Ministry of Education, Research and Culture
Technical University of Moldova
Software Engineering and Automation Departments

REPORT

Laboratory work No. 1

Course: Analysis and Design of Algorithms

Theme: Study and Empirical Analysis of Algorithms for Determining Fibonacci N-th Term

Author: Corneliu Catlabuga
FAF-213

Checked by: univ. assist.
Cristofor Fiștic

Chisinau 2023

Objective:

Study and analyze different algorithms for determining Fibonacci n-th term.

Task:

1. Implement at least 3 algorithms for determining Fibonacci n-th term;
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

Theoretical considerations:

An alternative to mathematical analysis of complexity is empirical analysis.

This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Implementation, practical results:

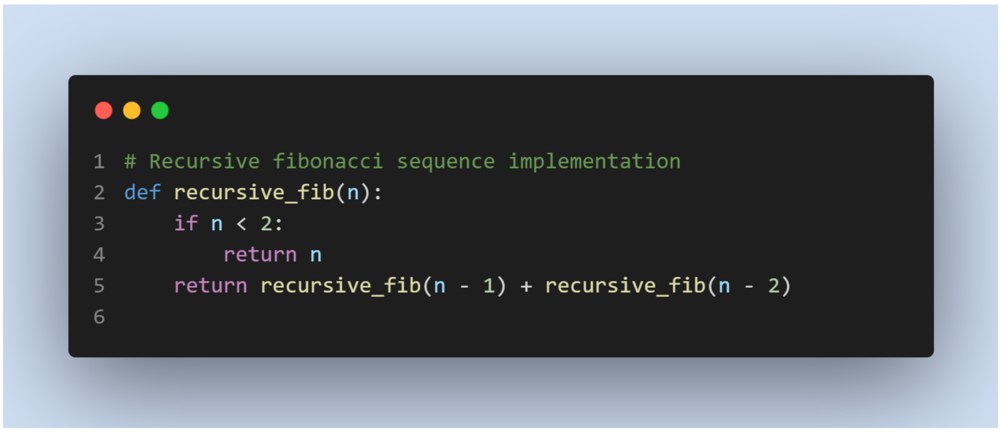
Prerequisites:

- The algorithms and tests are executed in the Python programming language.
- The used libraries are matplotlib, used for generating the graphs and time used for computing the elapsed time of the executed functions.
- The program has three arrays of numbers:

1. [0, 5, 10, 15, 20, 25] used for the recursive algorithm due to the inefficient nature of it;
2. [5, 12, 21, 42, 70, 101, 256, 589, 812, 1112, 1300] used for the Binet's formula and Lucas' formula due to the high computed numbers;
3. [501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162, 3981, 5012, 6310, 7943, 10000, 12589, 15849] used for the rest of the algorithms.

Implementation:

Recursive method



```python
# Recursive fibonacci sequence implementation
def recursive_fib(n):
    if n < 2:
        return n
    return recursive_fib(n - 1) + recursive_fib(n - 2)

```

Fig 1: Recursive method function

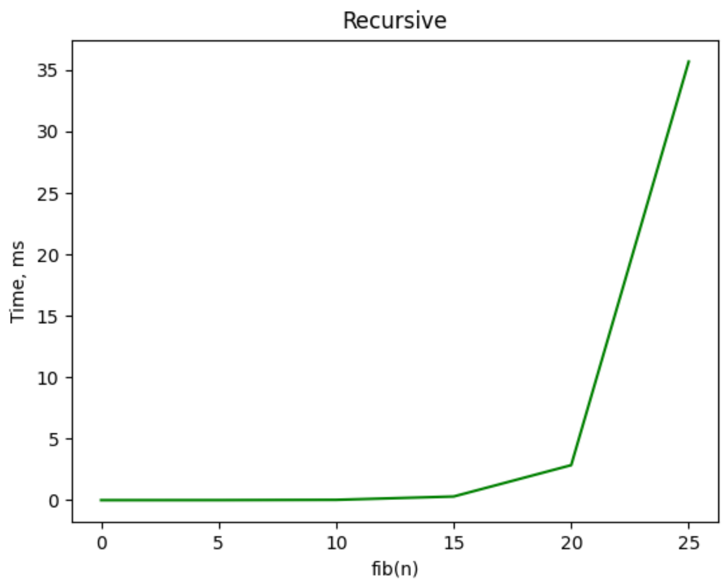
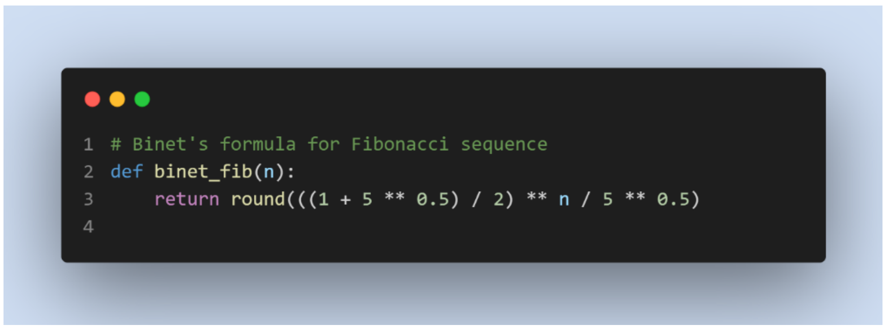


