Complexidade Assintótica

Eduardo Furlan Miranda

2024-02-01

Adaptado dos materiais dos Profs. N.T. Roman e F.L.S. Nunes

Análise de algoritmos

- Permite escolher o algoritmo mais eficiente
- Geralmente o tempo de duração de um algoritmo cresce com o tamanho da entrada
  - O tempo de execução de um programa pode ser descrito como uma função do tamanho de sua entrada
- Tamanho de entrada (*n*)
  - Geralmente é o nº de itens de entrada
- Tempo de execução
  - Quantidade de operações
  - Considerar que cada linha ***i*** tem um tempo constante c_i

```
A = [9, 8, 7, 6, 5, 4, 3, 2, 1]
for j in range(1, len(A)):
    chave = A[j]
    i = j - 1
    while i >= 0 and A[i] > chave:
        A[i + 1] = A[i]
        i -= 1
    A[i + 1] = chave
    print(A)
```

```
[8, 9, 7, 6, 5, 4, 3, 2, 1]
[7, 8, 9, 6, 5, 4, 3, 2, 1]
[6, 7, 8, 9, 5, 4, 3, 2, 1]
[5, 6, 7, 8, 9, 4, 3, 2, 1]
[4, 5, 6, 7, 8, 9, 3, 2, 1]
[3, 4, 5, 6, 7, 8, 9, 2, 1]
[2, 3, 4, 5, 6, 7, 8, 9, 1]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Ordenação por inserção (*insertion-sort*)

```
A = [9, 8, 7, 6, 5, 4, 3, 2, 1]
for j in range(1, len(A)):
    print(f"\n(j={j + 1}) ", end="")
    chave = A[j]
    i = j - 1
    while i >= 0 and A[i] > chave:
        print(i + 1, end=", ")
        A[i + 1] = A[i]
        i -= 1
    A[i + 1] = chave
```

```
(j=2) 1,
(j=3) 2, 1,
(j=4) 3, 2, 1,
(j=5) 4, 3, 2, 1,
(j=6) 5, 4, 3, 2, 1,
(j=7) 6, 5, 4, 3, 2, 1,
(j=8) 7, 6, 5, 4, 3, 2, 1,
(j=9) 8, 7, 6, 5, 4, 3, 2, 1,
```

Laço interno

- A parte interna do laço executa até 8 vezes
- A linha do *while* executa até 9 vezes
- t_j : nº de vezes que o laço executa, para aquele valor de **j**

Função de custo de um algoritmo

	ordenação por inserção	custo	vezes
1	`for j in range(1, len(A)):`	c_1	n
2	`    chave = A[j]`	c_2	n-1
3	`    # ordenando elementos à esquerda`	0	n-1
4	`    i = j - 1`	c_4	n-1
5	`    while i >= 0 and A[i] > chave:`	c_5	$\sum_{j=2}^{n} t_j$
6	`        A[i + 1] = A[i]`	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	`        i -= 1`	c_7	$\sum_{j=2}^{n} (t_j - 1)$
9	`    A[i + 1] = chave`	c_8	n-1

Tempo de execução do algoritmo - melhor caso

Entrada de *n* valores

$$T(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5\sum_{j=2}^{n} t_j+c_6\sum_{j=2}^{n}(t_j-1)+c_7\sum_{j=2}^{n}(t_j-1)+c_8(n-1)$$

- Melhor caso: vetor já ordenado
  - Na linha 5: *A[i]* ≤ *chave*, o que implica $t_j = 1$ para $j = 2, 3, \ldots, n$

