Ministerul Educaţiei şi Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 1:

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

Elaborated:
st. gr. FAF-211

Spătaru Dionisie

Verified:

asist. univ.

Fiștic Cristofor

Chişinău - 2023

ALGORITHM ANALYSIS

Objective

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

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 Notes:

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.

Introduction:

The Fibonacci sequence is the series of numbers where each number is the sum of the two preceding numbers. For example: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ... Mathematically we can describe this as: xn= xn-1 + xn-2.

Many sources claim this sequence was first discovered or "invented" by Leonardo Fibonacci. The Italian mathematician, who was born around A.D. 1170, was initially known as Leonardo of Pisa. In the 19th century, historians came up with the nickname Fibonacci (roughly meaning "son of the Bonacci clan") to distinguish the mathematician from another famous Leonardo of Pisa.

There are others who say he did not. Keith Devlin, the author of Finding Fibonacci: The Quest to Rediscover the Forgotten Mathematical Genius Who Changed the World, says there are ancient Sanskrit texts that use the Hindu-Arabic numeral system - predating Leonardo of Pisa by centuries.

But, in 1202 Leonardo of Pisa published a mathematical text, Liber Abaci. It was a “cookbook” written for tradespeople on how to do calculations. The text laid out the Hindu-Arabic arithmetic useful for tracking profits, losses, remaining loan balances, etc, introducing the Fibonacci sequence to the Western world.

Traditionally, the sequence was determined just by adding two predecessors to obtain a new number, however, with the evolution of computer science and algorithmics, several distinct methods for determination have been uncovered. The methods can be grouped in 4 categories, Recursive Methods, Dynamic Programming Methods, Matrix Power Methods, and Benet Formula Methods. All those can be implemented naively or with a certain degree of optimization, that boosts their performance during analysis.

As mentioned previously, the performance of an algorithm can be analyzed mathematically (derived through mathematical reasoning) or empirically (based on experimental observations).

Within this laboratory, we will be analyzing the 4 algorithms empirically.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n))

Input Format:

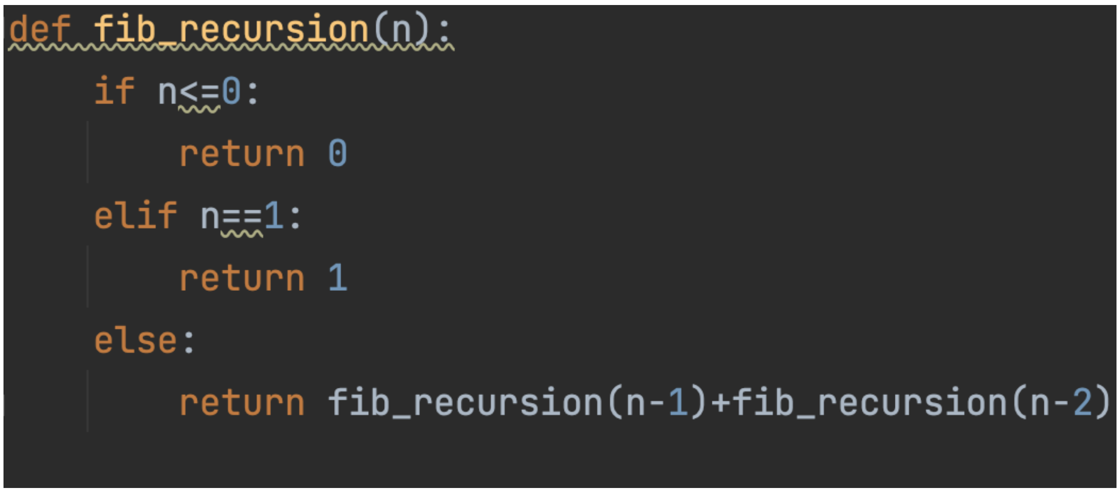
As input, random number given by the user.

IMPLEMENTATION

Recursive Method

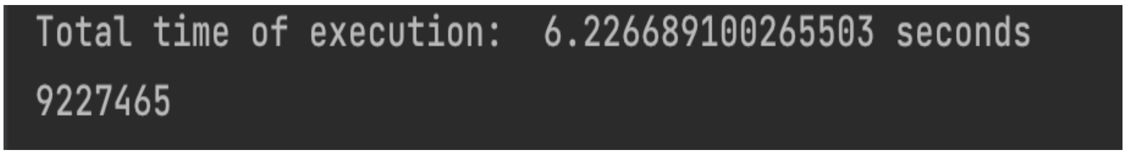
A recursive method for generating the Fibonacci sequence involves defining a function that calls itself with the two previous values of the sequence as input. The function terminates when the desired number of terms in the sequence has been generated. This method has exponential time complexity and is not efficient for large values of n.

