Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Algorithm Analysis

Laboratory work #1

Study and Empirical Analysis of Algorithms for Determining Fibonacci N-th Term

Author: Timur Cravtov
std. gr. FAF-231

Verified by: Cristofor Fistic
asist. univ

Chișinău
2025

1 Algorithm Analysis

1.1 Objective

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

1.2 Task

1. Implement at least 3 algorithms for determining the Fibonacci n-th term;
2. Decide properties of the 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.

1.3 Theoretical Notes

Empirical analysis is an alternative to mathematical complexity evaluation. It helps classify algorithm complexity, compare algorithm efficiency, assess different implementations, and evaluate performance on specific hardware.

The process typically includes:

1. Define the analysis objective;
2. Select an efficiency metric (e.g., operation count or execution time);
3. Determine input data properties (size, structure);
4. Implement the algorithm;
5. Generate test datasets;
6. Execute the program on each dataset;
7. Analyze the results.

The choice of efficiency metric depends on the goal. If verifying complexity, operation count is suitable; for implementation performance, execution time is better. Results are then analyzed using statistics or visualized in graphs.

1.4 Introduction

Fibonacci sequence is a sequence of numbers started with $0, 1$ (or $1, 1$) and each next term is calculated as the sum of two previous ones. For example, $0, 1, 1, 2, 3, 5, 8, 13, 21 \ldots$

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.

2 Implementation

All the algorithms will be implemented in Python as they are, in naive method, with no additional improvements in perfomance and language features.

2.1 Recursive method

Algorithm Description

The recursive method in its naive form calculates each term recursively, until it reaches 1: for 1st term; as shown below:

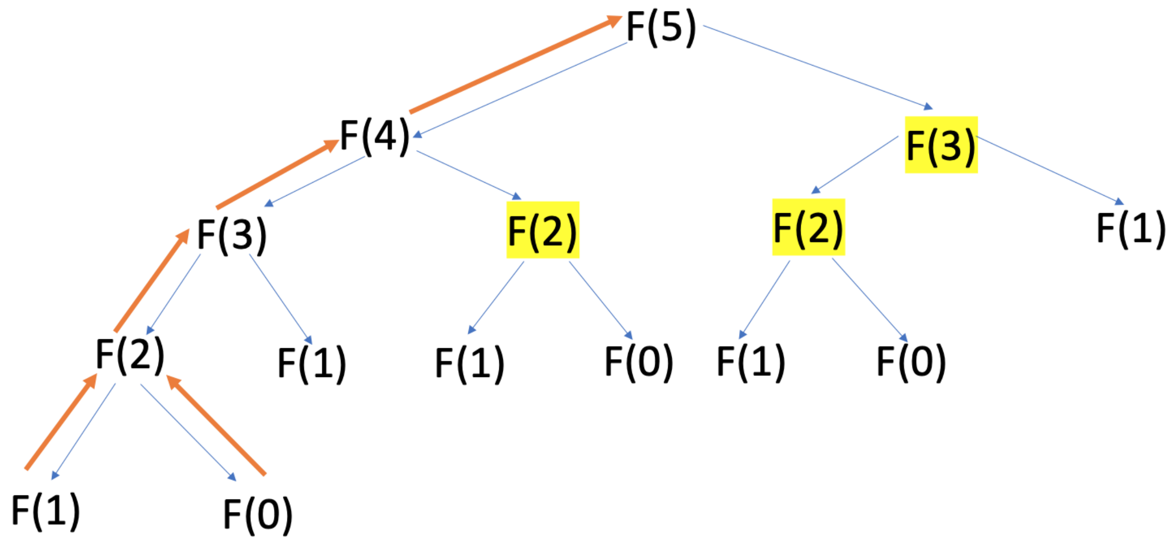

Figure 1: Visual representation of recursive method

```
FUNCTION FIBONACCI(N)
    IF N less or equal than 1 THEN
        RETURN N
    ELSE
        RETURN FIBONACCI(N - 1) + FIBONACCI(N - 2)
    END IF
END FUNCTION
```

Listing 1: Pseudocode for recursive algorithm

Implementation

```python
# first method, recursion
def fib_rec(n):
    return n if n <= 1 else fib_rec(n-1) + fib_rec(n-2)
```

Listing 2: Python fib_rec(n) function

Results

As shown on figure ??, already on the 39th Fibonacci term, the time needed for its calculation is around 14 seconds, which suggests a exponential complexity of the algorithm.

