Itaú Unibanco

Itaú

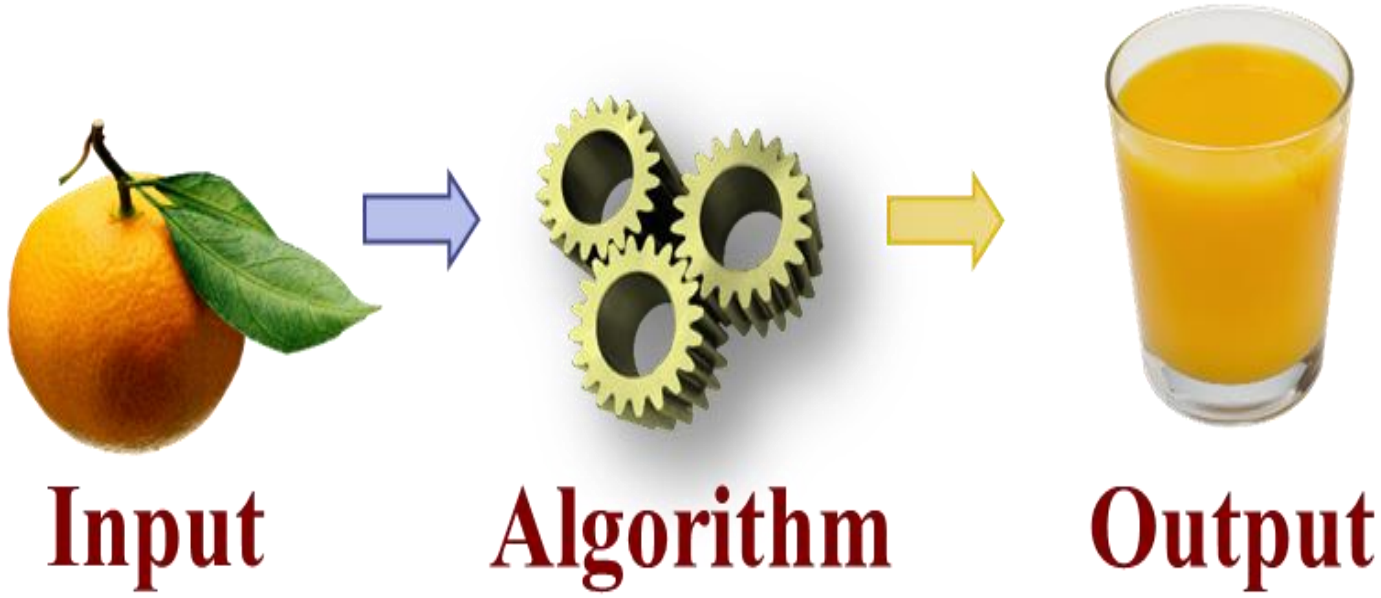
Programa de formação

ITAú analytics.

Módulo I – Fundamentos Computacionais
Sessão 3 - Aula 3 – Análise de Algoritmos (I)

Prof. Dr. Luiz Alberto Vieira Dias
Prof. Dr. Lineu Mialaret

Análise de Algoritmos
Tempo de Execução

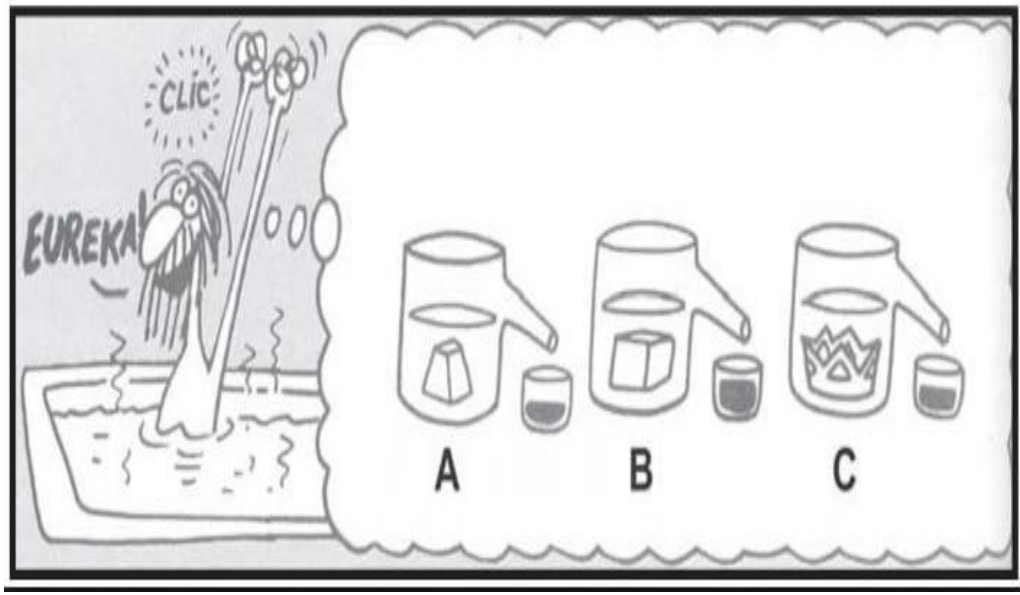


Análise de Algoritmos
Introdução

- Na história clássica, um rei pediu ao famoso matemático Arquimedes que determinasse se uma coroa dourada era realmente de ouro puro e não continha prata, conforme se desconfiava
- Arquimedes descobriu um modo de resolver esse problema quando estava em uma banheira
- Ele notou que a água transbordava da banheira à medida que ele entrava e, percebendo as implicações deste fato, ele imediatamente saiu do banho e correu sem roupa pela ruas da cidade gritando “Eureka, Eureka”

Análise de Algoritmos
Introdução (cont.)

- Ele descobriu uma ferramenta de análise (o deslocamento de água), que quando combinada com uma balança, poderia determinar se a coroa era de ouro puro ou não
- Esta descoberta foi infeliz para o ourives que fez a coroa...



Análise de Algoritmos
Explicação Física

- Densidade = massa/volume
- Densidade da água = 1 (g/cm3)
- Densidade do ouro = 19,3 (g/cm3)
- Densidade da prata = 10,5 (g/cm3)

Análise de Algoritmos
Explicação Física (cont.)

- Coroa da 100% ouro (Au)= massa/volume = 19,3
- Coroa 100% prata (Ag) = massa/volume =10,5
- Duas coroas coma mesma massa = 19,3*vol_Au = 10,5*vol_Ag
- Exemplo 1 000 g Au → vol_Au = 1000/19,3 =

Análise de Algoritmos
Introdução (cont.)

- Neste curso de Algoritmos e Estrutura de Dados, está-se interessado no projeto de “bons” algoritmos e estruturas de dados

- Um Algoritmo é um procedimento, passo a passo, para realizar alguma tarefa num tempo finito, com um número finito de passos

- Uma Estrutura de Dados é uma maneira sistemática de organizar e acessar dados

Análise de Algoritmos
Introdução (cont.)

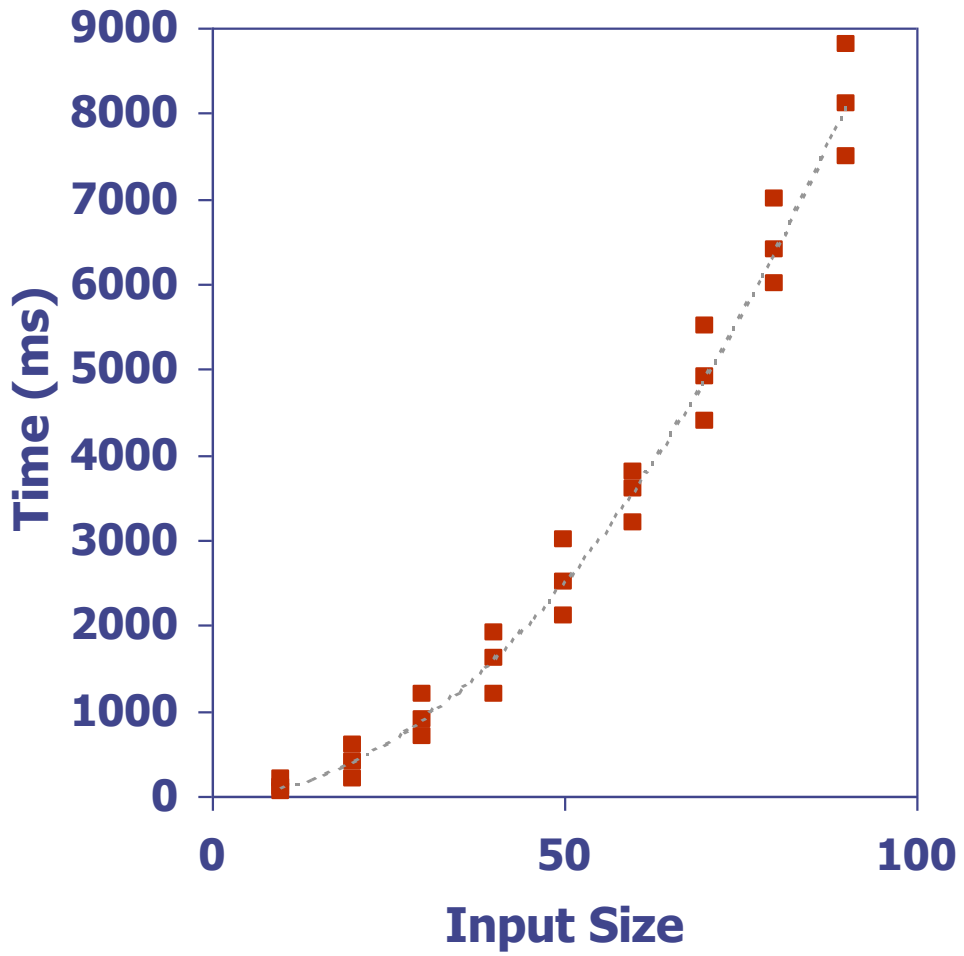
- Para se classificar algoritmos e estruturas de dados como “bons”, deve-se ter maneiras precisas de analisá-los
- O ferramental básico de análise envolve a caracterização do tempo de execução de algoritmos e operações sobre estruturas de dados
  - Tempo de execução (*running time*) é uma medida natural, pois tempo é um recurso precioso
- O consumo de memória também é de interesse

Análise de Algoritmos
Estudos Experimentais

- Pode-se estudar o tempo de execução de um algoritmo:
  - Implementando o algoritmo

```
from time import time
start_time = time( )
run algorithm
end_time = time( )
elapsed = end_time − start_time
```

  - Executando-o com entradas de tamanho e composição variada e observando o tempo de execução
  - Plotando os resultados



Análise de Algoritmos
Estudos Experimentais (cont.)

