( "First 'n' Cubes: Iterative Algorithm" )  
4     # Graph title.  
5     plt.title ( "CubeSum ( " + str ( n ) + " ): " + str ( _sum ) )  
6     # Parameter Time ( t ) of the graph.  
7     t = np.arange ( 0, count, ( count / ( len ( cubelist ) ) ) )  
8     # Parameter CubeSum ( n ) of the graph.  
9     c = np.arange ( 0, len ( cubelist ) )  
10    # Proposed function:  $g(n) = (3/2)n$ .  
11    _t = list ( map ( ( lambda x: x * ( 3 / 2 ) ), t ) )  
12    # Names of the axes.  
13    plt.ylabel ( "Time ( t )", color = ( 0.3, 0.4, 0.6 ), family = "cursive",  
14                size = "large" )  
15    plt.xlabel ( "CubeSum ( n )", color = ( 0.3, 0.4, 0.6 ), family = "cursive",  
16                size = "large" )  
17    # Plot.  
18    plt.plot ( c, _t, "#800000", linestyle = "—", label = " $g(n) = (3/2)(n)$ " )  
19    plt.plot ( c, t, "#778899", linewidth = 3, label = " $T(n) = (n)$ " )  
20    plt.legend ( loc = "lower right" )  
21    plt.show ( )
```

3.2.3 Cube-Sum Iterative:

The Cube-Sum are the numbers in the following integer sequence, and characterized by the fact that every number is the sum of all the previous one to the power of '3'.

- Normal number to the power of '3' sequence:

$$0, 1, 8, 27, 64, \dots \quad (7)$$

- Sum of the number to the power of '3' sequence:

$$0, 1, 9, 36, 100, \dots \quad (8)$$

The code below, calculate the Cube-Sum of any number where $C(n) = \text{sum}$ it's the sum result.

```
1 def cube ( n ):  
2     _sum = 0  
3     # Range: From 1 <= i <= n.  
4     for i in range ( 1, n + 1 ):  
5         _sum = _sum + ( i * i * i )  
6     # Return statement.  
7     return _sum
```

If we want to calculate the computational time $T(n)$ that the algorithm takes to find a number, it's necessary to put a counter in each line of our code.

```
1 def cube ( n ):  
2     _sum, count, cubelist = 0, 1, [ ]  
3     # Range: From 1 <= i <= n.  
4     for i in range ( 1, n + 1 ):  
5         count += 1  
6         _sum = _sum + ( i * i * i )  
7         count += 1  
8         cubelist.append ( _sum )  
9         count += 1  
10    # Return statement.  
11    count += 1  
12    return _sum, count, cubelist
```

***Observation:** Now, apart of return the sum result, the code will also return the counter and a list of all the previous numbers of the sequence after 'n'.*

***Observation:** The counter and the list are necessary to plot t against n where t it's the computation time.*

3.2.4 Cube-Sum Recursive:

The sequence C_n of Cube-Sum numbers is defined by the recurrence relation:

$$C_n = C_{n-1} + (n^3) \quad (9)$$

with seed values:

$$C_n = 1 \quad (10)$$

```

1 def cube ( n ):
2     if ( n == 1 ):
3         return 1
4     else:
5         return cube ( n - 1 ) + ( n * n * n )

```

For calculate the computation time of the algorithm it's necessary to modify some lines of the code:

```

1 def cube ( n, count ):
2     count += 1
3     if ( n == 1 ):
4         return 1, count
5     else:
6         a, count = cube ( n - 1, count )
7         b = n*n*n
8         return a + b, count

```

As we can see, now the Cube function receive two parameters, the 'n' and the counter to calculate the computational time, also in the *else* sentence, the return of the recursions are stored in 2 variables *a* and *counter* where *a* its the sum of the previous Cubes.

The final return statement of the algorithm will be stored in main, were the Cube function were initially called, and the values will be stored in a *list of tuples* where the values stored are ($C(n)$, *time*) where $C(n)$ it's the Cube sum and *time* it's the computational time that the algorithm takes to find that number. For Example: $n = 10$.

