Noções básicas de análise de complexidade

Prof. Paulo Henrique Pisani

fevereiro/2022

Tópicos

- Custo de um algoritmo;
- Crescimento assintótico;
- Notação O, Ω e Θ.

Custo de um algoritmo

Escolhendo estruturas e algoritmos

- Muitos problemas admitem diversas soluções (estruturas de dados e algoritmos);
- Como escolher uma estrutura de dados/algoritmo?

Escolhendo estruturas e algoritmos

- Muitos problemas admitem diversas soluções (estruturas de dados e algoritmos);
- Como escolher uma estrutura de dados/algoritmo?

Vamos escolher a de menor **CUSTO**

Como determinar o custo de um algoritmo?

Custo de um algoritmo

- Como determinar o custo de um algoritmo?
  - Tempo
  - Espaço (memória)

- Tempo e espaço não são infinitos! Portanto, devemos escolher estruturas e algoritmos eficientes no uso de recursos compucionais.

Aspecto chave:

Como o custo do algoritmo aumenta para entradas cada vez maiores?

Aspecto chave:

Como o custo do algoritmo aumenta para entradas cada vez maiores?

Aspecto chave:

Como o custo do algoritmo aumenta para entradas cada vez maiores?

Custo de um algoritmo

```
void insertion_sort(int v[], int n) {
    int i;
    for (i = 1; i < n; i++) {
        int atual = v[i];
        int k, indice_insercao = i;
        for (k = i - 1; k >= 0 && atual < v[k]; k--) {
            v[k + 1] = v[k];
            indice_insercao--;
        }
        v[indice_insercao] = atual;
    }
}
```

Qual o custo desse algoritmo?

Custo de um algoritmo

```
void insertion_sort(int v[], int n){
    int i;
    for (i = 1; i < n; i++) {
        int a = v[i];
        int k, indice_insercao = i;
        for (k = i - 1; k >= 0 && a < v[k]; k--){
            v[k + 1] = v[k];
            indice_insercao--;
        }
        v[indice_insercao] = a;
    }
}
```

Custo	Vezes
c_1	?
c_2	?
c_3	?
c_4	?
c_5	?
c_6	?
c_7	?

Custo de um algoritmo

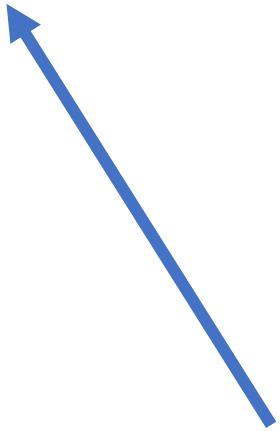
```
void insertion_sort(int v[], int n){
    int i;
    for (i = 1; i < n; i++) {
        int a = v[i];
        int k, indice_insercao = i;
        for (k = i - 1; k >= 0 && a < v[k]; k--){
            v[k + 1] = v[k];
            indice_insercao--;
        }
        v[indice_insercao] = a;
    }
}
```

Custo	Vezes
c_1	n
c_2	n-1
c_3	n-1
c_4	?
c_5	?
c_6	?
c_7	n-1

Custo de um algoritmo

```
void insertion_sort(int v[], int n){
    int i;
    for (i = 1; i < n; i++) {
        int a = v[i];
        int k, indice_insercao = i;
        for (k = i - 1; k >= 0 && a < v[k]; k--){
            v[k + 1] = v[k];
            indice_insercao--;
        }
        v[indice_insercao] = a;
    }
}
```

Custo	Vezes
c_1	n
c_2	n-1
c_3	n-1
c_4	?
c_5	?
c_6	?
c_7	n-1



Vamos chamar de q_i a quantidade de vezes que a comparação do *for* mais interno é executada para a iteração *i*.

Custo de um algoritmo

```
void insertion_sort(int v[], int n){
    int i;
    for (i = 1; i < n; i++) {
        int a = v[i];
        int k, indice_insercao = i;
        for (k = i - 1; k >= 0 && a < v[k]; k--){
            v[k + 1] = v[k];
            indice_insercao--;
        }
        v[indice_insercao] = a;
    }
}
```

Custo	Vezes
c_1	n
c_2	n-1
c_3	n-1
c_4	$\sum_{i=1}^{n-1} q_i$
c_5	$\sum_{i=1}^{n-1} (q_i - 1)$
c_6	$\sum_{i=1}^{n-1} (q_i - 1)$
c_7	n-1

Custo de um algoritmo

```
void insertion_sort(int v[], int n){
    int i;
    for (i = 1; i < n; i++) {
        int a = v[i];
        int k, indice_insercao = i;
        for (k = i - 1; k >= 0 && a < v[k]; k--){
            v[k + 1] = v[k];
            indice_insercao--;
        }
        v[indice_insercao] = a;
    }
}
```

Custo	Vezes
c_1	n
c_2	n-1
c_3	n-1
c_4	$\sum_{i=1}^{n-1} q_i$
c_5	$\sum_{i=1}^{n-1} (q_i - 1)$
c_6	$\sum_{i=1}^{n-1} (q_i - 1)$
c_7	n-1

Mede o custo de acordo com o tamanho da entrada (parâmetro *n*)

A **função de custo (ou função de complexidade)** pode ser escrita da seguinte forma:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_1^{n-1} q_i + c_5 \sum_1^{n-1}(q_i - 1) + c_6 \sum_1^{n-1}(q_i - 1) + c_7(n-1)$$

Custo de um algoritmo

```
void insertion_sort(int v[], int n){
    int i;
        }
        v[indice_insercao] = a;
    }
}
```

- Neste caso, a função de complexidade mede o tempo e portanto, pode ser chamada de **função de complexidade de tempo;**
- Se a função medir a quantidade de memória usada para executar o algoritmo, seria uma **função de complexidade de memória**.

A **função de custo (ou função de complexidade)** pode ser escrita da seguinte forma:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} q_i + c_5 \sum_{i=1}^{n-1}(q_i - 1) + c_6 \sum_{i=1}^{n-1}(q_i - 1) + c_7(n-1)$$

Melhor caso

- Qual o melhor caso para o ***insertion sort***?