vai ser executado uma vez

$$T(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5(n-1)+c_8(n-1)$$

$$T(n)=(c_1+c_2+c_4+c_5+c_8)n-(c_2+c_4+c_5+c_8)$$

a b

Podemos expressar como uma função linear: $an + b$

Tempo de execução do algoritmo - pior caso

$$T(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5\sum_{j=2}^{n} t_j+c_6\sum_{j=2}^{n}(t_j-1)+c_7\sum_{j=2}^{n}(t_j-1)+c_8(n-1)$$

- Pior caso: se o arranjo estiver ordenado em ordem inversa
  - Compara cada A[j] com A[1 .. j-1] → t_j = j para j = 2, 3, ..., n

$$\sum_{j=2}^{n} j=\frac{n(n+1)}{2}-1 \qquad \sum_{j=2}^{n}(j-1)=\frac{n(n-1)}{2}$$

Propriedades matemáticas

$$T(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5\left(\frac{n(n+1)}{2}-1\right)+c_6\left(\frac{n(n-1)}{2}\right)+c_7\left(\frac{n(n-1)}{2}\right)+c_8(n-1)$$

$an^2 + bn - c$

Usa-se apenas o termo mais importante

$$T(n)=\left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2+\left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n-\left(c_2+c_4+c_5+c_8\right)$$

a b c

- Tempo de execução de um algoritmo
  - Fixo para uma determinada entrada
    - Geralmente

- Normalmente analisamos apenas o *pior caso*
  - Limite superior do tempo, para qualquer entrada
  - O pior caso ocorre com frequência em alguns algoritmos
    - Ex.: procurar um registro inexistente em um banco de dados
  - Muitas vezes o caso médio é quase tão ruim quanto o pior

Taxa ou ordem de crescimento

- Considera apenas o termo inicial (ex.: an^2)
  - Termos de mais baixa ordem são relativamente insignificantes
- Ignora o coeficiente constante do termo inicial
  - Menos significativo para grandes entradas
- Notação: $\Theta(n^2)$ (“téta de n ao quadrado”)
  - função de custo (de tempo)
- Geralmente um algoritmo é mais eficiente se o tempo de execução no pior caso possui menor ordem de crescimento

Complexidade assintótica

- Método para descrever o comportamento de limites
  - Ex.: desempenho de algoritmos quando aplicados a um volume muito grande de dados de entrada

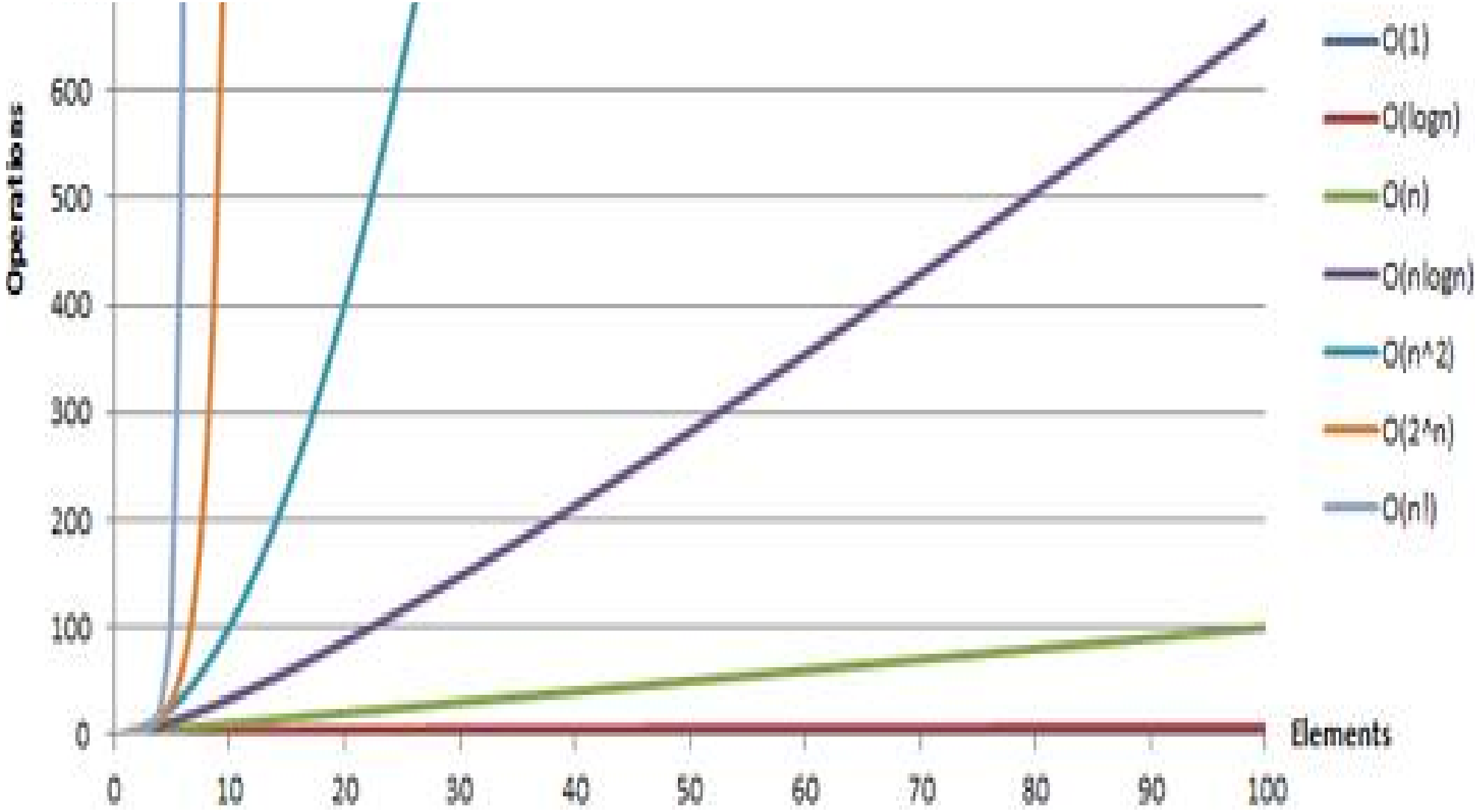
- A escolha do algoritmo pode se tornar um problema crítico quando *n* é grande
  - Analisa-se o comportamento assintótico das funções de custo
    - Comportamento das funções para valores grandes de n
    - O comportamento assintótico de f (n) representa o limite do comportamento do custo (complexidade) quando n cresce