Fn	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39
Time	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.04	0.10	0.31	0.77	1.95	5.04	13.70

Figure 2: Recursion alg. time execution raw data

And indeed, the diagram on figure ?? based on the above data looks like $f(t) = a^t$ graph which proofs the exponential nature of this recursive algorithm.

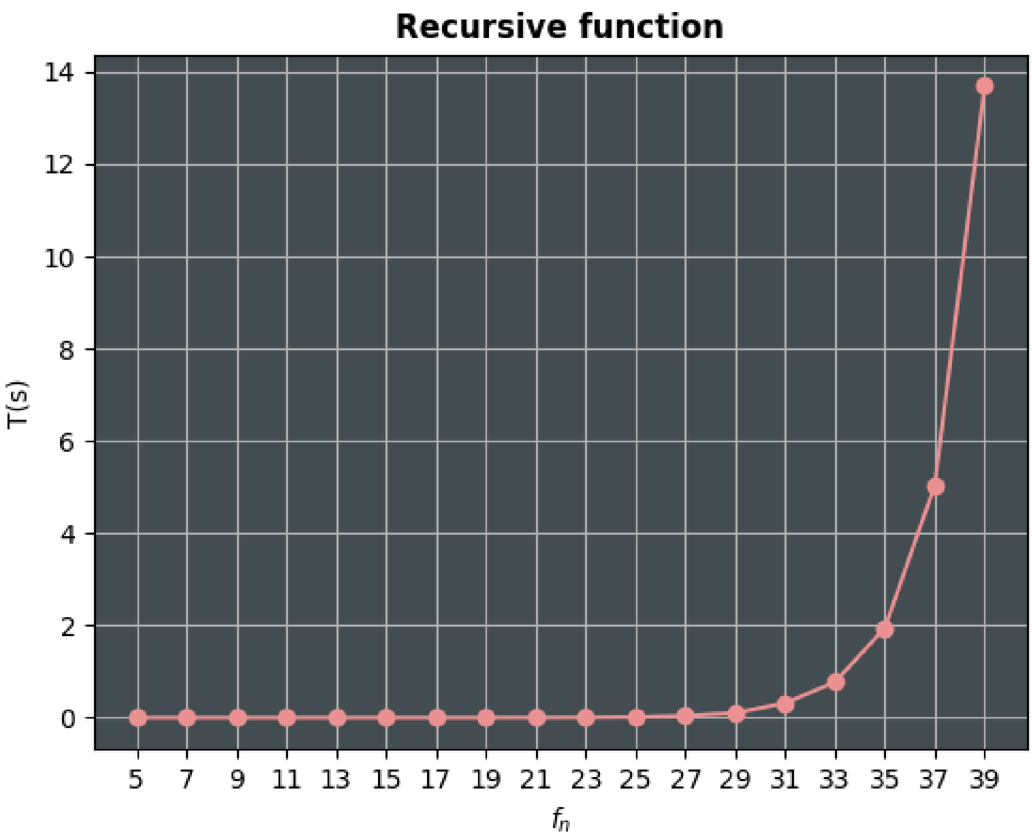

Figure 3: Recursion alg. time execution diagram

2.2 Iteration method

Algorithm description

For this iteration algorithm, two variables are created, which hold the value of the 1st and the 2nd term of the sequence. Each iteration, the first is replaced with the second, and the first is added to second. The pseudocode of the algorithm:

```
FUNCTION FIBONACCI(N)
    SET A TO 0
    SET B TO 1
    FOR I FROM 0 TO N - 1 DO
        SET TEMP TO A
        SET A TO B
        SET B TO TEMP + B
    END FOR
    RETURN A
```

```
END FUNCTION
```

