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

RAPORT

Laboratory work number 1
at the Analysis and Design of Algorithms

A efectuat:
st. gr. FAF-213 Gutu Dinu

A verificat:
asist. univ. Fistic Cristofor

Chişinău - 2023

Theme: Analysis and study of different algorithms for determining the n-th Fibonacci term

Objectives:

1. Empirical analysis of the algorithms
2. Theoretical analysis of the algorithms
3. Asymptotic complexity of the algorithms

Tasks:

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

Algorithm Analysis

Theoretical notes:

Empirical analysis is an alternative to mathematical analysis in evaluating algorithm complexity. It can be used to gain insight into the complexity class of an algorithm, compare the efficiency of different algorithms for solving similar problems, evaluate the performance of different implementations of the same algorithm, or examine the efficiency of an algorithm on a specific computer. The typical steps in empirical analysis include defining the purpose of the analysis, choosing a metric for efficiency such as number of operations or execution time, determining the properties of the input data, implementing the algorithm in a programming language, generating test data, running the program on the test data, and analyzing the results. The choice of efficiency metric depends on the purpose of the analysis, with number of operations being appropriate for complexity class evaluation and execution time being more relevant for implementation performance. After the program is run, the results are recorded and synthesized through calculations of statistics or by plotting a graph of problem size against efficiency measure.

Introduction:

In mathematics, the **Fibonacci numbers**, commonly denoted F_n, form a sequence, the **Fibonacci sequence**, in which each number is the sum of the two preceding ones. The sequence commonly starts from 0 and 1, although some authors start the sequence from 1 and 1 or sometimes (as did Fibonacci) from 1 and 2. Starting from 0 and 1, the first few values in the sequence are:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144.

The Fibonacci numbers were first described in Indian mathematics, as early as 200 BC in work by Pingala on enumerating possible patterns of Sanskrit poetry formed from syllables of two lengths. They are named after the Italian mathematician Leonardo of Pisa, later known as Fibonacci, who introduced the sequence to Western European mathematics in his 1202 book *Liber Abaci*.

The Fibonacci numbers may be defined by the recurrence relation

$F_0 = 0, F_1 = 1$,

and

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

for $n > 1$.

Under some older definitions, the value $F_0 = 0$ is omitted, so that the sequence starts with $F_1 = F_2 = 1$, *and the recurrence* $F_n = F_{n-1} + F_{n-2}$ *is valid for* $n > 2$.

Input:

As input, the algorithms will receive a list of numbers that will determine which Fibonacci terms to look up. The first list will be for the recursive case and as such it will be extremely small and very constrained in terms of its range of values it will be from 1 to 40 with a step of 2. The second series for the rest of the algorithms will be much more broader and it will range from 1 to 115000 with a step of 10000.

Implementations

Recursive method:

The recursive method is considered the least efficient due to its simple approach of finding the n-th term by first calculating its previous terms and then summing them up. This is done by repeatedly calling the same function multiple times, leading to an increase in memory usage and a theoretical doubling of its runtime.

Pseudocode:

```
Fibonacci(n):
 if n <= 1:
     return n
 else:
   return Fibonacci(n-1) + Fibonacci(n-2)
```

Implementation:

```python
def fibonacci_recursive(n):
    if n <= 1:
        return n
    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
```

Figure 1. Recursive algorithm

Results:

```
Test case 1:
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39]
Recursive Fibonacci
[1e-06, 2e-06, 2e-06, 4e-06, 1e-05, 2.6e-05, 6.6e-05, 0.00016, 0.000411, 0.001002, 0.003741, 0.008017, 0.020557, 0.055429, 0.150031, 0.344816, 0.884061, 2.281574, 6.059047, 15.520156]
```

Figure 2. Results for the recursive method

In figure 2 there are the inputs and the results for the recursive fibonacci method. It can be clearly seen in the later tests that the algorithm has an exponential time complexity, more precisely $T(2^n)$, the graph in figure 3 confirms this trend.

Graph:

