Ministerul Educaţiei și 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-213 Konjevic Alexandra

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.

Introduction:

The Fibonacci sequence was first introduced to the western world by Leonardo of Pisa, an Italian mathematician, in his book "Liber Abaci" published in 1202. It is named after him although it is not known if he actually discovered it.

The sequence starts with 0 and 1, and each subsequent number in the sequence is the sum of the previous two numbers. The sequence can be defined mathematically as: $F(n) = F(n-1) + F(n-2)$ where $F(0) = 0$ and $F(1) = 1$.

There are various techniques to find the n-th Fibonacci number in programming. A simple approach is to use recursion, where the function calls itself to find the n-1 and n-2 Fibonacci numbers and then adds them to find the n-th number. Another approach is to use dynamic programming and store the previously calculated values in an array, this avoids the repeated calculations and speeds up the program. Additionally, there is a closed-form expression called Binet's formula that can directly calculate the n-th Fibonacci number without the need for recursion or iteration.

In conclusion, the Fibonacci sequence is an important concept in mathematics with a rich history and various programming techniques for its calculation.

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

1. Recursive Method:

Algorithm Description:

This approach is an easy direct recursive implementation of the mathematical recurrence relation.

Algorithm complexity: O(n^2) - exponential, as every function call two other functions.

Implementation:

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

Figure 1. Fibonacci recursion in Python

```
PS C:\Users\PC\Desktop\semester4\AA\lab1\implementations> python recursion.py
n:          5      7      10      12      15      17      20      22      25      27      30      32      35      37      40      42      45
sec:      0.0    0.0    0.0     0.0     0.0     0.0     0.00.004003    0.02    0.04    0.188   0.488   2.07    5.362   19.34   50.48   212.9
```

Figure 2. Results for first set of inputs

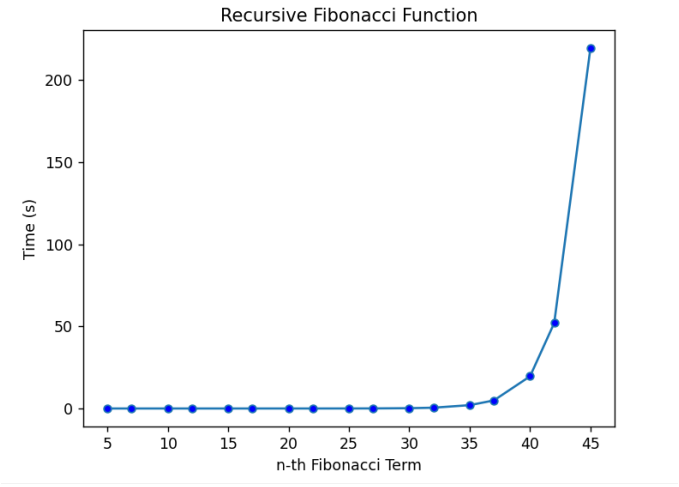

Figure 3. Graph of Recursive Fibonacci Function

We can easily see the time complexity growth that occurs after the 42nd term on the graph in Figure 4

that illustrates the rise of the time required for operations, which allows us to conclude that the time complexity is exponential.

2. Dynamic programming

Dynamic programming is an optimization technique used to solve problems by breaking them down into smaller subproblems and storing their solutions in an array (or cache) for future use.

```python
def fibonacci_dp(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib = [0] * (n + 1)
        fib[0] = 0
        fib[1] = 1
        for i in range(2, n + 1):
            fib[i] = fib[i - 1] + fib[i - 2]
        return fib[n]
```

Figure 4. Dynamic programming algorithm

n:	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.003993	0.004002	0.0	0.003999	0.012	0.012	0.016	0.028
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.004	0.0	0.0	0.0	0.003998	0.004002	0.004	0.012	0.01599	0.024
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.003999	0.0	0.0	0.0	0.004002	0.004	0.004001	0.008	0.012	0.016

