Introdução à Ciência da Computação II

Análise de Algoritmos: Parte I

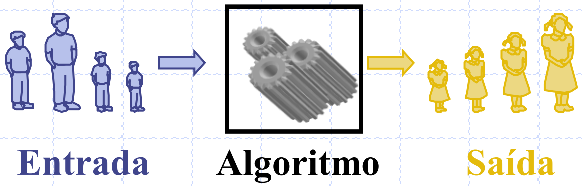
Prof. Ricardo J. G. B. Campello

Este material consiste de adaptações e extensões de slides disponíveis em http://ww3.datastructures.net (Goodrich & Tamassia).

Sumário

- Noções de Análise de Algoritmos
  - Método Experimental e Suas Limitações
  - Contagem de Operações Primitivas
  - Taxa de Crescimento da Contagem
  - Funções Importantes para Mensurar a Contagem
  - Exemplos e Exercícios

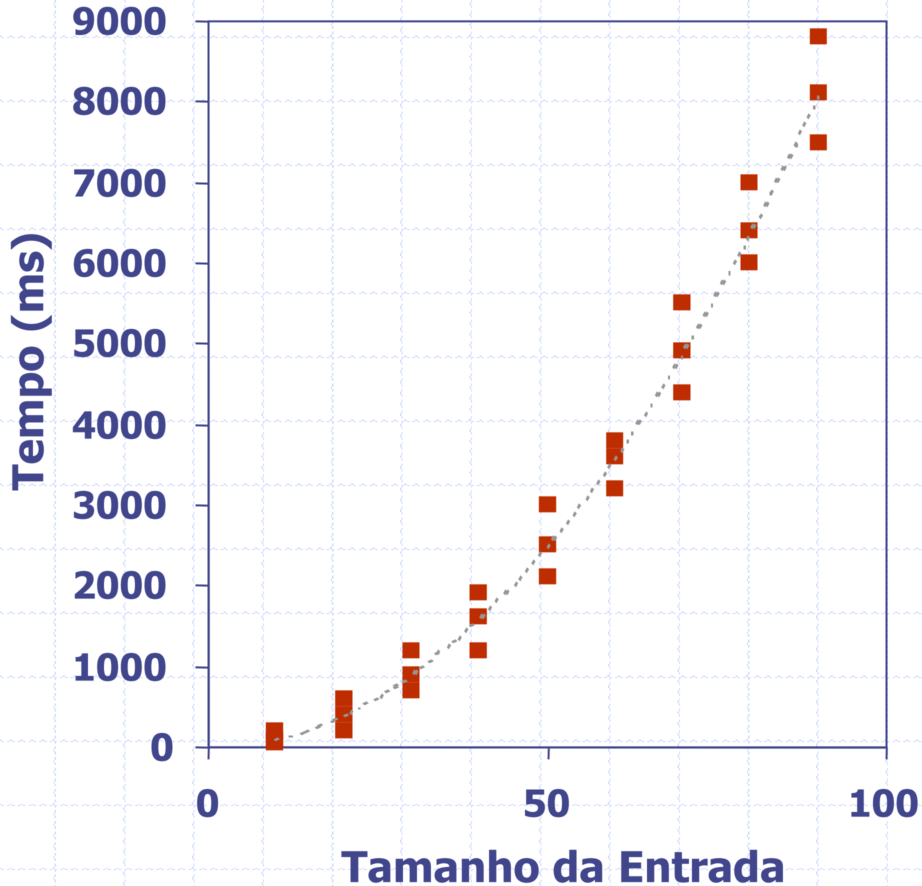
Tempo de Execução



- Um **algoritmo** é um procedimento passo-a-passo para realizar uma dada tarefa em tempo finito
- O **tempo de execução** de um algoritmo tipicamente cresce à medida que se aumenta o tamanho de sua entrada
- O **tamanho da entrada** é o valor de uma dada grandeza de interesse que reflete o tamanho de uma instância particular de problema a ser solucionado pelo algoritmo
- Exemplo (tamanho da entrada para um algoritmo de ordenação):
  - tamanho da entrada = no. de elementos a serem ordenados

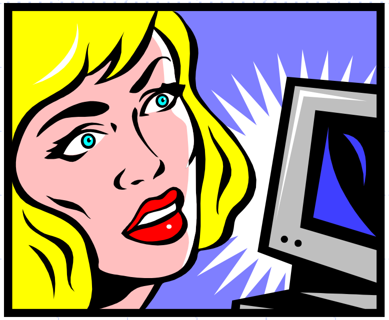
Método Experimental

- Escreva um programa que implemente um algoritmo
- Execute o programa variando a composição e o tamanho da entrada
- Use alguma funcionalidade da linguagem para obter medidas do tempo. Por exemplo:
  - Função clock() de C (time.h)
  - Método currentTimeMillis() de Java (classe System)
- Esboce um gráfico

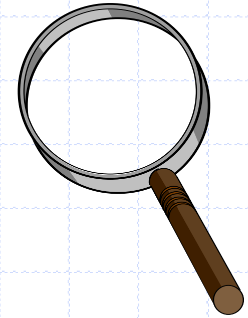


Limitações do Método Experimental

- É preciso implementar o algoritmo, muitas vezes complexo
- Os resultados podem não servir como indicativo do tempo de execução para outras entradas que não foram consideradas nos testes
- Para comparar dois algoritmos, devem ser utilizadas exatamente as mesmas condições, configurações e ambientes de hardware e software



Análise Teórica



- Baseada em uma descrição de alto nível do algoritmo, ao invés de uma dada implementação / linguagem
  - Descrição de alto nível: por ex., **pseudo-código**
- Caracteriza o tempo de execução como uma função do tamanho da entrada, n
- Leva em consideração todas as possíveis entradas
- Permite avaliar a rapidez de um algoritmo de forma independente de qualquer ambiente de hardware e/ou software

