

# **Algoritmos e Estruturas de Dados 1**

**Introdução à Análise de Algoritmos**  
(visão inicial, desempenho de tempo)

**Busca em Tabelas**  
(busca sequencial, busca binária)

# **Algoritmos e Estruturas de Dados 1**

**Introdução à Análise de Algoritmos**  
(visão inicial, desempenho de tempo)

# Analisar um Algoritmo

- Demonstrar que o algoritmo está correto
- Prever os recursos que ele utilizará
  - **Tempo de processamento**
  - Memória
  - Hardware
  - Largura de banda

# É possível

Analisar **um  
algoritmo específico**

Analisar **diferentes  
algoritmos para  
resolver um mesmo  
problema**, visando  
identificar a melhor  
opção de algoritmo

# Tempo de Processamento

- **$T(n)$**  – Função de Complexidade de Tempo, de um problema de tamanho  $n$ .
- Número de **Operações Críticas** do algoritmo **(\*)**.

**(\*)**

- exemplo: número de **comparações**;
- desconsiderar incremento de índice, por exemplo.

# Dependências

Tempo <b>não depende</b> da entrada	Tempo <b>depende</b> da entrada
<p data-bbox="202 835 879 978">Ex.: achar o menor elemento de um vetor</p> $T(n) = n-1$	<p data-bbox="1000 671 1850 813">Ex.: encontrar um elemento X em um vetor</p> <p data-bbox="1000 942 1574 1006">Busca Sequencial:</p> <ul data-bbox="1000 1021 1748 1335" style="list-style-type: none"><li>• <math>T(n)</math> - Pior caso: <math>n</math></li><li>• <math>T(n)</math> – melhor caso: <math>1</math></li><li>• <math>T(n)</math> – caso médio (difícil precisar)</li></ul>

# Análise Assintótica

- Estuda o comportamento de uma função  $f(n)$ , quando **n** tende ao **infinito**.

## Em outras palavras:

- Procuramos analisar a complexidade de um algoritmo a medida que o tamanho da entrada de dados ( $n$ ) aumenta muito;
- Para pequeno volume de dados, a diferença entre os algoritmos não será significativa.

# Notação O

## Limite assintótico superior

### Exemplo: $T(n) = O(n^2)$

- Limite assintótico superior do tempo de execução de um algoritmo, para um problema de tamanho  $n$  é  $O(n^2)$ ;
- Para qualquer  $n$ , o tempo de execução tem um limite superior determinado pelo valor  $c * n^2$ ;
- Ordem de grandeza; proporcionalidade;
- “Da ordem de”.

**Obs.:** Para algoritmo  $A$  composto por 2 trechos  $A1$  e  $A2$ , podemos definir  $O(A)$  como o maior dentre  $O(A1)$  e  $O(A2)$ .



# Algoritmos e Estruturas de Dados 1

## Busca em Tabelas

(busca sequencial, busca binária)

# Busca Sequencial

80	20	72	29	34	12	56	99	40	17
----	----	----	----	----	----	----	----	----	----



**Encontrar um elemento  $x$  em um vetor de tamanho  $n$**

# Busca Sequencial

80	20	72	29	34	12	56	99	40	17
----	----	----	----	----	----	----	----	----	----



# Busca Sequencial

80	20	72	29	34	12	56	99	40	17
----	----	----	----	----	----	----	----	----	----



# Busca Sequencial

80	20	72	29	34	12	56	99	40	17
----	----	----	----	----	----	----	----	----	----



# Busca Sequencial

80	20	72	29	34	12	56	99	40	17
----	----	----	----	----	----	----	----	----	----



```
int SequentialSearch (int v[], int n, int x) {  
    int i = 0;  
    while ((i < n) && (v[i] != x))  
        i++;  
    if (i < n)  
        return i;  
    else return -1;  
}
```

# Busca Sequencial

80	20	72	29	34	12	56	99	40	17
----	----	----	----	----	----	----	----	----	----



```
int SequentialSearch (int v[], int n, int x) {  
    int i = 0;  
    while ((i < n) && (v[i] != x))  
        i++;  
    if (i < n)  
        return i;  
    else return -1;  
}
```

## Tempo:

- Melhor Caso: 1
- **Pior caso:  $O(n)$**

## Espaço:

- $O(1)$

# Busca Sequencial

10	20	22	29	34	47	56	70	80	99
----	----	----	----	----	----	----	----	----	----



**vetor ordenado**

```
int SequentialSearch (int v[], int n, int x) {  
    int i = 0;  
    while ((i < n) && (v[i] < x))  
        i++;  
    if (i < n)  
        return i;  
    else return -1;  
}
```

