

**FCT/Unesp – Presidente Prudente**  
**Departamento de Matemática e Computação**

# Análise Assintótica

## Parte II

Prof. Danilo Medeiros Eler  
danilo.eler@unesp.br

# Eficiência de Algoritmos

- Algoritmos criados para resolver um mesmo problema podem diferenciar de forma quanto a sua eficiência

$n$	BubbleSort Tradicional	QuickSort	HeapSort	ShellSort	InsertionSort	SelectionSort	MergeSort
100	0	0.002	0	0	0	0	0
200	0.002	0.002	0.001	0	0	0.001	0
300	0.004	0.001	0.001	0.001	0.001	0.002	0.001
400	0.007	0.002	0.001	0	0.002	0.003	0.001
500	0.01	0.003	0.001	0	0.003	0.004	0.001
600	0.015	0.003	0.002	0	0.003	0.007	0.001
700	0.02	0.002	0.001	0.001	0.005	0.009	0.002
800	0.028	0.003	0.002	0.001	0.007	0.011	0.003
900	0.033	0.002	0.002	0.002	0.01	0.015	0.003
1000	0.042	0.003	0.002	0.001	0.011	0.018	0.003
2000	0.173	0.003	0.006	0.003	0.055	0.075	0.007
3000	0.449	0.006	0.009	0.006	0.095	0.155	0.01
4000	0.739	0.007	0.013	0.008	0.167	0.271	0.014
5000	1.18	0.009	0.016	0.009	0.26	0.423	0.017
7000	2.395	0.011	0.024	0.015	0.508	0.826	0.024
8000	3.17	0.012	0.028	0.017	0.658	1.075	0.027
9000	4.058	0.014	0.032	0.019	0.836	1.359	0.032
10000	5.052	0.016	0.036	0.022	1.034	1.677	0.035
20000	21.139	0.033	0.077	0.05	4.053	6.689	0.073
25000	33.122	0.041	0.099	0.065	6.306	10.446	0.092
30000	48.01	0.05	0.121	0.079	9.175	15.037	0.114
40000	86.402	0.068	0.167	0.108	16.101	26.712	0.152

# Análise de Algoritmos

- Possíveis técnicas de análise
  - Experimentação
  - Análise assintótica

# Análise de Algoritmos

- Existem vários componentes que precisamos definir antes de descrever uma metodologia de análise de algoritmos baseada em funções matemáticas
  - Uma **linguagem** para descrição de algoritmos
  - Um **modelo computacional** para execução de algoritmos
  - Uma **métrica** para medir o tempo de execução de algoritmos

# Análise de Algoritmos

Melhor Caso de Execução: entrada em ordem crescente

```
int menor(int vetor[], int n){
```

vetor = {1, 2, 3, 4, 5}

```
    int menor = MAX_INT; → 1
```

```
    para i=1 ate n faça → 1
```

```
        se (vetor[i] < menor) → 1
```

```
            menor = vetor[i]; → 1
```

```
    retorna(menor); → 1
```

```
}
```

Número de Instruções:  $1 + 1*n + 1 + 1 = n + 3$

# Análise de Algoritmos

Pior Caso de Execução: entrada em ordem decrescente

```
int menor(int vetor[], int n){
```

vetor = {5, 4, 3, 2, 1}

```
    int menor = MAX_INT; → 1
```

```
    para i=1 ate n faça
```

```
        se (vetor[i] < menor) → 1
```

```
            menor = vetor[i]; → 1
```

x n




```
    retorna(menor); → 1
```

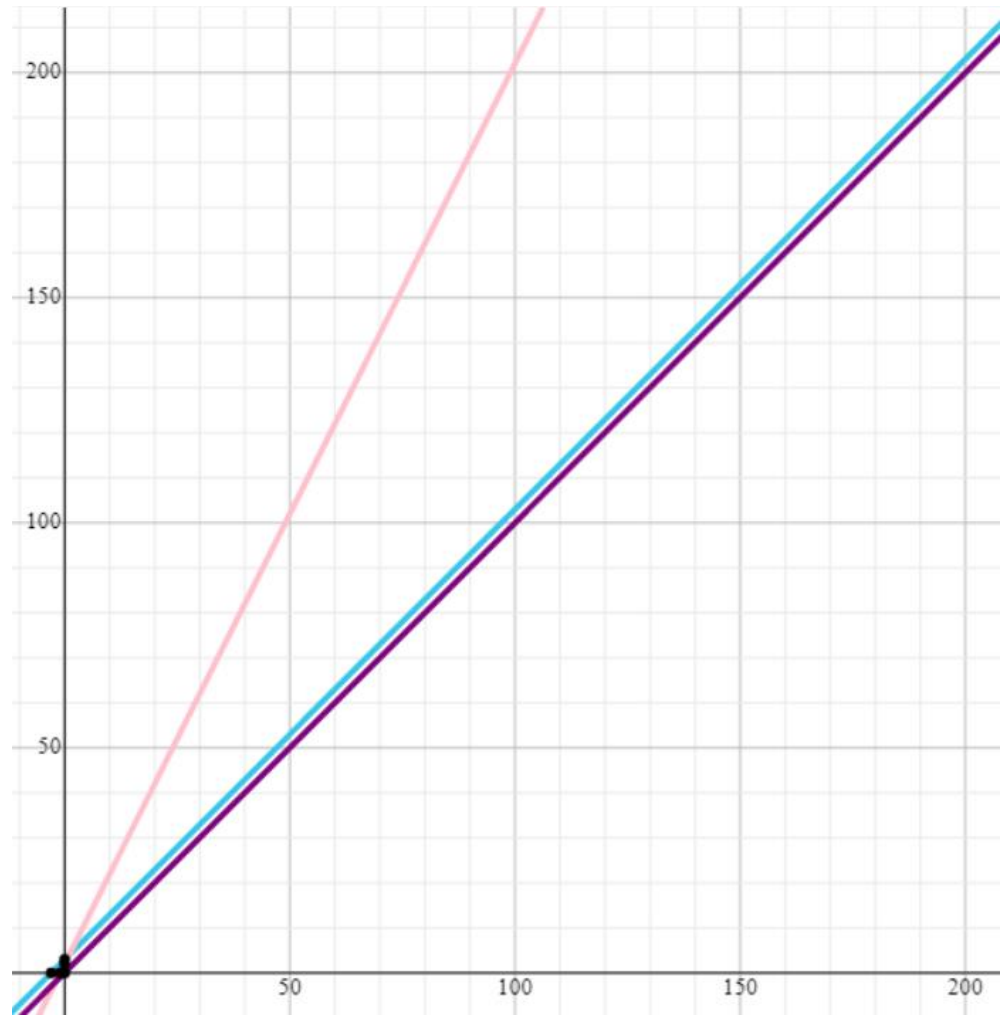
```
}
```

Número de Instruções:  $1 + 2*n + 1 = 2n + 2$

# Análise de Algoritmos




## ■ Comportamento de uma função linear

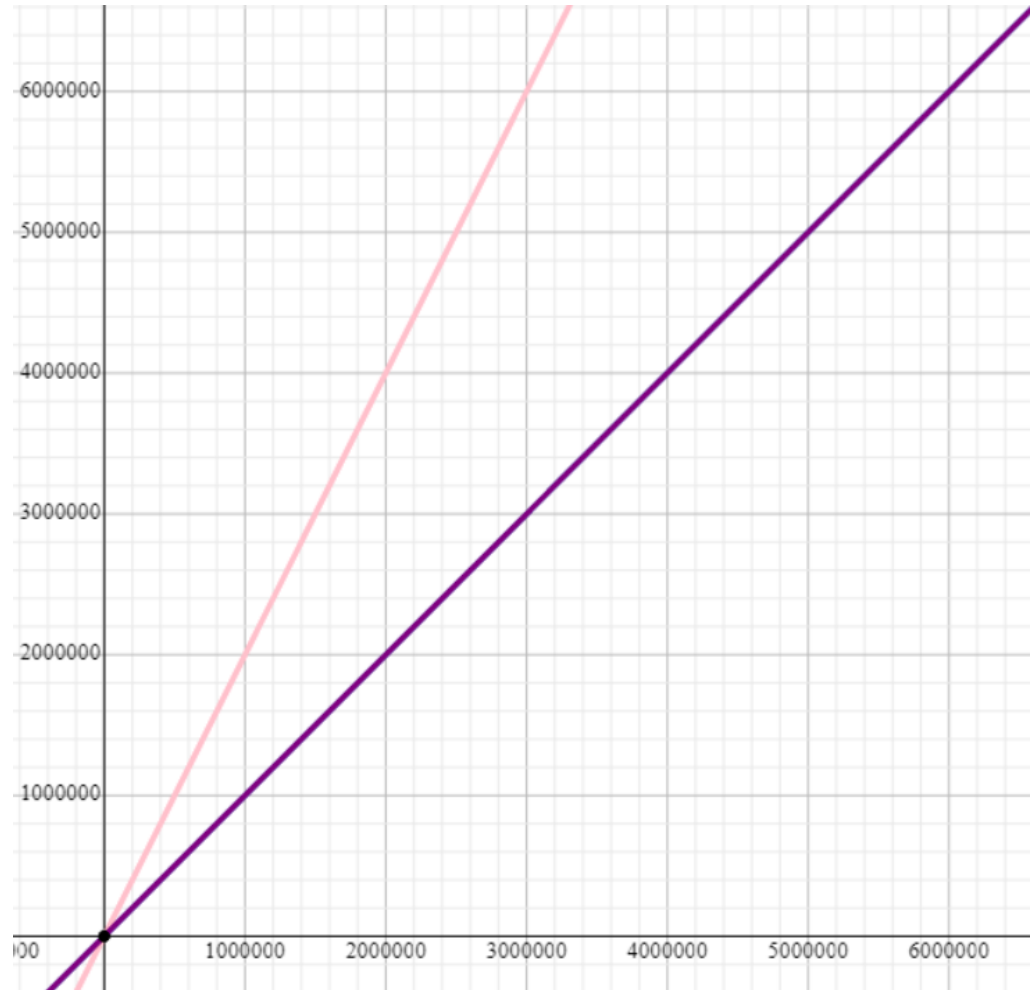
	$n + 3$	Melhor Caso
	$2n + 2$	Pior Caso
	$n$	



