Análise de Algoritmos (parte 2)

Prof. Bruno Travençolo
Baseado nos Slides: Prof. Moacir Ponti Jr. ICMC-USP

Sumário

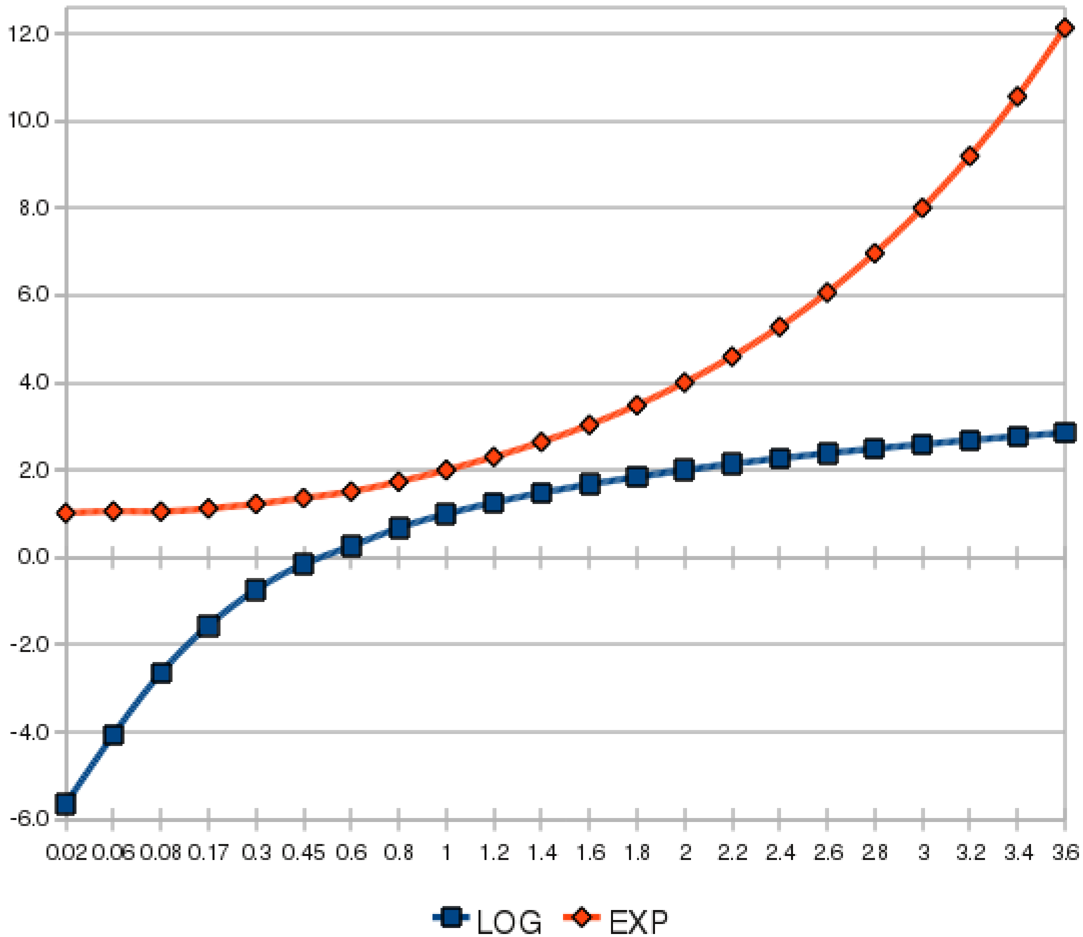
- Análise Assintótica: ordens de crescimento
  - Revisão de matemática
  - Abordagem: contagem de operações e tamanho da entrada

- Bibliografia

Revisão de matemática

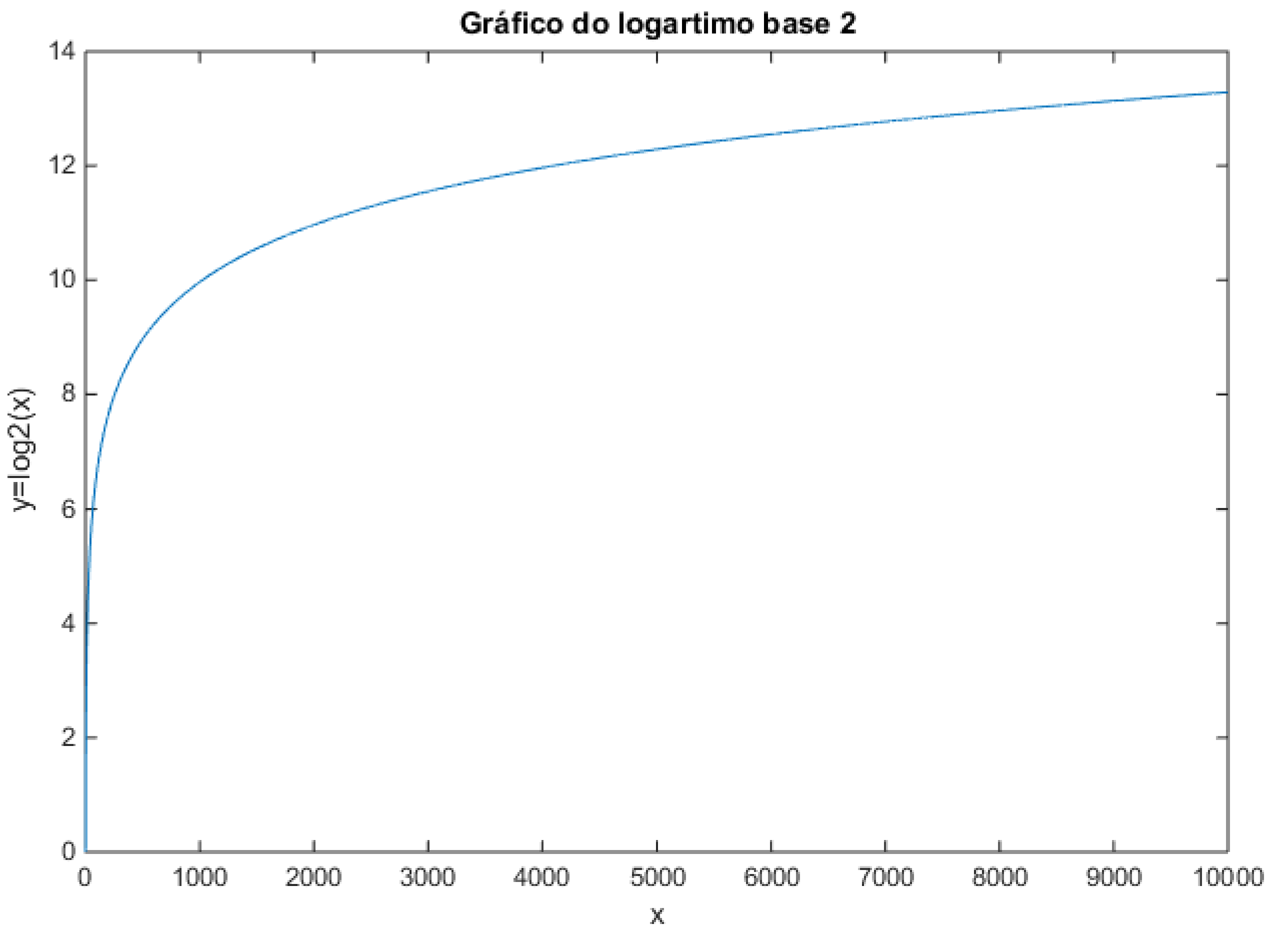
- Expoentes
  - $x^a \ x^b = x^{a+b}$
  - $x^a / \ x^b = x^{a-b}$
  - $(x^a)^b = \ x^{ab}$
  - $x^n + x^n = 2x^n$ (e não x^{2n})
  - $2^n + 2^n = 2^{n+1}$
- Logaritmos (por padrão, base 2)
  - $(x^a = b) \rightarrow (\log_x b = a)$
  - $\log_a b = \log_c b \, / \log_c a$ para $c > 0$
  - $\log ab = \log a + \log b$
  - $\log a/b = \log a - \log b$
  - $\log(a^b) = \mathrm{b} \log a$
  - $\log x < x$ para todo $x > 0$
  - Notação: $\lg n = \log_2 n$ (logaritmo binário)