    - Representa o limite do comportamento da função de custo

Eficiência assintótica dos algoritmos

- Maneira como o tempo de execução de um algoritmo aumenta com o tamanho da entrada no limite
  - À medida que o tamanho da entrada aumenta indefinidamente

- Um algoritmo que é assintoticamente mais eficiente será a melhor escolha para todas as entradas, exceto as pequenas

Eficiência assintótica dos algoritmos



	$n = 100$	$n = 1000$	$n = 10^4$	$n = 10^6$	$n = 10^9$
$\log n$	2	3	4	6	9
n	100	1000	10^4	10^6	10^9
$n \log n$	200	3000	$4 \cdot 10^4$	$6 \cdot 10^6$	$9 \cdot 10^9$
n^2	10^4	10^6	10^8	10^{12}	10^{18}
$100n^2 + 15n$	$1,0015 \cdot 10^6$	$1,00015 \cdot 10^8$	$\approx 10^{10}$	$\approx 10^{14}$	$\approx 10^{20}$
2^n	$\approx 1,26 \cdot 10^{30}$	$\approx 1,07 \cdot 10^{301}$	?	?	?

1 milhão de operações por segundo

n

Função de custo	10	20	30	40	50	60
n	0,00001s	0,00002s	0,00003s	0,00004s	0,00005s	0,00006s
n^2	0,0001s	0,0004s	0,0009s	0,0016s	0,0025s	0,0036s
n^3	0,001s	0,008s	0,027s	0,064s	0,125s	0,216s
n^5	0,1s	3,2s	24,3s	1,7min	5,2min	12,96min
2^n	0,001s	1,04s	17,9min	12,7dias	35,7 anos	366 séc.
3^n	0,059s	58min	6,5anos	3855séc.	10^8séc.	10^{13}séc.

Influência do tamanho do problema

- Ex.: em um mesmo tempo, aumentar 1000x a velocidade de um computador, resolve:
  - Um problema 10x maior de complexidade $\Theta(n^3)$
  - Um problema 1000x maior se a complexidade for $\Theta(n)$

Função de custo	Computador Atual (C)	Computador 100C	Computador 1000C
n	x	$100x$	$1000x$
n^2	x	$10x$	$31.6x$
n^3	x	$4,6x$	$10x$
2^n	x	$x + 6,6$	$x + 10$

Resumindo

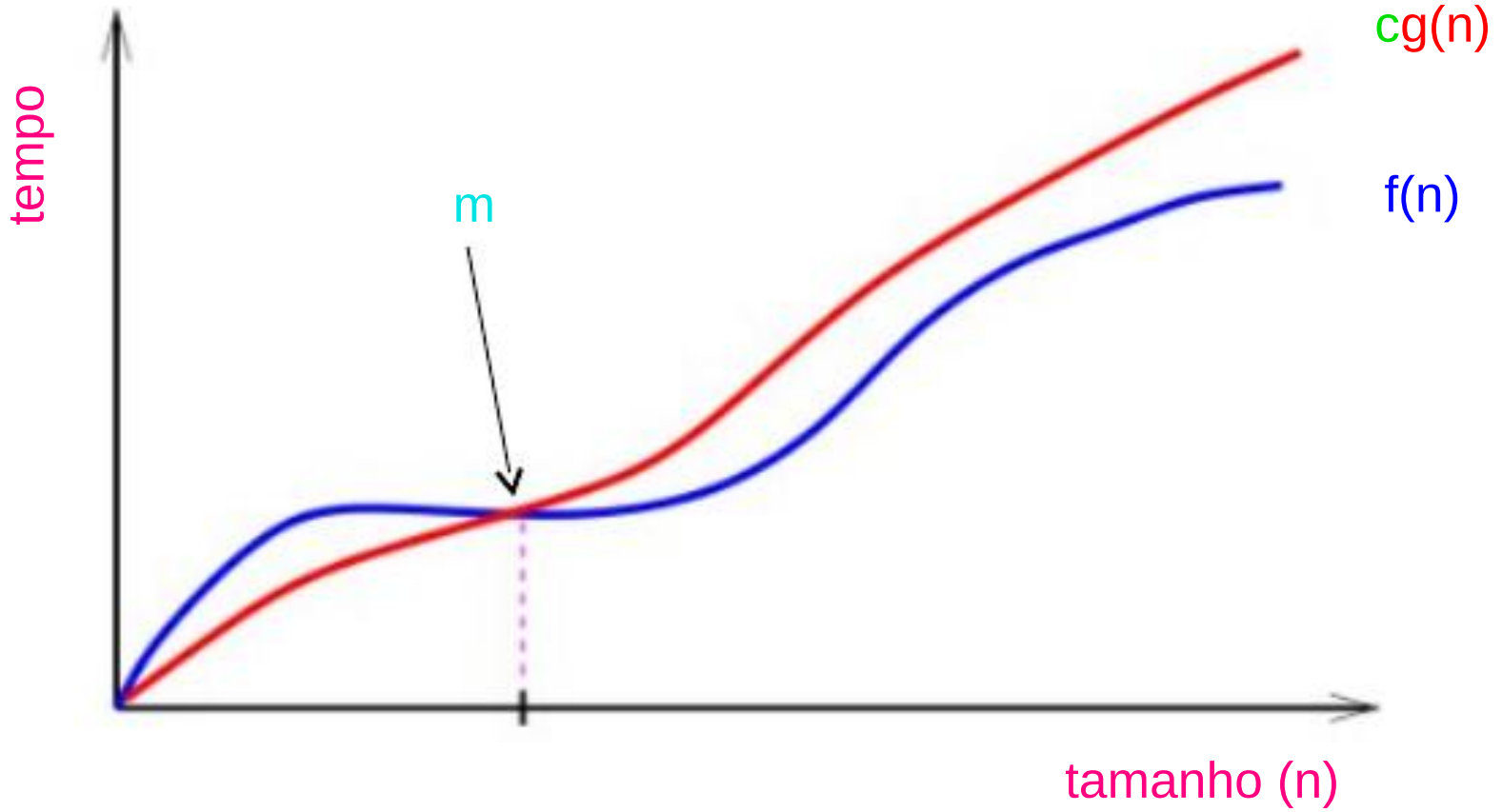
- Se $f(n)$ é a função de complexidade de um algoritmo A
  - O comportamento assintótico de $f(n)$ representa o limite do comportamento do custo (complexidade) de A quando n cresce

- A análise de um algoritmo (função de complexidade)
  - Geralmente considera apenas algumas operações elementares
    - Usa apenas os termos mais importantes

- A complexidade assintótica relata crescimento assintótico (no limite) das operações elementares

Dominar assintoticamente

- Definição: uma função $g(n)$ domina assintoticamente outra função $f(n)$ se existem duas constantes positivas c e m tais que, para $n \geq m$, tem-se $|f(n)| \leq c\,|g(n)|$

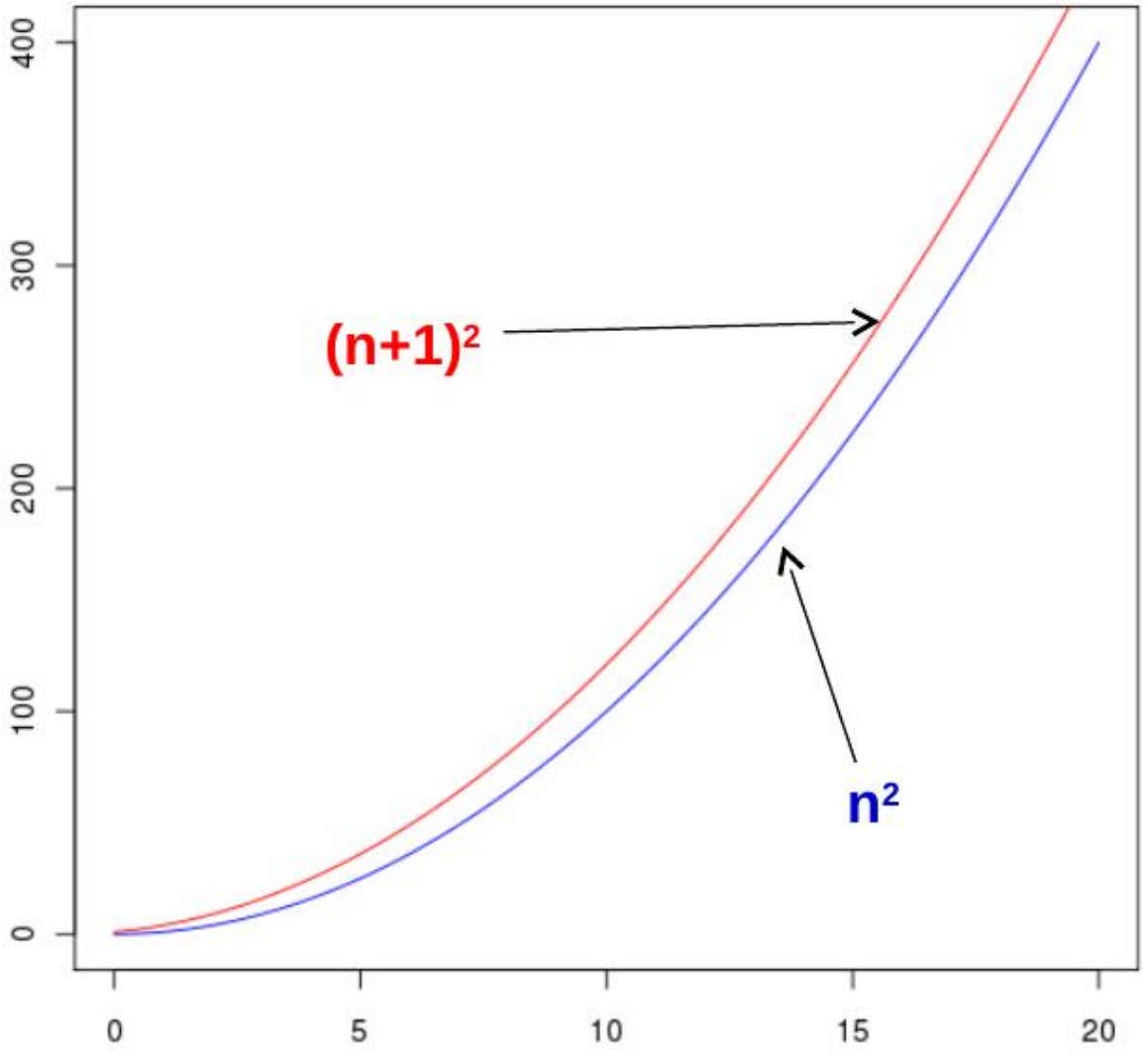


Exemplo 1 – quem domina quem?

- $g(n) = n$ e $f(n) = -n^2$
- Quem domina quem?