# Análise de Algoritmos

## ■ Comportamento de uma função linear






	$n + 3$	Melhor Caso
	$2n + 2$	Pior Caso
	$n$	

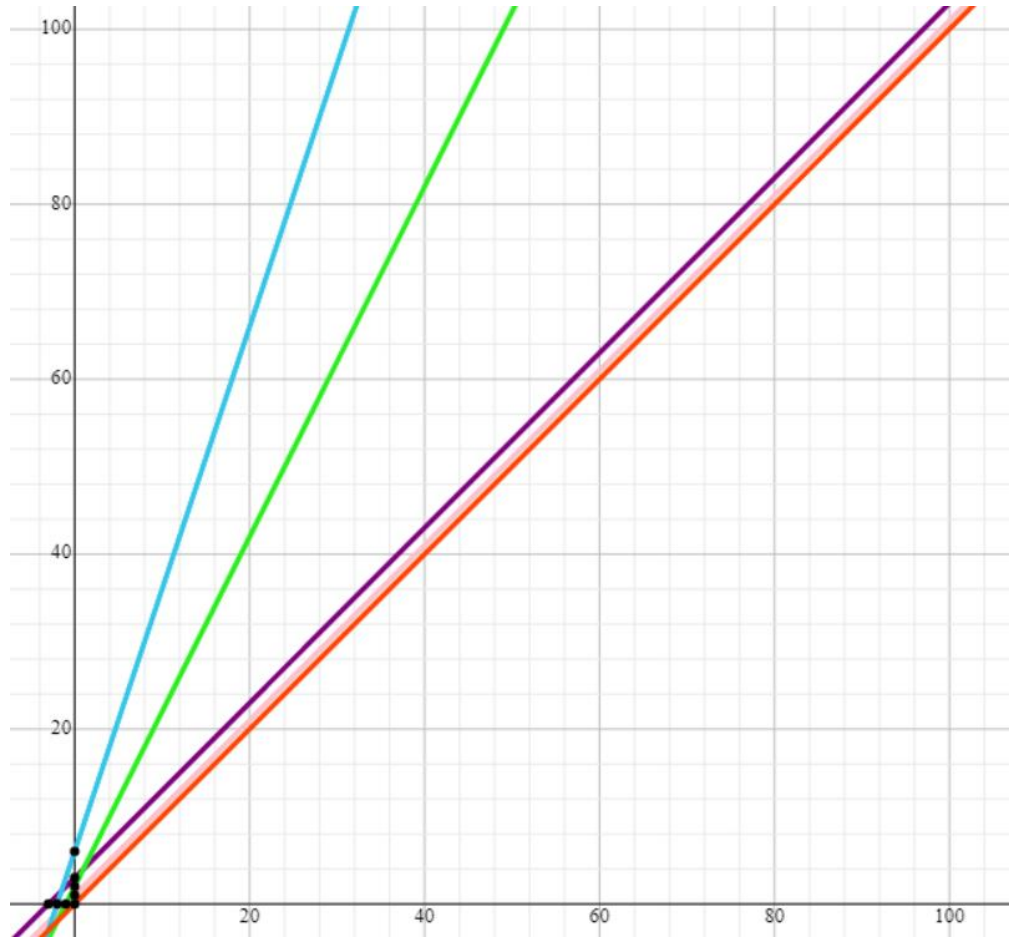




# Análise de Algoritmos






## ■ Funções com comportamento linear

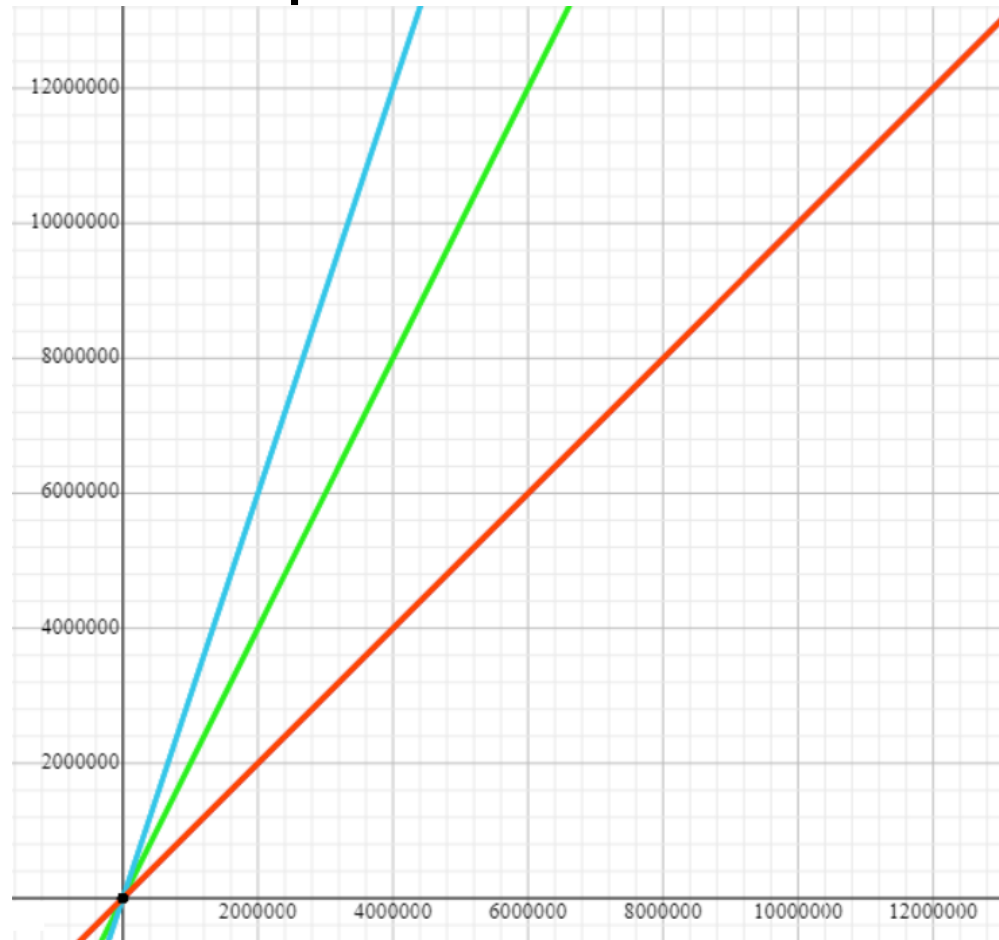
	$n + 1$
	$n + 3$
	$2n + 2$
	$3n + 6$
	$n$



# Análise de Algoritmos





## ■ Funções com comportamento linear

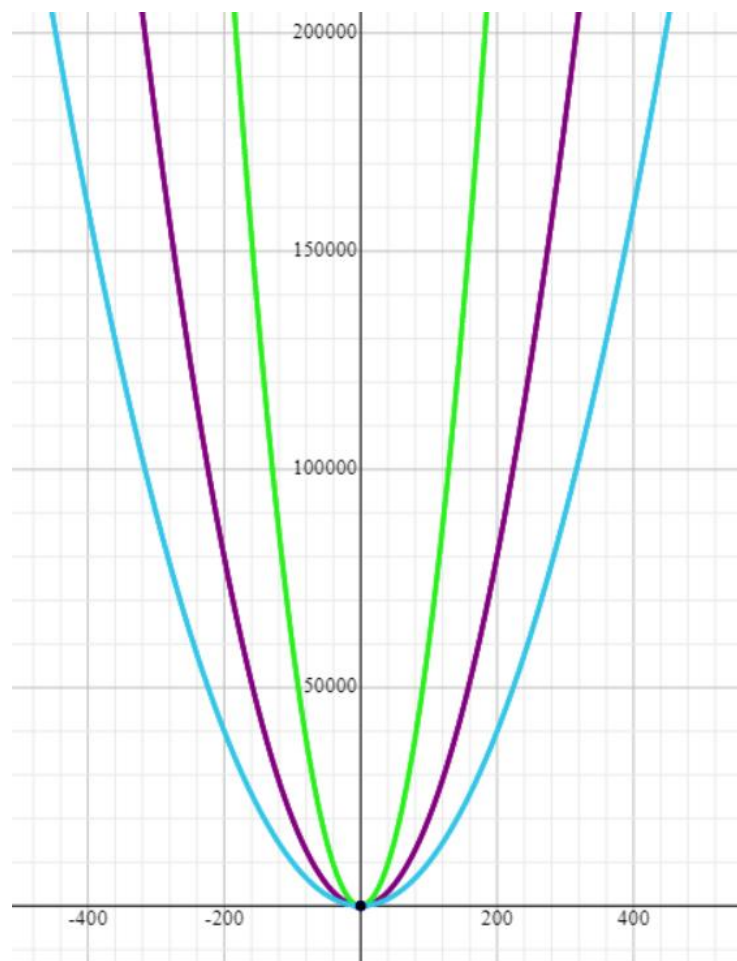
	$n + 1$
	$n + 3$
	$2n + 2$
	$3n + 6$
	$n$