Pseudo-Código

- Descrição de alto-nível de um algoritmo
- Mais estruturada que texto simples
- Menos detalhada que um programa
- Notação muito utilizada para descrever algoritmos
- Esconde detalhes de projeto dos programas
- Não existe um padrão universal

Exemplo: encontrar o valor máximo em um arranjo

```
Algoritmo arrayMax(A, n)
  Entrada: vetor A de n inteiros
  Saída: elemento máximo de A

  atualMax ← A[0]
  para i ← 1 até n – 1 faça
    se A[i] > atualMax então
      atualMax ← A[i]
  retorne atualMax
```

Exemplo de Convenção

- Controle do fluxo de execução:
  - **se** … **então** … [**senão** …]
  - **enquanto** … **faça** …
  - **repita** … **até que** …
  - **para** … **faça** …
  - indentação substitui chaves
- Declaração de rotinas:
  - **Algoritmo** ***nome*(*arg1*, *arg2*,** …)
    Entrada: …
    Saída: …
- Arranjos:
  - $A[0] \ldots A[N\text{-}1]$ (N elementos)
- Chamadas a rotinas:
  - ***nome*(*arg1*, *arg2*,** …)
- Retorno de rotinas:
  - **retorne** ***expressão*** ou ***valor***
- Expressões:

  $\leftarrow$ **Atribuição**
  (mesmo que $=$ em Java, C, C++)

  $=$ **Teste de igualdade**
  (mesmo que $==$ em Java, C, C++)

  n^2 Sobrescritos e outros formatos matemáticos são utilizados

Operações Primitivas

- São ações básicas executadas pelos algoritmos
- Aparecem no pseudo-código
- Não dependem da linguagem de programação
- Considera-se que tenham tempo de execução constante
  - que independe do algoritmo ou do problema em questão
- Exemplos:
  - Atribuição de valor a uma variável
  - Operação aritmética com dois números
  - Operação lógica
  - Comparação de duas variáveis simples
  - Indexação de célula em um arranjo
  - Seguir um ponteiro ou referência
  - Chamar ou retornar de uma rotina*

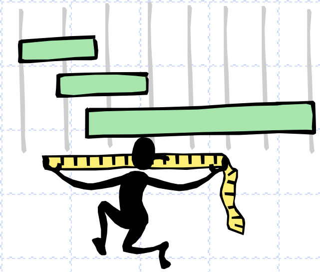
Contagem de Operações Primitivas

- Inspecionando o pseudo-código pode-se determinar o número máximo de operações primitivas executadas por um algoritmo, em função do tamanho da entrada n
- Exemplo (algoritmo para encontrar o valor máx. em um arranjo):

Algoritmo *arrayMax*(A, n)	# operações
$atualMax \leftarrow A[0]$	2
para $i \leftarrow 1$ **até** $n - 1$ **faça**	$1 + 2n$*
se $A[i] > atualMax$ **então**	$2(n - 1)$
$atualMax \leftarrow A[i]$	$2(n - 1)$
/* incremento implícito do contador i */	$2(n - 1)$*
retorne *atualMax*	1

Total: $6n \leq T(n) \leq 8n - 2$

Estimando o Tempo de Execução



- O algoritmo ***arrayMax*** executa:
  - $6n$ operações primitivas no **melhor caso**
  - $8n - 2$ operações primitivas no **pior caso**
- Suponhamos que:

  t_1 = tempo gasto pela operação primitiva mais rápida

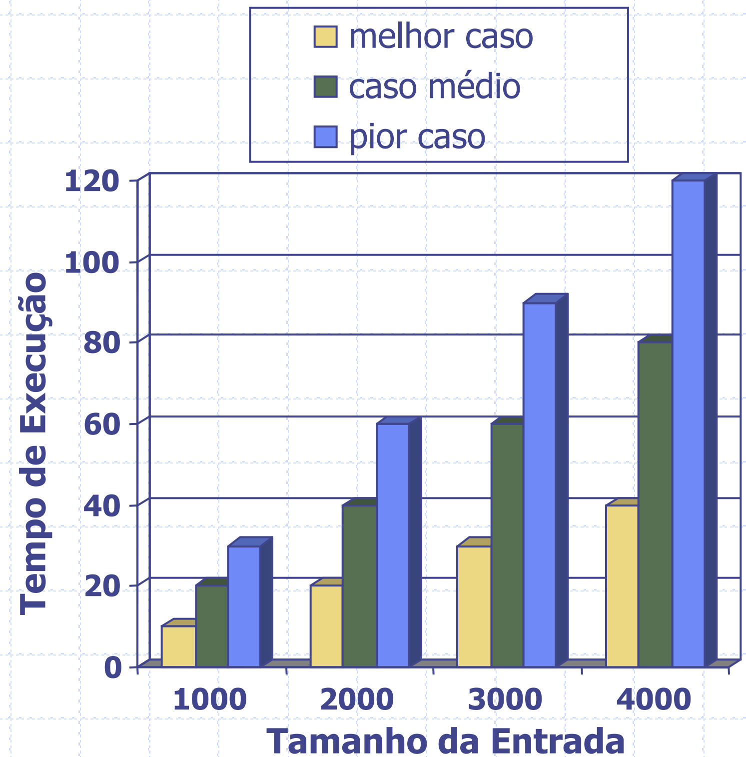
  t_2 = tempo gasto pela operação primitiva mais lenta
- Seja $T(n)$ o tempo de **pior caso** de ***arrayMax***. Então:

$$t_1\,(8n - 2) \;\leq\; T(n) \;\leq\; t_2\,(8n - 2)$$

- Portanto, $T(n)$ é limitado por duas funções **lineares**

Análise de Pior Caso

- O tempo de execução de um algoritmo tipicamente cresce à medida que se aumenta o *tamanho* de sua entrada
- O **caso médio** algumas vezes é difícil de se determinar
  - depende da *composição* da entrada
  - demanda teoria de probabilidade
- Em geral é melhor considerar o tempo de execução do **pior caso**:
  - Em geral é fácil identificar a pior entrada
  - Garantindo o desempenho do pior caso garante-se o desempenho dos demais



