APA laboratory_01

Terman Emil FAF161

October 11, 2017

UNIVERSITATEA TEHNICĂ A MOLDOVEI

Prof: M. Catruc

LaTeX

Subject: Algorithm analyzing

Purpose:

- Analiza empirică a algoritmilor.
- Analiza teoretică a algoritmilor.
- Determinarea complexităţii temporale şi asimptotice a algoritmilor

Conditions:

1. Efectuaţi analiza empirică a algoritmilor propuşi.
2. Determinaţi relaţia ce determină complexitatea temporală pentru aceşti algoritmi.
3. Determinaţi complexitatea asimptotică a algoritmilor.
4. Faceţi o concluzie asupra lucrării efectuate.

1 Recursive method

Algorithm 1: Recursive method

```
1 function fib1(n)
2     if n < 2 then
3         return n
4     else
5         return fib1(n - 1) + fib1(n - 2)
6
```

T(n) - ?

For line 2 and 3: O(1)
For line 5: T(n - 1) + T(n - 2)
So:
T(n) = 2, n $<$ 2
T(n) = T(n - 1) + T(n - 2) + 3 $\approx$ T(n - 1) + T(n - 2), n $\geq$ 2

$$t_n - t_{n-1} - t_{n-2} = 0$$
$$x^2 - x - 1 = 0$$

$$\left[\begin{array}{l} x_1 = \frac{1-\sqrt{5}}{2} \\ x_2 = \frac{1+\sqrt{5}}{2} \end{array}\right.$$

$$t_n = C_1(\tfrac{1-\sqrt{5}}{2})^n + C_2(\tfrac{1+\sqrt{5}}{2})^n$$

The fraction: $\frac{1+\sqrt{5}}{2}$ is also known as the *Golden Ratio* denoted as φ. The most significant part of t_n is φ, thus:

$$T(n) = O(\varphi^n)$$

This method is very uneficient since it has to recalculate multiple timess the same values. We can clearly see why it's not a good idea to use this algorithm:

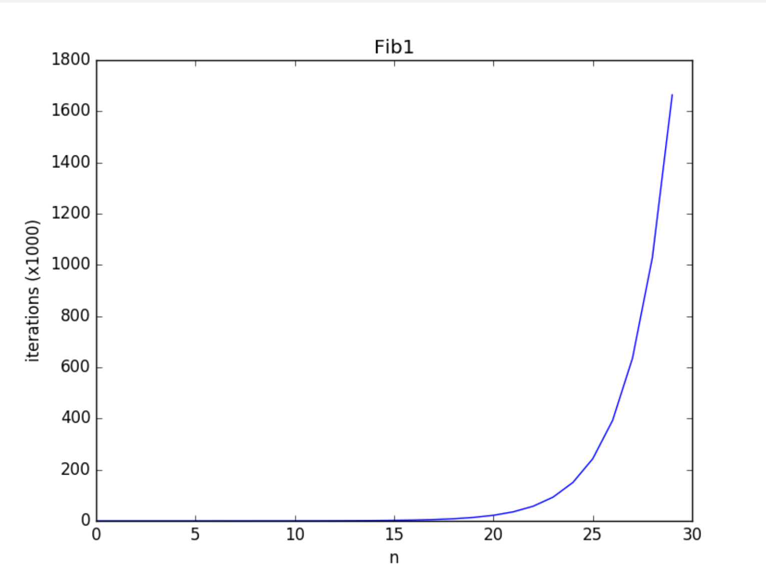

Figure 1: Algorithm 1

2 Iterative method

Algorithm 2: Iterative method

```
1 function fib2(n)
2     i ← 1;
3     j ← 0;
4     for k ← 1 to n do
5         j ← i + j;
6         i ← j − i;
7     return j
8
```

For line 2 and 3, the time is O(1).
Line 5 and 6 are executed n times, therefore the time for both of these lines will be $2 \cdot$ n.
We consider line 4 to be executed n times.