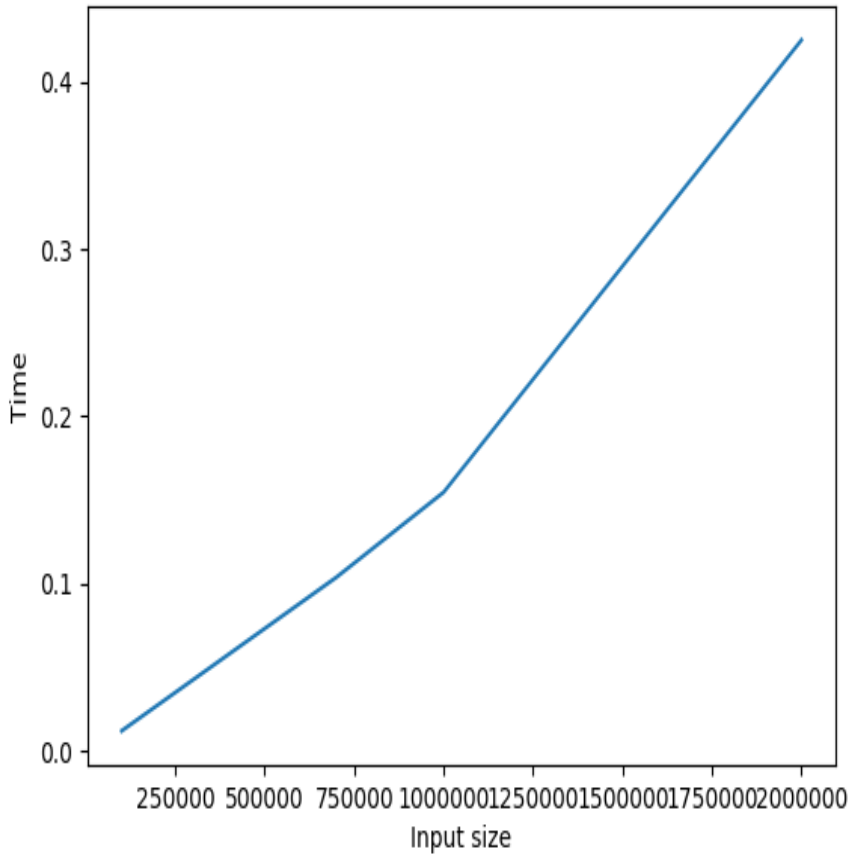
```
import os
from time import time
start_time = time( )
xvec = range(100000)
yvec = range(100000)
zvec = [0.5*(x+y) for x,y in zip(xvec,yvec)]
end_time = time( )
elapsed_time = end_time - start_time
print (elapsed_time)
```

100000 -> 0.011967182159423 8

700000 -> 0.103722572326660 1

1000000 -> 0.154588460922241 2

2000000 -> 0.424863576889038 1



Análise de Algoritmos
Desafios de Estudos Experimentais

- Estudos experimentais para a aferição do tempo de execução (*running time*) de algoritmos são afetados por:
  - Tamanho da entrada
  - Variações para entradas de mesmo tamanho
  - Ambiente de Hardware
    - Processador
    - Clock
    - Memória
    - Disco
    - ...

Análise de Algoritmos
Desafios de Estudos Experimentais (cont.)

- Estudos experimentais para a aferição do tempo de execução (*running time*) de algoritmos são afetados por:
  - Ambiente de Software
    - Sistema Operacional
    - Linguagem de Programação
    - Compilador
    - Interpretador
    - ...

Análise de Algoritmos
Desafios de Estudos Experimentais (cont.)

- Estudos experimentais sobre o tempo de execução (*running time*) de algoritmos são úteis mas têm três limitações principais:
  - Experimentos podem ser feitos apenas sobre um conjunto limitado de entradas de testes e podem não ser indicativos do tempo de execução para outras entradas não testadas
  - É difícil comparar a eficiência de 2 algoritmos a menos que os experimentos tenham sido feitos nas mesmas plataformas (hardware + software)
  - É necessário implementar e executar o algoritmo para poder estudar seu tempo de execução experimentalmente

Análise de Algoritmos
Além dos Estudos Experimentais

- Uma metodologia geral para análise de algoritmos deve:
  - Permitir avaliar a eficiência relativa de quaisquer 2 algoritmos, de forma independente do ambiente de hardware + software empregado
  - Considerar todas as possíveis entradas
  - Poder ser utilizada com base num estudo da descrição de alto nível do algoritmo, sem implementá-lo efetivamente ou sem aplicar abordagens experimentais sobre ele

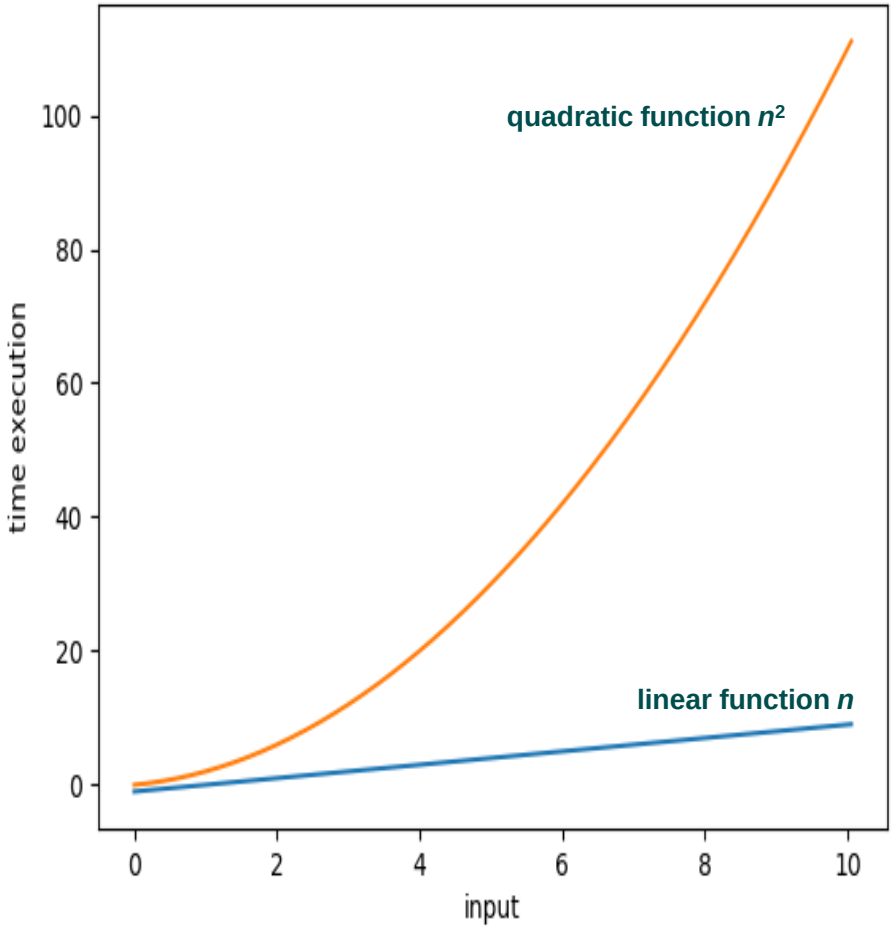
Análise de Algoritmos
Além dos Estudos Experimentais (cont.)

- A metodologia deve associar com cada algoritmo uma função ***f(n)*** que caracteriza o tempo de execução (*running time*) do algoritmo como uma função do tamanho ***n*** da entrada

- Dada a afirmação "O Algoritmo A roda em tempo proporcional a ***n***", isto significa que se forem efetuados experimentos nesse algoritmo, será observado que o tempo de execução real do algoritmo A com qualquer entrada de tamanho ***n*** nunca excede ***cn***, onde ***c*** é uma constante que depende do hardware e do software usado

Análise de Algoritmos
Além dos Estudos Experimentais (cont.)

- Dados dois algoritmos A e B, onde o algoritmo A é executado em tempo proporcional a ***n*** e o algoritmo B roda em tempo proporcional a n^2, o algoritmo A será preferido em relação ao algoritmo B, uma vez que a função ***n*** cresce mais devagar que a função n^2



Análise de Algoritmos
Contando Operações Primitivas

- Para se analisar o tempo de execução de um algoritmo sem realizar experimentos, pode-se efetuar uma análise diretamente sobre uma descrição em alto nível do algoritmo (ou no código fonte ou no pseudocódigo), usando-se o conceito de operações primitivas
- **Operações Primitivas** são as computações de alto nível que são independentes da linguagem de programação e podem ser identificadas no pseudocódigo ou em alguma descrição similar

Análise de Algoritmos
Contando Operações Primitivas (cont.)

- Pode-se definir um conjunto de operações primitivas de alto nível que são independentes da linguagem de programação utilizada, como por exemplo:
  - Atribuir um identificador para um objeto
  - Determinar o objeto associado com um identificador
  - Efetuar uma operação aritmética (somar 2 inteiros,...)
  - Comparar dois números
  - Acessar um único elemento de uma lista Python pelo índice
  - Chamar uma função ou método
  - Retornar de uma função ou método

Análise de Algoritmos
Contando Operações Primitivas (cont.)

- Desta forma, inspecionando-se o pseudocódigo, pode-se contar o número de operações primitivas executadas por um algoritmo

```
set total to zero

get list of numbers

loop through each number in the list
   add each number to total
end loop

if number more than zero
   print "it's positive" message
else
   print "it's zero or less" message
end if
```

Análise de Algoritmos
Contando Operações Primitivas (cont.)

- De forma mais específica, uma operação primitiva corresponde a uma instrução de baixo nível com um tempo de execução constante, mas que depende do ambiente de software e hardware
- Em vez de tentar determinar o tempo de execução específico de cada operação primitiva, simplesmente **contam-se** quantas operações primitivas serão executadas e usa-se este número ***t*** de operações primitivas obtidas como uma **estimativa** de alto nível do tempo de execução do algoritmo

Análise de Algoritmos
Contando Operações Primitivas (cont.)