Implementation

```python
# second method, using iteration
def fib_it(n):
    a, b = 0, 1
    for i in range(0, n):
        a, b = b, a + b
    return a
```

Listing 3: Python fib_it(n) function

Results

Iterative algorithm is much better than recursive one. The 15.000th term is calculated in below the second, suggesting that the algorithm is better than exponential one. Algorithm, apparently makes the straight line and has a form $O(n)$

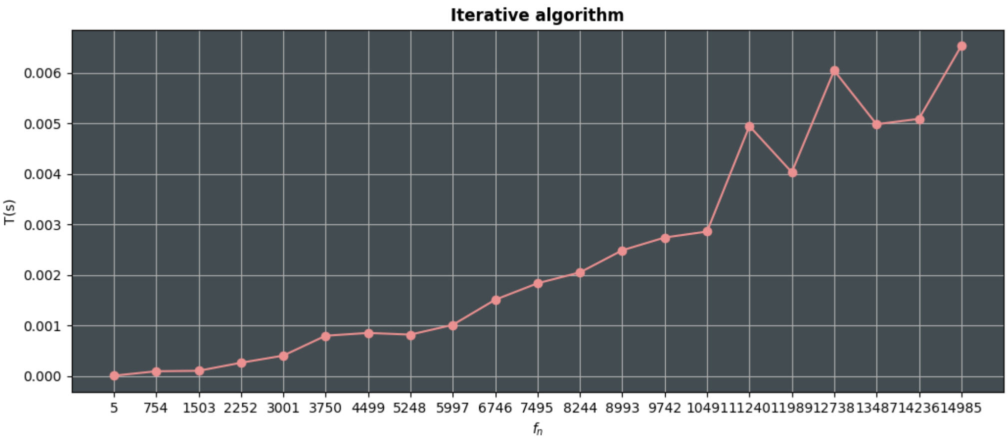

Figure 4: Iterative alg. time execution diagram

2.3 Matrix power method

The recursive relation of the Fibonacci terms can be rewritten as:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix}$$

By applying this transformation repeatedly, we obtain:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \left(\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \right)^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

Since we know that:

$$F_1 = 1, \quad F_0 = 0$$

it follows that:

$$F_n = \left(\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \right)_{0,0}$$

```
FUNCTION FIBONACCI_MATRIX(N)
    IF N      1 THEN
        RETURN N
    END IF

    SET MAT TO [[1, 1], [1, 0]]

    CALL MATRIX_POWER(MAT, N - 1)

    RETURN MAT[0][0]
END FUNCTION
```

Implementation

```python
# using matrix multimplication
def fib_matrix(n):
    F = [[1, 1],
         [1, 0]]
    if n == 0:
        return 0
    power(F, n - 1)
    return F[0][0]

### some util functions

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

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

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], F[0][1], F[1][0], F[1][1] = x, y, z, w
```

Listing 4: Python fib_matrix(n) function

Results

Fn	5	754	1503	2252	3001	3750	4499	5248	5997	6746	7495	8244	8993	9742	10491	11240	11989	12738	13487	14236	14985
Time	0.000008	0.000695	0.001480	0.002360	0.003402	0.006419	0.006386	0.009619	0.010967	0.013191	0.015347	0.018349	0.021772	0.022907	0.030834	0.035398	0.035155	0.037746	0.048009	0.045332	0.051026

