**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**REPORT**

Laboratory Work nr.1

*at Algorithms Analysis*

Elaborated by:

st. gr. FAF-211 Hariton Dan

Verified by:

asist. univ. Fiștic Cristofor

Chişinău - 2023

**ALGORITHM ANALYSIS**

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 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:**

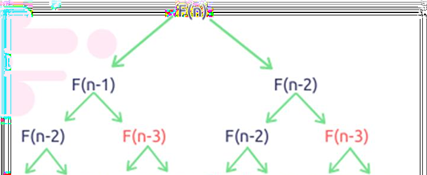
As input, each algorithm will receive two series of numbers that will contain the order of the Fibonacci terms being looked up. The first series will have a more limited scope, (10,20,30,40,50), to accommodate the recursive method, while the rest of the series will have a bigger scope to be able to compare the other algorithms between themselves (200,400,600,800,1000).

**IMPLEMENTATION**

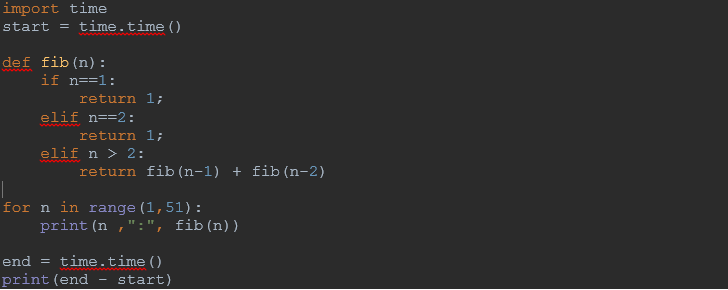
All six algorithms will be implemented in their naïve form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending o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

**Recursive Method:**

The recursive method, also considered the most inefficient method, follows a straightforward approach of computing the n-th term by computing it’s 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 it’s execution time.

*Figure 1 Fibonacci recursion*

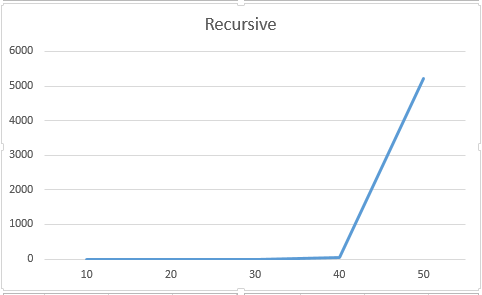
*Implementation:*

*Figure 2 Fibonacci recursion in Python*

*Results:*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Recursive | Dynamic | Iteration | Matrix | Kmethod | Binet |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 |
| 20 | 0.003024 | 0 | 0 | 0.000995 | 0 | 0 |
| 30 | 0.347015 | 0 | 0 | 0.001016 | 0 | 0 |
| 40 | 41.43546 | 0 | 0 | 0.001992 | 0 | 0 |
| 50 | 5209.482 | 0 | 0 | 0.001015 | 0 | 0 |

*Figure 3 Results for the first set of inputs*

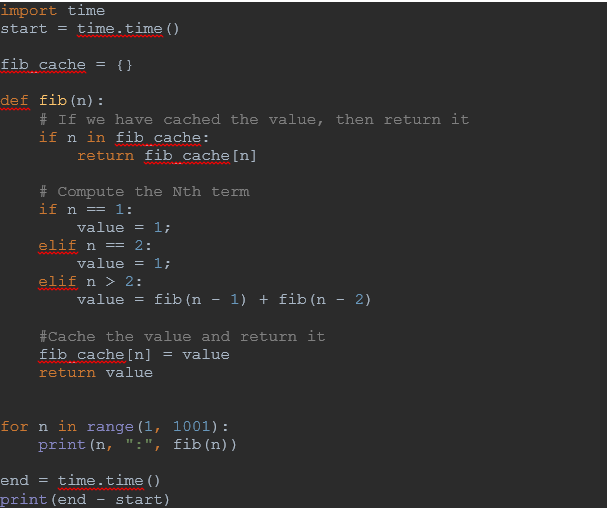
In Figure 3 is represented the table of results for the first set of inputs. The highest line (the name of the columns) denotes the Fibonacci n-th term for which the functions were run. The other methods are not growing for the 50-th term as they are close to 0

*Figure 4 Graph of Recursive Fibonacci Function*

Not only that, but also in the graph in Figure 4 that shows the growth of the time needed for the operations, we may easily see the spike in time complexity that happens after the 40nd term, leading us to deduce that the Time Complexity is exponential. T(2𝑛).