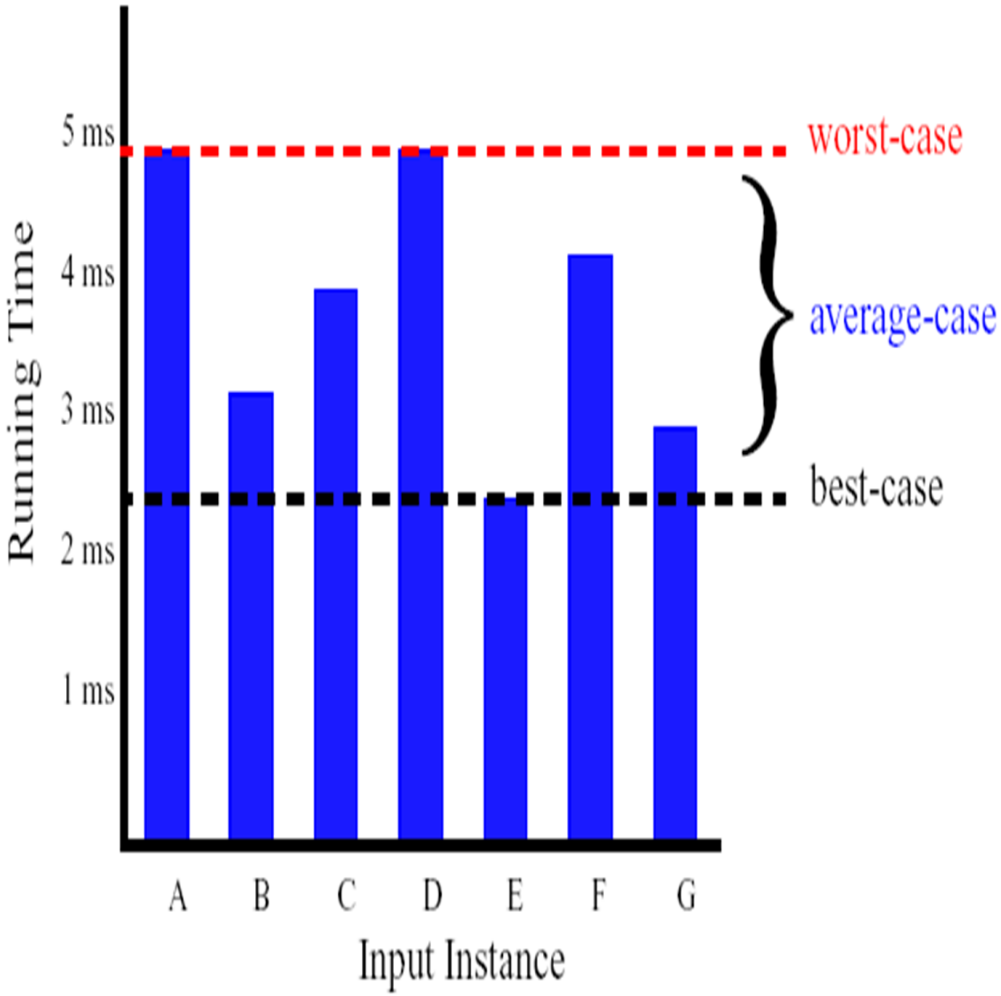
- Essa contagem de operações está relacionada com o tempo de execução em um hardware e software específicos, pois cada operação corresponde a uma instrução realizada em tempo constante e existe um número fixo de operações primitivas

Análise de Algoritmos
Contando Operações Primitivas (cont.)

- Nesta abordagem, assume-se implicitamente que os tempos de execução de operações primitivas diferentes serão similares
- Assim, o número ***t*** de operações primitivas que um algoritmo realiza será proporcional ao tempo de execução daquele algoritmo
- Finalmente, para se capturar a magnitude de crescimento do tempo de execução de um algoritmo, associa-se com cada algoritmo, uma função **f(*n*)** que caracteriza o número de operações primitivas que são executadas como uma função do tamanho da entrada ***n*** do algoritmo

Análise de Algoritmos
Pior Caso X Melhor Caso

- Algoritmos podem executar mais rápido com algumas entradas do que com outras do mesmo tamanho
- Obter o tempo médio de execução pode ser complicado (tratamento estatístico sofisticado)
- Avalia-se o pior caso
  - Executando bem no pior caso, rodará bem nos outros casos
  - Algumas aplicações (tráfego aéreo) precisam saber do pior caso

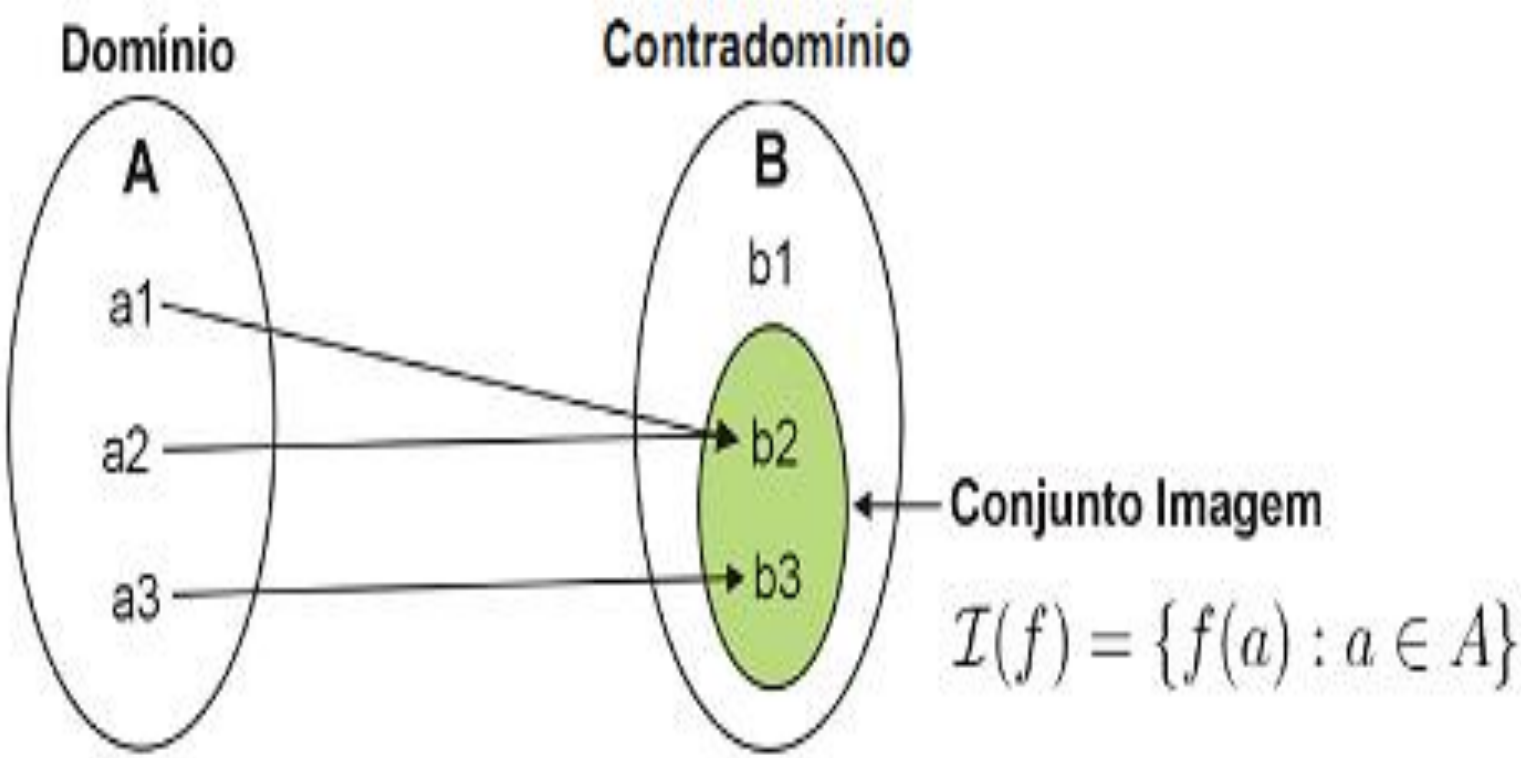


Análise de Algoritmos
Funções

- Terminologia para funções:
  - Sejam S e T conjuntos. Uma *função* (aplicação) f de S em T, denotada por $f{:}S \rightarrow T$, é um subconjunto de $S \times T$ onde cada elemento de S aparece exatamente uma única vez como primeiro componente de um par ordenado
  - S é o *domínio* e T é o *contradomínio* da função
  - Se (s, t) pertence à função, então t é denotado por $f(s)$
  - t é a *imagem* de s sob f
  - s é a *imagem inversa* de t sob f e f leva s em t
  - Para $A \subseteq S$, $f(A)$ denota $\{f(a) \mid a \in A\}$
- Uma função $f{:}S \rightarrow T$ é uma relação de S para T (um subconjunto de $S \times T$, tal que cada $s \in S$ pertence a um único par ordenado (s,t) na função f

Análise de Algoritmos
Funções (cont.)

- Funções são muito utilizadas em Computação e serão estudadas algumas das funções mais importantes

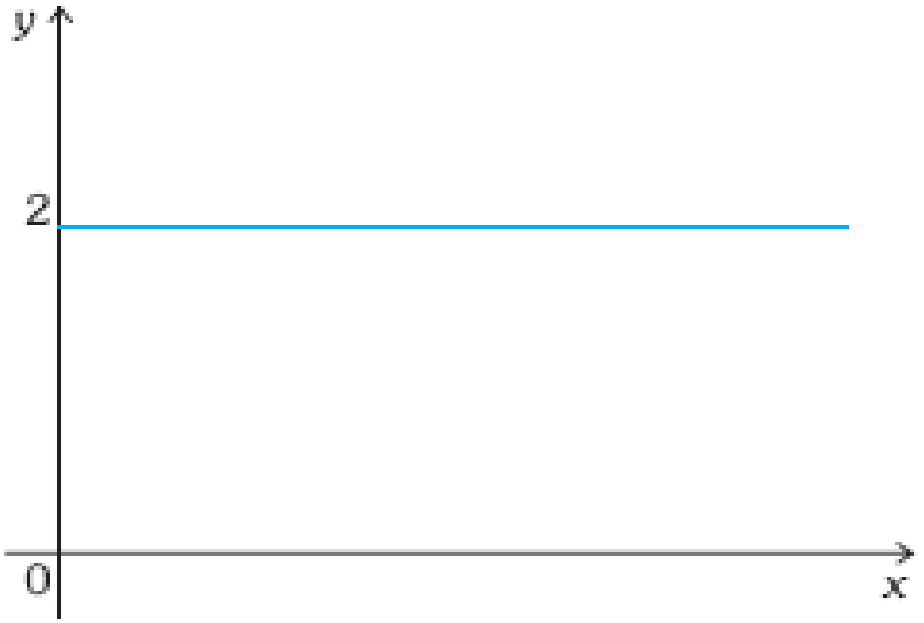


Exemplo da Terminologia de Função

Análise de Algoritmos
Funções - Função Constante

- Função Constante:
  - A função que associa cada elemento do seu domínio a um mesmo elemento do contradomínio é chamada de função constante

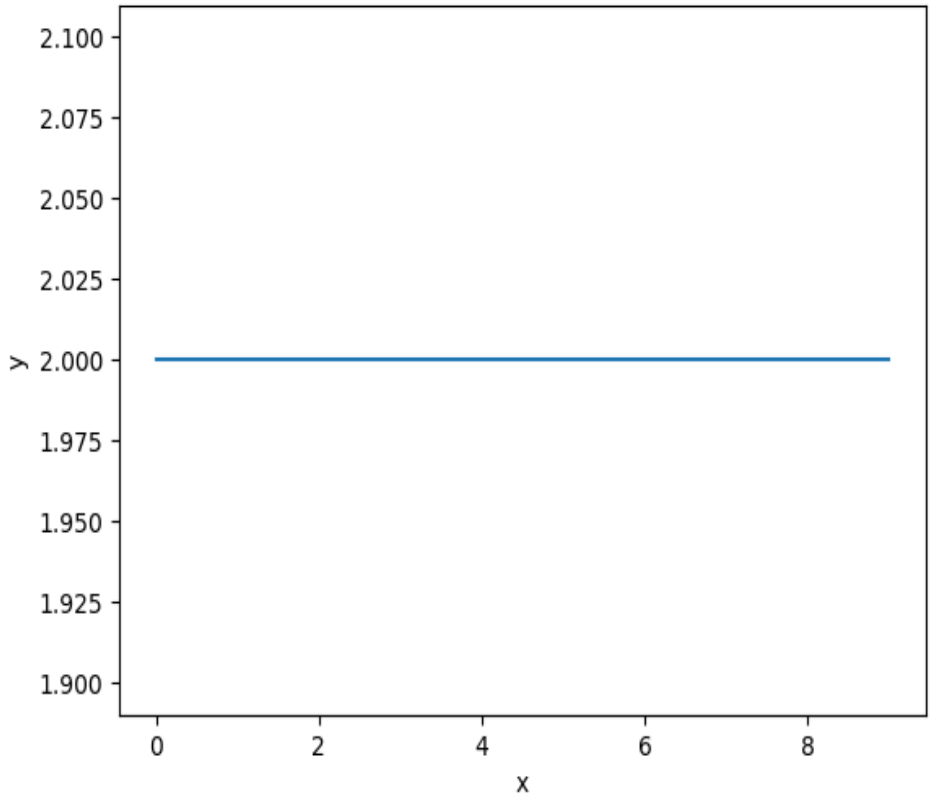
A função $f:[0,\infty) \rightarrow \mathbb{R}$, $f(x) = 2$, é uma função constante. Seu gráfico é mostrado na figura ao lado.