Figure 5: Matrix power method raw data

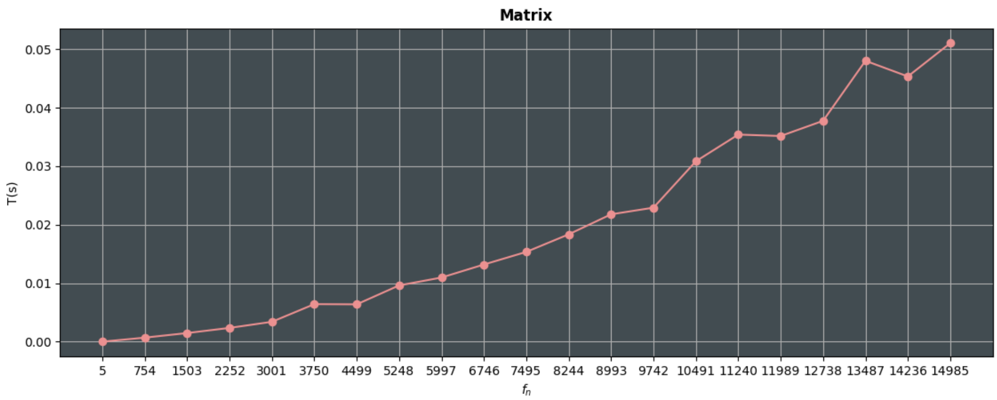

Figure 6: Matrix power method diagram

2.4 Fast doubling method

Algorithm description

The fast doubling method is build on the following properties (or relations) of the Fibonacci sequence, which are derived from the Matrix method

$$F_{2k} = F_k \cdot (2F_{k+1} - F_k)$$

$$F_{2k+1} = F_{k+1}^2 + F_k^2$$

where F_k and F_{k+1} are computed recursively for $k = \lfloor n/2 \rfloor$.

1. We first compute F_k and F_{k+1} using recursion.

2. Then, we use the doubling formulas:

- If n is even, $F_n = F_{2k}$.
- If n is odd, $F_n = F_{2k+1}$.

3. This method reduces the time complexity from $O(n)$ to $O(\log n)$.

The pseudocode is the following:

```
FUNCTION FIBONACCI_FAST_DOUBLING(N)
    IF N = 0 THEN
        RETURN 0
    ELSE IF N = 1 THEN
        RETURN 1
    END IF

    FUNCTION FIB_DOUBLING(K)
        IF K = 0 THEN
            RETURN (0, 1)
        END IF

        (FK, FK1) <- FIB_DOUBLING(K DIV 2)

        F2K <- FK * (2 * FK1 - FK)
        F2K1 <- FK1 * FK1 + FK * FK

        IF K MOD 2 = 0 THEN
            RETURN (F2K, F2K1)
        ELSE
            RETURN (F2K1, F2K + F2K1)
        END IF
    END FUNCTION

    RETURN FIB_DOUBLING(N)[0]
END FUNCTION
```

Implementation

```
# seventh method, using fast doubling formula
def fib_fast_doubling(n):
    if n == 0:
        return 0
    elif n == 1:
```

```python
        return 1

    def fib_doubling(k):
        if k == 0:
            return (0, 1)

        Fk, Fk1 = fib_doubling(k // 2)

        F2k = Fk * (2 * Fk1 - Fk)
        F2k1 = Fk1**2 + Fk**2

        return (F2k1, F2k + F2k1) if k % 2 else (F2k, F2k1)

    return fib_doubling(n)[0]
```

Listing 5: Python fib_fast_doubling(n) function

Results

Fn	5	7504	15003	22502	30001	37500	44999	52498	59997	67496	74995	82494	89993	97492	104991	112490	119989	127488	134987	142486	149985
Time	0.000013	0.000055	0.000141	0.000182	0.000240	0.000355	0.000494	0.000932	0.001175	0.001071	0.001092	0.001271	0.002471	0.001649	0.002293	0.002408	0.002155	0.002602	0.002772	0.004733	0.004182