Implementation:

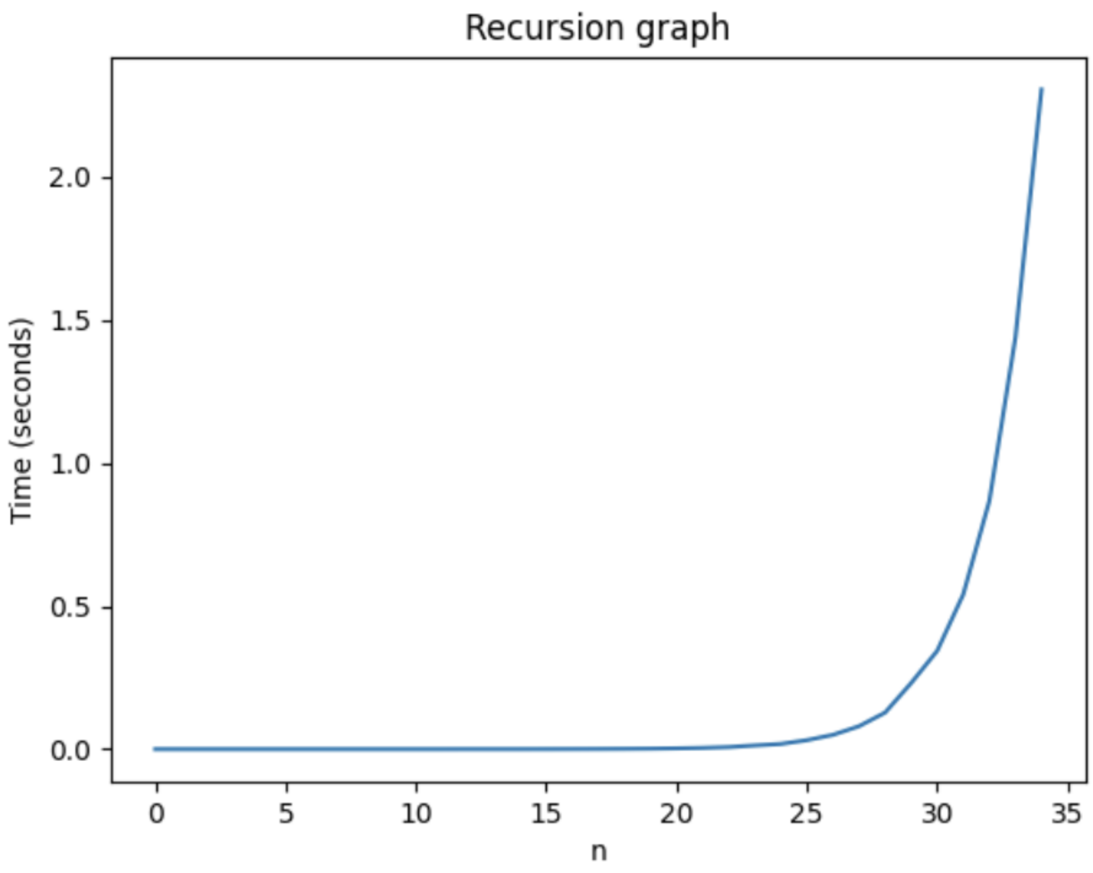


```
def fib_recursion(n):
    if n<=0:
        return 0
    elif n==1:
        return 1
    else:
        return fib_recursion(n-1)+fib_recursion(n-2)
```

Result:



```
Total time of execution:  6.226689100265503 seconds
9227465
```



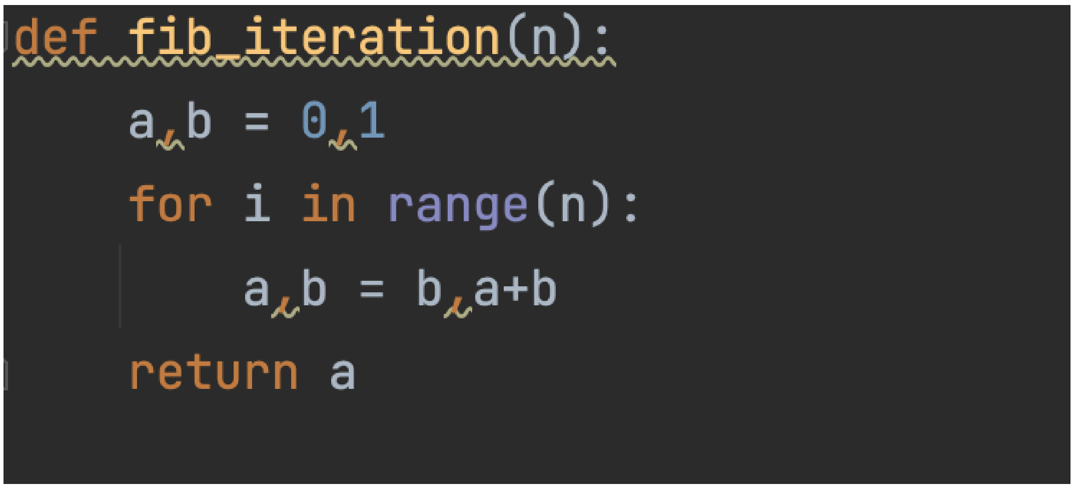
As we can see this method is very slow, to determine the 35th Fibonacci number we need 6.22 seconds, and last iteration stol more than 2 seconds.

Iteration Method

The iterative method for generating the Fibonacci sequence involves using a loop to iteratively calculate each subsequent number in the sequence based on the two preceding ones. Unlike the recursive method, the iterative method avoids redundant calculations and has linear time complexity.