As simple as it is, the constant function is useful in algorithm analysis because it characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to a variable, or comparing two numbers.

Análise de Algoritmos
Funções – Função Constante (cont.)

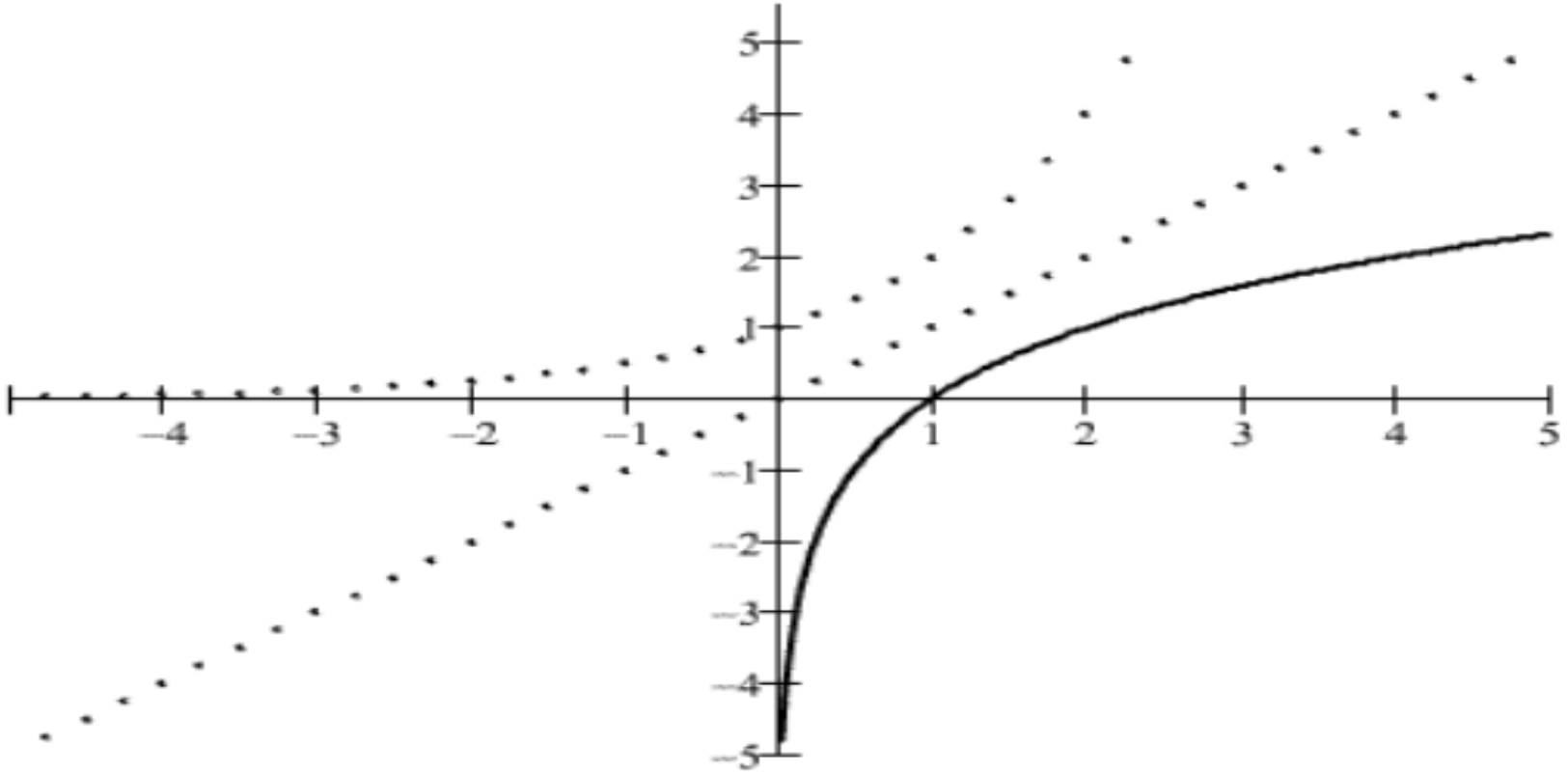
```
import matplotlib.pyplot as plt

x = [0,1,2,3,5,6,7,8,9]
y = [2,2,2,2,2,2,2,2,2]

plt.ylabel('y')
plt.xlabel('x')
plt.plot(x, y)
plt.show()
```



Análise de Algoritmos
Funções – Função Logaritmo

- Função Logaritmo:
  - Seja *a* um número positivo. O logaritmo de qualquer número positivo ***x*** na base ***a***, denotado por $\log_a x$ representa o expoente ao qual ***a*** precisa ser elevado para se obter ***x***
  - Ou seja,
    - $y = \log_a x$ e $a^y = x$
- Exemplo:
  - $\log_2 8 = 3$, já que $2^3 = 8$.
  - $\log_2 64 = 6$, já que $2^6 = 64$.

Análise de Algoritmos
Funções – Função Logaritmo (cont.)



$(a > 1)$

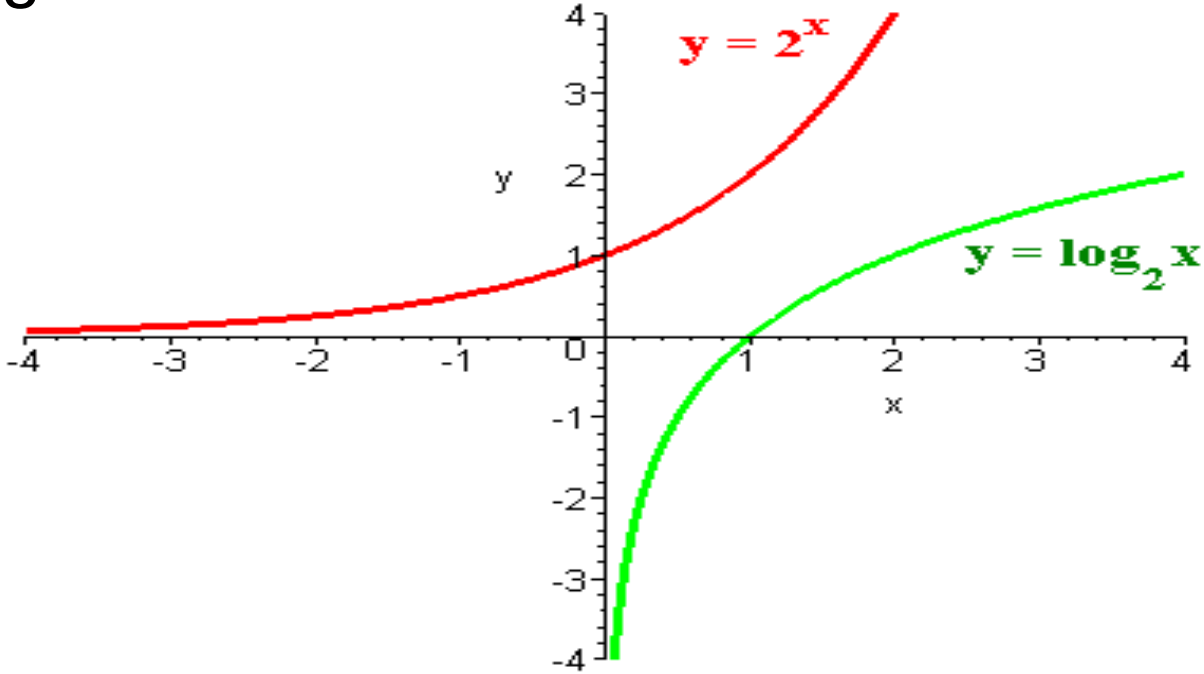
Gráfico da Função Logaritmo

Análise de Algoritmos
Funções – Função Logaritmo (cont.)

- Três classes de logaritmos são importantes:
  - Logaritmos na base 10, chamados de logaritmos decimais e representados por
    - $\log_{10} x$ ou $\log x$
  - Logaritmos na base *e*, chamados de logaritmos naturais ou neperianos, e representados por
    - $\log_e x$ ou $\ln x$
  - Logaritmos na base 2, os chamados logaritmos binários e representados por
    - $\log_2 x$ ou $\log x$ ou $\log_2 x$ ou $\lg x$

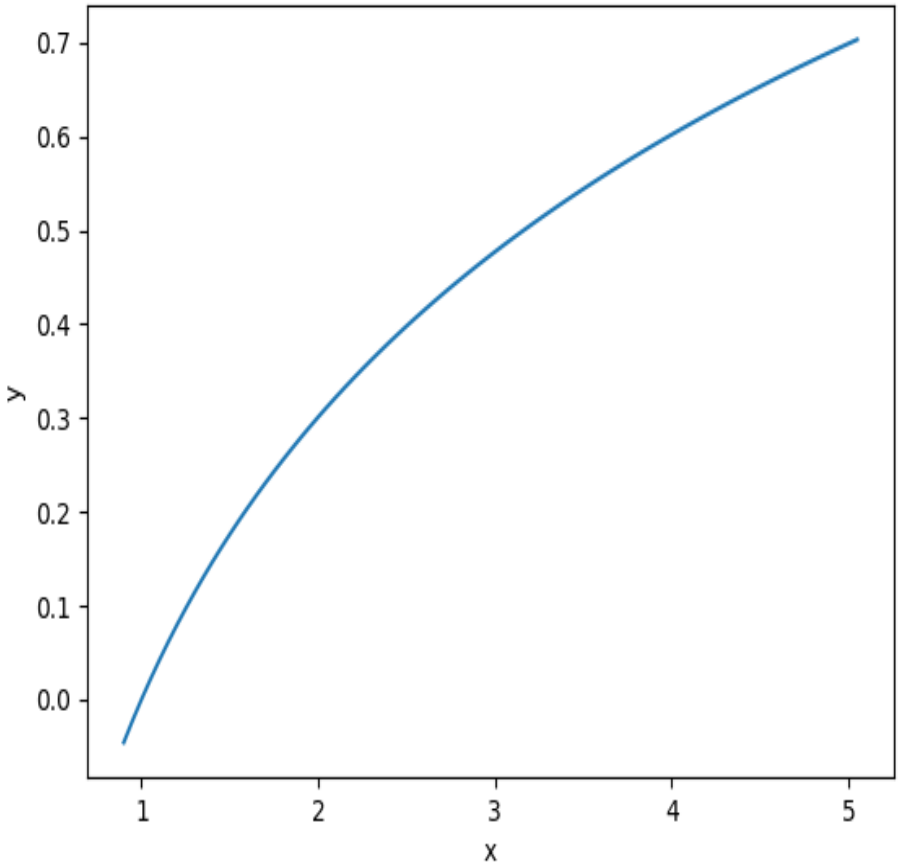
Análise de Algoritmos
Funções – Função Logaritmo (cont.)