Melhor caso

- Qual o melhor caso para o ***insertion sort***?

  Quando todos os elementos já estão ordenados. Nesse caso, nenhum deslocamento é realizado pelo algoritmo.

Melhor caso

- No melhor caso, quando o vetor está ordenado, temos que:

$$q_1 = q_2 = q_3 = \cdots = q_{(n-1)} = 1$$

A condição é sempre falsa quando o vetor está ordenado!

```
void insertion_sort(int v[], int n){
    int i;
    for (i = 1; i < n; i++) {
        int a = v[i];
        int k, indice_insercao = i;
        for (k = i - 1; k >= 0 && a < v[k]; k--){
            v[k + 1] = v[k];
            indice_insercao--;
        }
        v[indice_insercao] = a;
    }
}
```

Melhor caso

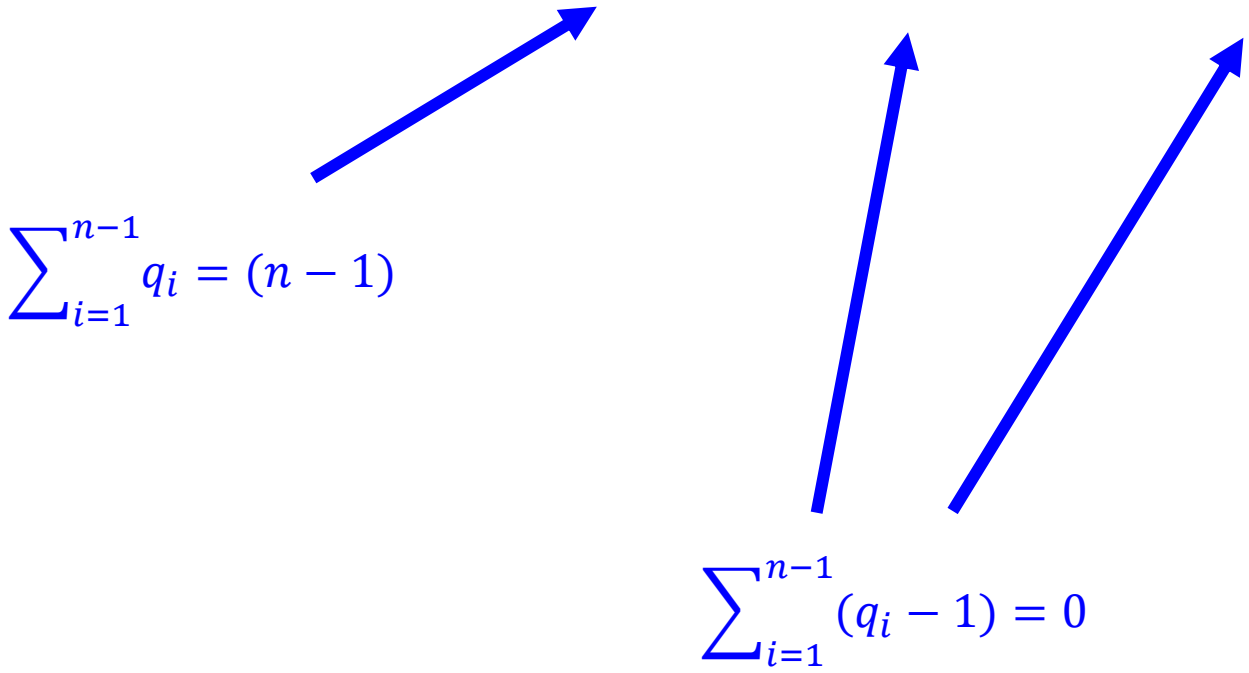
```
void insertion_sort(int v[], int n){
    int i;
    for (i = 1; i < n; i++) {
        int a = v[i];
        int k, indice_insercao = i;
        for (k = i - 1; k >= 0 && a < v[k]; k--){
            v[k + 1] = v[k];
            indice_insercao--;
        }
        v[indice_insercao] = a;
    }
}
```

Custo	Vezes
c_1	n
c_2	n-1
c_3	n-1
c_4	$\sum_{i=1}^{n-1} q_i$
c_5	$\sum_{i=1}^{n-1} (q_i - 1)$
c_6	$\sum_{i=1}^{n-1} (q_i - 1)$
c_7	n-1

$$q_1 = q_2 = q_3 = \cdots = q_{(n-1)} = 1$$

Melhor caso

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} q_i + c_5 \sum_{i=1}^{n-1}(q_i - 1) + c_6 \sum_{i=1}^{n-1}(q_i - 1) + c_7(n-1)$$



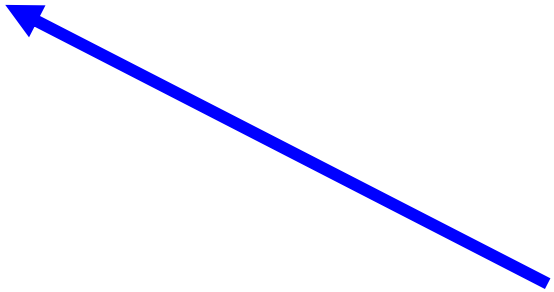
Melhor caso

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} q_i + c_5 \sum_{i=1}^{n-1}(q_i - 1) + c_6 \sum_{i=1}^{n-1}(q_i - 1) + c_7(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 0 + c_6 0 + c_7(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$



É uma função linear, pode ser representada como ***an + b***

Pior caso

- Qual o pior caso para o ***insertion sort***?

Pior caso

- Qual o pior caso para o ***insertion sort***?

Quando todos os elementos já estão ordenados na ordem inversa. Nesse caso, a quantidade de deslocamentos é a máxima possível, pois cada elemento será inserido na primeira posição.

Para a análise a seguir, vamos considerar que todos os elementos do vetor são distintos.

Pior caso

- No pior caso, quando o vetor está em ordem inversa, temos que:

$$q_1 = 2,\ q_2 = 3,\ q_3 = 4,\ \ldots,\ q_{(n-1)} = n$$

A condição é sempre verdadeira quando o vetor está em ordem inversa!