Figure 7: Fast doubling raw data

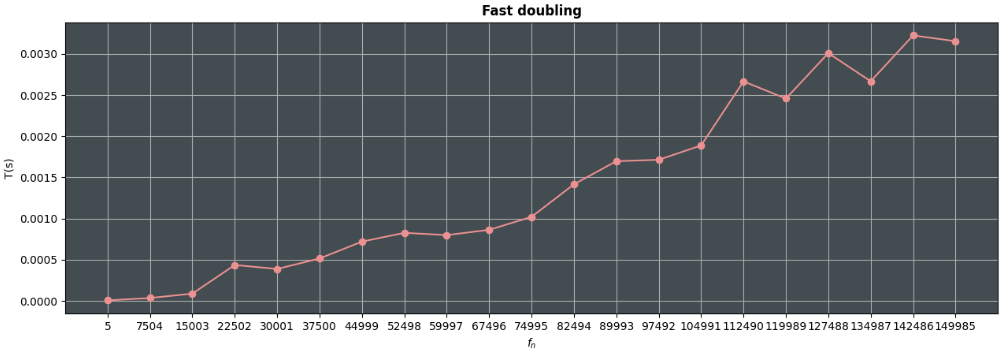

Figure 8: Fast doubling diagram

2.5 Binet's formula

Algorithm description

Binet's formula is an explicit formula for finding the n^{th} Fibbonacci term. It is so named because it was derived by mathematician Jacques Philippe Marie Binet, though it was already known by Abraham de Moivre.

If F_n is the n^{th} Fibbonacci number, then

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

Implementation

```
# using binet formula
def fib_binet(n):
    phi = (1 + sqrt(5)) / 2
    psi = (1 - sqrt(5)) / 2
    return round((phi**n - psi**n) / sqrt(5))
```

Listing 6: Python fib_binet(n) function

Since the formula doesn't have any loops or recursion, its complexity is obviously $O(1)$. In this case it's not useful to perform time analysis. However, since irrational numbers are present in formula, the rounding errors happens due to computer representation of numbers.

Results

For this purpose, a list of exact Fibonacci sequence terms were created using one of the already describer algorithms and then the difference is calculated and presented on figure **??**

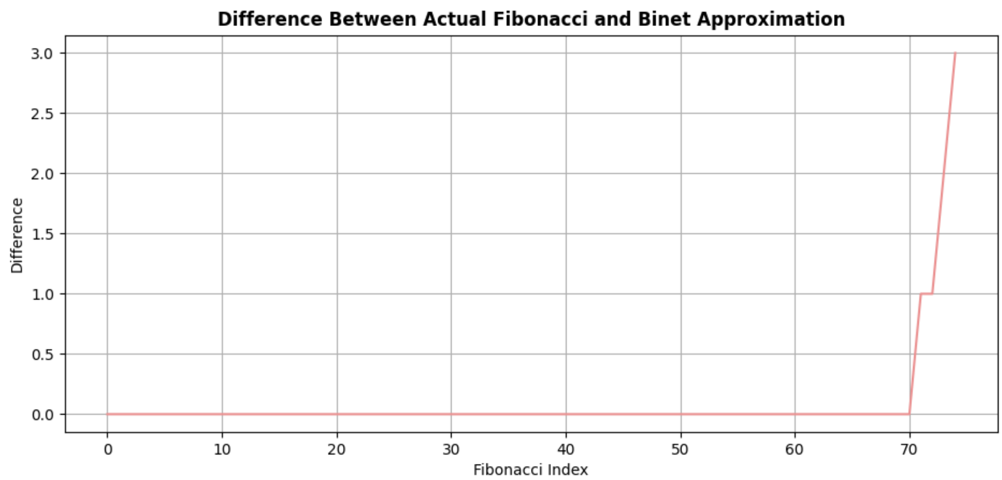

Figure 9: Enter Caption

One can easily observe that starting from 70-ish number, the rounding error leads to significant difference, which increases with index.

2.6 ϕ approximation

Algorithm description

This method is inspired by previous method with Binet formula. It's known that the ratio of the Fibonacci terms tend to infinity, or

$$\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \phi$$

So, combining the iteration method and this property, we can build a sequence $F_{n+1} = F_n \times \phi$.

Implementation

```python
# using phi approximation
def fib_phi(n):

    if n < 6:
        return f[n]

    t = 5
    fn = 5

    while t < n:
        fn = round(fn * PHI)
        t+=1

    return fn
```

Listing 7: Python fib_binet(n) function

Results

This iteration method is linear, so the time complexity is still $O(n)$. We can estimate its accuracy with the same method as for Binet's formula on figure **??**.

