Technical University of Moldova
Department of Software Engineering and Automatics

Analysis of algorithms (Execution time of algorithms)

FAF193: Miron Cristian Catalin
FAF.CC16.1 Calculability and Complexity
Professor: Mihai Gaidau
9th of February, 2021

Content

Laboratory objective

- To elaborate 4 different algorithms of finding n^{th} term of Fibonacci Series;
- Empiric analysis of these algorithms.

Prerequisites

- GCC Compiler;
- Internet Browser;
- Manual;
- SDA course on udemy.com;

Base tasks

- Implement the algorithms in a programming language;
- Determine the properties of the input data which will be analysed;
- Choose a metric for comparing the algorithms;
- Perform the empiric analysis on the algorithms;
- Make a graph out of the output data;
- Elaborate the conclusion over the work done;

Questions to be answered

- Enumerate the factors that influence the algorithms' time of execution;
- What are the stages of an empiric analyse?
- What are the cases when we need an empiric analyse?

1.Introduction

What is Fibonacci Series?

In mathematics, the Fibonacci numbers, commonly denoted *Fn*, form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1.

That is: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$, for n > 1;

And thus, the sequence looks like:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…

Fibonacci numbers are strongly related to the golden ratio: Binet's formula expresses the nth Fibonacci number in terms of n and the golden ratio, and implies that the ratio of two consecutive Fibonacci numbers tends to the golden ratio as n increases.

2.Implementation of the algorithms

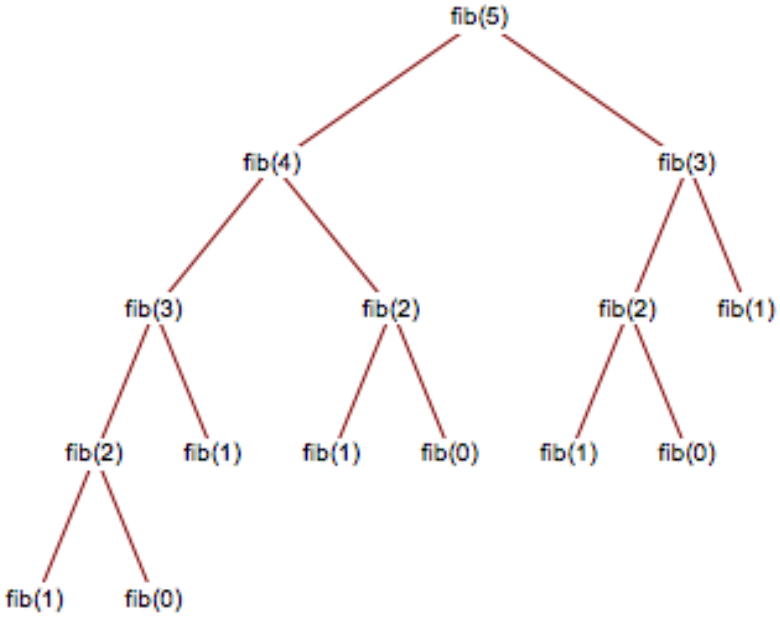
The implementation will be done in C;

2.1 Finding n^{th} term of Fibonacci series using recursive implementation

```
int recursiveFib(int n)
{
    if (n <= 1)
        return n;

    else
        return recursiveFib(n - 2) + recursiveFib(n - 1);
}
```

A rather simple implementation, but as we get into the details, we see that the simplicity comes at a cost.
To illustrate my point, here we have a tree graph of the recursive calls made by this function, with the input 5;



As we can observe, recursiveFibonacci(5) makes 15 recursive calls in total
recursiveFibonacci(3) = 5 calls;
recursiveFibonacci(4) = 9 calls;
recursiveFibonacci(6) = 31 calls;
recursiveFibonacci(7) = 63 calls and so on…

From the data presented above, we can deduce the formula, that will help us get the amount of calls, which is

recursiveFibonacci(n) = 2^{n-1}-1 calls;

And thus, the time complexity is **O(2^n)**, which means the time of execution grows **exponentially** as the input gets bigger.

The space required is proportional to the maximum depth of the recursion tree, because, that is the maximum number of elements that can be present in the implicit function call stack, hence the space complexity of recursiveFibonacci is O(N).