Outro Exemplo – Busca Binária

- O algoritmo compara o valor procurado x com o elemento que está no centro do arranjo V
  - Se forem iguais, o algoritmo pára
  - Se não forem, o algoritmo verifica se x pode estar na metade esquerda ou direita do arranjo, descartando a outra metade
- A busca prossegue na parte em que x pode ser encontrado...

Algoritmo $BB(V, n, x)$

Entrada: vetor V de n inteiros ordenados e valor x a ser encontrado

Saída: posição onde x foi encontrado no vetor ou –1 em caso de insucesso

```
i ← 0
s ← n – 1
enquanto i ≤ s faça
    k ← (i + s ) / 2;
    se x = V[k] então
            retorne k
    se x > V[k] então
            i ← k + 1
    senão   s ← k – 1
retorne (– 1)
```

Análise – Busca Binária

- É preciso descobrir quantas vezes o algoritmo repete o laço **enquanto** no pior caso (que corresponde ao insucesso da busca)
- Inicialmente, o intervalo de busca em V possui n elementos
  - $i \leftarrow 0$ e $s \leftarrow n-1$
- Após 1 comparação, restarão $n/2$ elementos
- Após 2 comparações, restarão $n/4$ elementos
- ...
- Após quantas comparações restarão **zero** elementos (**pior caso**)?
  - A resposta é $\lceil \log_2 n \rceil$ (teto($\log_2 n$) ou $\log_2 n$ arredondado para cima)

Análise de Pior Caso da BB

Algoritmo $BB(V, n, x)$

Entrada: vetor A de n inteiros ordenados e valor x a ser encontrado

Saída: posição onde x foi encontrado ou –1 se não for encontrado

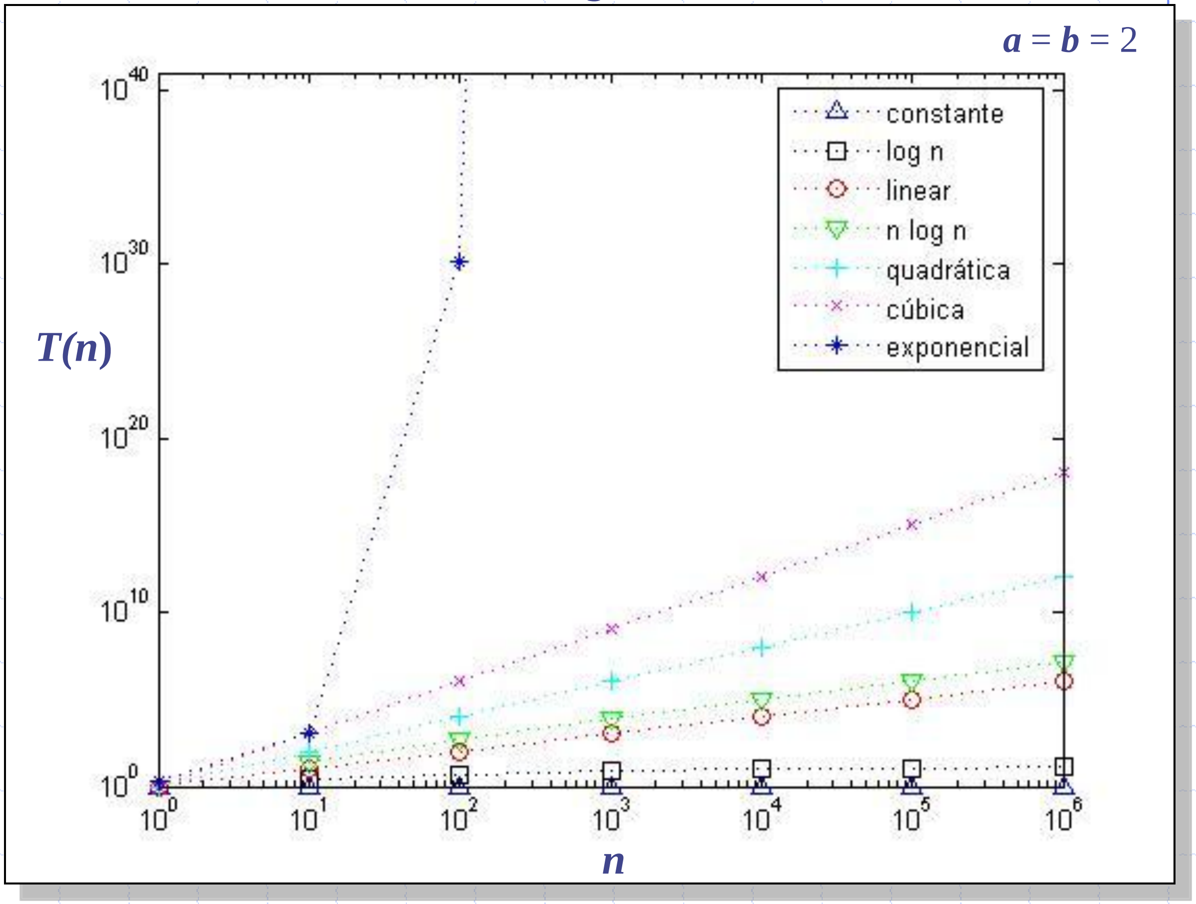
$i \leftarrow 0$	1
$s \leftarrow n - 1$	2
enquanto $i \leq s$ **faça**	$\log_2 n + 1$
$k \leftarrow (i + s)/2$;	$3 \log_2 n$
se $x = V[k]$ **então**	$2 \log_2 n$
retorne k	
se $x > V[k]$ **então**	$2 \log_2 n$
$i \leftarrow k + 1$	
senão $s \leftarrow k - 1$	$2 \log_2 n$
retorne (-1)	1

