Complexidade Assintótica de Programas

Letícia Rodrigues Bueno

Análise de Algoritmos

1. **Introdução**;

Análise de Algoritmos

1. **Introdução**;
2. **Conceitos básicos**;

Análise de Algoritmos

1. **Introdução**;
2. **Conceitos básicos**;
3. **Notação assintótica: notação** O;

Análise de Algoritmos

1. **Introdução**;
2. **Conceitos básicos**;
3. **Notação assintótica: notação** O;
4. Notação assintótica: notações Ω, Θ, o e ω;
5. Métodos de análise de algoritmos;
6. Teoria da complexidade.

Introdução

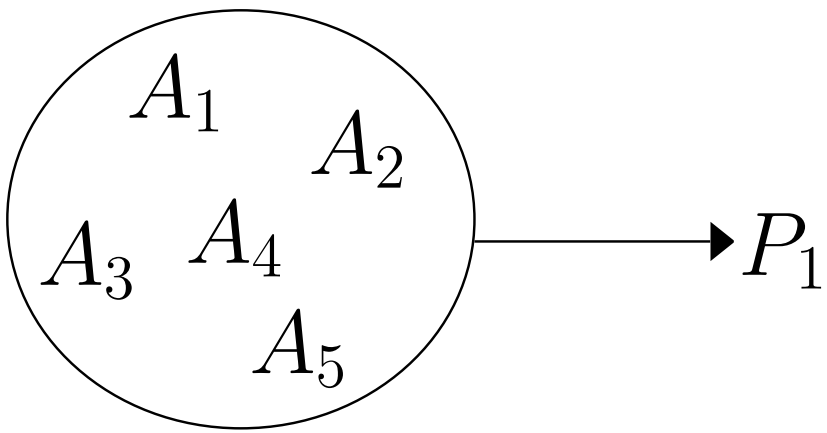
- **Objetivo**: possibilitar medir eficiência de algoritmos;

Introdução

- **Objetivo**: possibilitar medir eficiência de algoritmos;
- **Analisar um algoritmo**: prever os recursos que serão necessários;

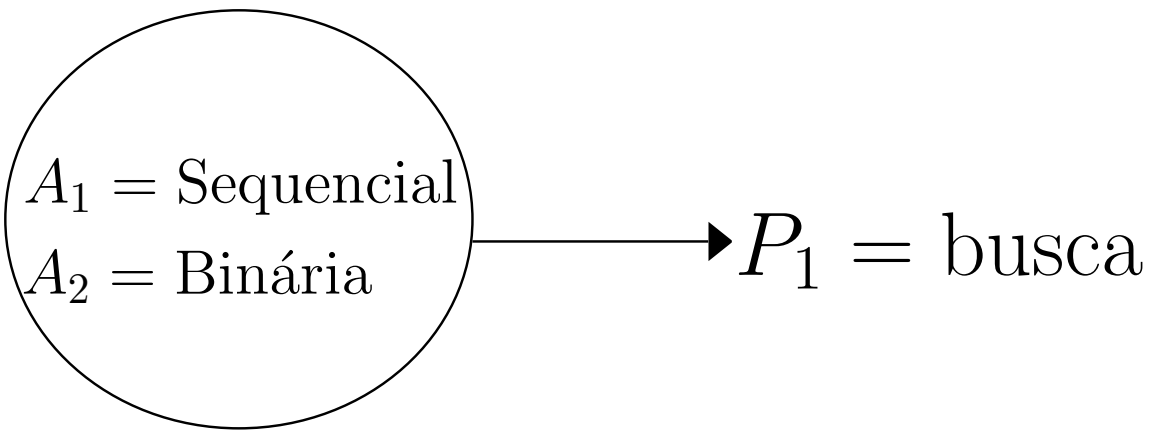
Introdução

- **Objetivo**: possibilitar medir eficiência de algoritmos;
- **Analisar um algoritmo**: prever os recursos que serão necessários;



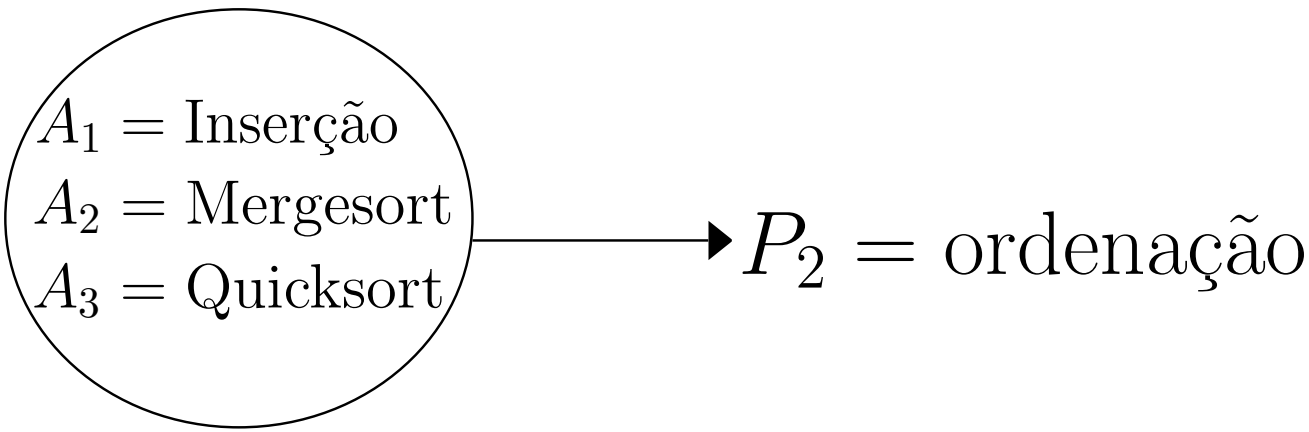
Introdução

- **Objetivo**: possibilitar medir eficiência de algoritmos;
- **Analisar um algoritmo**: prever os recursos que serão necessários;
- **Busca**: Sequencial ou Binária?



Introdução

- **Objetivo**: possibilitar medir eficiência de algoritmos;
- **Analisar um algoritmo**: prever os recursos que serão necessários;
- **Busca**: Sequencial ou Binária?
- **Ordenação**: Inserção, Seleção, Bolha, *Quicksort*, *Heapsort*, *Mergesort* ou *Shellsort*?