Fig 2: Recursive method elapsed time

The recursive method allows for easy calculation of small order Fibonacci sequence numbers. However, due to the necessity of computing each number every time it's met it has an exponentially increasing complexity. This issue can be mitigated using cashing, which implies storing the previous function calls results in an array/file and accessing them (if they exist) when the function is called.

Complexity: $O(2^n)$

Binet's formula



```python
# Binet's formula for Fibonacci sequence
def binet_fib(n):
    return round(((1 + 5 ** 0.5) / 2) ** n / 5 ** 0.5)
```

Fig 3: Binet's formula code

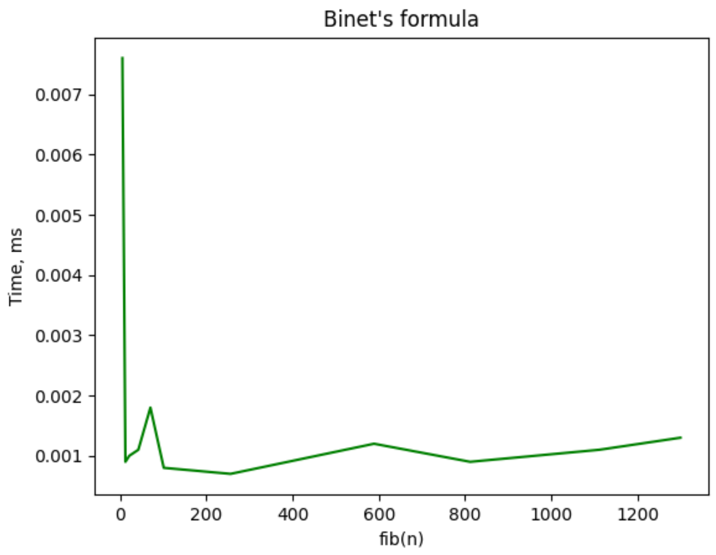
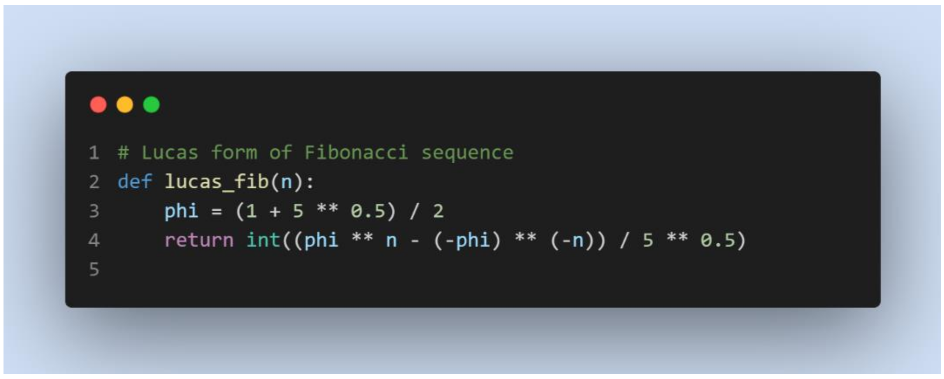


Fig 4: Binet's formula elapsed time

The Binet's formula method implies the usage of the Binet's formula which is: $F_n=\{[(\sqrt{5}+1)/2]^n\}/\sqrt{5}$, and rounding it to the nearest integer. This allows for almost constant time computation of the number. However, from n=72 the function will output a wrong result due to a rounding error.