Dessa forma, o loop irá parar apenas quando k = 0

```
void insertion_sort(int v[], int n){
    int i;
    for (i = 1; i < n; i++) {
        int a = v[i];
        int k, indice_insercao = i;
        for (k = i - 1; k >= 0 && a < v[k]; k--){
            v[k + 1] = v[k];
            indice_insercao--;
        }
        v[indice_insercao] = a;
    }
}
```

Pior caso

```
void insertion_sort(int v[], int n){
    int i;
    for (i = 1; i < n; i++) {
        int a = v[i];
        int k, indice_insercao = i;
        for (k = i - 1; k >= 0 && a < v[k]; k--){
            v[k + 1] = v[k];
            indice_insercao--;
        }
        v[indice_insercao] = a;
    }
}
```

Custo	Vezes
c_1	n
c_2	n-1
c_3	n-1
c_4	$\sum_{i=1}^{n-1} q_i$
c_5	$\sum_{i=1}^{n-1} (q_i - 1)$
c_6	$\sum_{i=1}^{n-1} (q_i - 1)$
c_7	n-1

$$q_1 = 2,\ q_2 = 3,\ q_3 = 4,\ \ldots,\ q_{(n-1)} = n$$

Pior caso

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \underbrace{\sum_{i=1}^{n-1} q_i} + c_5 \underbrace{\sum_{i=1}^{n-1}(q_i - 1)} + c_6 \underbrace{\sum_{i=1}^{n-1}(q_i - 1)} + c_7(n-1)$$

$$\sum_{i=1}^{n-1} q_i = \frac{(n-1)}{2}(q_1 + q_{(n-1)})$$

$$\sum_{i=1}^{n-1}(q_i - 1) = \frac{(n-1)}{2}(q_1 - 1 + q_{(n-1)} - 1)$$

Soma dos elementos de uma progressão aritmética (PA) de *m* elementos:

$$\sum_{j=1}^{m} a_j = \frac{m}{2}(a_1 + a_m)$$

Pior caso

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} q_i + c_5 \sum_{i=1}^{n-1}(q_i - 1) + c_6 \sum_{i=1}^{n-1}(q_i - 1) + c_7(n-1)$$

$$\sum_{i=1}^{n-1} q_i = \frac{(n-1)}{2}(q_1 + q_{(n-1)})$$

$$\sum_{i=1}^{n-1} (q_i - 1) = \frac{(n-1)}{2}(q_1 - 1 + q_{(n-1)} - 1)$$

No pior caso, temos que:

$q_1 = 2$

$q_{(n-1)} = n$

Pior caso

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} q_i + c_5 \sum_{i=1}^{n-1}(q_i - 1) + c_6 \sum_{i=1}^{n-1}(q_i - 1) + c_7(n-1)$$

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

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

No pior caso, temos que:

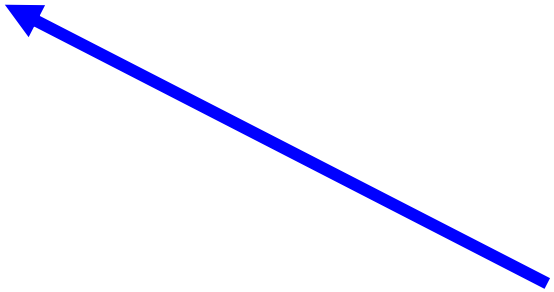
$q_1 = 2$

$q_{(n-1)} = n$

Pior caso

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\frac{n(n-1)}{2} + c_6\frac{n(n-1)}{2} + c_7(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n^2+n-1}{2}\right) + c_5\frac{n^2-n}{2} + c_6\frac{n^2-n}{2} + c_7(n-1)$$



É uma função quadrática, pode ser representada como $an^2 + bn + c$

Pior caso

- Frequentemente, o foco das análises são para o **pior caso**:
  - O pior caso define um limite superior;
  - Em alguns algoritmos, o pior caso ocorre diversas vezes (e.g. busca de uma chave inexistente em um vetor);
  - O caso médio pode ser tão ruim quanto o pior caso (e.g. para o insertion sort, o caso médio ainda é uma função quadrática).