Conceitos Básicos: definições

- **modelo assumido**: instruções executadas uma após a outra, sem operações concorrentes (ou simultâneas);

Conceitos Básicos: definições

- **modelo assumido**: instruções executadas uma após a outra, sem operações concorrentes (ou simultâneas);
- **problema com entrada de tamanho** n: função de custo ou função de complexidade $f(n)$;

Conceitos Básicos: definições

- **modelo assumido**: instruções executadas uma após a outra, sem operações concorrentes (ou simultâneas);
- **problema com entrada de tamanho** n: função de custo ou função de complexidade $f(n)$;
- **função de complexidade de tempo**: número de execuções de determinada operação considerada relevante;

Conceitos Básicos: definições

- **modelo assumido**: instruções executadas uma após a outra, sem operações concorrentes (ou simultâneas);
- **problema com entrada de tamanho** n: função de custo ou função de complexidade $f(n)$;
- **função de complexidade de tempo**: número de execuções de determinada operação considerada relevante;
- **função de complexidade de espaço**: espaço de memória ocupada;

Conceitos Básicos: exemplo

Exemplo: calcular o fatorial de um número n.
Tamanho da entrada do problema: n;

```
     fatorial(n)
1:   fat ← 1;
2:   para (i ← 1; i ≤ n; i++) faça
3:        fat ← fat * i;
4:   retorna fat
```

Conceitos Básicos: fatorial

Exemplo: calcular o fatorial de um número n.

```
     fatorial(n)
1:   fat ← 1;
2:   para (i ← 1; i ≤ n; i++) faça    % executa n vezes
3:        fat ← fat * i;              % tempo constante c1
4:   retorna fat
```

Complexidade de tempo: $c_1.n$;

Conceitos Básicos: melhor caso, pior caso e caso médio

Três cenários dependentes da entrada:

- **Melhor caso**: menor tempo de execução;

Conceitos Básicos: melhor caso, pior caso e caso médio

Três cenários dependentes da entrada:

- **Melhor caso**: menor tempo de execução;
- **Pior caso**: maior tempo de execução. Geralmente, priorizamos determinar o pior caso;

Conceitos Básicos: melhor caso, pior caso e caso médio

Três cenários dependentes da entrada:

- **Melhor caso**: menor tempo de execução;
- **Pior caso**: maior tempo de execução. Geralmente, priorizamos determinar o pior caso;
- **Caso médio**: média dos tempos de execução. Mais difícil de obter;

Conceitos Básicos: melhor caso, pior caso e caso médio

Exemplo: busca sequencial.
Operação relevante: comparação de x com elementos de V;

```
     buscaSequencial(x, V)
1:   i ← 1;
2:   enquanto (i ≤ n) e (V[i] ≠ x) faça      % executa n vezes no máximo
3:        i ← i + 1;
4:   se i > n então "Busca sem sucesso"
5:        senão "Busca com sucesso"
```

Conceitos Básicos: melhor caso

Melhor caso da busca sequencial: x está em $V[1]$!

```
     buscaSequencial(x, V)
1:   i ← 1;
2:   enquanto (i ≤ n) e (V[i] ≠ x) faça      % executa n vezes no máximo
3:        i ← i + 1;
4:   se i > n então "Busca sem sucesso"
5:             senão "Busca com sucesso"
```

Complexidade de tempo: 1

Conceitos Básicos: pior caso

Pior caso da busca sequencial: x está em $V[n]$ ou não está em V!

```
      buscaSequencial(x, V)
1:    i ← 1;
2:    enquanto (i ≤ n) e (V[i] ≠ x) faça      % executa n vezes no máximo
3:          i ← i + 1;
4:    se i > n então "Busca sem sucesso"
5:          senão "Busca com sucesso"
```

Complexidade de tempo: n

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1$

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1 + 2$

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1 + 2 + 3$

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1 + 2 + 3 + \ldots + n)$

Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1 + 2 + 3 + \ldots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right)$

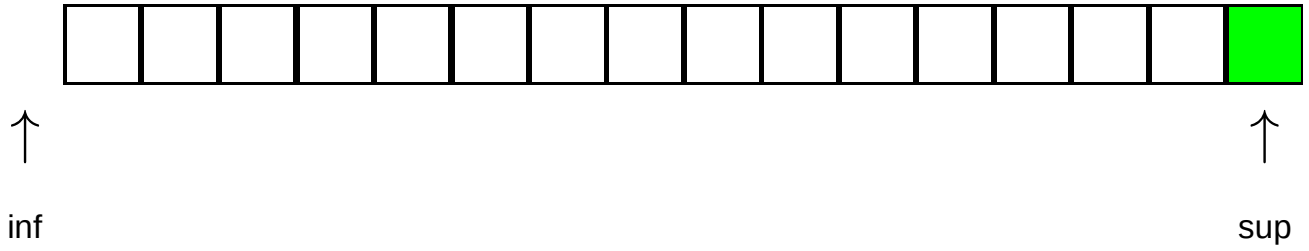
Conceitos Básicos: caso médio

- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1 + 2 + 3 + \ldots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$