Complexity: O(1)

Lucas' formula



```python
# Lucas form of Fibonacci sequence
def lucas_fib(n):
    phi = (1 + 5 ** 0.5) / 2
    return int((phi ** n - (-phi) ** (-n)) / 5 ** 0.5)
```

Fig 5: Lucas' formula code

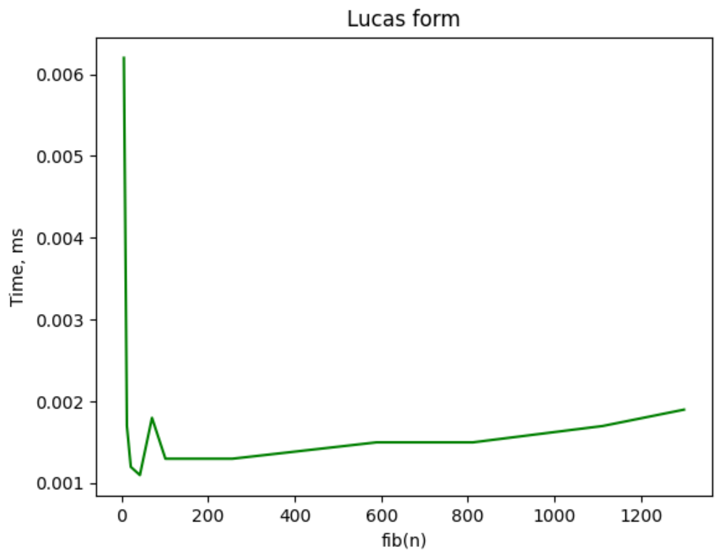
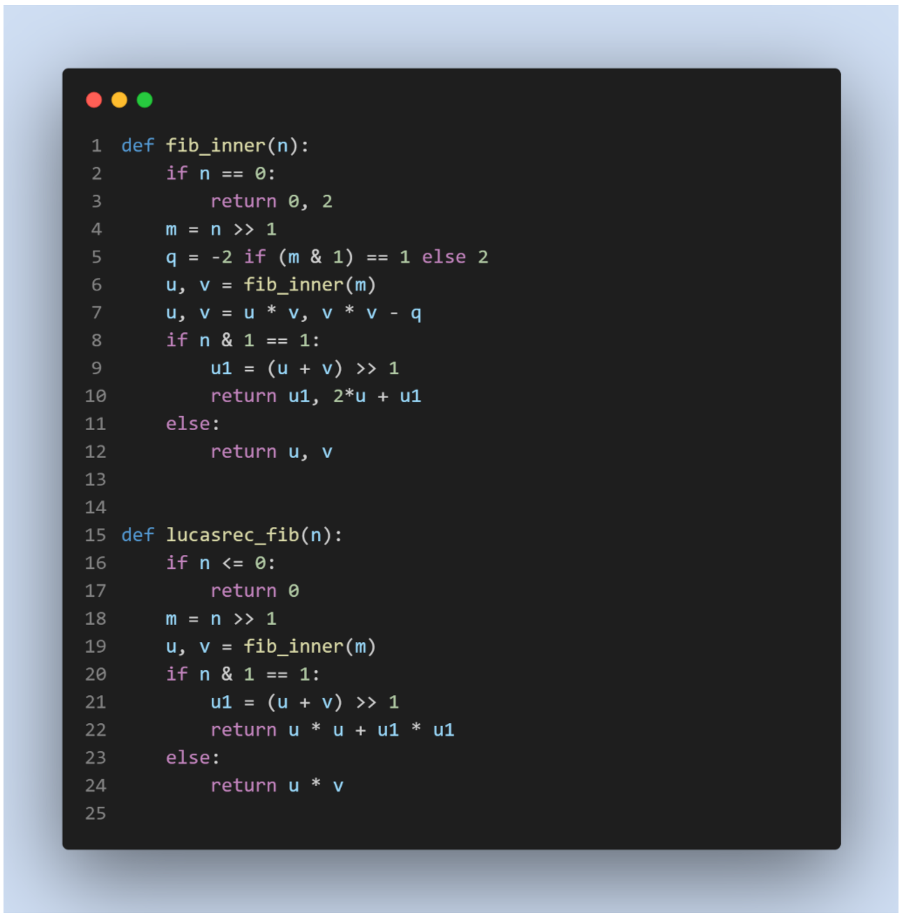


Fig 6: Lucas' formula elapsed time

The Lucas' formula method is a more complex form of Binet's formula, which is: F_n={[(1+√5)/2]^n-[(1-√5)/2]^n}/(√5).