Figure 5. Fibonacci DP set of results

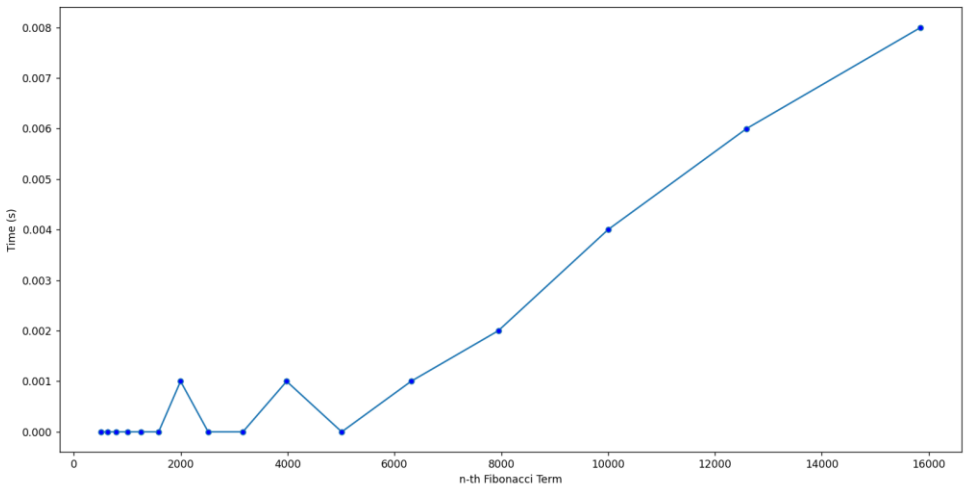

Figure 6. Fibonacci DP graph

In this implementation, the fibonacci terms are calculated and stored in an array fib. The values of fib[0] and fib[1] are set to 0 and 1 respectively. The for loop then calculates the value of each subsequent term in the sequence by adding the previous two values stored in fib.

Algorithm complexity: O(n)

3. Matrices algorithm

The fibonacci sequence can also be found using matrix exponentiation. This approach is more efficient than the traditional dynamic programming approach for large values of n.

```python
def multiplyMatrix(A, B):
    C = [[0, 0], [0, 0]]
    for i in range(2):
        for j in range(2):
            for k in range(2):
                C[i][j] += A[i][k] * B[k][j]
    return C

def power(A, n):
    if n == 0 or n == 1:
        return A
    else:
        B = power(A, n // 2)
        C = multiplyMatrix(B, B)
        if n % 2 == 0:
            return C
        else:
            return multiplyMatrix(A, C)

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

Figure 7. Matrix implementation

In this implementation, the matrix A represents the base case for the fibonacci sequence. The function power raises the matrix A to the power of n-1 and the function multiply is used to multiply two matrices. The result of power(A, n-1) is stored in the matrix F, and the value of the n-th fibonacci term is returned as F[0][0].

n:	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.0	0.0	0.0	0.0	0.001005
sec:	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	0.0009959	0.0	0.0
sec:	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	0.0009985	0.0	0.0

Figure 8.Matrix set of results

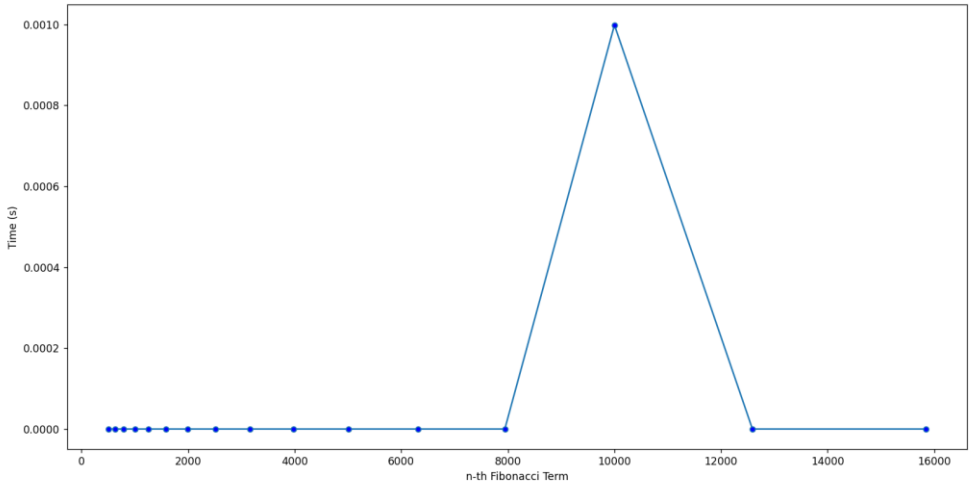

Figure 9. Fibonacci Matrix graph

The time complexity of the algorithm for finding the n-th term in the fibonacci sequence using matrices is O(log n). This is because matrix exponentiation can be performed in O(log n) time by using a divide and conquer approach, as demonstrated in the power function. The multiply function takes O(2^2) = O(1) time to multiply two 2x2 matrices, so the total time complexity of the algorithm is O(log n). This is a significant improvement over the time complexity of the traditional dynamic programming approach.

Algorithm complexity: O(log n).

4. Binet Formula Method

The Binet formula is an analytical formula that can be used to directly calculate the n-th term in the fibonacci sequence. The formula is given as follows:

Phi = (1 + sqrt(5))

Phi1 = (1 - sqrt(5))

F(n) = (phi^n - (1-phi1)^n) / (2^sqrt(5))

In this formula, the value of phi is raised to the power of n and the value of (1-phi) is also raised to the power of n. The result of these two calculations is divided by the square root of 5, and the resulting value is the n-th Fibonacci number. The formula can be implemented in any programming language that supports floating-point arithmetic and matrix exponentiation.

Algorithm complexity: O(1).