# Análise de Algoritmos





## ■ Funções com comportamento quadrático

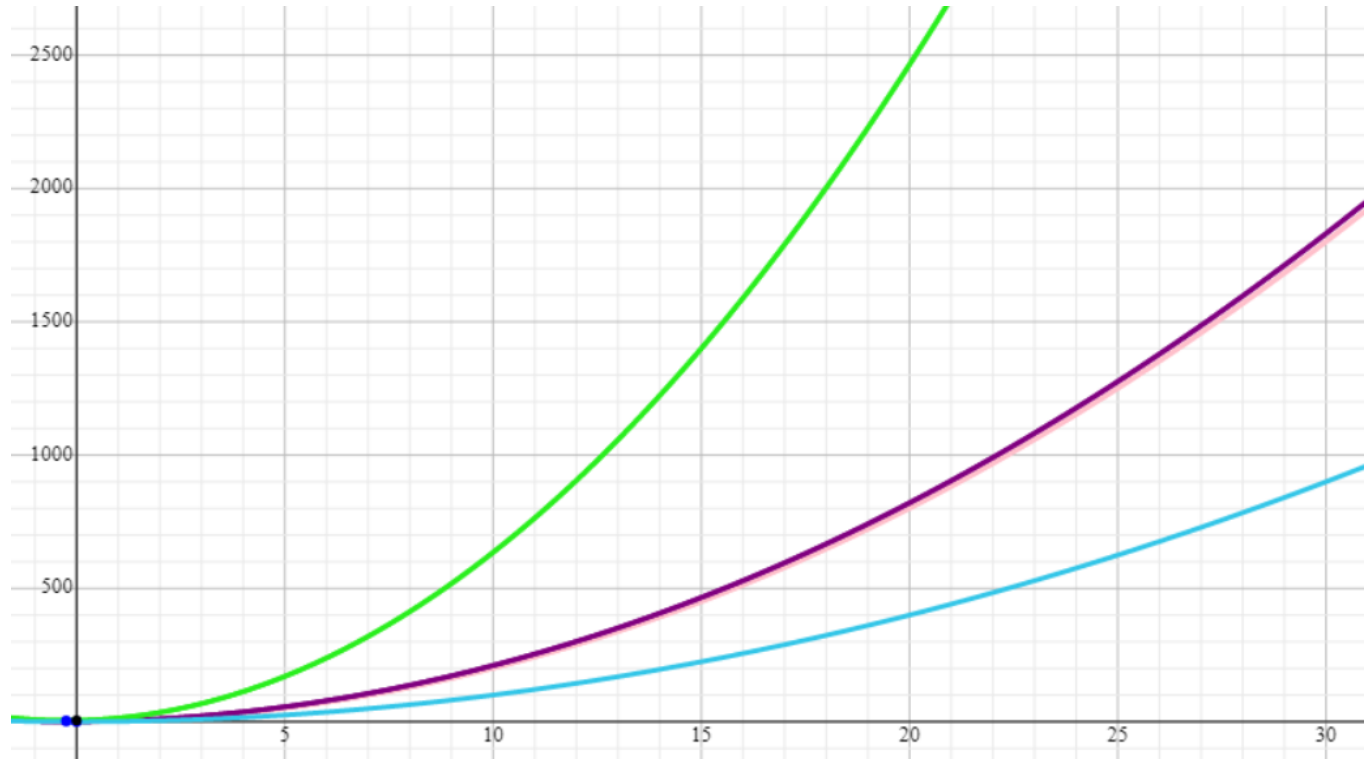
	$2n^2 + 1$
	$2n^2 + n + 1$
	$6n^2 + 3n + 5$
	$n^2$



# Análise de Algoritmos

## ■ Funções com comportamento quadrático

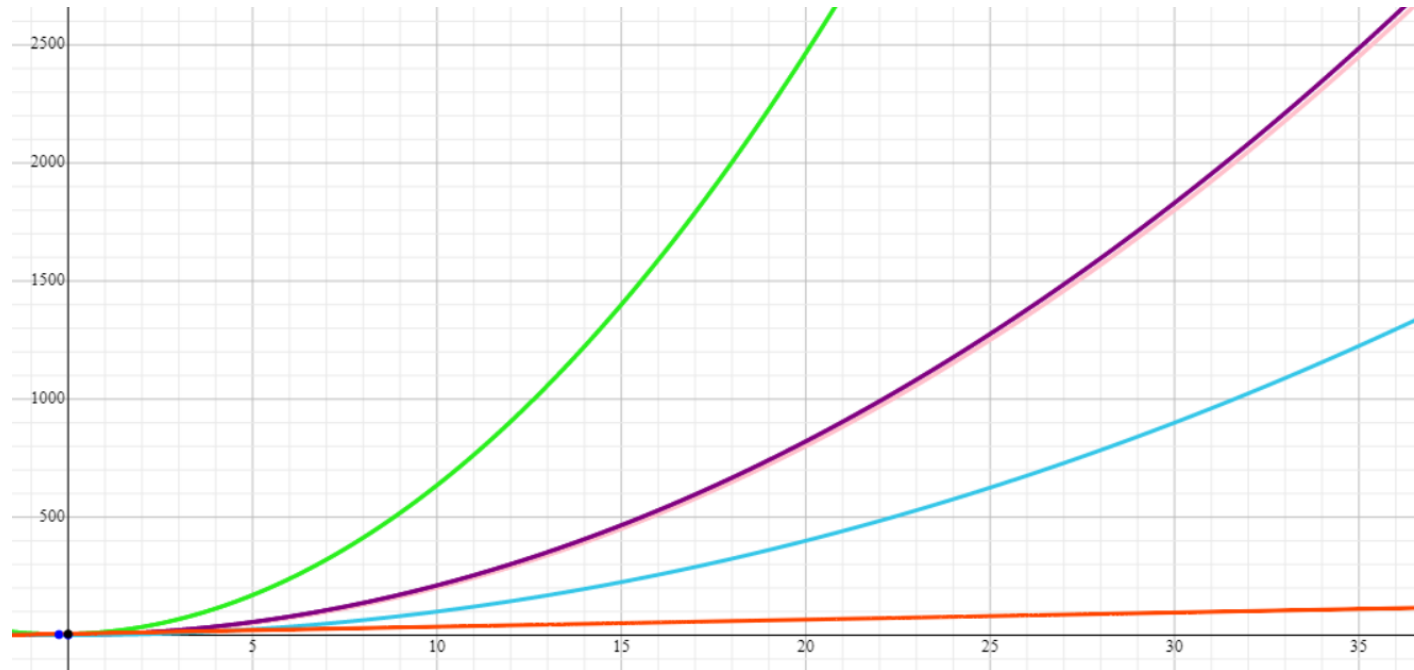
	$2n^2 + 1$
	$2n^2 + n + 1$
	$6n^2 + 3n + 5$
	$n^2$



