

Complexidade de Algoritmos

Prof. Diego Buchinger
diego.buchinger@outlook.com
diego.buchinger@udesc.br

Prof. Cristiano Damiani Vasconcellos
cristiano.vasconcellos@udesc.br

Análise de Complexidade de Tempo de Algoritmos Recursivos

Algoritmos Recursivos

Geralmente a análise de complexidade de algoritmos recursivos é um pouco mais complexa pois é preciso entender bem a recursividade do algoritmo.

- Quantas vezes um algoritmo chama a si mesmo?
- Existe um pior caso? Quantas chamadas recursivas são feitas?
- Devemos considerar o tempo gasto na chamada de função? E o espaço utilizado na memória?

Exemplo (Fatorial)

```
int fatorial( int n ){  
    if (n == 0)  
        return 1;  
    return n * fatorial( n-1 );  
}
```

Exemplo (Fatorial)

```
int fatorial( int n ){  
    if (n == 0)  
        return 1;  
    return n * fatorial( n-1 );  
}
```

Relação de Recorrência:

$$T(n) = T(n-1) + O(1)$$

$$T(0) = O(1)$$

Exemplo (Pesquisa Binária)

```
int pesqbin(int *v, int p, int r, int e){  
    int q;  
    if ( r < p )  
        return -1;  
    q = (p + r)/2;  
    if (e == v[q])  
        return q;  
    if (e < v[q])  
        return pesqbin(v, p, q-1, e);  
    return pesqbin(v, q+1, r, e);  
}
```

** Lembrete: o vetor deve estar ordenado!!*

Exemplo (Pesquisa Binária)

```
int pesqbin(int *v, int p, int r, int e){  
    int q;  
    if ( r < p )  
        return -1;  
    q = (p + r)/2;  
    if (e == v[q])  
        return q;  
    if (e < v[q])  
        return pesqbin(v, p, q-1, e);  
    return pesqbin(v, q+1, r, e);  
}
```

Relação de Recorrência:

$$T(n) = T(n/2) + O(1)$$

$$T(1) = O(1)$$

A relação abaixo descreve de forma mais precisa a execução, mas a simplificação acima não altera a complexidade:

$$T(n) = T(n/2 - 1) + O(1)$$

$$T(1) = O(1)$$

Relação de Recorrência

$$T(n) = T(n/2) + 1$$

$$T(n/2) = T(n/2^2) + 1$$

$$T(n/2^2) = T(n/2^3) + 1$$

...

...

$$T(n/2^{h-1}) = T(n/2^h) + 1$$

Relação de Recorrência:

$$T(n) = T(n/2) + O(1)$$

$$T(1) = O(1)$$

Relação de Recorrência

$$T(n) = T(\cancel{n/2}) + 1$$

$$\cancel{T(n/2)} = T(\cancel{n/2^2}) + 1$$

$$\cancel{T(n/2^2)} = T(\cancel{n/2^3}) + 1$$

...

$$\cancel{T(n/2^{h-1})} = T(n/2^h) + 1$$

Relação de Recorrência:

$$T(n) = T(n/2) + O(1)$$

$$T(1) = O(1)$$

$$T(n) = \underbrace{1 + 1 + \dots + 1}_{h \text{ vezes}} + T(1)$$

h vezes => ops... Mas quanto vale h??

Relação de Recorrência

$$T(n) = T(\cancel{n/2}) + 1$$

$$\cancel{T(n/2)} = T(\cancel{n/2^2}) + 1$$

$$\cancel{T(n/2^2)} = T(\cancel{n/2^3}) + 1$$

...

$$\cancel{T(n/2^{h-1})} = T(n/2^h) + 1$$

$$\frac{n}{2^h} = 1$$

$$\begin{aligned} n &= 2^h \\ h &= \log_2 n \end{aligned}$$

$$T(n) = \underbrace{1 + 1 + \dots + 1}_{h \text{ vezes}} + T(1)$$

h vezes, ou seja, $\log_2 n$ vezes

$$O(\log n)$$

Mudança de base:

$$\log_b n = \frac{\log_a n}{\log_a b}$$

Exemplo (Pesquisa Binária)

```
int pesqbin2 (int *v, int n, int e) {  
    int p, q, r;  
  
    p = 0; r = n-1;  
    do {  
        q = (p + r) / 2;  
        if (e == v[q])  
            return q;  
        if (e < v[q])  
            r = q - 1;  
        else  
            p = q + 1;  
    } while (p <= r);  
    return -1;  
}
```

Recursividade de cauda

Uma função apresenta recursividade de cauda se nenhuma operação é executada após o retorno da chamada recursiva, exceto retornar seu valor.

