Projeto e Análise de Algoritmos

Prof. Dr. Ednaldo B. Pizzolato

www.dc.ufscar.br/~ednaldo

ednaldo@dc.ufscar.br

BIG O

Teoria

Seja T(n) o tempo de execução de um programa, medido em função do tamanho do conjunto de entrada n.

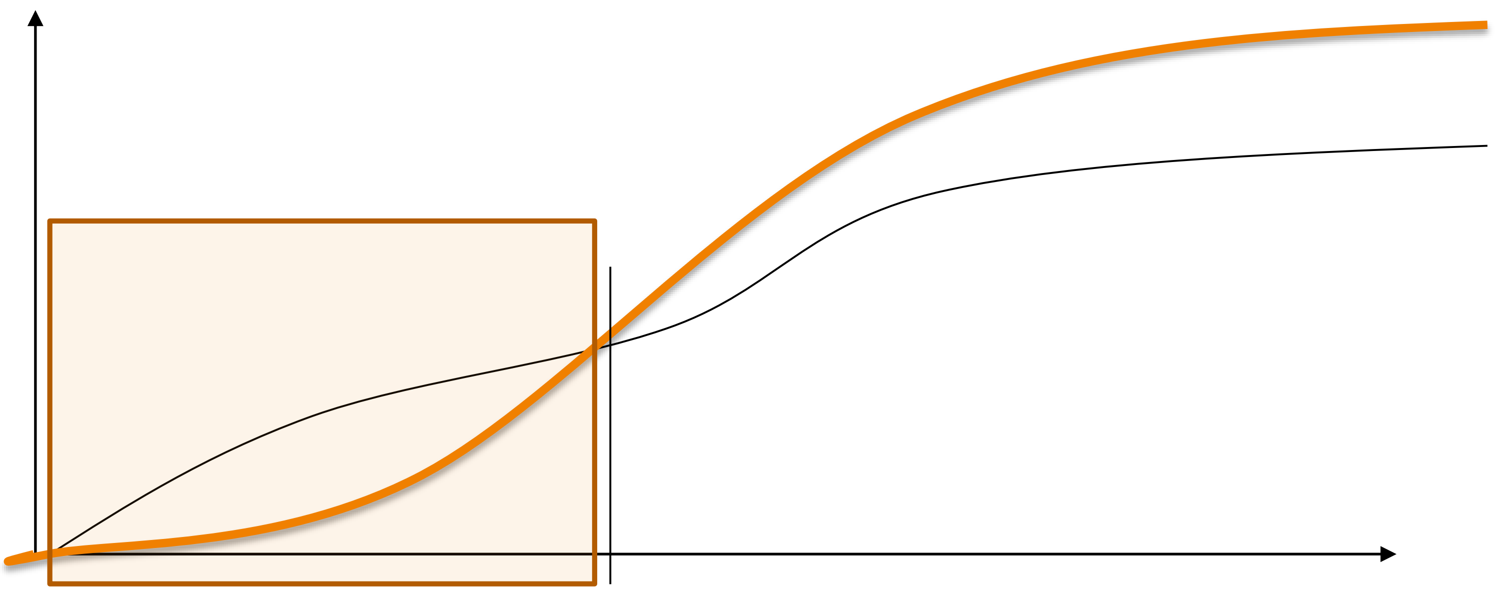
→ n é um valor não negativo

→ T(n) é um valor não negativo p/ qquer n

Seja f(n) outra função também definida no conjunto dos inteiros não negativos. Se T(n) é no máximo uma constante vezes f(n) – <u>excetuando-se, possivelmente, alguns valores pequenos de n</u> – pode-se dizer que T(n) é O(f(n)).

Teoria

- Definição formal
  - T(n) é O(f(n)) se existe um inteiro n_0 e uma constante c (c > 0) | $\forall$ n > n_0 → T(n) ≤ c. f(n)



Exemplo

Suponha que tenhamos um programa que tenha os seguintes tempos de execução:

$T(0) = 1; \quad T(1) = 4; \quad T(2) = 9$

em geral podemos resumir em $T(n)=(n+1)^2$

Assim, teríamos que $T(n)$ é $O(n^2)$

Prova: Pela definição tem-se que se escolhermos uma constante c e um valor n_0 tais que $T(n) \leq c \cdot n^2$, então $T(n)$ será $O(n^2)$.