- A relação básica entre as funções, $f(x) = a^x$ e $g(x) = \log_a x$, é que elas representam a inversa uma da outra; logo, os gráficos dessas funções estão relacionados geometricamente, conforme mostrado a seguir



Análise de Algoritmos
Funções – Função Logaritmo (cont.)

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0.9, 5.1, 0.05)
y = np.log2(x)

plt.plot(x, y)
plt.ylabel('y')
plt.xlabel('x')
plt.show()
```



Análise de Algoritmos
Logaritmos e Exponenciais

- No estudo de logaritmos e expoentes tem-se:
  $\log_b a = c$ se $a = b^c$

- Pode-se omitir a base b dos logaritmos quando $b = 2$, de forma que, por exemplo, log 1024 = 10

- Usa-se a seguinte notação abreviada:
  $\log^c n$ para denotar a função $(\log n)^c$
  $\log\log n$ para denotar $\log(\log n)$

Análise de Algoritmos
Logaritmos e Exponenciais (cont.)

- Com *a*, *b* e *c* sendo números reais positivos, tem-se as seguintes proposições:
  1. $\log_b ac = \log_b a + \log_b c$
  2. $\log_b a/c = \log_b a - \log_b c$
  3. $\log_b a^c = c\log_b a$
  4. $\log_b a = (\log_c a)/\log_c b$
  5. $b^{\log_c a} = a^{\log_c b}$
  6. $(b^a)^c = b^{ac}$
  7. $b^a b^c = b^{a+c}$
  8. $b^a/b^c = b^{a-c}$

Análise de Algoritmos
Logaritmos e Exponenciais (cont.)

- Exercício: Demonstrar as seguintes proposições:
  - $\log(2n\log n) = 1 + \log n + \log\log n$
  - $\log(n/2) = \log n - 1$
  - $\log \sqrt{n} = \log (n)^{1/2}$
  - $\log\log \sqrt{n} = \log\log n - 1$
  - $\log_4 n = (\log n)/2$
  - $\log 2^n = n$
  - $2^{\log n} = n$
  - $2^{2\log n} = n^2$
  - $4^n = (2^2)^n = 2^{2n}$
  - $n^2 2^{3\log n} = n^5$
  - $4^n/2^n = 2^n$

Propriedades

1. $\log_b ac = \log_b a + \log_b c$
2. $\log_b a/c = \log_b a - \log_b c$
3. $\log_b a^c = c\log_b a$
4. $\log_b a = (\log_c a)/\log_c b$
5. $b^{\log_c a} = a^{\log_c b}$
6. $(b^a)^c = b^{ac}$
7. $b^a b^c = b^{a+c}$
8. $b^a/b^c = b^{a-c}$

Análise de Algoritmos
Logaritmos e Exponenciais (cont.)

- Já que, tipicamente, um logaritmo não é um inteiro, apesar do tempo de execução de um algoritmo ser expresso como um inteiro, como por exemplo o número de operações efetuadas
- Assim na análise de algoritmos pode-se, as vezes, envolver o uso dos conceitos de funções conhecidas como função ***piso (floor)*** e função ***teto (ceiling)*** respectivamente:

  $\lfloor x \rfloor$ = o maior inteiro menor do que ou igual a x (limite inferior)

  $\lceil x \rceil$ = o menor inteiro maior que ou igual a x (limite superior)

- Essas funções fornecem uma maneira de converter funções com valores reais em funções com valores inteiros

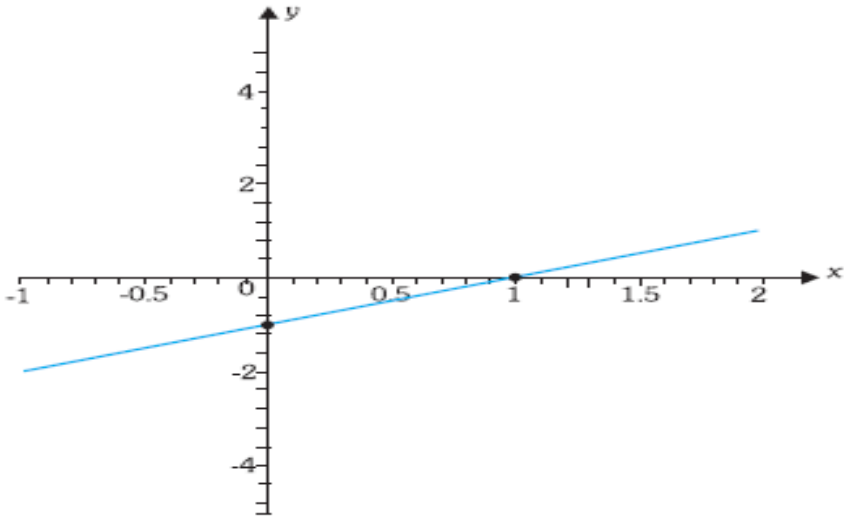
Análise de Algoritmos
Funções – Função Linear

- Função Linear:
  - Chama-se função afim ou linear qualquer função dada por $f(x) = ax + b$ onde os coeficientes a e b são números reais dados
  - O gráfico da função linear com domínio e contradomínio $\mathbb{R}$ é uma reta com coeficiente angular igual ao valor de a e que intercepta os eixos coordenados X e Y nos pontos $(-b/a,0)$ e $(0,b)$
- Exemplo:

O gráfico da função linear tomando-se $a = 1$ e $b = -1$, ou seja,

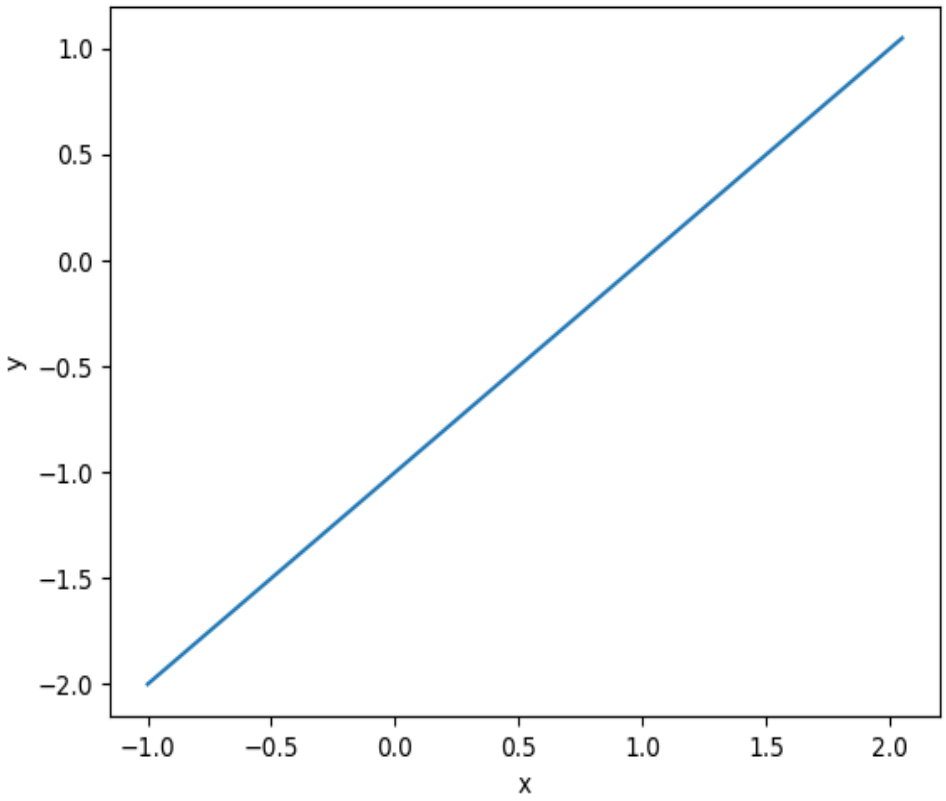
$$y = f(x) = 1x - 1 = x - 1,$$

no intervalo [–1, 2], é mostrado ao lado



Análise de Algoritmos
Funções – Função Linear (cont.)

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(-1.0, 2.1, 0.05)
y = 1*x - 1

plt.plot(x, y)
plt.ylabel('y')
plt.xlabel('x')
plt.show()
```

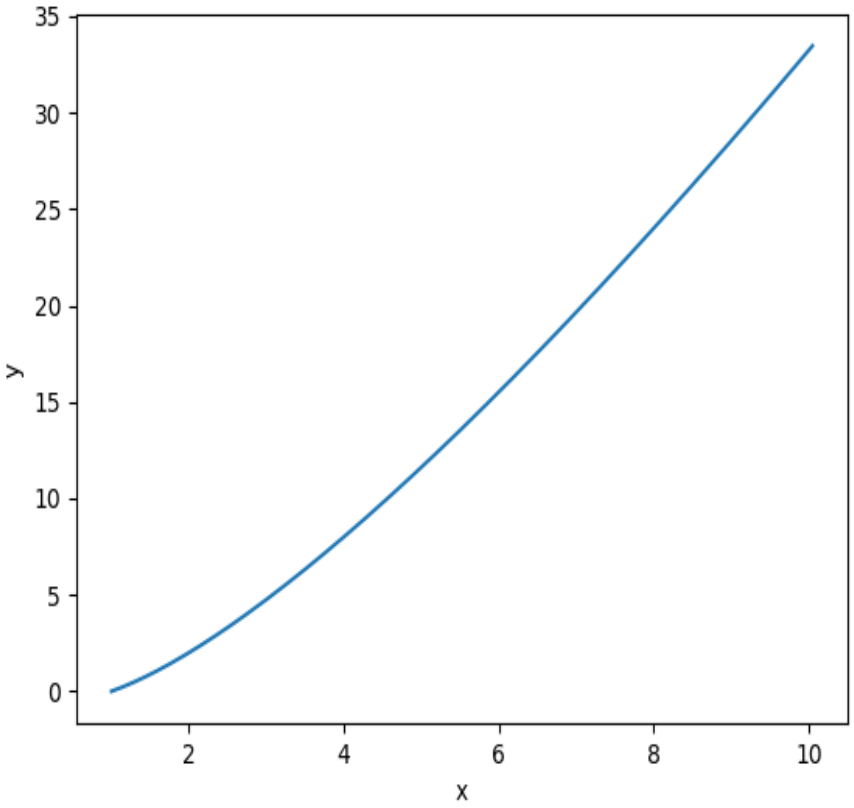


Análise de Algoritmos
Funções – Função N-LogN

- Função N-Log-N:
- É uma função caracterizada por f(n) = $n$log$n$
- Esta função tem um crescimento um pouco maior do que a função linear e bem menor que a função quadrática
  - É preferível ter um algoritmo executando em tempo $n$log$n$ do que em tempo quadrático
- Há vários algoritmos importantes que são executados em tempo $n$log$n$
  - Os algoritmos mais rápidos de classificação executam em tempo $n$log$n$

Análise de Algoritmos
Funções – Função N-LogN (cont.)