2.2 Finding n^{th} term of Fibonacci series using iterative implementation

```
int iterativeFibonacci(int n)
{
  int t0 = 0;
  int t1 = 1;
  int sum;

  if(n<=1)
    return n;

  for(int i=2; i<=n; ++i)
  {
    sum = t0 + t1;
    t0 = t1;
    t1 = sum;
  }
  return sum;
}
```

This algorithm contains only one for loop, which has 3 instructions, that will be executed n-1 times, so basically the formula for this one is:

iterativeFibonacci(n) ~ 4(n-1)

And thus, the time complexity is **O(N)**, which means the time of execution grows **linearly** as the input gets bigger.

The amount of space required is the same for iterativeFibonacci(6) and iterativeFibonacci(100), i.e. as N changes the space/memory used remains the same. Hence it's space complexity is O(1) or constant.

2.3 Finding n^{th} term of Fibonacci series using recursive implementation with memoization

```
int memoizationFibonacci(int n)
{
    int F[10] = {-1};
    for (int i = 0; i < 10; ++i)
    {
        F[i] = -1;
    }

    if (n <= 1)
    {
        F[n] = n;
        return n;
    }
    else
        {
        if (F[n - 2] == -1)
            F[n - 2] = memoizationFibonacci(n - 2);
        if (F[n - 1] == -1)
            F[n - 1] = memoizationFibonacci(n - 1);

        F[n] = F[n - 2] + F[n - 1];

        return F[n - 2] + F[n - 1];
    }
}
```

In this case, the number of instruction executions have been reduced to n times, and the formula looks like:

memoizationFibonacci(n) ~ n

Thus the time complexity is **O(N)**, and the space complexity is as well **O(N)**, as there are N elements that have a value.

2.4 Finding n^{th} term of Fibonacci series using Binet's formula

Binet's formula for finding n^{th} of Fibonacci series looks like this:

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

For this one, we'll need to include math.h library, since we need to compute sqrt(5), and we'll need to round up, as the output should be an integer;

The code will look like this:

```
long double BinetFibonacci(int n) {
    double rootOf5 = sqrt(5);
    double phiP = (1 + rootOf5) / 2;
    double phiN = (1 - rootOf5) / 2;

    return round((pow(phiP, n) - pow(phiN, n)) / rootOf5);
}
```

By implementing this formula, we shrink the time and space complexity down to just **O(1)!**

3.Analysing the data

Here’s a table with all the time/space complexities:

Function/ Complexity	BinetFibonacci	iterativeFibonacci	memoizationFib	recursiveFibonacci
Time	**O(1)**	**O(N)**	**O(N)**	**O(N^2)**
Space	**O(1)**	**O(1)**	**O(N)**	**O(N)**

And just to demonstrate how quick the first ones are compared to the slower ones:

Function/ n	BinetFibonacci	iterativeFibonacci	memoizationFib	recursiveFibonacci
5	0.0 sec	0.0 sec	0.0 sec	0.0 sec
10	0.0 sec	0.0 sec	0.0 sec	0.0 sec
25	0.0 sec	0.0 sec	0.0 sec	0.06 sec
50	0.0 sec	0.0 sec	0.0 sec	>10 min
100	0.0 sec	0.0 sec	0.015 sec	Too much
1 000	0.0 sec	0.0 sec	-	Way too much
100 000	0.0 sec	0.12 sec	-	PC dead
1 000 000	0.0 sec	9.84 sec	-	Send help pls

recursiveFibonacci(35) = 3 sec
recursiveFibonacci(40) = 12 sec

Coming up to n = 50, I didn’t want to wait, but I think the point is clear.

4.Answers to the questions

4.1 Enumerate the factors that influence the algorithms' time of execution:

- Input data;
- Quality of the code compiled by the compiler;
- Complexity of the algorithm;
- Speed of execution of the commands;

4.2 What are the stages of an empiric analyse?

- Set the goal;
- Select the unit of measure of efficiency (operations, seconds);
- Set the properties of the input data;
- Implement the algorithm;
- Generate a set of input data;
- Execute the program for each set;
- Analyse the output data.

4.3 What are the cases when we need an empiric analyse?

- To compare multiple algorithms with the same goal;
- To compare multiple implementations of the same algorithm;
- To check the efficiency of an algorithm on multiple PC's;

5.Conclusion

In this lab I had to research different ways of finding the nth term of Fibonacci series; I studied the algorithms that I found, implemented them and then analysed them with the same sets of input. This way, I found out what’s their time and space complexity, and which one would be more reasonable to use.