## Dynamic Programming Method:

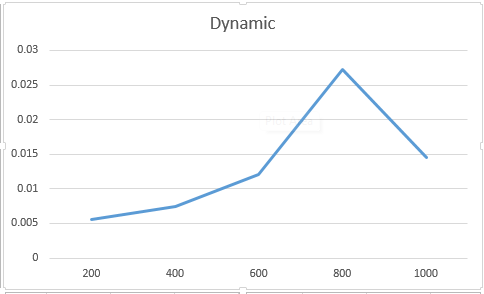
## The Dynamic Programming method, similar to the recursive method, takes the straightforward approach of calculating the n-th term. We can avoid the repeated work done in method 1 by storing the Fibonacci numbers calculated so far.

*Implementation:*

*Figure 5 Fibonacci DP in Python*

*Results:*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Dynamic | Iteration | Matrix | Kmethod | Binet |
| 200 | 0.005525 | 0.005013 | 0.017401 | 0.002534 | 0.005974 |
| 400 | 0.00743 | 0.008299 | 0.069026 | 0.008552 | 0.006988 |
| 600 | 0.012041 | 0.018805 | 0.162722 | 0.009456 | 0.006994 |
| 800 | 0.027264 | 0.029144 | 0.287054 | 0.014147 | 0.012522 |
| 1000 | 0.014523 | 0.043394 | 0.496808 | 0.014083 | 0.02008 |

*Figure 6 Results for the second set of inputs*

*Figure 7 Graph of DP Fibonacci Function*

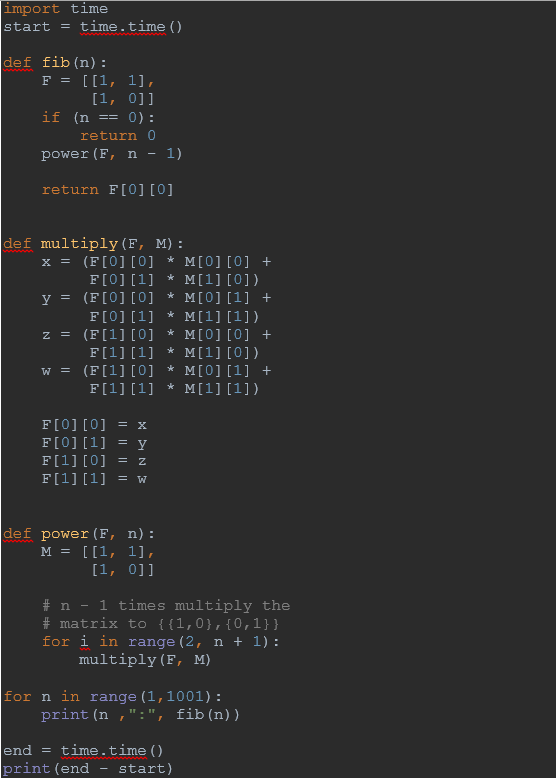
With the Dynamic Programming Method showing excellent results with a time complexity denoted in a corresponding graph of T(n),

## Matrix Power Method:

## This is another O(n) that 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. The matrix representation gives the following closed expression for the Fibonacci numbers:

## 

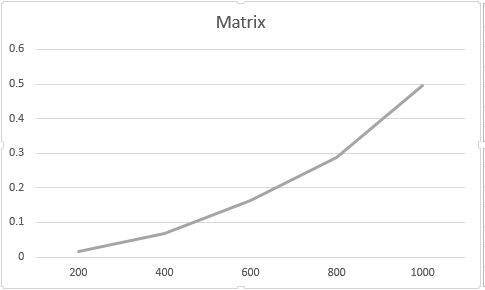
*Implementation:*



*Figure 8 Fibonacci Matrix in Python*

*Results:*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Dynamic | Iteration | Matrix | Kmethod | Binet |
| 200 | 0.005525 | 0.005013 | 0.017401 | 0.002534 | 0.005974 |
| 400 | 0.00743 | 0.008299 | 0.069026 | 0.008552 | 0.006988 |
| 600 | 0.012041 | 0.018805 | 0.162722 | 0.009456 | 0.006994 |
| 800 | 0.027264 | 0.029144 | 0.287054 | 0.014147 | 0.012522 |
| 1000 | 0.014523 | 0.043394 | 0.496808 | 0.014083 | 0.02008 |

*Figure 9 Results for the second set of inputs*

*Figure 10 Graph of Matrix Fibonacci Function*

Programming one, still performing pretty well, with the form f the graph indicating a pretty solid T(n) time complexity.

## Iterative Method:

## The classic iterative method implemented in Python for Fibonacci numbers

*Implementation:*

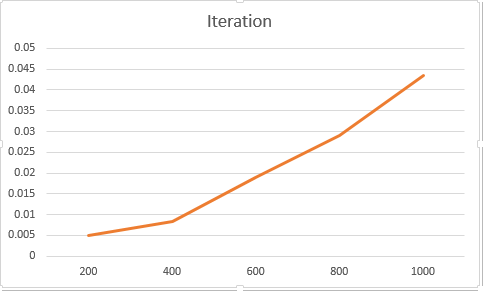
import time  
start = time.time()  
  
def fib(n):  
 a,b = 0,1  
 for i in range(n):  
 a,b = b,a+b  
 return a  
  
for n in range(1,1001):  
 print(n ,":", fib(n))  
  
end = time.time()  
print(end - start)

*Figure 11 Fibonacci Iterative in Python*

*Results:*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Dynamic | Iteration | Matrix | Kmethod | Binet |
| 200 | 0.005525 | 0.005013 | 0.017401 | 0.002534 | 0.005974 |
| 400 | 0.00743 | 0.008299 | 0.069026 | 0.008552 | 0.006988 |
| 600 | 0.012041 | 0.018805 | 0.162722 | 0.009456 | 0.006994 |
| 800 | 0.027264 | 0.029144 | 0.287054 | 0.014147 | 0.012522 |
| 1000 | 0.014523 | 0.043394 | 0.496808 | 0.014083 | 0.02008 |

*Figure 12 Results for the second set of inputs*

**

*Figure 13 Graph of Iterative Fibonacci Function*

## KMethod:

Below is one more interesting recurrence formula that can be used to find n’th Fibonacci Number in O(Log n) time.

**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)**

*Implementation:*

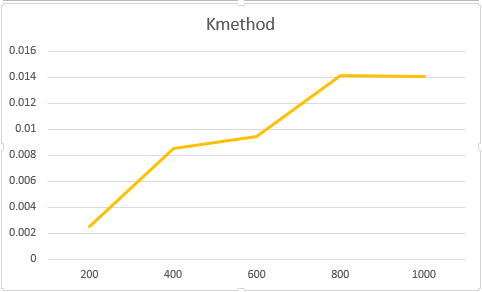
import time  
start = time.time()  
  
  
MAX = 1001  
  
# Create an array for memoization  
f = [0] \* MAX  
  
  
  
# Returns n'th fibonacci number using table f[]  
def fib(n) :  
 # Base cases  
 if (n == 0) :  
 return 0  
 if (n == 1 or n == 2) :  
 f[n] = 1  
 return (f[n])  
  
 # If fib(n) 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]  
  
  
# Driver code  
for n in range(1,1001):  
 print(n ,":", fib(n))  
  
  
  
  
end = time.time()  
print(end - start)