Função logarítmica X exponencial



Exponencial: $y = 2^x$, Logaritmo: $y = \log_2 x$

Sobre os logaritmos



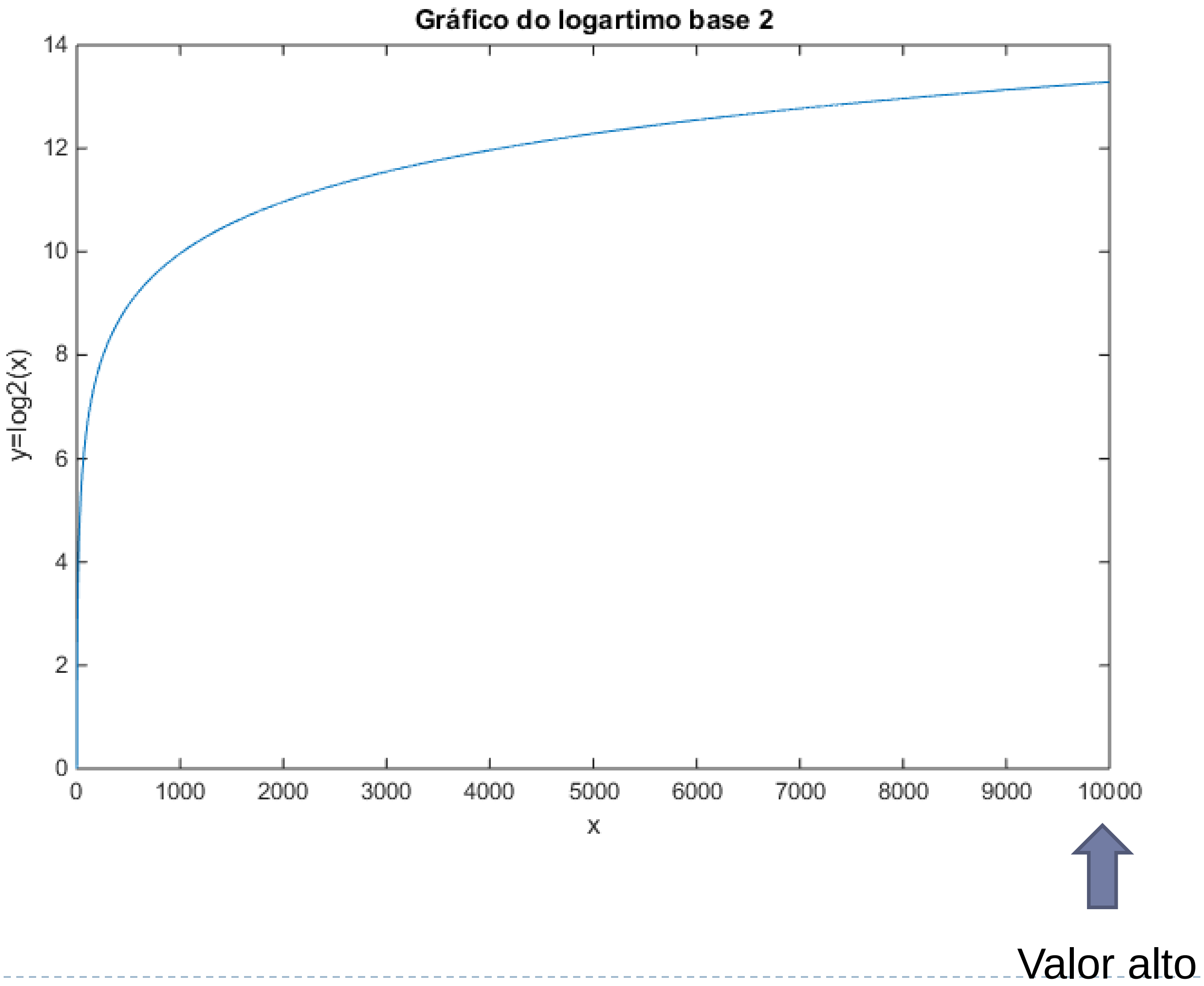
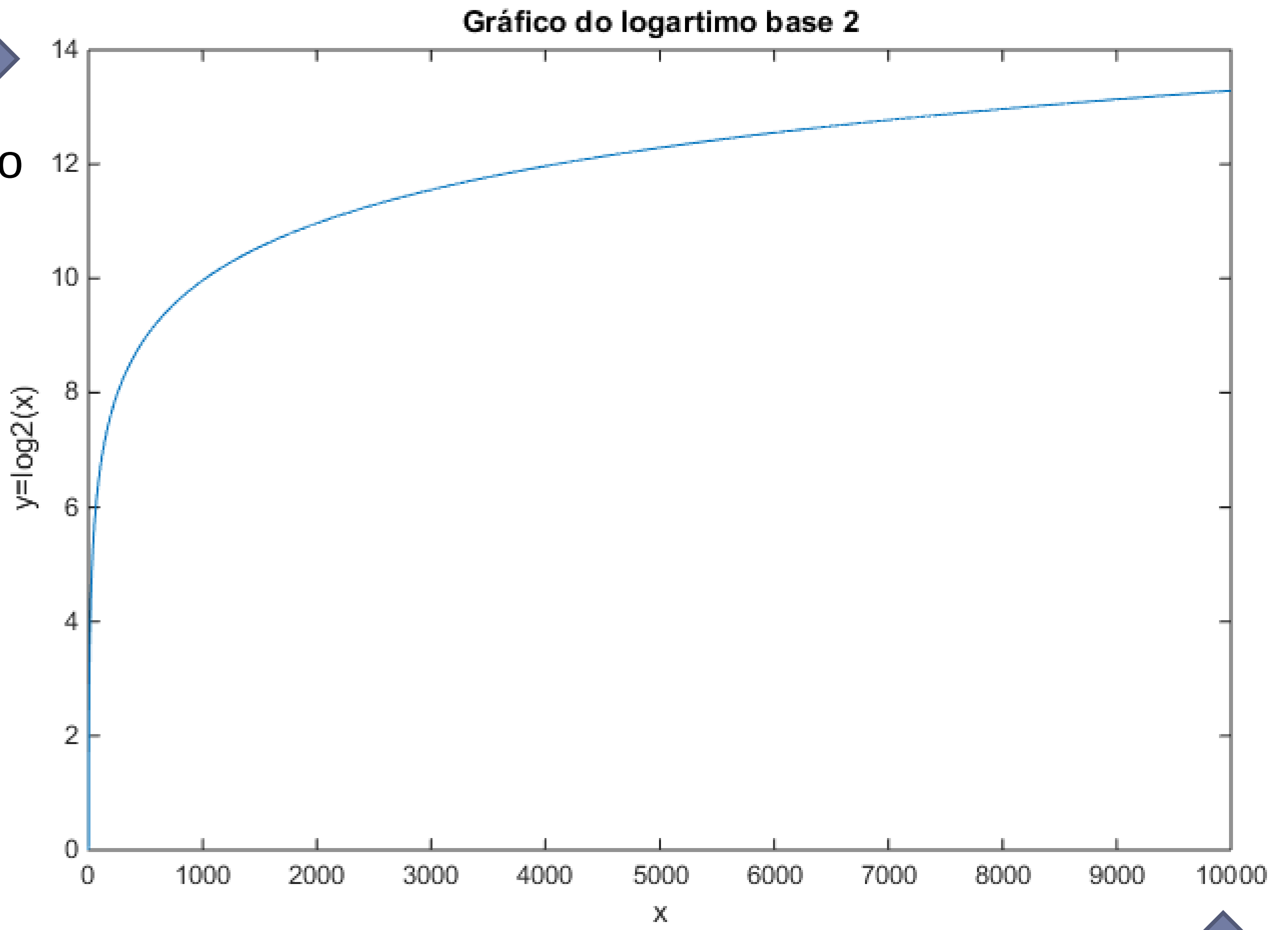


Gráfico do logartimo base 2
y=log2(x)
x
0
2
4
6
8
10
12
14
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Valor alto

Valor baixo



Valor alto

Logaritmos

- O que o logaritmo nos fornece é um número que corresponde ao expoente de um outro número. Por ser um expoente, seu valor é baixo e cresce bem lentamente