Em geral, compiladores, que executam otimizações de código, substituem as funções que apresentam recursividade de cauda por uma versão não recursiva dessa função.

Exemplo (Fibonacci)

```
/* Implementação ruim */  
int fib( int n ){  
    if( n == 0 || n == 1 )  
        return 1;  
    return fib( n-1 ) + fib( n-2 );  
}
```

Exemplo (Fibonacci)

```
/* Implementação ruim */  
int fib( int n ){  
    if( n == 0 || n == 1 )  
        return 1;  
    return fib( n-1 ) + fib( n-2 );  
}
```

Relação de Recorrência:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

$$T(1) = O(1)$$

$$T(0) = O(1)$$

Exemplo (Fibonacci)

```
/* Implementação ruim */  
int fib( int n ){  
    if( n == 0 || n == 1 )  
        return 1;  
    return fib( n-1 ) + fib( n-2 );  
}
```

Relação de Recorrência simplificada:

$$T(n) = 2T(n-1) + O(1)$$

$$T(0) = O(1)$$

Relação de Recorrência

$$T(n) = 2T(n-1) + O(1)$$

$$2T(n-1) = 2^2T(n-2) + 2O(1)$$

$$2^2T(n-2) = 2^3T(n-3) + 2^2O(1)$$

...

...

$$2^{n-1}T(1) = 2^nT(n-n) + 2^{n-1}O(1)$$

$$2^nT(0) = 2^n O(1)$$

$$T(n) = \sum_{i=0}^n 2^i = 2^{n+1} - 1$$

$$O(2^n)$$

Relação de Recorrência:

$$T(n) = 2T(n-1) + O(1)$$

$$T(0) = O(1)$$

Exemplo (Fibonacci)

Ao usar a relação de recorrência correta
chegaríamos em uma resposta parecida:

Relação de Recorrência:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

$$T(1) = O(1)$$

$$T(0) = O(1)$$

$$O(\varphi^n)$$

$$\varphi = \frac{1 + \sqrt{5}}{2}$$

$$\varphi \cong 1,6180$$

Propriedades dos Somatórios

$$\sum_{i=0}^n ca_i = c \sum_{i=0}^n a_i \quad (\text{Distributiva})$$

$$\sum_{i=0}^n (a_i + b_i) = \sum_{i=0}^n a_i + \sum_{i=0}^n b_i \quad (\text{Associativa})$$