Crescimento assintótico

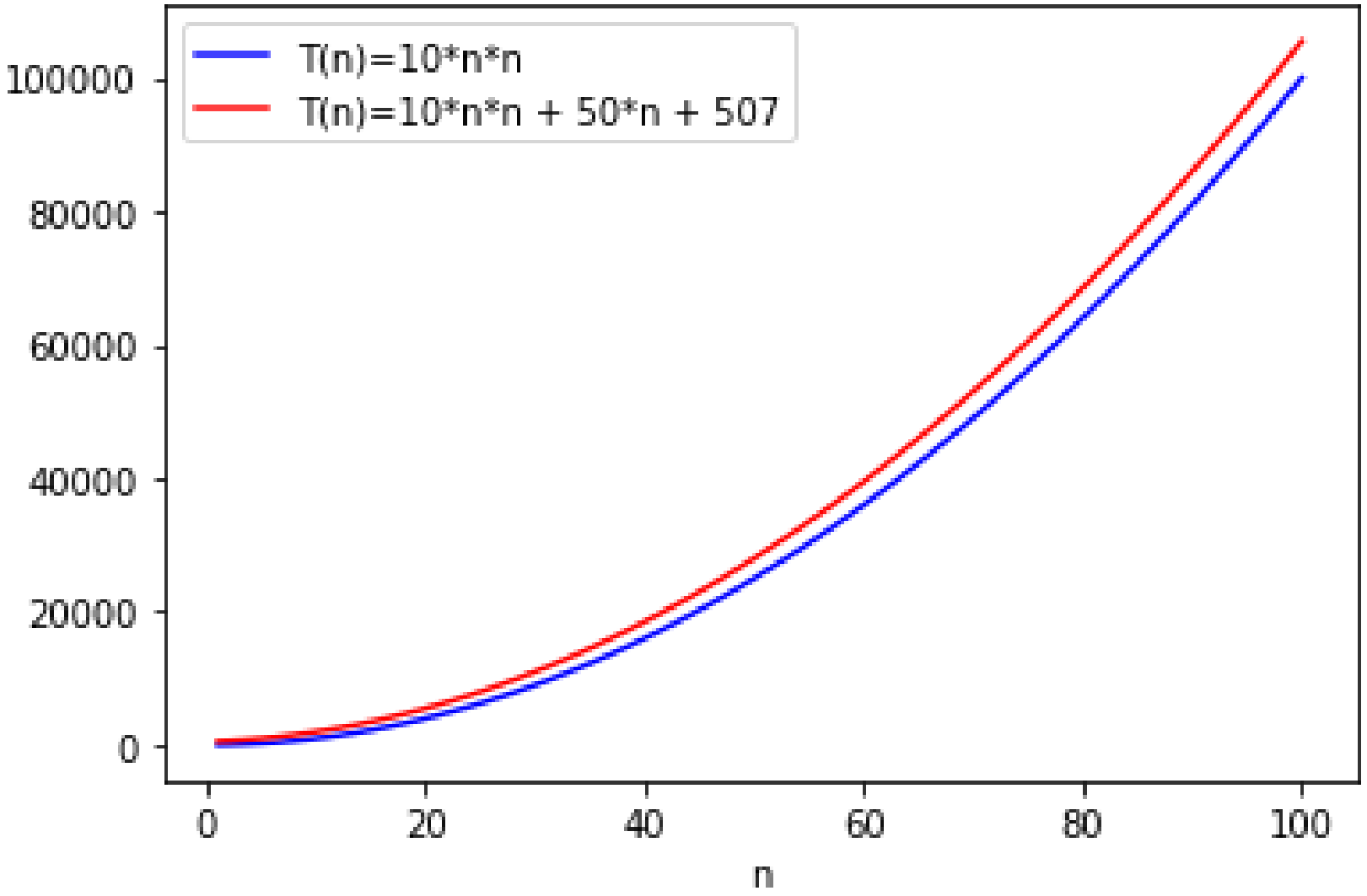
Comportamento assintótico

- Na análise de algoritmos, vamos utilizar valores grandes para o tamanho de entrada (o parâmetro *n* da função de custo);
  - Para entradas pequenas, praticamente qualquer algoritmo tem desempenho similar.
- Ou seja, vamos avaliar o **comportamento assintótico de T(n)**

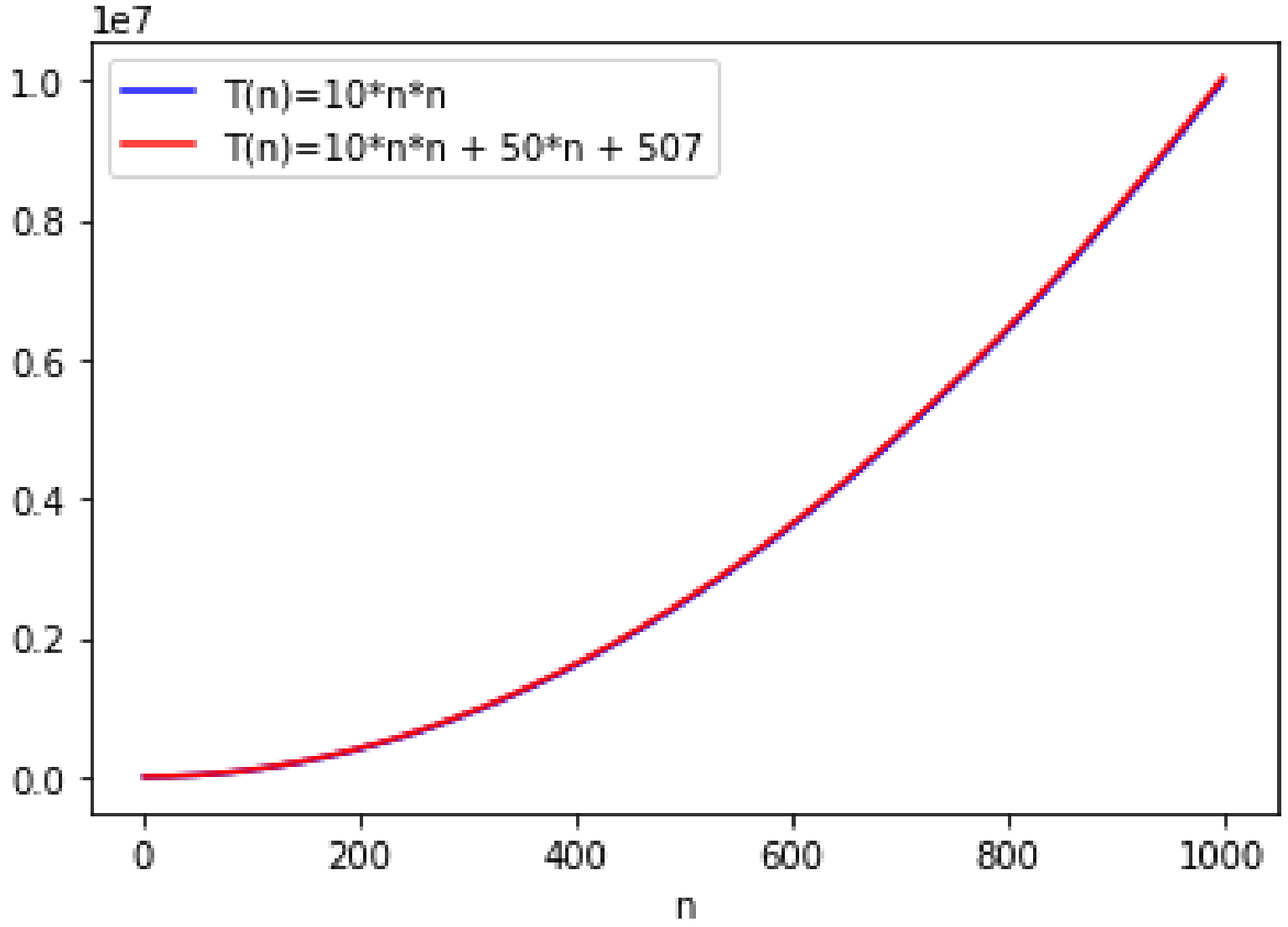
Comportamento assintótico

- Durante as análises, realizamos duas simplificações:
  1) **Apenas o termo de maior ordem será considerado** (pois os demais tem impacto relativamente insignificantes)
  2) **Constantes serão desconsideradas**