Total: $T(n) = 10 \log_2 n + 5$ (função **logarítmica**)

Sete Funções Importantes

- Sete funções que aparecem com mais frequência nas análises de algoritmos:
  - Constante: $\sim \mathbf{1}$
  - Logarítmica: $\sim \log_b n$
  - Linear: $\sim n$
  - n-log-n: $\sim n \log_b n$
  - Quadrática: $\sim n^2$
  - Cúbica: $\sim n^3$
  - Exponencial: $\sim a^n$
- Usualmente as bases logarítmica e exponencial são tais que $a = b = 2$
- Nesse curso adota-se a convenção de omitir a base logarítmica usual $b = 2$

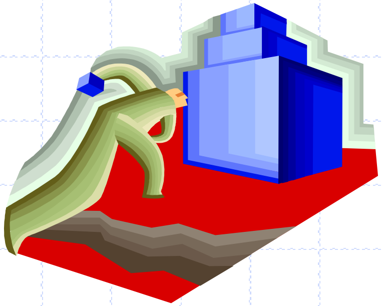
escalas logarítmicas



Taxa de Crescimento de $T(n)$

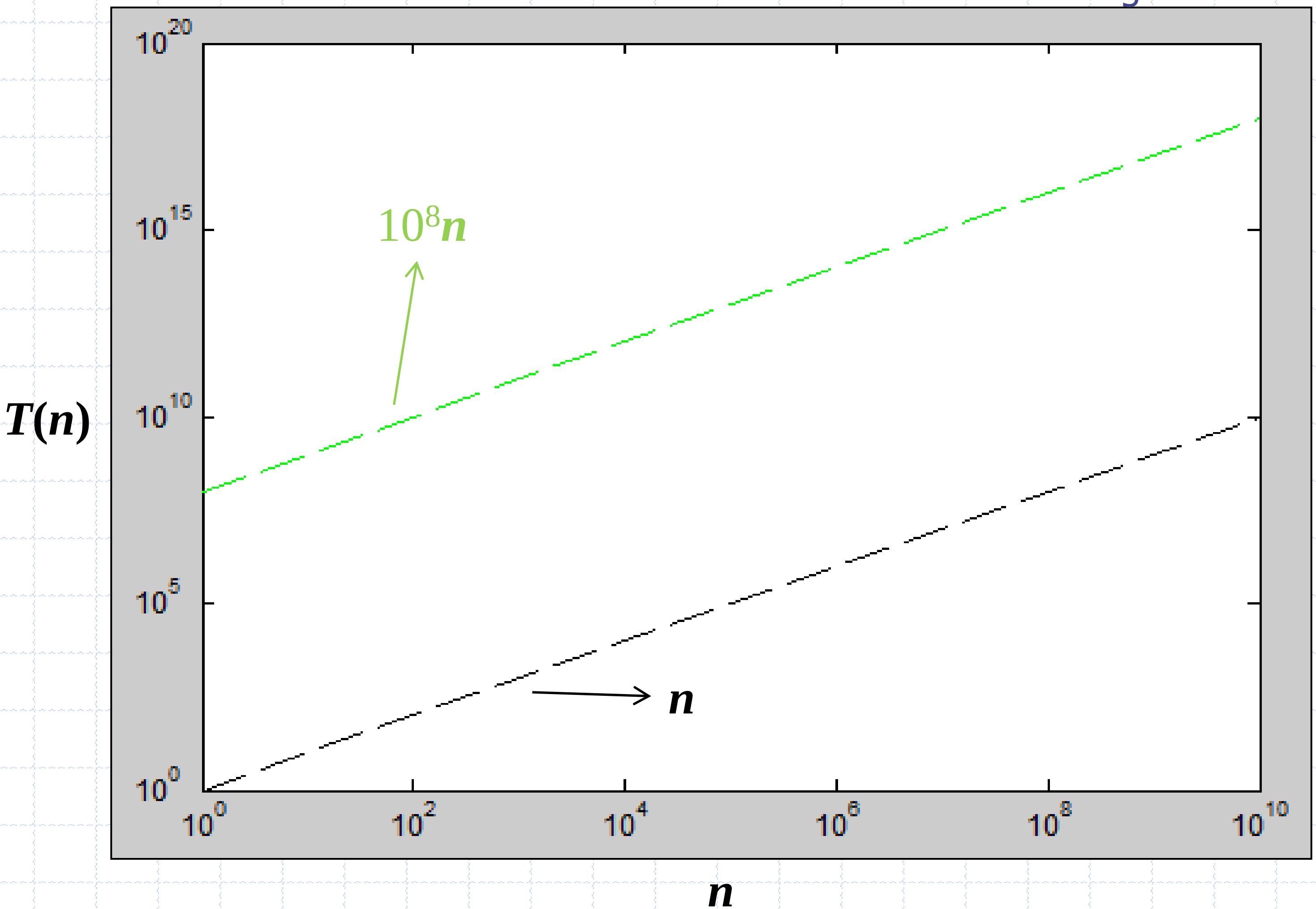
- Mudar o ambiente de hardware/ software:
  - Afeta $T(n)$ por um fator **constante**, mas
  - Não altera a **taxa de crescimento** de $T(n)$.

