

Universidade Federal de Mato Grosso do Sul

Campus Ponta Porã
Análise de Algoritmos I

Trabalho Prático I
Análise Assintótica

Aluno: Daniel de Leon Bailo da Silva
Professor: Eduardo Theodoro Bogue

Abril
2019

Sumário

1	Resumo	1
2	Fibonacci Recursivo	2
3	Fibonacci Iterativo	5
4	Busca Binária	6
5	Apêndice I	8
6	Conclusão	13

1 Resumo

2 Fibonacci Recursivo

```
1 def fib(n):  
2     if n==1 or n==2:  
3         return 1  
4     return fib(n-1)+fib(n-2)
```

Listing 1: Código da função do Fibonacci Recursivo

Temos que a recorrência do Fibonacci Recursivo pode ser escrita como:

$$T(n) = \begin{cases} O(1), n = 0, n = 1 \\ fib(n-1) + fib(n-2), n > 1 \end{cases} \quad (1)$$

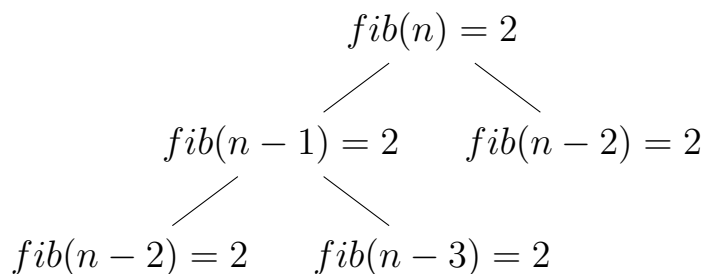
Número de operações

se $n = 0, n = 1$

Temos que o número de operações é igual a $fib(n) = 2$

se $n > 1$

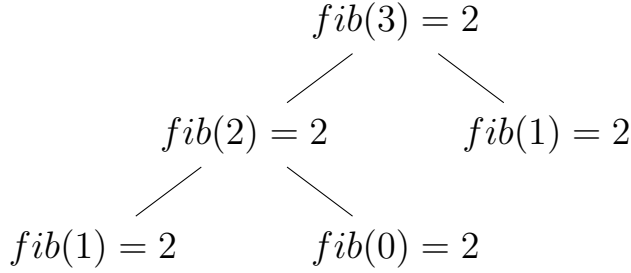
Temos que o número de operações é igual a $fib(n) = 2$



E assim sucessivamente até chegar no caso base. Temos que essa árvore, sempre crescerá mais para a esquerda. No pior caso, teremos que a cada nível da árvore, o número de operações é multiplicado por 2. Pois, cada nó terá dois filhos, e cada filho gasta 2 operações. Ou seja, podemos representar o custo por nível como:

$\{2, 4, 8, 16, 32, 64, \dots\}$

E que o tamanho dessa sequência, depende do n escolhido, temos como um exemplo $n = 3$:



Ou seja, temos que o tamanho da sequência é igual a n . Podemos afirmar que a sequência formada pelo número de operações é uma Progressão Geométrica. Logo, para descobrirmos o custo total dessa sequência, basta realizarmos a soma dos termos de uma PG . A soma dos termos de uma PG é representada como:

$$S_n = \frac{(q^n - 1)}{q - 1} \quad (2)$$

Onde q é a razão de um termo para outro e n é a quantidade termos da PG .

$$= \frac{(2^n - 1)}{2 - 1} \rightarrow \frac{2^n - 1}{1} \rightarrow 2^n - 1 + O(1) \rightarrow 2^n + 1 \quad (3)$$

Temos que $f(n) = 2^n + 1$. Para mostrar que $f(n) = O(2^n)$. Basta provarmos que para uma dada função $g(n)$, denotamos por $O(g(n))$ o conjunto de funções tal que:

$$O(g(n)) = \{f(n) : \exists c \wedge \exists n_0 \mid 0 \leq f(n) \leq cg(n) \forall n \geq n_0\} \quad (4)$$

$$0 \leq f(n) \leq cg(n) \quad (5)$$

$$0 \leq 2^n + 1 \leq c.2^n \quad (6)$$

$$0 \leq \frac{(2^n + 1)}{2^n} \leq c \quad (7)$$

$$0 \leq 1 + \frac{1}{2^n} \leq c \quad (8)$$

Portanto, temos que $c = 2$ para um $n_0 = 0$, pois quanto maior for o valor de n , temos que o resultado final tende a 1. Logo, assim provamos que para qualquer valor de n , com n começando de 0, nossa constante c será maior que o valor final da expressão. Então, temos que $f(n) = O(2^n)$