Comportamento assintótico



Comportamento assintótico



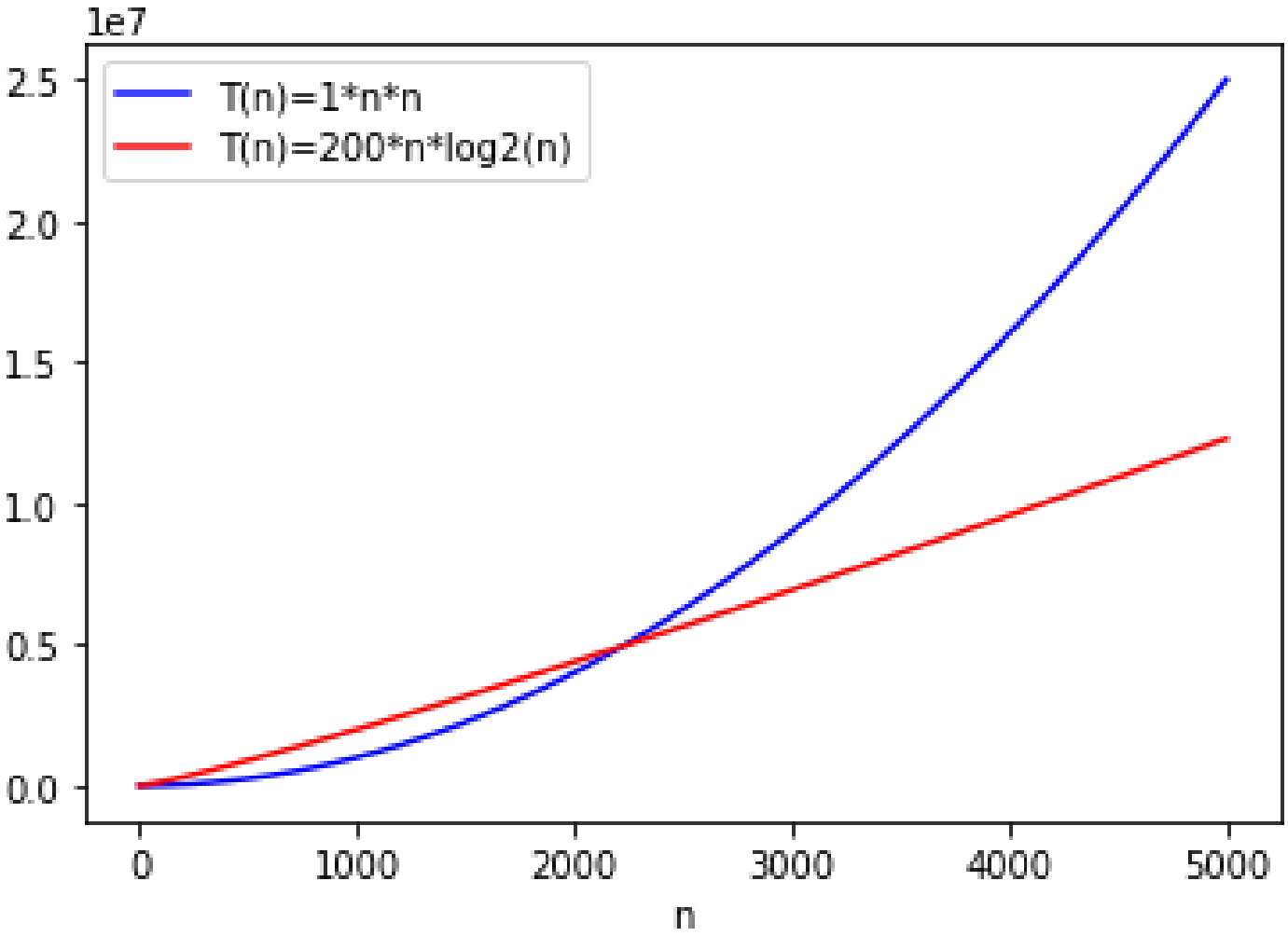
Diferenças de implementação

- Diferenças na ordem de crescimento de acordo com a entrada podem ter grande impacto;
  - Mesmo com diferenças de eficiência de implementação.

Implementação x Ordem de crescimento

- Exemplo:
  - Algoritmo A implementado na linguagem L1
  - Algoritmo B implementado na linguagem L2
- Algoritmo A: **$T(n) = c.n^2$**
- Algoritmo B: **$T(n) = c.n.\log(n)$**
- Linguagem **L1: c = 1**
- Linguagem **L2: c = 200**

Implementação x Ordem de crescimento

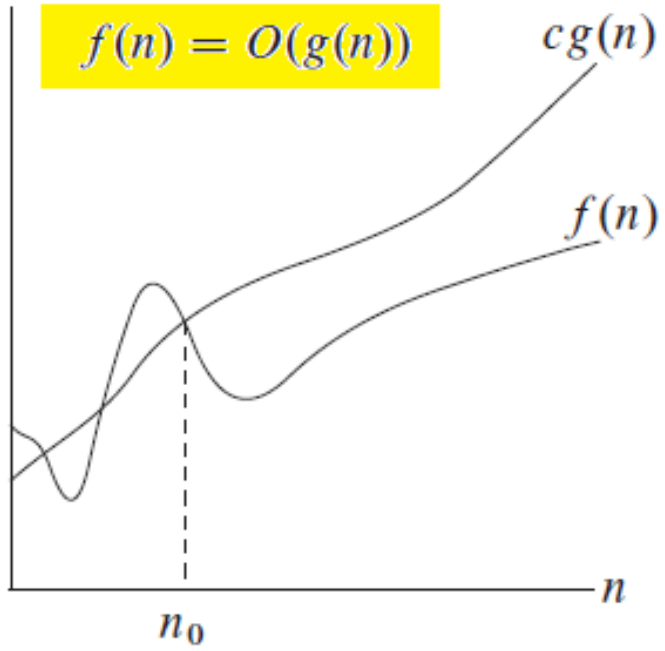


Notação O

Notação O

- Define um **limite superior** para o comportamento assintótico de uma função.

