Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică
Departamentul Ingineria Software și Automatică

Laboratory Work No.1

Theme: Algorithm Analysis (Execution time of algorithms)

Realized by:

Popa Eugeniu FAF-202

Chisinau 2021

Contents

1. Algorithm analysis

1.1 The purpose of the work:

Study empirical analysis of algorithms that determine the nth term in a Fibonacci sequence.

1.2 Theoretical notes:

An alternative to mathematical analysis of complexity is empirical analysis.

This may be useful for:

(i) obtaining preliminary information on the complexity class of an algorithm;
(ii) to compare the efficiency of two (or more) algorithms for solving the same problems;
(iii) to compare the efficiency of several implementations of the same algorithm;
(iv) to obtain information on the efficiency of implementing an algorithm on a particular computer.

The stages of empirical analysis. 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 are established in relation to which the analysis is performed (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 behavior implementation of an algorithm then the appropriate execution time is appropriate.

To perform an empirical analysis is not enough a single set of input data but several, which to highlight the different features of the algorithm. It is generally good to choose data different sizes so as to cover the range of all sizes that will appear in practice. On the other hand, the analysis of different values or configurations of the input data is also important.

If so, analyzes an algorithm that verifies whether a number is prime or not and testing is done only for numbers which are not prime or only for numbers that are prime then a relevant result will not be obtained.

The same thing can happen with an algorithm whose behavior depends on the degree of sorting of an array (if you choose only the array almost sorted according to the desired criterion or arrays ordered in reverse analysis will not be relevant).

In order to empirically analyze the implementation of the algorithm in a programming language will need

introduced sequences whose purpose is to monitor execution. If the efficiency metric is the number of executions of an operation then a counter is used which is incremented after each execution a that operation. If the metric is the execution time then the time of entry must be recorded the analyzed sequence and the time of exit.

Most programming languages offer measurement functions the time elapsed between two moments. This is especially important if you are active on your computer several tasks, to count only the time allocated to the execution of the analyzed program.

Especially if it is about measuring the time it is indicated to run the test program several times and to calculate the average value of time.

After the execution of the program for the test data the results are recorded and for the purpose of the analysis either calculates synthetic quantities (mean, standard deviation, etc.) or plot pairs of points shapes (problem size, efficiency measure).

Execution time of the whole algorithm is obtained by summing the execution times of the processing component.

The importance of the worst-case scenario. In the appreciation and comparison of algorithms, the most interesting unfavorable case because it provides the longest execution time relative to any size input data fixed. On the other hand, for some algorithms the worst case is relatively common. As for the analysis of the most favorable case, it provides a lower margin of time and can be useful for identifying inefficient algorithms (if an algorithm has a high cost in the most favorable case, then it cannot be considered an acceptable solution).

Average execution time. Sometimes, extreme cases (the most unfavorable and the most favorable) are rare, so the analysis of these cases does not provide enough information about the algorithm. In these situations, another measure of the complexity of the algorithms is useful, namely the average execution time.

This represents an average value of the execution times calculated in relation to the probability distribution corresponding to the input data space.

2. Introduction

Leonardo Fibonacci was a medieval Italian mathematician, famous for the Fibonacci sequence that is widely used in many domains, such as mathematics, architecture, art, nature etc.

In this laboratory work, we are going to examine some of the Fibonacci algorithm implementations, using empirical analysis, which will use concrete, accurate and repeatable real-world evidence, instead of personal perspective. The central idea behind empirical analysisis that direct observation is the best way to examine reality and find the truth.

In order to solve the problem, there must be used several possible algorithm implementations, and two main selection criteria must be respected for them:

1) the following algorithm implementation should be simple to understand, to code and to troubleshoot;

2) the algorithms should efficiently use computer resources, to have a short execution time.

In this case, if the program being written has to be run a small number of times, the first requirement is more important. In this situation, the time to set up the program is more important than its time so the simplest version of the program should be chosen. If the program is to be run a large number of times, with a large number of data processed, the algorithm that leads to a faster execution must be chosen. Even in this situation, it should previously implement the simplest algorithm and calculated the reduction of execution time that would bring it implementation of the complex algorithm.

Thus, by all that saying in the current laboratory work the comparison metric is going to be time of execution for each algorithm, the results being presented in a graphical and a tabular form.

3.Implementation

3.1 Method 1 (Use recursion)

A simple method that is a direct recursive implementation mathematical recurrence relation is given above.

```
// Fibonacci Series using Recursion

function fib1(n) {
    if (n <= 1)
        return n;
    return fib1( n: n-1) + fib1( n: n-2);
}
```

Time Complexity: O (2 ^ N) (Exponential Time Complexity).
Since every value is made up of previous 2 values we need to find 2 values for finding any Fibonacci number. This gives rise to 2 calls every time in our recursion tree. The tree can have at most n levels. This gives us at most 2 ^ n nodes in the tree.
Space Complexity: O (2 ^ N) (Exponential Space Complexity).
All the function calls are pushed in the function call stack. At any point of time at max 2 ^ n function calls are there in call stack therefore space complexity will be O (2 ^ n).