Para $c = 4$ e $n_0 > 1$ temos esta relação.

Exemplo

função $f(n)=(n+1)^2$

$n^2 + 2.n + 1 \quad \leq n^2 + 2.n^2 + n^2$

$\leq 4 . n^2$

Princípios Gerais

1. Constantes não são importantes

Se $T(n)$ é $O(d\,.T(n))$ para qualquer $d > 0$

e se

$$n_0 = 0 \text{ e } c \geq 1/d$$

Então

$$\begin{aligned} T(n) &\leq c\,.\,(d\,.\,T(n)) \\ &\leq c\,.\,d\,.\,T(n) \\ &\leq 1\,.\,T(n) \end{aligned}$$

Princípios Gerais

2. Termos de ordem menor não são importantes.

Seja T(n) um polinômio da forma:

$$a_k.n^k + a_{k-1}.n^{k-1} + \ldots + a_1.n + a_0 \quad (a_k>0)$$

Seja $n_0 = 1$ e $c = \sum a_i$ se $a_i > 0$

→ T(n) não é superior a c.T(n)

Exemplo: $2 \cdot n^3$ é $O(0.001 \cdot n^3)$

Seja $n_0 = 0$ e $c = 2 / 0.001$ (2000)

$$2.n^3 \le 2000 \cdot (0.001 \cdot n^3)$$
$$\le 2\, n^3$$

Princípios Gerais

Outro exemplo:

$T(n) = 3 . n^5 + 10 . n^4 – 4 . n^3 + n + 1$

$3.n^5 + 10.n^4 – 4.n^3 + n + 1 \leq 3.n^5 + 10.n^5 + n^5 + n^5$

$\leq 15 n^5$

Constatando através das proporções...

$3 n^2 + 10 n + 10$ é $O(n^2)$

p/ n = 10 → 73.2%; 24.4% e 2.4%

p/ n = 100 → 96.7%; 3.2%; ...

Princípios Gerais

A eliminação de elementos de menor ordem é conseqüência do fato de que o que é realmente importante é a taxa de crescimento e não o valor exato de T(n).

Desta forma, $T(n) = 2^n + n^3$ tem como resultado a avaliação de que $T(n) = O(2^n)$ pois $n^3/2^n$ tende a zero à medida que n cresce.

Agenda

- Revisão
- Big O
- Usando limites para comparar crescimento
- Outras medidas
- Analisando tempo de execução
- Classificação

USANDO LIMITES PARA COMPARAR ORDEM DE CRESCIMENTO

Limite

$$\lim_{n\to\infty}\frac{t(n)}{g(n)} = \begin{cases} 0 \ \text{ implica que } t(n) < g(n) \\ c \ \text{ implica que } t(n) \approx g(n) \\ \infty \ \text{ implica que } t(n) > g(n) \end{cases}$$

Exemplo

- Compare a ordem de crescimento $\frac{1}{2}n(n-1)$ com n^2

$$\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n})=\frac{1}{2}$$

têm a mesma ordem de crescimento.

Exemplo

- Compare a ordem de crescimento n! com 2^n

→ Lembrando da fórmula de Stirling n! ≈ $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\,\frac{n^n}{2^n e^n}=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n=\infty$$

n! cresce bem mais rápido que 2^n.

Princípios Gerais

$T(n) = 2^n + n^3$ tem como resultado a avaliação de que $T(n) = O(2^n)$

Prova: Seja $n_0=10$ e $c = 2$. Deve-se provar que para $n \geq 10$ tem-se que $2^n + n^3 \leq 2.\ 2^n$

Subtraindo-se 2^n de ambos os lados temos: $n^3 \leq 2^n (2 – 1)$

Para $n = 10$ tem-se que

$2^{10} = 1024$

$10^3 = 1000$

Princípios Gerais

Tabela

O(1)	Constante
O(log n)	Logarítmica
O(n)	Linear
O(n log n)	n log n
$O(n^2)$	Quadrática
$O(n^3)$	Cúbica
$O(2^n)$	Exponencial

Princípios Gerais

A relação de big-oh (O) é importante para se estabelecer a relação de ≤ entre as funções. Existem outras definições dentro da análise de algoritmos que apresentam outras relações: big Ω e big Θ.