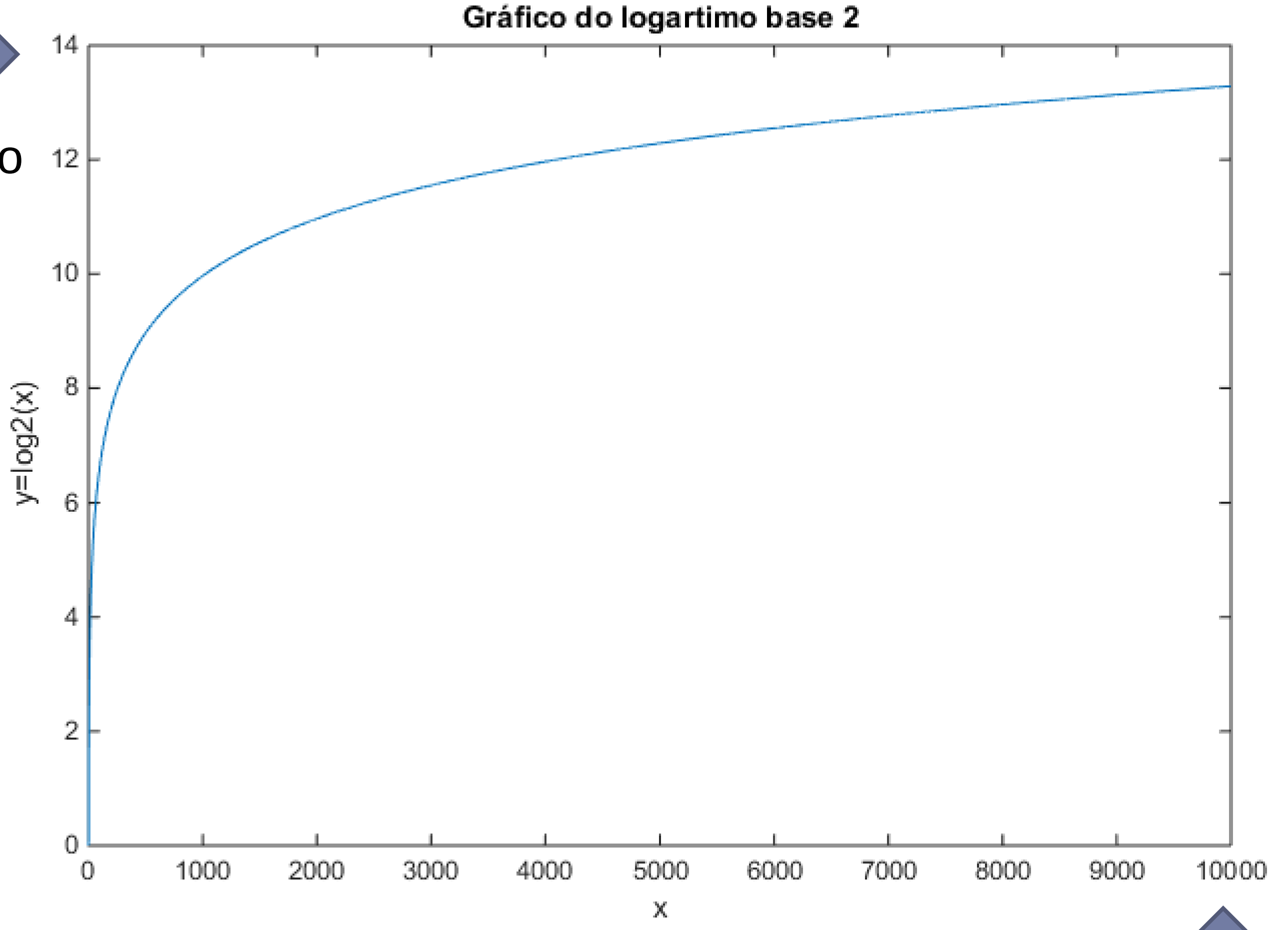
Logaritmos

- O que o logaritmo nos fornece é um número que corresponde ao expoente de um outro número. Por ser um expoente, seu valor é baixo e cresce bem lentamente
  - Qual é esse outro número??

Logaritmos

- O que o logaritmo nos fornece é um número que corresponde ao expoente de um outro número. Por ser um expoente, seu valor é baixo e cresce bem lentamente
  - Qual é esse outro número??
  - É a base do logaritmo