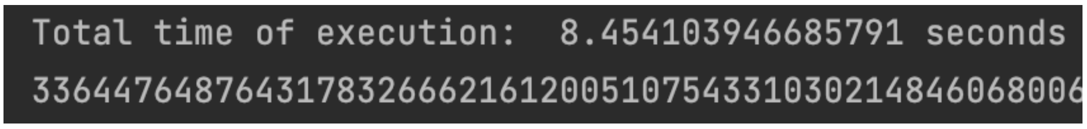
In this method, you start with an initial value of 0 and 1 and use a loop to generate the next numbers in the sequence by adding the current two numbers together. These sums are stored in variables and used to generate the next numbers in the sequence until the desired number of terms has been generated.

Implementation:

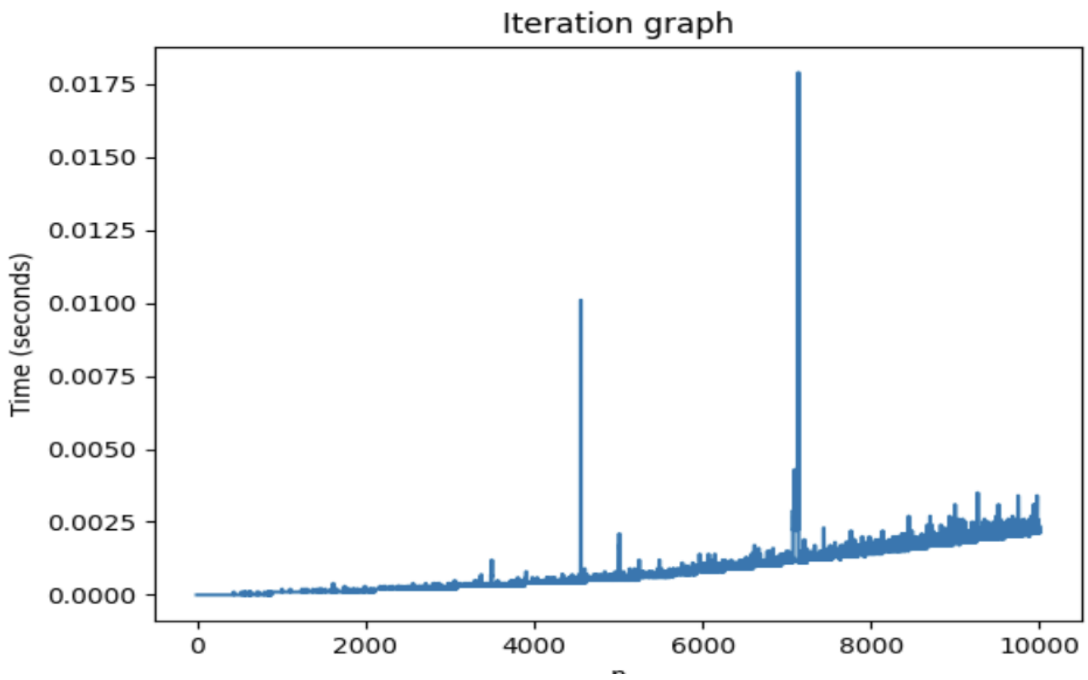


```python
def fib_iteration(n):
    a,b = 0,1
    for i in range(n):
        a,b = b,a+b
    return a
```

Result:



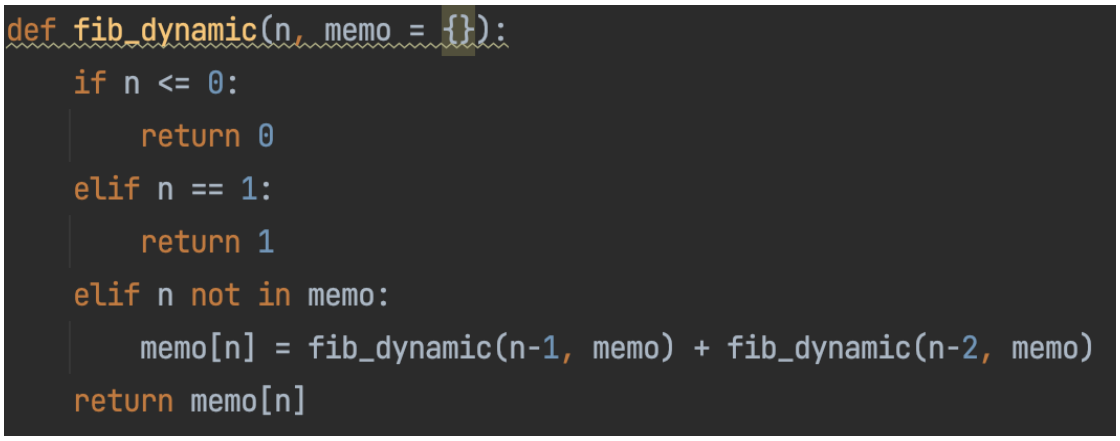
```
Total time of execution:  8.454103946685791 seconds
3364476487643178326662161200510754331030214846068006
```



