Ministry of Education, Culture and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work No. 1

Discipline: Algorithms' Analysis

Topic: Study and empirical analysis of algorithms for determining Fibonacci n-th term

Elaborated:
st. gr. FAF-211 Țărnă Cristina

Verified:
univ. asist. 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 theefficiency 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 (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: $x_n = x_{n-1} + x_{n-2}$.

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 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

All four algorithms will be implemented in their naïve form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar toother experimental observations, the particular efficiency in rapport with input will vary depending on memory of the device used.

The error margin determined will constitute 2.5 seconds as per experimental measurement.

1. Recursive Method

The recursive method, also considered the most inefficient method, follows a straightforwardapproach of computing the nth term by computing its predecessors first, and then adding them.

However, the method does it by calling upon itself a number of times and repeating the same operation,for the same term, at least twice, occupying additional memory and, in theory, doubling its execution time.

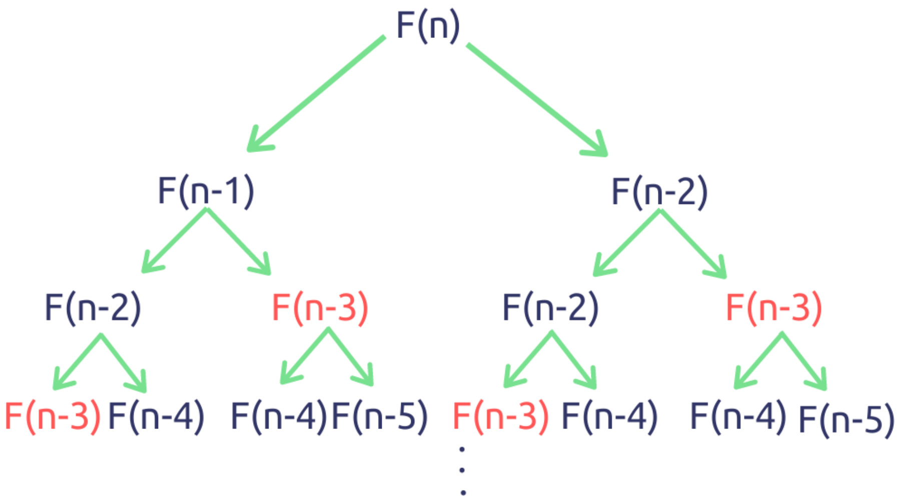

Figure 1. Fibonacci Sequence Algorithm

Implementation:

```
In 1
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

Figure 2. Fibonacci recursion in Python

Results:

After running the function for each n Fibonacci term in the list and saving the time far each n.

	Fibonacci term number	Fibonacci number	Execution time(wall time)	Execution time(CPU time)
0	5	5	0.0	0.0
1	7	13	0.0	0.0
2	10	55	0.0	0.0
3	12	144	0.0	0.0
4	15	610	0.997782	0.0
5	17	1597	0.0	0.0
6	20	6765	1.992702	0.0
7	22	17711	3.985643	0.0
8	25	75025	18.937826	15.625
9	27	196418	128.570557	78.125
10	30	832040	223.247528	218.75
11	32	2178309	637.868881	593.75
12	35	9227465	2661.147118	2640.625
13	37	24157817	7418.964386	7187.5
14	40	102334155	27649.382591	27375.0
15	42	267914296	74958.930016	74593.75
16	45	1134903170	310850.13628	310515.625

Figure 3. Result table Recursion

The plot:

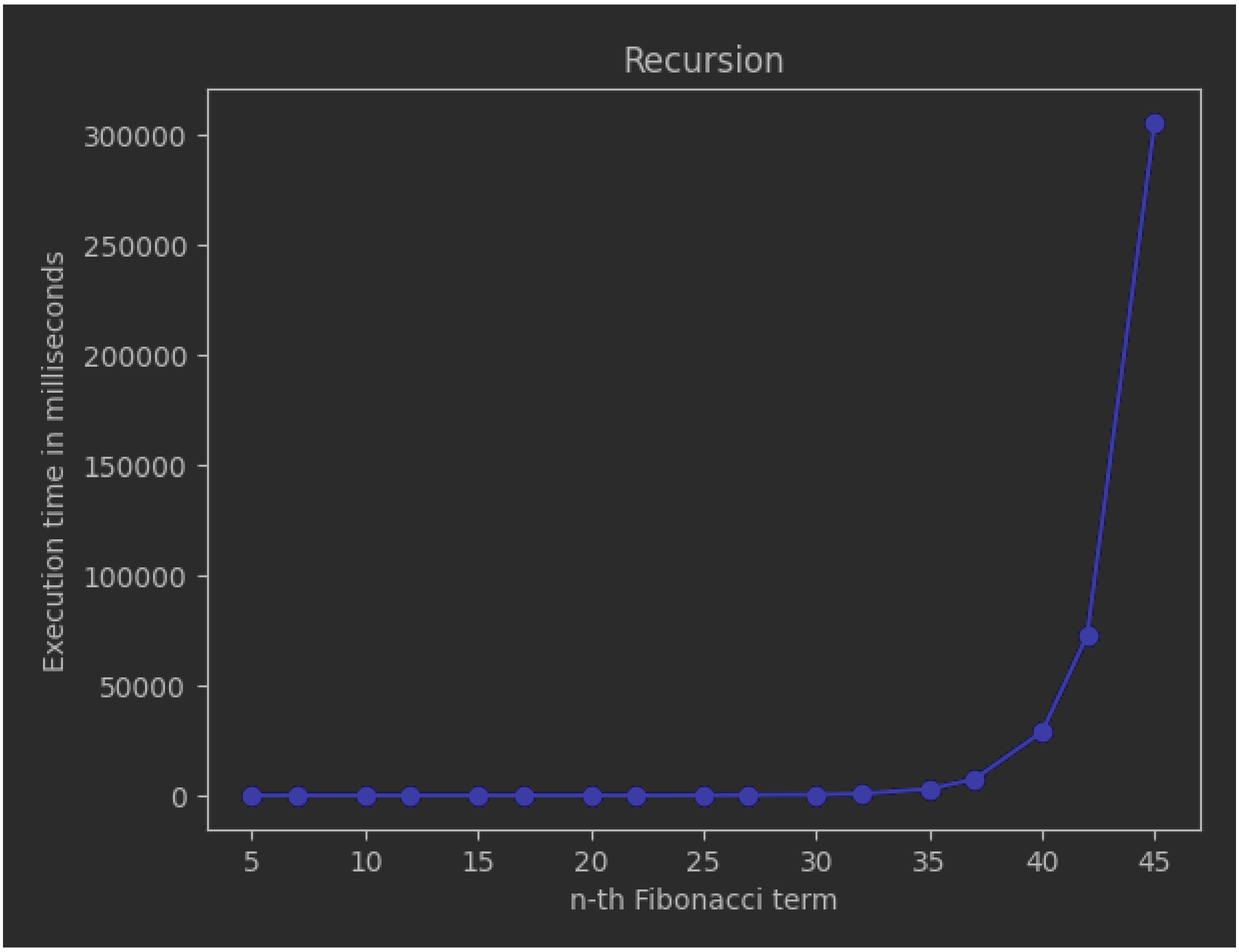

Figure 4. Plot of execution time Recursion

We can see from the plot that the time complexity is exponential.

2. Dynamic programming

The Dynamic Programming method, similar to the recursive method, takes the straightforward approach of calculating the n^{th} term. However, instead of calling the function upon itself, from top down it operates based on an array data structure that holds the previously computed terms, eliminating the need to recompute them.

Implementation:

```
In 18
def fib_dynamic(n):
    f = [0, 1]

    for j in range(2, n + 1):
        f.append(f[j - 1] + f[j - 2])
    return f[n]
```

Figure 5. Fibonacci dynamic programming in Python

Results:

	Fibonacci term number	Fibonacci number	Execution time(wall time)	Execution time(CPU time)
0	5	5	0.0	0.0
1	7	13	0.0	0.0
2	10	55	0.0	0.0
3	12	144	0.0	0.0
4	15	610	0.0	0.0
5	17	1597	0.0	0.0
6	20	6765	0.0	0.0
7	22	17711	0.0	0.0
8	25	75025	0.0	0.0
9	27	196418	0.0	0.0
10	30	832040	0.0	0.0
11	32	2178309	0.0	0.0
12	35	9227465	0.0	0.0
13	37	24157817	0.0	0.0
14	40	102334155	0.0	0.0
15	42	267914296	0.0	0.0
16	45	1134903170	0.0	0.0

Figure 6. Result using small numbers as in recursive example

	Fibonacci term number	Fibonacci number	Execution time(wall time)	Execution time(CPU time)
0	501	225591516161936330872512695036072072046...	0.0	0.0
1	631	332442083566252894269656048603015447982...	0.0	0.0
2	794	386101382166516474039396022520588129064...	0.0	0.0
3	1000	434665576869374564356885276750406258025...	1.009464	0.0
4	1259	583383407137109552695513219117000889452...	1.945734	0.0
5	1585	786909510268431451838838092589415915156...	2.003193	0.0
6	1995	380940630740594205573177927121038274638...	1.015186	0.0
7	2512	424099056011457202318232906058702243692...	2.015591	0.0
8	3162	294736192096351878735266163626864096063...	1.008272	0.0
9	3981	426884939334625738047084970564493277587...	0.926018	0.0
10	5012	124901579571097080178854699904336007676...	1.993895	0.0
11	6310	230422548253450925729996115890641097929...	5.982161	15.625
12	7943	435854469063995252923647608915864667455...	5.974293	15.625
13	10000	336447648764317832666216120051075433103...	20.048857	31.25
14	12589	394382926024589284651517836653329227676...	11.956453	15.625
15	15849	786367072004300968376836217969412585613...	9.923697	15.625

Figure 7. Results using big numbers - dynamic

Plot:

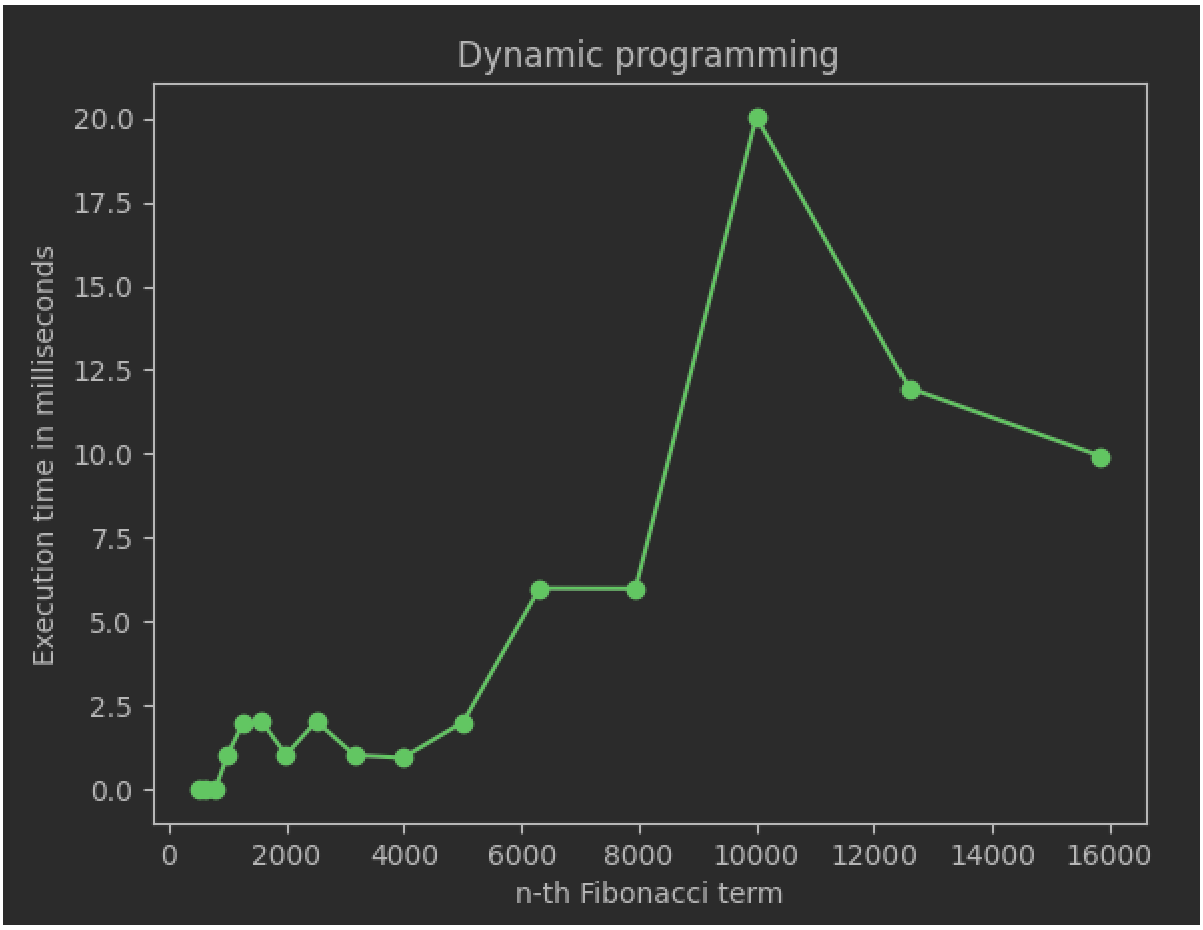

Figure 8. Plot of dynamic programming method with big numbers

Time complexity: O(n) for given n

Auxiliary space: O(n)

3. Optimized Dynamic programming

The Optimized Dynamic Programming method, is similar to the basic method, however it just stores the 2 numbers needed for the next Fibonacci number.

Implementation:

```python
def fib_opt_dynamic(n):
    a = 0
    b = 1
    if n < 0:
        print("Incorrect input")
    elif n == 0:
        return a
    elif n == 1:
        return b
    else:
        for k in range(2,n+1):
            c = a + b
            a = b
            b = c
        return b
```

Figure 9. Fibonacci Optimized Dynamic Programming in Python

Results:

	Fibonacci term number	Fibonacci number	Execution time(wall time)	Execution time(CPU time)
0	501	2255915161619363308725126950360720720 46...	0.0	0.0
1	631	332442083566252894269656048603015447982...	0.0	0.0
2	794	386101382166516474039396022520588129064...	0.0	0.0
3	1000	434665576869374564356885276750406258025...	0.0	0.0
4	1259	583383407137109552695513219117000889452...	0.0	0.0
5	1585	786909510268431451838838092589415915156...	0.0	0.0
6	1995	380940630740594205573177927121038274638...	0.0	0.0
7	2512	424099056011457202318232906058702243692...	0.0	0.0
8	3162	294736192096351878735266163626864096063...	0.995874	15.625
9	3981	426884939334625738047084970564493277587...	0.996113	0.0
10	5012	124901579571097080178854699904336007676...	0.0	0.0
11	6310	230422548253450925729996115890641097929...	0.995874	0.0
12	7943	435854469063995252923647608915864667455...	1.993656	0.0
13	10000	336447648764317832666216120051075433103...	0.995636	0.0
14	12589	394382926024589284651517836653329227676...	2.990007	0.0
15	15849	786367072004300968376836217969412585613...	4.983425	15.625

Figure 10. Result table Optimized Dynamic Programming

Plot:

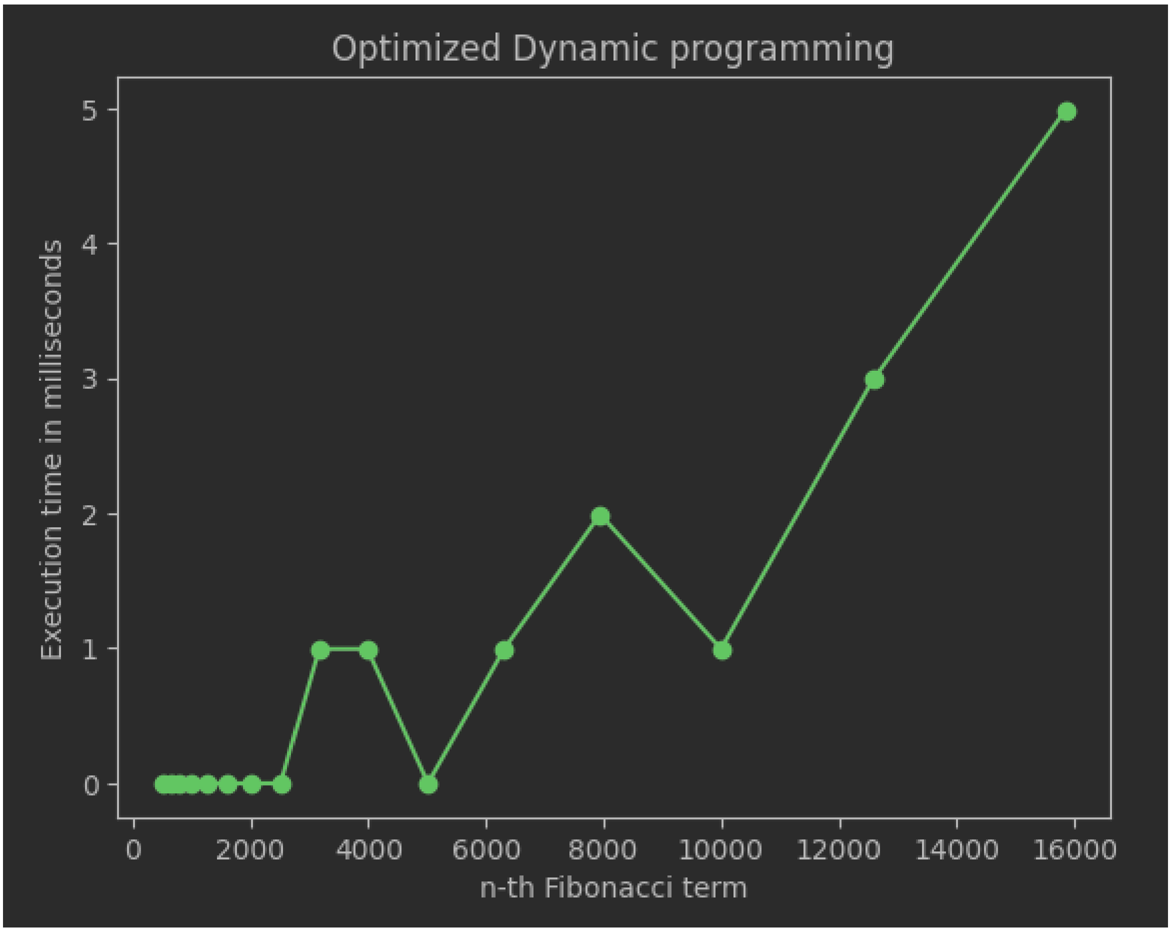

Figure 11. Plot of execution time Optimized Dynamic Programming method

Time Complexity: O(n)

Extra Space: O(1)

4. Matrix Power Method

Implementation:

```python
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

def power(F, n):

    M = [[1, 1],
         [1, 0]]

    for l in range(2, n + 1):
        multiply(F, M)

def fib_mat(n):
    F = [[1, 1],
         [1, 0]]
    if n == 0:
        return 0
    power(F, n - 1)

    return F[0][0]
```

Figure 12. Fibonacci Power Matrix Method in Python

Result:

	Fibonacci term number	Fibonacci number	Execution time(wall time)	Execution time(CPU time)
0	501	225591516161936330872512695036072072046...	0.99802	0.0
1	631	332442083566252894269656048603015447982...	0.0	0.0
2	794	386101382166516474039396022520588129064...	0.0	0.0
3	1000	434665576869374564356885276750406258025...	0.0	0.0
4	1259	583383407137109552695513219117000889452...	0.996113	0.0
5	1585	786909510268431451838838092589415915156...	0.0	0.0
6	1995	380940630740594205573177927121038274638...	0.0	0.0
7	2512	424099056011457202318232906058702243692...	1.015186	0.0
8	3162	294736192096351878735266163626864096063...	0.997066	0.0
9	3981	426884939334625738047084970564493277587...	0.0	0.0
10	5012	124901579571097080178854699904336007676...	0.99659	0.0
11	6310	230422548253450925729996115890641097929...	0.996828	0.0
12	7943	435854469063995252923647608915864667455...	1.995087	0.0
13	10000	336447648764317832666216120051075433103...	1.992464	0.0
14	12589	394382926024589284651517836653329227676...	1.991749	15.625
15	15849	786367072004300968376836217969412585613...	3.987312	0.0

Figure 13. Result table of Power Matrix Method

Plot:

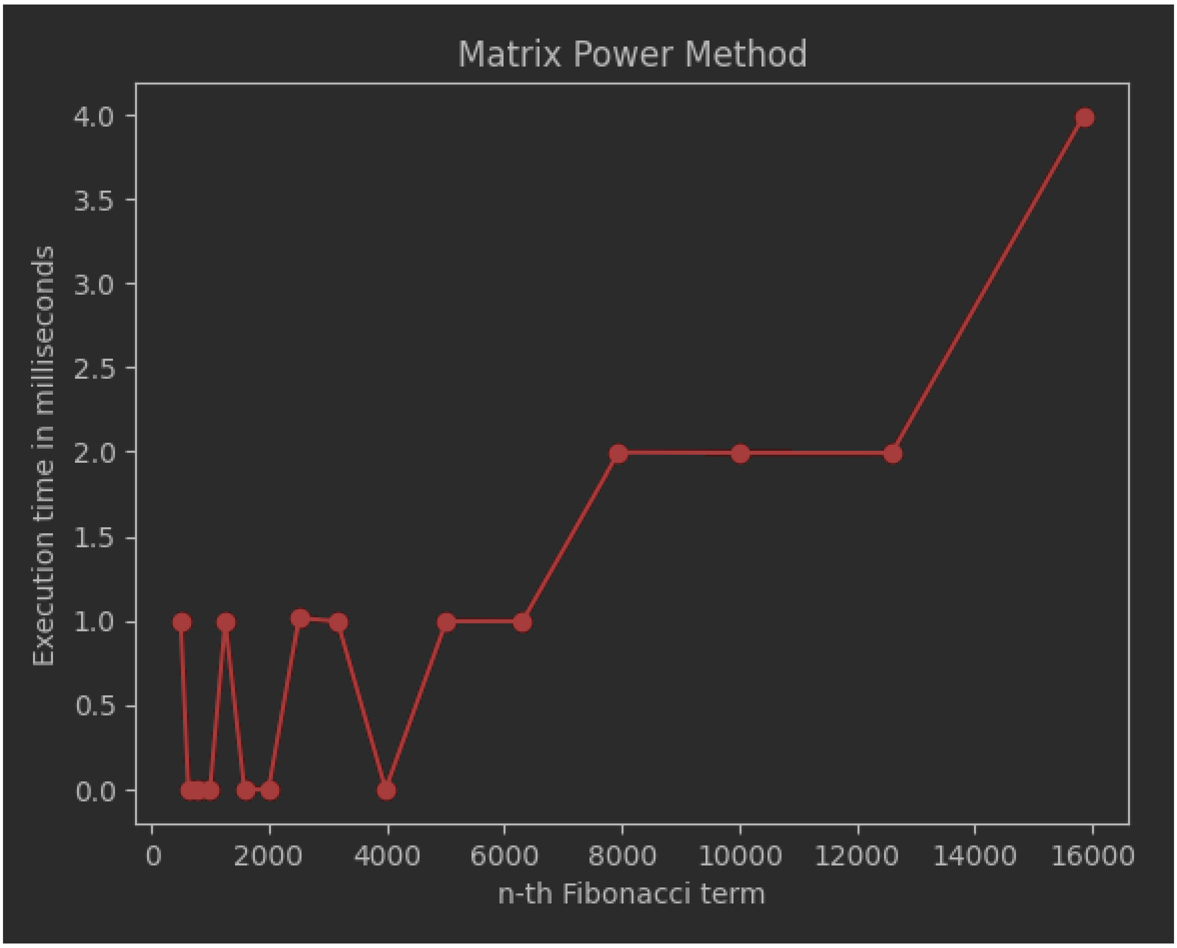

Figure 14. Plot time execution of Matrix Power method

Time Complexity: O(n)

Auxiliary Space: O(1)

5. O($\log_n$) time method

Algorithm description:

If n is even then k = n/2:

F(n) = [2*F(k-1) + F(k)]*F(k)

If n is odd then k = (n + 1)/2

F(n) = F(k)*F(k) + F(k-1)*F(k-1)

Taking determinant on both sides, we get $(-1)^n = F_{n+1}F_{n-1} - F_n^2$

Moreover, since $A^nA^m = A^{n+m}$ for any square matrix A, the following identities can be derived (they are obtained from two different coefficients of the matrix product)

$F_mF_n + F_{m-1}F_{n-1} = F_{m+n-1}$ ---------------------------(1)

By putting n = n+1 in equation (1),

$F_mF_{n+1} + F_{m-1}F_n = F_{m+n}$ --------------------------(2)

Putting m = n in equation (1).

$F_{2n-1} = F_n^2 + F_{n-1}^2$

Putting m = n in equation (2)

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

(By putting $F_{n+1} = F_n + F_{n-1}$)

To get the formula to be proved, we simply need to do the following

If n is even, we can put k = n/2

If n is odd, we can put k = (n+1)/2

Implementation:

