Technical University of Moldova
Department of Software Engineering and Automatics

Report
of laboratory work №1

Theme: Algorithm Analysis. Execution time analysis.

Fulfilled: st.gr. FAF-191 Boico Alexandr
Controlled: Mihai Gaidau

Chişinău 2021

PURPOSE OF THE WORK:

Study and empirical analysis of the algorithms that determine the N-th term in the Fibonacci sequence

BASIC TASK:

1. Implement at least 3 algorithms that determine the N-th term of the Fibonacci string in a programming language
2. Determine the properties of the input data in relation to which the analysis is performed
3. Choose the metrics to compare the algorithms
4. Perform the empirical analysis of the proposed algorithms
5. Make a graphical presentation of the data obtained
6. Make a conclusion about the work done.

TABLE OF CONTENTS:

1. Algorithms Descriptions
    1) Recursive Algorithm
    2) Dynamic Programming Algorithm
    3) Binet Formula
    4) Matrix Derivation Algorithm
2. Algorithms Comparison
    1) Recursive Algorithm
    2) Effective Algorithms
3. Conclusion

1. ALGORITHMS

1) Recursive Algorithm

```
long long RecursiveAlgorithm(int n) {
    if (n <= 1)
        return n;
    return RecursiveAlgorithm(n - 1) + RecursiveAlgorithm(n - 2);
}
```

Fig. 1.1

Recursive Algorithm is the easiest to implement in code, but it is the slowest one and its complexity grows exponentially and is equal to $O(n) = 2^n$, because this implementation does a lot of repeated work. Also, it should be noticed that this realization can overflow the stack.

This algorithm can be improved(till $O(n) = n$) by using memoization, but it will complicate code.

2) Dynamic Programming Algorithm

```
long long DynamicAlgorithm(int n) {
    int f[2] = { 0, 1 };
    int reminder;
    for (int i = 2; i <= n; i++)
    {
        reminder = f[1];
        f[1] = f[0] + f[1];
        f[0] = reminder;
    }
    return f[1];
}
```

Fig. 2.1

This implementation shows direct calculations from 1st Fibonacci number to n-th. To do this is needed only two last Fibonacci numbers. The following code has a linear execution time and a fixed memory usage. Its complexity is equal to $O(n) = n$, but memory use - $O(1)$.

3) The Binet Formula

Interesting way to calculate Fibonacci numbers is the Binet Formula.

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}$$

where $\varphi = \dfrac{1 + \sqrt{5}}{2} = 1,61803398874989\ldots$ is Phidias number, also known as golden ratio; but $\psi = \dfrac{1 - \sqrt{5}}{2} = -\dfrac{1}{\varphi}$

```
long long BinetFormule(int n) {

    double index = pow(5, 0.5);
    double phi = (1 + index) / 2;

    return round(pow(phi, n) / index);
}
```

Fig. 3.1

You can see that my realisation(Fig. 3.1) doesn't use ψ, because it is less than 1 and exponentiation transforms it into 0, so we can neglect it.

For this algorithm the complexity is O(n) = log(n) and uses O(1) memory.

It should be said that the first error appears on n = 71(Fig. 3.2). $F_{71} = 308061521170129$, but the program returned us 308061521170130.

```
Консоль отладки Microsoft Visual Studio
n = 71
====================
        Dynamic Programic Algorithm
Time of execution : 0 microseconds
Answer : 308061521170129
====================
        Binet Formula
Time of execution : 17 microseconds
Answer : 308061521170130
====================
        Matrix Derivation Algorithm
Time of execution : 17 microseconds
Answer : 308061521170129
====================
```

Fig 3.2

4) Derivation From Matrix Equation

The formula can be derived from the above matrix equation.

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

Taking determinant on both sides, we get

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

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

$$F_m F_n + F_{m-1} F_{n-1} = F_{m+n-1}$$

By putting $n+1$ in place of n,

$$F_m F_{n+1} + F_{m-1} F_n = F_{m+n}$$

Putting $m = n$

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

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

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

If n is even, we put $k = n/2$.

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

```
long long MatrixDerivationAlgorithm(int n)
{
    if (n == 0)
        return 0;
    if (n == 1 || n == 2)
        return 1;
    if (f[n])
        return f[n];

    int k = (n & 1) ? (n + 1) / 2 : n / 2;

    return (f[n] = (n & 1) ? (MatrixDerivationAlgorithm(k) * MatrixDerivationAlgorithm(k)
        + MatrixDerivationAlgorithm(k - 1) * MatrixDerivationAlgorithm(k - 1))
        : (2 * MatrixDerivationAlgorithm(k - 1) + MatrixDerivationAlgorithm(k)) * MatrixDerivationAlgorithm(k));
}
```

Fig 4.1

Complexity of this algorithm is O(n) = log(n).