Dynamic Method

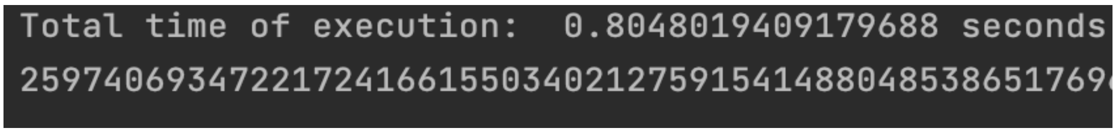
The dynamic programming method for generating the Fibonacci sequence involves using an array or similar data structure to store previously calculated values and use them to avoid redundant calculations. It has the same linear time complexity as the iterative method but often requires less memory usage. In this method, an array is used to store the intermediate results as the sequence is generated. Each time a new number in the sequence is calculated, it is stored in the array and used as needed for the subsequent calculations. This allows the method to use the stored values to avoid recalculating the same values over and over again, which greatly improves its efficiency.

Implementation:

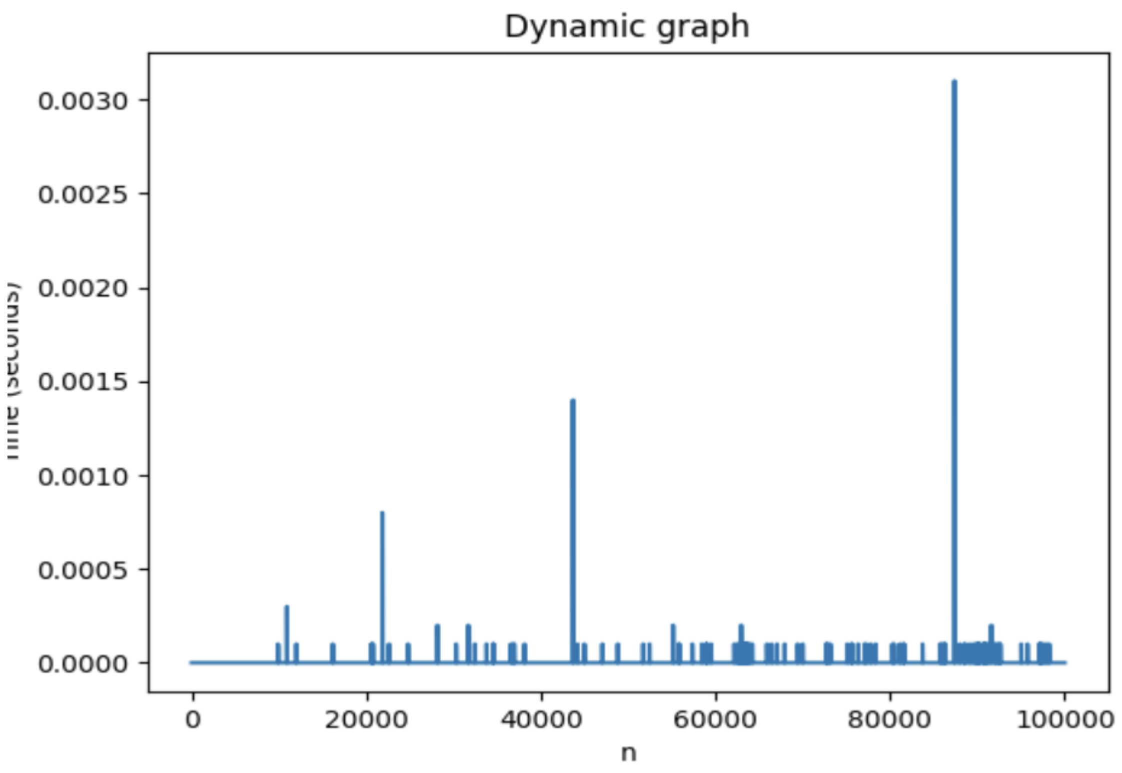


```python
def fib_dynamic(n, memo = {}):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    elif n not in memo:
        memo[n] = fib_dynamic(n-1, memo) + fib_dynamic(n-2, memo)
    return memo[n]
```

Result:



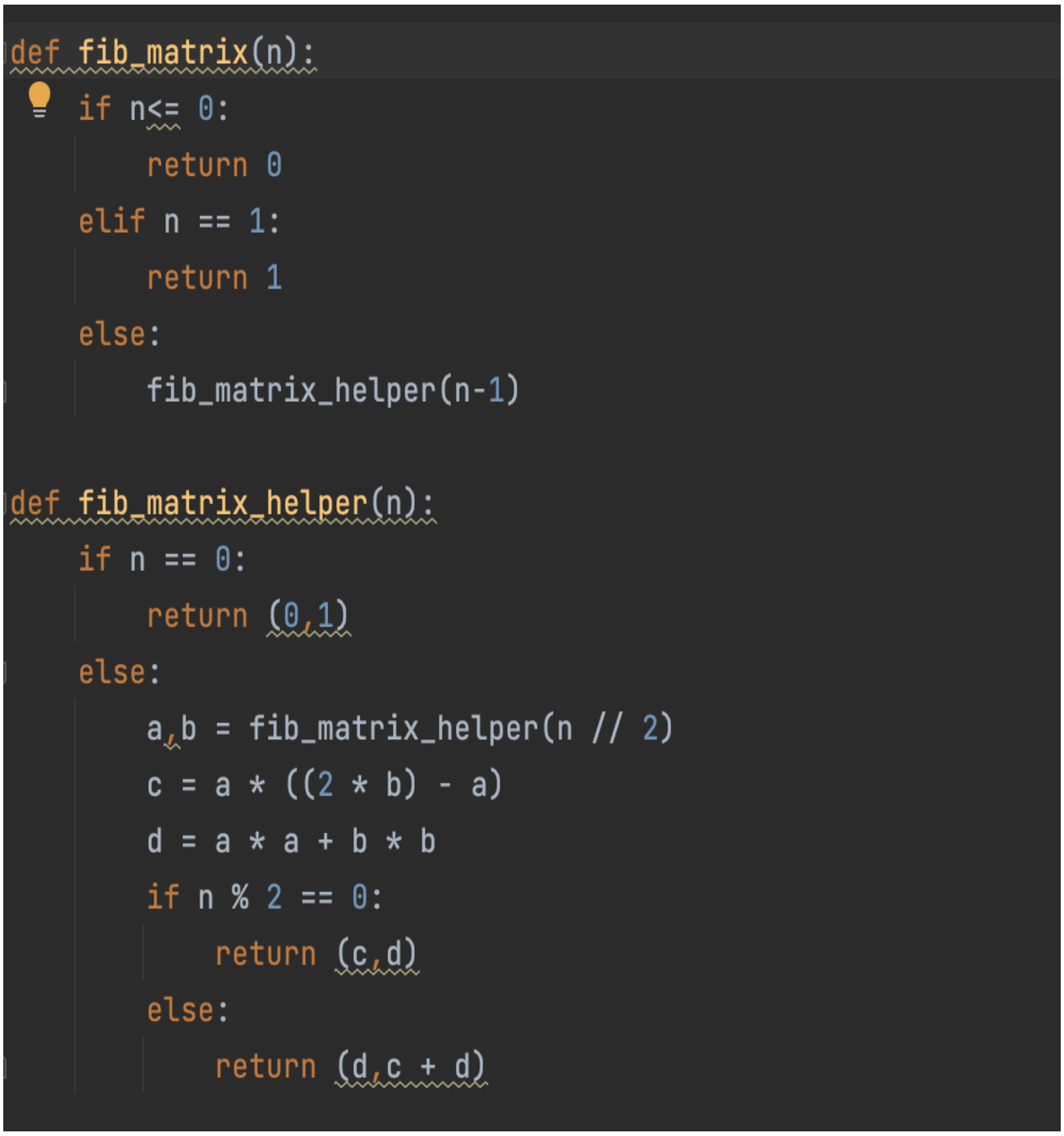
```
Total time of execution:  0.8048019409179688 seconds
2597406934722172416615503402127591541488048538651769
```



Matrix Method

The matrix method for generating the Fibonacci sequence is a mathematical approach that involves using matrix multiplication to efficiently generate the numbers in the sequence. This method has a time complexity of O(log n), which is significantly faster than both the iterative and dynamic programming methods.Overall, the matrix method is a highly efficient and elegant way to generate the Fibonacci sequence, but it requires a good understanding of matrix mathematics to understand and implement.

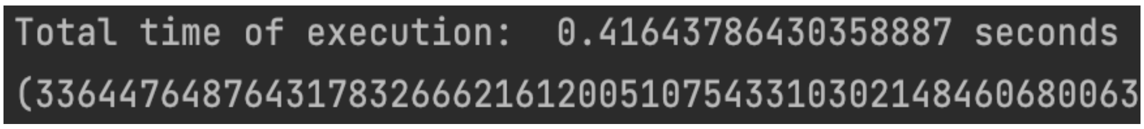
Implementation:



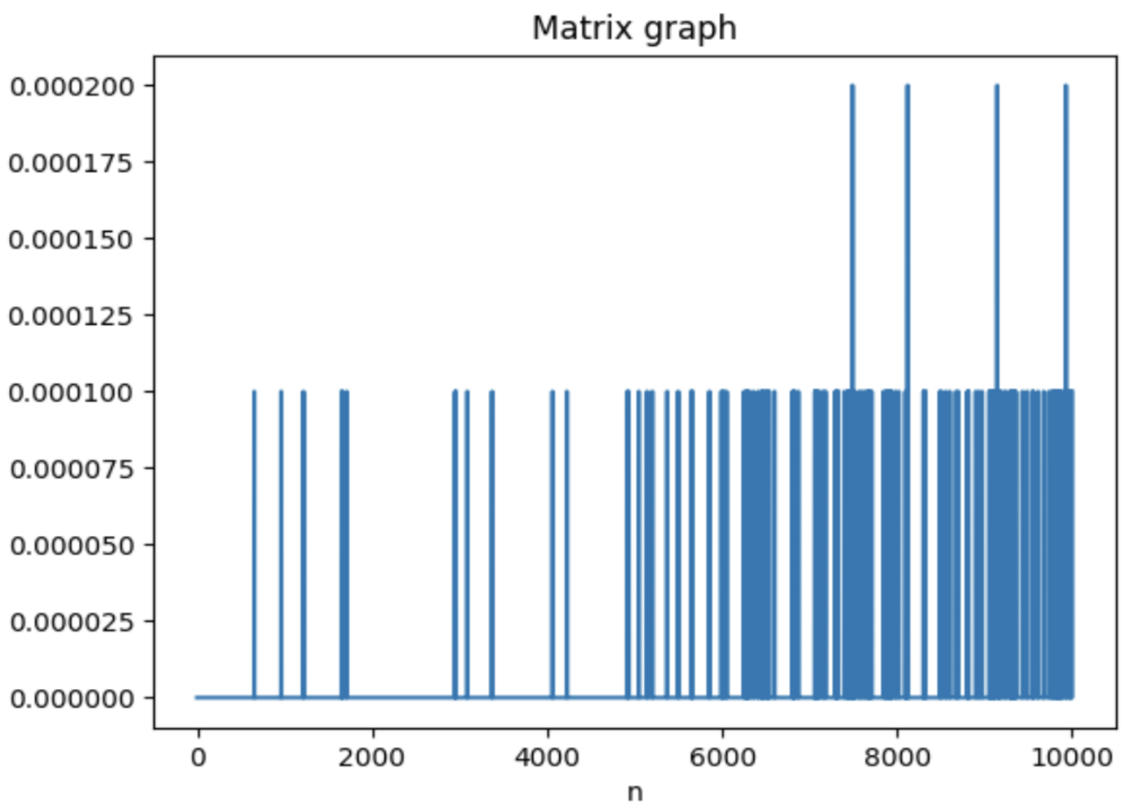
```
def fib_matrix(n):
    if n<= 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_matrix_helper(n-1)

def fib_matrix_helper(n):
    if n == 0:
        return (0,1)
    else:
        a,b = fib_matrix_helper(n // 2)
        c = a * ((2 * b) - a)
        d = a * a + b * b
        if n % 2 == 0:
            return (c,d)
        else:
            return (d,c + d)
```

Result:

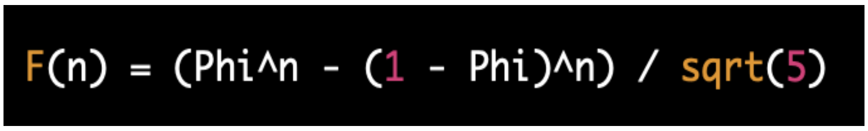


```
Total time of execution:  0.41643786430358887 seconds
(336447648764317832666216120051075433103021484606800063
```



Binet Method

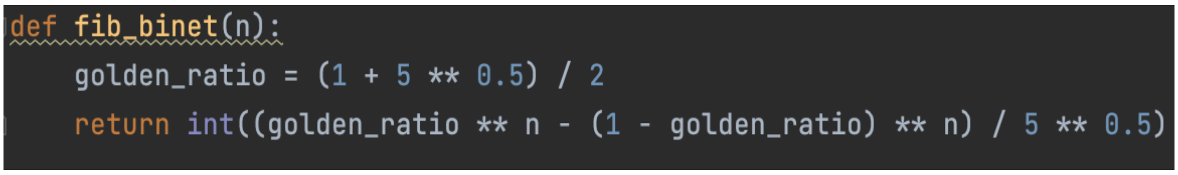
The Binet method is an efficient way to generate any specific number in the Fibonacci sequence, but it does not provide an easy way to generate the entire sequence. Despite this limitation, the Binet method is still an important tool for understanding the properties of the Fibonacci sequence and for solving certain mathematical problems that involve Fibonacci numbers.It provides a closed-form solution that doesn't require the iterative calculation of each number in the sequence.



```
F(n) = (Phi^n - (1 - Phi)^n) / sqrt(5)
```

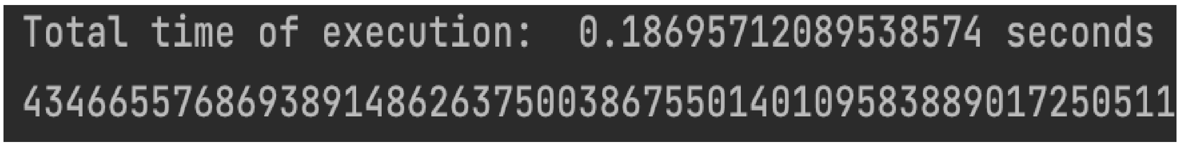
where Phi is the golden ratio, approximately equal to 1.6180339887

Implementation:

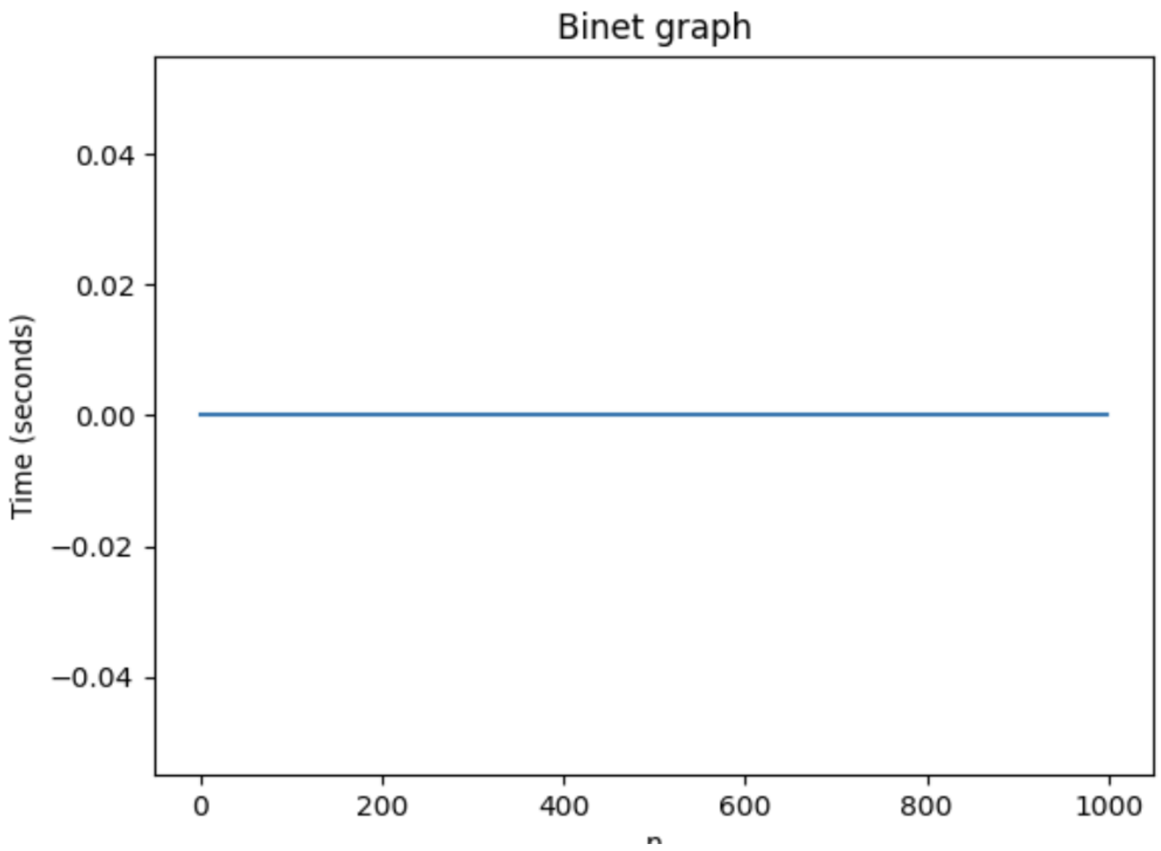


```
def fib_binet(n):
    golden_ratio = (1 + 5 ** 0.5) / 2
    return int((golden_ratio ** n - (1 - golden_ratio) ** n) / 5 ** 0.5)
```

Result:



```
Total time of execution:  0.18695712089538574 seconds
434665576869389148626375003867550140109583889017250511
```

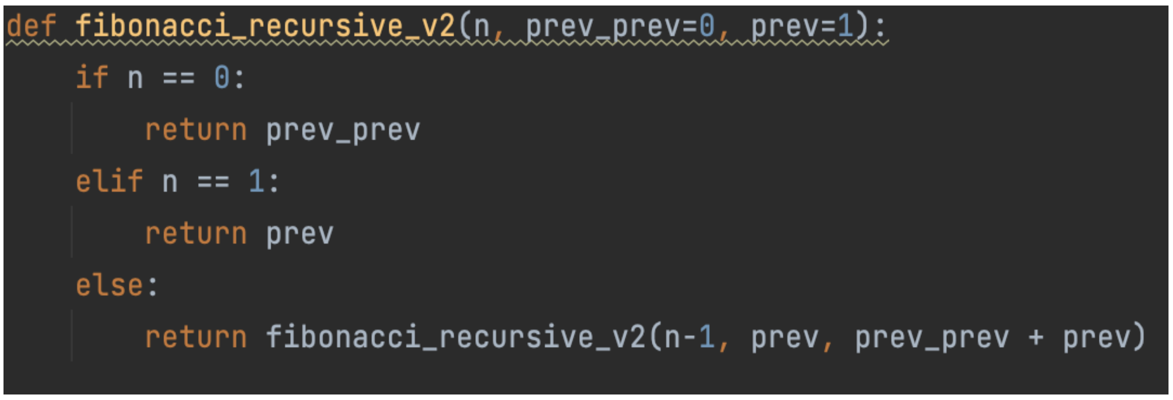


Tail recursive Method

The tail recursive method for generating the Fibonacci sequence is a variation of the recursive method that eliminates the need for additional memory to store intermediate results. It uses a technique called tail recursion, which allows the function to be called recursively as the very last step of its execution, so that the intermediate results are not stored in the call stack and do not take up additional memory.