# Análise de Algoritmos






## ■ Comparando o crescimento das funções

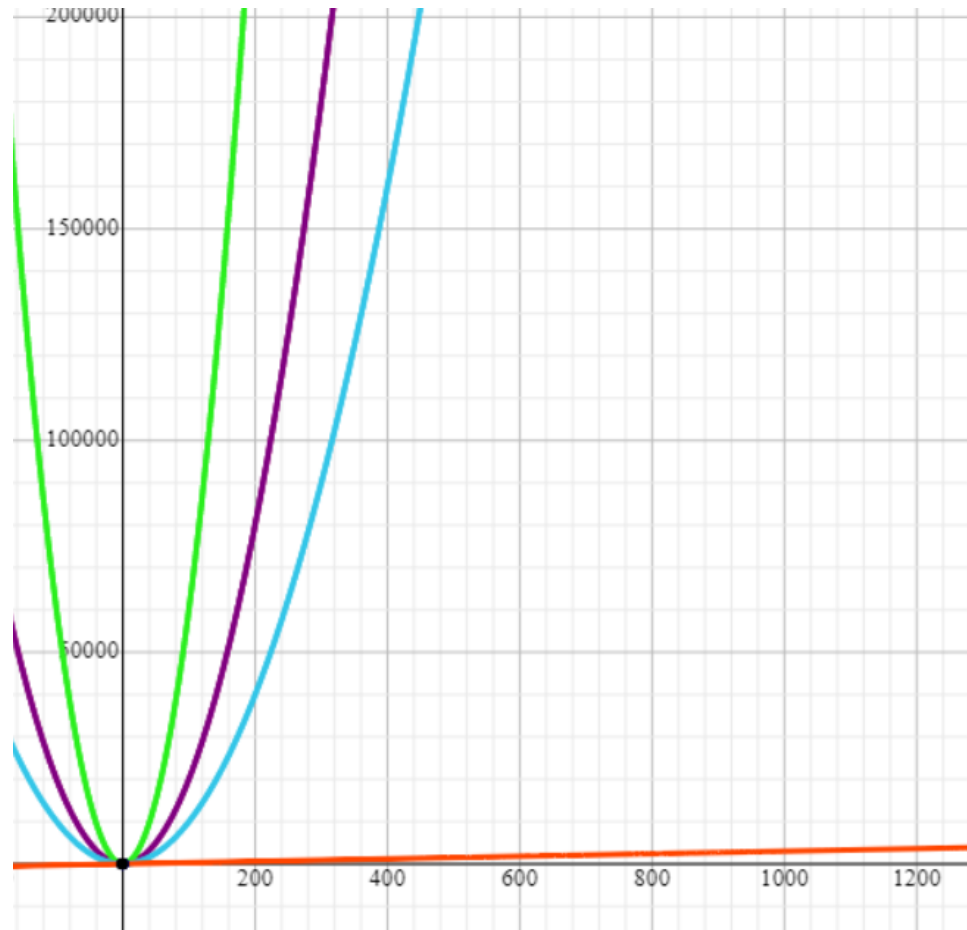
●	$2n^2 + 1$
●	$2n^2 + n + 1$
●	$6n^2 + 3n + 5$
●	$n^2$
●	$3n + 6$



# Análise de Algoritmos

## ■ Comparando o crescimento das funções

	$2n^2 + 1$
	$2n^2 + n + 1$
	$6n^2 + 3n + 5$
	$n^2$
	$3n + 6$



# Comparando o crescimento das funções

## ■ Lineares

- Ex.:  $n + 1$ ;  $n + 3$ ;  $2n + 2$ ;  $3n + 6$
- Podem ser representadas por  $n$

## ■ Quadráticas

- Ex.:  $2n^2 + 1$ ;  $2n^2 + n + 1$ ;  $6n^2 + 3n + 5$
- Podem ser representadas por  $n^2$

# Comparando o crescimento das funções

- Crescimento da função quadrática é maior do que o crescimento da função linear
  - $n < n^2$
- Podemos indicar essa relação com a notação assintótica
  - $n = O(n^2)$
- Nossos algoritmos serão descritos com essa notação, por exemplo:
  - $n+3 \rightarrow O(n)$
  - $2n+2 \rightarrow O(n)$
  - $2n^2 + n + 1 \rightarrow O(n^2)$



# Análise de Algoritmos

- A notação  $O$  indica um limite superior, ou seja, uma determinada função  $f(x)$  está limitada ao crescimento de uma função  $g(x)$ 
  - $f(x) = O(g(x))$
  - Assintoticamente,  $g(x)$  domina  $f(x)$
  - $g(x)$  apresenta um crescimento igual ou superior a  $f(x)$
- Por isso, é relacionada ao pior caso de execução de um algoritmo

# Comparando o crescimento das funções

- Crescimento da função linear é menor do que o crescimento da função quadrática
  - $n^2 > n$
- Podemos indicar essa relação com a notação assintótica
  - $n^2 = \Omega(n)$
- Nossos algoritmos também serão descritos com essa notação, por exemplo:
  - $n+3 \rightarrow \Omega(n)$
  - $2n+2 \rightarrow \Omega(n)$
  - $2n^2 + n + 1 \rightarrow \Omega(n^2)$

# Análise de Algoritmos

- A notação  $\Omega$  indica um limite inferior, ou seja, uma determinada função  $f(x)$  está inferiormente limitada ao crescimento de uma função  $g(x)$ 
  - $f(x) = \Omega(g(x))$
  - Assintoticamente,  $g(x)$  é dominada por  $f(x)$
  - $g(x)$  apresenta um crescimento igual ou inferior a  $f(x)$
- Por isso, é relacionada ao melhor caso de execução de um algoritmo

# Análise de Algoritmos

## ■ Calcular melhor e pior caso

```
int menor(int vetor[], int n){
    int menor = MAX_INT;
    para i=1 ate n faça
        se (vetor[i] < menor)
            menor = vetor[i];
    se menor < 0
        para i=1 ate n faça
            para j=1 ate n faça
                vetor[i] = vetor[i]^(i+j);
    retorna(menor);
}
```

# Análise de Algoritmos

- Melhor caso: vetor em ordem crescente com elementos positivos (ou menor valor maior ou igual a zero)

vetor = {1, 2, 3, 4, 5}

```
int menor(int vetor[], int n){
  int menor = MAX_INT;
  para i=1 ate n faça
    se (vetor[i] < menor)
      menor = vetor[i];
  se menor < 0
    para i=1 ate n faça
      para j=1 ate n faça
        vetor[i] = vetor[i]^(i+j);
  retorna(menor);
}
```

Diagram illustrating the number of instructions for each line of code in the algorithm:

- int menor = MAX\_INT; → 1
- para i=1 ate n faça → x n
- se (vetor[i] < menor) → 1
- menor = vetor[i]; → 1
- se menor < 0 → 1
- para i=1 ate n faça → 1
- para j=1 ate n faça → 1
- vetor[i] = vetor[i]^(i+j); → 1
- retorna(menor); → 1

Número de Instruções:  $1 + 1*n + 1 + 1 + 1 = n + 4$

# Análise de Algoritmos

- Pior caso: vetor em ordem decrescente e menor elemento negativo

vetor = {5, 4, 3, 2, -5}

```
int menor(int vetor[], int n){
  int menor = MAX_INT;
  para i=1 ate n faça
    se (vetor[i] < menor)
      menor = vetor[i];
  se menor < 0
    para i=1 ate n faça
      para j=1 ate n faça
        vetor[i] = vetor[i]^(i+j);
  retorna(menor);
}
```

Diagram illustrating the execution flow of the algorithm with instruction counts:

- Initialization: `int menor = MAX_INT;` (1 instruction)
- Outer loop: `para i=1 ate n faça` (1 instruction, repeated  $n$  times)
- Inner loop condition: `se (vetor[i] < menor)` (1 instruction, repeated  $n$  times)
- Assignment: `menor = vetor[i];` (1 instruction, repeated  $n$  times)
- Condition: `se menor < 0` (1 instruction)
- Inner loop: `para i=1 ate n faça` (1 instruction, repeated  $n$  times)
- Inner loop: `para j=1 ate n faça` (1 instruction, repeated  $n$  times)
- Inner loop: `vetor[i] = vetor[i]^(i+j);` (1 instruction, repeated  $n$  times)
- Return: `retorna(menor);` (1 instruction)



Número de Instruções:  $1 + 2*n + 1 + 1*n*n + 1 = n^2 + 2n + 3$

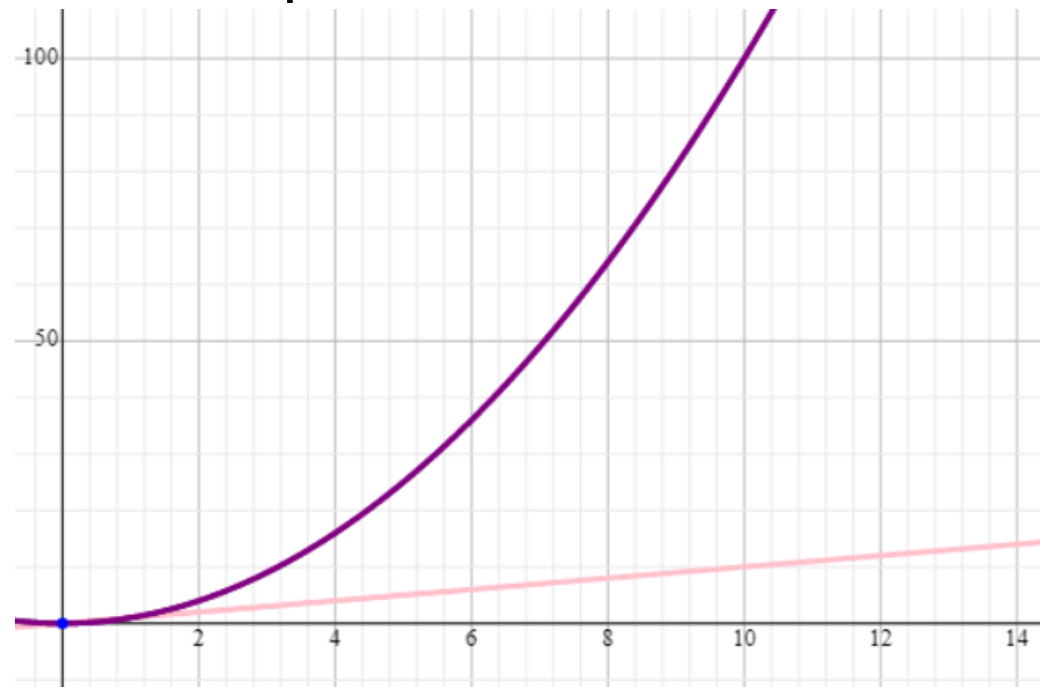
# Análise de Algoritmos

- Melhor Caso
  - $1 + 1*n + 1 + 1 + 1 = n + 4$  (linear)
- Pior Caso
  - $1 + 2*n + 1 + 1*n*n + 1 = n^2 + 2n + 3$  (quadrático)
- A complexidade desse algoritmo pode ser descrita como:  $\Omega(n)$  e  $O(n^2)$ 
  - Executa no mínimo com comportamento linear
  - Executa no máximo com comportamento quadrático