## **Tempo:**

- Melhor Caso: 1
- **Pior caso:  $O(n)$**
- Caso médio melhora

## **Espaço:**

- $O(1)$



# Busca Sequencial

10	20	22	29	34	47	56	70	80	99
----	----	----	----	----	----	----	----	----	----



**vetor ordenado**

```
int SequentialSearch (int v[], int n, int x) {  
    int i = 0;  
    v[n] = x;  
    while ((v[i] < x)  
           i++;  
    if (i < n)  
        return i;  
    else return -1;  
}
```

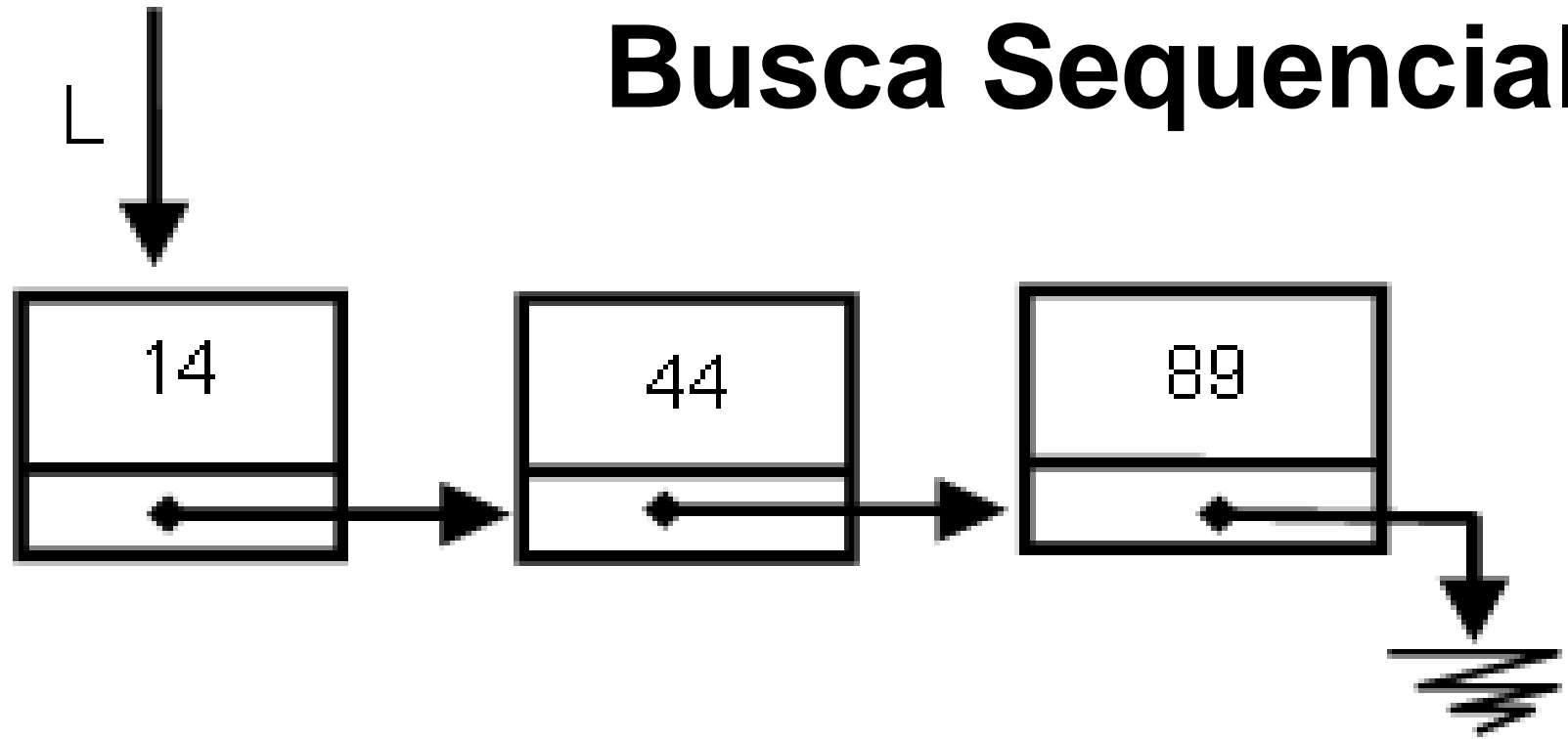
## Tempo:

- Melhor Caso: 1
- **Pior caso:  $O(n)$**
- Constante melhora

## Espaço:

- $O(1)$

# Busca Sequencial



Exercício 1: Busca Sequencial em Lista Encadeada Não Ordenada; algoritmo e análise;

Exercício 2: Busca Sequencial em Lista Encadeada Ordenada; algoritmo e análise;

Exercício 3: Busca Sequencial em Lista Encadeada Ordenada com header; algoritmo e análise.