- $|n| \leq |-n^2|$ para todo $n \in N$
- Para $c = 1$ e $m = 0$ $\Rightarrow$ $|g(n)| \leq |f(n)|$
- Portanto $f(n)$ domina assintoticamente $g(n)$

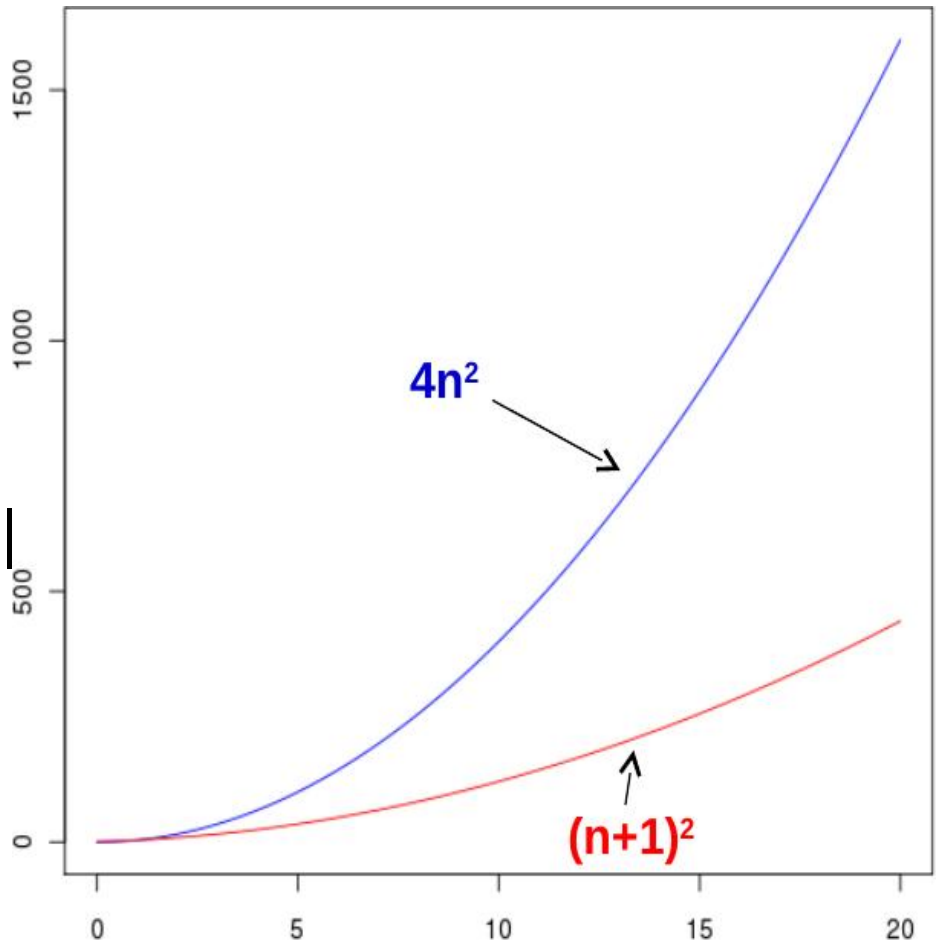
Exemplo 2

- $g(n) = (n + 1)^2$ e $f(n) = n^2$
- $|n^2| \leq |(n+1)^2|$ para $n \geq 0$
- $g(n)$ domina $f(n)$



Exemplo 3

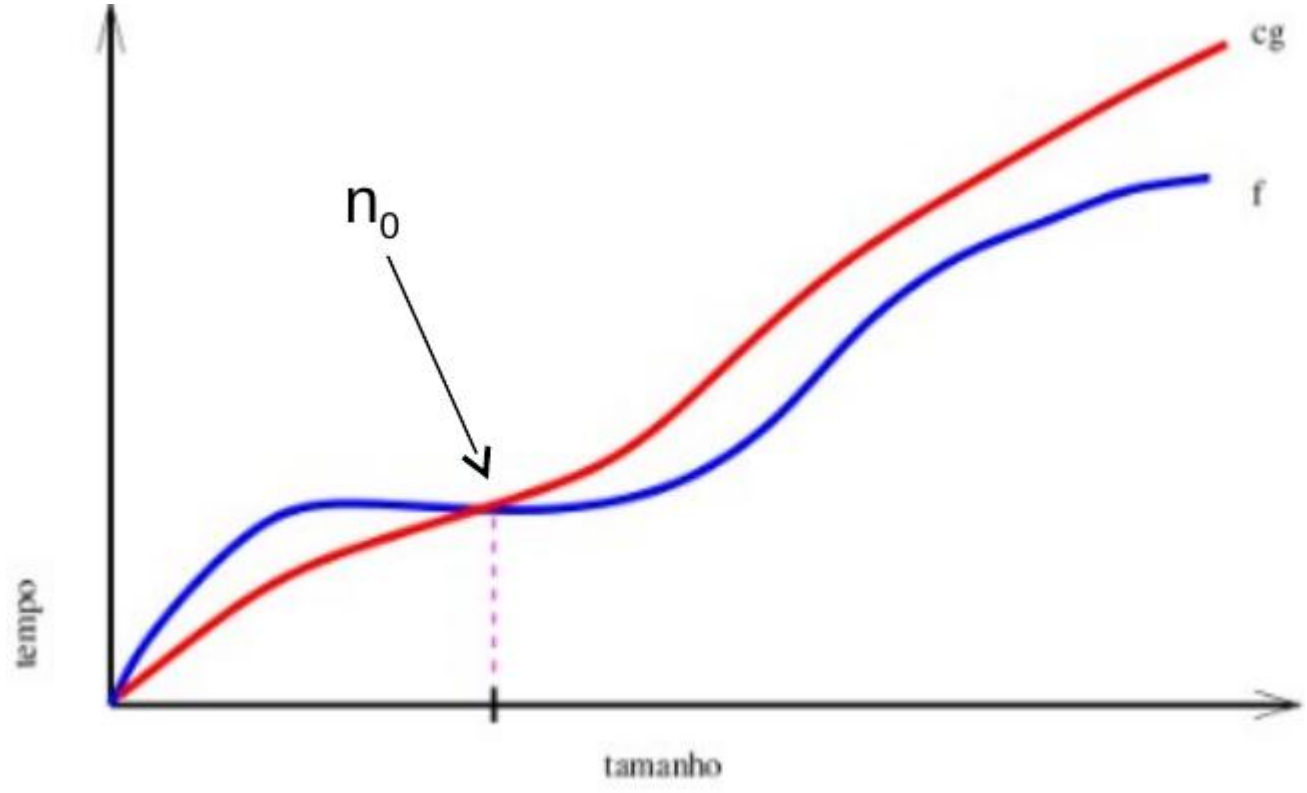
- $g(n) = (n+1)^2$ e $f(n) = n^2$
- Suponha que queremos $g(n) \leq cf(n)$
- Então $|(n+1)^2| \leq |cn^2|$
- Para isso, basta que $|(n+1)^2| \leq |(\sqrt{c}n)^2|$
- Se $\sqrt{c} = 2$, ou seja, $c=4$, isso é verdade
- $|(n+1)^2| \leq |4n^2|$, $\forall\, n \geq 1$
- f(n) domina g(n), $\forall\, n \geq 1$
  - f(n) e g(n) dominam assintoticamente uma a outra



Notação O (lê-se “O grande”)

- Definição: $O(g(n)) = \{f(n)$: existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq cg(n), \forall\ n \geq n_0\}$

- Informalmente dizemos que se $f(n) \in O(g(n))$, então f(n) cresce no máximo tão rapidamente quanto g(n)



Notação O

- Expressa que g(n) domina assintoticamente f(n)
  - Escreve-se f(n)=O(g(n)) e lê-se "f(n) é da ordem no máximo g(n)"
- Usamos para dar um limite superior sobre uma função
- Muitas vezes calcular a função de complexidade g(n) de um algoritmo A é complicado
  - É mais fácil determinar que f(n) é O(g(n))
    - Assintoticamente f(n) cresce no máximo como g(n)
- Podemos descrever o tempo de execução de um algoritmo apenas inspecionando a sua estrutura global

Exemplo

	ordenação por inserção	custo	vezes
1	`for j in range(1, len(A)):`	c_1	n
2	`    chave = A[j]`	c_2	n-1
3	`    # ordenando elementos à esquerda`	0	n-1
4	`    i = j - 1`	c_4	n-1
5	`    while i >= 0 and A[i] > chave:`	c_5	[n(n + 1)]/2 - 1
6	`        A[i + 1] = A[i]`	c_6	[n(n - 1)]/2