- As taxas de crescimento linear e logarítmica de $T(n)$ são **propriedades intrínsecas** dos algoritmos ***arrayMax*** e ***BB***, respectivamente.



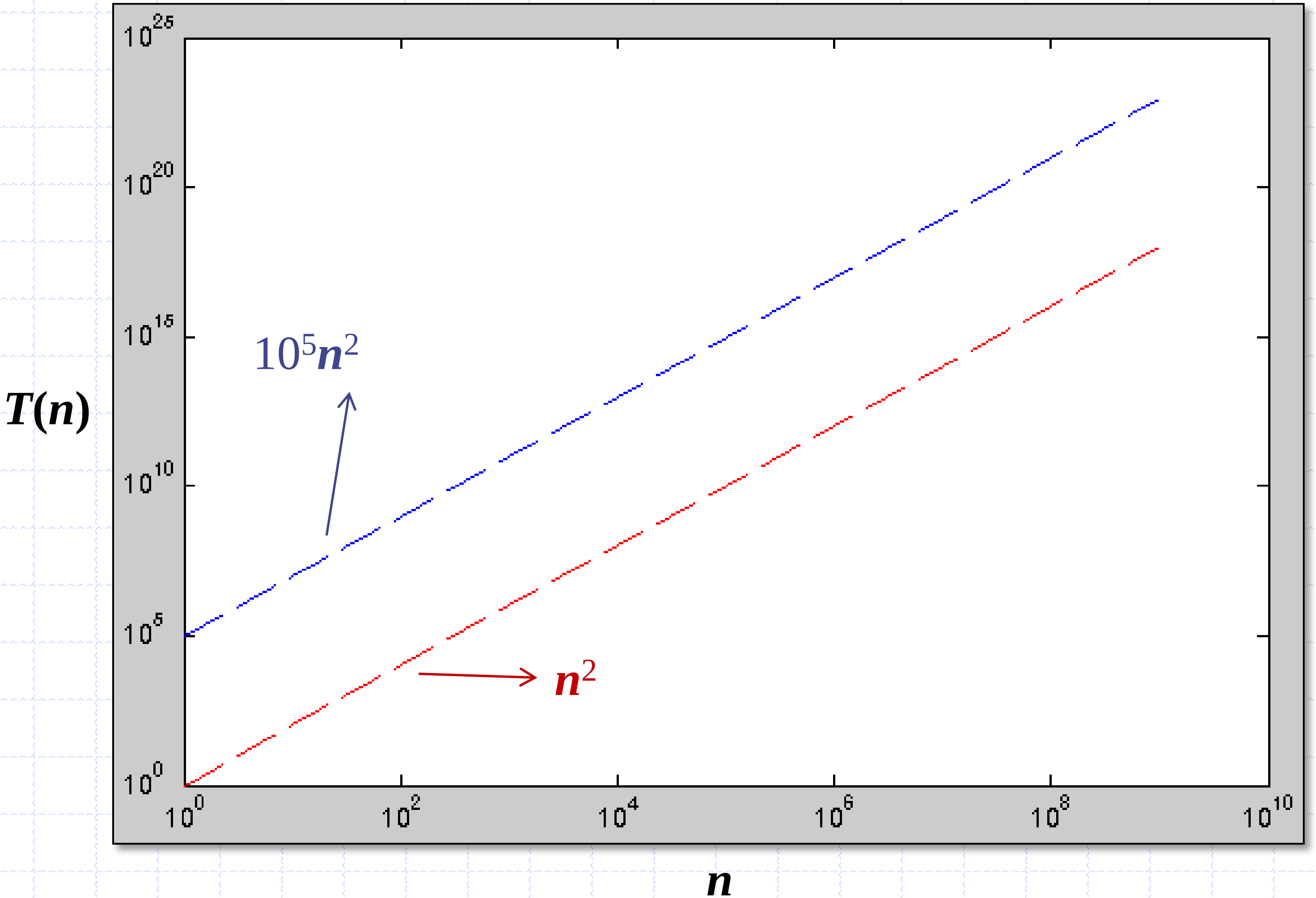
Fatores Constantes

escalas logarítmicas



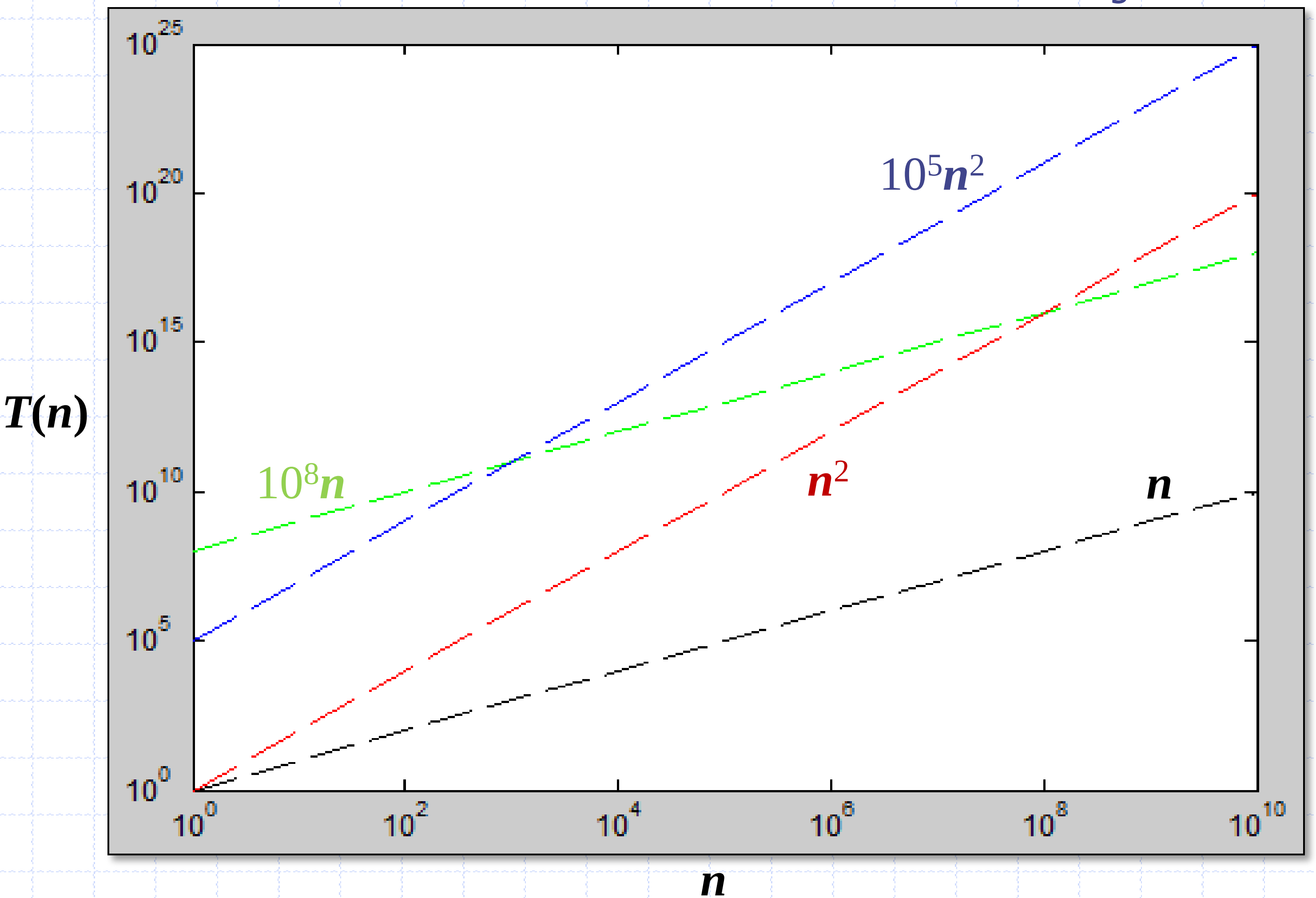
Fatores Constantes