# Análise de Algoritmos

- A complexidade do algoritmo pode ser descrita como
  - $\Omega(n)$  → limitante inferior
  - $O(n^2)$  → limitante superior



	$n$
	$n^2$





# Análise de Algoritmos

- A complexidade do algoritmo pode ser descrita como
  - $\Omega(n)$  → limitante inferior
  - $O(n^2)$  → limitante superior

	$n$
	$n^2$





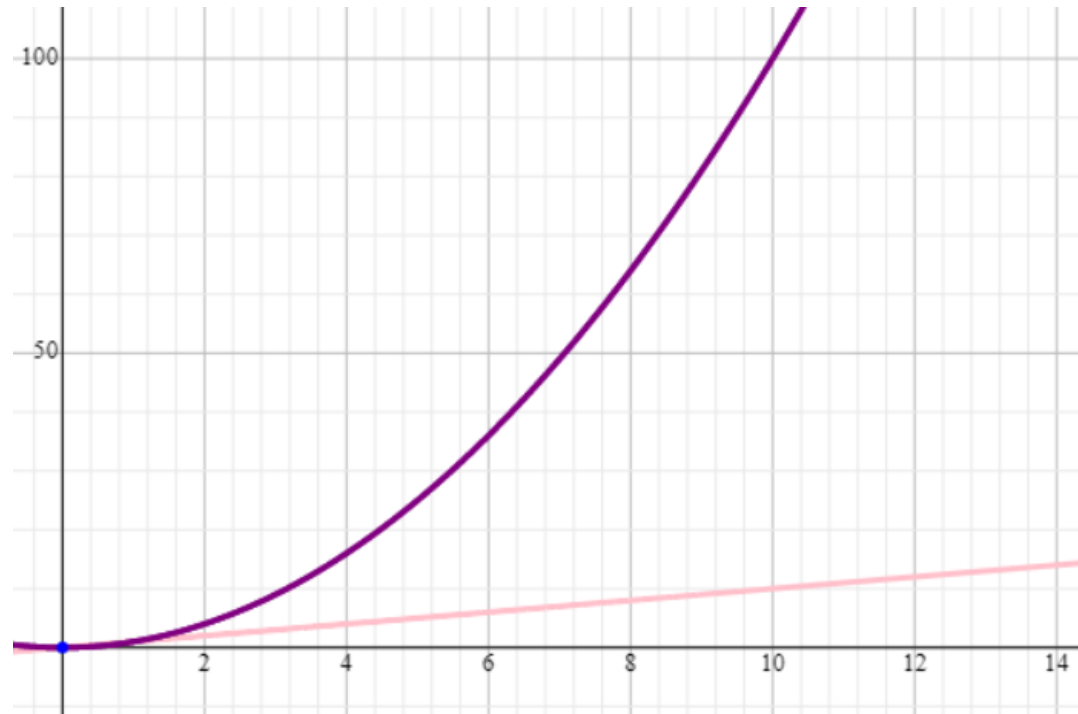
# Análise de Algoritmos

- Um erro comum é utilizar a notação  $O$  para descrever os casos dos algoritmos. Exemplo:
  - Melhor caso é  $O(n)$
  - Pior caso é  $O(n^2)$
- A notação  $O$  é um limitante superior, deixando aberto a interpretação para as execuções que estão abaixo desse limitante
  - Deve-se tomar cuidado
- Por isso, nesse caso, a complexidade do algoritmo é descrita como:  $\Omega(n)$  e  $O(n^2)$

# Análise de Algoritmos



- Por exemplo
  - Melhor caso é  $O(n)$
  - Pior caso é  $O(n^2)$

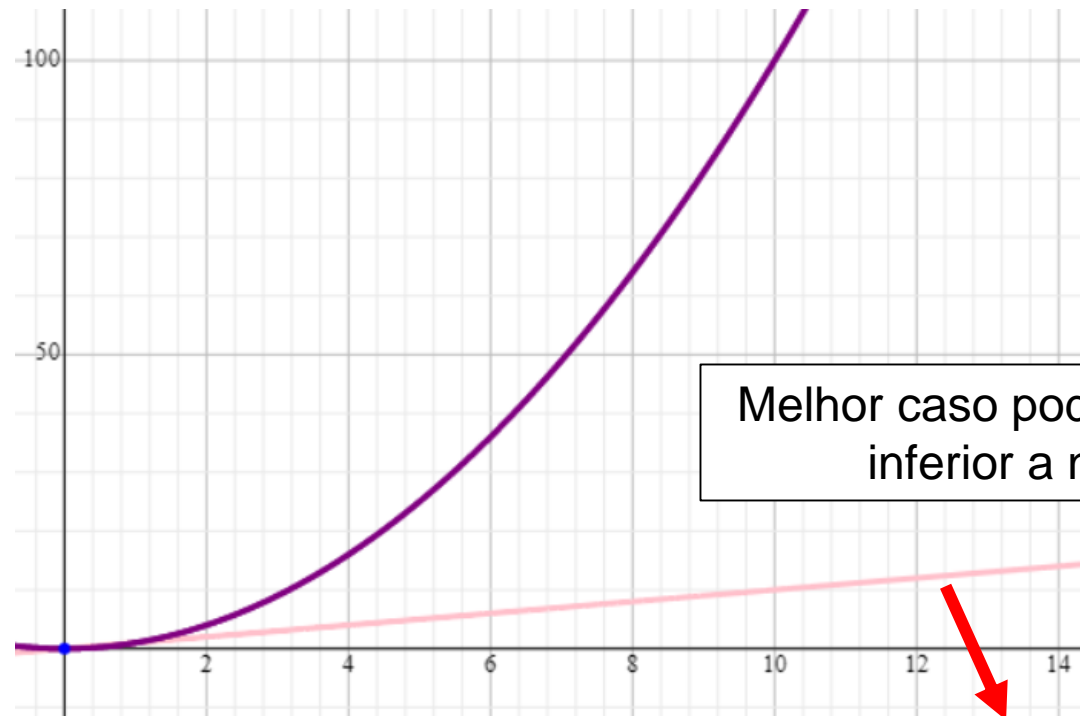
	$n$
	$n^2$



# Análise de Algoritmos



- Por exemplo
  - Melhor caso é  $O(n)$
  - Pior caso é  $O(n^2)$

	$n$
	$n^2$



# Análise de Algoritmos



- Por exemplo
  - Melhor caso é  $O(n)$
  - Pior caso é  $O(n^2)$

	$n$
	$n^2$



# Análise de Algoritmos

- A complexidade do algoritmo é descrita como
  - $\Omega(n)$  → limitante inferior
  - $O(n^2)$  → limitante superior