3 Fibonacci Iterativo

Para o Fibonacci Iterativo, temos o seguinte código da função e seu respectivo custo/linha e quantas vezes cada linha é executada.

```

1 def fib(n):
2     a, b = 0, 1
3     for _ in range(0, n):
4         a, b = b, a + b
5     return a

```

Listing 2: Código da função do Fibonacci Iterativo

Linha	Custo	Vezez
2	c_1	1
3	c_2	n
4	c_3	n-1
5	c_4	1

$$T(n) = c_1 1 + c_2 n + c_3 (n - 1) + c_4 1 \quad (9)$$

$$T(n) = c_1 1 + c_2 + c_3 n - c_3 + c_4 \quad (10)$$

$$T(n) = (c_2 + c_3)n + c_1 + c_4 - c_3 \quad (11)$$

Logo, temos que $T(n)$ pode ser expresso como $an + b$. Assim, podemos afirmar que $T(n)$ se comporta como uma função linear.

Para mostrar que $f(n) = O(n)$ basta provar o mesmo processo das equações (4) e (5).

$$0 \leq f(n) \leq cg(n) \quad (12)$$

$$0 \leq an + b \leq cn \quad (13)$$

$$0 \leq a + \frac{b}{n} \leq c \quad (14)$$

Logo, temos que $c = a + b$ para $n_0 = 1$ e $c = a$ para $n_0 = \infty$. Portanto $c = a + b$ e $f(n) = O(n)$.

4 Busca Binária

Para a Busca Binária Recursiva, temos a seguinte função para realizar a busca em um dado vetor ordenado.

```

1 def BBRecurs(A, p, r, x):
2     if p == r-1:
3         return r
4     else: q = (p+r)/2
5         if A[q] < x
6             return BBRecurs(A, p, r, x)
7         else return BBRecurs(A, p, r, x)

```

Listing 3: Código da função da Busca Binária

Linha	Custo	Veze
2	c_1	1
3	c_2	1
4	c_3	1
5	c_4	$T((n-1)/2)$
6	c_5	$T((n-1)/2)$

Segue daí que o tempo de pior caso, $T(n)$, é definido pela recorrência:

$$T(n) = \begin{cases} T(0) = O(1) \\ T(n) = T((n-1)/2) + O(1) \end{cases} \quad (15)$$

Para mostrar o custo da recorrência da Busca Binária, basta aplicarmos o Teorema Mestre, onde

$$T(n) = aT\left(\frac{n}{b}\right) + f(n); a \leq 1, b > 1 \wedge f(n) > 0, \forall n > n_0. \quad (16)$$

temos que para $T(n)$ que

$$\begin{aligned} f(n) &= 1 \\ n^{\log_b a} &= n^{\log_2 1} = n^0 = 1 \end{aligned}$$

Temos que para esses valores, utilizaremos o caso 2 do Teorema Mestre que atribui o seguinte:

Se temos a seguinte igualdade, $f(n) = \theta(n^{\log_b a})$, vale a solução:

$$T(n) = \theta(n^{\log_b a} \log n)$$

onde

$$T(n) = \theta(1 \log n) \rightarrow T(n) = \theta(\log n)$$

Portanto, temos que $T(n) = \theta(\log n)$.

5 Apêndice I

Neste apêndice, será mostrado o código completo de execução do Fibonacci Recursivo e Iterativo, e além disso, para cada código, será calculado o tempo de execução para o n -ésimo termo da sequência e em seguida um gráfico de comparação entre os dois códigos.

Cada código gera calcula 16 valores para um dado n , as variáveis de t_0 à t_{15} armazenam o tempo de execução de cada $fib(n)$ onde fib é a função que retorna o termo da sequência para um dado n .

Após realizar o cálculo dos valores, tendo esses valores e o tempo de execução para cada valor, cada código gera um gráfico de tempo em função de $fib(n)$. Os gráficos gerados pelos algoritmos serão mostrados mais abaixo no tópico de **Análise Gráfica**.

Para executar os algoritmos, basta copiar o código fonte e salvar um arquivo com extensão .py e realizar a chamada pelo arquivo no terminal/prompt com o python3.