Conceitos Básicos: caso médio

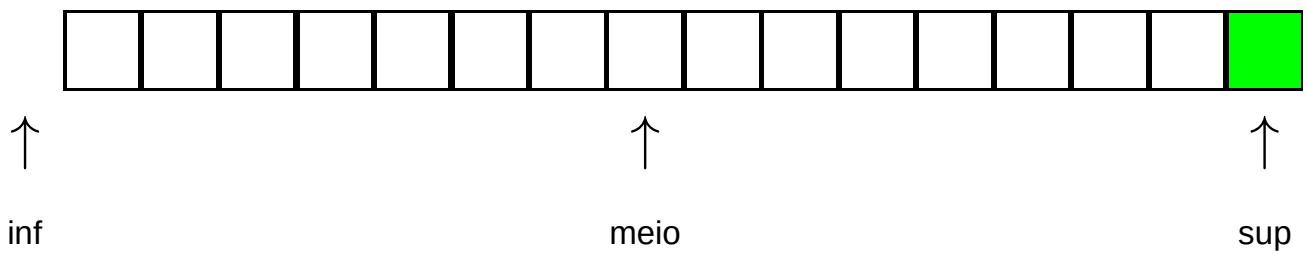
- **Caso médio da busca sequencial:** assumindo que x está em V, $f(n) = 1 \times p_1 + 2 \times p_2 + \ldots + n \times p_n$ onde p_i é probabilidade de x estar na posição i;
- **probabilidades são iguais**: $p_i = \frac{1}{n}$;
- $f(n) = \frac{1}{n}(1 + 2 + 3 + \ldots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$
- **Complexidade de tempo:** $\frac{n+1}{2}$, ou seja, uma pesquisa bem-sucedida examina aproximadamente metade dos registros.

```
1:  BuscaBinária(chave, lista[0...n])
2:  inf ← −1
3:  sup ← n
4:  enquanto inf < sup − 1 faça
5:      meio ← ⌊(inf+sup)/2⌋
6:      se chave ≤ lista[meio] então
7:          sup ← meio
8:      senão
9:          inf ← meio
10: se chave = lista[sup] então
11:     retorna lista[sup]
12: senão
13:     retorna elemento não encontrado
```



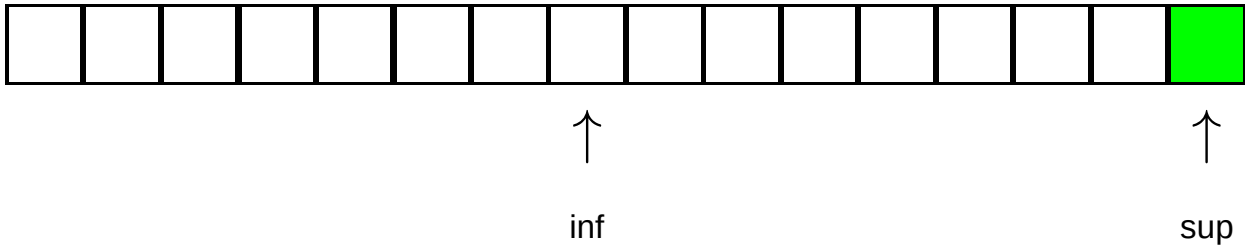
inicializ.

```
1:  BuscaBinária(chave, lista[0...n])
2:  inf ← −1
3:  sup ← n
4:  enquanto inf < sup − 1 faça
5:      meio ← ⌊(inf+sup)/2⌋
6:      se chave ≤ lista[meio] então
7:          sup ← meio
8:      senão
9:          inf ← meio
10: se chave = lista[sup] então
11:     retorna lista[sup]
12: senão
13:     retorna elemento não encontrado
```



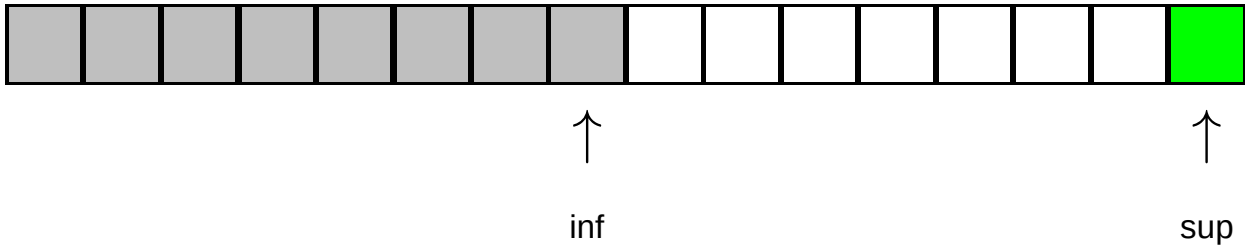
1a iter.

```
1:  BuscaBinária(chave, lista[0...n])
2:  inf ← −1
3:  sup ← n
4:  enquanto inf < sup − 1 faça
5:      meio ← ⌊(inf + sup)/2⌋
6:      se chave ≤ lista[meio] então
7:          sup ← meio
8:      senão
9:          inf ← meio
10: se chave = lista[sup] então
11:     retorna lista[sup]
12: senão
13:     retorna elemento não encontrado
```



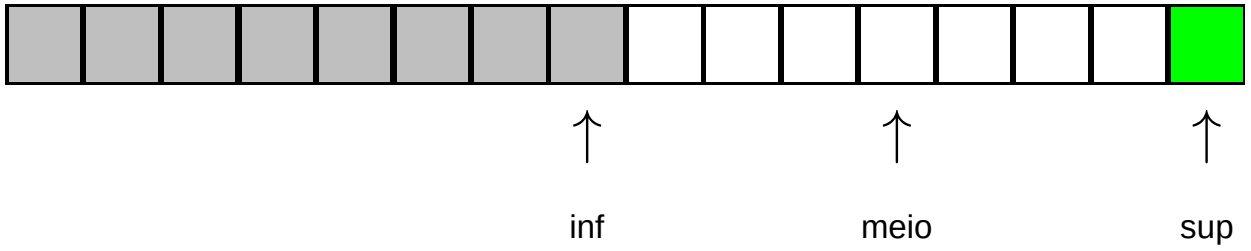
1a iter.

```
1:  BuscaBinária(chave, lista[0...n])
2:  inf ← −1
3:  sup ← n
4:  enquanto inf < sup − 1 faça
5:      meio ← ⌊(inf+sup)/2⌋
6:      se chave ≤ lista[meio] então
7:          sup ← meio
8:      senão
9:          inf ← meio
10: se chave = lista[sup] então
11:     retorna lista[sup]
12: senão
13:     retorna elemento não encontrado
```