# Achar X em **Árvore Binária de Busca**

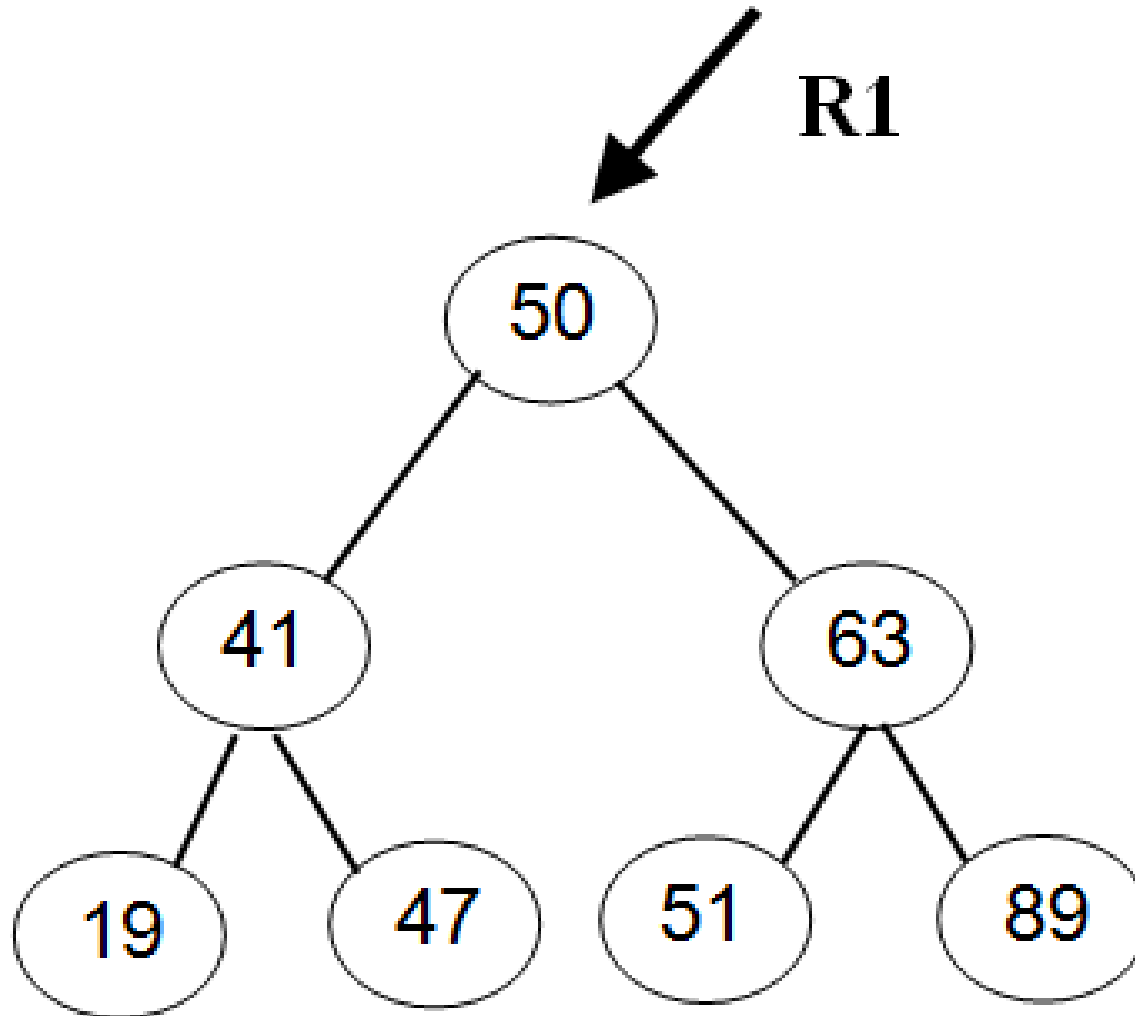


ABB nniformemente distribuída

# Está na Árvore?

Boolean **EstaNaArvore** (parâmetro por referência **R** do tipo ABB, parâmetro **X** do tipo Inteiro) {

Se (R == Null)

Então Retorne Falso; // **Caso 1:** Árvore vazia; X não está na Árvore; acabou

Senão Se (X == R→Info)

Então Retorne Verdadeiro; // **Caso 2:** X está na árvore; acabou

Senão Se (R→Info > X)

Então Retorne ( Está\_Na\_Árvore (R→**Esq**, X ) );

// **Caso 3:** se estiver na Árvore, estará na Sub Esquerda

Senão Retorne ( Está\_Na\_Árvore (R→**Dir**, X ) );