Valor baixo

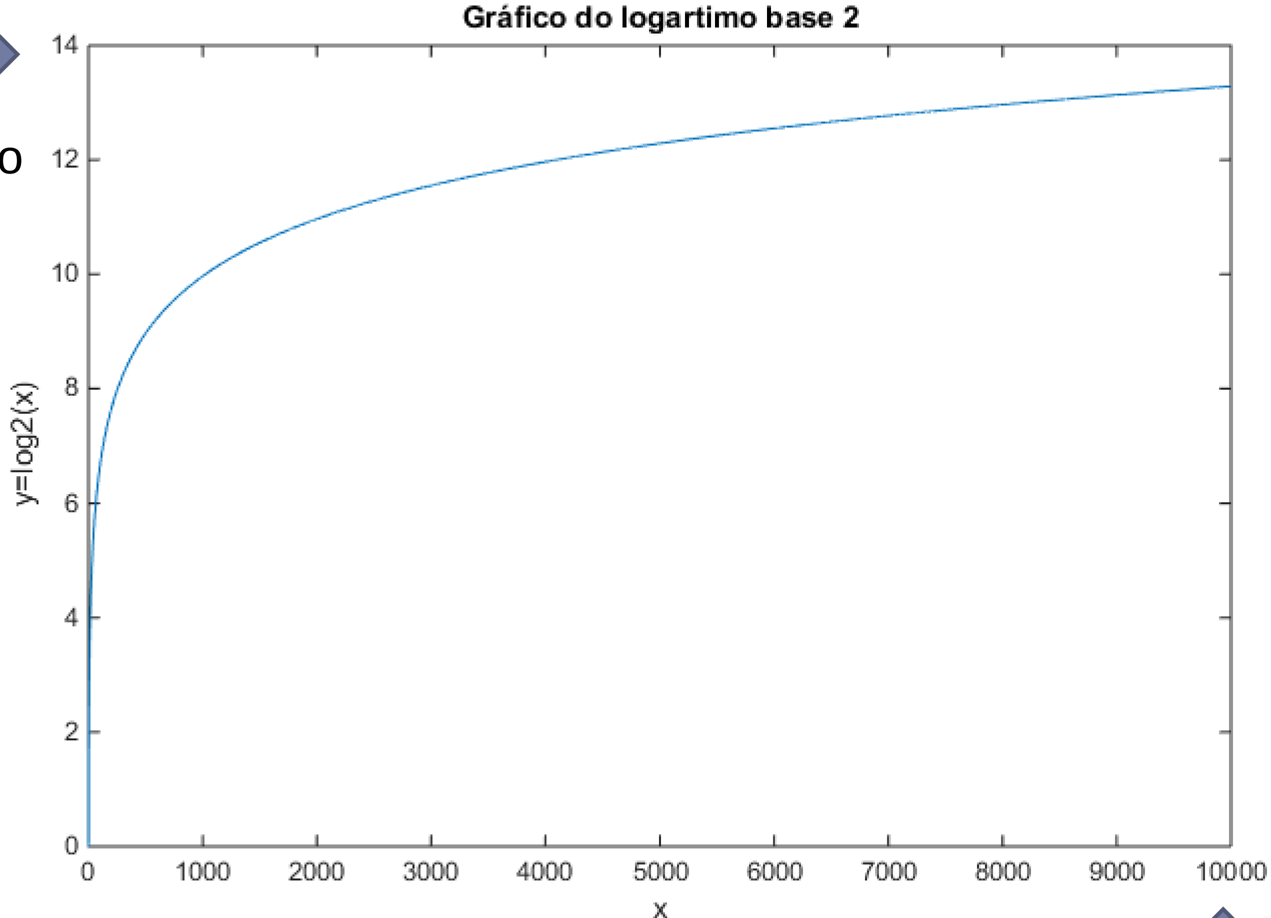


$$2^{y} = x \Leftrightarrow y = \log_2 x$$

Valor alto

A base é 2

Valor baixo

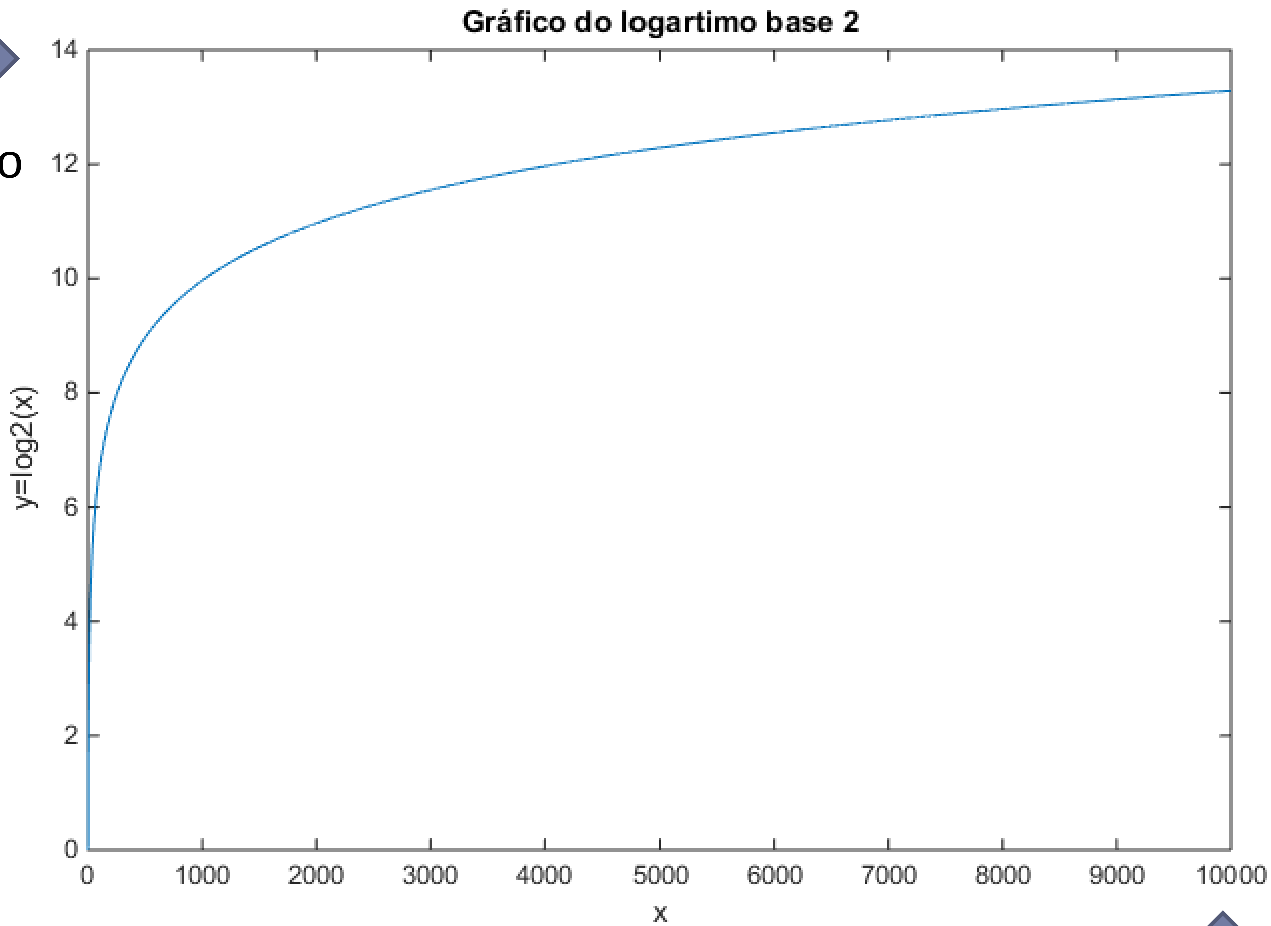


$$2^y = \boxed{x}$$

Valor alto

A base é 2

Valor baixo

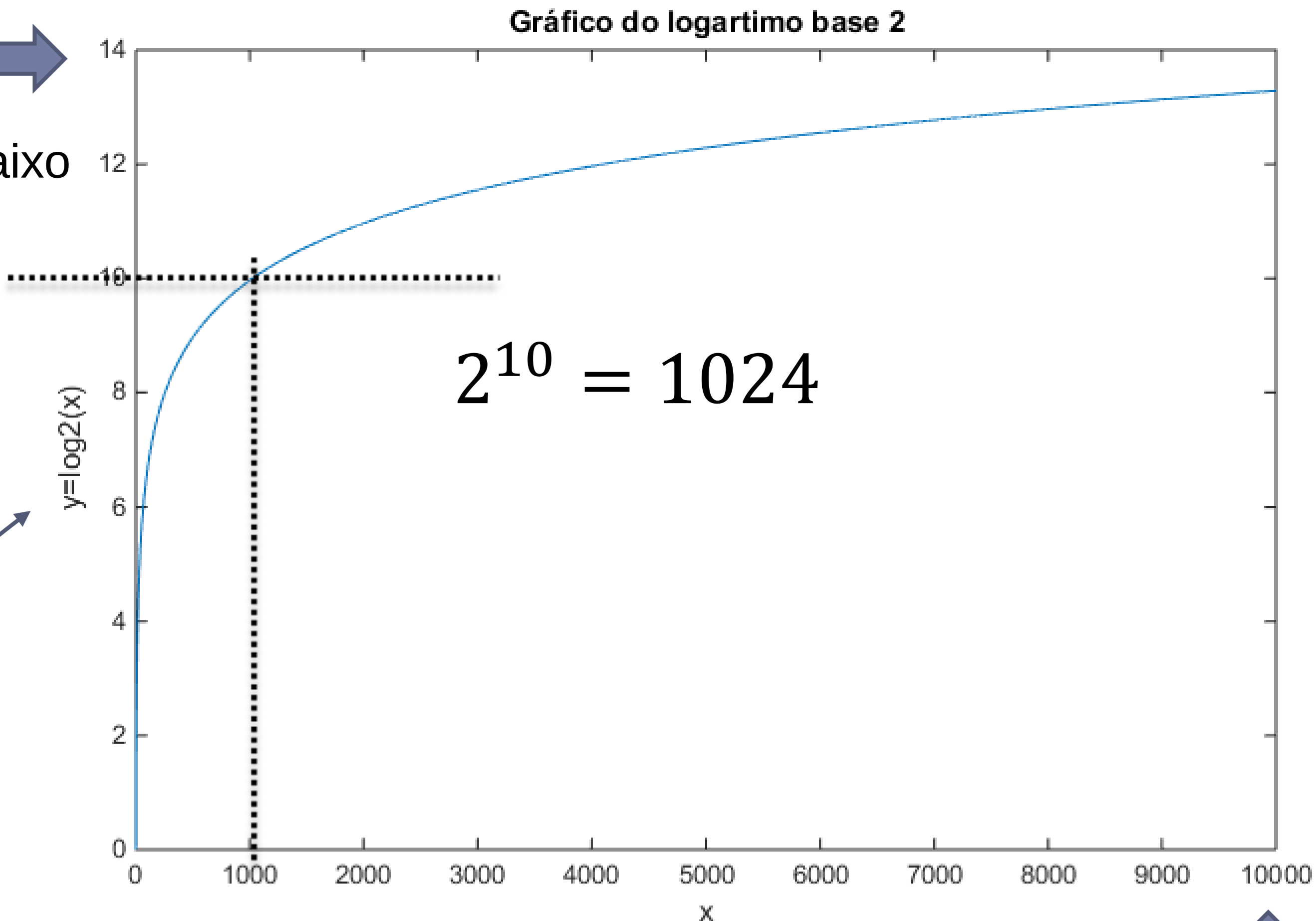


$$2^{\boxed{y}} = \boxed{x}$$

Valor alto

A base é 2

Valor baixo



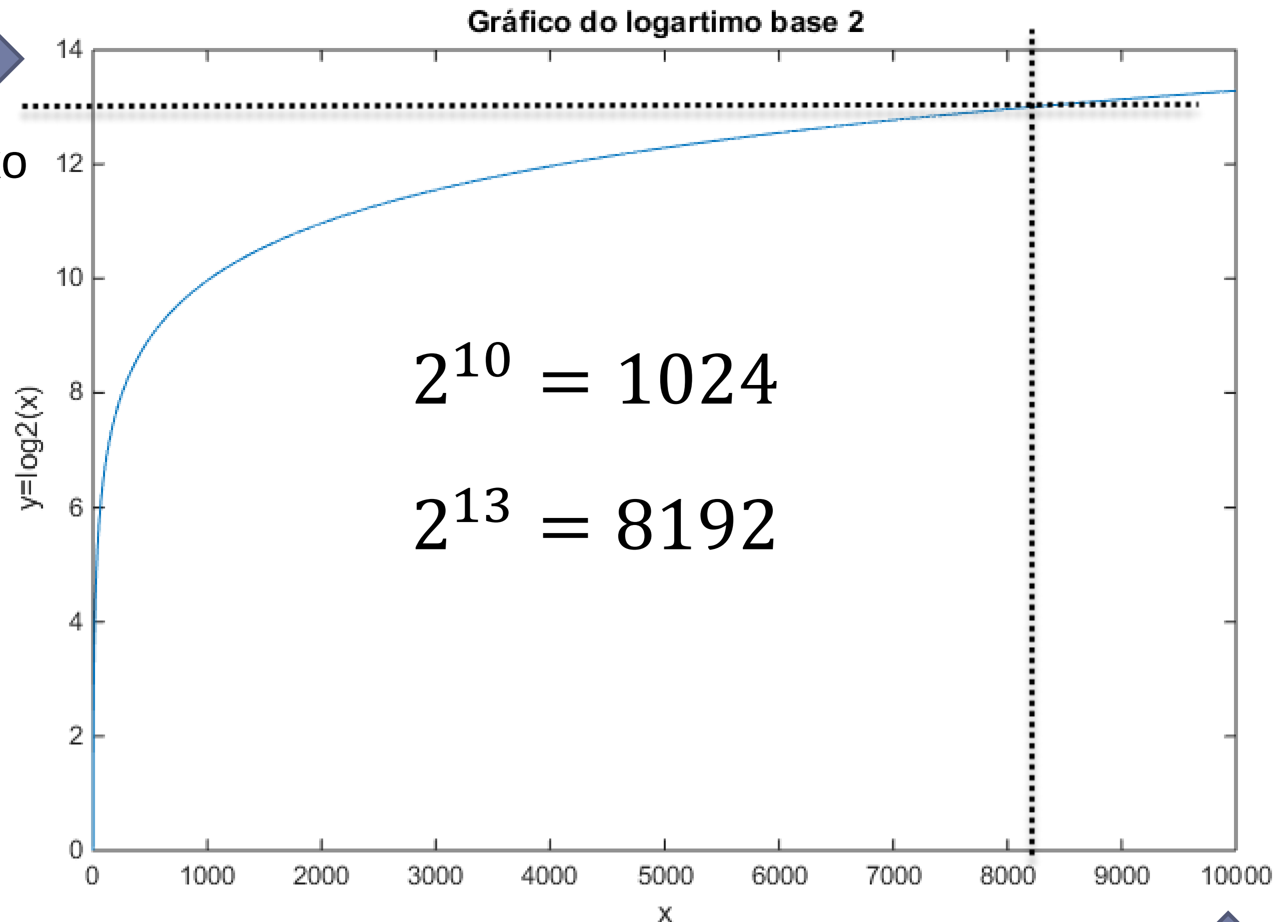
$$2^{10} = 1024$$

$$2^{\boxed{y}} = \boxed{x}$$

Valor alto

A base é 2

Valor baixo



$$2^{10} = 1024$$

$$2^{13} = 8192$$

$$2^{\boxed{y}} = \boxed{x}$$

Valor alto

A base é 2

Uso do log

- Dado um valor, usamos log quando queremos descobrir qual o expoente que resulta neste número.
- Exemplo:
  - Considere o número 1538. Qual é o expoente de 2 que resulta em 1538?

$$2^y = 1538$$

Uso do log

- Considere o número 1538. Qual é o expoente de 2 que resulta em 1538?

$$\boxed{2^y = 1538}$$

- Sabemos que $2^{10} = 1024$ e que $2^{11} = 2048$
- Então podemos concluir que o valor y procurado esta entre 10 e 11

$$\boxed{\begin{gathered} 2^y = 1538 \\ y = \log_2 1538 \end{gathered}}$$

- Consultando uma tabela, vemos que $y = 10{,}5868$

$$\boxed{2^{10{,}5868} = 1538}$$

Revisão de matemática

- Séries

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

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

Abordagem: contagem de operações e tamanho da entrada

- Relembrando o objetivo
  - ser capaz de, dado um problema, mapeá-lo em uma **classe de algoritmos** e encontrar a **“melhor” escolha** entre os algoritmos, com base em sua eficiência.
- Complexidade computacional e eficiência
  - A **complexidade computacional** está ligada à eficiência. Algoritmos mais caros computacionalmente são menos eficientes.
- Abordagem para análise assintótica da complexidade
  - O número de **passos básicos** necessários em função do **tamanho da entrada** que o algoritmo recebe.
    - descorrelaciona a performance da máquina da performance do algoritmo.
    - reduz a análise ao número de operações realizadas em função do tamanho da entrada.