```python
MAX = 1000

# Create an array for memoization
f = [0] * MAX

# Returns n-th fibonacci number using table f[]
def fib(n) :
    # Base cases
    if n == 0:
        return 0
    if n == 1 or n == 2:
        f[n] = 1
        return f[n]

    # If fib(n) is already computed
    if f[n]:
        return f[n]

    if n & 1 :
        k = (n + 1) // 2
    else :
        k = n // 2

    # Applying above formula Note value n&1 is 1
    # if n is odd, else 0.
    if n & 1:
        f[n] = (fib(k) * fib(k) + fib(k-1) * fib(k-1))
    else :
        f[n] = (2*fib(k-1) + fib(k))*fib(k)

    return f[n]
```

Figure 15. Fibonacci in Python

Results:

	Fibonacci term number	Fibonacci number	Execution time(wall time)	Execution time(CPU time)
0	501	225591516161936330872512695036072072046...	0.994205	0.0
1	631	332442083566252894269656048603015447982...	0.0	0.0
2	794	386101382166516474039396022520588129064...	0.0	0.0
3	1000	434665576869374564356885276750406258025...	0.0	0.0
4	1259	583383407137109552695513219117000889452...	0.0	0.0
5	1585	786909510268431451838838092589415915156...	0.0	0.0
6	1995	380940630740594205573177927121038274638...	0.0	0.0
7	2512	424099056011457202318232906058702243692...	0.0	0.0
8	3162	294736192096351878735266163626864096063...	0.0	0.0
9	3981	426884939334625738047084970564493277587...	0.0	0.0
10	5012	124901579571097080178854699904336007676...	0.99206	0.0
11	6310	230422548253450925729996115890641097929...	1.950264	0.0
12	7943	435854469063995252923647608915864667455...	0.991583	0.0
13	10000	336447648764317832666216120051075433103...	0.985146	0.0
14	12589	394382926024589284651517836653329227676...	1.995564	0.0
15	15849	786367072004300968376836217969412585613...	4.985094	15.625

Figure 16. Result table method 5

Plot:

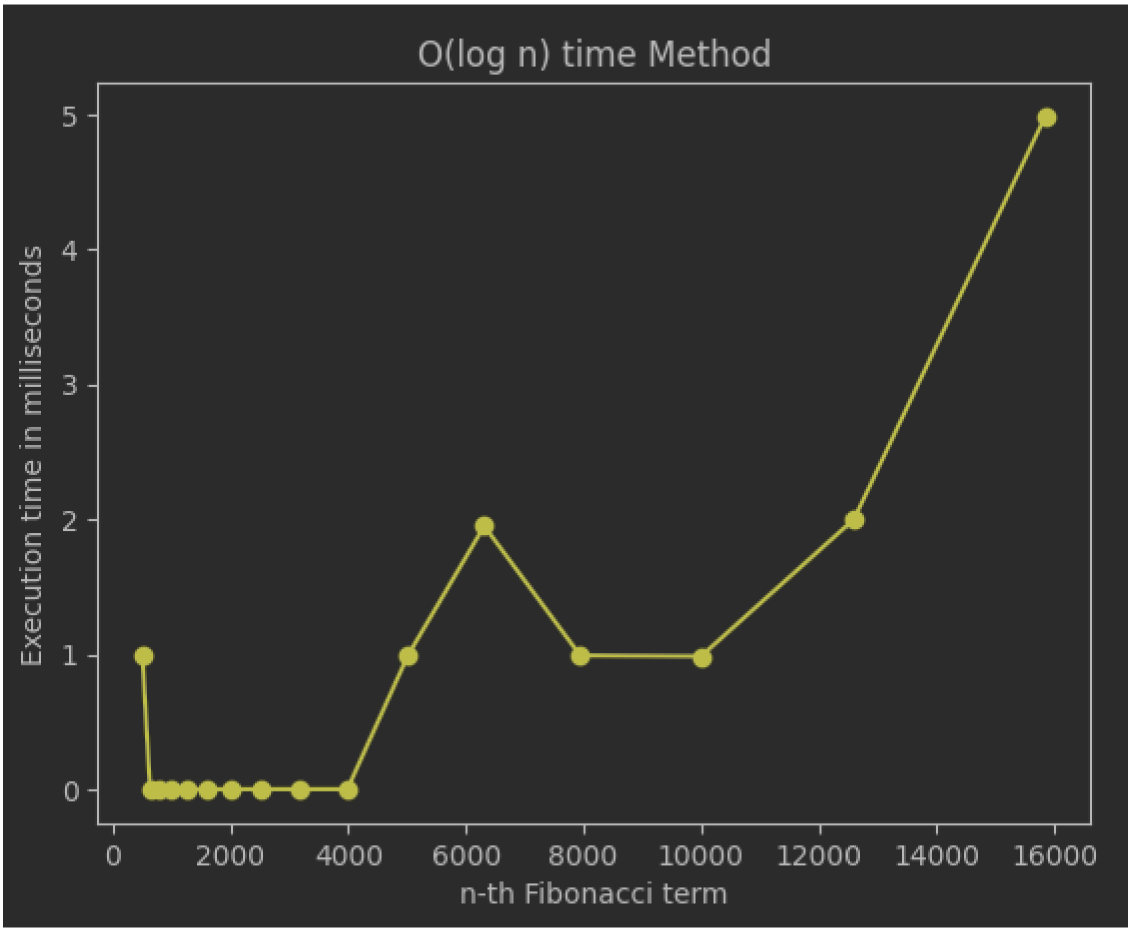

Figure 17. Plot execution time