// **Caso 4:** se estiver na Árvore, estará na Sub Direita

} // fim EstáNaÁrvore

# Achar X em **Árvore Binária de Busca**

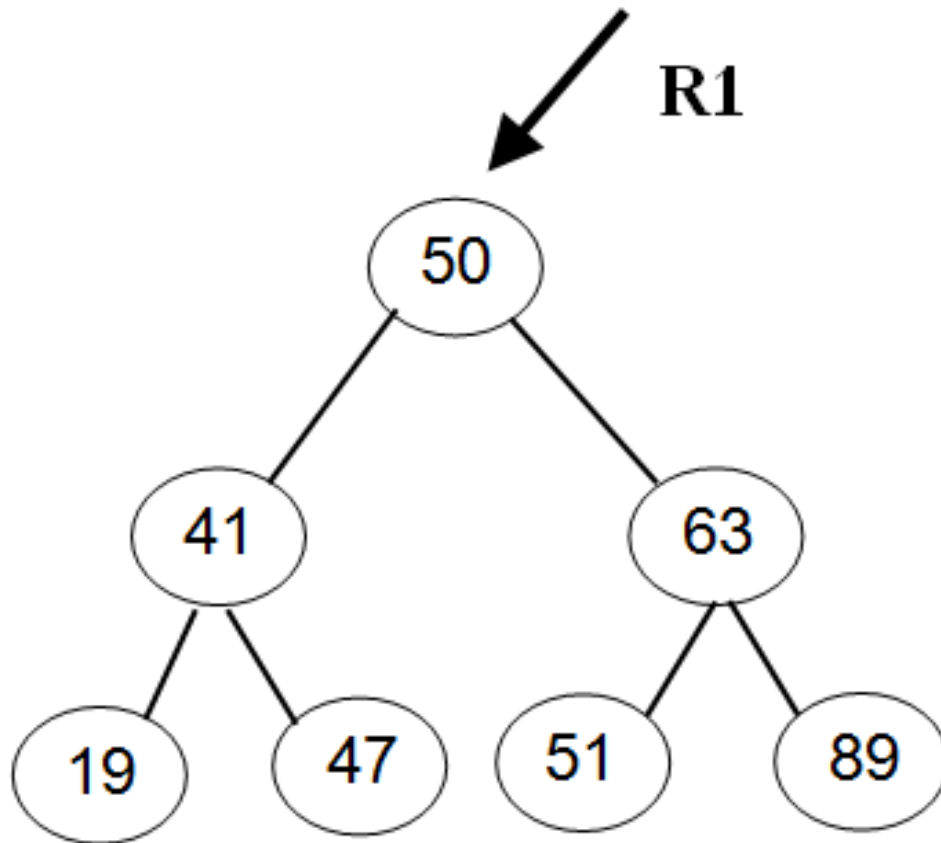


ABB uniformemente distribuída

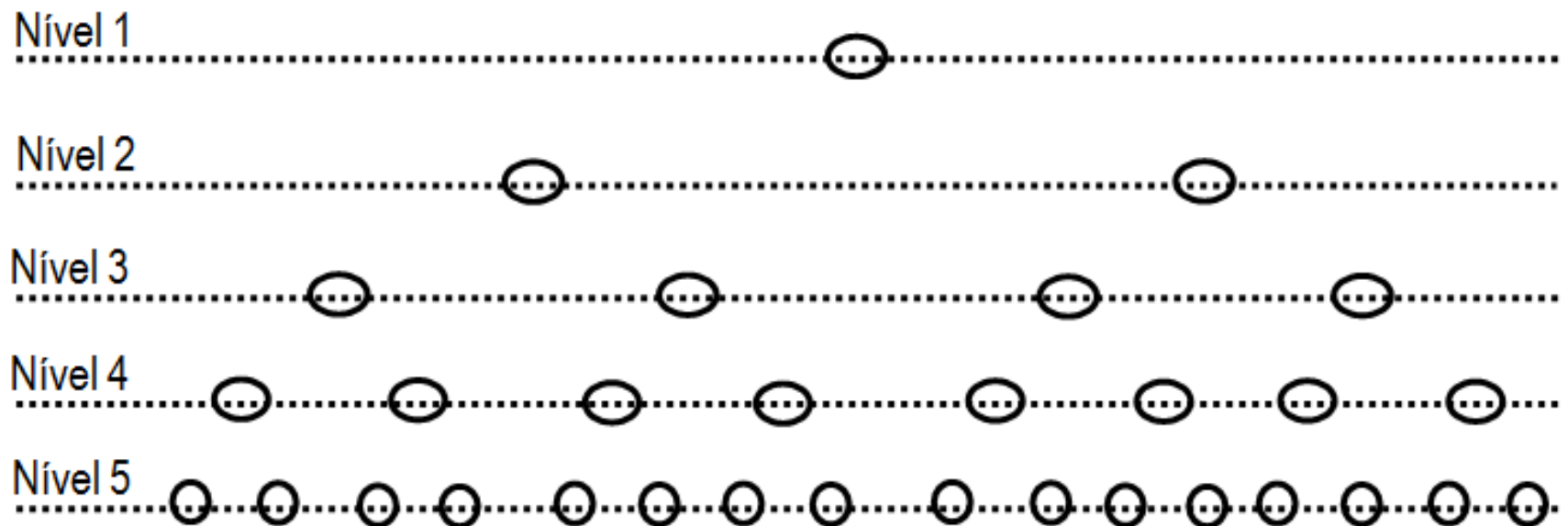
**Tempo:**

- Melhor Caso: 1
- **Pior caso:**

**Espaço:**

- $O(1)$

