Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică şi Microelectronică

RAPORT

Lucrarea de laborator nr.1
la Analiza și Proiectarea Algoritmilor

A efectuat:
st. gr. FAF-212 Nume Prenume

A verificat:
asist. univ. Andreea Manole

Chişinău - 2023

ALGORITHM ANALYSIS

Objective:

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

Tasks:

- Implement at least 3 algorithms for determining Fibonacci n-th term;
- Decide properties of input format that will be used for algorithm analysis;
- Decide the comparison metric for the algorithms;
- Analyze empirically the algorithms;
- Present the results of the obtained data;
- 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:

- The purpose of the analysis is established.
- Choose the efficiency metric to be used (number of executions of an operation (s) or time
- execution of all or part of the algorithm.
- The properties of the input data in relation to which the analysis is performed are established
- (data size or specific properties).
- The algorithm is implemented in a programming language.
- Generating multiple sets of input data.
- Run the program for each input data set.
- 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 has 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 naïve 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, each algorithm will receive two series of numbers that will contain the order of the Fibonacci terms being looked up.

The first series will have a more limited scope, (5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 37, 40, 42, 45), to accommodate the recursive method, while the second series will have a bigger scope to be able to compare the other algorithms between themselves (501, 631, 794, 1000,1259, 1585, 1995, 2512, 3162, 3981, 5012, 6310, 7943, 10000, 12589, 15849).

Implementation

Recursive method

Recursion is the concept of something being defined in terms of itself. It sounds like it's circular - but it's not necessarily so. A circular definition, like defining a rose as a rose, is termed infinite recursion. But some recursive definitions aren't circular:

They have a base case, where the recursive definition no longer applies, and the definition for any other case eventually reaches the base case.

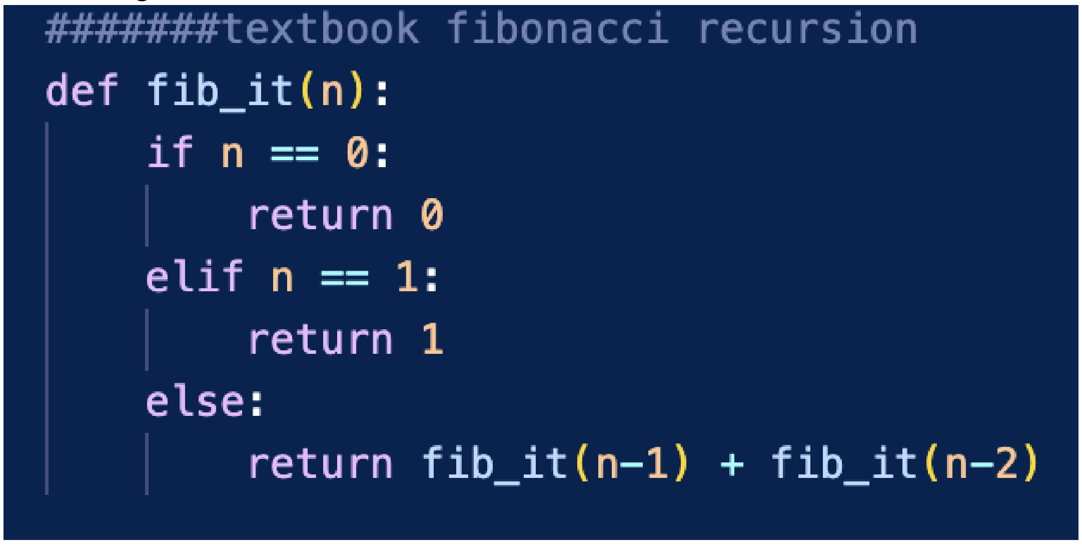
In mathematics, things are often defined recursively. For example, the Fibonacci numbers are often defined recursively. The Fibonacci numbers are defined as the sequence beginning with two 1's, and where each succeeding number in the sequence is the sum of the two preceding numbers.

1 1 2 3 5 8 13 21 34 55 …

A formal mathematical definition would define this using mathematical symbols.

$$F(n) = \{ \begin{array}{ll} 1 & \text{if } n = 0 \text{ or } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{array}$$

A simple method that is a direct recursive implementation mathematical recurrence relation is given below.



```
#######textbook fibonacci recursion
def fib_it(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_it(n-1) + fib_it(n-2)
```

Figure 1 Recursive method

Time Complexity: Exponential, as every function calls two other functions.

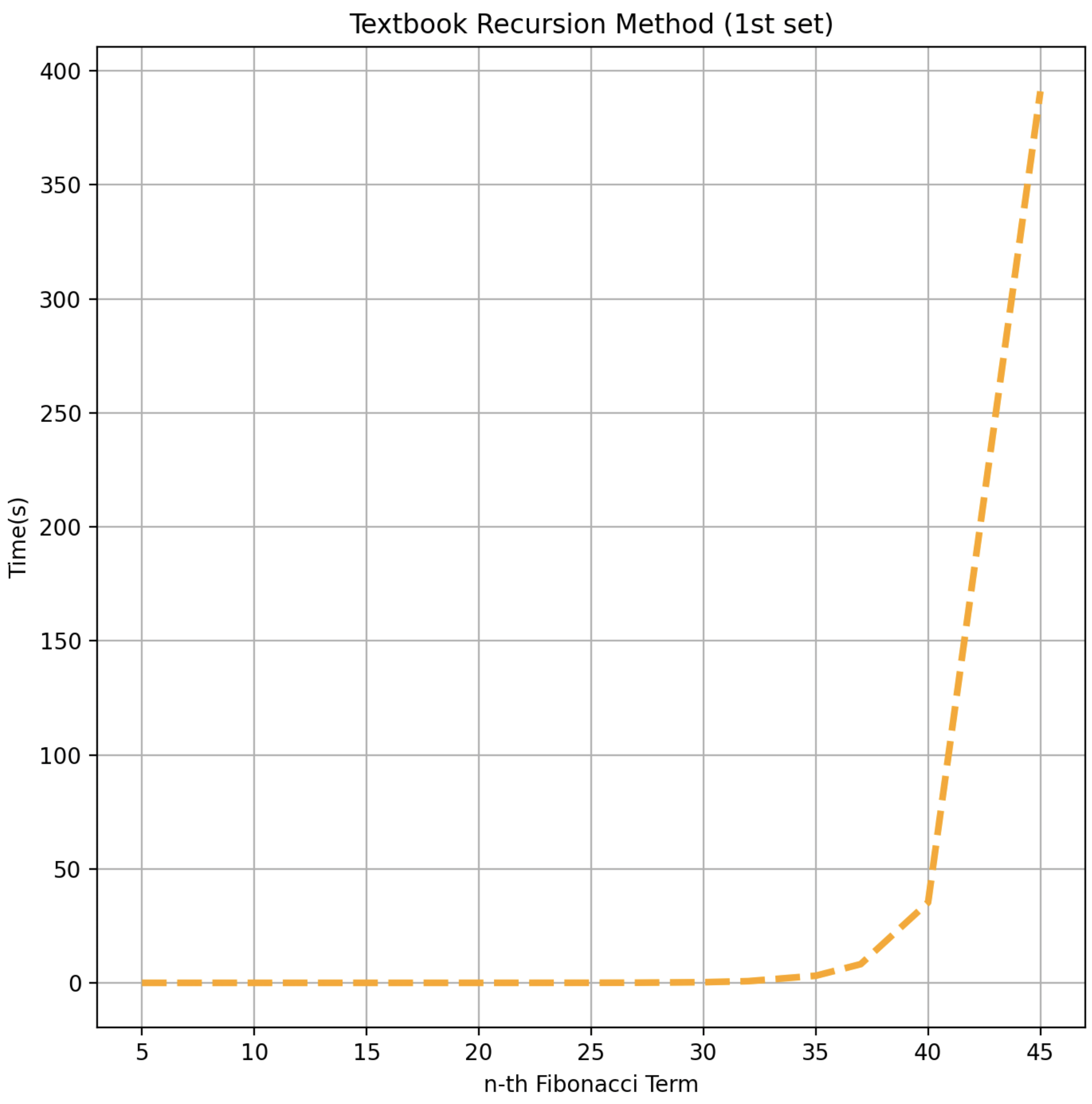
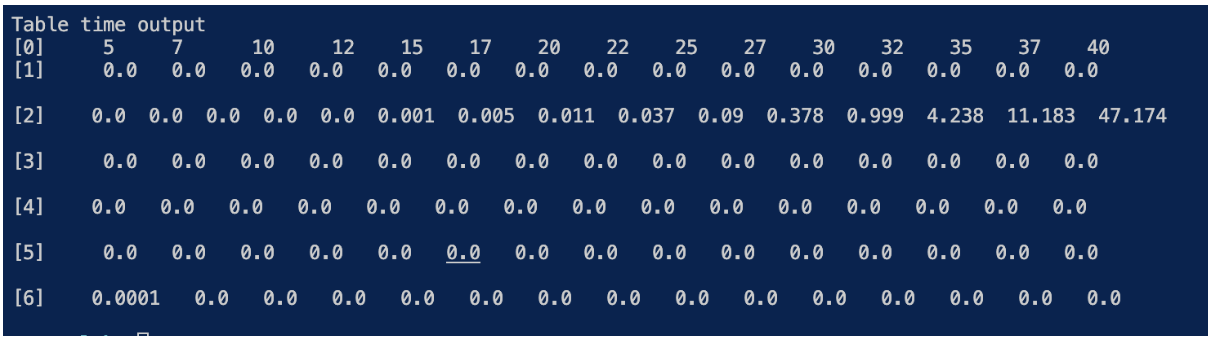