```
[(1, 1), (9, 2), (36, 3), (100, 4), (225, 5), (441, 6), (784, 7), (1296, 8), (2025, 9), (3025, 10)]
```

Figure 3.2.4.0: Return statement of CubeSum recursive.

According to the Figure 3.2.4.0:

Cubesum	Time
1	1
9	2
36	3
100	4
225	5
441	6
784	7
1296	8
2025	9
3025	10

Table 2: Cube sum against Computation Time.

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. Also, I attach a table with the plot points for each test.

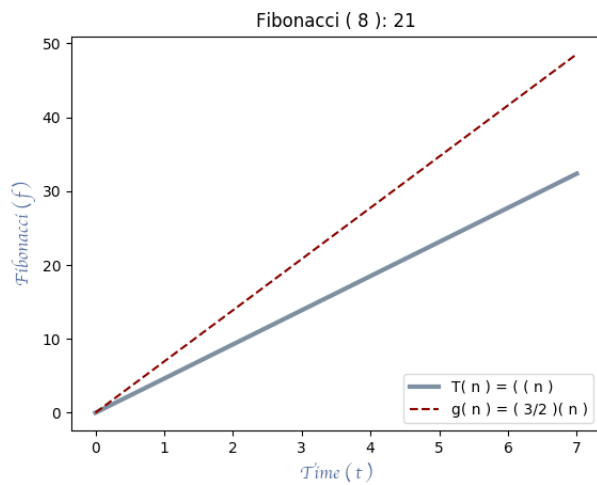
4.1 Fibonacci Iterative:

First test of the Fibonacci Iterative Algorithm. The program will plot the **time** that the algorithm takes to find the **eight** Fibonacci number.

```
(myenv) MacBook-Pro-de-David:Iterative Davestrings$ python3 main.py
Fibonacci Number to Calculate: 8
Fibonacci ( 8 ): 21
```



Figure 4.1.0: Console output of Fibonacci Iterative.



Time (t)	Fibonacci (n)
0	F (1) = 1
4.80	F (2) = 1
9.60	F (3) = 2
14.40	F (4) = 3
19.20	F (5) = 5
24	F (6) = 8
28.80	F (7) = 13
33.60	F (8) = 21

Table 3: Plot points of Figure 4.1.1.

Figure 4.1.1: Plot of Figure 4.1.0.

Observation: The plot has two curves, the **blue** one it's the computation time of our algorithm $T(n) = n$ and the **red** one it's the proposed function $g(n) = \frac{3}{2}n$, where $T(n) \in O(g(n))$.

Second test of the Fibonacci Iterative Algorithm. The program will plot the *time* that the algorithm takes to find the *fifteenth* Fibonacci number.

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

Fibonacci Number to Calculate: 15

Fibonacci ( 15 ): 610
```



Figure 4.1.2: Console output of Fibonacci Iterative.

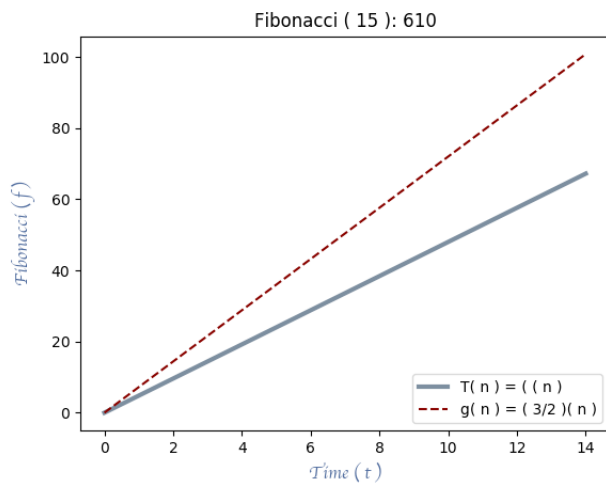


Figure 4.1.3: Plot of Figure 4.1.2.