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(1.0, 10.1, 0.05)
y = np.log2(x)*x

plt.plot(x, y)
plt.ylabel('y')
plt.xlabel('x')
plt.show()
```



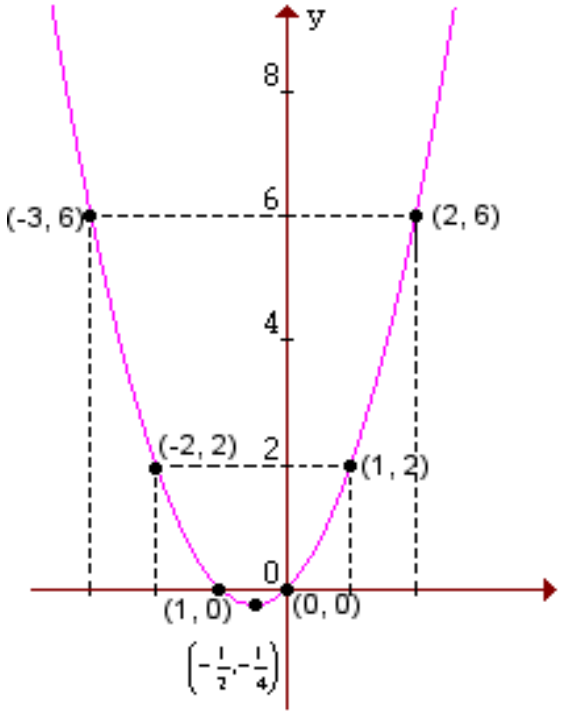
Análise de Algoritmos
Funções – Função Quadrática

- Função Quadrática:
  - Sejam a, b e c números reais quaisquer com $a \neq 0$. A função f definida em $\mathbb{R}$ e dada por $y = f(x) = ax^2 + bx + c$ recebe nome de função quadrática

- Exemplo:

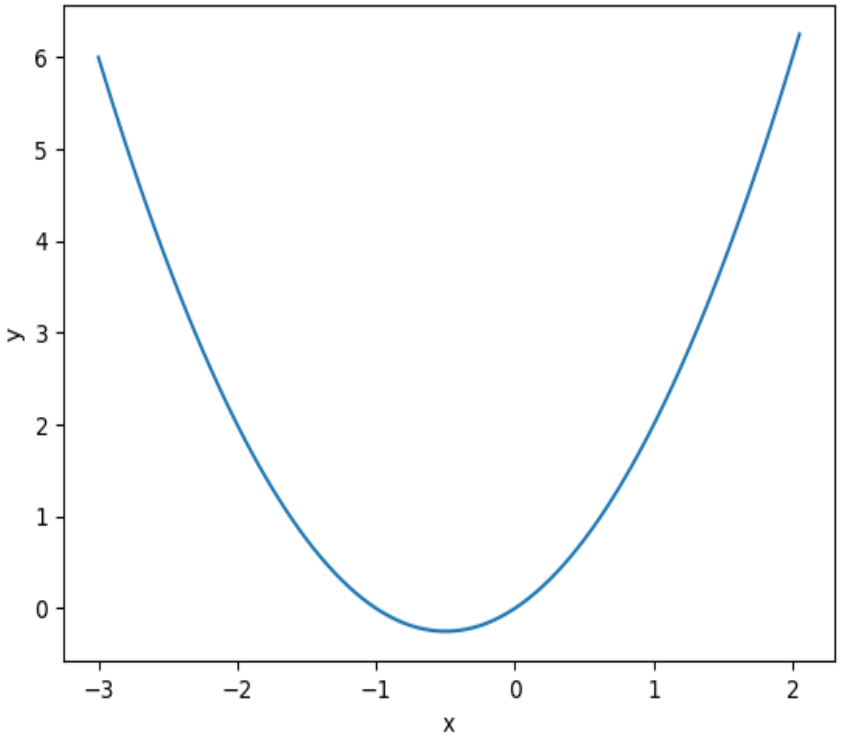
  O gráfico da função quadrática $y = x^2 + x$ é apresentado ao lado

x	y
-3	6
-2	2
-1	0
-1/2	-1/4
0	0
1	2
2	6



Análise de Algoritmos
Funções – Função Quadrática (cont.)

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(-3.0, 2.1, 0.05)
y = x* x + x

plt.plot(x, y)
plt.ylabel('y')
plt.xlabel('x')
plt.show()
```



Análise de Algoritmos
Funções – Função Polinomial

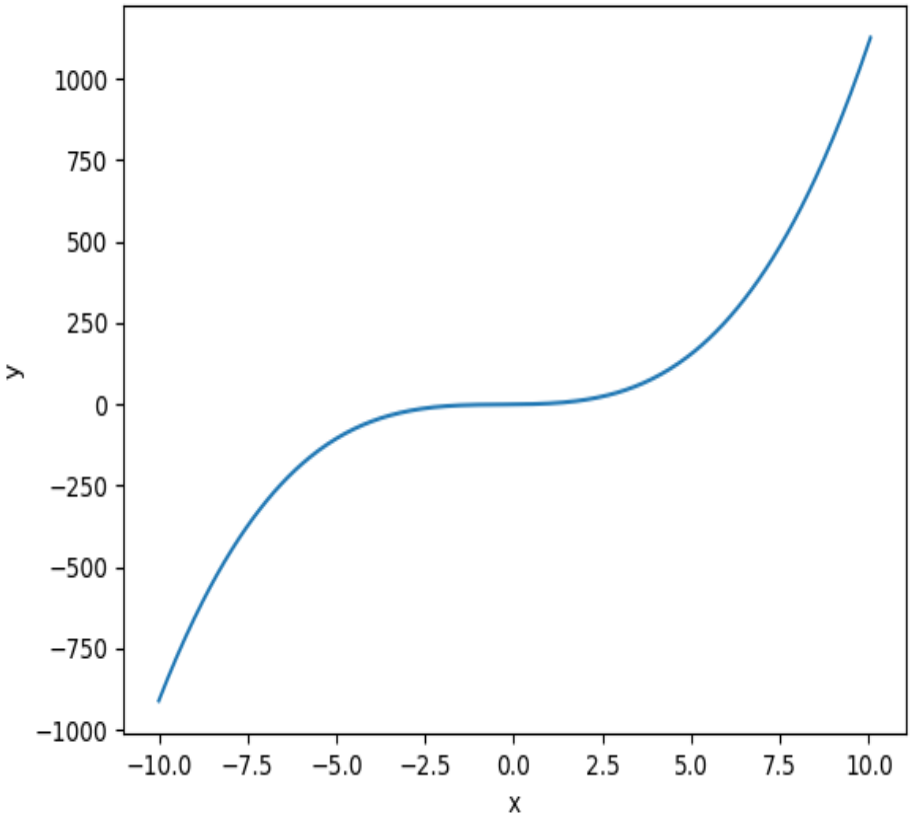
- Função Polinomial:
  - É toda função cuja regra de associação é um polinômio, ou seja, $f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$, onde os coeficientes a_0, a_1, ..., a_n são números reais e n é um número natural chamado de grau de $f(x)$
- Exemplo:
  - A função linear é um exemplo de função polinomial de grau $n = 1$
  - A função quadrática $f(x)=ax^2 + bx + c$, a ≠ 0, é uma função polinomial de grau $n = 2$
  - A função f(x) = $2x^4 - x^3 + 3x^2 - 5x + 1$ é uma função polinomial de grau $n = 4$

Polinômio é um conjunto de monômios separados por operações.

Monômio ou termo algébrico é toda expressão algébrica determinada por apenas um número real, uma variável ou pelo produto de números e variáveis.

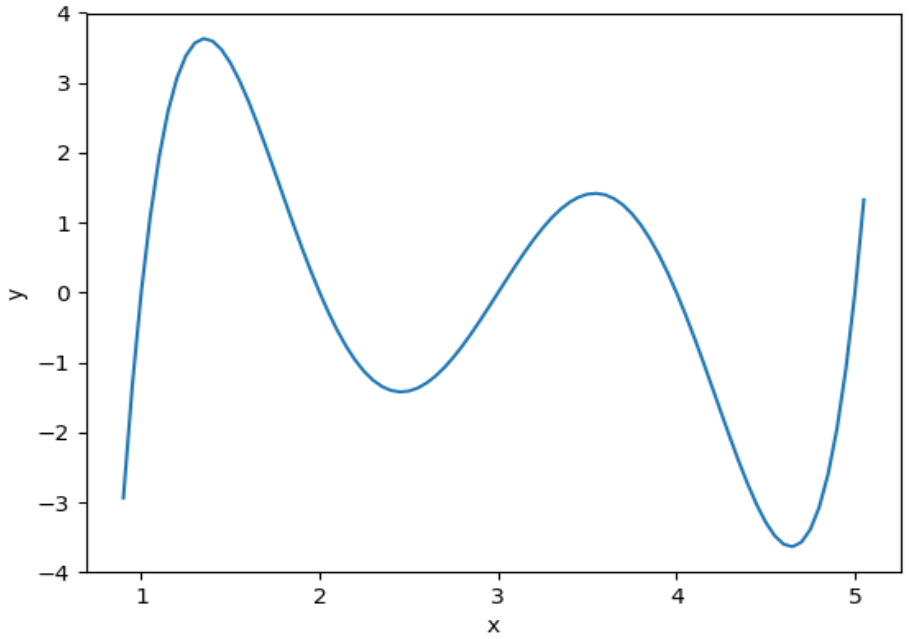
Análise de Algoritmos
Funções – Função Polinomial (cont.)

```
# cubic function
import numpy as np
import matplotlib.pyplot as plt


x = np.arange(-10.0, 10.1, 0.05)
y = x*x*x + x* x + x


plt.plot(x, y)
plt.ylabel('y')
plt.xlabel('x')
plt.show()
```



Análise de Algoritmos
Funções – Função Polinomial (cont.)

```
import numpy as np
import matplotlib.pyplot as
plt

x = np.arange(0.9, 5.1, 0.05)
y = x ** 5 - 15 * x ** 4 + 85 * x
** 3 - 225 * x ** 2 + 274 * x -
120

plt.plot(x, y)
plt.ylabel('y')
plt.xlabel('x')
plt.show()
```



Análise de Algoritmos
Somatórios

- Outra notação que surge com frequência na análise de algoritmos é o somatório, que é uma expressão matemática da seguinte forma:

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \ldots + f(b)$$

onde *f* é uma função, *i* é o índice, *a* é o valor de inicio e *t* é o valor final

- Somatórios surgem na análise de algoritmos porque o tempo de execução de laços pode ser representado naturalmente por meio de somas
- Por exemplo, uma soma que surge naturalmente e frequentemente na análise de algoritmos é a progressão geométrica

Análise de Algoritmos
Somatórios (cont.)