Complexity: O(1)

Lucas recursive formula



```python
def fib_inner(n):
    if n == 0:
        return 0, 2
    m = n >> 1
    q = -2 if (m & 1) == 1 else 2
    u, v = fib_inner(m)
    u, v = u * v, v * v - q
    if n & 1 == 1:
        u1 = (u + v) >> 1
        return u1, 2*u + u1
    else:
        return u, v


def lucasrec_fib(n):
    if n <= 0:
        return 0
    m = n >> 1
    u, v = fib_inner(m)
    if n & 1 == 1:
        u1 = (u + v) >> 1
        return u * u + u1 * u1
    else:
        return u * v

```

Fig 7: Lucas' recursive formula

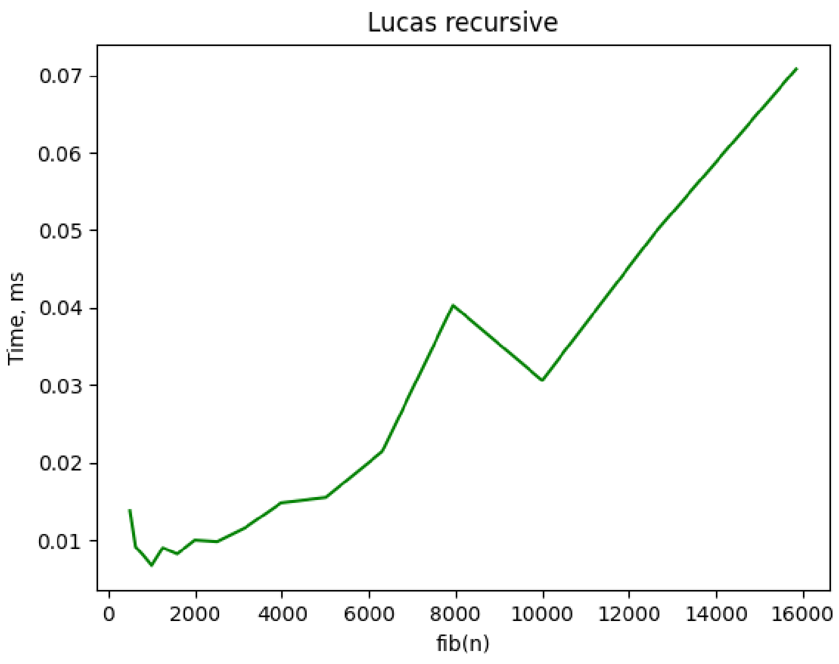
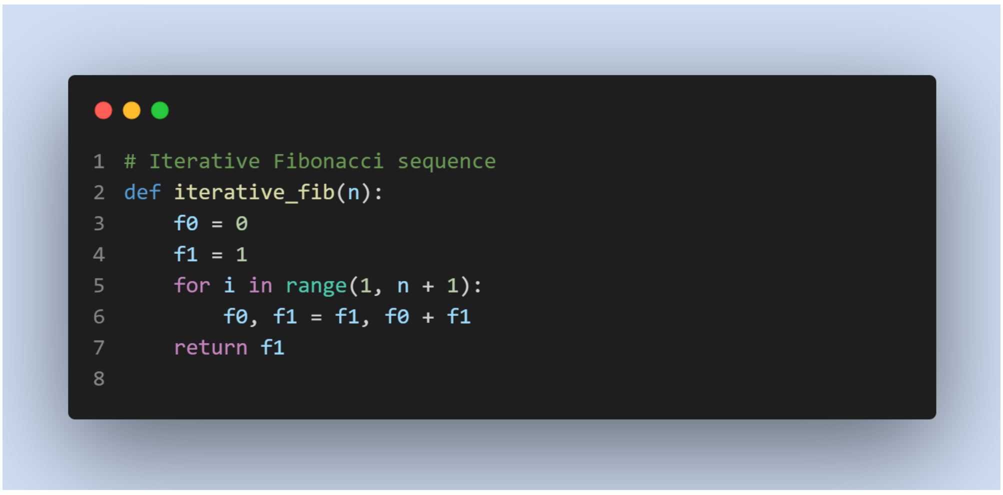