```python
def fibonacci_binet(n):
    ctx = Context(prec=60, rounding=ROUND_HALF_EVEN)
    phi = Decimal((1 + Decimal(math.sqrt(5))))
    phi1 = Decimal((1 - Decimal(math.sqrt(5))))
    return int((ctx.power(phi, Decimal(n)) - ctx.power(phi1, Decimal(n)))/(2**n * Decimal(5**(1/2))))
```

Figure 10. Binet Formula implementation

n:	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001015	0.0	0.0	0.0	0.0	0.0009866	0.0
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0009997	0.0	0.0	0.0	0.0	0.0	0.0	0.0009997
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0009999	0.0	0.0	0.0	0.0	0.0009997

Figure 11. Binet Formula set of results

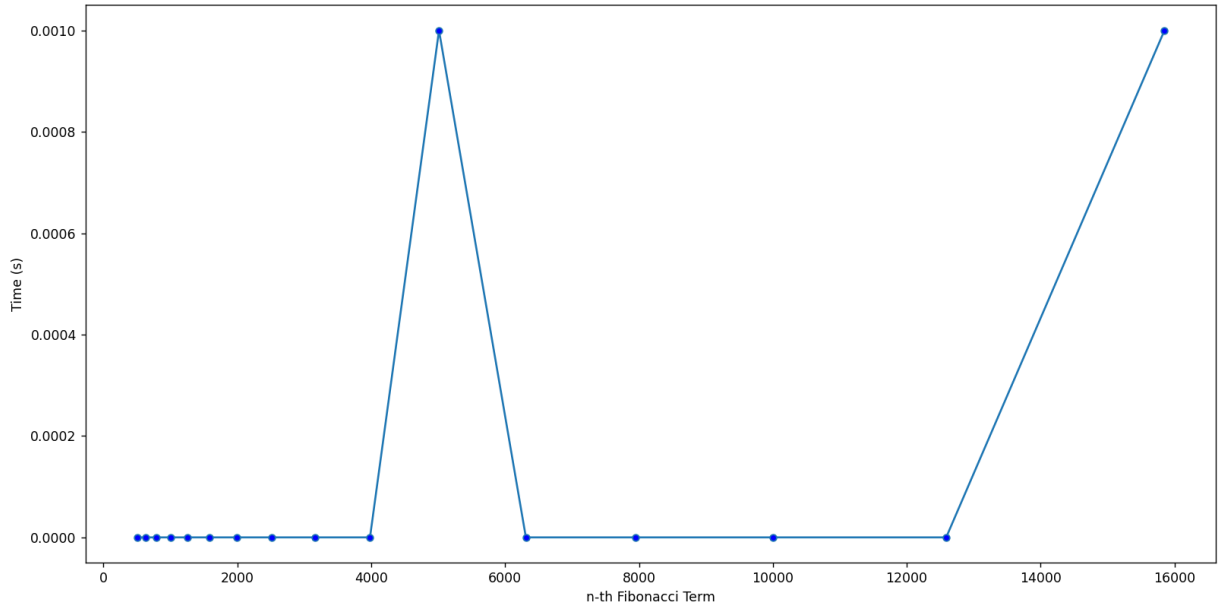

Figure 12. Binet formula graph

5. Iterative method