Figure 2 recursive method graph

As we can see above, the graph proves its time complexity, and what we must note is that even for *n=45* it took up to 400 seconds for the program to run, this is wildly inefficient. We must *definitely* not use this algorithm using case 2 input array.

Results:



```
Table time output
[0]     5     7      10     12     15     17     20     22     25     27     30     32     35     37     40
[1]     0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

[2]    0.0  0.0  0.0  0.0  0.0  0.001  0.005  0.011  0.037  0.09  0.378  0.999  4.238  11.183  47.174

[3]     0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

[4]    0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

[5]     0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

[6]    0.0001   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
```

Figure 3 Table time output (1st set)

Row[3] denotes the time complexity results for the recursion method for up to *n=40,* as we can see, this is simply inefficient.

Dynamic Programming methods:

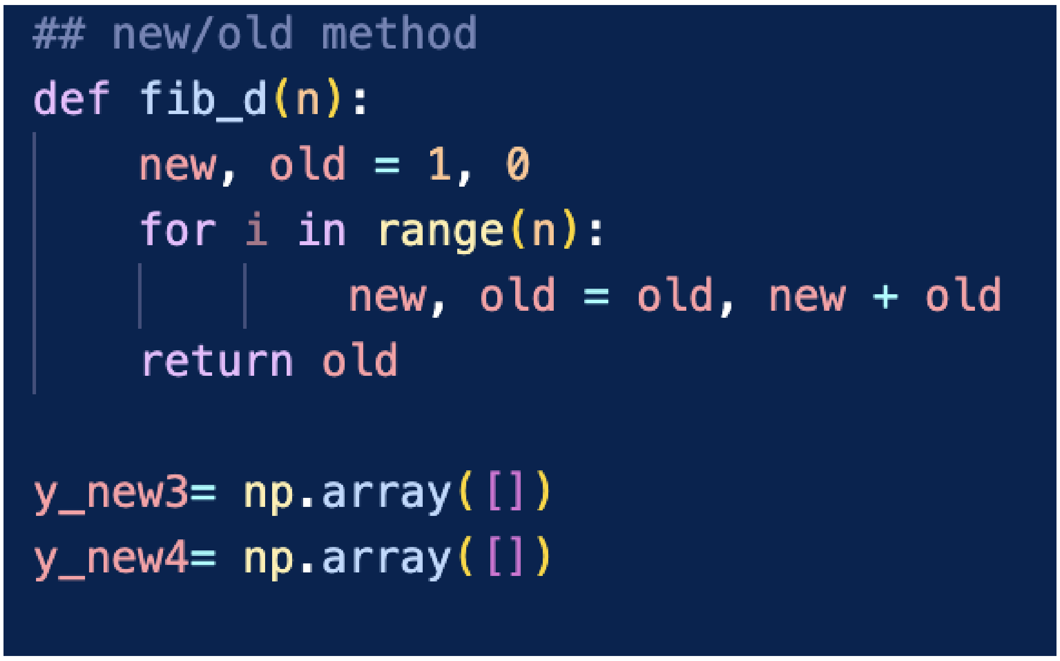
New/Old approach:

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming.

The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.

This simple optimization reduces time complexities from exponential to polynomial.

For example, if we write simple recursive solutions for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.



```
## new/old method
def fib_d(n):
    new, old = 1, 0
    for i in range(n):
            new, old = old, new + old
    return old

y_new3= np.array([])
y_new4= np.array([])
```