Fibonacci Recursivo

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from timeit import timeit
4
5 def fib(n):
6     if n==1 or n==2:
7         return 1
8     return fib(n-1)+fib(n-2)
9
10 if __name__ == "__main__":
11     t0 = timeit('fib(1)', 'from __main__ import fib', number=1)
12     t1 = timeit('fib(3)', 'from __main__ import fib', number=1)
13     t2 = timeit('fib(6)', 'from __main__ import fib', number=1)
14     t3 = timeit('fib(9)', 'from __main__ import fib', number=1)
15     t4 = timeit('fib(12)', 'from __main__ import fib', number=1)
16     t5 = timeit('fib(15)', 'from __main__ import fib', number=1)
17     t6 = timeit('fib(18)', 'from __main__ import fib', number=1)
18     t7 = timeit('fib(21)', 'from __main__ import fib', number=1)
```

```

19 t8 = timeit('fib(23)', 'from __main__ import fib', number=1)
20 t9 = timeit('fib(24)', 'from __main__ import fib', number=1)
21 t10 = timeit('fib(27)', 'from __main__ import fib', number=1)
22 t11 = timeit('fib(30)', 'from __main__ import fib', number=1)
23 t12 = timeit('fib(33)', 'from __main__ import fib', number=1)
24 t13 = timeit('fib(36)', 'from __main__ import fib', number=1)
25 t14 = timeit('fib(39)', 'from __main__ import fib', number=1)
26 t15 = timeit('fib(42)', 'from __main__ import fib', number=1)
27
28 y = np.array([t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13,
29              t14, t15])
30
31 x = np.array([1, 3, 6, 9, 12, 15, 18, 21, 23, 24, 27, 30, 33, 36, 39, 42])
32
33 yp = None
34
35 xi = np.linspace(x[0], x[-1], 100)
36 yi = np.interp(xi, x, y, yp)
37
38 fig, ax = plt.subplots()
39 ax.plot(x, y, 'o', xi, yi,)
40
41 plt.grid()
42 plt.title('Taxa de Crescimento do Fibonacci Recursivo')
43 plt.xlabel('Fibonacci(x)')
44 plt.ylabel('Tempo(segundos)')
45 plt.savefig('fibonacci_recursivo.png')
46
47 plt.show()

```

Fibonacci Iterativo

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from timeit import timeit
4
5 def fib(n):
6     a, b = 0, 1
7     for _ in range(0, n):
8         a, b = b, a + b
9     return a
10
11 if __name__ == "__main__":
12     t0 = timeit('fib(10)', 'from __main__ import fib', number=1)
13     t1 = timeit('fib(100)', 'from __main__ import fib', number=1)
14     t2 = timeit('fib(200)', 'from __main__ import fib', number=1)
15     t3 = timeit('fib(300)', 'from __main__ import fib', number=1)
16     t4 = timeit('fib(400)', 'from __main__ import fib', number=1)
17     t5 = timeit('fib(500)', 'from __main__ import fib', number=1)
18     t6 = timeit('fib(600)', 'from __main__ import fib', number=1)

```

```

19 t7 = timeit('fib(650)', 'from __main__ import fib', number=1)
20 t8 = timeit('fib(750)', 'from __main__ import fib', number=1)
21 t9 = timeit('fib(850)', 'from __main__ import fib', number=1)
22 t10 = timeit('fib(905)', 'from __main__ import fib', number=1)
23 t11 = timeit('fib(1050)', 'from __main__ import fib', number
    =1)
24 t12 = timeit('fib(1200)', 'from __main__ import fib', number
    =1)
25 t13 = timeit('fib(1300)', 'from __main__ import fib', number
    =1)
26 t14 = timeit('fib(1400)', 'from __main__ import fib', number
    =1)
27 t15 = timeit('fib(1500)', 'from __main__ import fib', number
    =1)
28
29 y = np.array([t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,
    t14,t15])
30 x = np.array([10,100,200,300,400,500,600,650,750,
31 850,905,1050,1200,1300,1400,1500])
32
33 yp = None
34
35 xi = np.linspace(x[0], x[-1],100)
36 yi = np.interp(xi, x, y, yp)
37
38 fig, ax = plt.subplots()
39 ax.plot(x, y, 'o', xi, yi,)
40
41 plt.grid()
42 plt.title('Taxa de Crescimento do Fibonacci Iterativo')
43 plt.xlabel('Fibonacci(x)')
44 plt.ylabel('Tempo(segundos)')
45 plt.savefig('fibonacci_iterativo.png')
46
47 plt.show()

```

Análise Gráfica

Fibonacci Recursivo

Tendo visto que a complexidade do algoritmo do Fibonacci Recursivo é $O(2^n)$, podemos ver que seu custo computacional é muito alto. Como podemos analisar nos gráficos, quando $n = 39$ e $n = 42$, seu tempo de execução em segundos aumenta drasticamente para 3 unidades a mais de n soment.

Mesmo para valores pequenos de n , sua execução demanda muito tempo. Na computação, é desejável que resultados sejam alcançados da forma mais rápida possível, sendo inviável a utilização deste algoritmo para determinado cálculo sabendo que existe formas mais rápidas para encontrar os valores da sequência.