- Para qualquer inteiro $n \geq 0$ e qualquer número real $0 < a \neq 1$, considere o seguinte somatório

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

para qualquer inteiro $n > 0$ e qualquer real $a \neq 1$, e que $a^0 = 1$

- O resultado desse somatório é:

$$\frac{1 - a^{n+1}}{1 - a}$$

- Qualquer cientista da computação deveria saber que:

$$1 + 2 + 4 + 8 + \ldots + 2^{n-1} = 2^n - 1$$

- Que é o maior inteiro que pode ser representado em notação binária usando n bits

Análise de Algoritmos
Somatórios (cont.)

- Outro somatório importante que surge em vários contextos é:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + (n-2) + (n-1) + n$$

- Essa soma surge na análise de laços em casos em que o número de operações efetuadas dentro do laço aumenta em um valor fixo a cada iteração do laço
- Para qualquer inteiro $n \geq 1$, tem-se que o resultado desse somatório é :

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

Análise de Algoritmos
Somatórios (cont.)

- Em 1787, um professor alemão de uma escola primária decidiu manter seus alunos de 9 e 10 anos ocupados com a tarefa de somar todos os números inteiros de 1 a 100
- Logo depois de apresentar o exercício, uma das crianças afirmou ter a resposta
- O professor ficou desconfiado, pois o aluno tinha apenas a resposta em suas anotações, sem nenhum cálculo. Mas a resposta estava correta (Qual é a resposta?)
- Esse aluno era Karl Friedrich Gauss, que seria depois um dos maiores matemáticos do século XIX

Análise de Algoritmos
Somatórios (cont.)

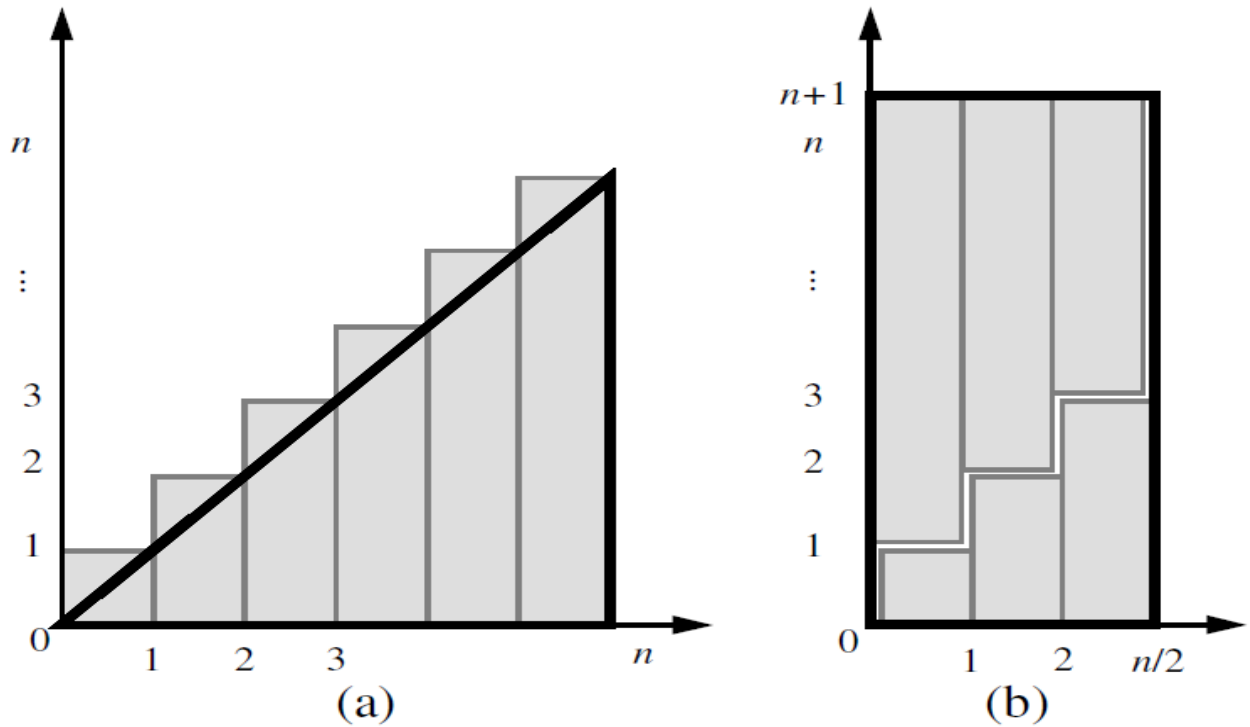


Figure 3.3: Visual justifications of Proposition 3.3. Both illustrations visualize the identity in terms of the total area covered by n unit-width rectangles with heights $1, 2, \ldots, n$. In (a), the rectangles are shown to cover a big triangle of area $n^2/2$ (base n and height n) plus n small triangles of area $1/2$ each (base 1 and height 1). In (b), which applies only when n is even, the rectangles are shown to cover a big rectangle of base $n/2$ and height $n+1$.

Análise de Algoritmos
Somatórios (cont.)

- Exercício: provar por indução que :

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

- Dica de solução:
  - 1) Prove the base case
  - 2) Prove that any case n implies the next case $(n + 1)$ is also true

Análise de Algoritmos
Somatórios (cont.)

- Exercício: mostrar que:

$$\sum_{i=p}^{q}(a_i + b_i) = \sum_{i=p}^{q}(a_i) + \sum_{i=p}^{q}(b_i)$$

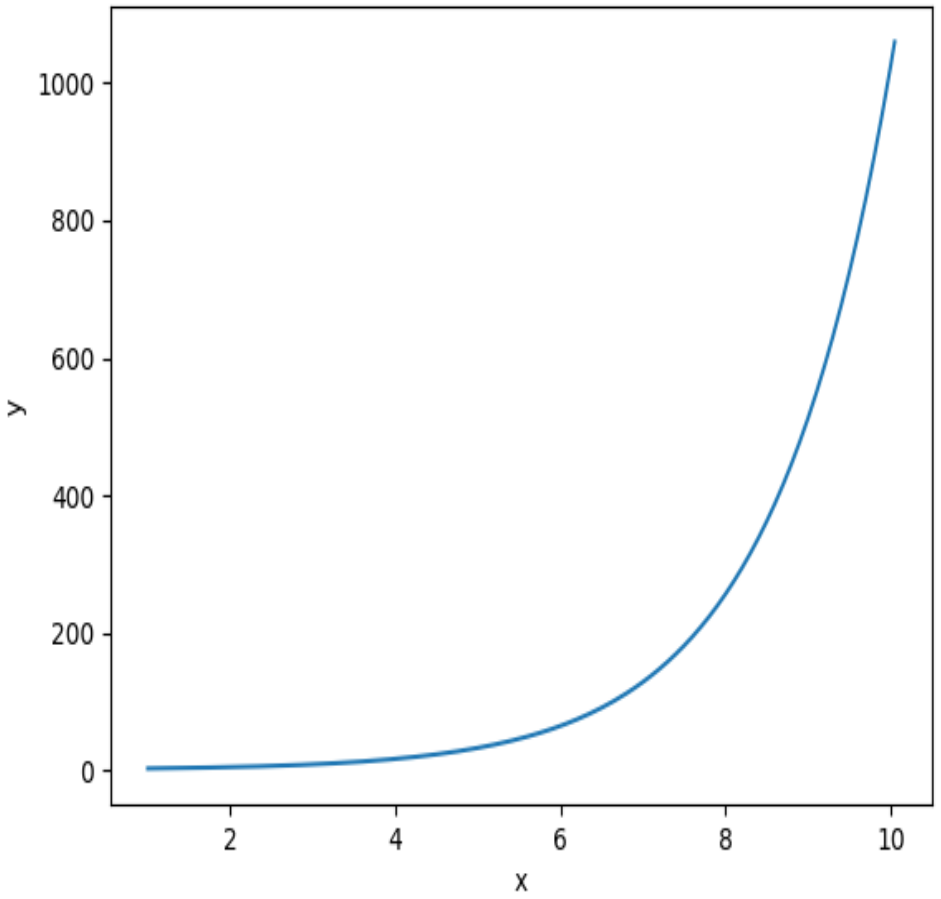
- Dica de solução: decompor o somatório

Análise de Algoritmos
Funções – Função Exponencial

- Função Exponencial:
  - Relembrando
    - $a^m = a.a.a...a$ (m vezes).
    - $a^0 = 1$
    - $a^{-m} = 1/a^m$
    - $a^{m/n} = \sqrt[n]{a^m} = (\sqrt[n]{a})^m$
  - A função exponencial, $f(x) = a^x$, é definida para todos os números reais
  - O gráfico da função exponencial é apresentado a seguir

Análise de Algoritmos
Funções – Função Exponencial (cont.)

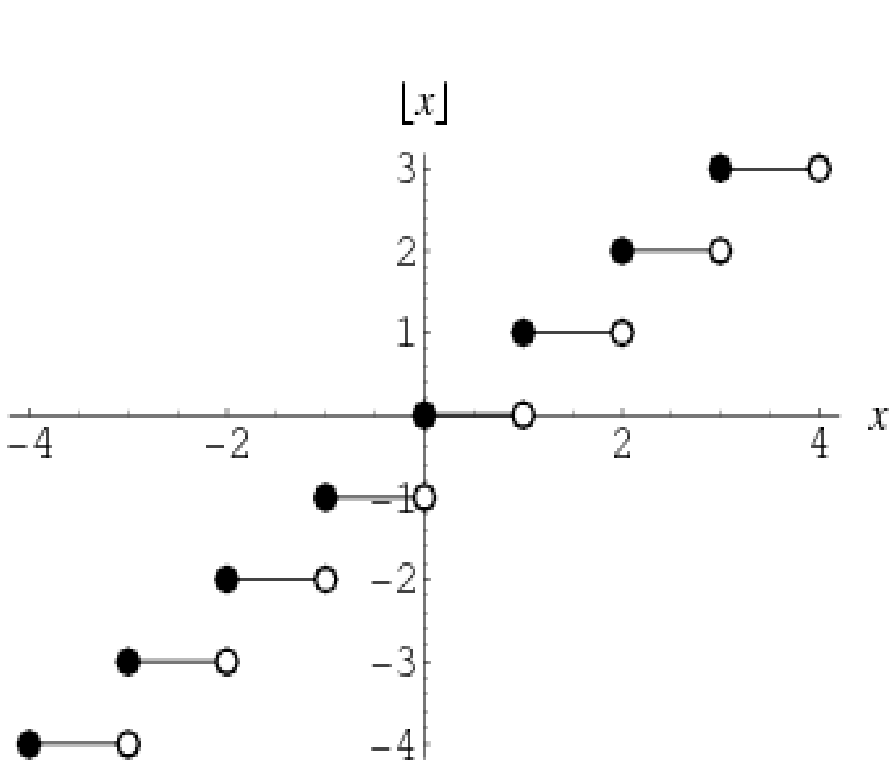
```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(1.0, 10.1, 0.05)
y = np.power(2,x)

plt.plot(x, y)
plt.ylabel('y')
plt.xlabel('x')
plt.show()
```