Análise Assintótica: passos básicos e tamanho da entrada

- Tamanho da entrada?
  - Depende do problema, mas geralmente é relativo ao número de elementos da entrada que são processados pelo algoritmo
    - o número de elementos em um arranjo, lista, árvore, etc.
    - o tamanho de um inteiro que é passado por parâmetro.
- Passos básicos?
  - Se referem às operações primitivas utilizadas pela máquina:
    - operações aritméticas,
    - comparações,
    - atribuições,
    - resolver um ponteiro ou referência,
    - indexação em um arranjo,
    - chamadas e retornos de funções.

Análise Assintótica: ordens de crescimento

- Suponha que
  - o tamanho da entrada é n
  - cada operação leva aproximadamente o mesmo tempo (constante).
  - a memória é infinita
- Eficiência com base na ordem de crescimento
  - a eficiência de um algoritmo representada por uma função
  - **eficiência assintótica** descreve a eficiência de um algoritmo quando **n** torna-se grande.
- Para comparar algoritmos
  - determinamos suas ordens de crescimento (eficiência assintótica)
  - o algoritmo com a **menor** ordem de crescimento deverá executar mais rápido para tamanhos de entradas maiores.

Análise Assintótica: ordens de crescimento

- A análise assintótica reduz o problema a uma resposta menos precisa, mas fácil de derivar e de interpretar.
- Algumas “consequências” desse tipo de análise são:
  - Ter de definir um **modelo de máquina** único com as operações básicas.
  - A eficiência de um algoritmo pode estar relacionada à detalhes dos dados de entrada além do seu tamanho, e portanto existem diversos cenários: **melhor caso**, **pior caso** e **caso esperado** (médio).

Análise Assintótica: ordens de crescimento

- Melhor caso: não é uma boa análise
  - Pode nunca ocorrer na prática!
- Caso esperado (médio): seria o ideal (intuitivamente), mas determiná-lo não é uma tarefa trivial.
  - Usado em algumas situações
  - É preciso conhecer a **distribuição de probabilidade** (descreve a chance de uma variável aleatória assumir um valor ao longo de um espaço de valores) típica da entrada, e utilizar teoria da probabilidade para determinar.
- Pior caso: recomendado
  - Fácil de identificar
  - Como se trata de um **limite superior** sobre o tempo de execução para cada entrada, não há surpresas!
  - Para diversos algoritmos o pior caso ocorre com frequência
  - Em muitos casos o caso esperado está próximo ao pior caso.

Contagem de operações

```
// entrada: arranjo A de n inteiros
// saída: elemento máximo em A
int maxArranjo(int *A, int n) {
    int i, atualMax = *A;
    for (i = 0; i < n; i++) {
            if (*A >= atualMax) {
                    atualMax = *A;
            }
            A++;
    }
    return atualMax;
}
```

Quantas operações são efetuadas nesta função?

*ps: esse algoritmo não é o mais eficiente para facilitar o cálculos a seguir

Contagem de operações

```
// entrada: arranjo A de n inteiros
// saída: elemento máximo em A
int maxArranjo(int *A, int n) {
    int i, atualMax = *A;
    for (i = 0; i < n; i++) {
        if (*A >= atualMax) {
            atualMax = *A;
        }
        A++;
    }
    return atualMax;
}
```

Número de operações: 2

1 desreferenciamento (*A)
1 atribuição (=)

Contagem de operações

```
// entrada: arranjo A de n inteiros
// saída: elemento máximo em A
int maxArranjo(int *A, int n) {
   int i, atualMax = *A;
   for (i = 0; i < n; i++) {
         if (*A >= atualMax) {
                  atualMax = *A;
         }
         A++;
   }
   return atualMax;
}
```

Em um laço for, temos que analisar três elementos

1) A <u>inicialização</u> da variável que armazena o número de vezes que o laço é executado

i = 0

2) A <u>condição</u> que controla o número de vezes que o laço é executado

i < n

3) O <u>incremento</u> que atualiza a variável que controla o número de vezes que o laço é executado

i++

Contagem de operações

```
// entrada: arranjo A de n inteiros
// saída: elemento máximo em A
int maxArranjo(int *A, int n) {
   int i, atualMax = *A;
   for (i = 0; i < n; i++) {
         if (*A >= atualMax) {
               atualMax = *A;
         }
         A++;
   }
   return atualMax;
}
```

1) A inicialização da variável que armazena o número de vezes que o laço é executado

i = 0

É feita somente uma operação de atribuição. Então:

Num. Operações: 1

Contagem de operações

```
// entrada: arranjo A de n inteiros
// saída: elemento máximo em A
int maxArranjo(int *A, int n) {
   int i, atualMax = *A;
   for (i = 0; i < n; i++) {
         if (*A >= atualMax) {
               atualMax = *A;
         }
         A++;
   }
   return atualMax;
}
```

2) A condição, que controla o número de vezes que o laço é executado

i < n

A cada passo do loop esse teste é feito. Note que para poder sair do loop, a condição (i<n) deve ser falsa. Ou seja, n+1 testes são feitos até quebrar o loop.

Exemplo:

Se n = 2, o teste i<n é feito 3 vezes:

i < 0 → Verdadeiro

i < 1 → Verdadeiro

i < 2 → Falso

Num. Operações: n+1

Contagem de operações

```
// entrada: arranjo A de n inteiros
// saída: elemento máximo em A
int maxArranjo(int *A, int n) {
   int i, atualMax = *A;
   for (i = 0; i < n; i++) {
         if (*A >= atualMax) {
               atualMax = *A;
         }
         A++;
   }
   return atualMax;
}
```

3) Incremento, que atualiza a variável que controla o número de vezes que o laço é executado

i++

lembre que i++ equivale à i = i +1

Assim, i++ possui duas operações, uma soma e uma atribuição. E, dentro de um loop, essa operação é realizada n vezes

Exemplo: n=2, i inicializa em 0

i++ → i =1

i++ → i =2 (quando o loop falha)

Num. Operações: 2n

Contagem de operações

```
// entrada: arranjo A de n inteiros
// saída: elemento máximo em A
int maxArranjo(int *A, int n) {
   int i, atualMax = *A;
   for (i = 0; i < n; i++) {
         if (*A >= atualMax) {
               atualMax = *A;
         }
         A++;
   }
   return atualMax;
}
```

Número de operações total do laço *for*: 3n+2

1 atribuição (i=0)
n+1 comparações (i < n)
n atribuições (i++)
n somas (i++)

Contagem de operações

```
// entrada: arranjo A de n inteiros
// saída: elemento máximo em A
int maxArranjo(int *A, int n) {
    int i, atualMax = *A;
    for (i = 0; i < n; i++) {
        if (*A >= atualMax) {
            atualMax = *A;
        }
        A++;
    }
    return atualMax;
}
```

Os comandos dentro do loop são executados n vezes.

Dessa forma, faremos a contagem dessas operações e multiplicaremos por n, pois elas ocorrem n vezes

Contagem de operações

```
// entrada: arranjo A de n inteiros
// saída: elemento máximo em A
int maxArranjo(int *A, int n) {
    int i, atualMax = *A;
    for (i = 0; i < n; i++) {
        if (*A >= atualMax) {
                atualMax = *A;
        }
        A++;
    }
    return atualMax;
}
```

Número de operações:
2

1 desreferenciamento (*A)
1 operação relacional (>)

Num. Operações total: 2n

(pois o loop faz com que essas operações repitam n vezes)

Contagem de operações

```
// entrada: arranjo A de n inteiros
// saída: elemento máximo em A
int maxArranjo(int *A, int n) {
   int i, atualMax = *A;
   for (i = 0; i < n; i++) {
        if (*A >= atualMax) {
             atualMax = *A;
        }
        A++;
   }
   return atualMax;
}
```

Número de operações:
2

1 desreferenciamento (*A)
1 atribuição (=)

Num. Operações total: 2n

isso no pior caso, pois esse comando pode nunca ser executado – caso em que o primeiro elemento é o maior do arranjo

Contagem de operações

```
// entrada: arranjo A de n inteiros
// saída: elemento máximo em A
int maxArranjo(int *A, int n) {
    int i, atualMax = *A;
    for (i = 0; i < n; i++) {
        if (*A >= atualMax) {
            atualMax = *A;
        }
        A++;
    }
    return atualMax;
}
```

Número de operações:
2

1 soma
1 atribuição

Num. Operações total: 2n

Contagem de operações

```
// entrada: arranjo A de n inteiros
// saída: elemento máximo em A
int maxArranjo(int *A, int n) {
    int i, atualMax = *A;
    for (i = 0; i < n; i++) {
        if (*A >= atualMax) {
            atualMax = *A;
        }
        A++;
    }
    return atualMax;
}
```