Agenda

- Big O
- Usando limites para comparar crescimento
- Outras medidas
- Analisando tempo de execução
- Classificação

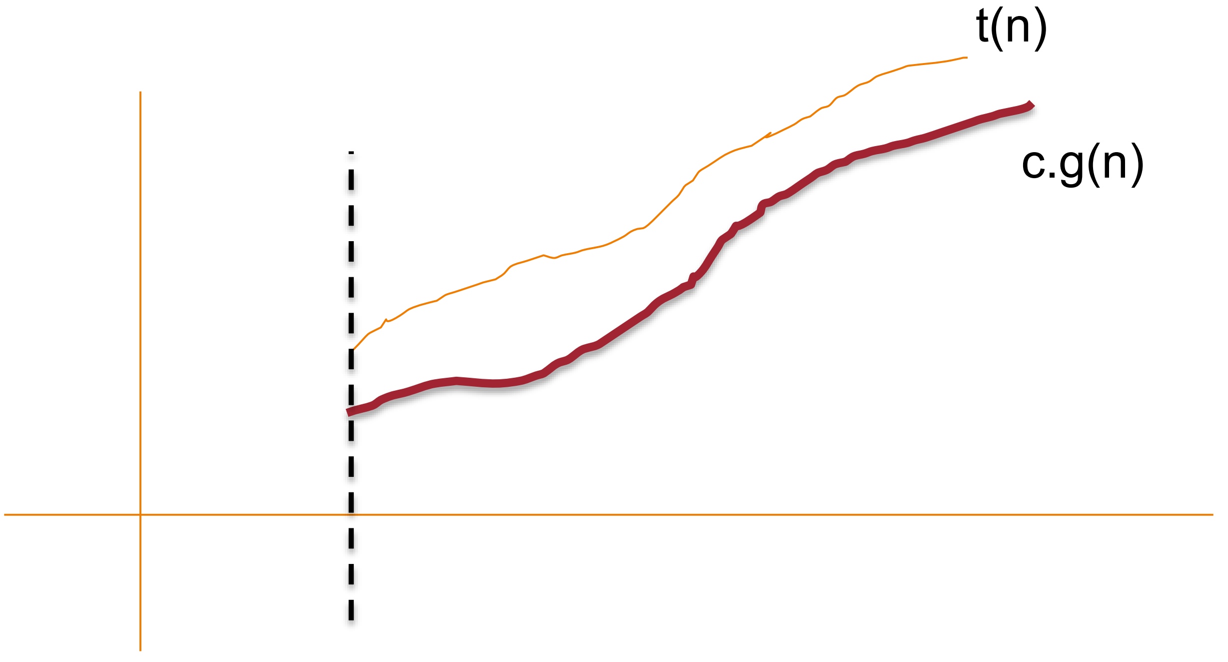
BIG Ω

Big Ω

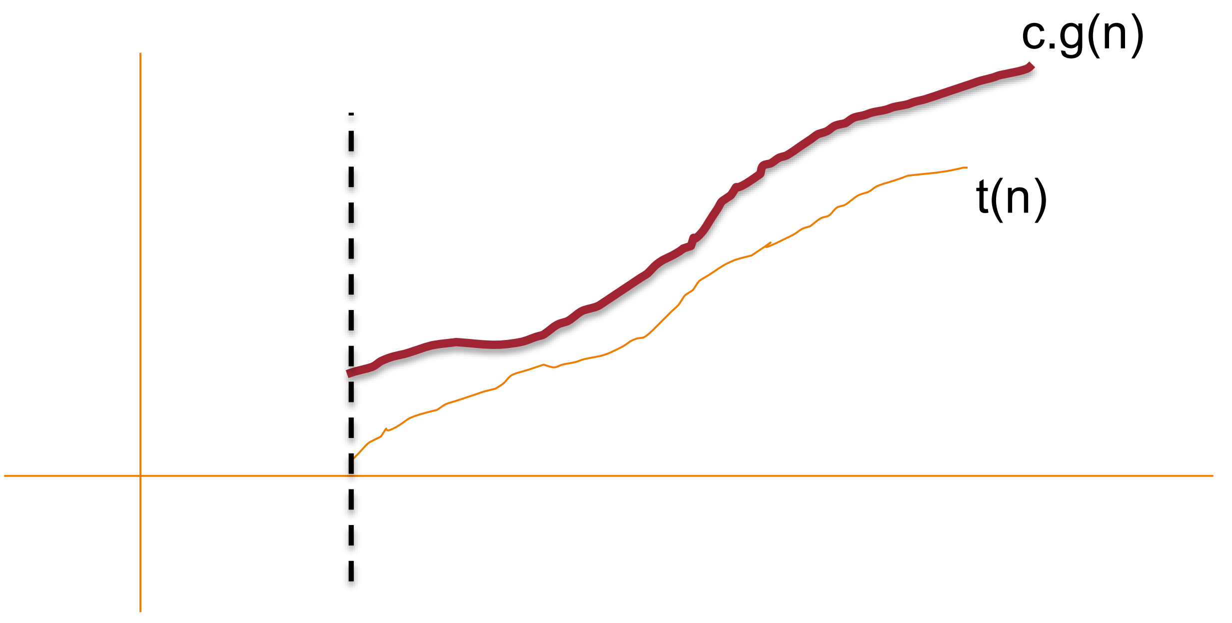
Definição: Uma função $t(n)$ é considerada $\Omega(g(n))$ e denotada por $t(n) \in \Omega(g(n))$ se $g(n)$ é um limite inferior de $t(n)$ tendo como diferença uma constante positiva c:

$$t(n) \geq c.g(n)$$

Big Ω

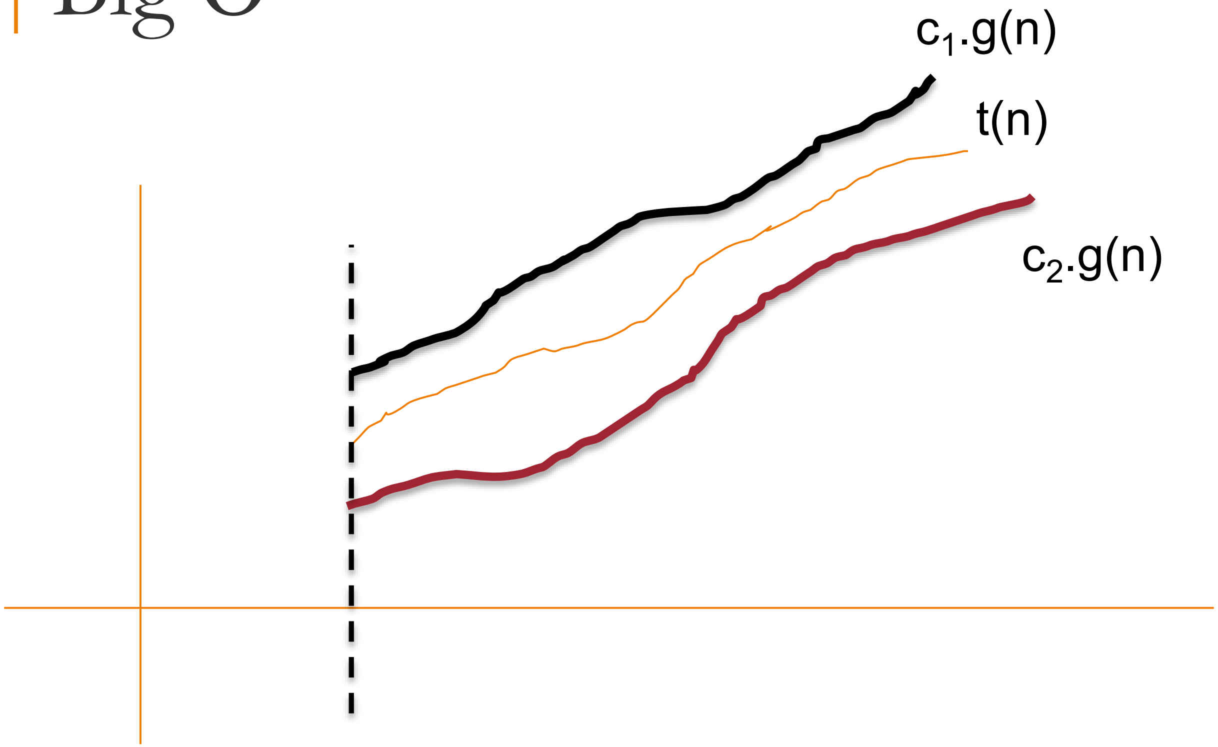


Relembrando o big O…



BIG Θ

Big Θ



Big Θ

Definição: Uma função $t(n)$ é considerada $\Theta(g(n))$ e denotada por $t(n) \in \Theta(g(n))$ se $g(n)$ é um limite inferior e também superior de $t(n)$ tendo como diferença as constantes positivas c_1 e c_2:

$$c_2.g(n) \le t(n) \le c_1.g(n) \text{ para todos } n \ge n_0$$

Exemplo

Provar que $\frac{1}{2}n(n-1) \in \Theta(n^2)$

$c_2.g(n) \leq t(n) \leq c_1.g(n)$ para todos $n \geq n_0$

Inequação lado direito: $\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$

Portanto, para o lado direito, temos que $c_1 = ½$ e $n_0 = 0$

Exemplo

Provar que $\frac{1}{2}n(n-1) \in \Theta(n^2)$

$c_2.g(n) \leq t(n) \leq c_1.g(n)$ para todos $n \geq n_0$

Inequação lado esquerdo: $\frac{1}{2}n(n-1) \geq c_2 n^2$

Exemplo

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

$$\frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n$$

$$\frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{4}n^2$$

$$\frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{4}n^2$$ Para todo $n \geq 2$