7	`        i -= 1`	c_7	[n(n - 1)]/2
9	`    A[i + 1] = chave`	c_8	n-1

(continua)

Exemplos

- Algoritmo *insertion-sort* visto anteriormente, $an^2 + bn - c$
  - Estrutura de laço duplamente aninhado
  - Limite superior $O(n^2)$ no pior caso
    - Custo de cada iteração é limitado acima por $O(1)$ (constante)
    - Os índices *i* e *j* são no máximo *n*
    - O laço interno é executado uma vez para cada um dos n^2 pares de valores *i* e *j* ← $[\mathbf{n(n}+1)]/2 - 1$

- $3/2\ n^2 - 2\ n \in O(n^2)$ $\quad f(n) \leq cg(n)$
  - Fazendo $c=3/2$ teremos $3/2\ n^2 - 2n \leq 3/2\ n^2 \ \forall \ n_0 \geq 2$
  - Outras constantes podem existir, mas o que importa é que existe alguma escolha para as constantes

Notação *O* – o pior caso

- Como a notação *O* dá um limite superior, quando empregado ao pior caso...
  - Indica que esse limite vale para qualquer instância daquele algoritmo

- Assim, o limite $O(n^2)$ do pior caso do *insertion sort* também se aplica a qualquer entrada

- Veremos que o mesmo não é verdadeiro para a notação Θ

Operações com a notação *O*

$$f(n) = O(f(n))$$

$$C \times f(n) = O(f(n)),\ c \text{ é uma constante}$$

$$O(f(n)) + O(f(n)) = O(f(n))$$

$$O(O(f(n))) = O(f(n))$$

$$O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$$

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

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

Operações com a notação *O*

- A regra $O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$ pode ser usada para calcular o tempo de execução de uma sequência de trechos de um programa

- Suponha 3 trechos: $O(n)$, $O(n^2)$ e $O(n \log n)$
  - Qual o tempo de execução do algoritmo como um todo?
    - Lembre-se que o tempo de execução é a soma dos tempos de cada trecho