	$n$
	$n^2$

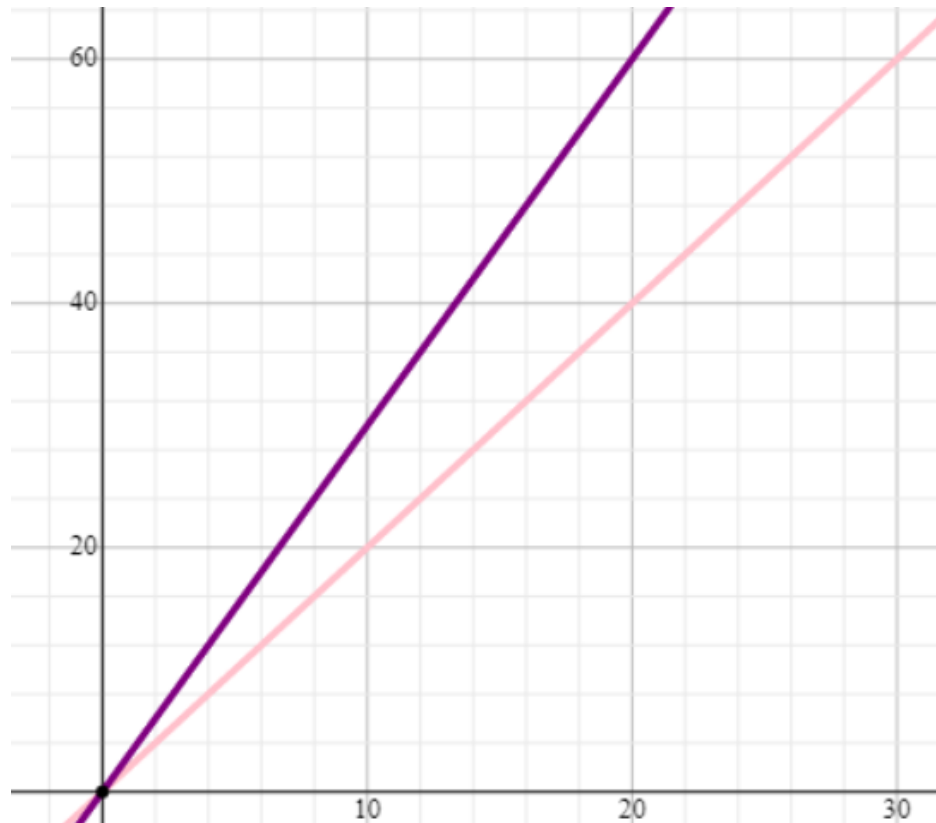


# Análise de Algoritmos

- Um erro comum é utilizar a notação  $O$  para descrever os casos dos algoritmos. Exemplo:
  - Melhor caso é  $O(n)$
  - Pior caso é  $O(n^2)$
- Existem outras notações que podem ser usadas nesse caso. Por exemplo, a notação  $\theta$  indica um limite restrito e seria correto dizer
  - Melhor caso é  $\theta(n)$
  - Pior caso é  $\theta(n^2)$

# Análise de Algoritmos

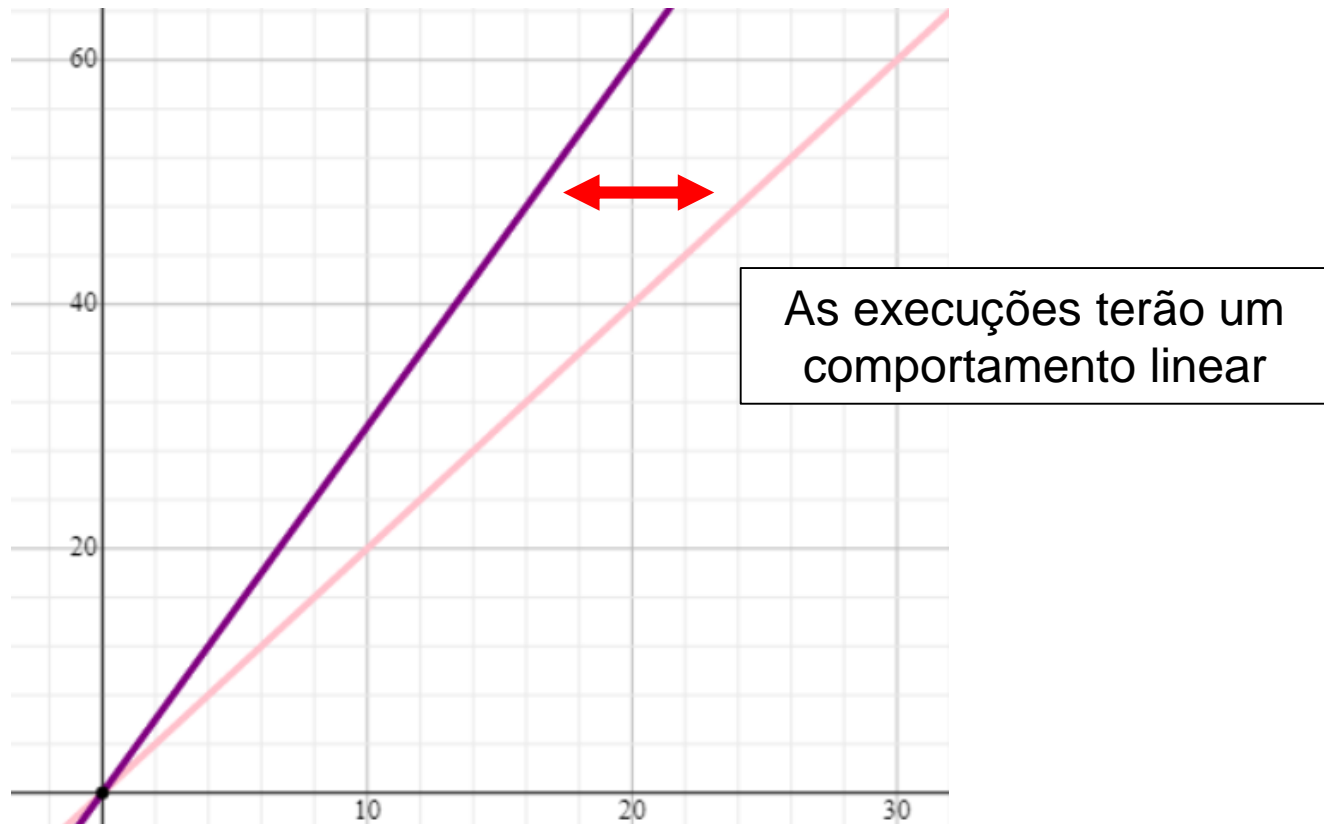
- No caso do  $\theta(n)$ 
  - Restrito por funções de comportamento linear