Fig 8: Lucas recursive formula elapsed time

Complexity: O(log n)

Iterative method



```python
# Iterative Fibonacci sequence
def iterative_fib(n):
    f0 = 0
    f1 = 1
    for i in range(1, n + 1):
        f0, f1 = f1, f0 + f1
    return f1

```

Fig 9: Iterative method code

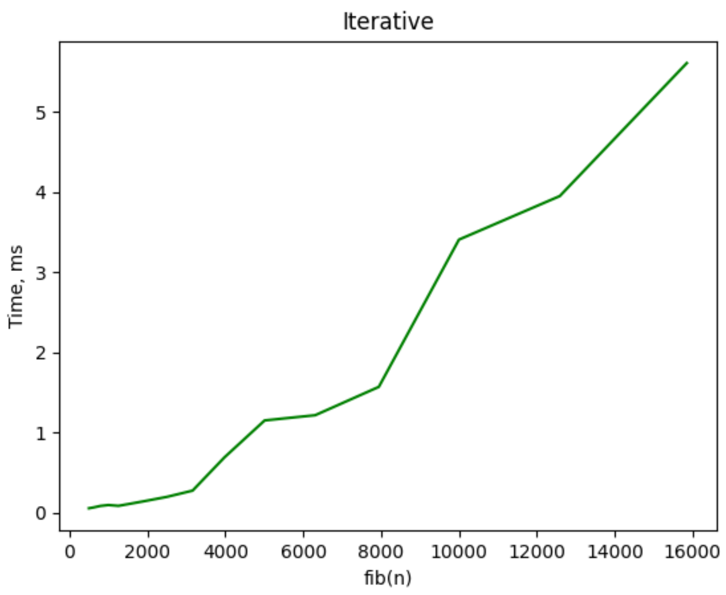
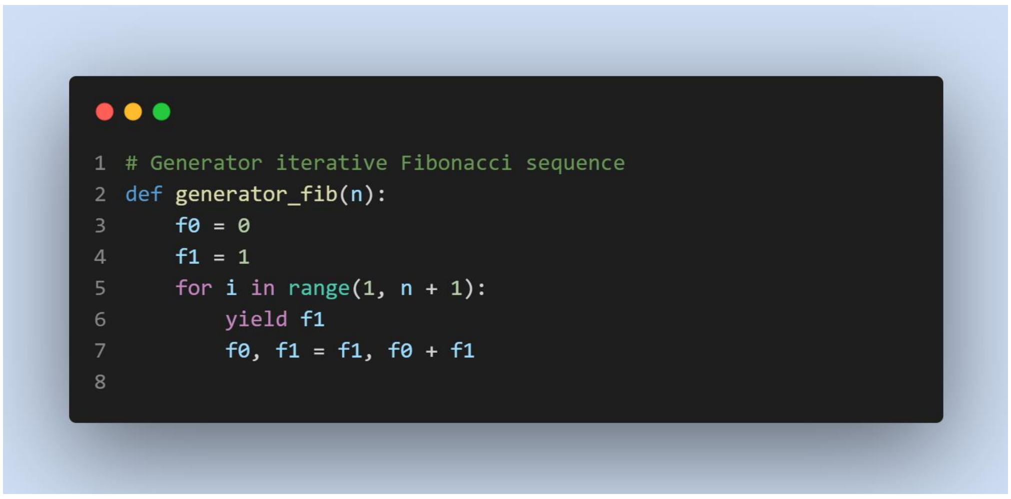


Fig 10: Iterative method elapsed time

The iterative method is one of the more straight forward methods for computing the Fibonacci sequence, since it computes each number starting from 0, which implies the fact that execution time is directly tied to how big the number is. Same as the recursive method, the execution time can be improved by caching the previous results.