escalas logarítmicas



Fatores Constantes

escalas logarítmicas



Termos de Menor Ordem

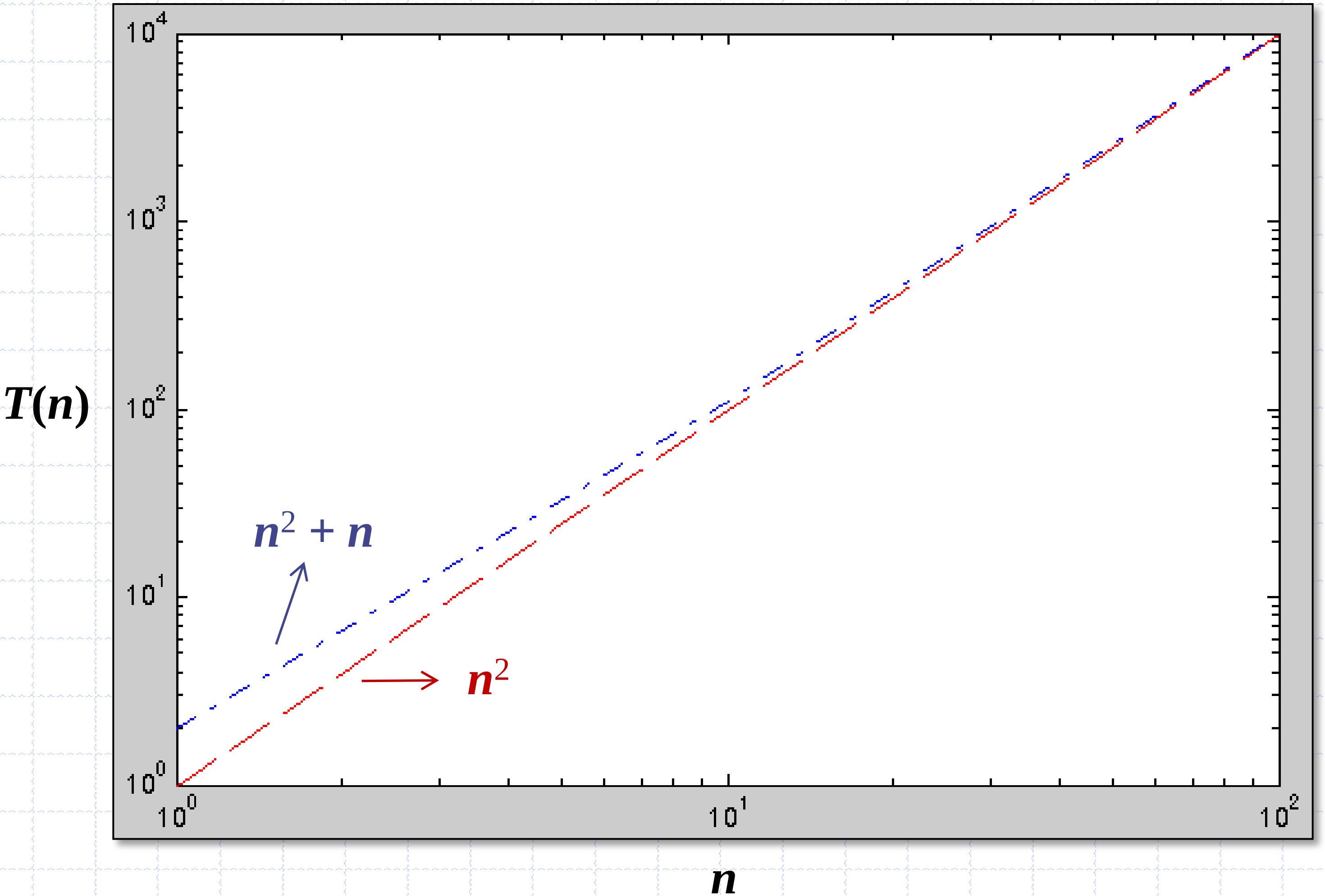
- Assim como fatores constantes não afetam a taxa de crescimento, termos de menor ordem também tendem a não mais afetá-la conforme o tamanho da entrada ***n*** cresce
- Exemplo:

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%

- Incremento dado por um termo de menor ordem aumenta em termos absolutos, mas diminui em termos relativos

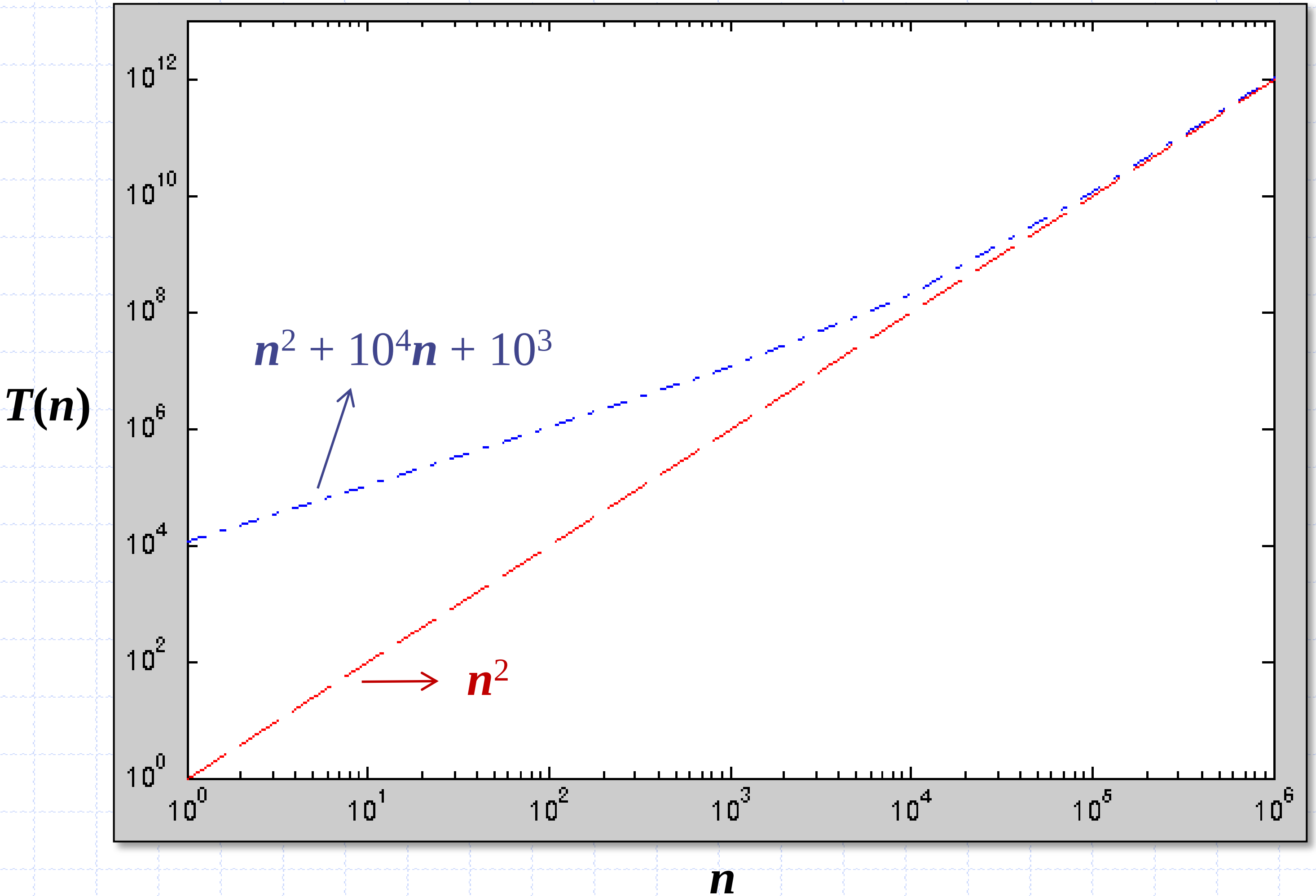
Termos de Menor Ordem

escalas logarítmicas



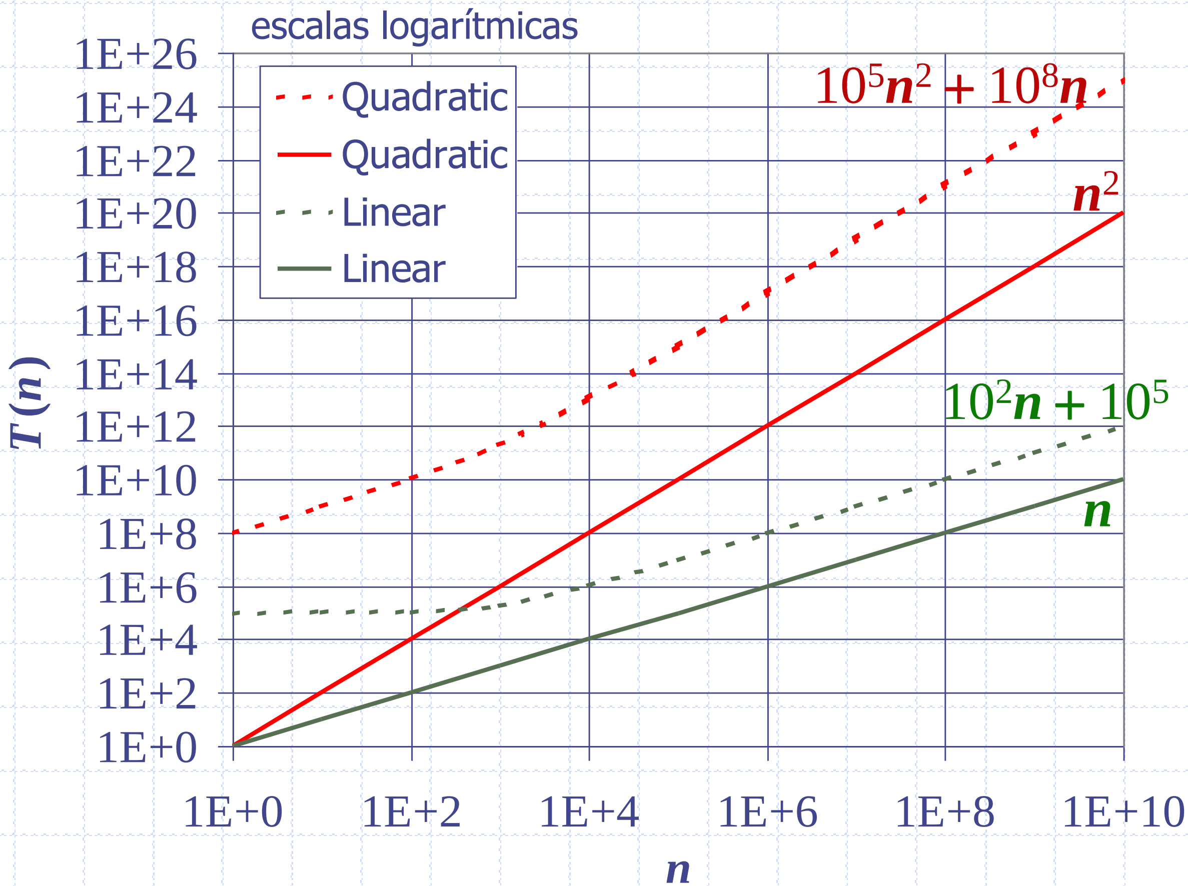
Termos de Menor Ordem

escalas logarítmicas



Em Resumo

- Assintoticamente, a taxa de crescimento é dominada pelo termo de maior ordem
- Exemplos:
  - $10^2 n + 10^5$ é uma função **linear** do tamanho da entrada n
  - $10^5 n^2 + 10^8 n$ é uma f. **quadrática** de n
- "Não importam" :
  - fatores constantes ou
  - termos de menor ordem



$T(n)$ *vs.* Tempo Cronológico

(ciano)	segundos
(bege)	minutos
(lilás)	séculos

Características Aproximadas do Hardware	
Número de Instruções executadas por Ciclo do relógio (IPC)	8
Frequência (1 / período do ciclo em min.)	3E+09
No. de Instruções por minuto	24E+09

$T(n)$	n = 20	n = 40	n = 60	n = 80
n	5,3E-08	1,1E-07	1,6E-07	2,1E-07
$n \log n$	2,3E-07	5,7E-07	9,5E-07	1,3E-06
n^2	1,1E-06	4,3E-06	9,6E-06	1,7E-05
n^3	2,1E-05	1,7E-04	5,8E-04	1,4E-03
2^n	2,8E-03	48,9	1,0	1,0E+06
3^n	0,2	5,4E+08	1,9E+18	6,6E+27

Exercícios

- Estime o número de operações primitivas da rotina abaixo para cálculo iterativo do **fatorial**:

```
double FAT(double X){