# Por Que uma Árvore Binária de Busca É Boa?



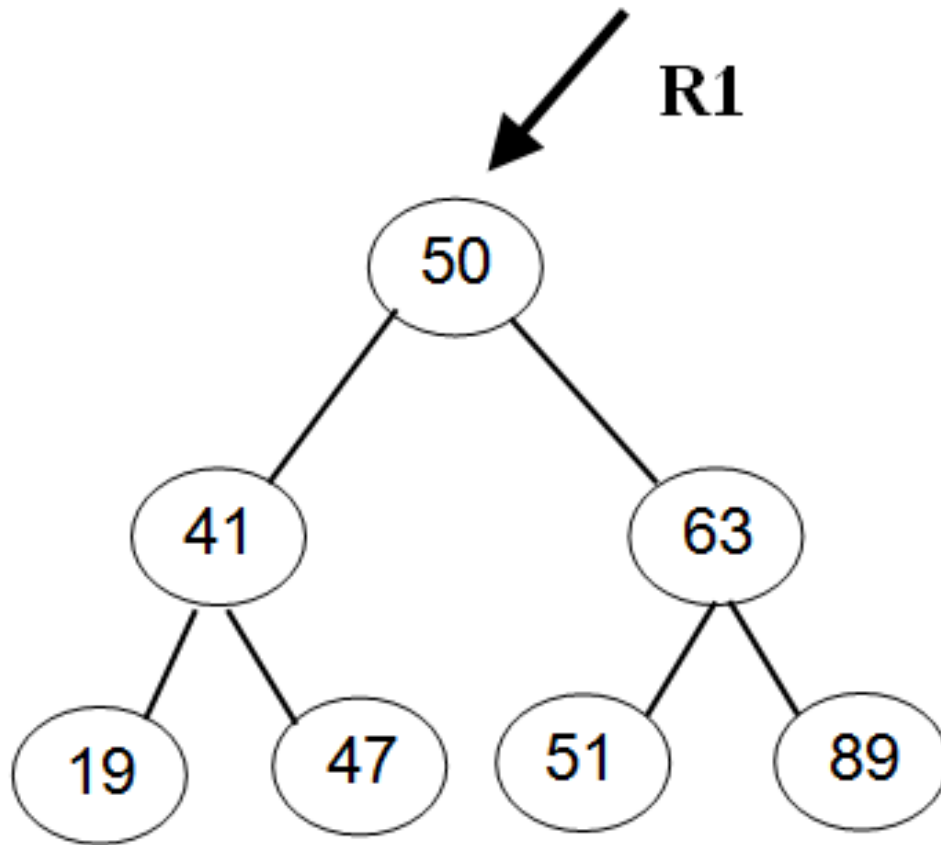
**ABB Uniformemente Distribuída**

**Por Que  
uma  
Árvore  
Binária de  
Busca É  
Boa?**

<b>Níveis na Árvore</b>	<b>Quantos Nós Cabem na Árvore</b>
1	1
2	3
3	7
4	15
5	31
<b>N</b>	<b><math>2^N - 1</math></b>
10	1023
13	8191
16	65535
18	262143
20	1 milhão (aprox)
30	1 bilhão (aprox)
40	1 trilhão (aprox)