3.2 Method 2 (Use dynamic programming)

We can avoid the repeated work done in method 1 by storing the Fibonacci numbers calculated so far.

Time Complexity: O (N) (Linear Time Complexity)
This is because for finding the nth Fibonacci number we move through an array of size n (actually we are moving (n-2) indices in the array).
Space Complexity: O (N) (Linear Space Complexity)
This is because for storing answers of all the smaller subproblems before the nth Fibonacci number we need an array of size n.

```
function fib2(n) {
    // Declare an array to store Fibonacci numbers
    let f = new Array( arrayLength: n+2); // 1 extra to handle case, n = 0
    let i;
    // 0th and 1st number of the series are 0 and 1
    f[0] = 0;
    f[1] = 1;
    for (i = 2; i <= n; i++) {
        // Add the previous 2 numbers in the series and store them
        f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```

3.3 Method 3 (Space optimized method 2)

We can optimize the space used in method 2 by storing the previous two numbers only because that is all we need to get the next Fibonacci number in series.

Time Complexity: O (n)
Space Complexity: O (1)

```
// Fibonacci Series using Space Optimized Method

function fib3(n) {
    let a = 0, b = 1, c, i;
    if(n == 0)
        return a;
    for(i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

3.4 Method 4 (Use power of the matrix {{1, 1}, {1, 0}})

This another O(n) which 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 resulting matrix.

The matrix representation gives the following closed expression for the Fibonacci numbers:

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

Time Complexity: O (n)
Space Complexity: O (1)

```
// Fibonacci Series using power of the matrix {{1, 1}, {1, 0}}

function fib4(n) {
    let F = [ [ 1, 1 ], [ 1, 0 ] ];
    if (n == 0)
        return 0;
    power(F, n: n - 1);
    return F[0][0];
}

```

```
// Helper function that multiplies 2 matrices F and M of size 2*2, and
// puts the multiplication result back to F[][]

function 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] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

// Helper function that calculates F[][] raise to the
// power n and puts the result in F[][]

function power(F, n) {
    var i;
    var M = [[ 1, 1 ], [ 1, 0 ]];

    // n - 1 times multiply the matrix to {{1,0},{0,1}}
    for(i = 2; i <= n; i++)
        multiply(F, M);
}
```

3.5 Method 5 (Use Binet's Formula)

In this method we directly implement the formula for nth term in the Fibonacci series.
It involves our golden section number Phi and its reciprocal phi.
Formula:
Fn = {[(√5 + 1) / 2] ^ n} / √5

Time Complexity: O (logn), this is because calculating phi ^ n takes logn time.
Space Complexity: O (1)

```
// Fibonacci Series using Binet's Formula

function fib5(n) {
    let phi = (1 + Math.sqrt( x 5)) / 2;
    return Math.round( x Math.pow(phi, n) / Math.sqrt( x 5));
}
```

4. Analysis of the results. Empirical analysis of the algorithms using time

Code snippet for analyzing the time comparison of the algorithms:

Written form:

```
87
88  // Time comparison of the algorithms in written form
89
90  const { performance } = require('perf_hooks')
91  const { plot } = require('nodeplotlib');
92
93  let fibNums = [1, 5, 10, 20, 30]
94
95  function runFib(nums) {
96      var performance1 = []
97      var performance2 = []
98      var performance3 = []
99      var performance4 = []
100     var performance5 = []
101
102     for (let i = 0; i < fibNums.length; i++) {
103         let time;
104         console.log(`${fibNums[i]} nth Fibonacci number`)
105
106         var startTime1 = performance.now()
107         fib1(fibNums[i])
108         var endTime1 = performance.now()
109         time = endTime1 - startTime1
110         performance1.push(endTime1 - startTime1)
111
```

```
        var startTime2 = performance.now()
        fib2(fibNums[i])
        var endTime2 = performance.now()
        time = endTime2 - startTime2
        performance2.push(endTime2 - startTime2)

        var startTime3 = performance.now()
        fib3(fibNums[i])
        var endTime3 = performance.now()
        time = endTime3 - startTime3
        performance3.push(endTime3 - startTime3)

        var startTime4 = performance.now()
        fib4(fibNums[i])
        var endTime4 = performance.now()
        time = endTime4 - startTime4
        performance4.push(endTime4 - startTime4)

        var startTime5 = performance.now()
        fib5(fibNums[i])
        var endTime5 = performance.now()
        time = endTime5 - startTime5
        performance5.push(endTime5 - startTime5)

        console.log(`Method 1`);
        console.log(`Execution time: ${performance1[i]}`);
        console.log(`Method 2`);
        console.log(`Execution time: ${performance2[i]}`);
        console.log(`Method 3`);
        console.log(`Execution time: ${performance3[i]}`);
        console.log(`Method 4`);
        console.log(`Execution time: ${performance4[i]}`);
        console.log(`Method 5`);
        console.log(`Execution time: ${performance5[i]}\n`);
    }
```