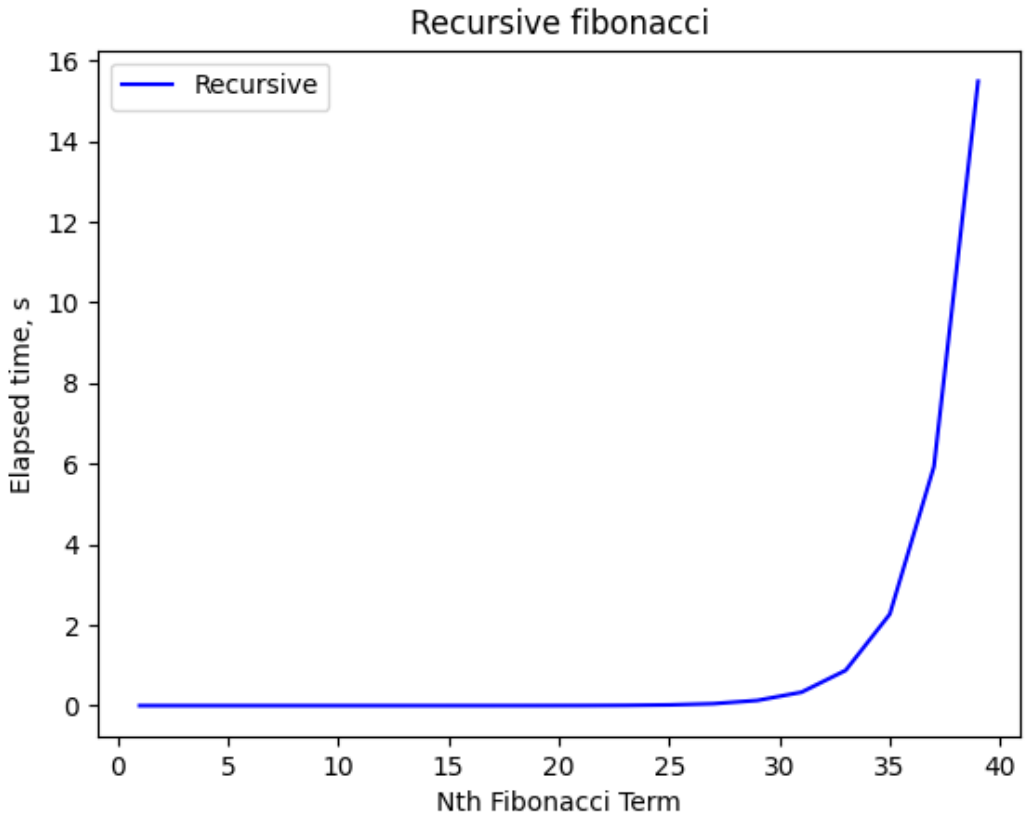


Figure 3. Graph for the recursive method

Binet Formula:

Binet's formula is an explicit formula used to find the n-th term of the Fibonacci sequence. It is so named because it was derived by mathematician Jacques Philippe Marie Binet, though it was already known by Abraham de Moivre.

Formula description:

`If` F_n `is the n-th Fibonacci number, then:`

$\varphi_1 = \frac{1+\sqrt{5}}{2}$, $\varphi_2 = \frac{1-\sqrt{5}}{2}$

$$F_n = \frac{\varphi_1^n - \varphi_2^n}{\sqrt{5}}$$

Implementation:

```
def fibonacci_binet(n):
    phi = (1 + Decimal(5).sqrt()) / 2
    psi = (1 - Decimal(5).sqrt()) / 2
    return int((pow(phi, n) - pow(psi, n)) / Decimal(5).sqrt())
```

Figure 4. Binet formula

Results:

```
Test case 2:
[1, 10001, 20001, 30001, 40001, 50001, 60001, 70001, 80001, 90001, 100001, 110001]
Binet
[5.4e-05, 0.000324, 0.00122, 0.002712, 0.004921, 0.007516, 0.010994, 0.014787, 0.019326, 0.024916, 0.030375, 0.042599]
Doubling Fibonacci
[2e-06, 0.000143, 7.5e-05, 0.000487, 0.000269, 0.000454, 0.000445, 0.000785, 0.0012, 0.00167, 0.000982, 0.001814]
Dynamic Programming Fibonacci
[2e-06, 0.011358, 0.019009, 0.038599, 0.0801, 0.099361, 0.136276, 0.182817, 0.234119, 0.275336, 0.503485, 0.522651]
Kartik's K Sequence
[2e-06, 0.001148, 0.003764, 0.008703, 0.015979, 0.021586, 0.038957, 0.047486, 0.043673, 0.054249, 0.069775, 0.081794]
Matrix Fibonacci
[3e-06, 0.023397, 0.071587, 0.172288, 0.244552, 0.364211, 0.457296, 0.615623, 0.770546, 0.956497, 1.152573, 1.365444]
Optimized Matrix
[2e-06, 0.000147, 0.000394, 0.000737, 0.001169, 0.001787, 0.002177, 0.002913, 0.00358, 0.004674, 0.005369, 0.006276]
```

Figure 5. Results for all of the methods (Binet - fourth row)

In figure 5 we can see that Binet's formula has an almost constant time but given the nth power to which the terms are being raised it is actually $O(\log n)$, the graph somewhat tells that it is extremely fast, although the algorithm is not very accurate and needs to be checked for rounding errors.

Graph:

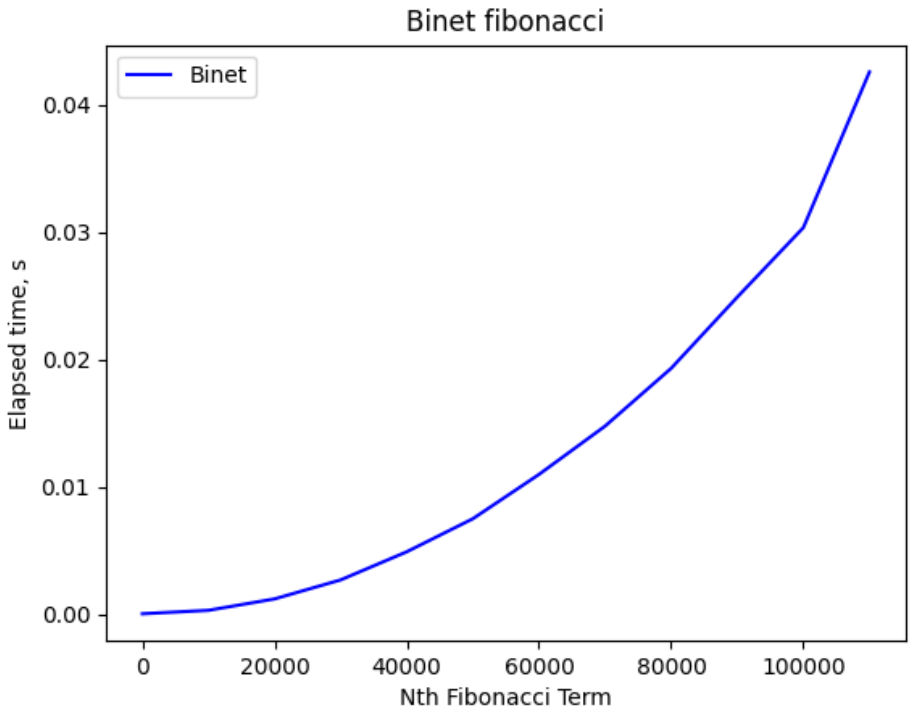


Figure 6. Binet formula graph

Doubling method:

The doubling fibonacci method is based on a formula derived from the matrix algorithm:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

Algorithm description:

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)

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

By putting n = n+1 in equation(1),
$F_mF_{n+1} + F_{m-1}F_n = F_{m+n}$

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 Fn+1 = Fn + Fn-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:

```
f = [0] * 500000


def fib(n):
    if n == 0:
        return 0
    if n == 1 or n == 2:
        f[n] = 1
        return f[n]

    if f[n]:
        return f[n]

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

    # Applying above formula: 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 7. Doubling method

Results:

```
Test case 2:
[1, 10001, 20001, 30001, 40001, 50001, 60001, 70001, 80001, 90001, 100001, 110001]
Binet
[5.4e-05, 0.000324, 0.00122, 0.002712, 0.004921, 0.007516, 0.010994, 0.014787, 0.019326, 0.024916, 0.030375, 0.042599]
Doubling Fibonacci
[2e-06, 0.000143, 7.5e-05, 0.000487, 0.000269, 0.000454, 0.000445, 0.000785, 0.0012, 0.00167, 0.000982, 0.001814]
Dynamic Programming Fibonacci
[2e-06, 0.011358, 0.019009, 0.038599, 0.0801, 0.099361, 0.136276, 0.182817, 0.234119, 0.275336, 0.503485, 0.522651]
Kartik's K Sequence
[2e-06, 0.001148, 0.003764, 0.008703, 0.015979, 0.021586, 0.038957, 0.047486, 0.043673, 0.054249, 0.069775, 0.081794]
Matrix Fibonacci
[3e-06, 0.023397, 0.071587, 0.172288, 0.244552, 0.364211, 0.457296, 0.615623, 0.770546, 0.956497, 1.152573, 1.365444]
Optimized Matrix
[2e-06, 0.000147, 0.000394, 0.000737, 0.001169, 0.001787, 0.002177, 0.002913, 0.00358, 0.004674, 0.005369, 0.006276]
```

Figure 8. Results for all of the methods (Doubling- sixth row)

In figure 8 we can see that the Doubling method is extremely fast having a time complexity of O($\log n$), the graph below gives an idea of how fast the method actually is.

Graph:

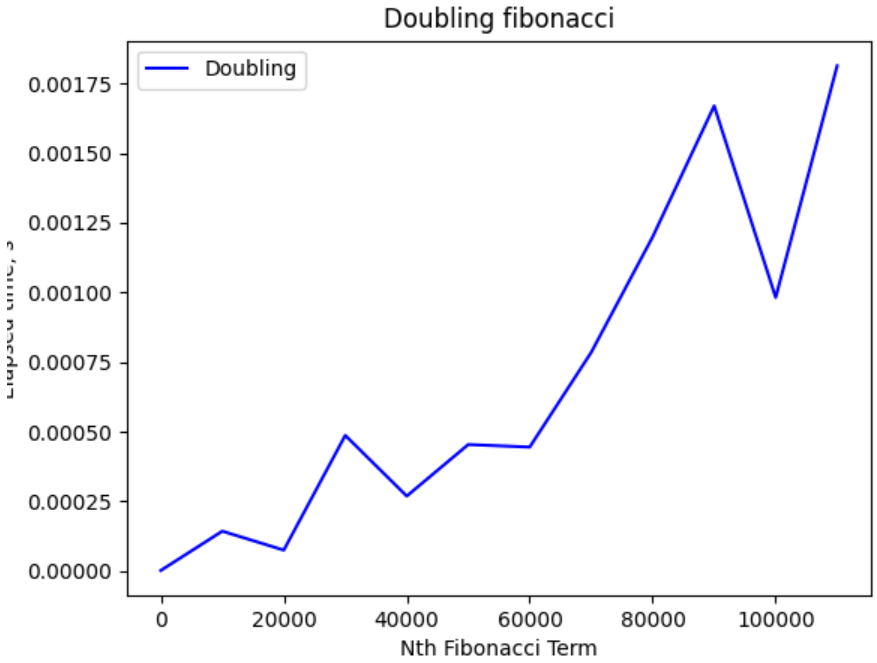


Figure 9. Doubling method graph

Dynamic Programming method:

The Dynamic Programming approach, similar to the recursive method, aims to calculate the n-th term using a straightforward approach. However, instead of repeatedly calling the function on itself, it operates from top to bottom using an array data structure that stores previously computed terms, thereby avoiding the need for redundant computation.

Pseudocode:

```
Fibonacci(n):
Array A;
A[0]<-0;
A[1]<-1;
     for i <- 2 to n - 1 do
     A[i]<-A[i-1]+A[i-2];
return A[n-1]
```

Implementation:

```python
def fibonacci_dp(n):
    f = [0, 1]

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

Figure 10. Dynamic programming implementation

Results:

```
Test case 2:
[1, 10001, 20001, 30001, 40001, 50001, 60001, 70001, 80001, 90001, 100001, 110001]
Binet
[5.4e-05, 0.000324, 0.00122, 0.002712, 0.004921, 0.007516, 0.010994, 0.014787, 0.019326, 0.024916, 0.030375, 0.042599]
Doubling Fibonacci
[2e-06, 0.000143, 7.5e-05, 0.000487, 0.000269, 0.000454, 0.000445, 0.000785, 0.0012, 0.00167, 0.000982, 0.001814]
Dynamic Programming Fibonacci
[2e-06, 0.011358, 0.019009, 0.038599, 0.0801, 0.099361, 0.136276, 0.182817, 0.234119, 0.275336, 0.503485, 0.522651]
Kartik's K Sequence
[2e-06, 0.001148, 0.003764, 0.008703, 0.015979, 0.021586, 0.038957, 0.047486, 0.043673, 0.054249, 0.069775, 0.081794]
Matrix Fibonacci
[3e-06, 0.023397, 0.071587, 0.172288, 0.244552, 0.364211, 0.457296, 0.615623, 0.770546, 0.956497, 1.152573, 1.365444]
Optimized Matrix
[2e-06, 0.000147, 0.000394, 0.000737, 0.001169, 0.001787, 0.002177, 0.002913, 0.00358, 0.004674, 0.005369, 0.006276]
```

Figure 11. Results for all of the methods (DP- eighth row)

In figure 11 we can see the results for the Dynamic programming method which is a lot faster than the recursive one, having a time complexity of O(n).

Graph:

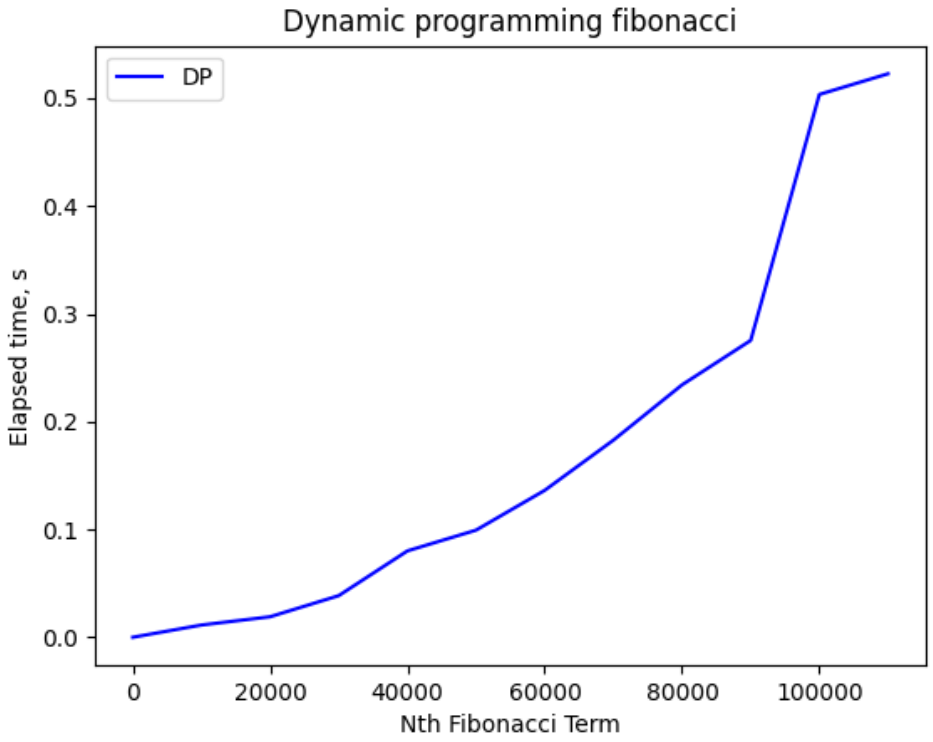


Figure 12. Dynamic programming graph

Kartik's K Sequence method:

Kartik's K sequence is a function that maps a positive integer n to another integer in a unique way.