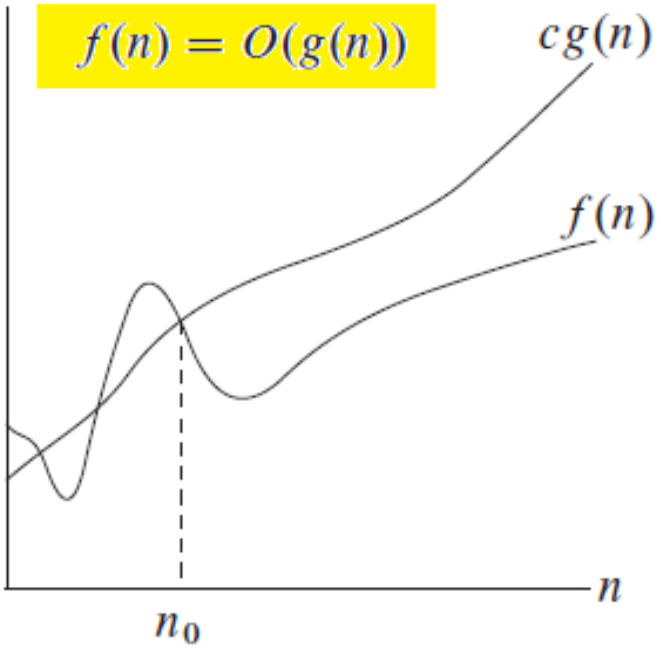
f(n) = O(g(n))



Notação O

- Define um **limite superior** para o comportamento assintótico de uma função;
- Se existem duas constantes positivas c e n_0 que, para $n \geq n_0$, temos $0 \leq f(n) \leq c.g(n)$, dizemos que **f(n) = O(g(n))**

- **As funções f e g devem ser assintoticamente não negativas.**



Notação O

- **Exemplo 1**: para a função:

$$T(n) = n^3 + 2n + 507$$

- Podemos dizer que: $T(n) = O(n^3)$

- Também podemos dizer que: $T(n) = O(n^{50})$
  - Entretanto normalmente buscamos expressar a notação com limites mais próximos.

Notação O

- **Exemplo 2**: no pior caso do insertion sort, temos

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n^2+n-1}{2}\right) + c_5\frac{n^2-n}{2} + c_6\frac{n^2-n}{2} + c_7(n-1)$$