Número de operações: 1

1 retorno

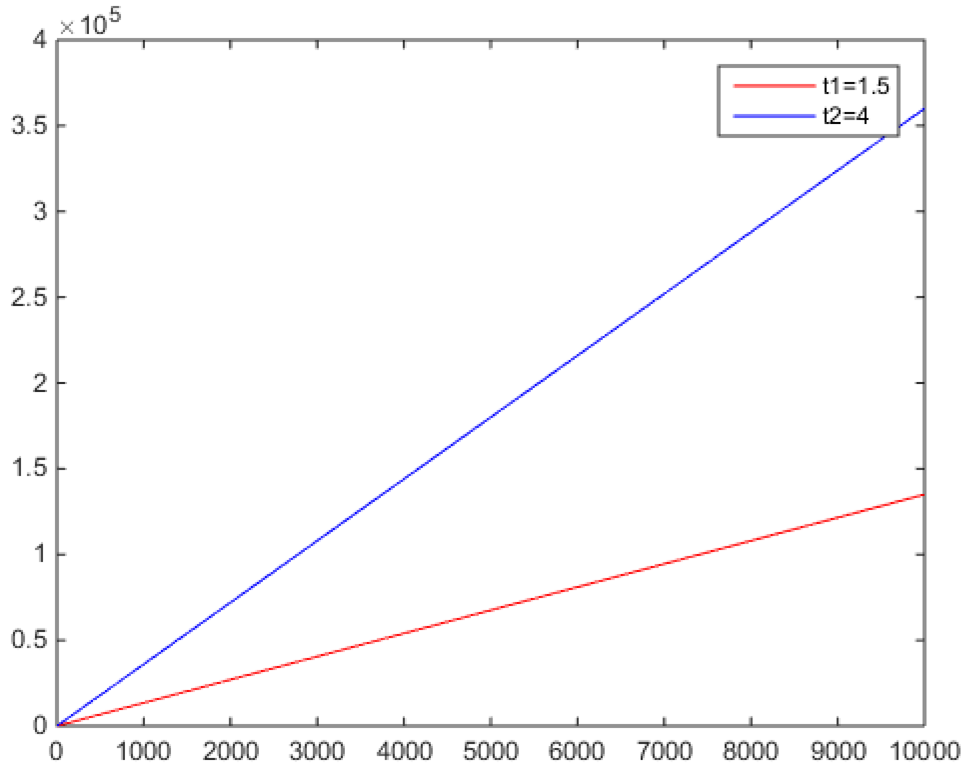
Contagem de operações

	Número de operações
`// entrada: arranjo A de n inteiros`	
`// saída: elemento máximo em A`	
`int maxArranjo(int *A, int n) {`	
`    int i, atualMax = *A;`	**2**
`    for (i = 0; i < n; i++) {`	**3n +2**
`        if (*A >= atualMax) {`	**2n**
`            atualMax = *A;`	**2n**
`        }`	
`        A++;`	**2n**
`    }`	
`    return atualMax;`	**1**
`}`	

Estimando a eficiência de maxArranjo

- O algoritmo executa:
  - No pior caso: $2 + 3n + 2 + 3 * (2n) + 1 = 9n + 5$ operações.
  - No melhor caso:$2 + 3n + 2 + 2 * (2n) + 1 = 7n + 5$ operações.
    - Caso em que if (*A > atualMax) é falso.
- Suponha que:
  - t_1 é o tempo gasto pela primitiva (operação) mais rápida
  - t_2 é o tempo gasto pela primitiva (operação) mais lenta
- Se T(n) é o tempo de **pior caso** de maxArranjo, então:
  - $$t_1(9n + 5) \leq T(n) \leq t_2(9n + 5)$$
- T(n) é limitado por duas funções **lineares**.

Estimando a eficiência de maxArranjo



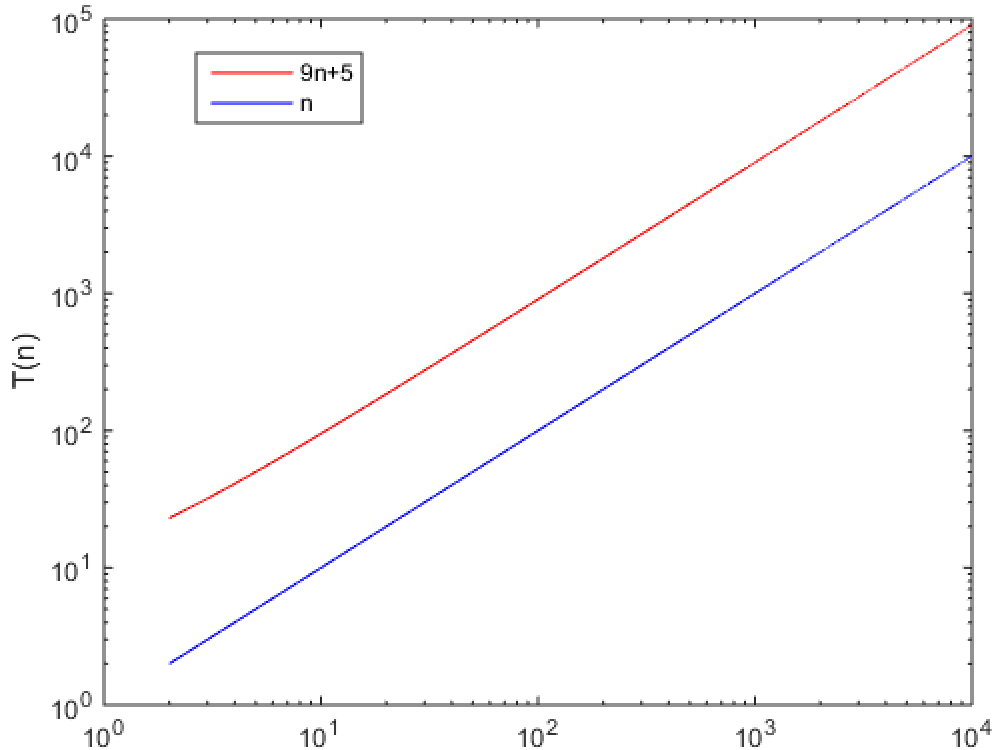
Exemplo: $t_1 = 1{,}5$ e $t_2 = 4{,}0$

Estimando a eficiência de maxArranjo

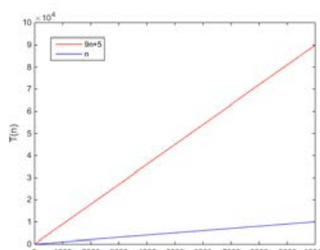
- Após essa análise, a qual conclusão podemos chegar quanto à eficiência do algoritmo quando o tamanho da entrada aumenta?
  - **A função que caracteriza a complexidade computacional do algoritmo é de ordem linear.**
  - Podemos desprezar todos os fatores constantes e termos de menor ordem, pois esses não afetam a taxa de crescimento em si. $t(n) = 9n + 5 \approx n$

T(n)	1	10	100	1.000	10.000
9n+5	14	95	905	9.005	90.005
9n	9	90	900	9.000	90.000
n	1	10	100	1.000	10.000

Estimando a eficiência de maxArranjo



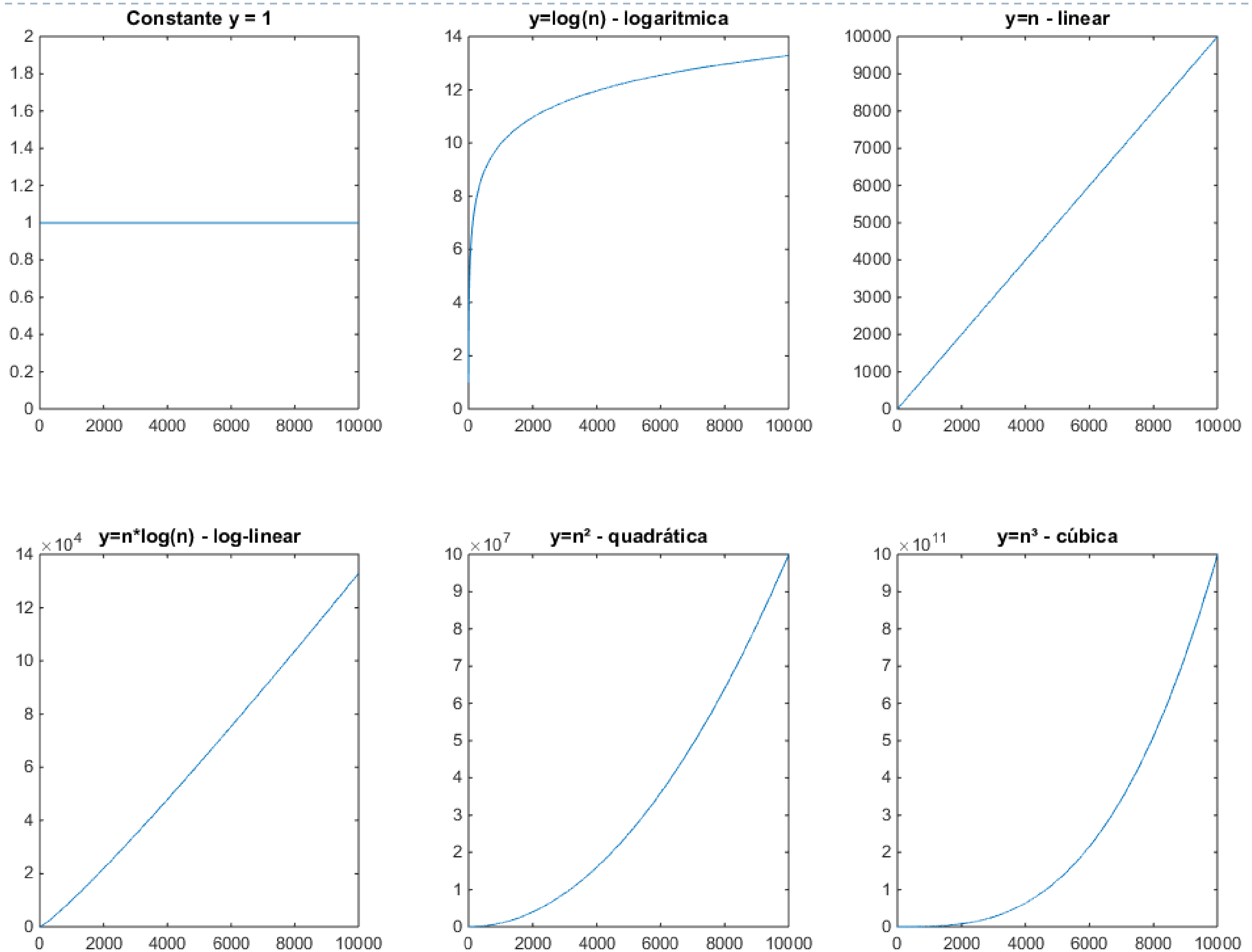
Exemplo: em escala logarítmica – essa escala permite visualizar a taxa de crescimento – em ambos casos o crescimento é o mesmo