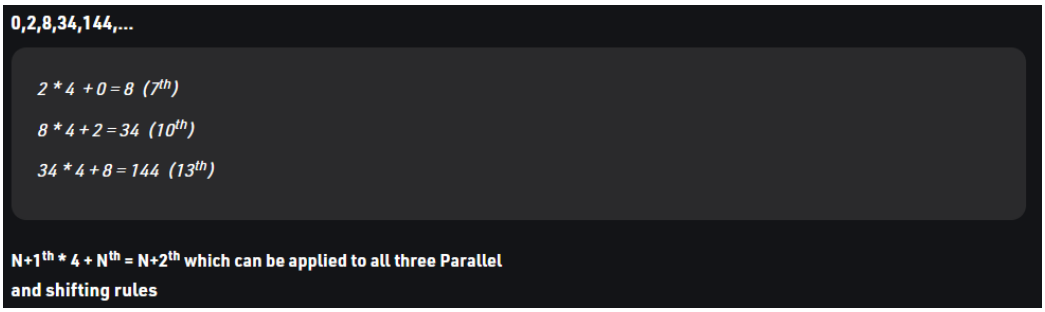
Algorithm description:

`For k = 3`

1) **0**,1,1,**2**,3,5,**8**,13,21,**34**,55,89,**144**,..... (Parallel 0 highlighted with Bold)

2) 0,**1**,1,2,**3**,5,8,**13**,21,34,**55**,89,144,..... (Parallel 1 highlighted with Bold)

3) 0,1,**1**,2,3,**5**,8,13,**21**,34,55,**89**,144,..... (Parallel 2 highlighted with Bold)



Implementation:

```python
def fibonacci_kartik(n):
    if n > 0:
        n1, n2 = 1, 1
        if n > 3:
            for _ in range((n // 3)):
                n1, n2 = n2, (n2 << 2) + n1  # << 2: multiply by 4
        if n % 3 == 0:
            return n1
        elif n % 3 == 1:
            return (n2 - n1) >> 1  # >> 1: divide by 2
        elif n % 3 == 2:
            return (n2 + n1) >> 1  # >> 1: divide by 2
    else:
        return -1
```

Figure 13. Kartik implementation

Results:

```
Test case 2:
[1, 10001, 20001, 30001, 40001, 50001, 60001, 70001, 80001, 90001, 100001, 110001]
Binet
[5.4e-05, 0.000324, 0.00122, 0.002712, 0.004921, 0.007516, 0.010994, 0.014787, 0.019326, 0.024916, 0.030375, 0.042599]
Doubling Fibonacci
[2e-06, 0.000143, 7.5e-05, 0.000487, 0.000269, 0.000454, 0.000445, 0.000785, 0.0012, 0.00167, 0.000982, 0.001814]
Dynamic Programming Fibonacci
[2e-06, 0.011358, 0.019009, 0.038599, 0.0801, 0.099361, 0.136276, 0.182817, 0.234119, 0.275336, 0.503485, 0.522651]
Kartik's K Sequence
[2e-06, 0.001148, 0.003764, 0.008703, 0.015979, 0.021586, 0.038957, 0.047486, 0.043673, 0.054249, 0.069775, 0.081794]
Matrix Fibonacci
[3e-06, 0.023397, 0.071587, 0.172288, 0.244552, 0.364211, 0.457296, 0.615623, 0.770546, 0.956497, 1.152573, 1.365444]
Optimized Matrix
[2e-06, 0.000147, 0.000394, 0.000737, 0.001169, 0.001787, 0.002177, 0.002913, 0.00358, 0.004674, 0.005369, 0.006276]
```

Figure 14. Results for all of the methods (Kartik- tenth row)

In figure 14 we can see the results for the Kartik method which is also pretty fast having a time complexity that is in between O($\log n$) and O(n) or (n/3).

Graph:

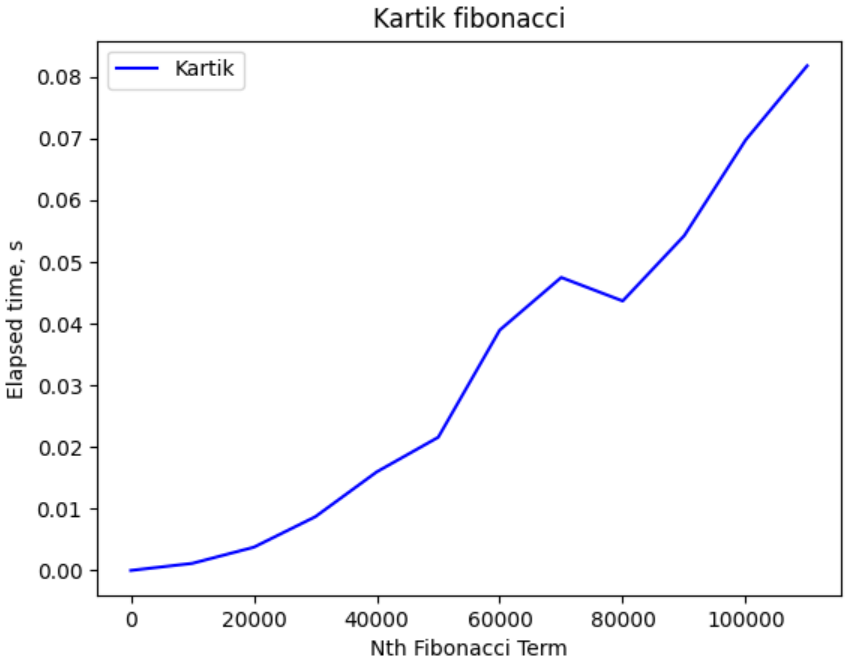


Figure 15. Kartik method graph

Matrix method:

Algorithm description:

The matrix algorithm is based on a formula that connects the fibonacci terms to a matrix raised to the nth power which is deduced through matrix multiplication properties.