$$\sum_{i=n}^0 a_i = \sum_{i=0}^n a_i \quad (\text{Comutativa})$$

Propriedades dos Somatórios

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1}$$

$$\sum_{i=1}^{\log n} i = n \log n$$

$$\sum_{i=0}^n 2^i = 2^{n+1} - 1$$

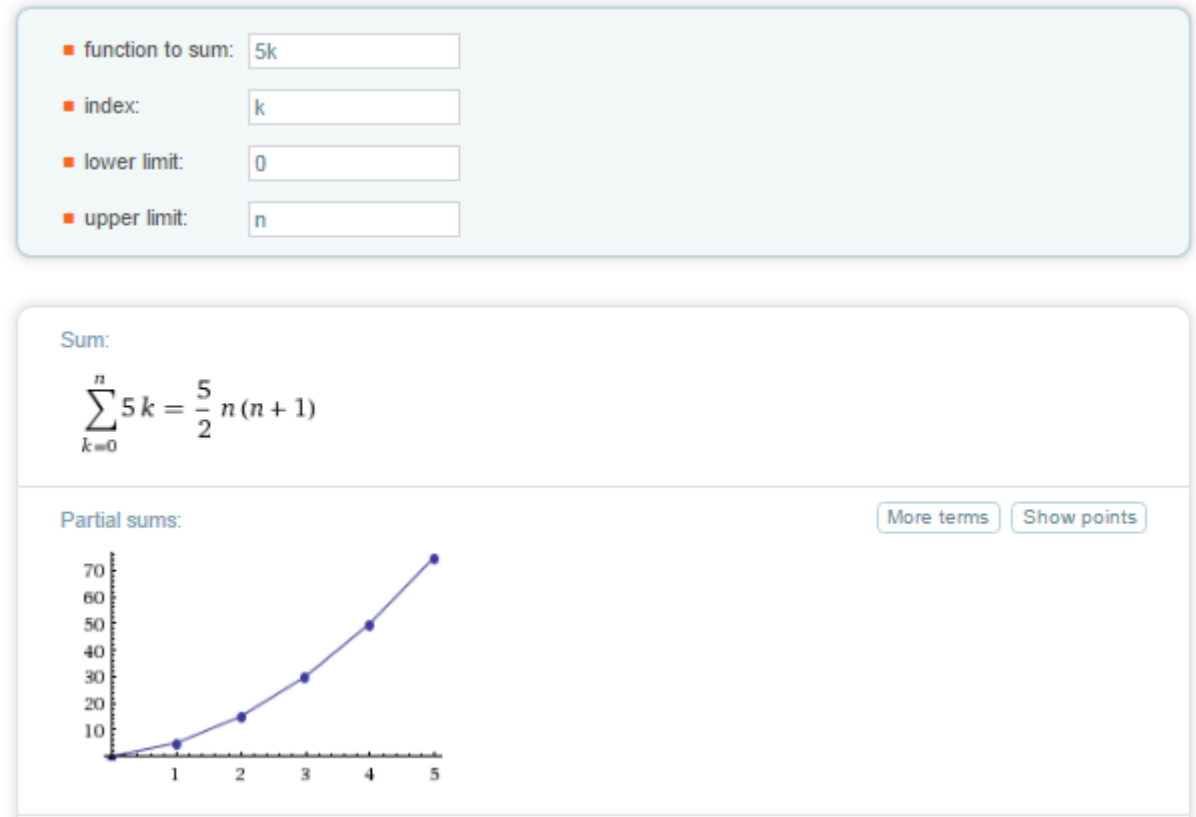
$$\sum_{i=0}^{\log n} 2^i = 2n - 1$$

$$\sum_{i=1}^n \log i = \log(n!)$$

$$\frac{1}{2}n \log n \leq \log(n!) \leq n \log n \quad \therefore \quad \log(n!) = \Theta(n \log n)$$

Somatórios

Existem algumas ferramentas que calculam as fórmulas dos somatórios e até mesmo geram gráficos.



Ex: <https://www.wolframalpha.com/input/?i=sum>

Propriedades das Potências

$$a^m a^n = a^{m+n}$$

$$\frac{a^m}{a^n} = a^{m-n}, a \neq 0$$

$$(ab)^n = a^n b^n$$

$$\left(\frac{a}{b}\right)^n = \frac{a^n}{b^n}, b \neq 0$$

$$(a^m)^n = a^{nm}$$

Propriedades dos Logaritmos

$$a^b = c \Leftrightarrow \log_a c = b$$

$$a^{\log_a b} = b$$

$$\log_c (ab) = \log_c a + \log_c b$$

$$\log_b a^c = c \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$a^{\log_b c} = c^{\log_b a}$$

Atividade

Escreva duas versões do algoritmo de fibonacci: a versão “ruim” apresentada e uma versão “boa” usando vetores. Faça uma comparação de tempo de execução com valores entre 25 e 60.

Qual a complexidade da versão “boa” do algoritmo?

Atividade

Elabore um algoritmo recursivo para calcular a raiz quadrada de um número real (positivo) qualquer. Após implementar o algoritmo, escreva a relação de recorrência e a respectiva complexidade do algoritmo para o pior caso.

*Lembrete: comparação de números reais através de aproximação $\Rightarrow |x - 3.2| \leq 0.00001$

*OBS: $r = \sqrt{n}$ quando $n \geq 1$, $1 \leq r \leq n$
 quando $n < 1$, $n \leq r \leq 1$

Atividade

Resolva as seguintes recorrência por substituição:

(Utilize os somatórios conhecidos para chegar a um resultado final ou utilize a ferramenta apresentada)

a. $T(n) = T(n-1) + \Theta(\log n)$

$$T(1) = \Theta(\log 1)$$

b. $T(n) = 3T(n-1) + \Theta(n)$

$$T(1) = \Theta(1)$$

c. $T(n) = T(\sqrt{n}) + \Theta(1)$

$$T(2) = \Theta(1)$$

Atividade

Resolva os seguintes somatórios:

$$a) \sum_{i=0}^n (2i + 1)$$

$$c) \sum_{i=0}^{\log_2 n} 2^i$$

$$b) \sum_{i=0}^n (ni - 5i)$$

$$d) \sum_{i=0}^{\sqrt{n}} i$$