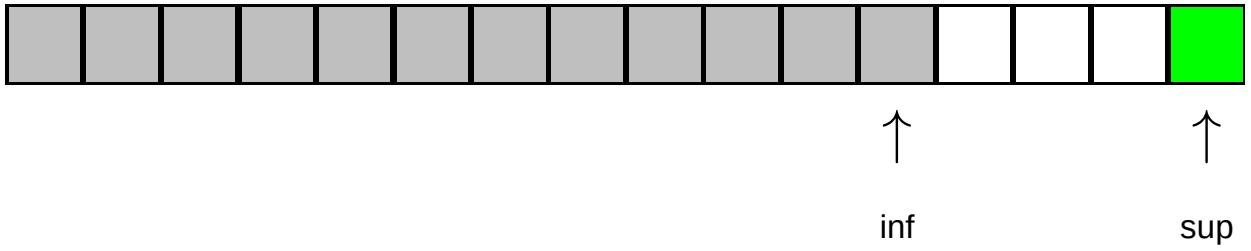
2a iter.

```
1:  BuscaBinária(chave, lista[0...n])
2:  inf ← −1
3:  sup ← n
4:  enquanto inf < sup − 1 faça
5:      meio ← ⌊(inf + sup)/2⌋
6:      se chave ≤ lista[meio] então
7:          sup ← meio
8:      senão
9:          inf ← meio
10: se chave = lista[sup] então
11:     retorna lista[sup]
12: senão
13:     retorna elemento não encontrado
```



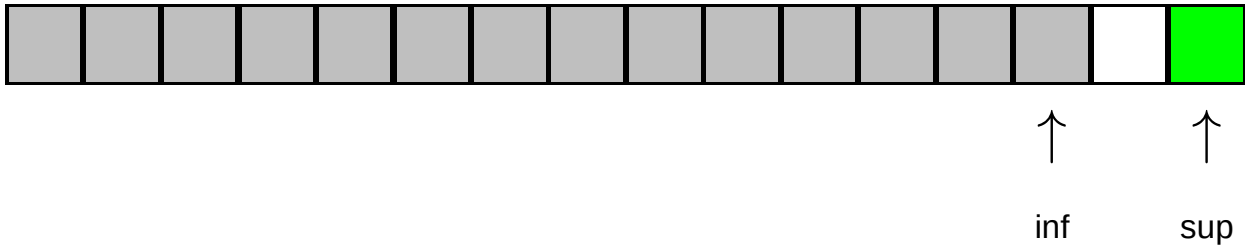
2a iter.

```
1:  BuscaBinária(chave, lista[0...n])
2:  inf ← −1
3:  sup ← n
4:  enquanto inf < sup − 1 faça
5:      meio ← ⌊(inf+sup)/2⌋
6:      se chave ≤ lista[meio] então
7:          sup ← meio
8:      senão
9:          inf ← meio
10: se chave = lista[sup] então
11:     retorna lista[sup]
12: senão
13:     retorna elemento não encontrado
```



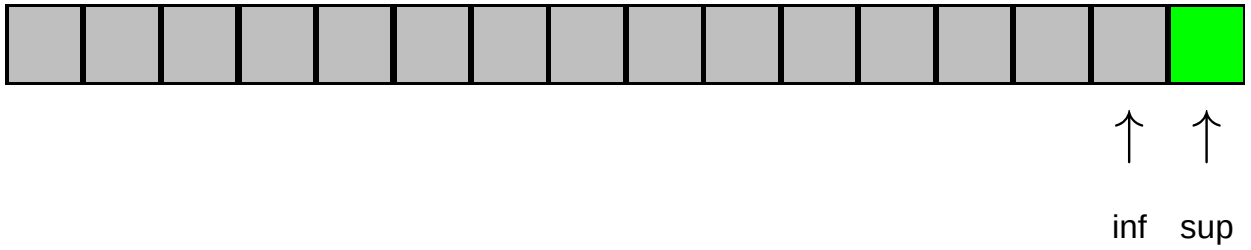
3a iter.

```
1:  BuscaBinária(chave, lista[0...n])
2:  inf ← −1
3:  sup ← n
4:  enquanto inf < sup − 1 faça
5:      meio ← ⌊(inf+sup)/2⌋
6:      se chave ≤ lista[meio] então
7:          sup ← meio
8:      senão
9:          inf ← meio
10: se chave = lista[sup] então
11:     retorna lista[sup]
12: senão
13:     retorna elemento não encontrado
```



4a iter.

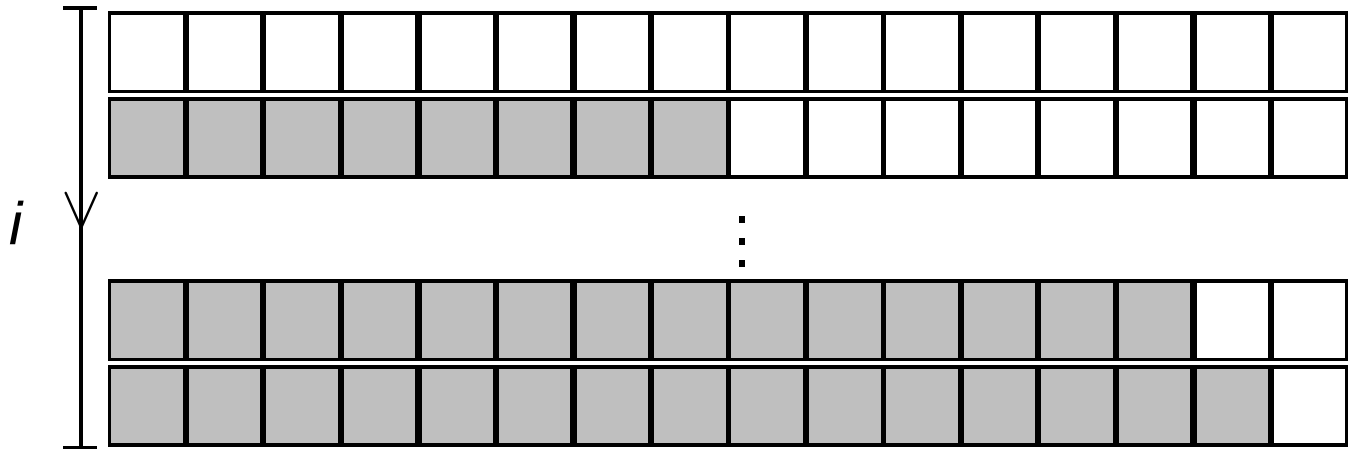
```
1:  BuscaBinária(chave, lista[0 . . . n])
2:  inf ← −1
3:  sup ← n
4:  enquanto inf < sup − 1 faça
5:      meio ← ⌊(inf + sup)/2⌋
6:      se chave ≤ lista[meio] então
7:          sup ← meio
8:      senão
9:          inf ← meio
10: se chave = lista[sup] então
11:     retorna lista[sup]
12: senão
13:     retorna elemento não encontrado
```



5a iter.