Figure 4 New/Old method

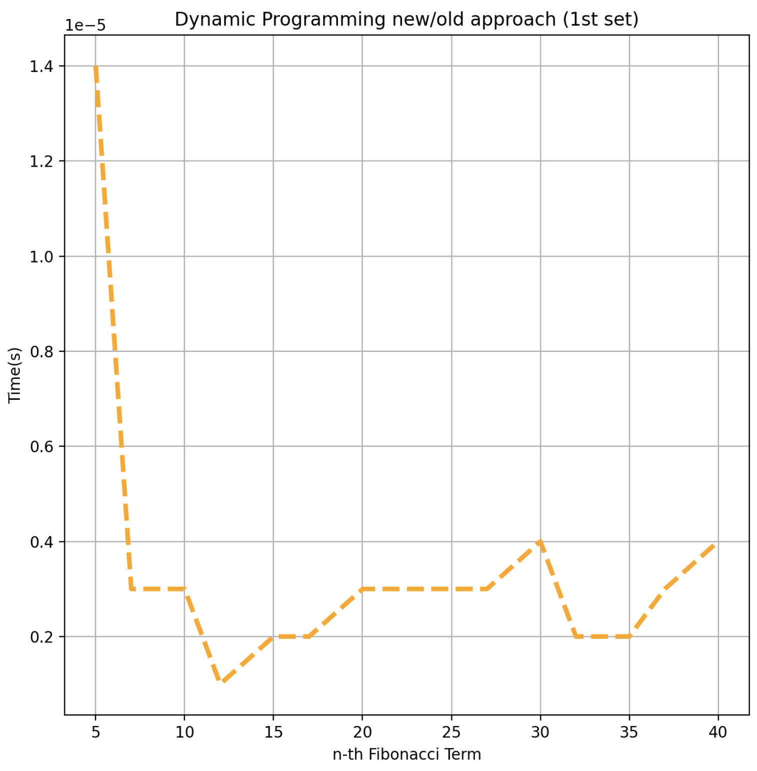


Figure 5

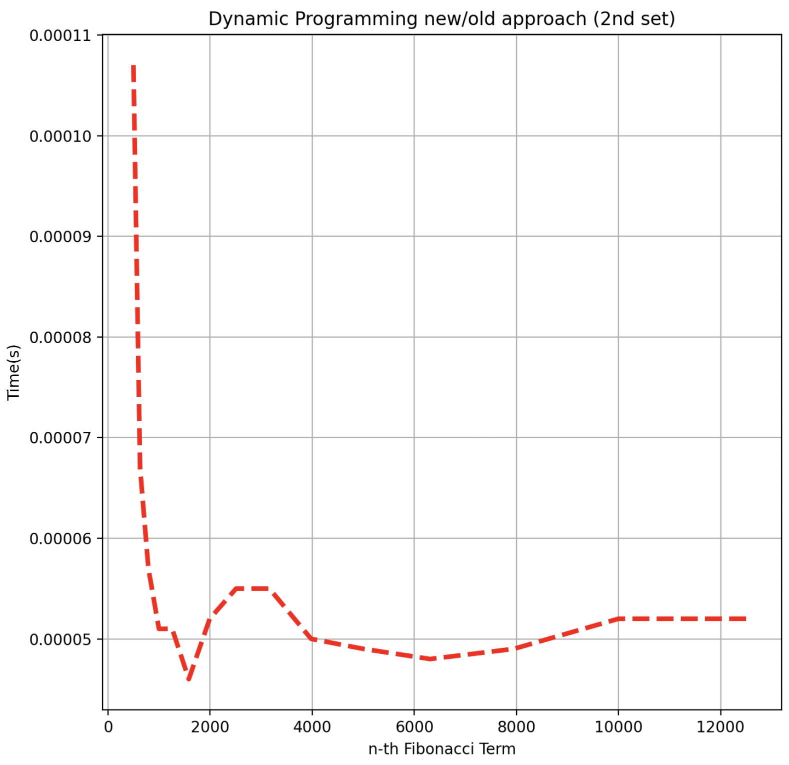
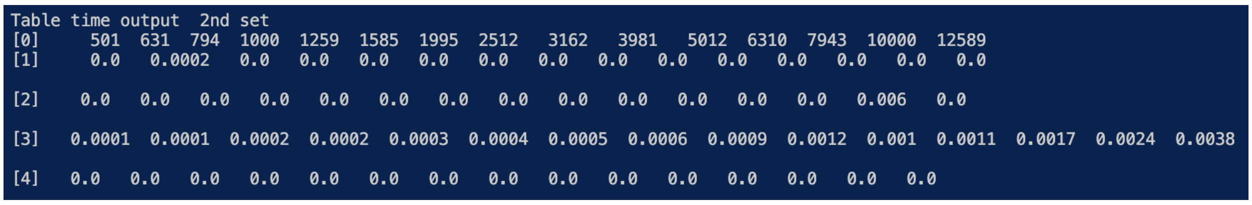


Figure 6

Time Complexity: Linear (more efficient than textbook recursive)

As we can see below the algorithm is efficient enough this time to test both test cases tables.



```
Table time output  2nd set
[0]     501  631  794  1000  1259  1585  1995  2512   3162   3981   5012  6310  7943  10000  12589
[1]     0.0   0.0002   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

[2]    0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.006   0.0

[3]   0.0001  0.0001  0.0002  0.0002  0.0003  0.0004  0.0005  0.0006  0.0009  0.0012  0.001  0.0011  0.0017  0.0024  0.0038

[4]   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
```

Figure 7 Table time output (2nd set)

Row[2] resembles the outstanding results of this algorithm for the second input array.

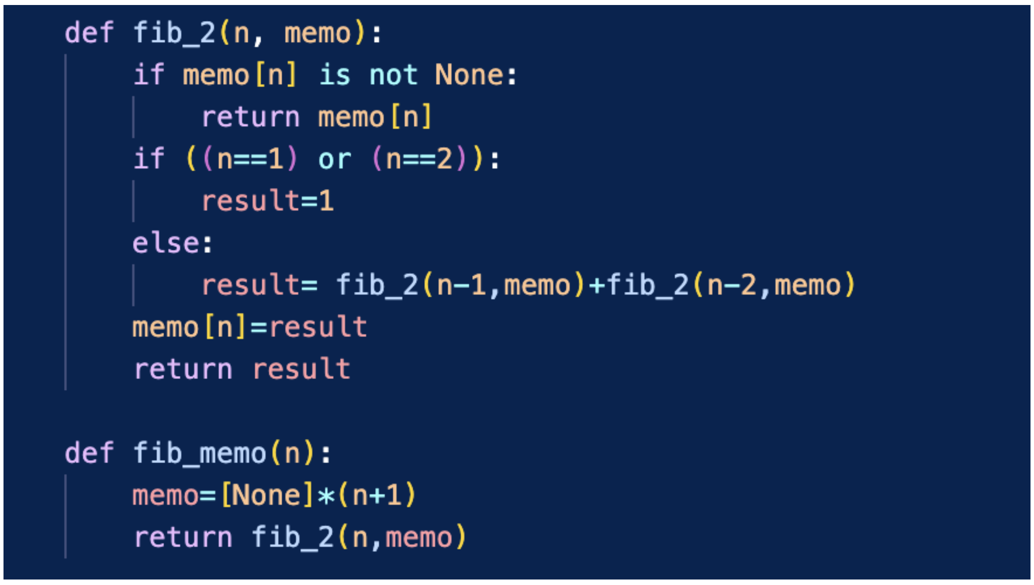
The purpose of caching is to improve the performance of our programs and keep data accessible that can be used later. It basically stores the previously calculated result of the subproblem and uses the stored result for the same subproblem. This removes the extra effort to calculate again and again for the same problem. And we already know that if the same problem occurs again and again, then that problem is recursive in nature.

Memo approach:

The term “Memoization” comes from the Latin word “memorandum” (to remember), which is commonly shortened to “memo” in American English, and which means “to transform the results of a function into something to remember.”.

In computing, memoization is used to speed up computer programs by eliminating the repetitive computation of results, and by avoiding repeated calls to functions that process the same input.

Memoization is a specific form of caching that is used in dynamic programming. The purpose of caching is to improve the performance of our programs and keep data accessible that can be used later. It basically stores the previously calculated result of the subproblem and uses the stored result for the same subproblem. This removes the extra effort to calculate again and again for the same problem. And we already know that if the same problem occurs again and again, then that problem is recursive in nature.



```
def fib_2(n, memo):
    if memo[n] is not None:
        return memo[n]
    if ((n==1) or (n==2)):
        result=1
    else:
        result= fib_2(n-1,memo)+fib_2(n-2,memo)
    memo[n]=result
    return result

def fib_memo(n):
    memo=[None]*(n+1)
    return fib_2(n,memo)
```