- $O(n) + O(n^2) + O(n \log n) = max(O(n), O(n^2), O(n \log n)) = O(n^2)$

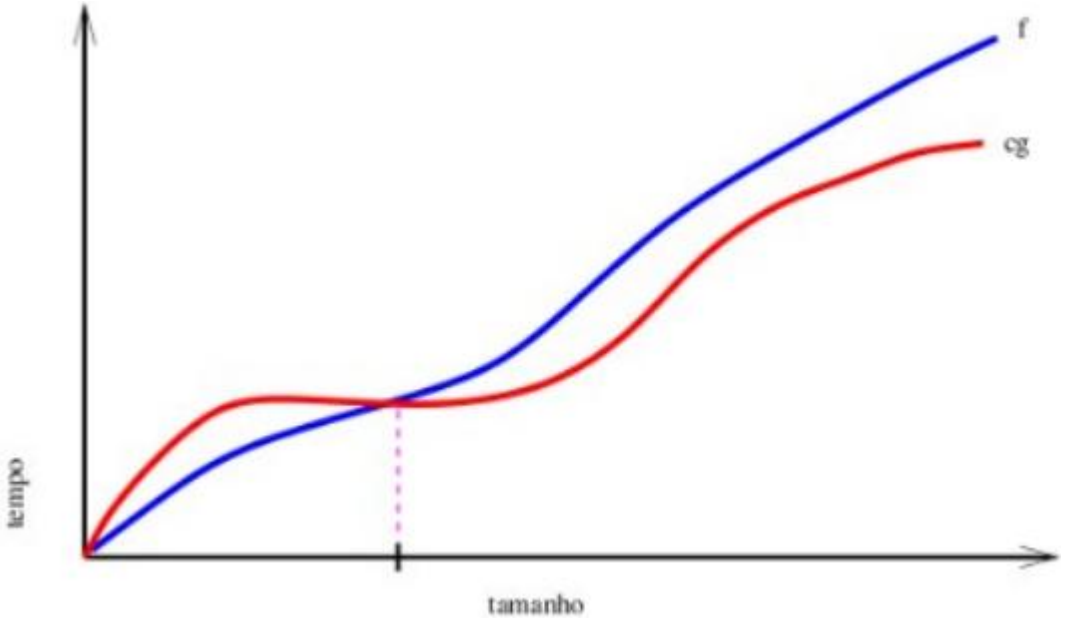
Notação Ω

"big omega de g de n"

- Definição: $\Omega(g(n)) = \{f(n)$: existem constantes positivas c e n_0 tais que $0 \leq cg(n) \leq f(n), \ \forall \ n \geq n_0\}$

- Informalmente dizemos que se $f(n) \in \Omega(g(n))$, então $f(n)$ cresce no mínimo tão lentamente quanto $g(n)$

- Note que se $f(n) \in O(g(n))$ define um limite superior para f(n), $\Omega(g(n))$ define um limite inferior



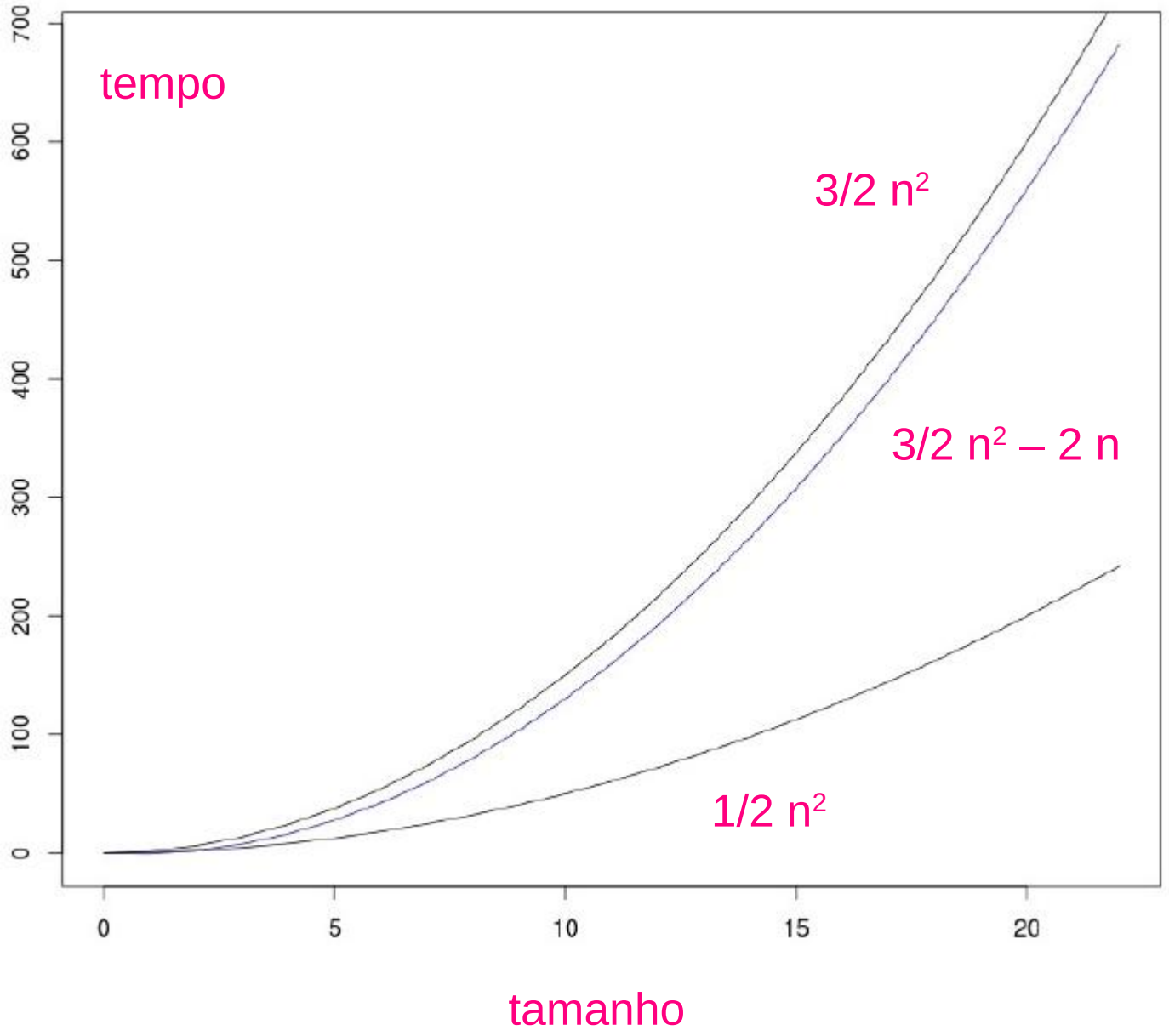
Exemplo

f(n) g(n)

- $3/2\ n^2 - 2\ n \in \Omega(n^2)$

- Fazendo $c=1/2$ teremos $3/2\ n^2 - 2\ n \geq 1/2\ n^2$, $\forall\ n_0 \geq 2$
  - Outras constantes podem existir, mas o que importa é que existe alguma escolha para as constantes

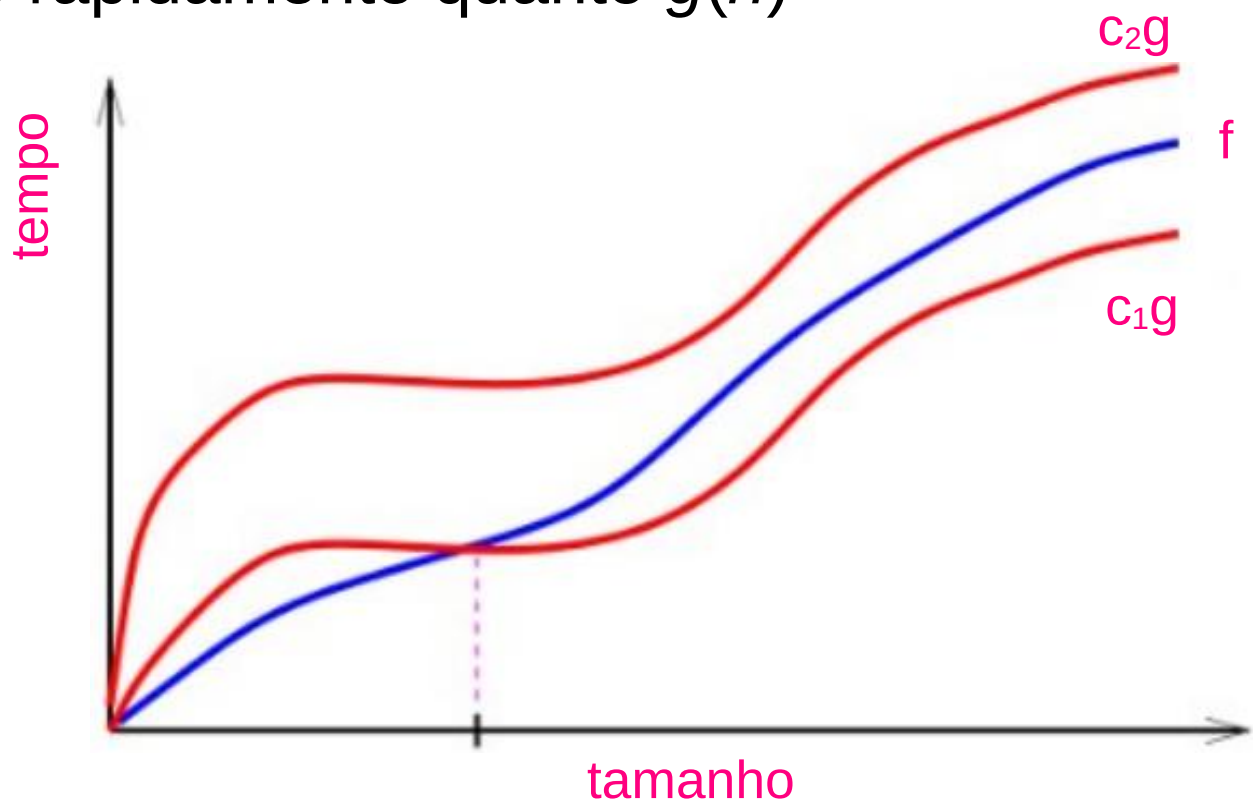
Notação O e Ω



Notação Θ

- Definição: $\Theta(g(n))$ = {$f(n)$: existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, $\forall$ $n \geq n_0$}

- Informalmente dizemos que se $f(n) \in \Theta(g(n))$, então $f(n)$ cresce tão rapidamente quanto $g(n)$



Exemplo

f(n) g(n)

- $3/2\ n^2 - 2\ n \in \Theta(n^2)$

- Fazendo $c_1=1/2$ e $c_2 = 3/2$ teremos
  $|1/2n^2| \leq |3/2n^2 - 2n| \leq |3/2n^2|$, $\forall\ n_0 \geq 2$
  - Outras constantes podem existir, mas o que importa é que existe alguma escolha para as constantes

$3/2\ n^2 - 2\ n \in O(n^2)$	→	$3/2\ n^2 - 2n \leq 3/2\ n^2$
$3/2\ n^2 - 2\ n \in \Omega(n^2)$	→	$3/2\ n^2 - 2\ n \geq 1/2\ n^2$
$3/2\ n^2 - 2\ n \in \Theta(n^2)$	→	$\lvert 1/2n^2 \rvert \leq \lvert 3/2n^2 - 2n \rvert \leq \lvert 3/2n^2 \rvert$

Se $f(n) \in O(g(n))$ e $f(n) \in \Omega(g(n))$, então $f(n) \in \Theta(g(n))$

Notação Θ – pior/melhor caso