Time Complexity: O (Log n), as we divide the problem in half in every recursive call.

Auxiliary Space: O(n)

6. Binet's formula method

I directly implement the formula for the nth term in the Fibonacci series. $F_n = \{[(\sqrt{5} + 1)/2] \wedge n\} / \sqrt{5}$

However, this method will fail for n>=71

Implementation:

```python
def fib_b(n):
    phi = (1 + math.sqrt(5)) / 2

    return round(pow(phi, n) / math.sqrt(5))
```

Figure 18. Binet's formula in Python

Result:

	Fibonacci term number	Fibonacci number	Execution time(wall time)	Execution time(CPU time)
0	501	225591516161936330872512695036072072046...	0.0	0.0
1	631	332442083566252894269656048603015447982...	0.0	0.0
2	794	386101382166516474039396022520588129064...	0.0	0.0
3	1000	434665576869374564356885276750406258025...	0.0	0.0
4	1259	583383407137109552695513219117000889452...	0.998259	0.0
5	1585	786909510268431451838838092589415915156...	0.997066	0.0
6	1995	380940630740594205573177927121038274638...	0.0	0.0
7	2512	424099056011457202318232906058702243692...	0.0	0.0
8	3162	294736192096351878735266163626864096063...	0.998259	0.0
9	3981	426884939334625738047084970564493277587...	0.998497	0.0
10	5012	124901579571097080178854699904336007676...	0.0	0.0
11	6310	230422548253450925729996115890641097929...	0.992298	0.0
12	7943	435854469063995252923647608915864667455...	0.995159	0.0
13	10000	336447648764317832666216120051075433103...	1.992941	15.625
14	12589	394382926024589284651517836653329227676...	1.992226	0.0
15	15849	786367072004300968376836217969412585613...	3.988028	0.0

Figure 19. Result table execution time method 6

Plot:

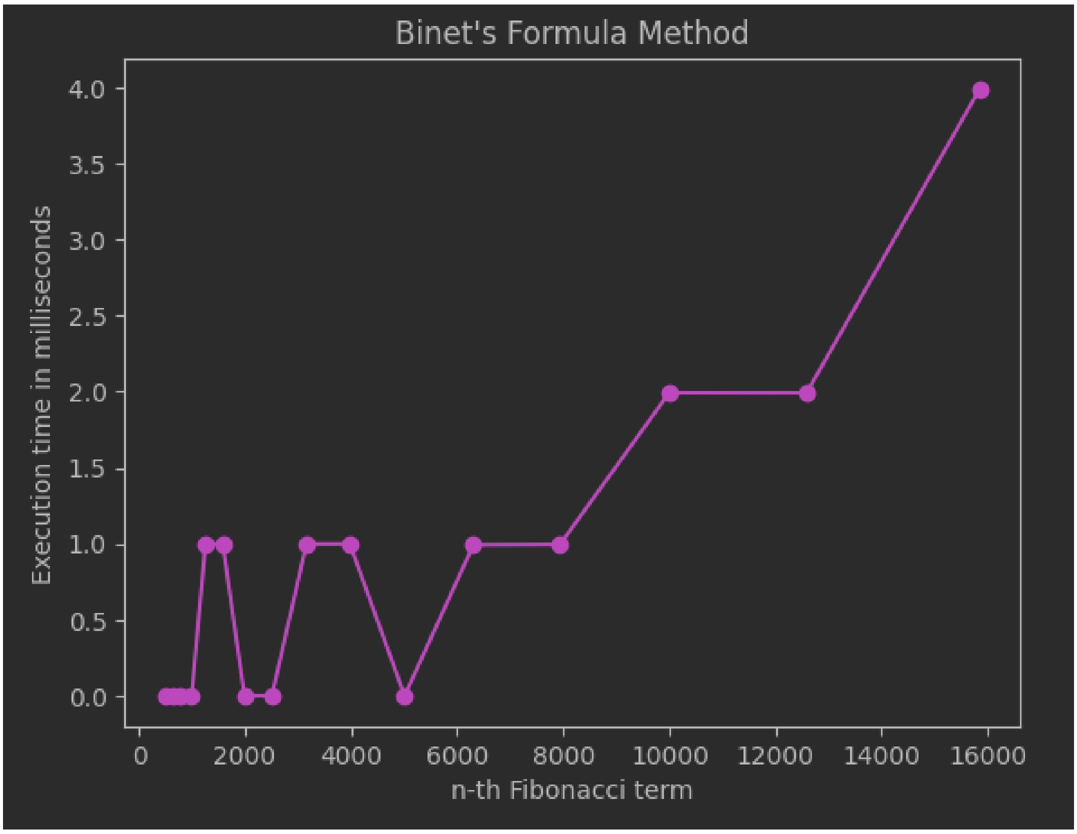

Figure 20. Plot time execution of Binet's Formula

Time Complexity: O(logn), this is because calculating phi^n takes logn time
Auxiliary Space: O(1)

7. DP using memoization

Implementation:

```python
# Initialize array of dp
dp = [-1 for i in range(10)]

def fib_(n):
    if n <= 1:
        return n
    global dp

    if dp[n - 1] != -1:
        first = dp[n - 1]
    else:
        first = fib(n - 1)
    if dp[n - 2] != -1:
        second = dp[n - 2]
    else:
        second = fib(n - 2)
    dp[n] = first + second

    # Memoization
    return dp[n]
```

Figure 21. Python implementation

Results:

	Fibonacci term number	Fibonacci number	Execution time(wall time)	Execution time(CPU time)
0	501	2255915161619363308725126950360720720 46...	0.0	0.0
1	631	33244208356625289426965604860301544 7982...	0.0	0.0
2	794	386101382166516474039396022520588129064...	0.0	0.0
3	1000	434665576869374564356885276750406258025...	0.0	0.0
4	1259	583383407137109552695513219117000889452...	0.0	0.0
5	1585	786909510268431451838838092589415915156...	0.0	0.0
6	1995	380940630740594205573177927121038274638...	0.0	0.0
7	2512	424099056011457202318232906058702243692...	0.0	0.0
8	3162	294736192096351878735266163626864096063...	0.0	0.0
9	3981	426884939334625738047084970564493277587...	0.996828	0.0
10	5012	124901579571097080178854699904336007676...	0.0	0.0
11	6310	230422548253450925729996115890641097929...	0.994205	0.0
12	7943	435854469063995252923647608915864667455...	0.993967	0.0
13	10000	336447648764317832666216120051075433103...	2.988815	0.0
14	12589	394382926024589284651517836653329227676...	3.010273	0.0
15	15849	786367072004300968376836217969412585613...	5.978107	0.0

Figure 22. Result table of execution time