Complexity: O(n)

Generator iterative method



```python
# Generator iterative Fibonacci sequence
def generator_fib(n):
    f0 = 0
    f1 = 1
    for i in range(1, n + 1):
        yield f1
        f0, f1 = f1, f0 + f1
```

Fig 11: Generator iterative method

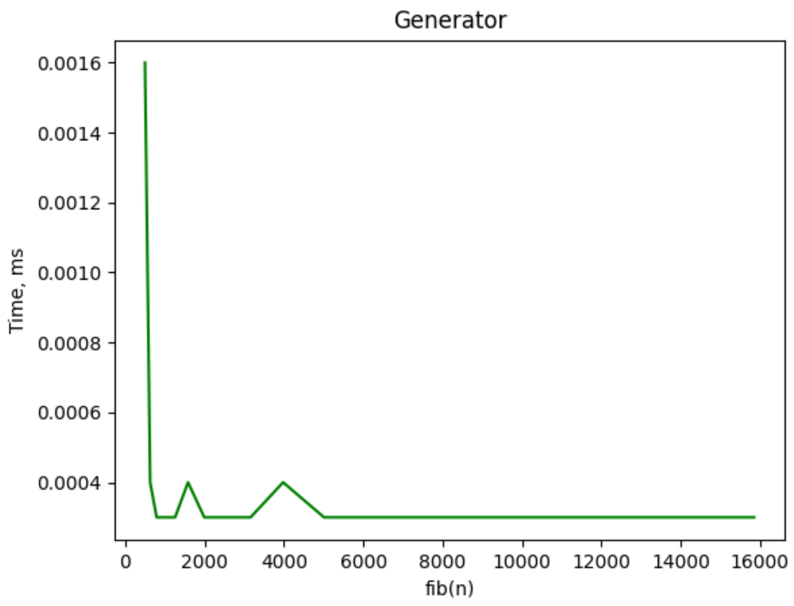
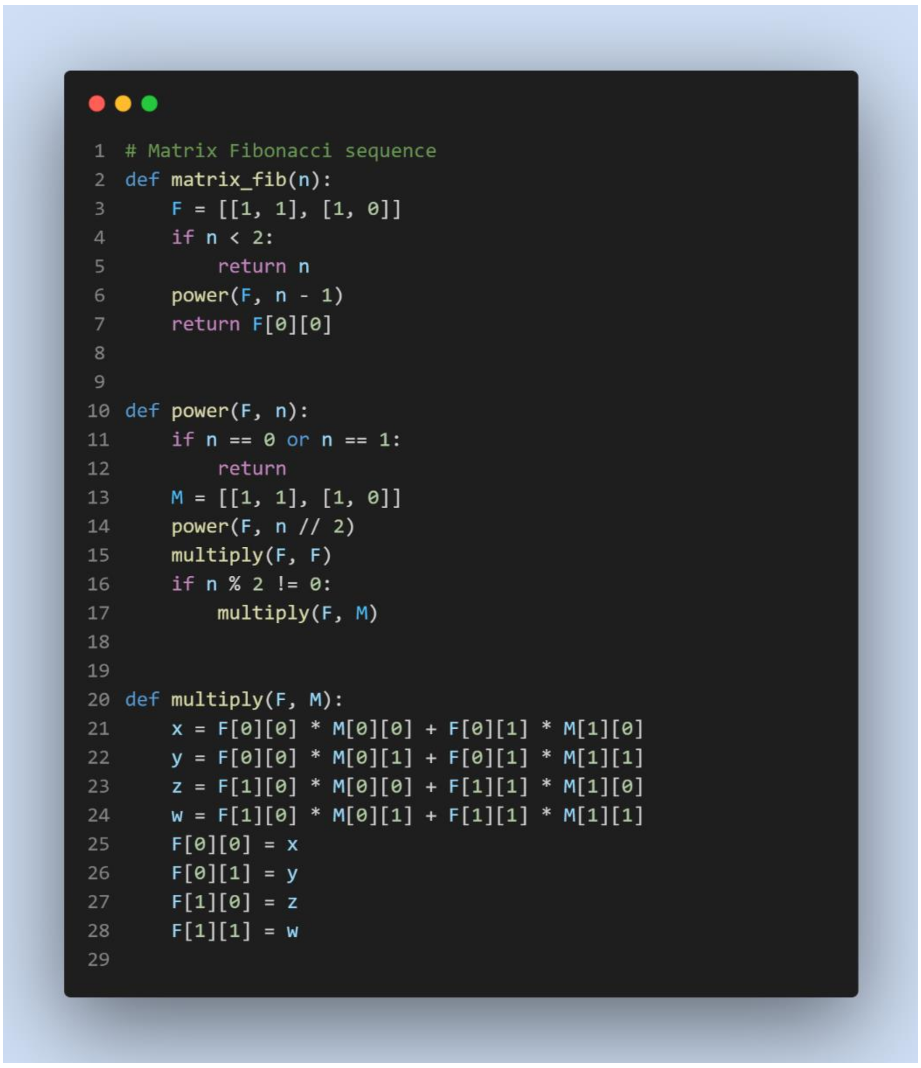


Fig 12: Generator iterative method elapsed time

I don’t know how the generator method works, since I found it on the internet and it outputs the wrong results.