- O tempo limite de $\Theta(n^2)$ para o pior caso do *insertion sort*
  - Não implica um tempo $\Theta(n^2)$ para qualquer entrada

- Por exemplo, se pegarmos o **melhor caso**, vemos que ele tem $\Theta(n)$

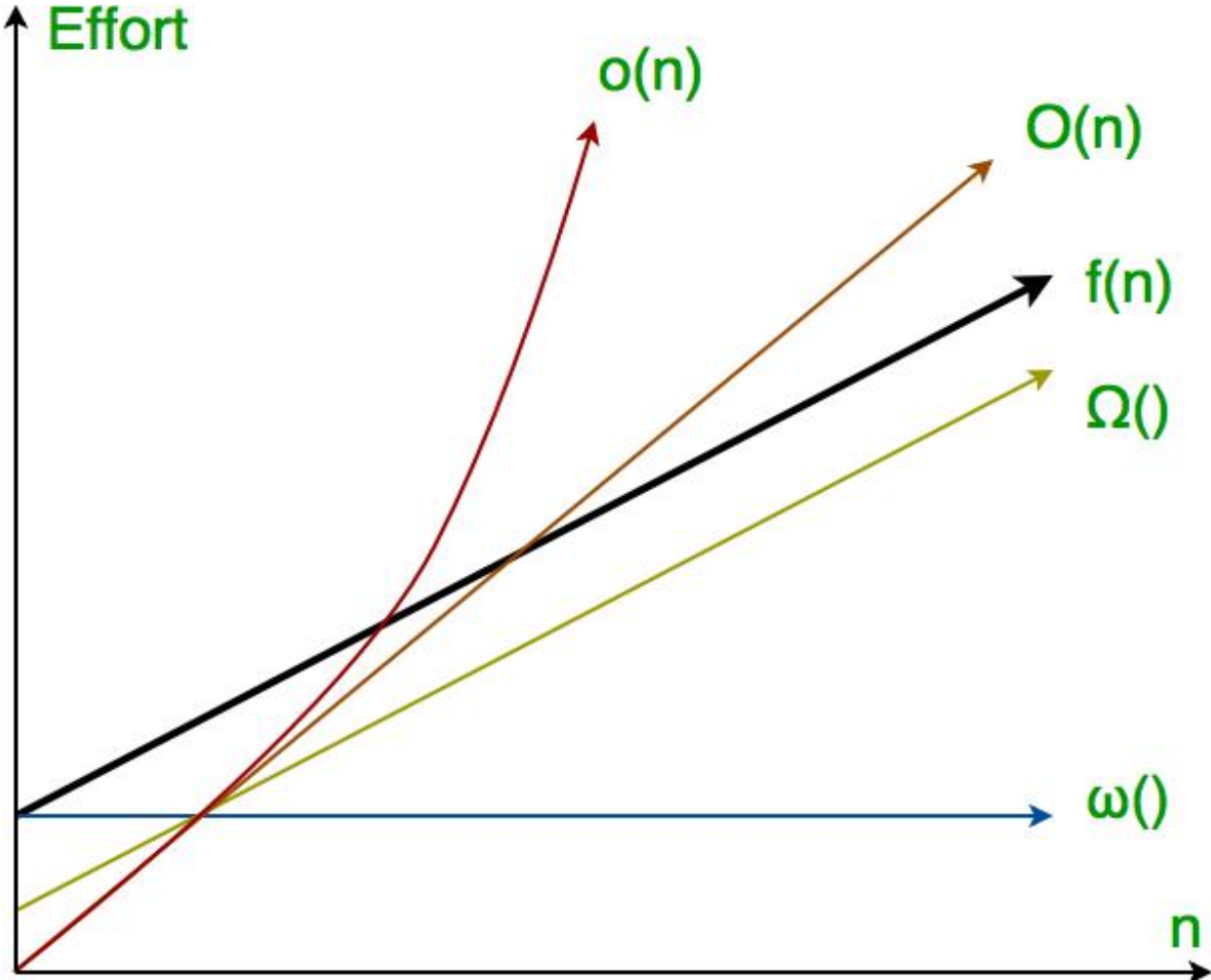
Notação *o*

("o pequeno de g de n")

- $o(g(n)) = \{f(n)$: para toda constante positiva c, existe uma constante $n_0 > 0$ tal que $0 \leq f(n) < cg(n)$, $\forall \; n \geq n_0\}$to

- Informalmente dizemos que se $f(n) \in o(g(n))$, então $f(n)$ cresce mais lentamente que $g(n)$

- Intuitivamente na notação *o* a função $f(n)$ tem crescimento muito menor que $g(n)$ quando n tende para o infinito



Exemplo

- $1000\ n^2 \quad \in \quad o(n^3) \quad ?$

- Para todo valor de *c*, um n_0 que satisfaz a definição é:

$$n_0 = |1000/c| + 1$$

Diferença entre *O* e *o*

- O: existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq cg(n)$, $\forall\ n \geq n_0$
  - A expressão $0 \leq f(n) \leq cg(n)$ é válida para alguma constante $c>0$

- o: para toda constante positiva c, existe uma constante $n_0 > 0$ tal que $0 \leq f(n) < cg(n)$, $\forall\ n \geq n_0$
  - A expressão $0 \leq f(n) < cg(n)$ é válida para toda constante $c>0$

Notação ω

- Definição: $\omega(g(n)) = \{f(n)$: para toda constante positiva c, existe uma constante $n_0 > 0$ tal que $0 \leq cg(n) < f(n)$, para todo $n \geq n_0 \}$
- Informalmente, dizemos que, se $f(n) \in \omega(g(n))$, então $f(n)$ cresce mais rapidamente que $g(n)$
- Intuitivamente, na notação ω, a função $f(n)$ tem crescimento muito maior que $g(n)$ quando n tende para o infinito
- ω está para Ω, da mesma forma que o está para O
  - O e Ω são chamados de assintoticamente firmes

Exemplo

- $|1/1000\ n^2| \in \omega(n)$?