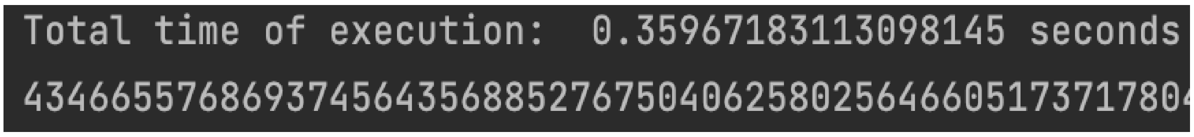
In this method, the function call to generate the nth number in the sequence is performed as the last step of the function, so that the intermediate results can be discarded immediately, rather than being stored in the call stack for later use. This eliminates the need for additional memory to store intermediate results and makes the method more efficient and scalable for larger values of n.

Implementation:

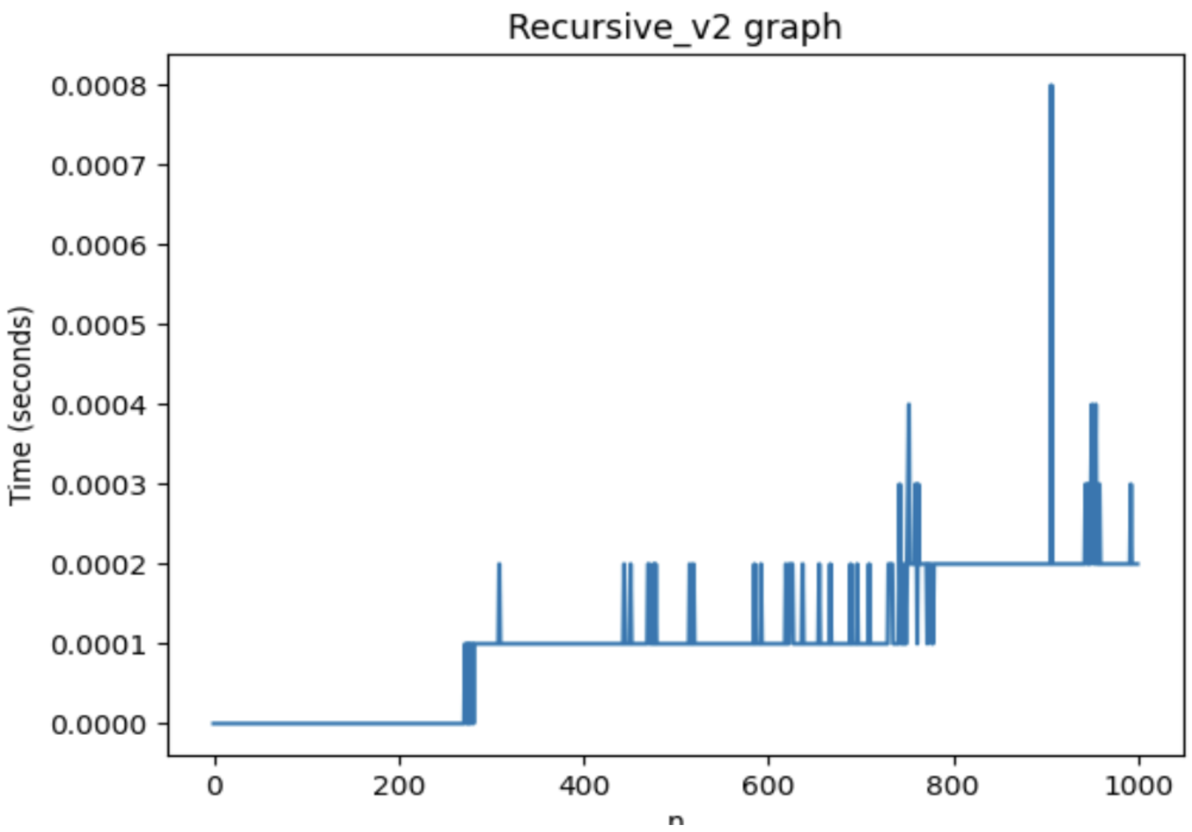


```python
def fibonacci_recursive_v2(n, prev_prev=0, prev=1):
    if n == 0:
        return prev_prev
    elif n == 1:
        return prev
    else:
        return fibonacci_recursive_v2(n-1, prev, prev_prev + prev)
```

Result:



```
Total time of execution:  0.35967183113098145 seconds
43466557686937456435688527675040625802564660517371780
```



CONCLUSION

In conclusion, there are several methods for generating the Fibonacci sequence, each with its own strengths and weaknesses. The recursive method is easy to understand but has exponential time complexity and is inefficient for large values of n. The iterative and dynamic programming methods have linear time complexity and are more efficient, but they still require significant memory usage. The matrix method has a time complexity of O(log n) and is highly efficient, but it requires a good understanding of matrix mathematics. The Binet method provides a closed-form solution but doesn't generate the entire sequence. The tail recursive method eliminates the need for additional memory and is more efficient, but it requires support for tail call optimization.

Ultimately, the choice of which method to use depends on the specific requirements and constraints of the problem being solved. The iterative and dynamic programming methods are often a good choice for many applications due to their simplicity and efficiency, but the other methods may be more appropriate in certain situations.