Figure 8 Memo algo

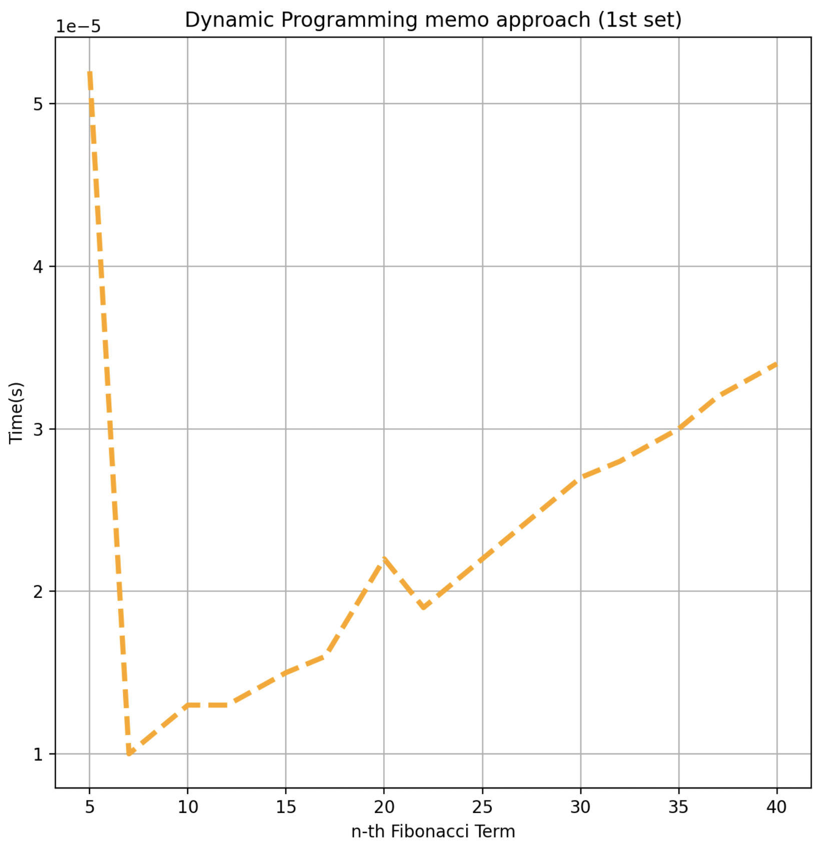


Figure 9

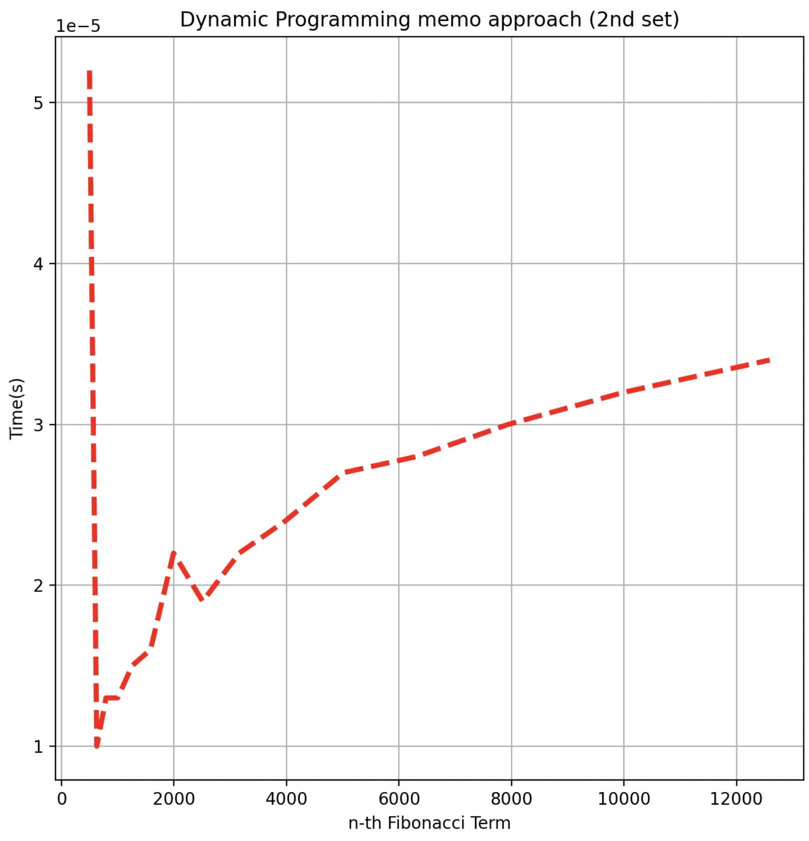
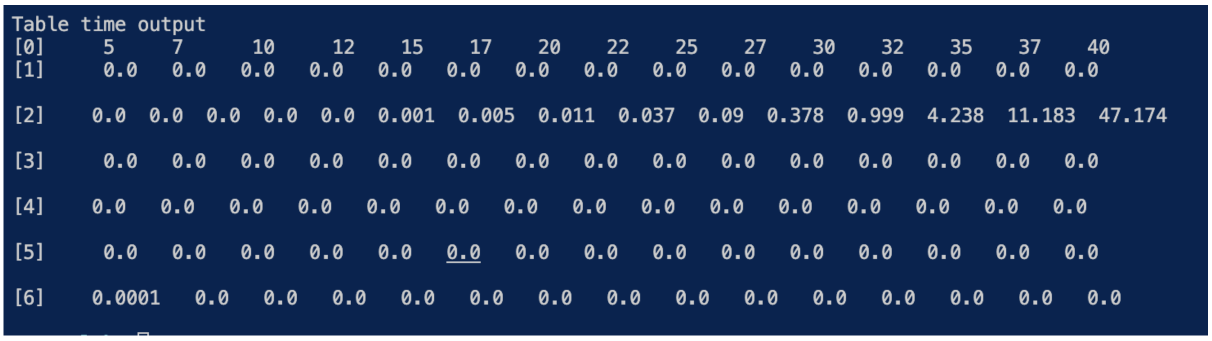


Figure 10

Time Complexity: Linear (more efficient than textbook recursive)

Results:



```
Table time output
[0]      5      7       10      12     15     17     20     22     25     27     30     32     35     37     40
[1]      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0

[2]     0.0  0.0  0.0  0.0  0.0  0.001  0.005  0.011  0.037  0.09  0.378  0.999  4.238  11.183  47.174

[3]      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0

[4]     0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0

[5]      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0

[6]     0.0001    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0
```

Figure 11

Row[4] represents the results to the memo approach,as we can see the output is very much acceptable, but we won't consider this algo for the *top 3 battle*.

Bottom-up approach:

In the bottom-up dynamic programming approach, we'll reorganize the order in which we solve the subproblems.

We'll compute F(0), then F(1), then F(2), and so on:

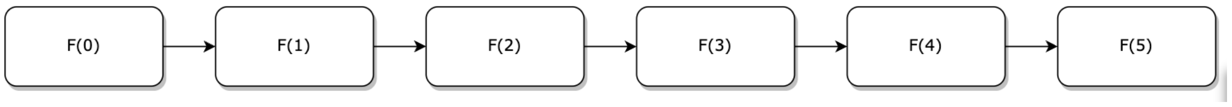
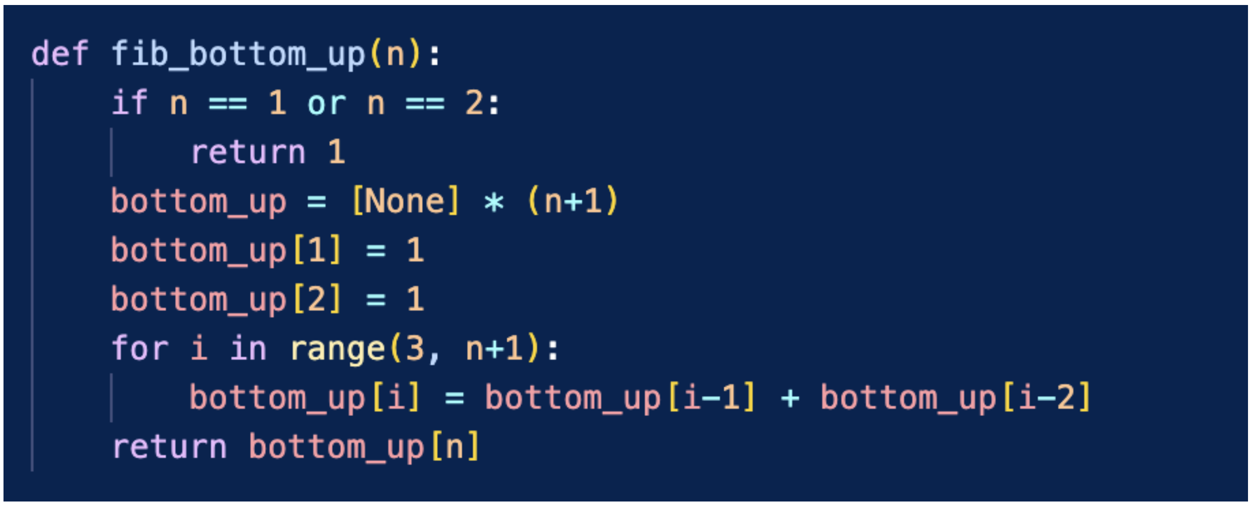


Figure 12

This will allow us to compute the solution to each problem only once, and we'll only need to save two intermediate results at a time.