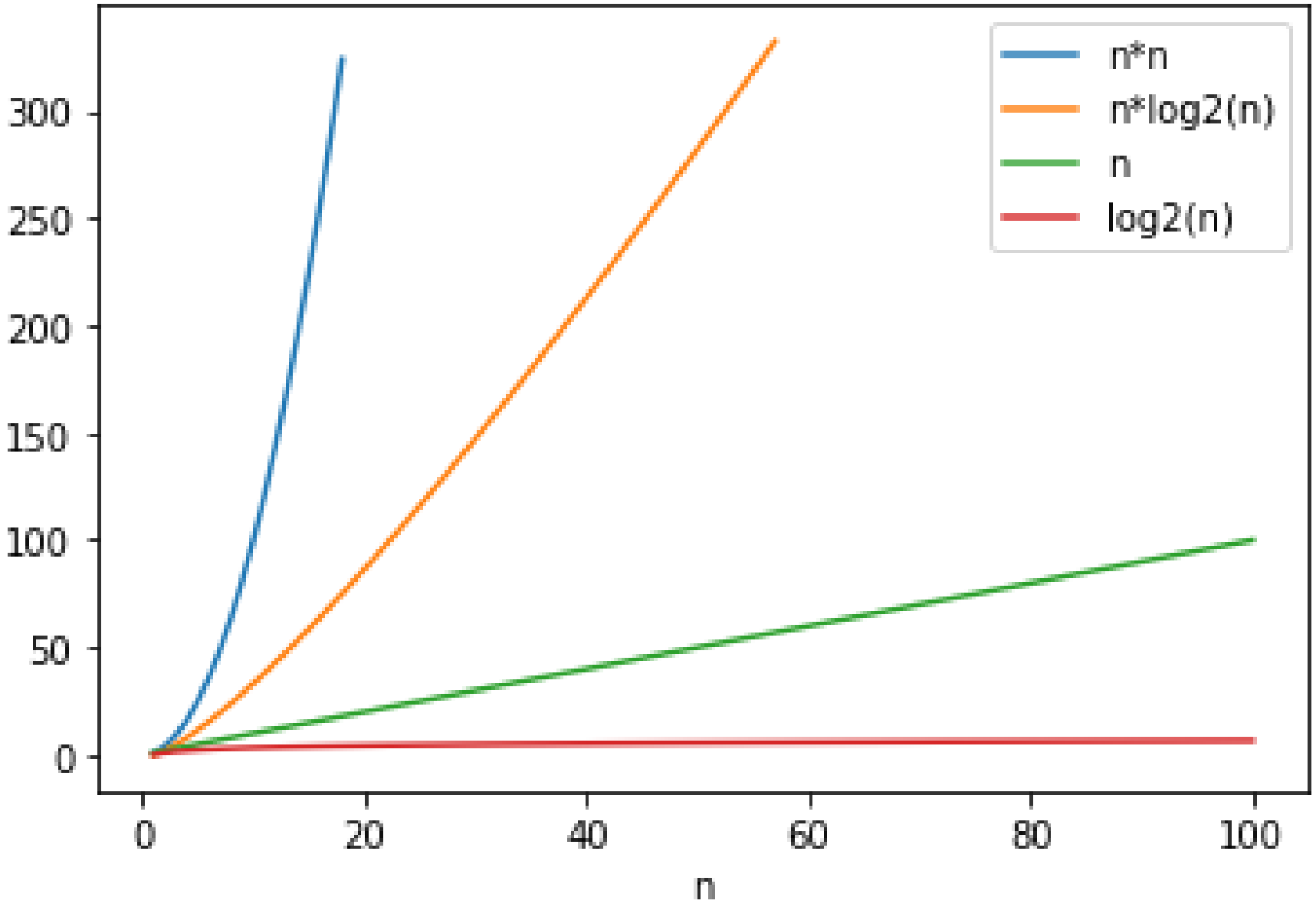
- Podemos dizer que: $T(n) = O(n^2)$

Lista em ordem crescente de complexidade

Complexidade	Nome
$O(1)$	Constante
$O(\log\log(n))$	Duplo logarítimo
$O(\log(n))$	Logarítimo
$O(n)$	Linear
$O(n.\log(n))$	Loglinear
$O(n^2)$	Quadrático
$O(n^c)$	Polinomial
$O(c^n)$	Exponencial
$O(n!)$	Fatorial

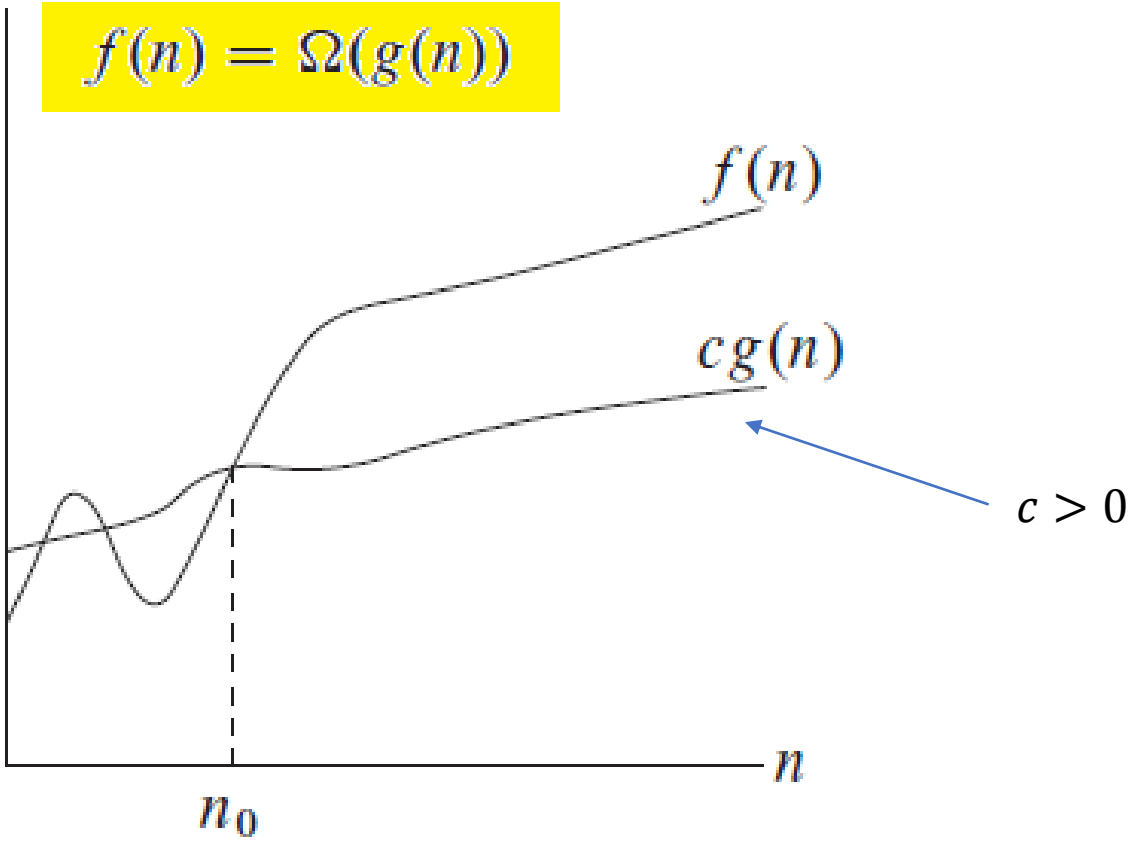
n f(n)	log n	n	n log n	n^2	2^n	n!
10	0.003ns	0.01ns	0.033ns	0.1ns	1ns	3.65ms
20	0.004ns	0.02ns	0.086ns	0.4ns	1ms	77years
30	0.005ns	0.03ns	0.147ns	0.9ns	1sec	$8.4x10^{15}$yrs
40	0.005ns	0.04ns	0.213ns	1.6ns	18.3min	--
50	0.006ns	0.05ns	0.282ns	2.5ns	13days	--
100	0.07	0.1ns	0.644ns	0.10ns	$4x10^{13}$yrs	--
1,000	0.010ns	1.00ns	9.966ns	1ms	--	--
10,000	0.013ns	10ns	130ns	100ms	--	--
100,000	0.017ns	0.10ms	1.67ms	10sec	--	--
1'000,000	0.020ns	1ms	19.93ms	16.7min	--	--
10'000,000	0.023ns	0.01sec	0.23ms	1.16days	--	--
100'000,000	0.027ns	0.10sec	2.66sec	115.7days	--	--
1,000'000,000	0.030ns	1sec	29.90sec	31.7 years	--	--

Tabela de: http://cooervo.github.io/Algorithms-DataStructures-BigONotation/index.html


n*n
n*log2(n)
n
log2(n)
300
250
200
150
100
50
0
0
20
40
60
80
100
n

Notação Ω

Notação Ω

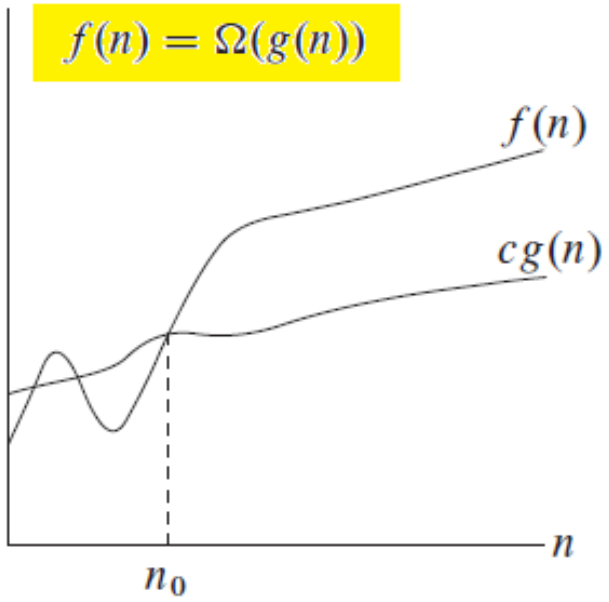


- Define um **limite inferior** para o comportamento assintótico de uma função.