*Figure 14 Fibonacci KMethod in Python*

*Results:*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Dynamic | Iteration | Matrix | Kmethod | Binet |
| 200 | 0.005525 | 0.005013 | 0.017401 | 0.002534 | 0.005974 |
| 400 | 0.00743 | 0.008299 | 0.069026 | 0.008552 | 0.006988 |
| 600 | 0.012041 | 0.018805 | 0.162722 | 0.009456 | 0.006994 |
| 800 | 0.027264 | 0.029144 | 0.287054 | 0.014147 | 0.012522 |
| 1000 | 0.014523 | 0.043394 | 0.496808 | 0.014083 | 0.02008 |

*Figure 15 Results for the second set of inputs*

**

*Figure 16 Graph of Kmethod Fibonacci Function*

## Binet Method:

The Binet Formula Method is another unconventional way of calculating the n-th term of the Fibonacci series, as it operates using the Golden Ratio formula, or phi. However, due to its nature of requiring the usage of decimal numbers, at some point, the rounding error of python that accumulates, begins affecting the results significantly. The observation of error starting with around 70-th number making it unusable in practice, despite its speed.

*Implementation:*

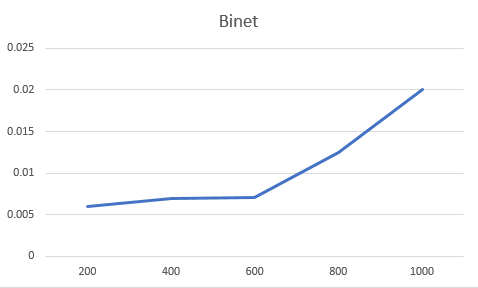
import time  
start = time.time()  
  
  
# fibonacci Number  
import math  
  
  
def fib(n):  
 phi = (1 + math.sqrt(5)) / 2  
  
 return round(pow(phi, n) / math.sqrt(5))  
  
  
# Driver code  
for n in range(1,1001):  
 print(n ,":", fib(n))  
  
  
end = time.time()  
print(end - start)

*Figure 17 Fibonacci Binet in Python*

*Results:*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Dynamic | Iteration | Matrix | Kmethod | Binet |
| 200 | 0.005525 | 0.005013 | 0.017401 | 0.002534 | 0.005974 |
| 400 | 0.00743 | 0.008299 | 0.069026 | 0.008552 | 0.006988 |
| 600 | 0.012041 | 0.018805 | 0.162722 | 0.009456 | 0.006994 |
| 800 | 0.027264 | 0.029144 | 0.287054 | 0.014147 | 0.012522 |
| 1000 | 0.014523 | 0.043394 | 0.496808 | 0.014083 | 0.02008 |

*Figure 18 Results for the second set of inputs*



*Figure 19 Graph of Iterative Fibonacci Function*

The Binet Formula Function is not accurate enough to be considered within the analysed limits and is recommended to be used for Fibonacci terms up to 72. At least in its naïve form in python, as further modification and change of language may extend its usability further.

**Conclusion:**

The proposed work's aim, which consists of the time-based and theoretical analysis of algorithms for determining the n-th Fibonacci number, was successfully accomplished. Six different algorithms were used: recursive, iterative, dynamic, matrix, k-method, binet to determine their temporal complexity and to highlight the most efficient algorithm that will display the desired result using the least time that passed by. The comparison of the algorithms was performed using a table where we collected time based data on the iterations received during the work and the graphs constructed in Excel, which show the noticeable difference between the complexities of the algorithms. From this we observe that the recursive method is the most inefficient, as it recalculates the same values multiple times, while the formula method gave us much more favorable results. The rest of them look pretty even. But at a closer inspection we can see that the Matrix algorithm is the second most inefficient method. Thus is better than the recursive algorithm by a long distance is still less efficient then the other ones. The other 4 algorithms are efficient when computing numbers up to 1000, the differences are small, and different ones can be used in various situations depending on the number of terms that we want to compute

**Link to GitHub**: <https://github.com/haritondan/AA-Labs>