For example, when we're trying to find F(2), we only need to have the solutions to F(1) and F(0) available. Similarly, for F(3), we only need to have the solutions to F(2) and F(1).



```
def fib_bottom_up(n):
    if n == 1 or n == 2:
        return 1
    bottom_up = [None] * (n+1)
    bottom_up[1] = 1
    bottom_up[2] = 1
    for i in range(3, n+1):
        bottom_up[i] = bottom_up[i-1] + bottom_up[i-2]
    return bottom_up[n]
```

Figure 13

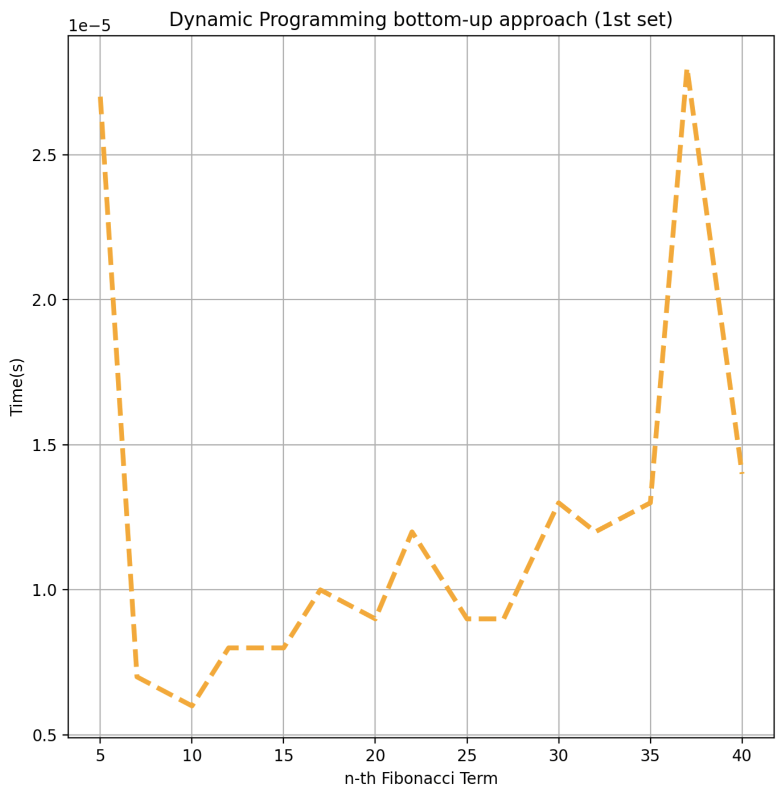


Figure 14

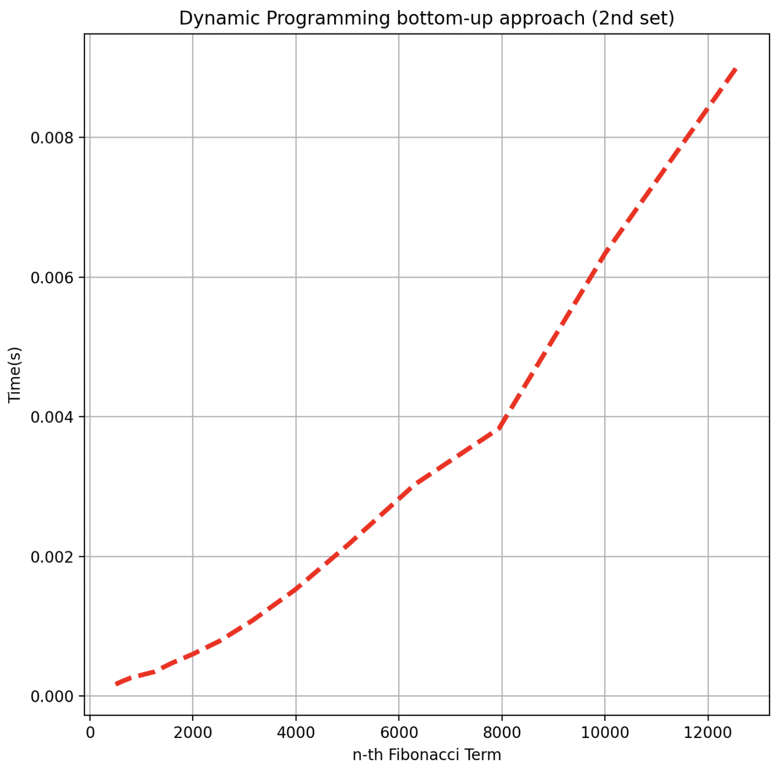
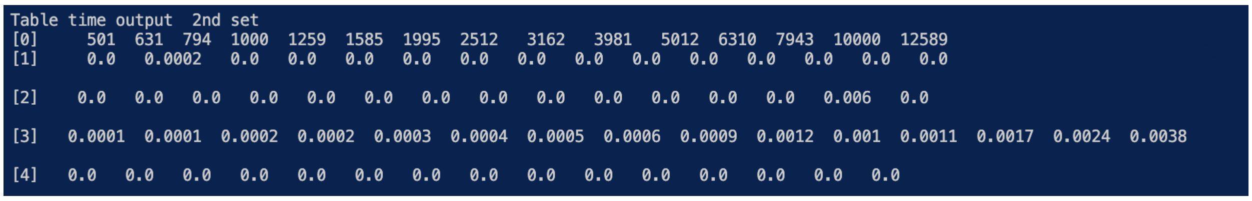


Figure 15

Time Complexity:So the time complexity of the algorithm is also O(N). Since we only use two variables to track our intermediate results, our space complexity is constant, O(1).

Results:



```
Table time output  2nd set
[0]     501  631  794  1000  1259  1585  1995  2512   3162   3981   5012  6310  7943  10000  12589
[1]     0.0   0.0002   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

[2]    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.006    0.0

[3]   0.0001  0.0001  0.0002  0.0002  0.0003  0.0004  0.0005  0.0006  0.0009  0.0012  0.001  0.0011  0.0017  0.0024  0.0038

[4]   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
```