Exemplo

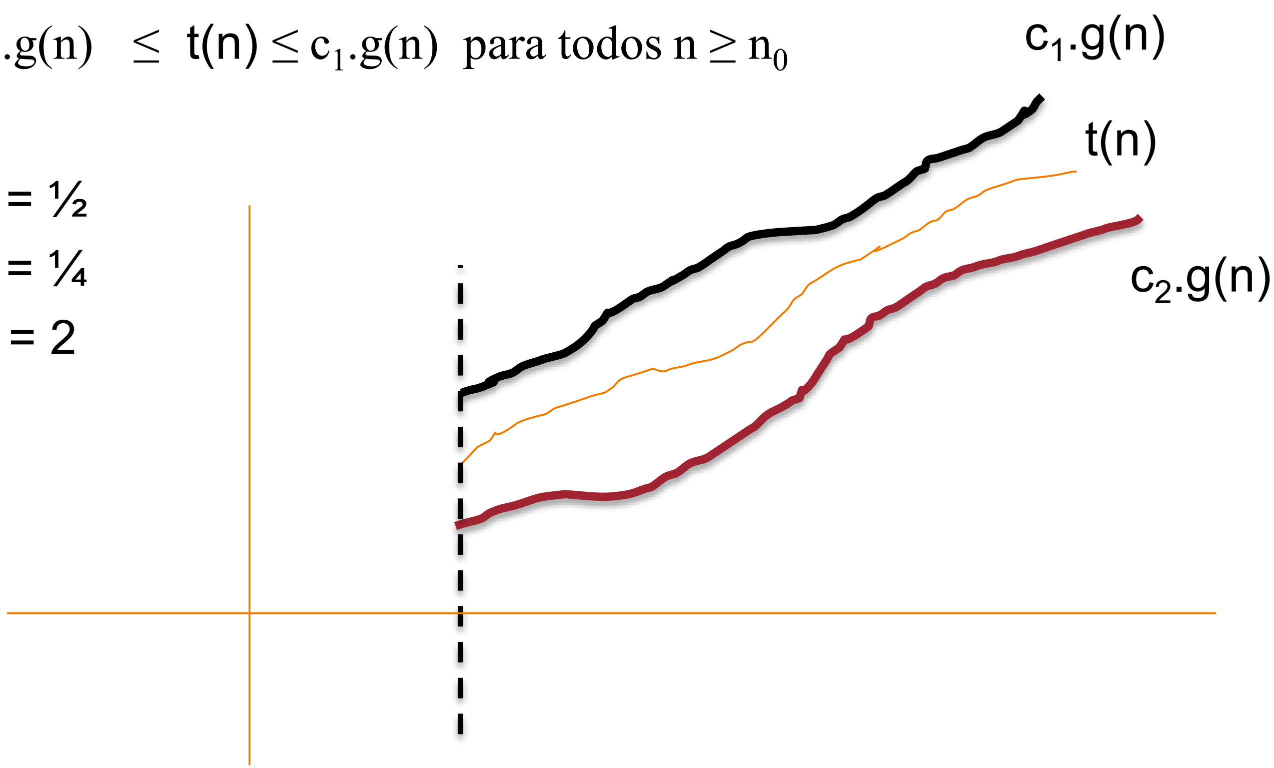
Provar que

$c_2.g(n) \le t(n) \le c_1.g(n)$ para todos $n \ge n_0$

$c_1 = ½$

$c_2 = ¼$

$n_0 = 2$



Agenda

- Revisão
- Big O
- Usando limites para comparar crescimento
- Outras medidas
- Analisando tempo de execução
- Classificação

Analisando tempo de execução

Comandos simples

→ atribuição, leitura, escrita... O(1)

Exemplo:

a ← 1

leia (x)

escreva (z)

Analisando tempo de execução

Repetições

→ Os limites superiores (loops determinados) indicam o limite superior do número de vezes que os comandos dentro do loop serão repetidos.

Exemplo 1: Para j ← 1 até n faça

a[j] ← j

Como atribuição é O(1) tem-se: n . O(1) = O(n)

Exemplo 2: Para i ← 1 até n faça

Para j ← 1 até n faça

a[i][j] ← i*j

Como atribuição é O(1) tem-se: n . O(1) = O(n)

Como tem-se outro loop (externo): n . O(n) = O(n^2)

Analisando tempo de execução

Repetições (while e repeat)

→ Não têm limite superior (indeterminado). Deve-se detectar um limite superior (pior caso).

Exemplo 1: i ← 1

Enquanto i <> a[i] faça

i ← i + 1

O(n)

Analisando tempo de execução

Comandos condicionais

→ Normalmente o comando condicional é O(1) – a menos que tenha uma chamada de função ou o cálculo de um número complicado. Assim, as partes então e senão do comando devem ser analisadas individualmente e uma estimativa deve ser atribuída ao conjunto.

Exemplo 1:

```
Se a[1][1] = 0 então
    Para i ← 1 até n faça
        Para j ← 1 até n faça