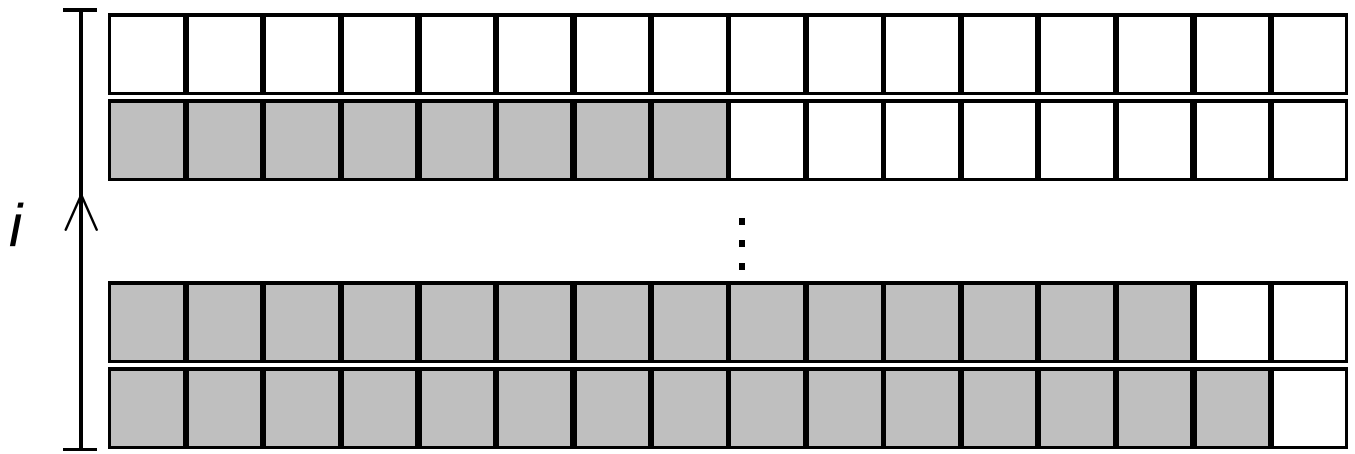
Complexidade da Busca Binária

- Cada iteração (linhas 4–9) divide ao meio a faixa entre *inf* e *sup*.
- Termina quando faixa tem apenas 1 elemento.
- Quantas iterações, dividindo ao meio, até 1 elemento?
-



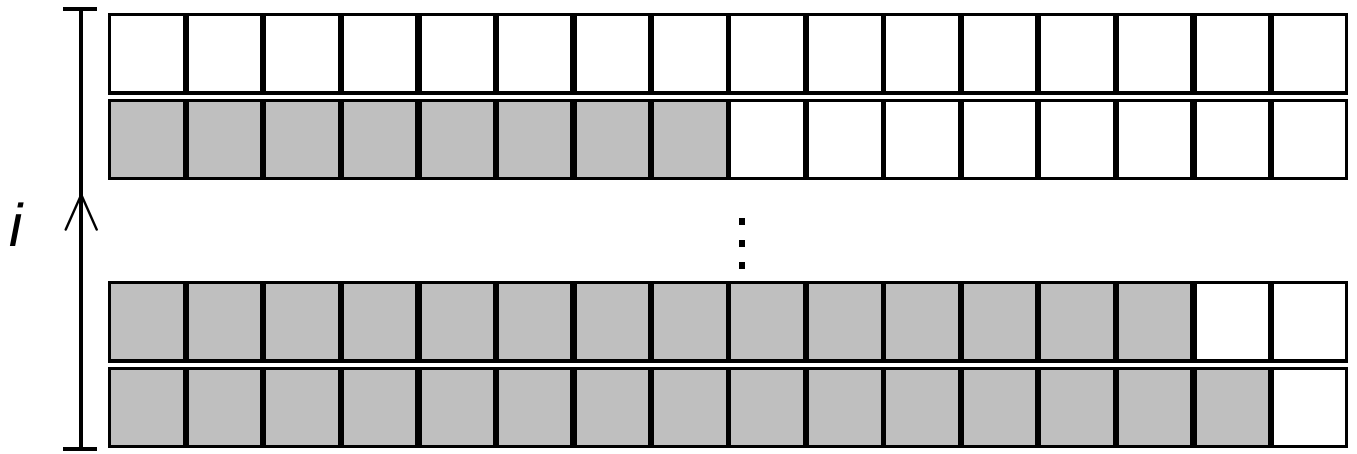
Complexidade da Busca Binária

- Cada iteração (linhas 4–9) divide ao meio a faixa entre *inf* e *sup*.
- Termina quando faixa tem apenas 1 elemento.
- Quantas iterações, dividindo ao meio, até 1 elemento?
- Equivalente: quantas vezes devo dobrar até chegar a *n*?



Complexidade da Busca Binária

- Cada iteração (linhas 4–9) divide ao meio a faixa entre *inf* e *sup*.
- Termina quando faixa tem apenas 1 elemento.
- Quantas iterações, dividindo ao meio, até 1 elemento?
- Equivalente: quantas vezes devo dobrar até chegar a n?



$$2^i = (n) \quad \Leftrightarrow \quad i = \log_2(n)$$

Complexidade da Busca Binária

	1:	BuscaBinária($chave, lista[0 \ldots n]$)
	2:	$inf \leftarrow -1$
	3:	$sup \leftarrow n$
$(\log_2 n) \times 1$	4:	**enquanto** $inf < sup - 1$ **faça**
1	5:	$meio \leftarrow \lfloor \frac{inf+sup}{2} \rfloor$
1	6:	**se** $chave \leq lista[meio].chave$ **então**
1	7:	$sup \leftarrow meio$
1	8:	**senão**
1	9:	$inf \leftarrow meio$
1	10:	**se** $chave = lista[sup].chave$ **então**
	11:	**retorna** $lista[sup]$
	12:	**senão**
	13:	**retorna** elemento não encontrado

Total operações: $1 + \log_2 n$, para n elementos.