```python
def fibonacci_iterative(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        a, b = 0, 1
        for i in range(2, n+1):
            c = a + b
            a = b
            b = c
        return b
```

Figure 13. Iterative method implementation

n:	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001001	0.0	0.0	0.0009999	0.0009997	0.0009995	0.001001	0.003005
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000998	0.0	0.0	0.0	0.0009961	0.001	0.001	0.002002	0.001998
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.001001	0.0	0.0	0.0	0.0	0.001001	0.000998	0.001	0.002	0.002

Figure 14. Iterative method set of results

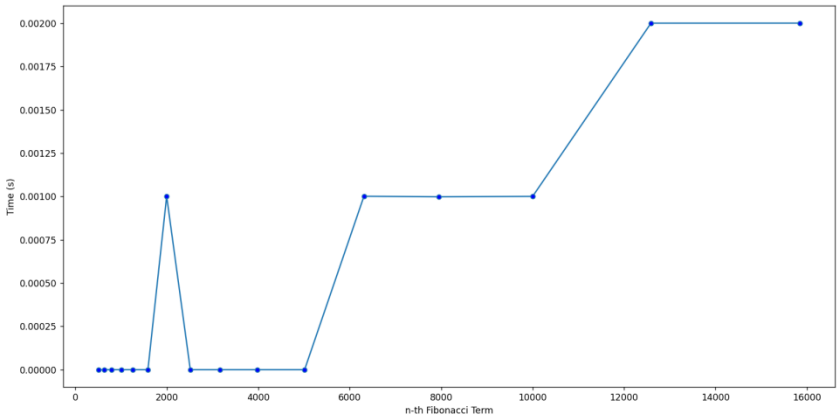

Figure 15. Iterative method graph

The iterative method for finding the n-th Fibonacci number involves using a loop to calculate each number in the sequence, one by one. The loop starts at the second Fibonacci number (since the first two numbers are already known), and continues until the n-th number is reached.

The idea behind the iterative method is to keep track of the two previous Fibonacci numbers and use them to calculate the next number in the sequence. In each iteration of the loop, the value of the next Fibonacci number is calculated by adding the two previous numbers, and then updating the values of the previous numbers for the next iteration.

Algorithm complexity: O(n).

6. Memoization

The optimized recursion algorithm for finding the n-th Fibonacci number is called "Memoization" or "Top-Down Dynamic Programming". In this approach, the solution to each subproblem is stored in an array or cache so that it can be reused later, rather than being recomputed each time. This reduces the time complexity from exponential to linear, resulting in a much faster solution. The basic idea is to store the results of expensive function calls and return the cached result when the same inputs occur again. In the case of the Fibonacci sequence, each number can be calculated by summing the previous two numbers, so the cache is used to store the intermediate results of the Fibonacci sequence and retrieve them when needed, rather than recalculating them each time. **Algorithm complexity:** O(n).

```python
def fib_memo(n, cache={0:0, 1:1}):
    if n in cache:
        return cache[n]
    cache[n] = fib_memo(n - 1, cache) + fib_memo(n - 2, cache)
    return cache[n]
```

Figure 16. Memoization implementation

n:	501	631	794	1000	1259	1585	1995	2512	3162
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
sec:	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 17. Memoization set of results

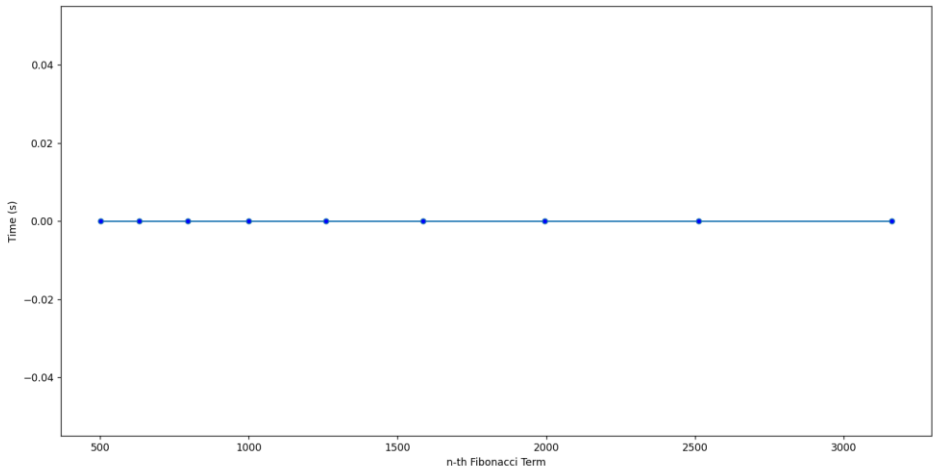

Figure 18. Memoization method graph

7. Tail-Call Optimization

This approach uses a recursive function, but it is optimized to prevent stack overflow by using tail-call optimization. This optimization eliminates the need for a new stack frame for each recursive call, reducing the memory overhead of the recursive approach.

Algorithm complexity: O(n).