Complexity: O(1)

Matrix multiplication method



```python
# Matrix Fibonacci sequence
def matrix_fib(n):
    F = [[1, 1], [1, 0]]
    if n < 2:
        return n
    power(F, n - 1)
    return F[0][0]


def power(F, n):
    if n == 0 or n == 1:
        return
    M = [[1, 1], [1, 0]]
    power(F, n // 2)
    multiply(F, F)
    if n % 2 != 0:
        multiply(F, M)


def multiply(F, M):
    x = F[0][0] * M[0][0] + F[0][1] * M[1][0]
    y = F[0][0] * M[0][1] + F[0][1] * M[1][1]
    z = F[1][0] * M[0][0] + F[1][1] * M[1][0]
    w = F[1][0] * M[0][1] + F[1][1] * M[1][1]
    F[0][0] = x
    F[0][1] = y
    F[1][0] = z
    F[1][1] = w

```

Fig 13: Matrix multiplication method

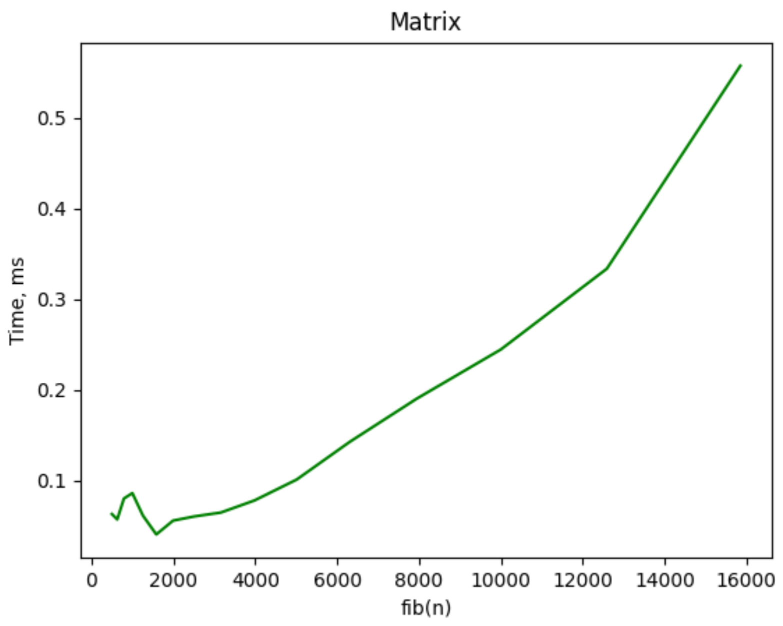
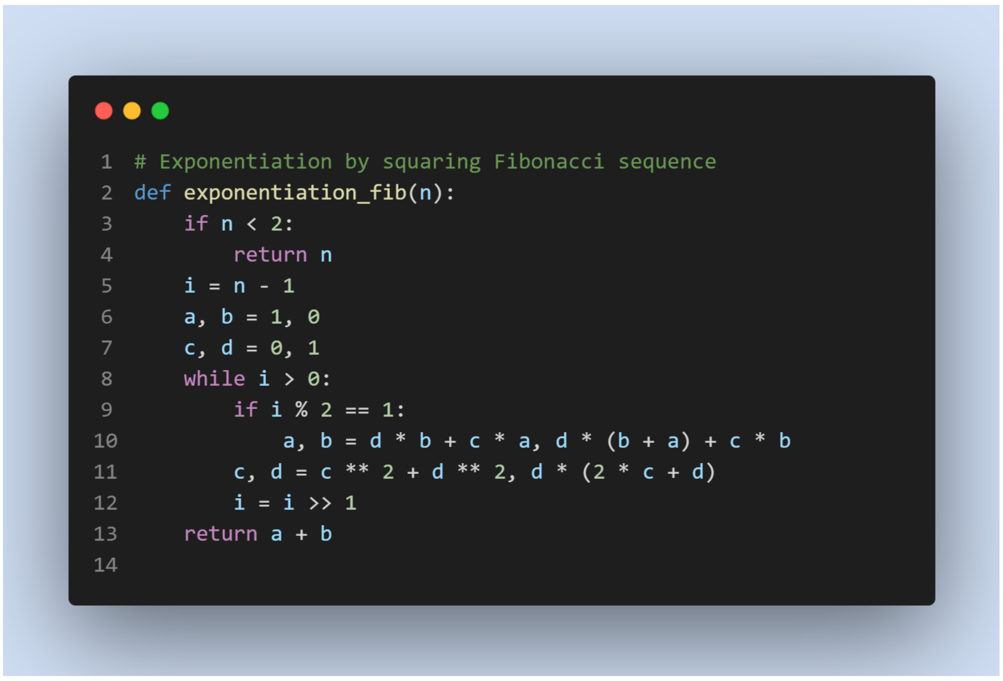