**ABB Uniformemente Distribuída**

# Achar X em **Árvore Binária de Busca**



**Tempo:**

- Melhor Caso: 1
- **Pior caso:  $O(\log n)$**

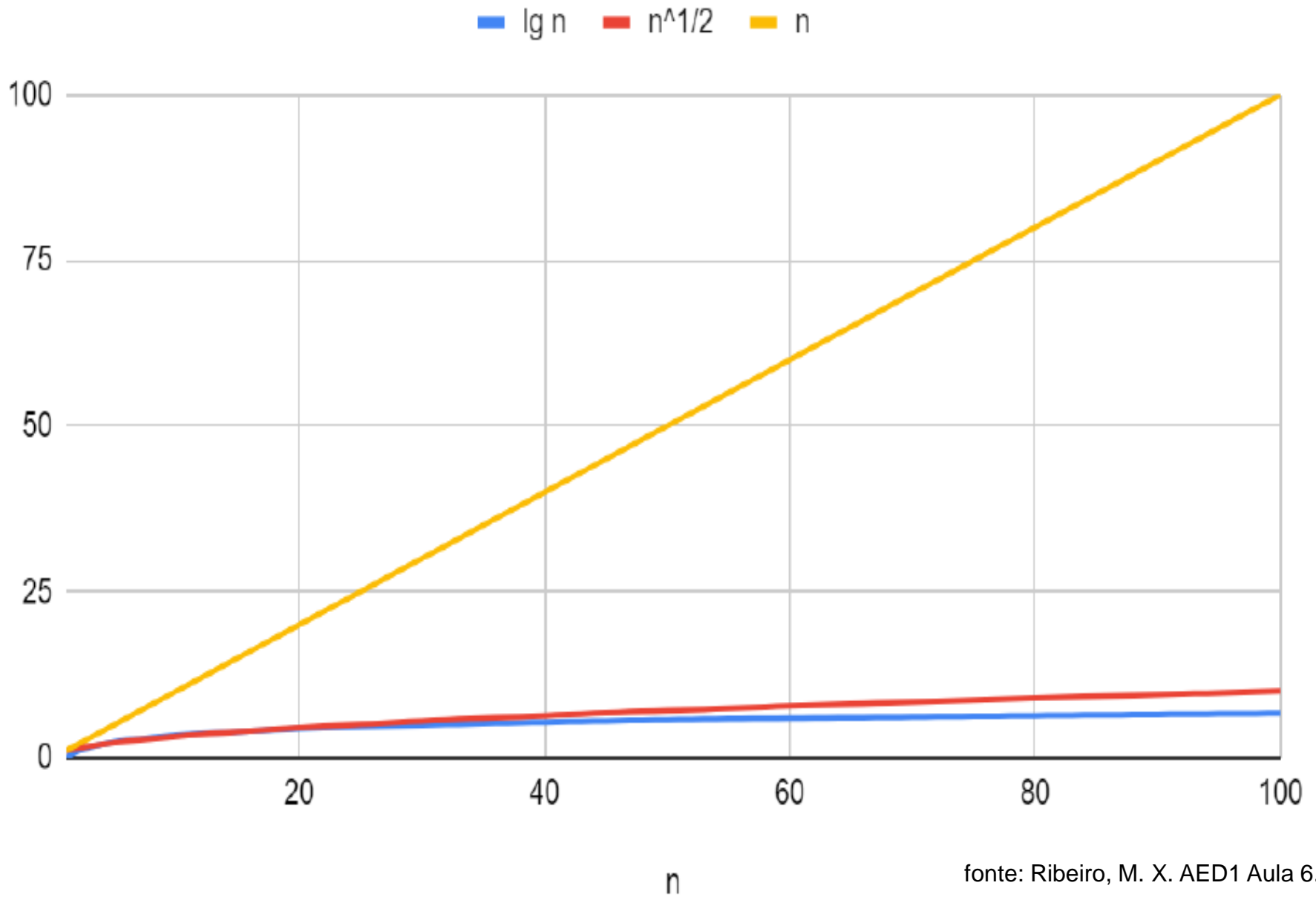
**Espaço:**

- $O(1)$

ABB uniformemente distribuída

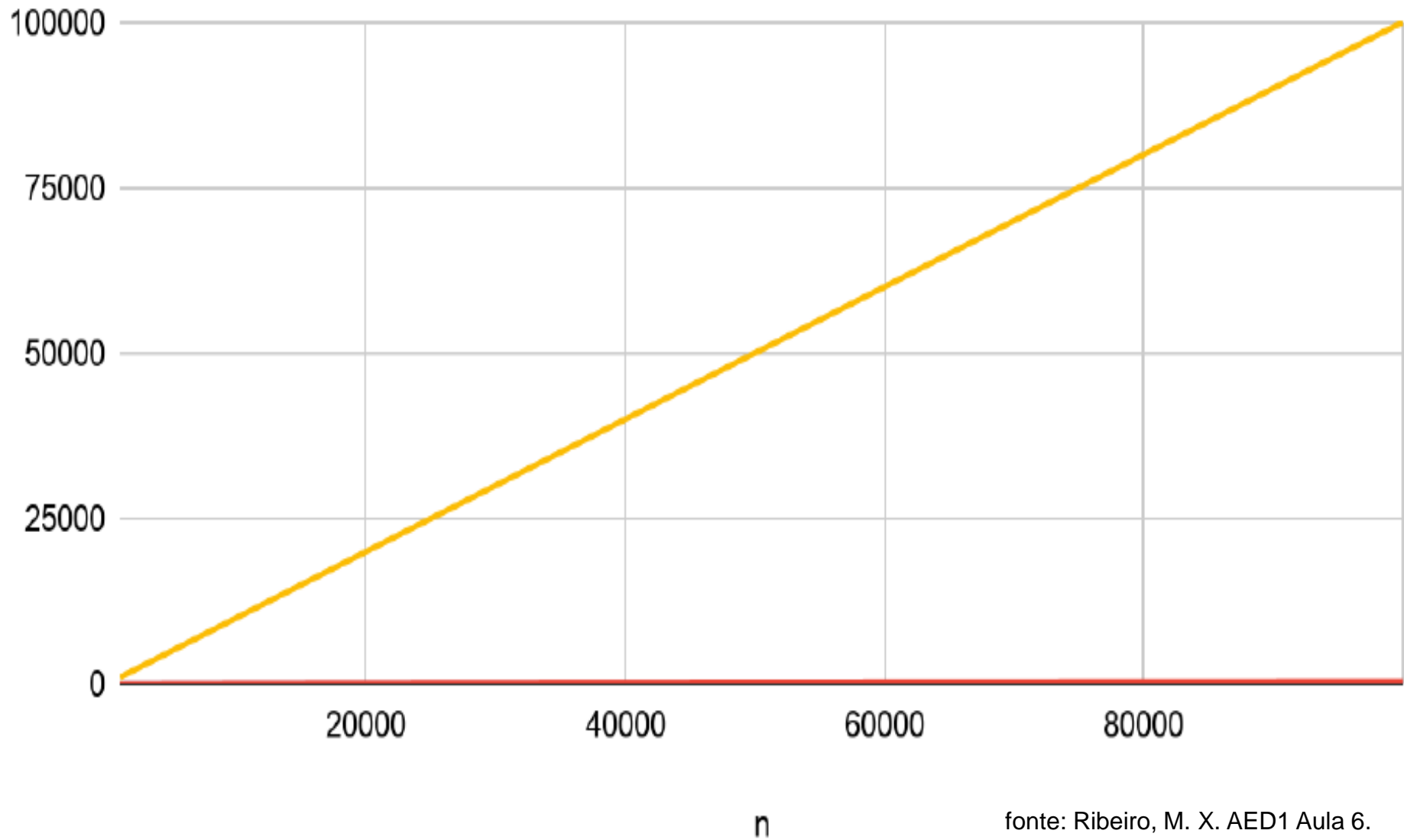


$\lg n$ ,  $n^{1/2}$  e  $n$



$\lg n$ ,  $n^{1/2}$  e  $n$

$\lg n$   $n^{1/2}$   $n$