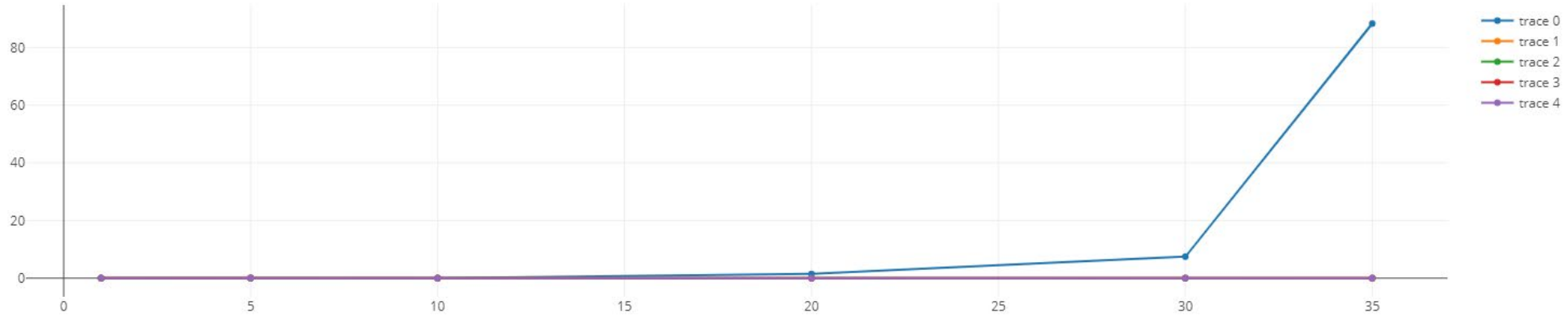
Graphical form:

```
147
148         // Time comparison of the algorithms in graphical form
149
150         const method1 = {x: fibNums, y: performance1, type: 'line'}
151         const method2 = {x: fibNums, y: performance2, type: 'line'}
152         const method3 = {x: fibNums, y: performance3, type: 'line'}
153         const method4 = {x: fibNums, y: performance4, type: 'line'}
154         const method5 = {x: fibNums, y: performance5, type: 'line'}
155
156         plot( data: [
157             method1,
158             method2,
159             method3,
160             method4,
161             method5
162         ])
163     }
164
165     runFib(fibNums)
```

Below you can see the results of the empirical analysis in written form.

```
1 nth Fibonacci number
Method 1
Execution time: 0.04119999706745148
Method 2
Execution time: 0.041999999433755875
Method 3
Execution time: 0.031700000166893005
Method 4
Execution time: 0.05510000139474869
Method 5
Execution time: 0.029500000178813934

5 nth Fibonacci number
Method 1
Execution time: 0.00780000165104866
Method 2
Execution time: 0.037700001150369644
Method 3
Execution time: 0.001499999314546585
Method 4
Execution time: 0.15789999812841415
Method 5
Execution time: 0.00469999760389328

10 nth Fibonacci number
Method 1
Execution time: 0.014600001275539398
Method 2
Execution time: 0.004599999636411667
Method 3
Execution time: 0.0016000010073184967
Method 4
Execution time: 0.031599998474121094
Method 5
Execution time: 0.006400000303983688

20 nth Fibonacci number
Method 1
Execution time: 1.4626000002026558
Method 2
Execution time: 0.025299999862909317
Method 3
Execution time: 0.001400001347064972
Method 4
Execution time: 0.02500000223517418
Method 5
Execution time: 0.004799999296665192

30 nth Fibonacci number
Method 1
Execution time: 7.440600000321865
Method 2
Execution time: 0.03720000013709068
Method 3
Execution time: 0.0017000027000904083
Method 4
Execution time: 0.038100000470876694
Method 5
Execution time: 0.01419999822974205

35 nth Fibonacci number
Method 1
Execution time: 84.30040000006557
Method 2
Execution time: 0.005699999630451202
Method 3
Execution time: 0.001499999314546585
Method 4
Execution time: 0.03529999777674675
Method 5
Execution time: 0.010099999606609344
```

Below you can see the results of the empirical analysis in graphical form.



5. Conclusion

In this laboratory work we have performed an empirical analysis of the Fibonacci implementations, deducing that the best way to implement these algorithms is to use formulae, rather than direct calculations like recursion, because the results show that large numbers take longer to calculate, simply because the program does a lot of calculations to return only one number. The best method is Binet's formula, since it requires the least memory and time, and only one formula is required for the entire calculation.

At the same time, methods 2, 3 and 4 can also be considered good, because, if we look at the empirical analysis, we can see that the time taken to perform the operations is even better sometimes, rather than for Binet's formula, so it means that these implementations can also be used.

On the other hand, if we look at the first method, we can see a huge difference in the results in comparison with other algorithms. It can be explained by the fact that in the following method we have an exponentialcomplexity, which is provoked by recursion calculations.

Based on the calculations, we can draw the conclusion:

1) Recursion is the most ineffective algorithm and using any solutions that grow exponentially is not recommended;
2) Recursive algorithm can be improved till O(n) = n, but such upgrades are more complicated than the dynamic programming algorithm that has the same complexity.
3) Linear algorithms are effective for small numbers, whereas they save simplicity of implementing in code;
4) Logarithmic algorithms are the most effective for big numbers, but less effective for the small ones.