Conceitos Básicos: busca sequencial $\times$ busca binária

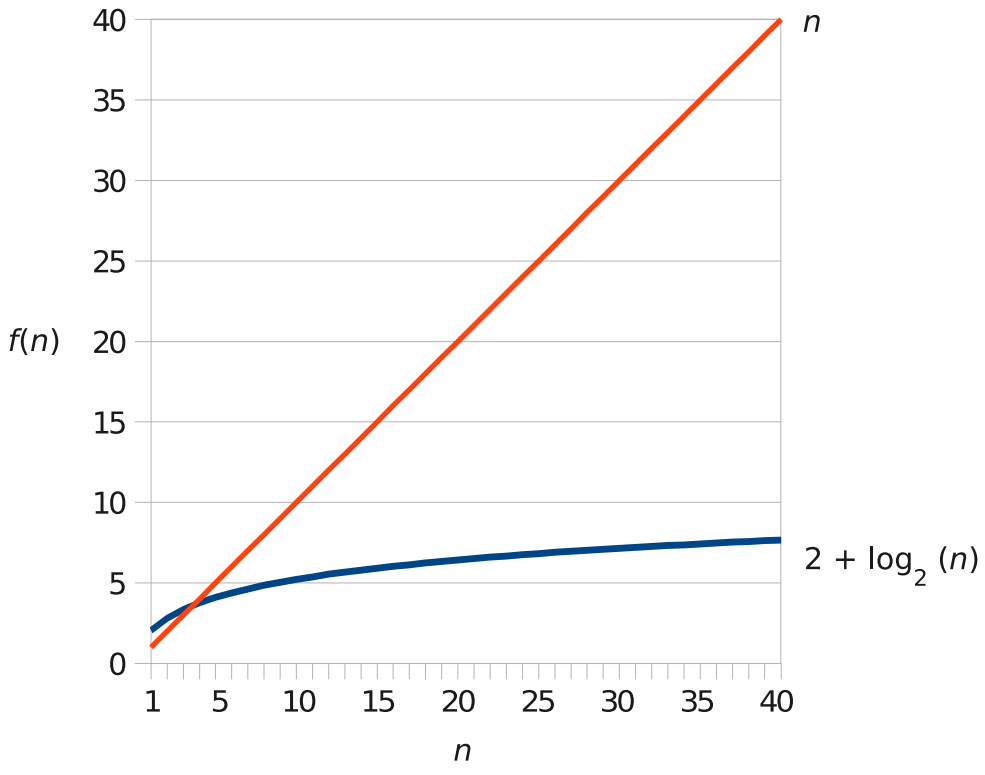
- **Problema de busca no pior caso**: busca binária leva $g(n) = 1 + \log_2 n$ passos e busca sequencial leva $f(n) = n$ passos;

Conceitos Básicos: busca sequencial $\times$ busca binária

- **Problema de busca no pior caso**: busca binária leva $g(n) = 1 + \log_2 n$ passos e busca sequencial leva $f(n) = n$ passos;
- Vamos comparar $f(n) = n$ e $g(n) = 1 + \log_2 n$:

Conceitos Básicos: busca sequencial $\times$ busca binária

- **Problema de busca no pior caso**: busca binária leva $g(n) = 1 + \log_2 n$ passos e busca sequencial leva $f(n) = n$ passos;
- Vamos comparar $f(n) = n$ e $g(n) = 1 + \log_2 n$:

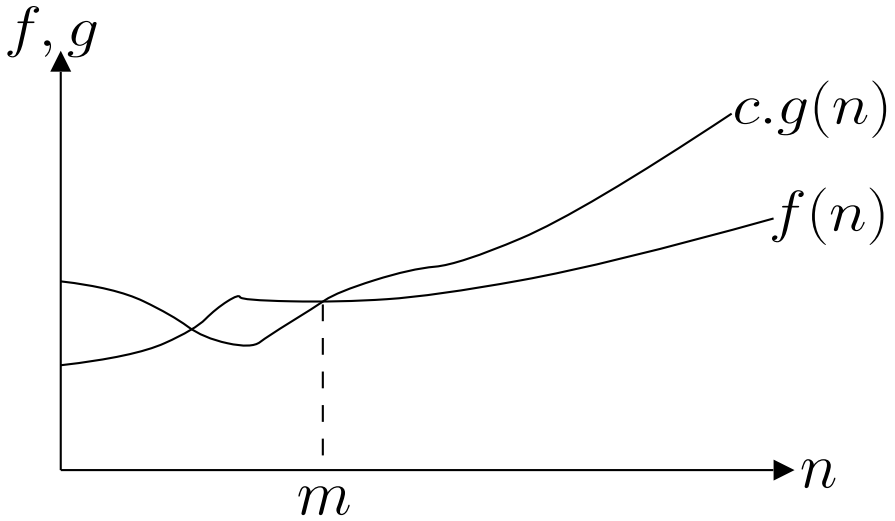


Notação Assintótica: Notação O

- **Análise assintótica dos programas:** mede como $f(n)$ se comporta conforme n aumenta indefinidamente;

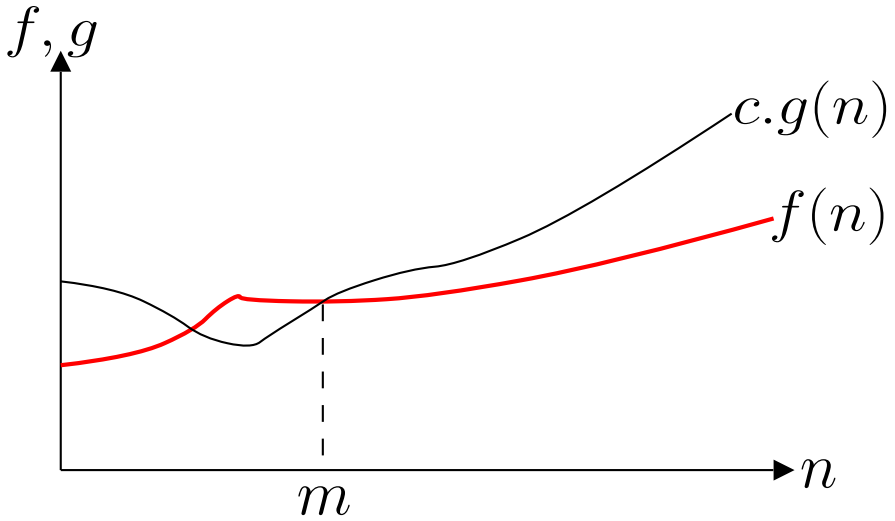
Notação Assintótica: Notação O

- **Análise assintótica dos programas:** mede como $f(n)$ se comporta conforme n aumenta indefinidamente;
- Uma função $f(n)$ é $O(g(n))$ (notação $f(n) = O(g(n))$) se existem duas constantes positivas c e m tais que $f(n) \leq c.g(n)$, para todo $n \geq m$;