```python
def fibonacci_tail_call(n, a=0, b=1):
    if n == 0:
        return a
    elif n == 1:
        return b
    else:
        return fibonacci_tail_call(n-1, b, a + b)
```

Figure 19. Tail-Call method implementation

n:	5	7	10	12	15	17	20	22	25	27	30	32	35	37	40	42	45
sec:	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	0.0	0.0
sec:	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	0.0	0.0
sec:	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	0.0	0.0

Figure 20. Tail-Call method set of results

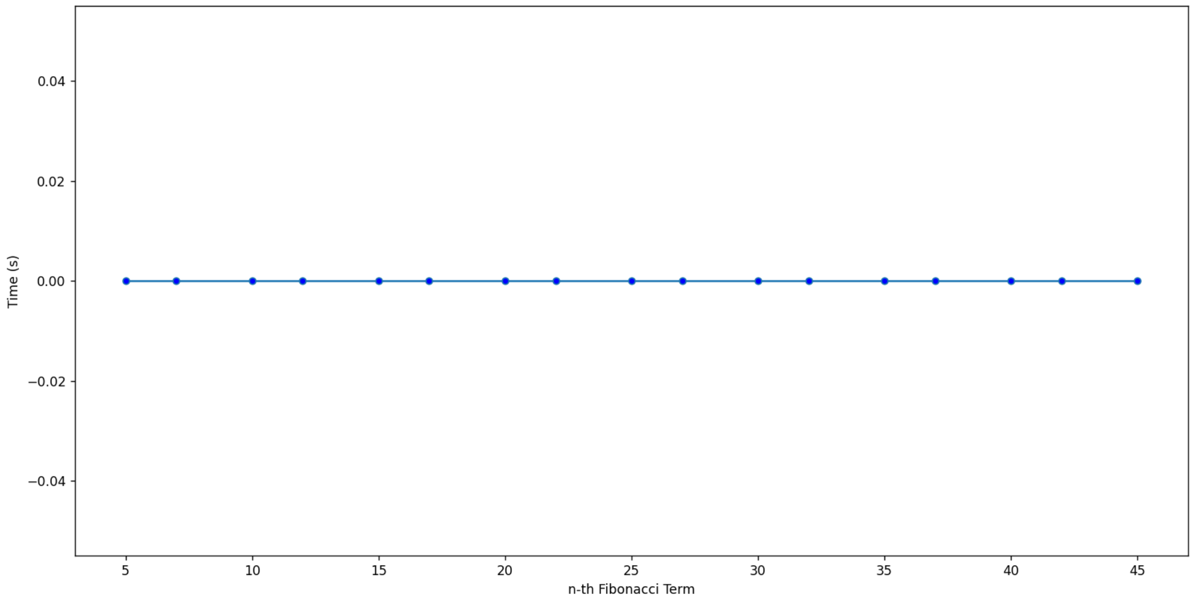

Figure 21. Tail-Call method graph

Conclusion

In conclusion, the analysis of various algorithms for finding the n-th Fibonacci number has shown that there are several efficient methods available. The methods analyzed included recursive, iterative, matrix exponentiation, Binet formula, memoization and dynamic programming solutions.

The results indicated that the Binet's formula solution and the matrix exponentiation method have the smallest time complexity and are the most efficient among the analyzed algorithms. However, it is important to note that the Binet's formula solution is only suitable for smaller values of n and the matrix exponentiation method requires a deeper understanding of mathematical concepts. Therefore, the choice of algorithm will depend on the specific requirements of the application. Overall, this report highlights the importance of considering different algorithms and evaluating their time complexity when solving computational problems.

Github repository: https://github.com/alya1007/Labs-semester-4/tree/master/AA