$$t_n = 2 + n + 2n$$

Therefore:

$$T(n) = O(n)$$

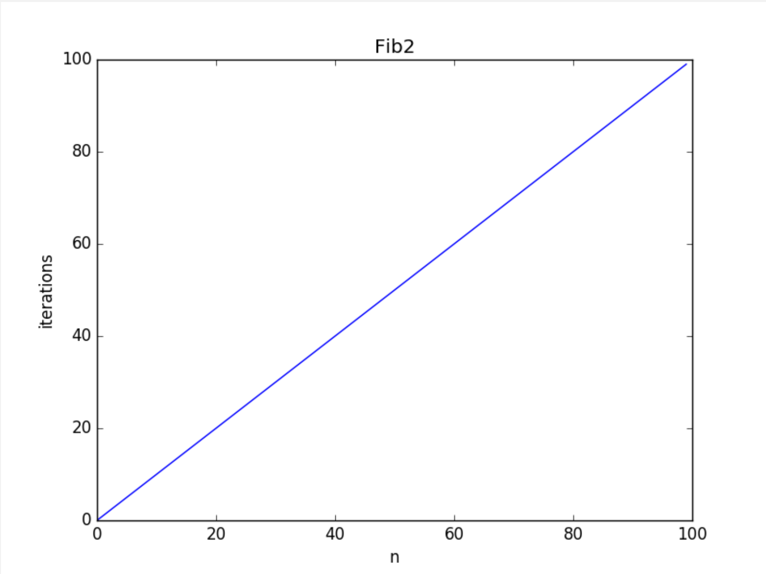

Figure 2: Algorithm 2

3 Logarithmic method

Algorithm 3: Logarithmic method

```
1  function fib3(n)
2      i ← 1;
3      j ← 0;
4      k ← 0;
5      h ← 1;
6      while n > 0 do
7          if n mod 2 == 1 then
8              t ← j · h;
9              j ← i · h + j · k + t;
10             i ← i · k + t;
11         t ← h · h;
12         h ← 2 · k · h + t;
13         k ← k · k + t;
14         n ← n div 2;
15     return j
16
```

From the line 14, we can see that the while loop will be executed $\log_2 n$ times, due to the operation:

$$n \leftarrow n \ div \ 2$$

Therefore:

$$T(n) = log_2 n$$

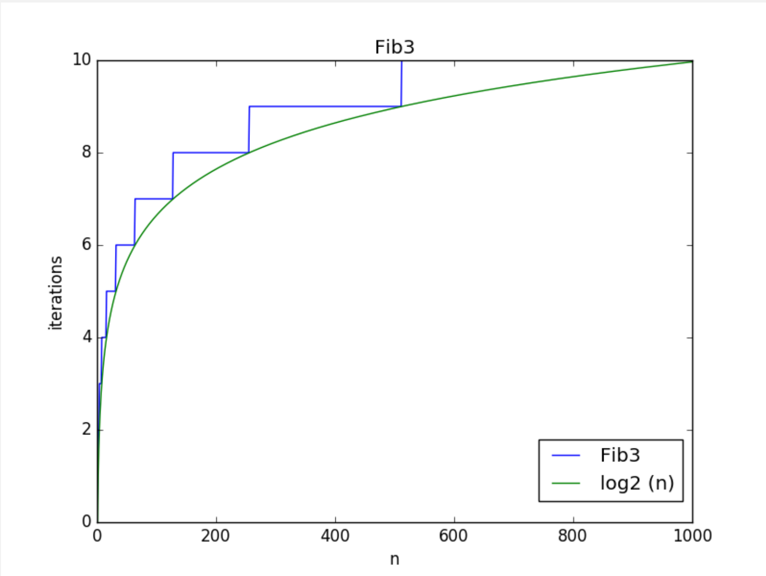

Figure 3: Algorithm 3

Those steps are generated when the *if* statement is executed.

4 Iterative with saved values

Algorithm 4: Iterative with saved values

```
1 function fib4(n)
2     declare fibVals : array[1, n + 2] of int;
3
4     fibVals[0] ← 0;
5     fibVals[1] ← 1;
6     for i ← 2 to n + 1 do
7         fibVals[i] = fibVals[i − 1] + fibVals[i − 2];
8     return fibVals[n];
9
```

- For the line 2, we declare (n + 2) · *sizeof(int)* = (n + 2) * 4 *Bytes* of memory. That's roughly 4 · n *Bytes*.
- Line 4 and 5 have a complexity of O(n).
- The line 7 has 4 operations, that is another O(n).
- The *for* loop is executed $n - 1$ times. So the inside part of the loop will have $4 \cdot (n - 1)$ operations. Resulting that the entire loop will have complexity of O(n).