2. RESULTS OF THE CALCULATIONS

1) Recursion

n	Time (s)
1	0
5	0.000001
10	0.000007
20	0.000628
30	0.084475
40	10.442687
50	634.568467

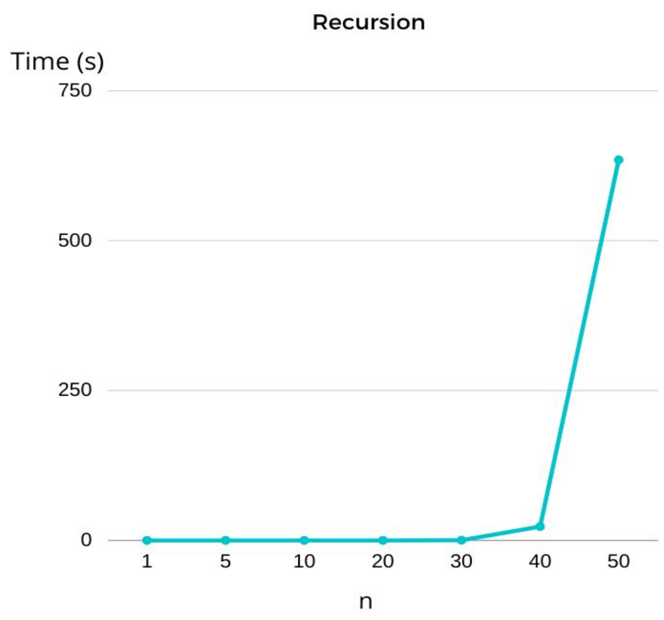

Fig. 5.1

2) Effective Algorithms

n	Dynamic Programming (µs)	Binet Formula (µs)	Matrix Derivation (µs)
1	1	20	20
5	1	16	16
10	1	24	24
50	1	24	24
100	1	18	18
250	1	26	26
500	2	17	17
1000	8	19	19
2500	8	17	17
5000	17	16	16
10000	33	16	16
25000	200	15	15
50000	163	16	16

100000	332	32	32
250000	824	16	16
500000	1656	19	19
1000000	3319	20	20

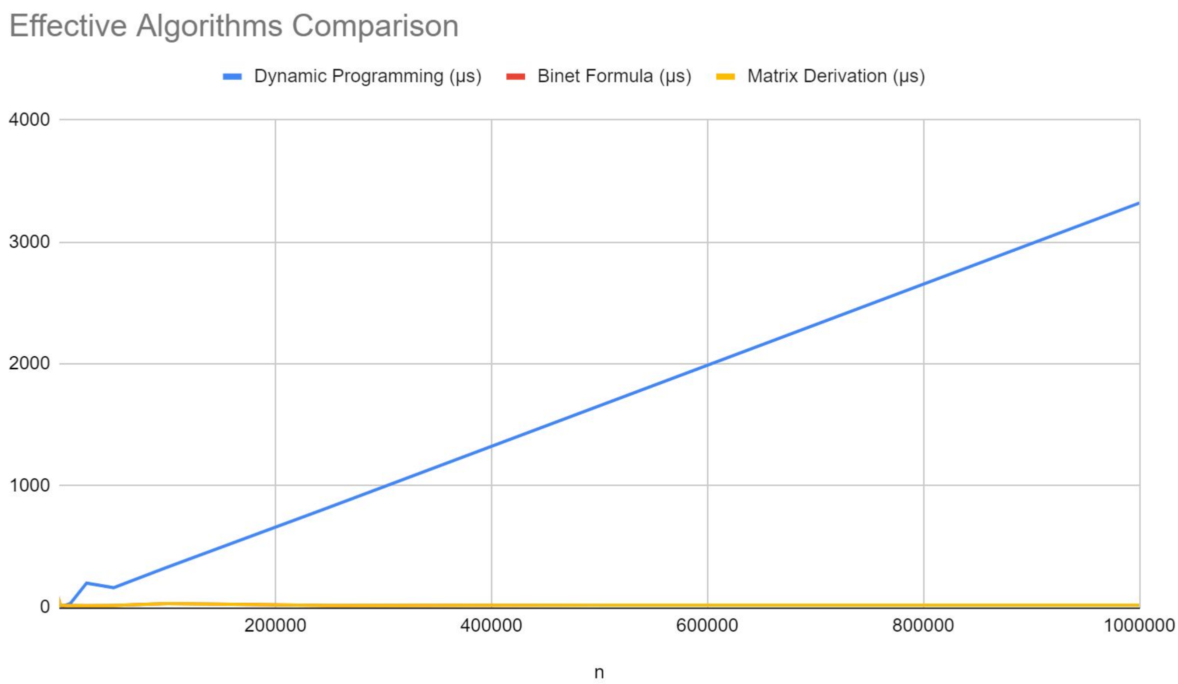

Fig 5.2

CONCLUSION

Based on the calculations, we can draw the conclusion:

1) Recursion is most ineffective algorithm and using all solutions that grows exponentially is not recommended;

2) Recursive algorithm can be improved till O(n) = n, but such upgrades are more complicated than Dynamic Programming algorithm that has the same complexity.

3) Linear algorithms are effective for small numbers, wherein they save simplicity of implementing in code;

4) Logarithmic algorithms are most effective for big numbers, but loose for small.

5) Binet Formula is an approximation to Figonacci sequence and is applicable for $n < 71$. After that it returns answers with error.