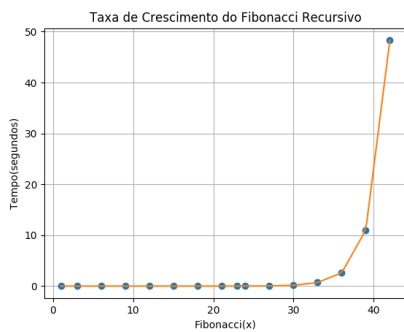


Figura 1: Fibonacci Recursivo

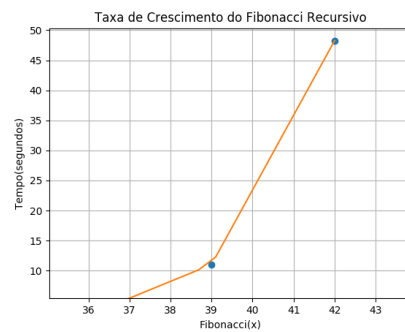


Figura 2: Ampliado num Intervalo

Fibonacci Iterativo

Sabendo que a complexidade do Fibonacci Iterativo é $O(n)$, onde este se comporta como uma função linear, é fácil visualizar que seu custo computacional é muito mais leve do que uma função exponencial.

Olhando para o gráfico, podemos ver que os valores de n são muito maiores do que os valores utilizados no Fibonacci Recursivo, e este algoritmo retorna o valor desejado praticamente instantaneamente. Se tivesse testado esses valores de n no Fibonacci Recursivo, além de muito processamento exigido, poderia demorar horas, quem sabe dias para retornar o resultado desejado.

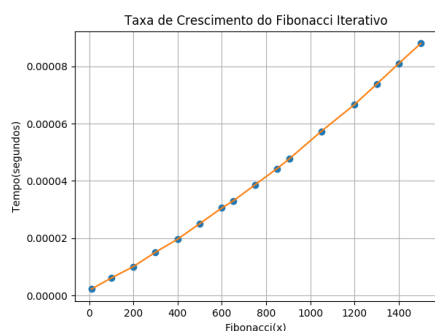


Figura 3: Fibonacci Iterativo

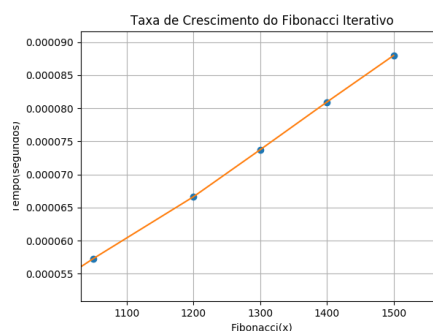


Figura 4: Ampliado num Intervalo

6 Conclusão

Portanto, é possível concluir que, para um dado problema, quanto menor possível for a complexidade para resolução do mesmo, melhor. É claro que, este é um exemplo simples e muito conhecido na computação, porém, é muito interessante para estudantes da computação que ainda não tem o conhecimento de Análise Assintótica e não sabem o que é e como se comporta a complexidade de um algoritmo a leitura deste trabalho, pois provavelmente a maioria dos alunos dos cursos de computação já escreveram um programa para mostrar a *Sequência de Fibonacci*, então seria um exemplo de fácil compreensão.

Existe algoritmos melhores para retornar o n -ésimo número desejado desta sequência em $O(\log n)$. Porém, este foi somente um trabalho para comparar estes dois algoritmos considerando que eu já deveria realizar a Análise Assintótica de cada um como trabalho proposto pelo professor ministrante da disciplina.

Abaixo estão as figuras lado a lado dos dois algoritmos comparados, ampliadas num intervalo para a melhor visualização e comparação de tempo em relação ao n -ésimo termo da sequência.

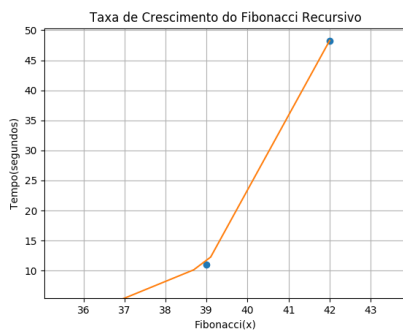


Figura 5: Fibonacci Recursivo Ampliado

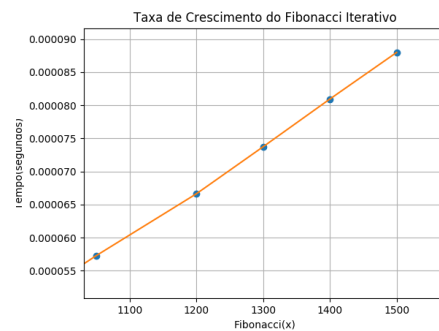


Figura 6: Fibonacci Iterativo Ampliado