## Crescimento de funções

n	$10^3$	$10^6$	$10^9$
$\log_2 n$	10	20	30
$n^{1/2}$	32	$10^3$	$3 \cdot 10^4$
n	$10^3$	$10^6$	$10^9$
$n \log_2 n$	$10^4$	$2 \cdot 10^7$	$3 \cdot 10^{10}$
$n^2$	$10^6$	$10^{12}$	$10^{18}$
$n^3$	$10^9$	$10^{18}$	$10^{27}$
$2^n$	$10^{300}$	$10^{300000}$	$10^{(3 \cdot 10^8)}$

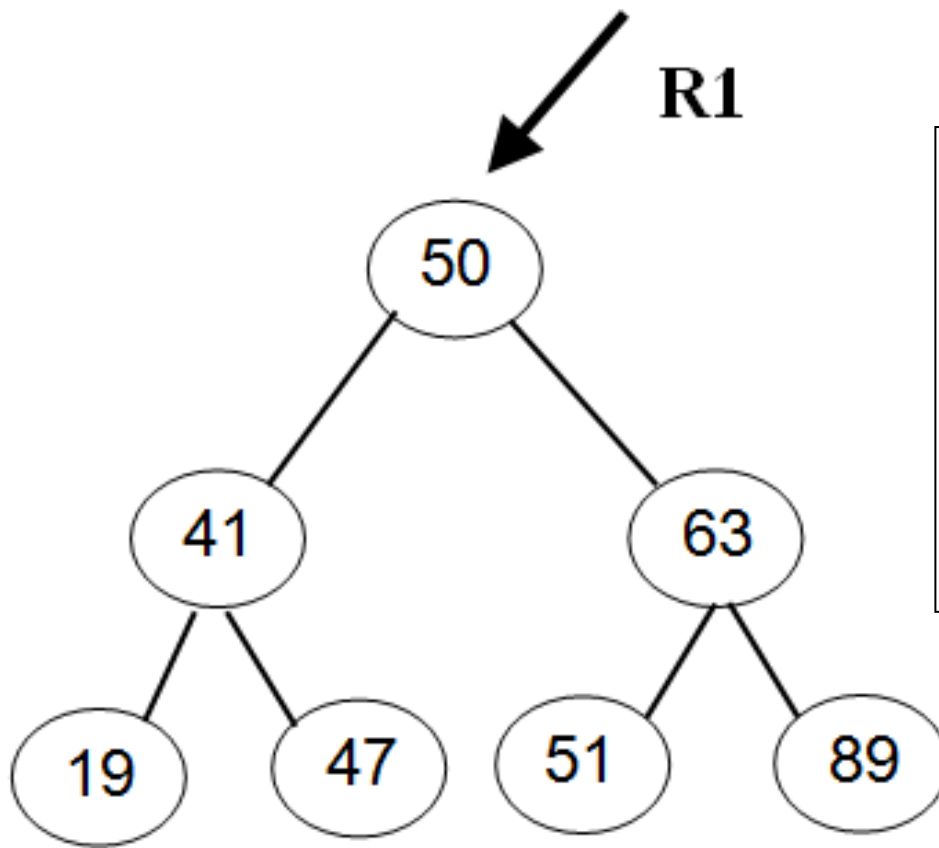
Interpretação temporal considerando um computador de 1GHz