Funções importantes

- As funções que aparecem na análise de algoritmos:
  - Constante: ≈ 1
  - Logarítmica: $\approx \log_b n$
  - Linear: $\approx n$
  - Log linear (ou n-log-n): $\approx n \cdot \log_b n$
  - Quadrática: $\approx n^2$
  - Cúbica: $\approx n^3$
  - Exponencial: $\approx a^n$
  - Fatorial: $\approx n!$

Funções importantes



Observe a ordem de grandeza no eixo y

- Encontre o erro

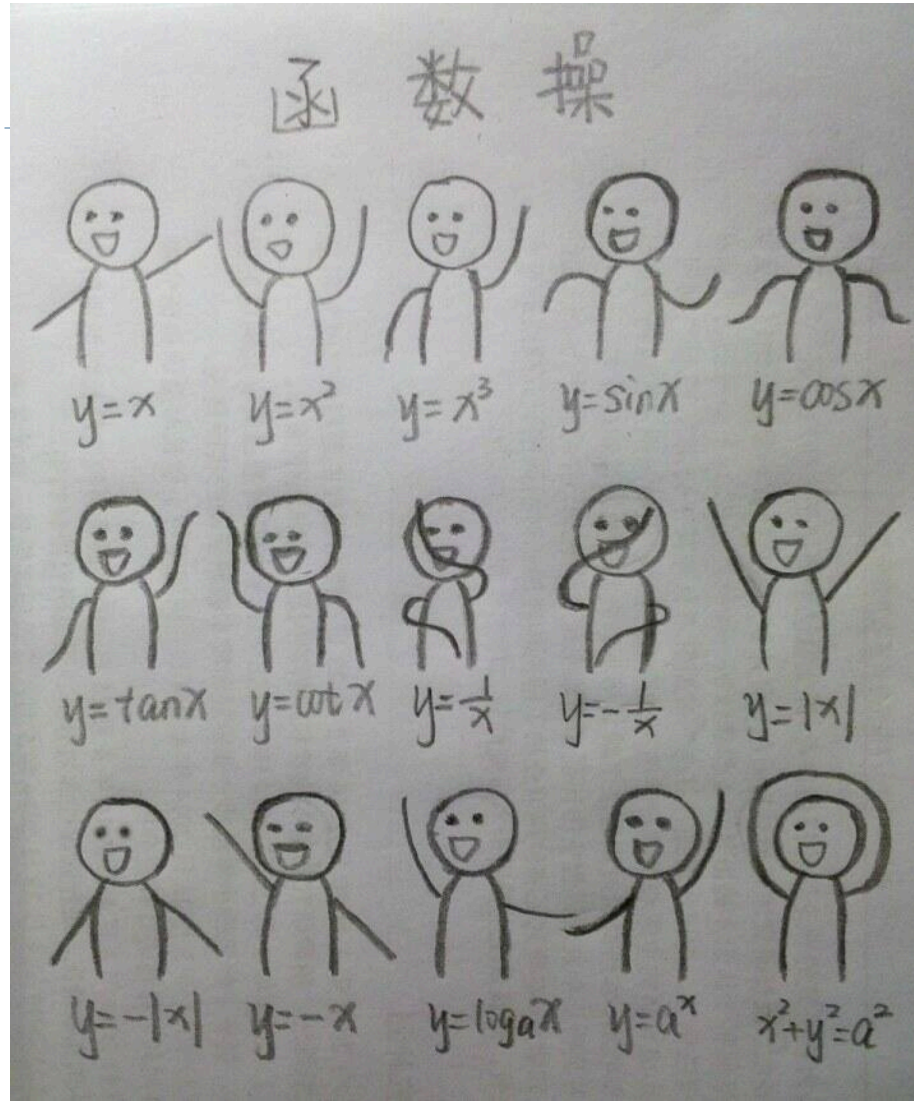


Imagem: facebook

- Encontre o erro

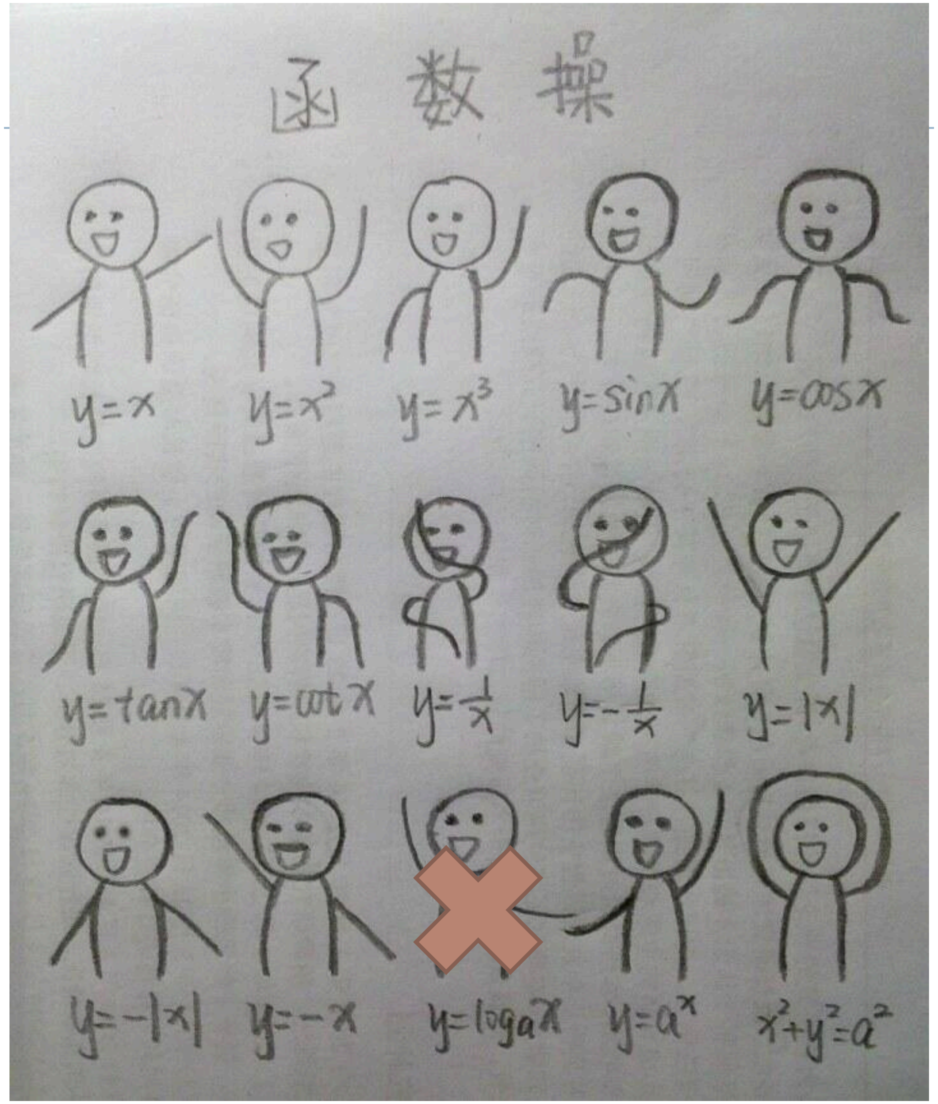
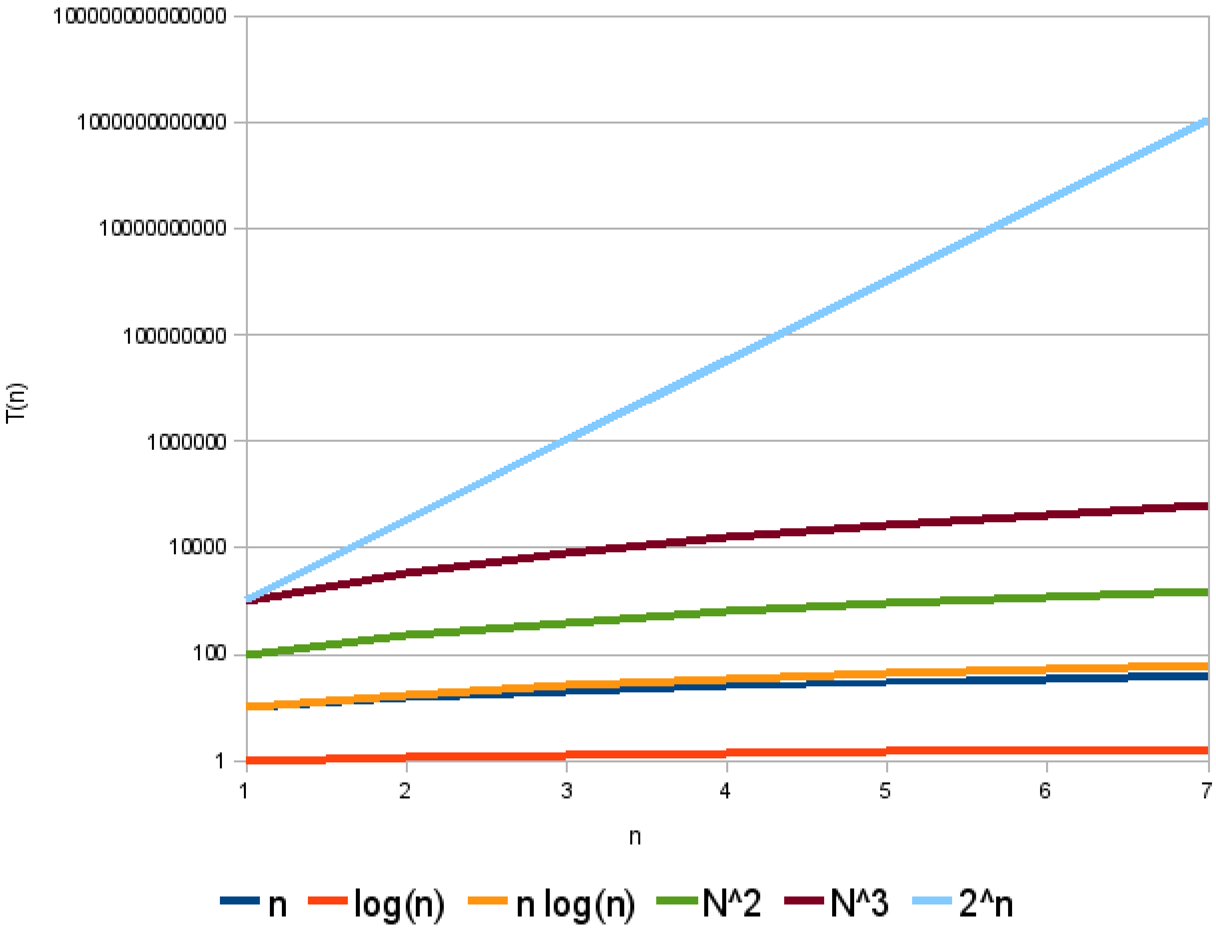