            a[i][j] ← i*j
senão
    Para j ← 1 até n faça
            a[i][j] ← 10
```

Quando não se tem ideia do que acontecerá → assumir pior caso

Analisando tempo de execução

Procedimentos

Se todos os procedimentos são não recursivos, pode-se começar a computar o tempo de todo o programa à partir dos procedimentos que não chamam outros. Parte-se, então, para aqueles que utilizam os procedimentos cujos valores já foram calculados... E assim por diante...

Analisando tempo de execução

Procedimentos Recursivos

Análise um pouco mais difícil.

função fatorial (n)

    se n ≤ 1 então

        fatorial ← 1

O(1)

    senão

        fatorial ← n . fatorial (n-1)

2.O(1) + O(n-1)

Analisando tempo de execução

Indução $T(n) = O(1) + T(n-1)$

Caso base : $T(1) = O(1) = a$

O(1)

Indução: $T(n) = b + T(n-1)$

$T(2) = b + T(1) = b + a$

$T(3) = b + T(2) = b + b + a = 2.b + a$

$T(4) = b + T(3) = b + 2.b + a = 3.b + a$

...

$T(n) = (n-1).b + a$

Analisando tempo de execução

Substituição repetida

$T(m) = b + T(m-1) \quad m > 1$

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

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

...

$T(2) = b + T(1)$

$T(1) = a$

Analisando tempo de execução

Substituição repetida

T(n-1)

$T(n) = b + b + T(n-2)$

T(n-2)

$T(n) = b + b + b + T(n-3)$

T(n-3)

$T(n) = b + b + b + b + T(n-4)$

$T(n) = 3.b + T(n-3)$ ou $4.b + T(n-4)$

...

$T(n) = (n-1).b + T(n-(n-1)) = (n-1).b + a$

THE END