Fig 14: Matrix multiplication method elapsed time

The matrix method relies on the fact that if we n times multiply the matrix M = {{1,1},{1,0}} to itself (in other words calculate power(M, n)), then we get the (n+1)th Fibonacci number as the element at row and column (0, 0) in the resultant matrix. Even though it has the O(n) complexity as the iterative method, it is ~10 times faster.

Complexity: O(n)

Exponentiation method



```
# Exponentiation by squaring Fibonacci sequence
def exponentiation_fib(n):
    if n < 2:
        return n
    i = n - 1
    a, b = 1, 0
    c, d = 0, 1
    while i > 0:
        if i % 2 == 1:
            a, b = d * b + c * a, d * (b + a) + c * b
        c, d = c ** 2 + d ** 2, d * (2 * c + d)
        i = i >> 1
    return a + b

```

Fig 15: Exponentiation method code

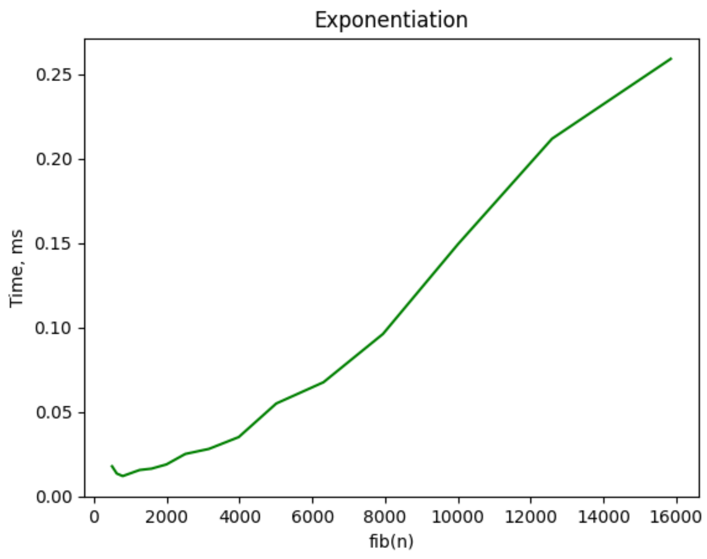


Fig 16: Exponentiation method elapsed time

The exponentiation by squaring is a general method for fast computation of large positive integer powers of a number. It has the same complexity as the matrix method but the execution time is two times faster.

Complexity: O(n)

Conclusions:

The Fibonacci sequence has several methods of computing it due to the golden ration being at its core which allows for multiple interpretations of it.

The recursive method is the easiest one to write and remember since it follows the main principle of the Fibonacci sequence, which is that the next number in the series is the sum of the previous two numbers. However, due to the redundant computations for each number is takes increasing time and computational resources as it need to store the previous computations in the recursion.

The iterative method is similar to the recursive method the main difference being that the computation starts from the beginning, meaning that it computes each number only once.

The formulas that use the golden ratio are the most optimal ones since it has a constant complexity, by using a general formula. However, due to the nature of real numbers being computed by the programing language it generates rounding errors, which can return wrong results for bigger numbers.

References:

1. https://en.wikibooks.org/wiki/Algorithm_Implementation/Mathematics/Fibonacci_Number_Program#Python
2. https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/
3. https://en.wikipedia.org/wiki/Exponentiation_by_squaring

4. https://en.wikipedia.org/wiki/Fibonacci_number