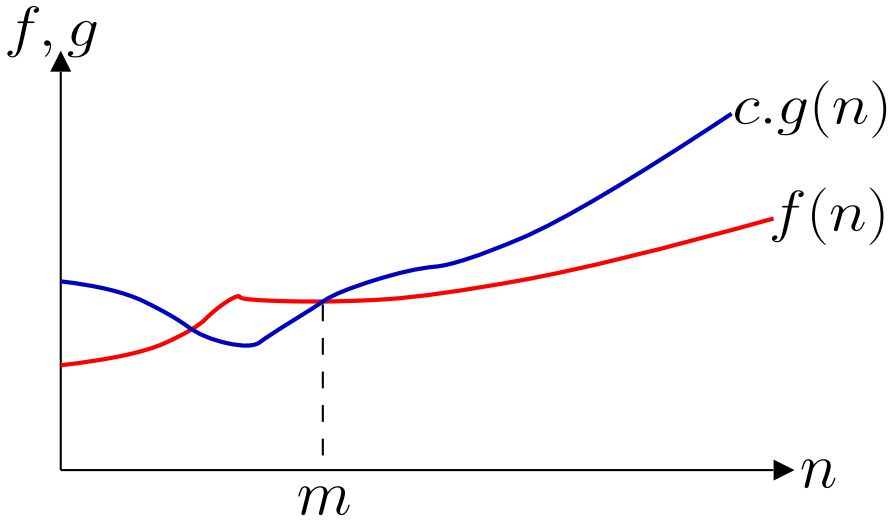
Notação Assintótica: Notação O

- **Análise assintótica dos programas:** mede como $f(n)$ se comporta conforme n aumenta indefinidamente;
- Uma função $f(n)$ é $O(g(n))$ (notação $f(n) = O(g(n))$) se existem duas constantes positivas c e m tais que $f(n) \leq c.g(n)$, para todo $n \geq m$;
- $f(n) = O(g(n))$: $g(n)$ é limite superior para $f(n)$;



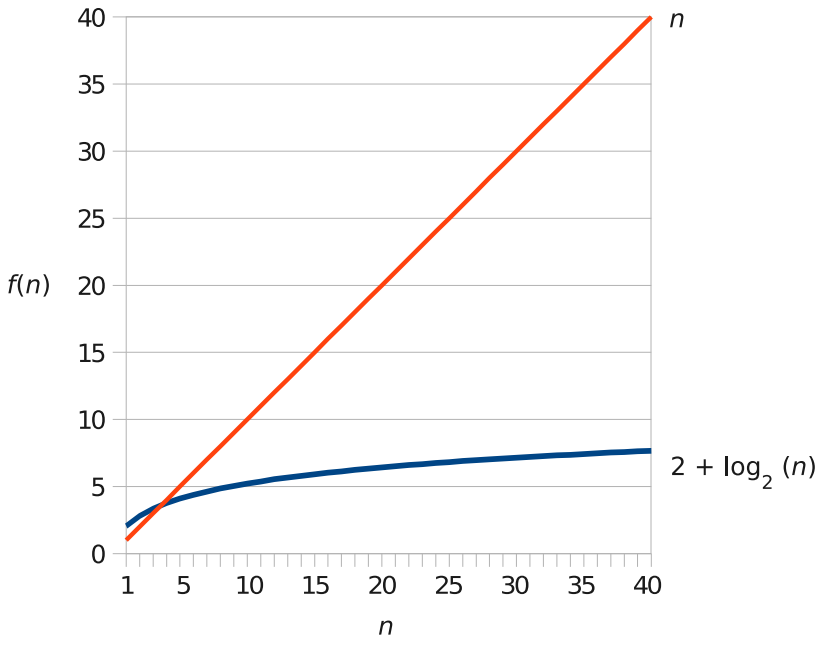
Notação Assintótica: Notação O

- **Análise assintótica dos programas:** mede como $f(n)$ se comporta conforme n aumenta indefinidamente;
- Uma função $f(n)$ é $O(g(n))$ (notação $f(n) = O(g(n))$) se existem duas constantes positivas c e m tais que $f(n) \leq c.g(n)$, para todo $n \geq m$;
- $f(n) = O(g(n))$: $g(n)$ é limite superior para $f(n)$;



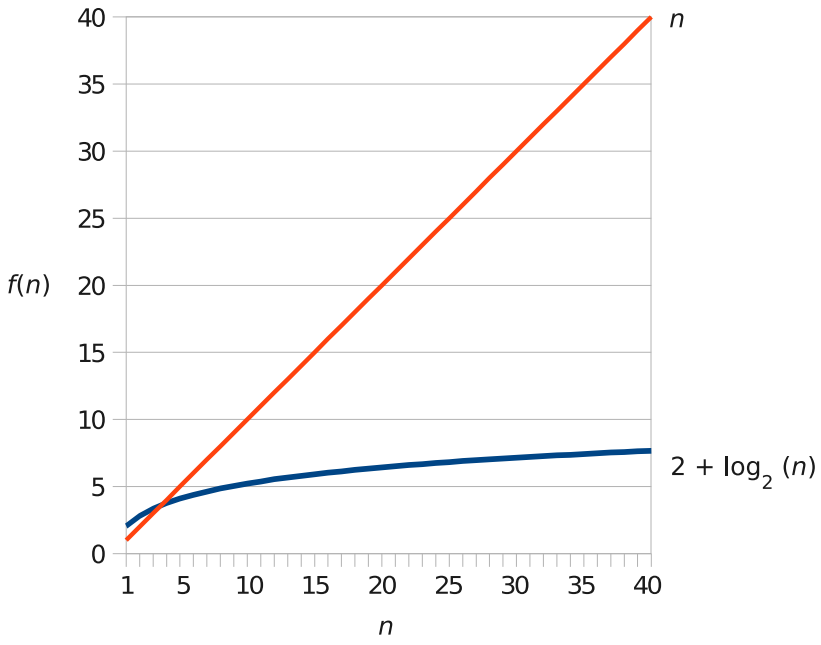
Notação Assintótica: Notação *O*

- Qual a complexidade assintótica da busca sequencial e da busca binária?