Figure 16

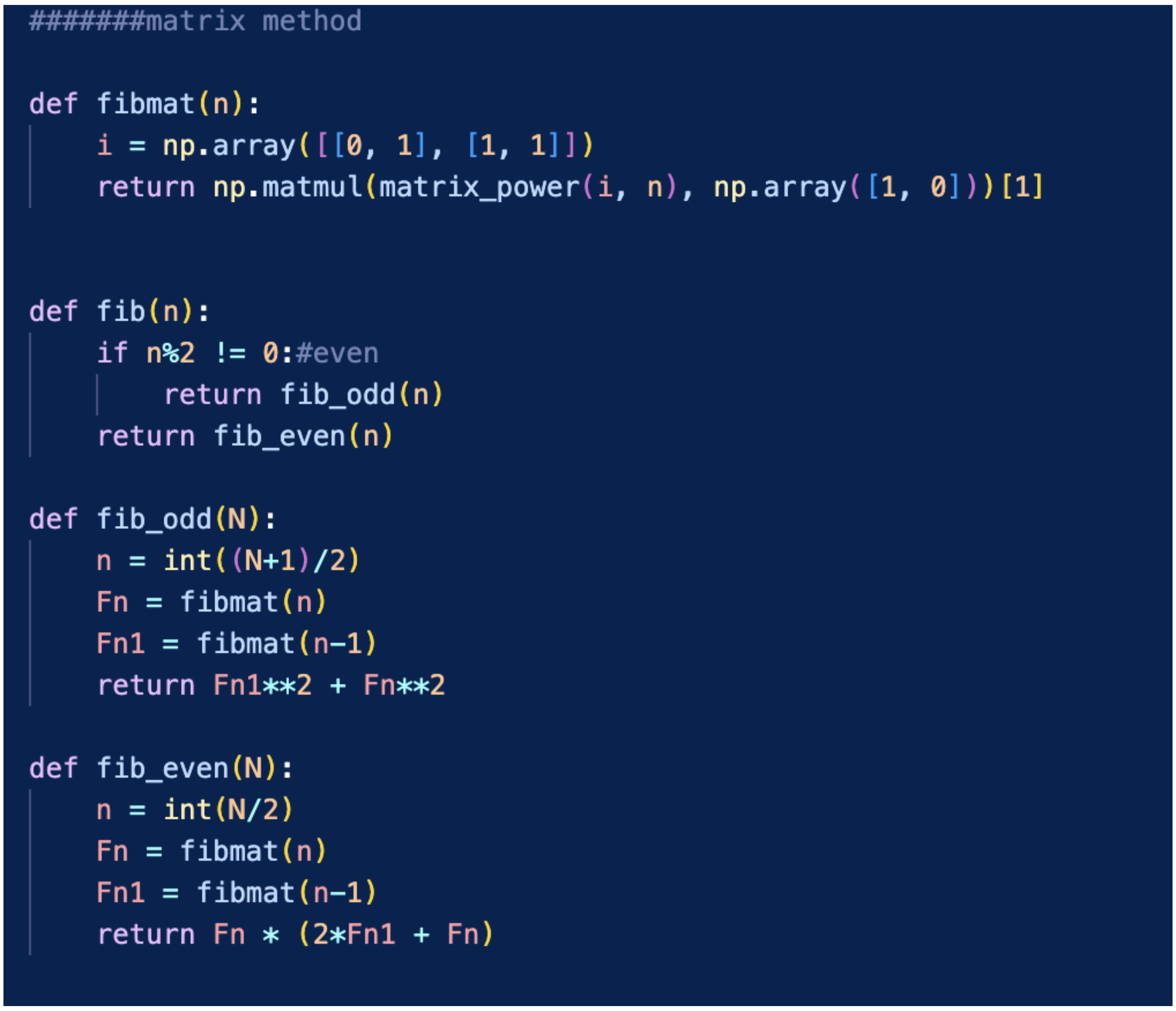
Row[3] describes the time output for the bottom-up method using the second table input array, outstanding results. We will see later how this performs alongside two following algorithms.

Double Fibonacci Matrix Exponentiation:

This is one of the most used techniques in competitive programming. Let us first consider below simple question.

What is the minimum time complexity to find n’th Fibonacci Number?

We can find n’th Fibonacci Number in O(Log n) time using Matrix Exponentiation. In this post, a general implementation of Matrix Exponentiation is discussed.



```
#######matrix method

def fibmat(n):
    i = np.array([[0, 1], [1, 1]])
    return np.matmul(matrix_power(i, n), np.array([1, 0]))[1]


def fib(n):
    if n%2 != 0:#even
        return fib_odd(n)
    return fib_even(n)

def fib_odd(N):
    n = int((N+1)/2)
    Fn = fibmat(n)
    Fn1 = fibmat(n-1)
    return Fn1**2 + Fn**2

def fib_even(N):
    n = int(N/2)
    Fn = fibmat(n)
    Fn1 = fibmat(n-1)
    return Fn * (2*Fn1 + Fn)
```

Figure 17 Matrix Approach

Given F(k)   and F(k+1) , we can calculate these:

F(2k)F(2k+1)=F(k)[2F(k+1)−F(k)].=F(k+1)^2+F(k)^2.

These identities can be extracted from the matrix exponentiation algorithm.
In a sense, this algorithm is the matrix exponentiation algorithm with the redundant calculations removed. It should be a constant factor faster than matrix exponentiation, but the asymptotic time complexity is still the same.