- Para todo valor de *c*, um n_0 que satisfaz a definição é:

  - $n_0 = |1000c| + 1$

Propriedades das classes

- Reflexividade
  - $f(n) \in O(f(n))$
  - $f(n) \in \Omega(f(n))$
  - $f(n) \in \Theta(f(n))$
- Simetria
  - $f(n) \in \Theta(g(n))$ se, e somente se, $g(n) \in \Theta(f(n))$
- Simetria Transposta
  - $f(n) \in O(g(n))$ se, e somente se, $g(n) \in \Omega(f(n))$
  - $f(n) \in o(g(n))$ se, e somente se, $g(n) \in \omega(f(n))$

Propriedades das classes

- Transitividade

Se $f(n) \in O(g(n))$ e $g(n) \in O(h(n))$, então $f(n) \in O(h(n))$

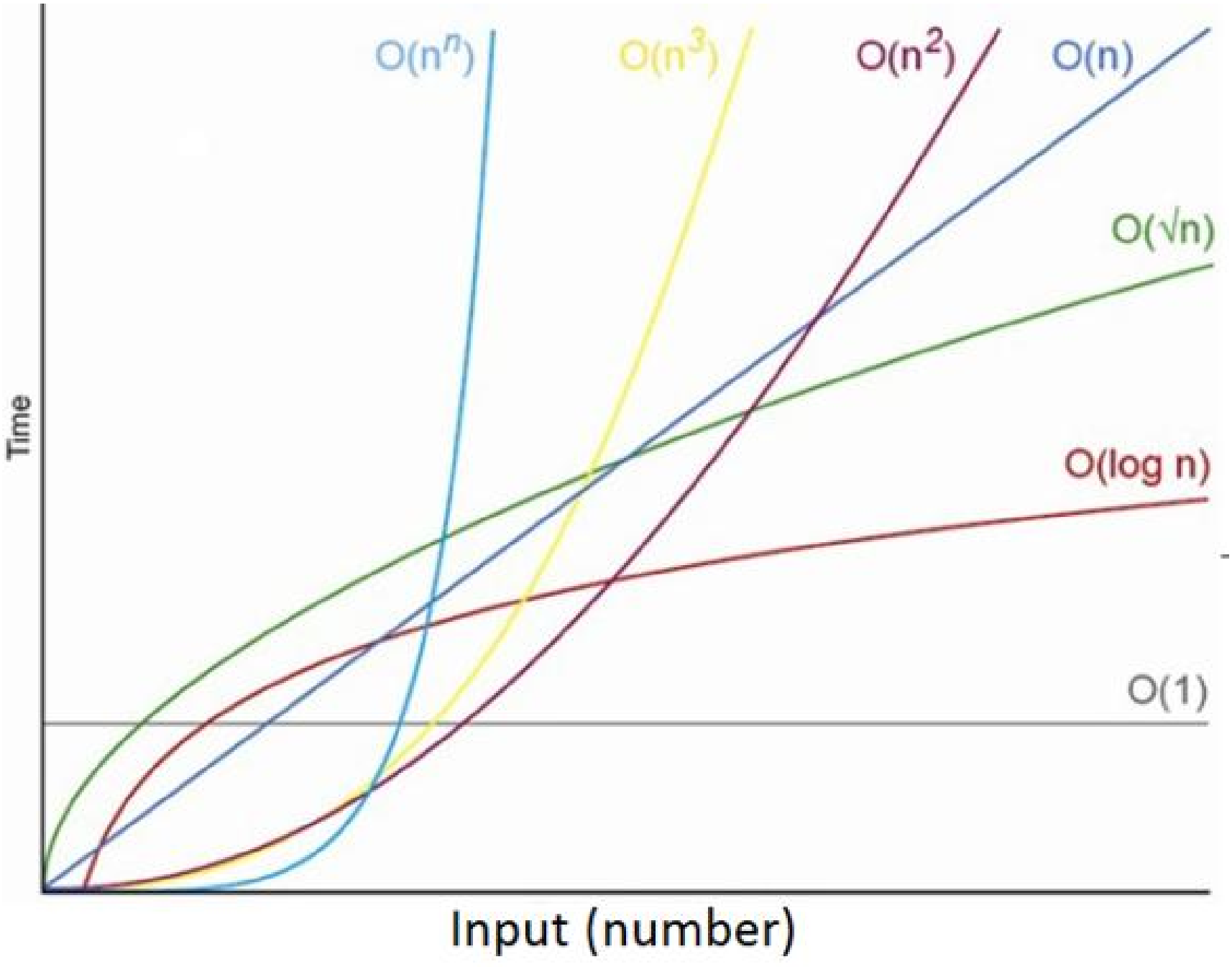
Se $f(n) \in \Omega(g(n))$ e $g(n) \in \Omega(h(n))$, então $f(n) \in \Omega(h(n))$

Se $f(n) \in \Theta(g(n))$ e $g(n) \in \Theta(h(n))$, então $f(n) \in \Theta(h(n))$

Se $f(n) \in o(g(n))$ e $g(n) \in o(h(n))$, então $f(n) \in o(h(n))$

Se $f(n) \in \omega(g(n))$ e $g(n) \in \omega(h(n))$, então $f(n) \in \omega(h(n))$

Hierarquia de funções



Principais Classes de Problemas

Ordem de Complexidade	Nome
$O(1)$	Constante
$O(log\ n)$	Logarítmica
$O(n)$	Linear
$O(n\ log\ n)$	Linearitmica
$O(n^2)$	Quadrática
$O(n^3)$	Cúbica
$O(2^n)$	Exponencial
$O(n!)$	Exponencial

Referências

CORMEN, T. H. Algoritmos - Teoria e Prática. [S. l.]: GEN LTC, 2012.

ROMAN, N. T.; NUNES, F. L. S. Complexidade Assintótica. [S. l.]: USP, 2017. Disponível em: https://edisciplinas.usp.br/mod/resource/view.php?id=2197531&forceview=1.

FORTES, R. Análise de Algoritmos. [S. l.]: Universidade Federal de Ouro Preto, UFOP, 2014. Disponível em: http://www.decom.ufop.br/reinaldo/site_media/uploads/2014-01-bcc202/aulas/aula_04_-_analise_de_algoritmos_(parte_1)_(v1).pdf.