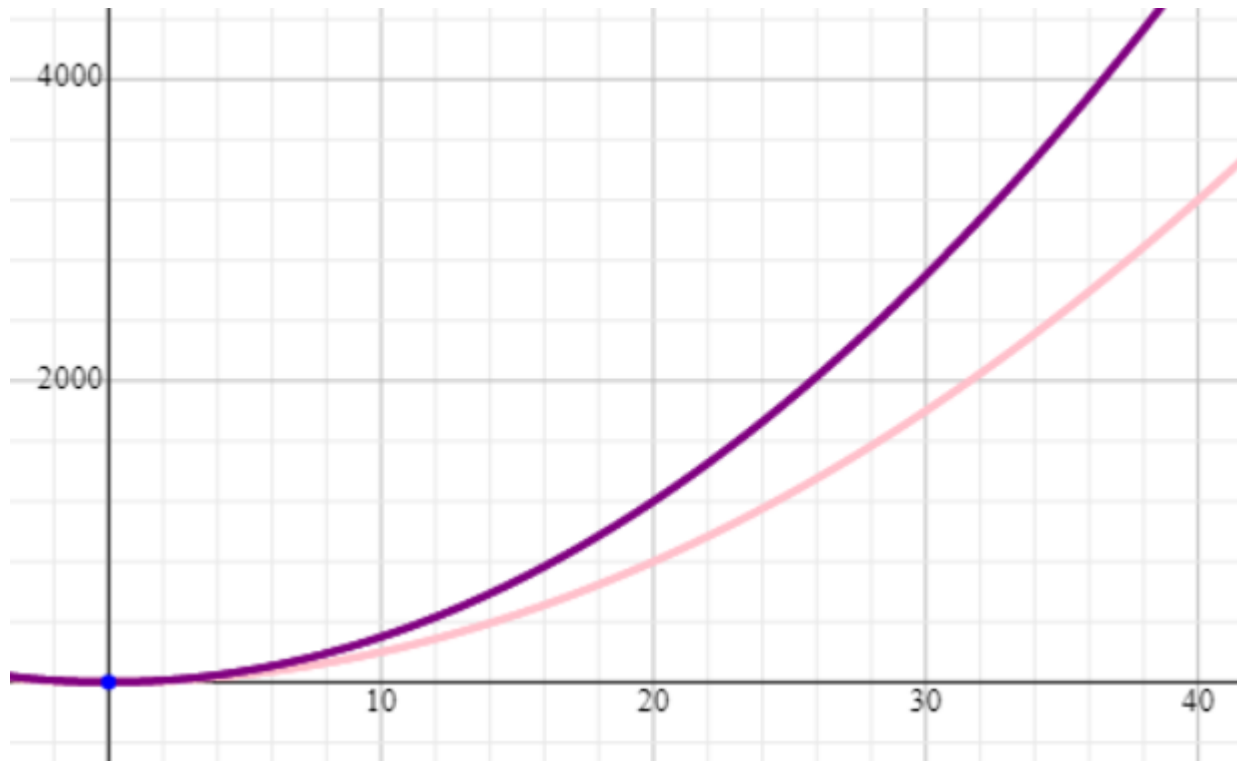
# Análise de Algoritmos

- No caso do  $\theta(n)$ 
  - Restrito por funções de comportamento linear



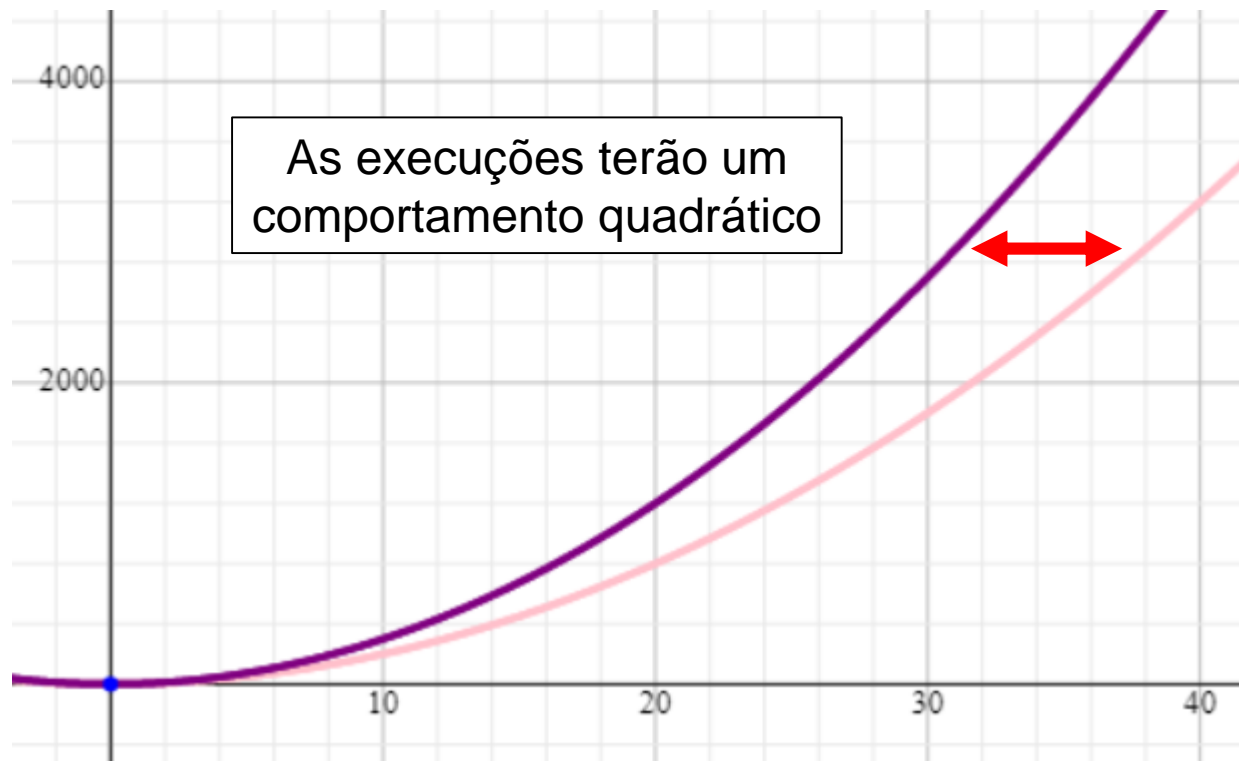
# Análise de Algoritmos

- No caso do  $\theta(n^2)$ 
  - Restrito por funções de comportamento quadrático



# Análise de Algoritmos

- No caso do  $\theta(n^2)$ 
  - Restrito por funções de comportamento quadrático



# Análise de Algoritmos

- Notações mais utilizadas
  - $O$ : define um limite superior
  - $\Omega$ : define um limite inferior
  - $\theta$ : define um limite restrito
  
- Veremos os detalhes dessa notações nas próximas aulas

---

# Classes de Comportamento Assintótico

Apresentação adaptada (ver referências)

---

# Classes de Comportamento Assintótico

- $f(n) = O(1)$ 
  - O uso do algoritmo independe do tamanho de  $n$
  - As instruções do algoritmo são executadas um número fixo de vezes
- O que significa um algoritmo ser  $O(2)$  ou  $O(5)$ ?

# Classes de Comportamento Assintótico

- $f(n) = O(\log n)$ 
  - Ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores.
  - Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande.
- Supondo que a base do logaritmo seja 2:
  - Para  $n = 1\ 000$ ,  $\log(n)$  é aproximadamente 10.
  - Para  $n = 1\ 000\ 000$ ,  $\log(n)$  é aproximadamente 20.
- Exemplo:
  - Algoritmo de pesquisa binária.

# Classes de Comportamento Assintótico

- $f(n) = O(n)$ 
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada.
  - Esta é a melhor situação possível para um algoritmo que tem que processar/produzir  $n$  elementos de entrada/saída
  - Cada vez que  $n$  dobra de tamanho, o tempo de execução também dobra
- Exemplos:
  - Algoritmo de pesquisa seqüencial



# Classes de Comportamento Assintótico

- $f(n) = O(n \log n)$ 
  - Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois agrupando as soluções.
  - Caso típico dos algoritmos baseados no paradigma divisão-e-conquista.
- Supondo que a base do logaritmo seja 2:
  - Para  $n = 1.000$ ,  $n \log n$  é aproximadamente 10.000
  - Para  $n = 1.000.000$ ,  $n \log n$  é aproximadamente 20.000.000
- Exemplo:
  - Algoritmo de ordenação *Merge Sort*

# Classes de Comportamento Assintótico

- $f(n) = O(n^2)$ 
  - Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são **processados aos pares**
  - Para  $n = 1000$ , o número de operações é da ordem de 1000000
  - Sempre que  $n$  dobra o tempo de execução é multiplicado por 4
  - Algoritmos deste tipo são úteis para resolver problemas de tamanhos relativamente pequenos.
- Exemplos:
  - Algoritmos de ordenação simples como seleção e inserção

# Classes de Comportamento Assintótico

- $f(n) = O(n^3)$ 
  - Algoritmos desta ordem de complexidade geralmente são úteis apenas para resolver problemas relativamente pequenos
  - Para  $n = 100$ , o número de operações é da ordem de 1.000.000
  - Sempre que  $n$  dobra o tempo de execução é multiplicado por 8.
- Exemplo:
  - Algoritmo para multiplicação de matrizes

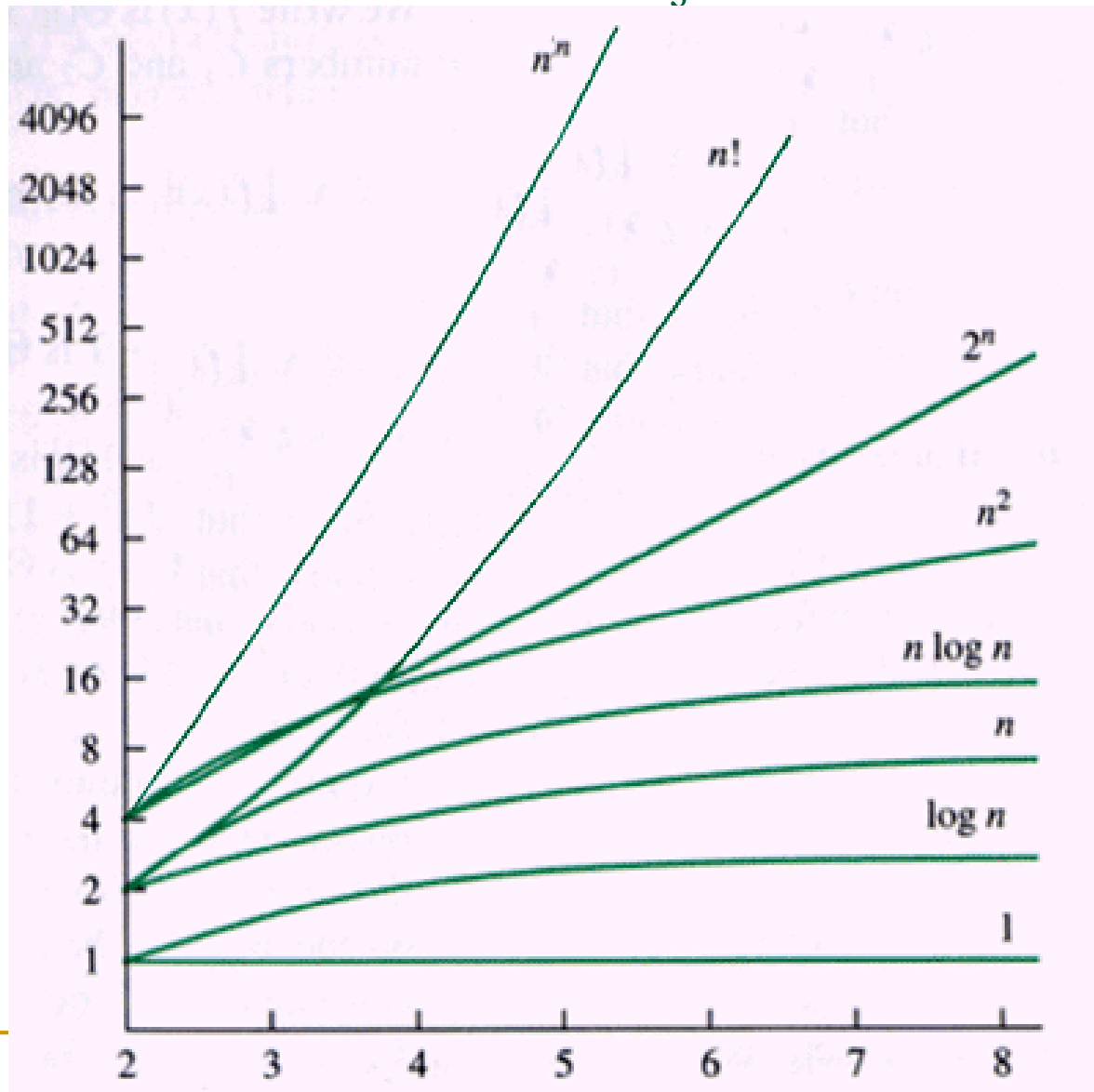
# Classes de Comportamento Assintótico

- $f(n) = O(2^n)$ 
  - Algoritmos desta ordem de complexidade não são úteis sob o ponto de vista prático
  - Eles ocorrem na solução de problemas quando se usa a força bruta para resolvê-los
  - Para  $n = 20$ , o tempo de execução é cerca de 1 000 000
  - Sempre que  $n$  dobra o tempo de execução fica elevado ao quadrado
- Exemplo
  - Algoritmo do Caixeiro Viajante