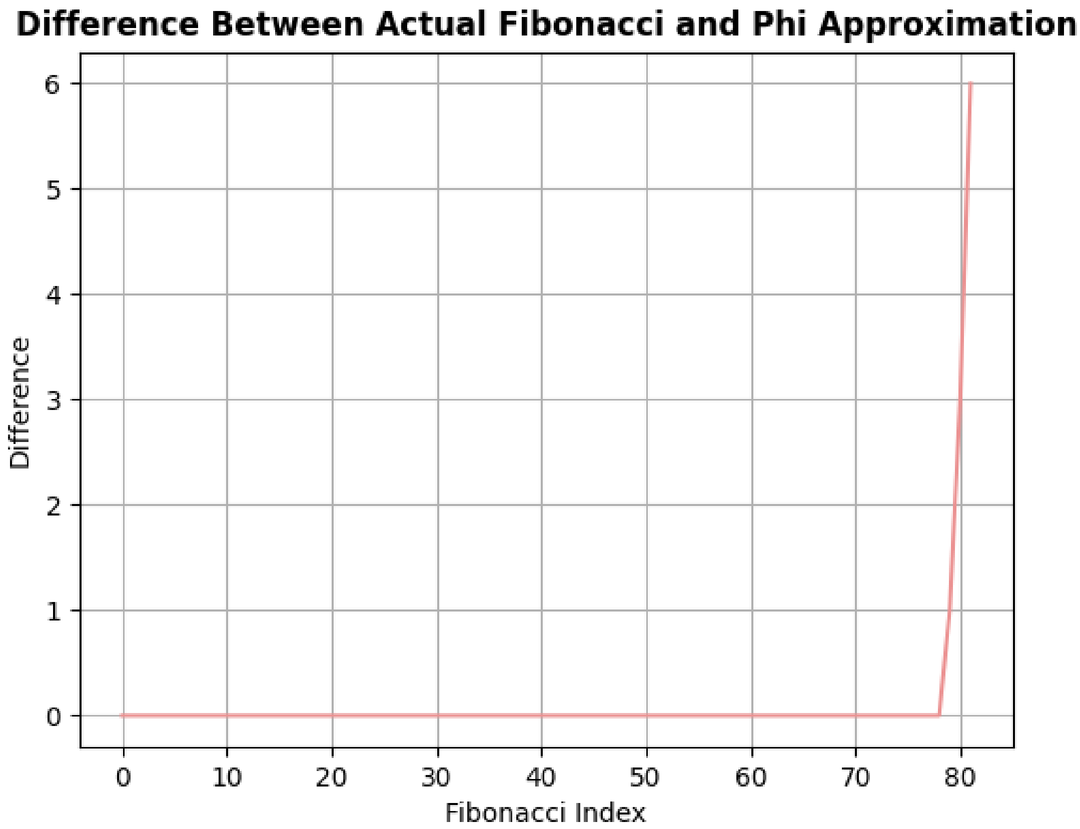

Figure 10: Enter Caption

It's easy to notice that almost at the same number, at 75-ish index, the error appears. Slighly more accurate that the Binet's, but due to linear time complexity, useless from practical point of view.

3 Conclusions

During this laboratory work, I implemented 6 algorithms for determining the nth Fibonacci term. All of them were written in their naive form, with no optimizations in terms of speed and accuracy, which Python can provide with use of its advanced features.

To sum up the methods, here is a brief analysis of each:

1. *Recursive method* - is mathematically easiest method. Easy to write, but useless for numbers bigger than 30 due to its exponential nature

2. Iteration method - still used direct approach, but is linear ($O(n)$), hence faster than recursive, but not faster than others.

3. *Matrix power method* - uses matrix properties. Has logarithmic complexity, quite fast.

4. Fast doubling method - the best method. Still used logarithmic complexity but much faster than Matrix power method (see absolute time values in **??**, **??**)

5. *Binet's formula* - the only one described method which runs in ($O(1)$). Despite its speed, rounding errors start to accumulate from the 70th term.

6. ϕ *Approximation* - is a simpler variation of the Binet's formula. Going from the fact, the fraction of a Fibonacci term and it's predecessor tends to ϕ, this constant is used to create a new value. It's linear, and it's slighly more accurate than Binet's one, so not really useful.