Time (t)	Fibonacci (n)
0	F (1) = 1
4.80	F (2) = 1
9.60	F (3) = 2
14.40	F (4) = 3
19.20	F (5) = 5
24	F (6) = 8
28.80	F (7) = 13
33.60	F (8) = 21
38.40	F (9) = 34
43.20	F (10) = 55
48	F (11) = 89
52.80	F (12) = 144
57.60	F (13) = 233
62.40	F (14) = 377
67.20	F (15) = 610

Table 4: Plot points of Figure 4.1.3.

Observation: As we can see, according to section 3.3 It is demonstrated that the algorithm is of linear order.

4.2 Fibonacci Recursive:

First test of the Fibonacci Recursive Algorithm. The program will plot the *time* that the algorithm takes to find the *eight* Fibonacci number.

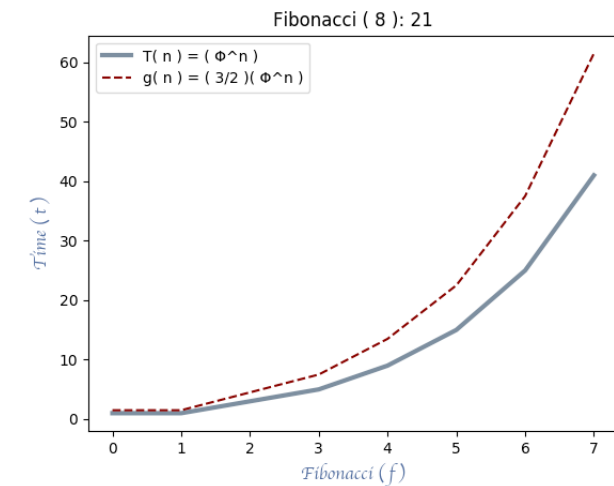
```
(myenv) MacBook-Pro-de-David:Recursive Davestring$ python3 main.py

Fibonacci Number to Calculate: 8

Fibonacci ( 8 ): 21
```



Figure 4.2.0: Console output of Fibonacci Recursive.



Time (t)	Fibonacci (n)
1	F (1) = 1
1	F (2) = 1
3	F (3) = 2
5	F (4) = 3
9	F (5) = 5
15	F (6) = 8
25	F (7) = 13
41	F (8) = 21

Table 5: Plot points of Figure 4.2.1.

Figure 4.2.1: Plot of Figure 4.2.0.

Observation: The plot has two curves, the *blue* one it's the computation time of our algorithm $T(n) = (\phi^n)$ and the *red* one it's the proposed function $g(n) = \frac{3}{2}(\phi^n)$, where $T(n) \in O (g(n))$.

Second test of the Fibonacci Recursive Algorithm. The program will plot the *time* that the algorithm takes to find the *fifteenth* Fibonacci number.

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

Fibonacci Number to Calculate: 15

Fibonacci ( 15 ): 610
```



Figure 4.2.2: Console output of Fibonacci Relative.

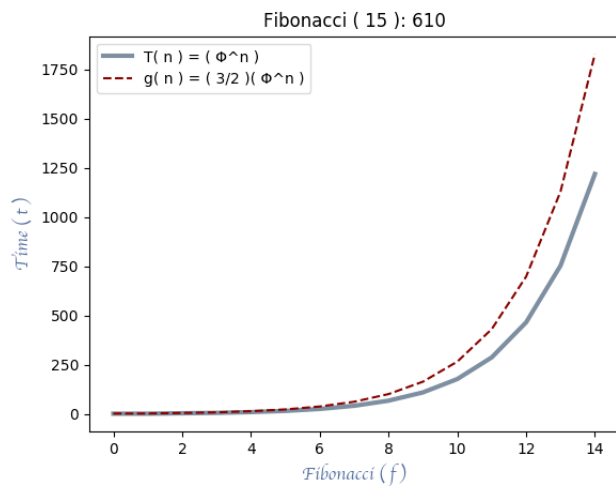


Figure 4.2.3: Plot of Figure 4.2.2.

Time (t)	Fibonacci (n)
1	F (1) = 1
1	F (2) = 1
3	F (3) = 2
5	F (4) = 3
9	F (5) = 5
15	F (6) = 8
25	F (7) = 13
41	F (8) = 21
67	F (9) = 34
109	F (10) = 55
177	F (11) = 89
287	F (12) = 144
465	F (13) = 233
753	F (14) = 377
1219	F (15) = 610

Table 6: Plot points of Figure 4.2.3.

4.3 Cube-Sum Iterative:

First test of the Cubes-Sum Iterative Algorithm. The program will plot the **time** that the algorithm takes to find the sum of the **tenth** first numbers to the power '3', excluding '0'.