        double I, P;
        P = 1;
        for(I=1; I<=X; I++)
             P=P*I;
        return P;
}
```

- OBS: ignore a chamada da função e declaração das variáveis

Exercícios

- Estime o no. de operações primitivas do famoso algoritmo de **ordenação Bubble-Sort** (algoritmo da bolha):

Algoritmo ***Bubble***(***V***, ***n***)

Entrada: vetor ***V*** de ***n*** inteiros

Saída: vetor ***V*** ordenado em ordem crescente

para $j \leftarrow n - 1$ **até** 1 **faça**

  para $i \leftarrow 0$ **até** $j - 1$ **faça**

    se $V[i] > V[i+1]$ **então** /* troca V[i] com V[i+1] */

      $aux \leftarrow V[i]$

      $V[i] \leftarrow V[i+1]$

      $V[i+1] \leftarrow aux$

retorne ***V*** /* retorno de uma referência (ponteiro) para ***V*** */

- OBS: faça a estimativa para os números **mínimo** e **máximo**, e explique quais tipos de entrada (vetor V) levariam a esses extremos

Leitura e Exercícios

- Slides do Prof. João Luis Rosa
- Lista de Exercícios Extras

Vide Webpage da Disciplina!

Bibliografia

- N. Ziviani, *Projeto de Algoritmos*, Thomson, 2a. Edição, 2004
- M. T. Goodrich & R. Tamassia, *Data Structures and Algorithms in Java*, John Wiley & Sons, 3rd Edition, 2004
- T. H. Cormen et al., *Introduction to Algorithms*, MIT Press, 2nd Edition, 2001