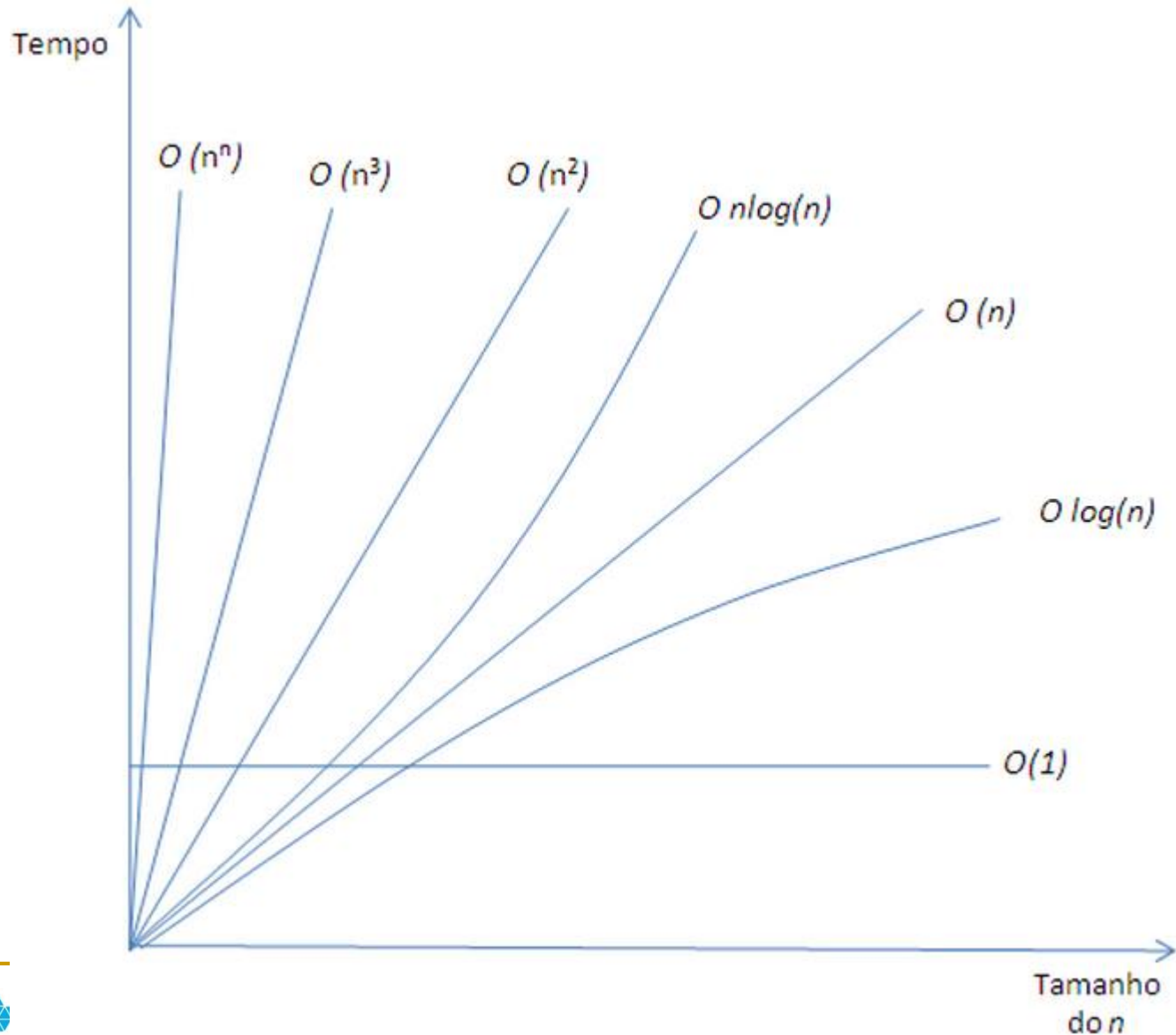
# Classes de Comportamento Assintótico

- $f(n) = O(n!)$ 
  - Um algoritmo de complexidade  $O(n!)$  é dito ter complexidade exponencial
    - Apesar de ter comportamento pior do que  $O(2^n)$
  - Geralmente ocorrem quando se usa força bruta na solução do problema
- Considerando:
  - $n = 20$ , temos que  $20! = 2432902008176640000$ , um número com 19 dígitos
  - $n = 40$  temos um número com 48 dígitos

# Crescimento das Funções




# Crescimento de Funções



# Notação Assintótica

Baseados nos gráficos, vemos que podemos classificar as funções em ordem crescente de tempo de execução

Esta ordem pode ser dada por:

Constante		$O(c)$
Logarítmica		$\log N$
Linear		$N$
$N \log N$		$N \log N$
Quadrática		$N^2$
Cúbica		$N^3$
Exponencial		$2^N$
Fatorial		$N!$



# Tempo de execução e complexidade

Função de custo	Tamanho $n$			
	10	20	30	40
$n$	0,00001 s	0,00002 s	0,00003 s	0,00004 s
$n^2$	0,0001 s	0,0004 s	0,0009 s	0,0016 s
$n^3$	0,001 s	0,008 s	0,027 s	0,064 s
$n^5$	0,1 s	3,2 s	24,3 s	102,4 s
$2^n$	0,001 s	0,002 s	0,008 s	0,032 s
$3^n$	0,059 s	0,243 s	1,024 s	4,398 s

# Tempo de execução e complexidade

Função de custo	Tamanho $n$	
	10	20
$n$	0,00001 s	0,00002 s
$n^2$	0,0001 s	0,0004 s
$n^3$	0,001 s	0,008 s
$n^5$	0,1 s	3,2 s
$2^n$	0,001 s	1 s
$3^n$	0,059 s	58 min

# Tempo de execução e complexidade

Função de custo	Tamanho $n$		
	10	20	30
$n$	0,00001 s	0,00002 s	0,00003 s
$n^2$	0,0001 s	0,0004 s	0,0009 s
$n^3$	0,001 s	0,008 s	0,027 s
$n^5$	0,1 s	3,2 s	24,3 s
$2^n$	0,001 s	1 s	17,9 min
$3^n$	0,059 s	58 min	6,5 anos

# Tempo de execução e complexidade

Função de custo	Tamanho $n$			
	10	20	30	40
$n$	0,00001 s	0,00002 s	0,00003 s	0,00004 s
$n^2$	0,0001 s	0,0004 s	0,0009 s	0,0016 s
$n^3$	0,001 s	0,008 s	0,027 s	0,64 s
$n^5$	0,1 s	3,2 s	24,3 s	1,7 min
$2^n$	0,001 s	1 s	17,9 min	12,7 dias
$3^n$	0,059 s	58 min	6,5 anos	3855 séc.

# Tempo de execução e complexidade

Função de custo	Tamanho $n$				
	10	20	30	40	50
$n$	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s
$n^2$	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,035 s
$n^3$	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s
$n^5$	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min
$2^n$	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos
$3^n$	0,059 s	58 min	6,5 anos	3855 séc.	$10^8$ séc.

# Tempo de execução e complexidade

Função de custo	Tamanho $n$					
	10	20	30	40	50	60
$n$	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
$n^2$	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
$n^3$	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
$n^5$	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
$2^n$	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
$3^n$	0,059 s	58 min	6,5 anos	3855 séc.	$10^8$ séc.	$10^{13}$ séc.

# Referências Bibliográficas

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; (2002). Algoritmos – Teoria e Prática. Tradução da 2ª edição americana. Rio de Janeiro. Editora Campus
- TAMASSIA, ROBERTO; GOODRICH, MICHAEL T. (2004). Projeto de Algoritmos - Fundamentos, Análise e Exemplos da Internet
- ZIVIANI, N. (2007). Projeto e Algoritmos com implementações em Java e C++. São Paulo. Editora Thomson
- [http://www.ime.usp.br/~pf/analise\\_de\\_algoritmos/aulas/Oh.html](http://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/Oh.html)

# Referências de Material

- Adaptado do material de
  - Professor Alessandro L. Koerich da *Pontifícia Universidade Católica do Paraná (PUCPR)*
  - Professor Humberto Brandão da *Universidade Federal de Alfenas (Unifal-MG)*
  - Professor Ricardo Linden da *Faculdade Salesiana Maria Auxiliadora (FSMA)*
  - Professor Antonio Alfredo Ferreira Loureiro da *Universidade Federal de Minas Gerais (UFMG)*