Imagem: facebook

Funções importantes



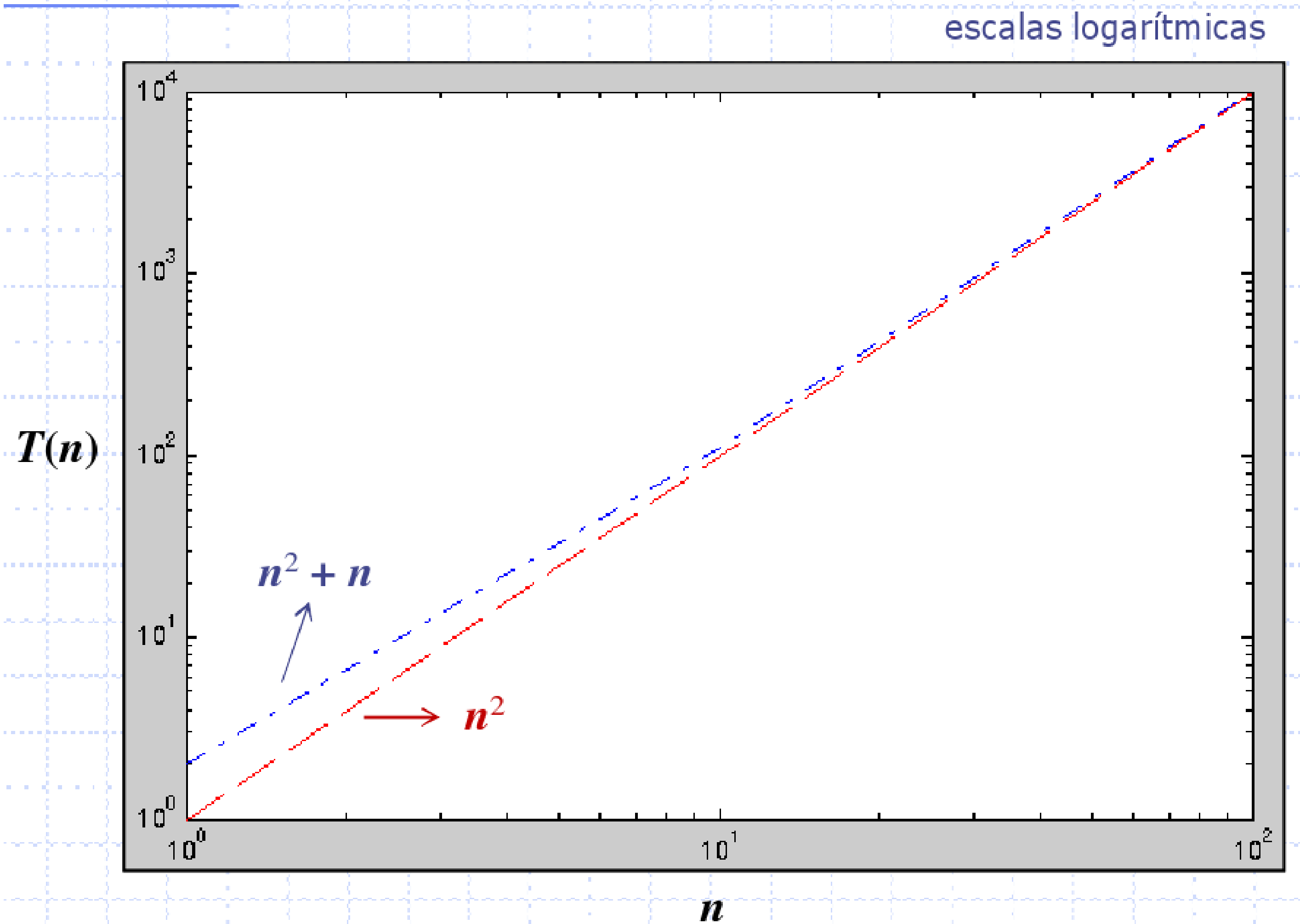
Exemplo: em escala logarítmica – para ser possível visualizar todas as função ao mesmo tempo

Fatores constantes e de termos de menor ordem

- Mudar o hardware/software afeta a taxa de crescimento por um fator constante, mas não altera a ordem de complexidade dos algoritmos.
- Termos de menor ordem também não afetam o crescimento conforme o tamanho da entrada cresce

T(n)	1	10	100	1.000
n^2	1	100	10.000	1.000.000
n^2+n	2	110	10.100	1.001.000
Δ	100%	10%	1%	0,1%

Fatores constantes e de termos de menor ordem



Fonte da figura: notas de aula do Prof. Ricardo Campello

Exercício

- Quantas unidades de tempo são necessárias para rodar o algoritmo abaixo? Mostre sua a função de complexidade

```
Inicio
  i, j: inteiro
  A: vetor inteiro de n posicoes
  i = 1
  enquanto (i < n) faca
       A[i] = 0
       i = i + 1
  para i = 1 ate n faca
       para j = 1 ate n faca
              A[i] = A[i] + (i*j)
fim
```

Bibliografia

- Slides: Prof. Moacir Ponti Jr. ICMC-USP 02-Análise de Algoritmos(1) 2014/1

- CORMEN, T. H. et al. **Algoritmos: Teoria e Prática** (Caps. 1-3). Campus. 2002.
- ZIVIANI, N. **Projeto de algoritmos**: com implementações em Pascal e C (Cap. 1). 2. ed. Thomson, 2004.
- FEOFILOFF, P. **Minicurso de Análise de Algoritmos**, 2010. Disponível em: http://www.ime.usp.br/~pf/livrinho-AA/.
- DOWNEY, A. B. **Analysis of algorithms** (Cap. 2), Em: Computational Modeling and Complexity Science. Disponível em: http://www.greenteapress.com/compmod/html/book003.html
- ROSA, J. L. **Notas de Aula de Introdução a Ciência de Computação II**. Universidade de São Paulo. Disponível em: http://coteia.icmc.usp.br/mostra.php?ident=639