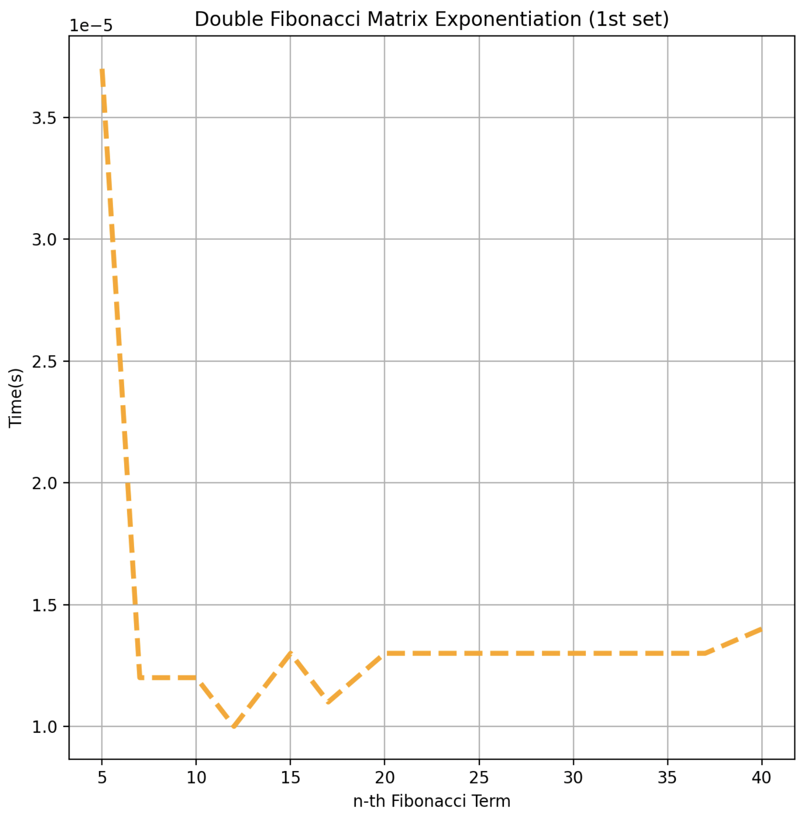


Figure 18

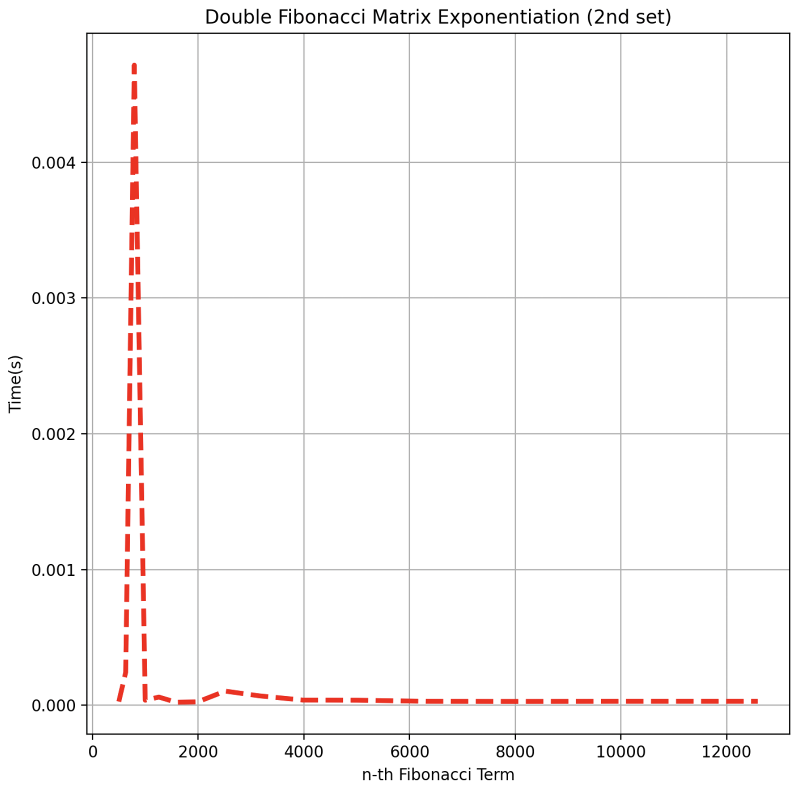
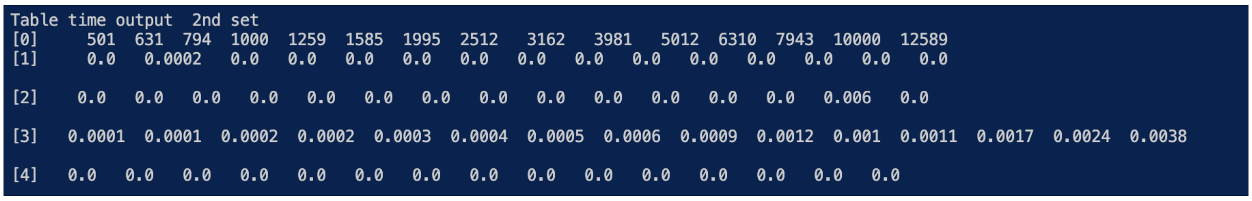


Figure 19

Time Complexity:We can find n'th Fibonacci Number in O(Log n) time using Matrix Exponentiation.

Results:



```
Table time output  2nd set
[0]     501  631  794  1000  1259  1585  1995  2512   3162   3981   5012  6310  7943  10000  12589
[1]     0.0   0.0002   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

[2]    0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.006   0.0

[3]   0.0001  0.0001  0.0002  0.0002  0.0003  0.0004  0.0005  0.0006  0.0009  0.0012  0.001  0.0011  0.0017  0.0024  0.0038

[4]   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
```

Figure 20

Row[1] represents the results to the Matrix Exponentiation method, outstanding results again, even using the 2nd array input case.

Binet's Formula:

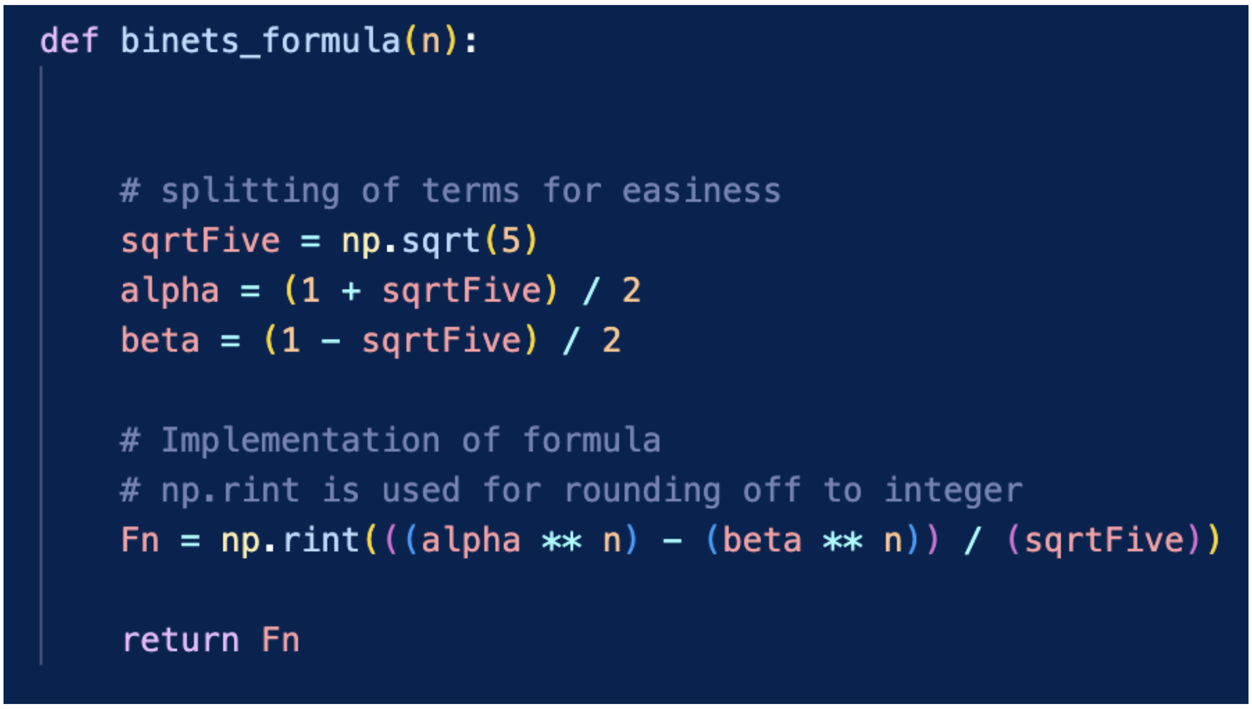
Binet's formula is a special case of the U_n Binet form with m=1, corresponding to the nth Fibonacci number,

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

$$= \frac{\left(1+\sqrt{5}\right)^n - \left(1-\sqrt{5}\right)^n}{2^n\sqrt{5}},$$

Figure 21 Binet's Formula

where phi is the golden ratio. It was derived by Binet in 1843, although the result was known to Euler, Daniel Bernoulli, and de Moivre more than a century earlier.



```python
def binets_formula(n):


    # splitting of terms for easiness
    sqrtFive = np.sqrt(5)
    alpha = (1 + sqrtFive) / 2
    beta = (1 - sqrtFive) / 2

    # Implementation of formula
    # np.rint is used for rounding off to integer
    Fn = np.rint(((alpha ** n) - (beta ** n)) / (sqrtFive))

    return Fn
```

Figure 22 Binet's Formula Approach

After importing math for its sqrt and pow functions we have the function which actually implements Binet's Formula to calculate the value of the Fibonacci Sequence for the given term n. The formula uses √5 no less than three times so I have assigned it to a separate variable, both for efficiency and to make the code slightly more concise.