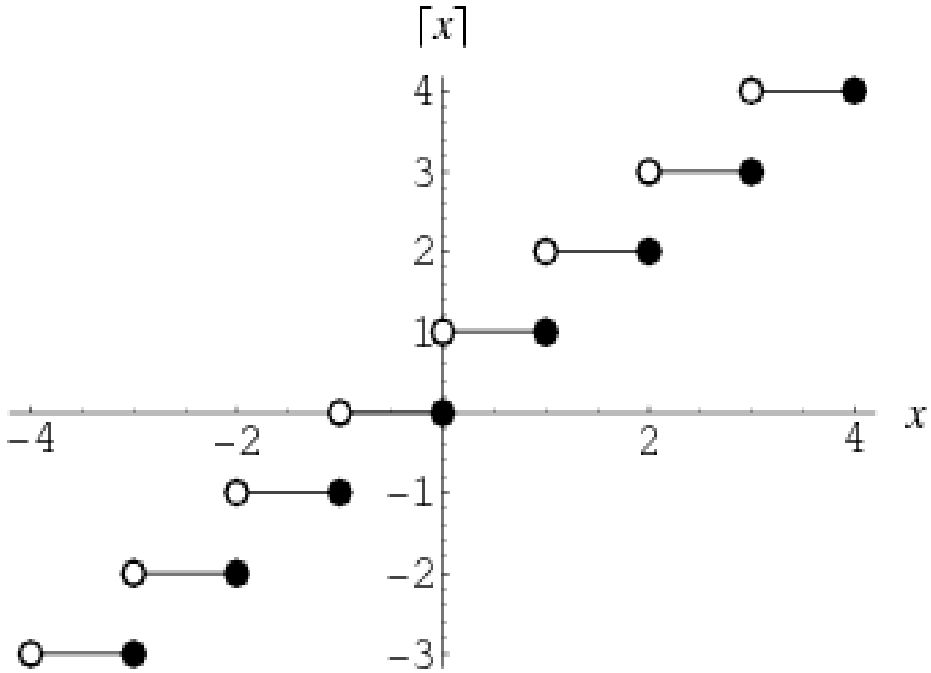


Análise de Algoritmos
Funções – Funções Piso e Teto

- Função Piso (*Floor*) e Função Teto (*Ceiling*):
  - Seja x um número real qualquer. Então x está compreendido entre dois inteiros conhecidos como *piso* e *teto* de x
  - $\lfloor x \rfloor$, denominado de piso (*floor*) de x, é o maior inteiro que é menor ou igual a x
  - $\lceil x \rceil$, denominado de teto (*ceiling*) de x, é o menor inteiro que é maior ou igual a x
- Exemplo:
  - $\lfloor 3{,}14 \rfloor = 3$; $\lfloor -8{,}5 \rfloor = -9$; $\lfloor 7 \rfloor = 7$;
  - $\lfloor -4 \rfloor = ?$
  - $\lceil 3{,}14 \rceil = 4$; $\lceil -8{,}5 \rceil = -8$; $\lceil 7 \rceil = 7$;
  - $\lceil -4 \rceil = ?$

Análise de Algoritmos
Funções – Funções Piso e Teto (cont.)





Gráficos das Funções Piso e Teto

Análise de Algoritmos
Funções – Taxas de Crescimento

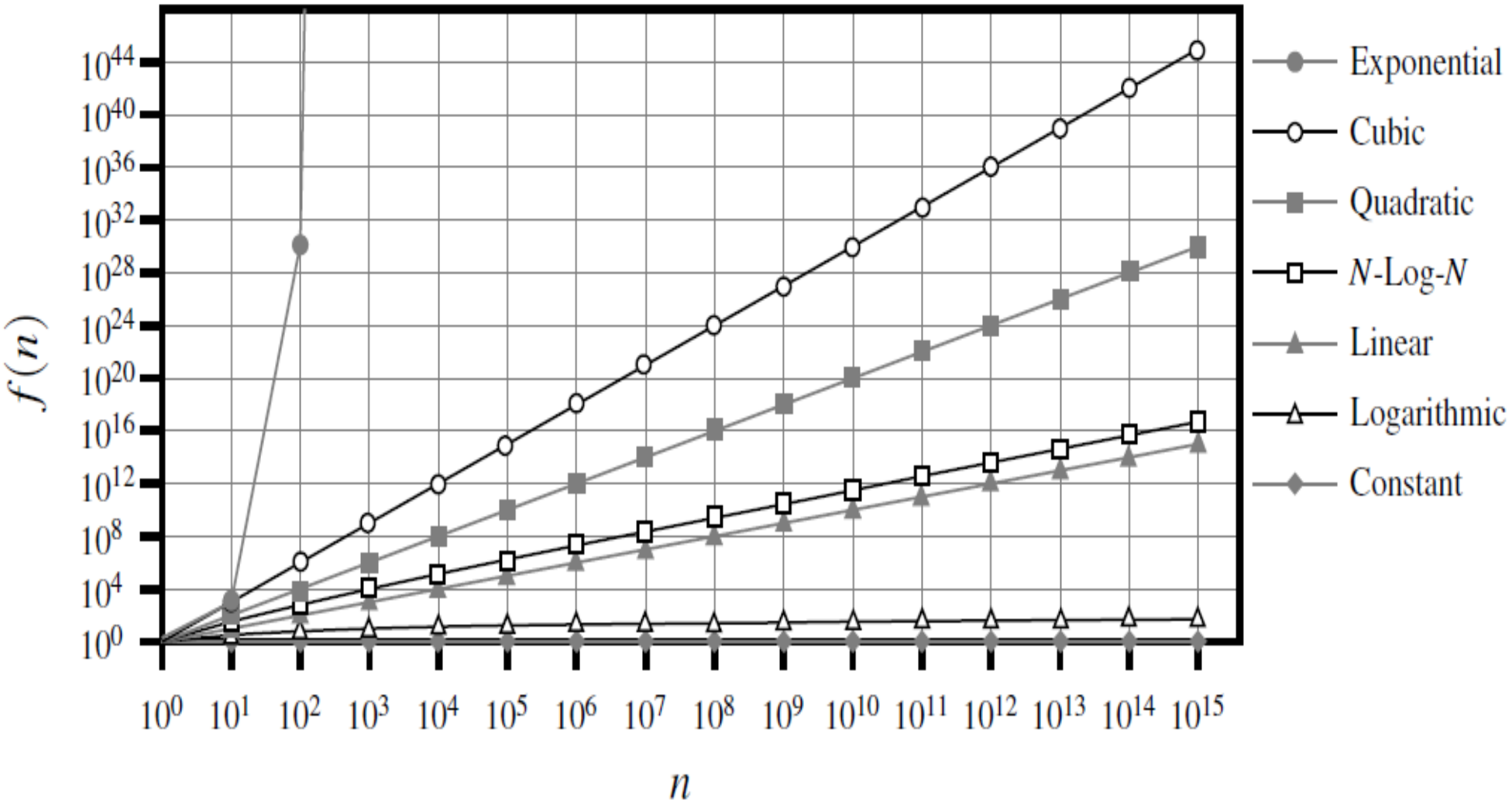


Gráfico Comparativo das Funções

Análise de Algoritmos
Funções – Taxas de Crescimento

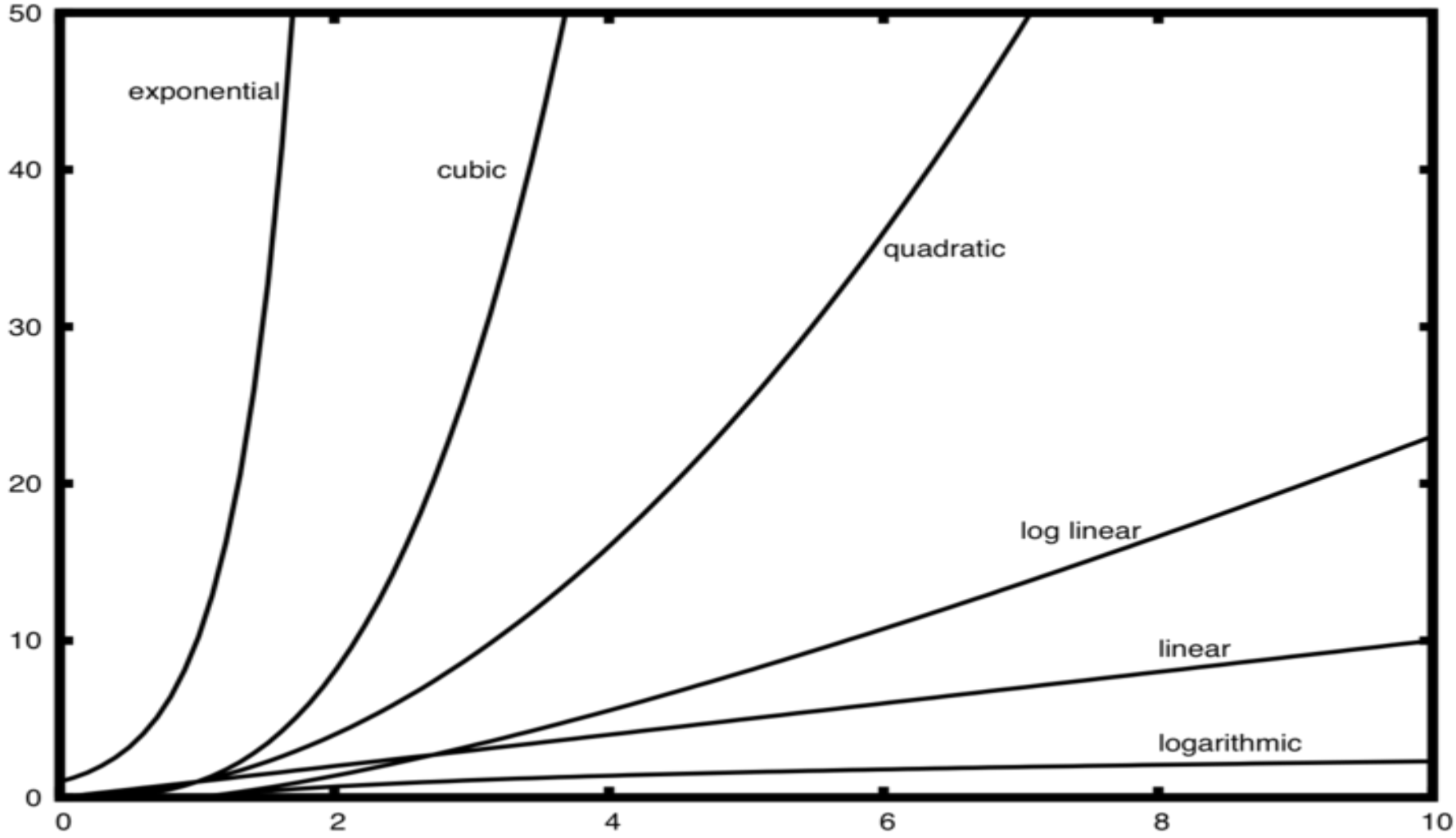


Gráfico Comparativo das Funções