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

Pseudocode:

```
Fibonacci(n):
F<- []
vec <- [[0], [1]]
Matrix <- [[0, 1],[1, 1]]
F <-power(Matrix, n)
```

```
F <- F * vec
Return F[0][0]
```

Implementation:

```python
def fibonacci_matrix(n):
    if n == 0:
        return 0
    else:
        A = [[1, 1], [1, 0]]
        power(A, n - 1)
        return A[0][0]


def power(A, n):
    M = [[1, 1],
         [1, 0]]

    for i in range(2, n + 1):
        matrixMultiplication(A, M)


def matrixMultiplication(A, B):
    a = A[0][0] * B[0][0] + A[0][1] * B[1][0]
    b = A[0][0] * B[0][1] + A[0][1] * B[1][1]
    c = A[1][0] * B[0][0] + A[1][1] * B[1][0]
    d = A[1][0] * B[0][1] + A[1][1] * B[1][1]

    A[0][0] = a
    A[0][1] = b
    A[1][0] = c
    A[1][1] = d
```

Figure 16. Matrix method implementation

Results:

```
Test case 2:
[1, 10001, 20001, 30001, 40001, 50001, 60001, 70001, 80001, 90001, 100001, 110001]
Binet
[5.4e-05, 0.000324, 0.00122, 0.002712, 0.004921, 0.007516, 0.010994, 0.014787, 0.019326, 0.024916, 0.030375, 0.042599]
Doubling Fibonacci
[2e-06, 0.000143, 7.5e-05, 0.000487, 0.000269, 0.000454, 0.000445, 0.000785, 0.0012, 0.00167, 0.000982, 0.001814]
Dynamic Programming Fibonacci
[2e-06, 0.011358, 0.019009, 0.038599, 0.0801, 0.099361, 0.136276, 0.182817, 0.234119, 0.275336, 0.503485, 0.522651]
Kartik's K Sequence
[2e-06, 0.001148, 0.003764, 0.008703, 0.015979, 0.021586, 0.038957, 0.047486, 0.043673, 0.054249, 0.069775, 0.081794]
Matrix Fibonacci
[3e-06, 0.023397, 0.071587, 0.172288, 0.244552, 0.364211, 0.457296, 0.615623, 0.770546, 0.956497, 1.152573, 1.365444]
Optimized Matrix
[2e-06, 0.000147, 0.000394, 0.000737, 0.001169, 0.001787, 0.002177, 0.002913, 0.00358, 0.004674, 0.005369, 0.006276]
```

Figure 17. Results for all of the methods (Matrix- twelfth row)

In figure 17 we can see the results for the Matrix method which is comparable to the dynamic programming one, also having a time complexity of O(n).

Graph:

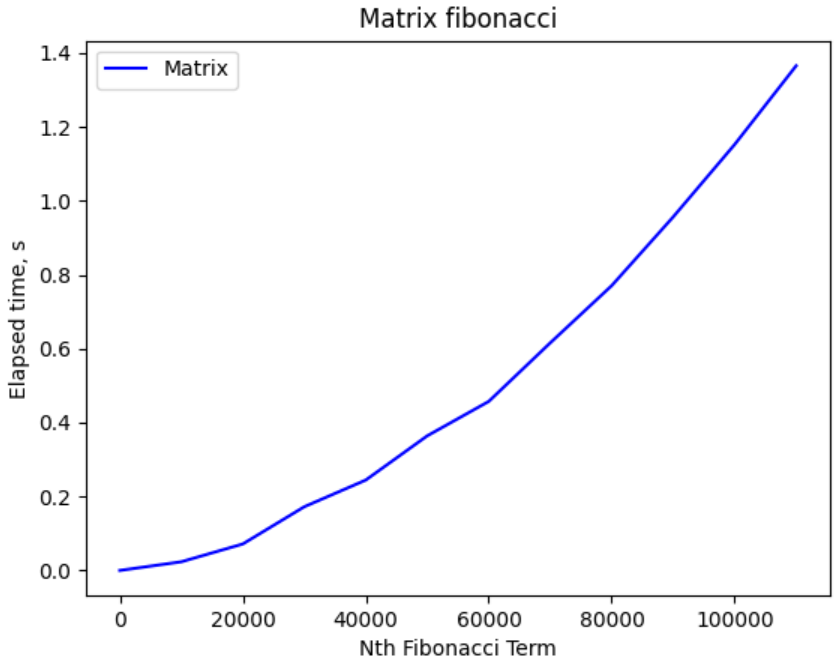


Figure 18. Matrix method graph

Optimized matrix method:

The method follows the same formula as in the matrix algorithm, only that it improves on the power function making it faster at computing nth fibonacci terms

Algorithm description:

The optimized matrix algorithm is based on the same formula as the normal matrix algorithm:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

Implementation:

```
def fibonacci_matrix_op(n):
    if n == 0:
        return 0
    else:
        A = [[1, 1], [1, 0]]
        power(A, n - 1)
        return A[0][0]


# Optimized power method

def power(A, n):
    if n == 0 or n == 1:
        return

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

    power(A, n // 2)
    matrixMultiplication(A, A)

    if n % 2 != 0:
        matrixMultiplication(A, M)


def matrixMultiplication(A, B):
    a = A[0][0] * B[0][0] + A[0][1] * B[1][0]
    b = A[0][0] * B[0][1] + A[0][1] * B[1][1]
    c = A[1][0] * B[0][0] + A[1][1] * B[1][0]
    d = A[1][0] * B[0][1] + A[1][1] * B[1][1]

    A[0][0] = a
    A[0][1] = b
    A[1][0] = c
    A[1][1] = d
```

Figure 19. Optimized matrix method

Results:

```
Test case 2:
[1, 10001, 20001, 30001, 40001, 50001, 60001, 70001, 80001, 90001, 100001, 110001]
Binet
[5.4e-05, 0.000324, 0.00122, 0.002712, 0.004921, 0.007516, 0.010994, 0.014787, 0.019326, 0.024916, 0.030375, 0.042599]
Doubling Fibonacci
[2e-06, 0.000143, 7.5e-05, 0.000487, 0.000269, 0.000454, 0.000445, 0.000785, 0.0012, 0.00167, 0.000982, 0.001814]
Dynamic Programming Fibonacci
[2e-06, 0.011358, 0.019009, 0.038599, 0.0801, 0.099361, 0.136276, 0.182817, 0.234119, 0.275336, 0.503485, 0.522651]
Kartik's K Sequence
[2e-06, 0.001148, 0.003764, 0.008703, 0.015979, 0.021586, 0.038957, 0.047486, 0.043673, 0.054249, 0.069775, 0.081794]
Matrix Fibonacci
[3e-06, 0.023397, 0.071587, 0.172288, 0.244552, 0.364211, 0.457296, 0.615623, 0.770546, 0.956497, 1.152573, 1.365444]
Optimized Matrix
[2e-06, 0.000147, 0.000394, 0.000737, 0.001169, 0.001787, 0.002177, 0.002913, 0.00358, 0.004674, 0.005369, 0.006276]
```

Figure 20. Results for all of the methods (Optimized matrix- fourteenth row)

In figure 20 we can see the results for the Optimized Matrix method which becomes a lot faster than the normal matrix method, having a time complexity of $O(\log n)$

Graph:

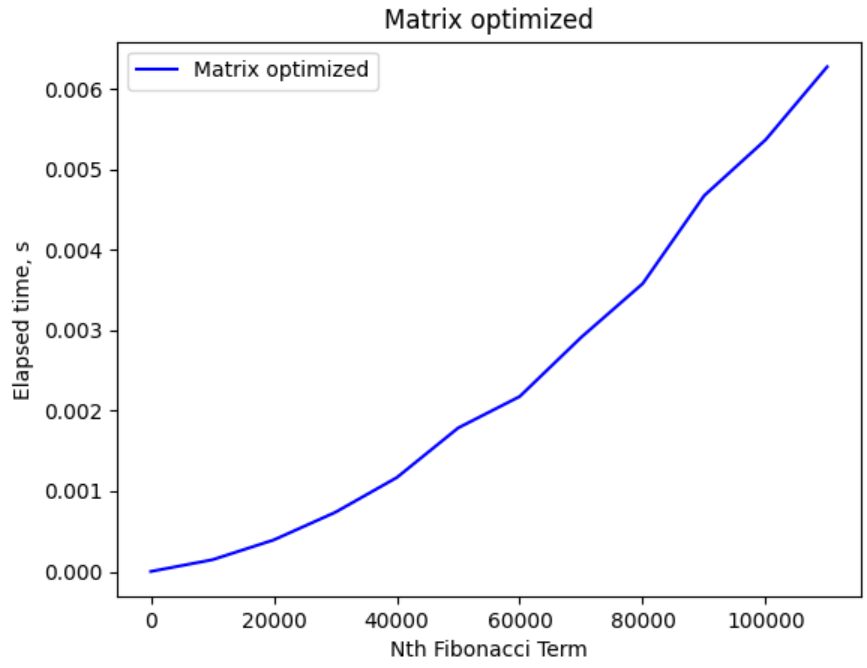


Figure 21. Optimized matrix graph

Algorithms Comparison

The recursive method was excluded from this comparison as it is clearly a lot slower and would only affect the comparison graph