n	$10^3$	$10^6$	$10^9$
$\log_2 n$	$\ll 1s$	$\ll 1s$	$\ll 1s$
$n^{1/2}$	$\ll 1s$	$\ll 1s$	$\ll 1s$
n	$\ll 1s$	$\ll 1s$	1s
$n \log_2 n$	$\ll 1s$	$< 1s$	30s
$n^2$	$\ll 1s$	16 min	31 anos
$n^3$	1s	31 anos	31709791 milênios
$2^n$	esquece...		

# Busca Binária

10	20	22	29	34	47	56	70	80	99
----	----	----	----	----	----	----	----	----	----

**vetor ordenado**



**Tempo:**

- Melhor Caso: 1
- **Pior caso:  $O(\log n)$**

**Espaço:**

- $O(1)$

# Busca Binária

**X = 10**

10	20	22	29	34	47	56	70	80	99
0	1	2	3	4	5	6	7	8	9



# Busca Binária

**X = 10**

10	20	22	29	34	47	56	70	80	99
0	1	2	3	4	5	6	7	8	9

10	20	22	29						
0	1	2	3	4	5	6	7	8	9



# Busca Binária

**X = 10**

10	20	22	29	34	47	56	70	80	99
0	1	2	3	4	5	6	7	8	9

10	20	22	29						
0	1	2	3	4	5	6	7	8	9


10									
0	1	2	3	4	5	6	7	8	9





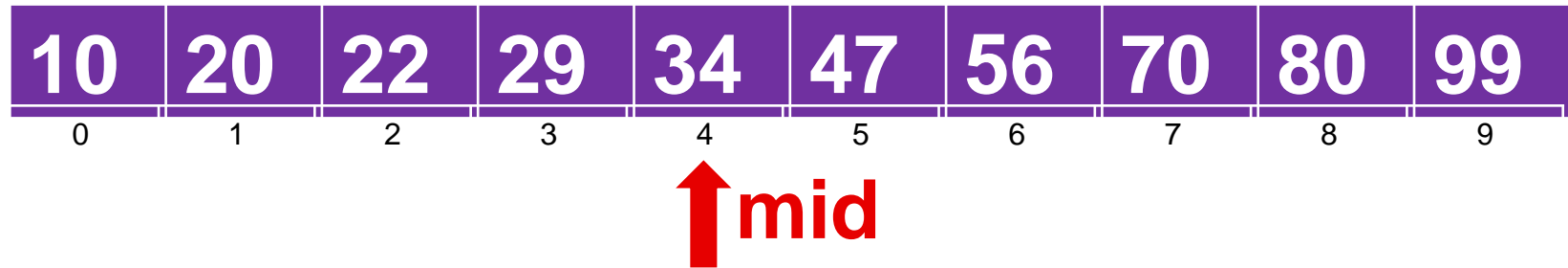
# Busca Binária

10	20	22	29	34	47	56	70	80	99
0	1	2	3	4	5	6	7	8	9

 **mid**

```
int _BinarySearch (int v[], int right, int left, int x) {  
    if (left > right)  
        return -1;  
    int mid = (left + right) / 2;  
    if (v[mid] == x)  
        return mid;  
    if (x > v[mid])  
        return (_BinarySearch(v, mid+1, right, x));  
    else return (_BinarySearch(v, left, mid-1, x));  
}
```

# Busca Binária



```
int BinarySearch (int v[], int n, int x) {  
    return (_BinarySearch(v, 0, n-1, x));  
}
```

## Tempo:

- Melhor Caso: 1
- **Pior caso:  $O(\log n)$**

## Espaço:

- $O(1)$

# Exercícios:

Exercício 4: Busca Binária Iterativo (não recursivo).  
Algoritmo e análise de desempenho.