Plot:

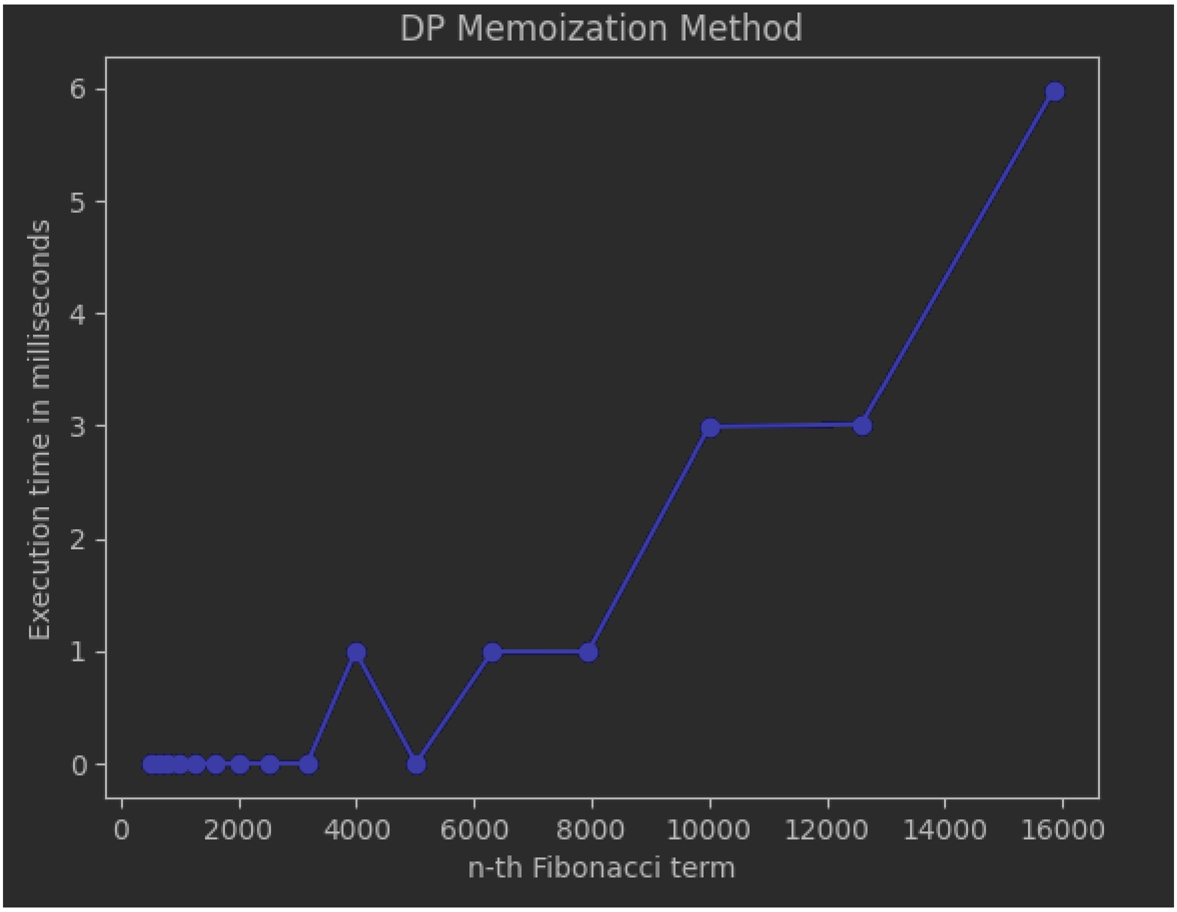

Figure 23. Plot of execution time

Time Complexity: O(n)
Auxiliary Space: O(n)

Now let's compare all the algorithms.
Plot:

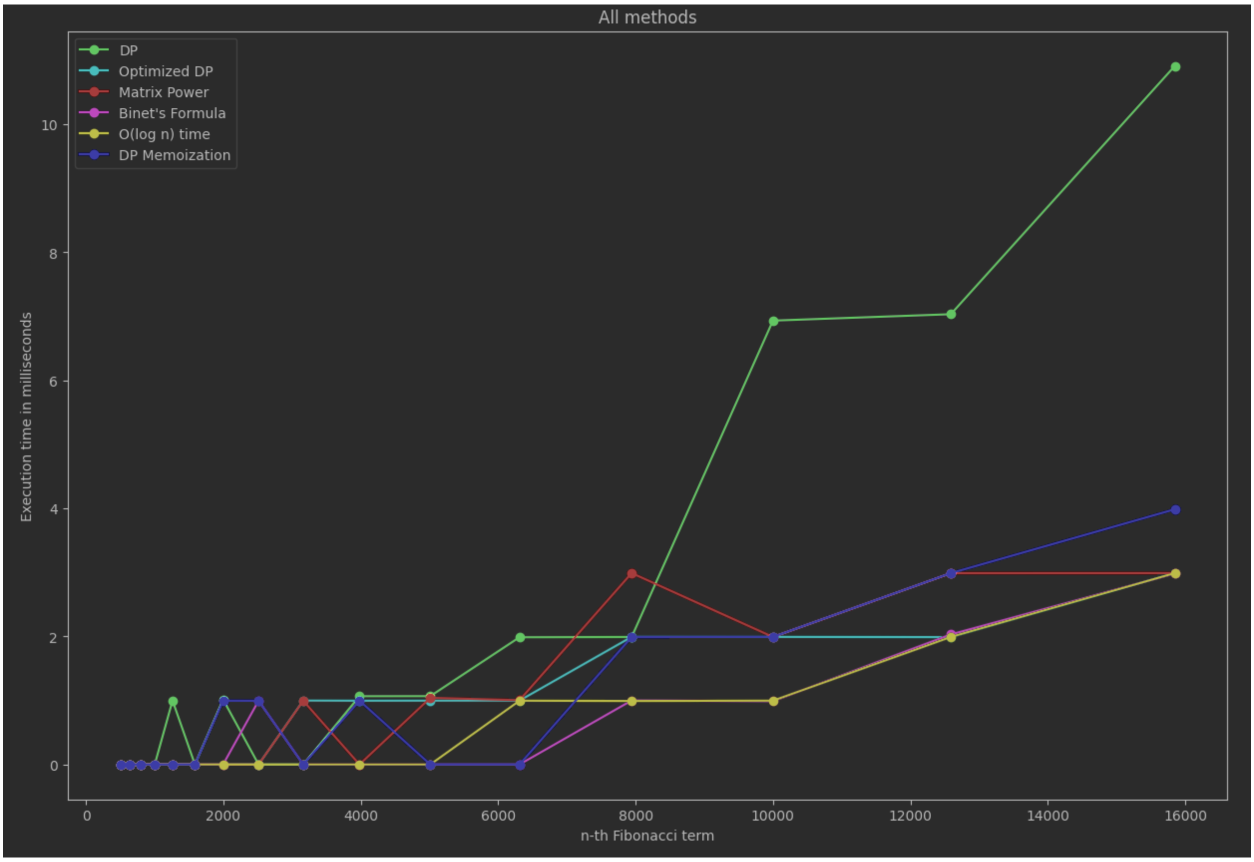

Figure 24. Plot all methods' time execution

Conclusion from figure 24 is that Binet's formula, O (log $_n$) time and Optimized DP seem to be the fastest algorithms. But we should keep in mind that Binet's formula gives us just an approximation.

CONCLUSION

Through Empirical Analysis, within this paper, 7 methods have been tested in their efficiency at both their providing of accurate results, as well as at the time complexity required for their execution, to delimit the scopes within which each could be used, as well as possible improvements that could be further done to make them more feasible.

The Recursive method, being the easiest to write, but also the most difficult to execute with an exponential time complexity, can be used for smaller order numbers, such as numbers of order up to 30 with no additional strain on the computing machine and no need for testing of patience.

The Binet method, the easiest to execute with an almost constant time complexity, could be used when computing numbers of order up to 70, as there could rounding errors dueto its formula that uses the Golden Ratio.

The Dynamic Programming gives us an exact answer and fast. However, Matrix Multiplication, DP Memoization, Optimized DP, and O (log $_n$) time methods are even faster.