Notação Ω

- Define um **limite inferior** para o comportamento assintótico de uma função.
- Se existem duas constantes positivas c e n_0 que, para $n \geq n_0$, temos $0 \leq c.g(n) \leq f(n)$, dizemos que $f(n) = \Omega(g(n))$

- **As funções f e g devem ser assintoticamente não negativas.**



Notação Ω

- **Exemplo**: para a função:

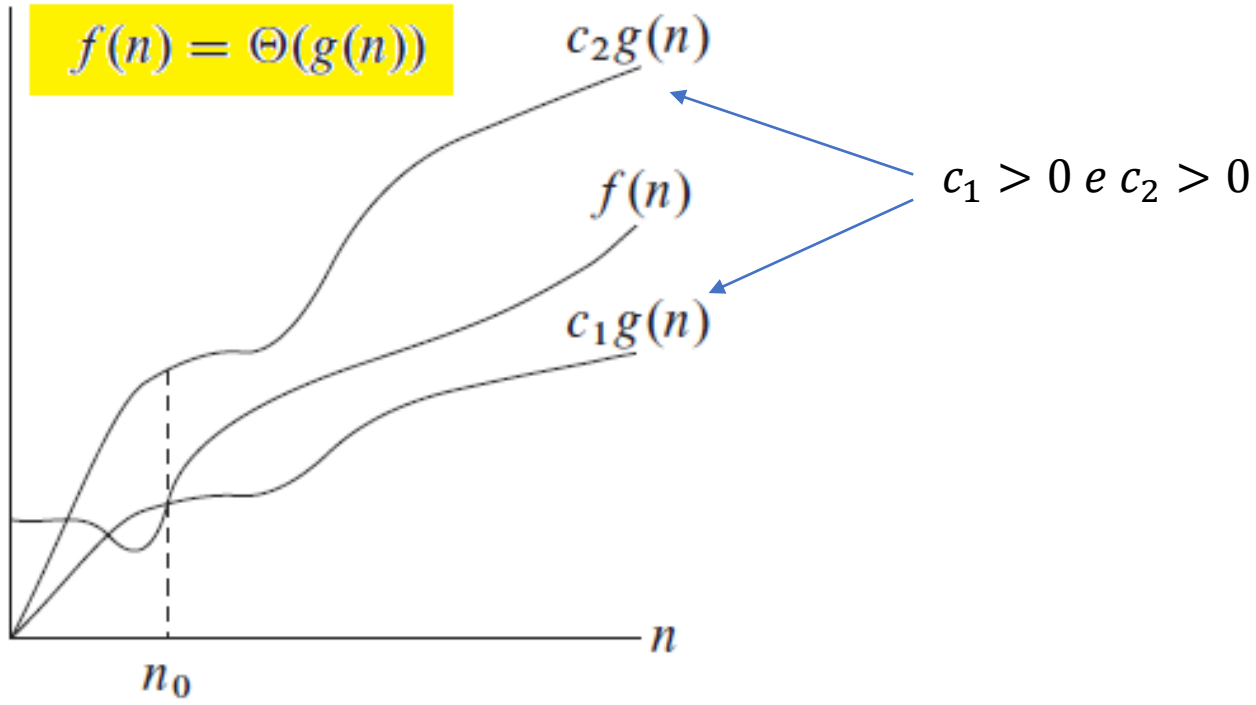
$$T(n) = n^3 + 2n + 507$$

- Podemos dizer que: $T(n) = \Omega(n^3)$

Notação Θ

Notação Θ

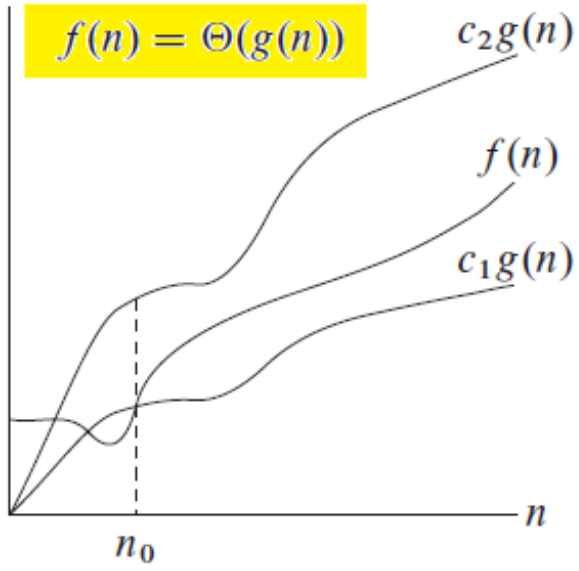
- Define um **limite restrito** para o comportamento assintótico de uma função.



Notação Θ

- Define um **limite restrito** para o comportamento assintótico de uma função.
- Se existem três constantes positivas c_1, c_2 e n_0 que, para $n \geq n_0$, temos $0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)$, dizemos que $f(n) = \Theta(g(n))$

- **As funções f e g devem ser assintoticamente não negativas.**



Notação Θ

- **Exemplo**: para a função:

$$T(n) = n^3 + 2n + 507$$

- Podemos dizer que: $T(n) = \Theta(n^3)$

Referências

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 3. ed. Rio de Janeiro, RJ: Elsevier, 2012.
- Nivio Ziviani. Projeto de Algoritmos: com implementações em Pascal e C. Cengage Learning, 2015.