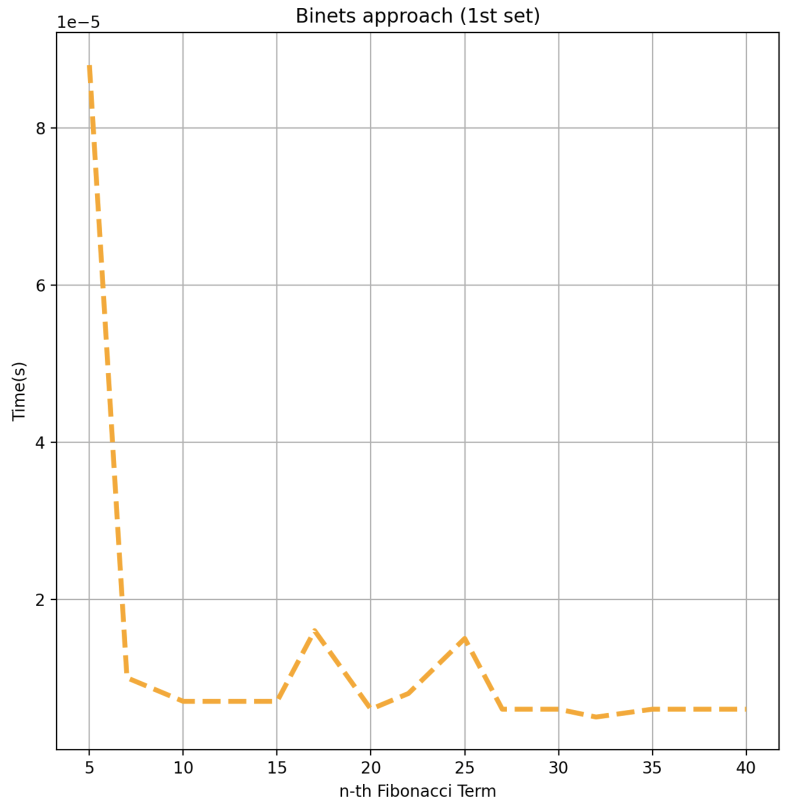


Figure 23

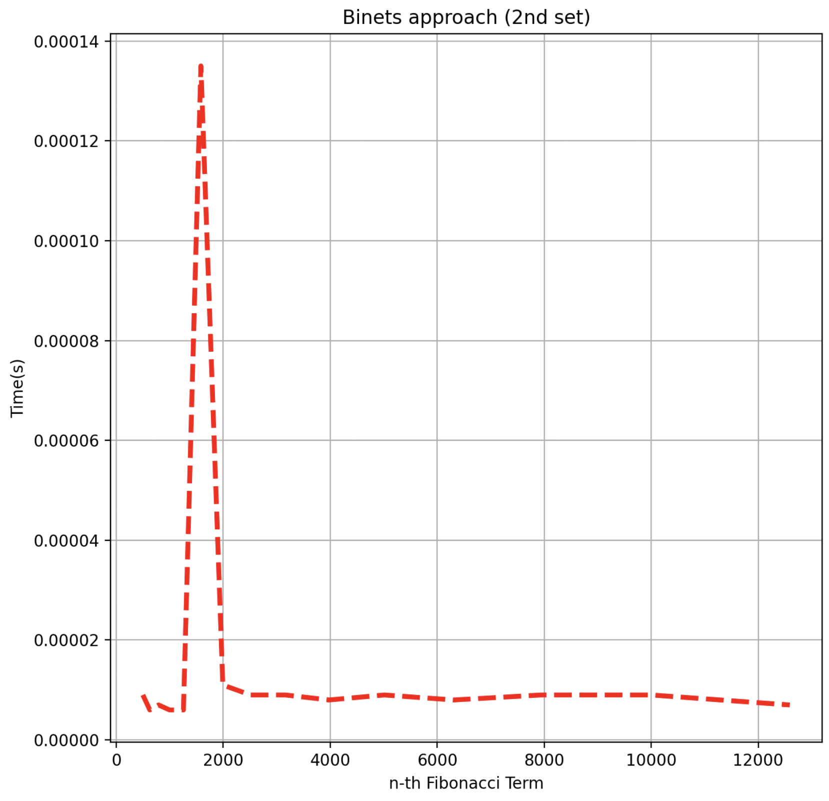


Figure 24

As we can see, this algorithm would technically prove itself to be the most efficient, however, The Binet Formula Function is not accurate enough to be considered within the analyzed limits and is recommended to be used for Fibonacci terms up to 80.

At least in its naïve form in python, as further modification and change of language may extend its usability further.

Final test:

We may draw final conclusions by analyzing the following graph:

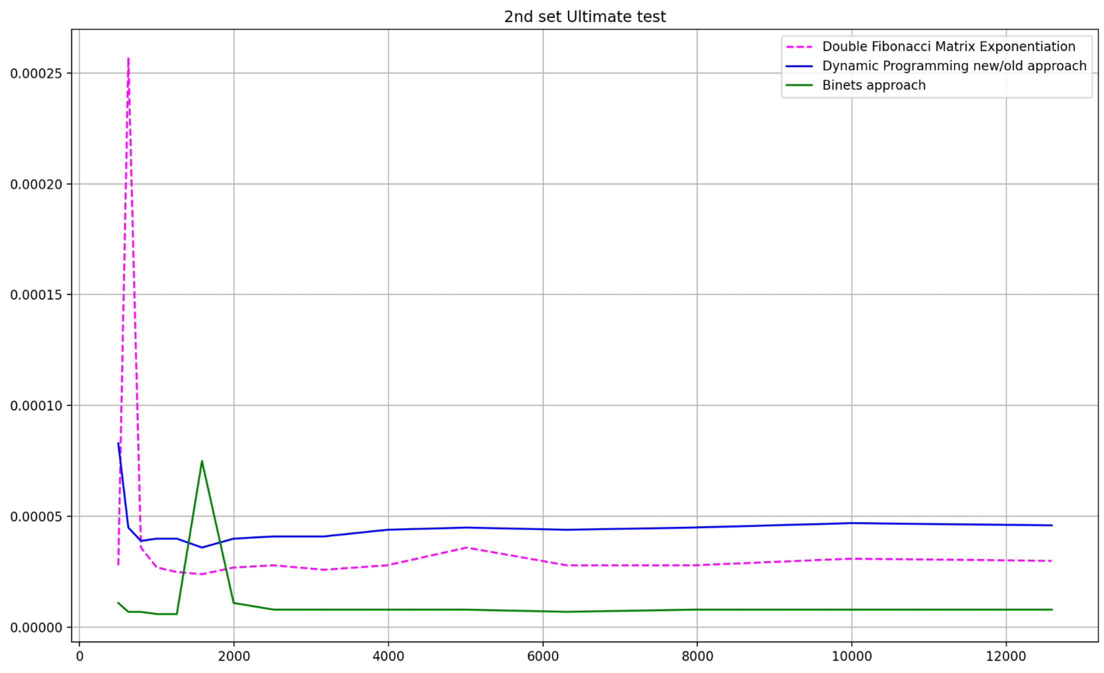


Figure 25

As we can see out of these 3 methods, all technically acceptable for high inputs, nothing proves as efficient as the Binet's formula implementation. This would definitely be more likely to be considered plausible in our case if we were to somehow get rid of the minor errors mentioned in the implementation above.

Conclusions:

Four classes of methods have been evaluated in this paper using empirical analysis to determine how well they perform in terms of accuracy in delivering results as well as execution time complexity, thereby defining the application domains for each method as well as identifying potential future improvements that could be made to make them more practical.

The Recursive approach can be used for smaller order numbers, such as numbers of order up to 30, without putting additional burden on the computer or requiring a test of patience.

It is the simplest to write but also the most challenging to execute due to its exponential time complexity.

The Binet approach, which is the simplest to use and has a nearly constant time complexity, can be used to compute numbers up to order 80 when the recursive method is rendered unworkable.

Given that its formula incorporates the Golden Ratio, it is advised that the findings be double-checked depending on the language being utilized.

Additional Fibonacci numbers can be calculated using the Dynamic Programming and Matrix Multiplication Methods, both of which produce accurate results and have a linear complexity in their innocence that could be lowered to a logarithmic scale with more tricks and optimizations.

Repository link: https://github.com/andreeamnl/AlgorithmAnalysisCourse.git