Results:

```
Test case 2:
[1, 10001, 20001, 30001, 40001, 50001, 60001, 70001, 80001, 90001, 100001, 110001]
Binet
[5.4e-05, 0.000324, 0.00122, 0.002712, 0.004921, 0.007516, 0.010994, 0.014787, 0.019326, 0.024916, 0.030375, 0.042599]
Doubling Fibonacci
[2e-06, 0.000143, 7.5e-05, 0.000487, 0.000269, 0.000454, 0.000445, 0.000785, 0.0012, 0.00167, 0.000982, 0.001814]
Dynamic Programming Fibonacci
[2e-06, 0.011358, 0.019009, 0.038599, 0.0801, 0.099361, 0.136276, 0.182817, 0.234119, 0.275336, 0.503485, 0.522651]
Kartik's K Sequence
[2e-06, 0.001148, 0.003764, 0.008703, 0.015979, 0.021586, 0.038957, 0.047486, 0.043673, 0.054249, 0.069775, 0.081794]
Matrix Fibonacci
[3e-06, 0.023397, 0.071587, 0.172288, 0.244552, 0.364211, 0.457296, 0.615623, 0.770546, 0.956497, 1.152573, 1.365444]
Optimized Matrix
[2e-06, 0.000147, 0.000394, 0.000737, 0.001169, 0.001787, 0.002177, 0.002913, 0.00358, 0.004674, 0.005369, 0.006276]
```

Figure 22. Results for all of the methods

Graphs:

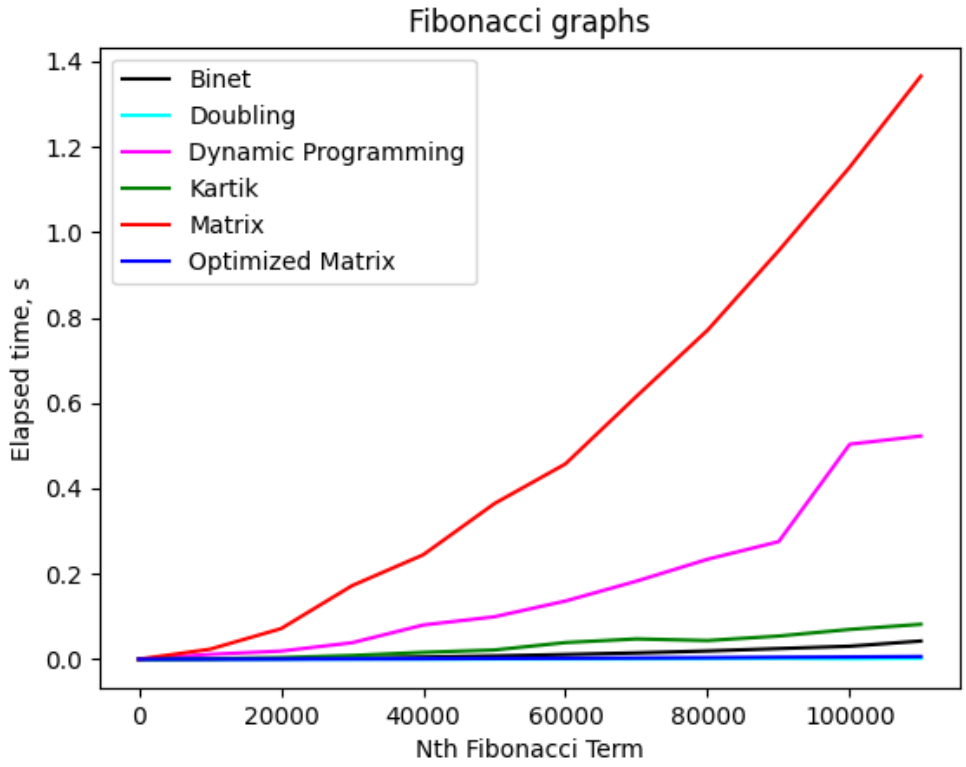


Figure 23. Results for all of the methods' graphs

Conclusion

Through empirical analysis, in this laboratory work were tested 7 methods for obtaining the Nth fibonacci term. Both their complexity and accuracy were taken into consideration when deciding their efficiency. The test cases were chosen as to best reflect the time complexity of each algorithm and also show each's weakness and possibilities for optimization.

The recursive case was the easiest and simplest to write and understand but it's drawback was the exponential time complexity, and after the 50 fibonacci term it slows down to levels that it is not feasible to compute it using this algorithm

The Binet formula was one of the fastest ones and also easiest when it came down to writing the code. The only drawback is that it is not accurate when computing large fibonacci terms and needs to be checked for errors that may appear when rounding the golden ratio.

The Dynamic Programming method was one that had properties from both worlds, that is faster than the recursive case and also extremely easy to write and understand. Furthermore, having enough space left for improvements and in comparison to the binet formula it was accurate when it came to the results.

The Doubling method given by its graph and in comparison to the other algorithms proved to be one of the fastest ones, altough its formula is somewhat harder to derive the implementation itself is not very complicated.

Kartik's K sequence method turned out to be one of the most difficult algorithms in terms of understanding and writing it, but its efficiency was undoubtedly one of best ones. Its basic concept was that of mapping a positive integer to another integer in a unique way.

The matrix method and its optimized form were also a lot faster than the recursive case and not that hard to understand as the base formula could easily be derived through the use of matrix multiplication and exponentiation’ properties. The optimized method was actually one of the most efficient ones, given the improvements done to the matrix multiplication function.