Since the most significant part is the *for* loop, results that:

$$T(n) = O(n)$$

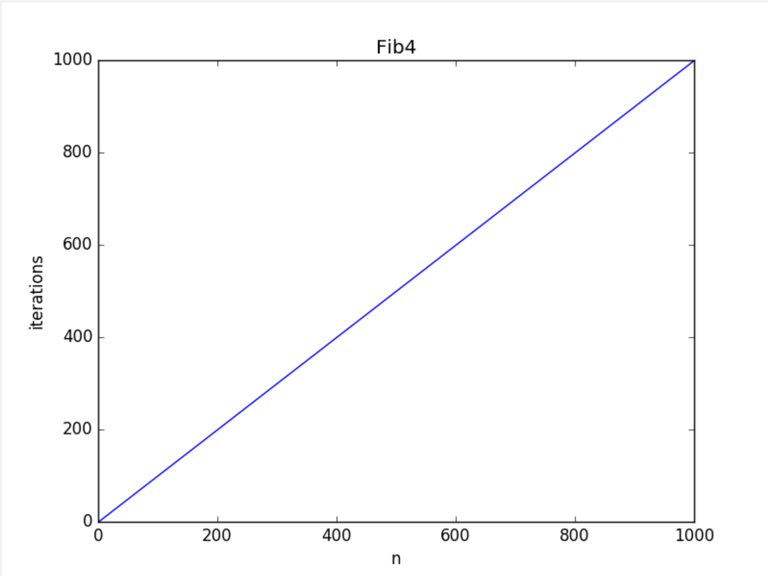

Figure 4: Algorithm 4

This method has a very big drawback: it needs more memory than other algorithms.

5 Conclusion

- The fastest and the most eficient Fibonacci algorithm is the $O(\log_2 n)$ algorithm.
- Sometimes, it's enough to have a very simple implemented algorithm, but less efficient, because we don't always need a huge performance.
- The calculation of the time complexity of an algorithm helps us to choose which is the best algorithm of all.

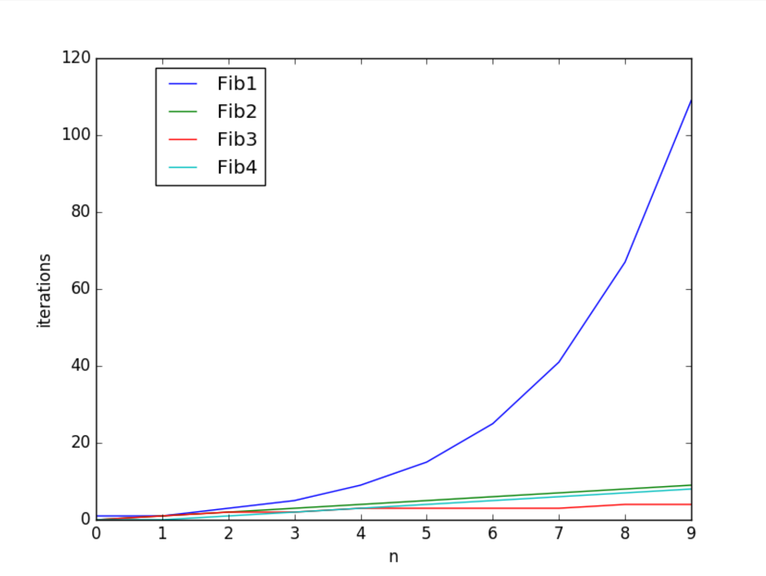

Figure 5: All algorithms

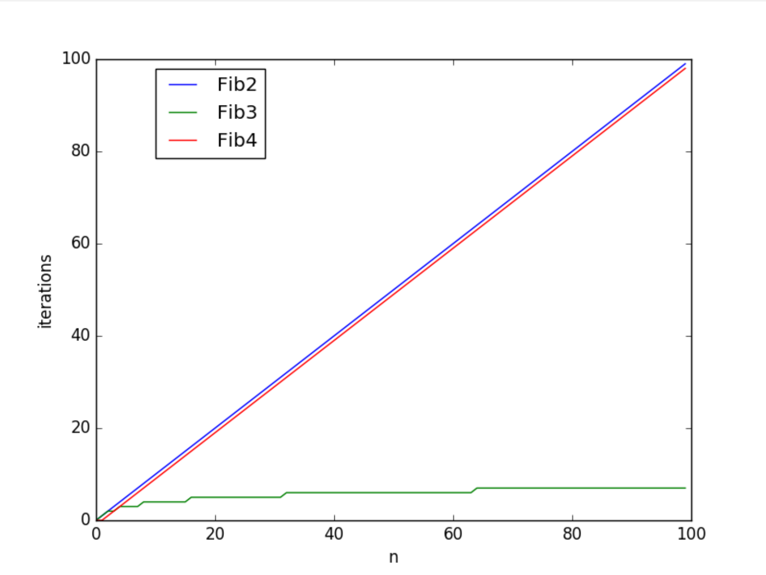

Figure 6: All algorithms but factorial