```

((myenv) MacBook-Pro-de-David:Iterative Davestring$ python3 main.py

Number to calculate firts n-Cubes: 10

The sum of the first 10 cubes is: C ( 10 ) = 3025

```




Figure 4.3.0: Console output of Cubes-Sum Iterative.

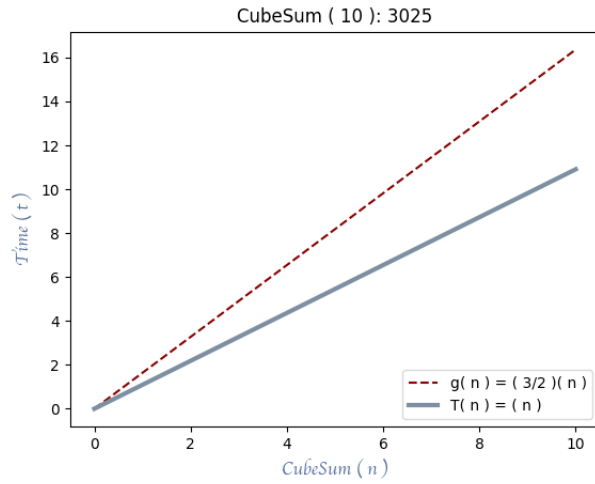


Figure 4.3.1: Plot of Figure 4.3.0.

Time (t)	CubeSum (n)
0	C (0) = 0
1.2	C (1) = 1
2.4	C (2) = 9
3.6	C (3) = 36
4.8	C (4) = 100
6.0	C (5) = 224
7.2	C (6) = 441
8.4	C (7) = 784
9.6	C (8) = 1296
10.8	C (9) = 2025
12	C (10) = 3025

Table 7: Plot points of Figure 4.3.1.

Observation: The plot has two curves, the **blue** one it's the computation time of our algorithm $T(n) = n$ and the **red** one it's the proposed function $g(n) = \frac{3}{2}n$, where $T(n) \in O(g(n))$.

4.4 Cube-Sum Recursive:

First test of the Cubes-Sum Recursive Algorithm. The program will plot the **time** that the algorithm takes to find the sum of the **tenth** first numbers to the power '3', excluding '0'.

```
(myenv) MacBook-Pro-de-David:Recursive Davestring$ python main.py

Number to calculate firts n-Cubes: 10

Cubesum ( 10 ): 3025
```



Figure 4.4.0: Console output of Cubes-Sum Recursive.

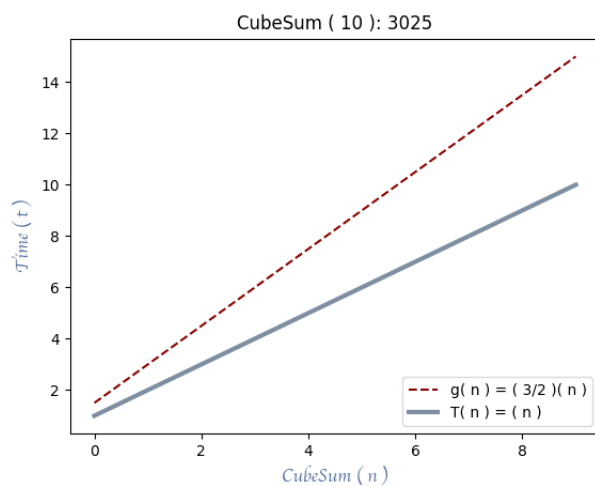


Figure 4.4.1: Plot of Figure 4.4.0.

Time (t)	CubeSum (n)
0	C (0) = 0
1	C (1) = 1
2	C (2) = 9
3	C (3) = 36
4	C (4) = 100
5	C (5) = 224
6	C (6) = 441
7	C (7) = 784
8	C (8) = 1296
9	C (9) = 2025
10	C (10) = 3025

Table 8: Plot points of Figure 4.4.1.

Observation: The plot has two curves, the **blue** one it's the computation time of our algorithm $T(n) = n$ and the **red** one it's the proposed function $g(n) = \frac{3}{2}n$, where $T(n) \in O(g(n))$.

5 Annexes:

5.1 Bubble-Sort Formal Demonstration:

Calculate the temporal complexity of the Bubble-Sort algorithm:

```

1 for i = 1 to i <= n - 1 do
2   for j = n - 1 to j >= i do
3     if ( A [ j ] < A [ j - 1 ] )
4       exchange: A [ j ] with A [ j - 1 ]

```

- *Where $n = A.length$:*

As we can see, the algorithm sort the array A in his ascendant form, so, the worst case, it's when the array already it's sorted but in his descendant form.

- ***Solution:***

$$T(n) = C_1(n) + C_2\left[\sum_{i=1}^{n-1}(t_i)\right] + C_3\left[\sum_{i=1}^{n-1}(t_{i-1})\right] + C_4 \quad (11)$$

For t_i :

i	t_i	range
1	n - 1	n - 1 ≥ j ≥ 1
2	n - 2	n - 1 ≥ j ≥ 2
3	n - 3	n - 1 ≥ j ≥ 3
-	then	-
i	n - i	n - 1 ≥ j ≥ i

Table 9: t_i value.

Let $t_i = n - 1$, substituting in (4):

$$T(n) = C_1(n) + C_2\left[\sum_{i=1}^{n-1}(n - i)\right] + C_3\left[\sum_{i=1}^{n-1}((n - 1) - i)\right] + C_4 \quad (12)$$

$$= C_1(n) + C_2\left[\sum_{i=1}^{n-1}(n)\right] - C_2\left[\sum_{i=1}^{n-1}(i)\right] + C_3\left[\sum_{i=1}^{n-1}(n - 1)\right] - C_3\left[\sum_{i=1}^{n-1}(i)\right] + C_4 \quad (13)$$

Using:

- (i) $\sum_{i=m}^n(k) = k(n - m + 1)$ where k constant.
- (ii) $\sum_{i=m}^n(i) = \frac{(n+1-m)(n+m)}{2}$.

Then:

$$T(n) = C_1(n) + C_2[n(n - 1)] - C_2\left[\frac{(n - 1)(n)}{2}\right] + C_3[(n - 1)(n - 1)] - C_3\left[\frac{(n - 1)(n)}{2}\right] + C_4 \quad (14)$$

$$= C_1(n) + C_2[n^2 - n] - C_2\left[\frac{n^2 - n}{2}\right] + C_3[n^2 - 2n + 1] - C_3\left[\frac{n^2 - n}{2}\right] + C_4 \quad (15)$$

$$= C_1(n) + C_2n^2 - C_2n - C_2\frac{n^2}{2} + C_2\frac{n}{2} + C_3n^2 - 2C_3n + C_3 - C_3\frac{n^2}{2} + C_3\frac{n}{2} + C_4 \quad (16)$$

$$= [n^2][C_2\frac{1}{2} + C_3\frac{1}{2}] + [n][C_1 - C_2\frac{1}{2} - C_3\frac{3}{2}] + [C_3 + C_4] \quad (17)$$

$$= an^2 + bn + c \quad (18)$$

Finally:

$$T(n) \in \theta(n^2) \quad (19)$$

5.2 Iterative Fibonacci Formal Demonstration:

Demonstrate that the Fibonacci iterative algorithm has *linear* order:

```
1 a, b = 1, 0
2 for i = 0 to i < n do
3     f = a + b
4     b = a
5     a = f
```

- *Demonstration:*

$$T(n) = C_1 + C_2[n + 1] + C_3[n] + C_4[n] + C_5[n] \quad (20)$$

$$= C_1 + C_2[n] + C_2 + C_3[n] + C_4[n] + C_5[n] \quad (21)$$

$$= [n][C_2 + C_3 + C_4 + C_5] + [C_1 + C_2] \quad (22)$$

Substituting:

(i) Let $a = C_2 + C_3 + C_4 + C_5$.

(ii) Let $b = C_1 + C_2$.

Then:

$$T(n) = an + b \quad (23)$$

Finally:

$$T(n) \in O(n). \quad (24)$$

5.3 Iterative Cube-Sum Formal Demonstration:

Demonstrate that the Cube-Sum iterative algorithm has *linear* order:

```
1 sum = 0
2 for i = 1 to i <= n do:
3     sum = sum + ( i * i * i )
```

- *Demonstration:*

$$T(n) = C_1 + C_2(n + 1)C_3(n) \quad (25)$$

$$= [n][C_2 + C_3] + [C_1 + C_2] \quad (26)$$

Substituting:

(i) Let $a = C_2 + C_3$.

(ii) Let $b = C_1 + C_2$.

Then:

$$T(n) = an + b \quad (27)$$

Finally:

$$T(n) \in O(n). \quad (28)$$

5.4 Recursive Cube-Sum Formal Demonstration:

Demonstrate that the Cube-Sum recursive algorithm has *linear* order:

```

1  if ( n == 1 ) return 1
2  else return S ( n - 1 ) + n*n*n

```

$$M(n) = \begin{cases} 1 & \text{if } n = 1 \\ M(n-1) + 1 & \text{if } n > 1 \end{cases} \quad (29)$$

• *Demonstration:*

$$M(n) = M(n-1) + 1 \quad (30)$$

$$= [M(n-2) + 1] + 1 \quad (31)$$

$$= [(M(n-3) + 1) + 1] + 1 \quad (32)$$

$$= [((M(n-4) + 1) + 1) + 1] + 1 \quad (33)$$

$$(34)$$

So on:

$$M(n) = M(n-i) + i \quad (35)$$

Substituting:

$$(i) \quad n - i = 1.$$

$$(ii) \quad i = n - 1.$$

then:

$$M(n) = M(1) + (n-1) \quad (36)$$

$$(37)$$

Finally:

$$M(n) \in O(n). \quad (38)$$

6 Conclusion:

As we have seen up to this point in our career, there are many forms to solve the same problem. We where analyzing the Fibonacci algorithm in this practice using a *recursive* and a *iterative* code. But they aren't the only ways to solve this problem, in **practice 1** I wrote the Fibonacci algorithm using a *generator*, and also I know we can use *memories* to help optimize the algorithm. At the end, we as programmers will use the one who solve the problem better, using the computer resources optimally, and all our knowledge gained over this time. It's quite interesting see that, apart of being the same problem solved in two different ways, the graph of the recursive algorithm vs the iterative are very different because of the time that our computer takes to solve the same problem.

- Hernandez Martinez Carlos David.

Sometimes we think that the recursion is better than iterative algorithm, in this case we demonstrate that this premise is not always true, because there are variables in the environmental development that influences in the computational time like hardware, memory, cores, processors, etc. The recursion is a way to solve problems but in code is not always the better solution, sometimes the lines of code is higher than the other one and therefore many times the constants of time grow in number, I think recursion is better in the cases when the complex of the development "code" is bigger, the best way to have an idea of what kind of algorithm implementation is better than the other is to calculate the order complexity.

- Burciaga Ornelas Rodrigo Andres.

7 Bibliography References:

- [1] Baase and Van Gelder. "Computer Algorithms: Introduction to Design and Analysis". Addison-Wesley.
- [2] Thomas H. Cormen. "Introduction to Algorithms". The MIT press.