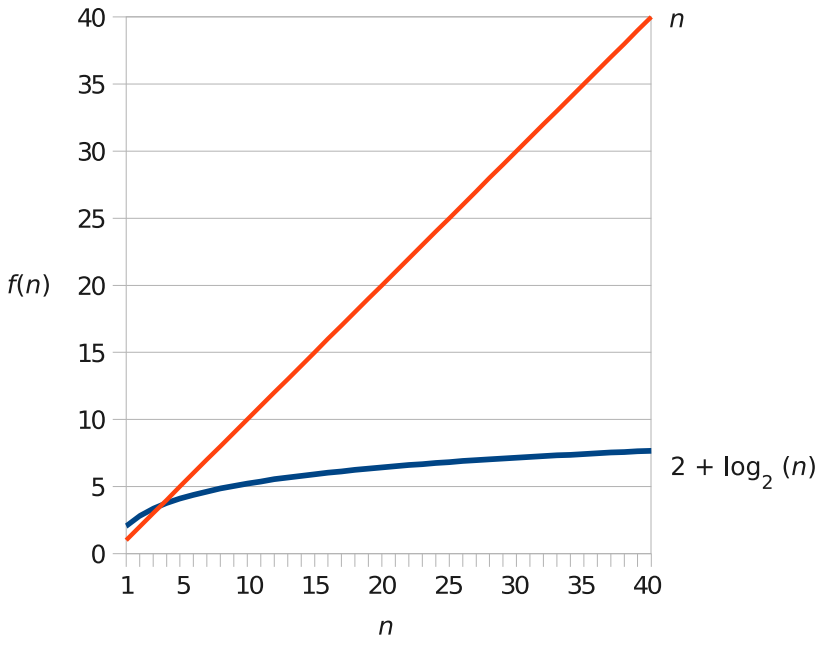
Notação Assintótica: Notação O

- Qual a complexidade assintótica da busca sequencial e da busca binária?
- Busca sequencial: $n = O(n)$, para $c \geq 1$, $m = 1$;



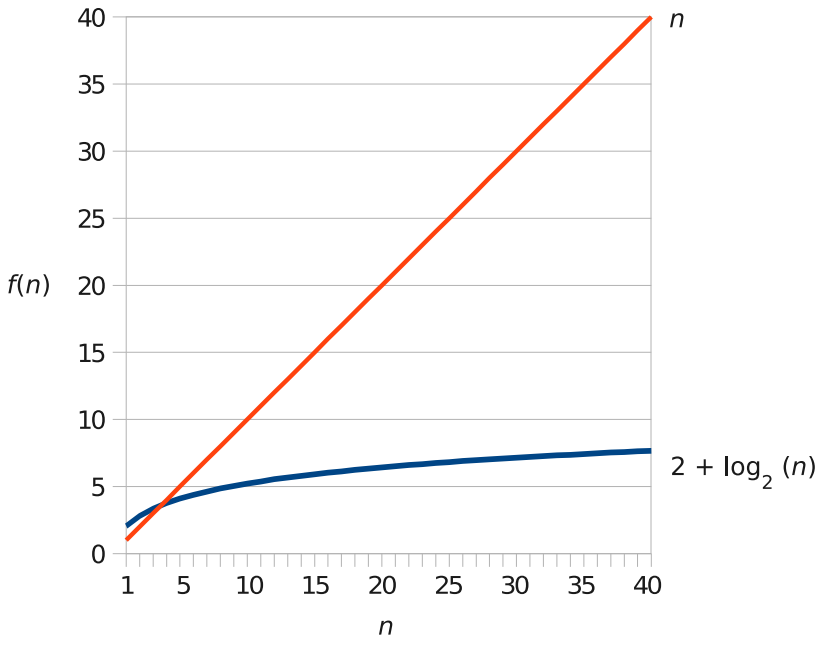
Notação Assintótica: Notação O

- Qual a complexidade assintótica da busca sequencial e da busca binária?
- Busca sequencial: $n = O(n)$, para $c \geq 1$, $m = 1$;
- Busca binária: $1 + \log_2 n = O(\log_2 n)$, para $c \geq 1$, $m = 2$;



Notação Assintótica: Notação O

- Qual a complexidade assintótica da busca sequencial e da busca binária?
- Busca sequencial: $n = O(n)$, para $c \geq 1$, $m = 1$;
- Busca binária: $1 + \log_2 n = O(\log_2 n)$, para $c \geq 1$, $m = 2$;
- Como $1 + \log_2 n = O(n)$, $c \geq 1$, $m = 1$, a busca binária é assintoticamente mais eficiente que a busca sequencial!



Notação Assintótica: Notação O

Vamos analisar o trecho de código abaixo:

1: $sum1 \leftarrow 0$;
2: **para** ($i = 1; i \leq n; i++$) **faça**
3: **para** ($j = 1; j \leq n; j++$) **faça**
4: $sum1 \leftarrow sum1 + 1$;

Notação Assintótica: Notação O

Vamos analisar o trecho de código abaixo:

```
1:  sum1 ← 0;
2:  para (i = 1; i ≤ n; i++) faça          % n vezes
3:      para (j = 1; j ≤ n; j++) faça
4:          sum1 ← sum1 + 1;
```

Notação Assintótica: Notação O

Vamos analisar o trecho de código abaixo:

```
1:  sum1 ← 0;
2:  para (i = 1; i ≤ n; i++) faça          % n vezes
3:      para (j = 1; j ≤ n; j++) faça      % n vezes
4:          sum1 ← sum1 + 1;
```

Notação Assintótica: Notação O

Vamos analisar o trecho de código abaixo:

```
1:  sum1 ← 0;
2:  para (i = 1; i ≤ n; i++) faça          % n vezes
3:      para (j = 1; j ≤ n; j++) faça      % n vezes
4:          sum1 ← sum1 + 1;               % n² vezes
```

Complexidade assintótica: $O(n^2)$!

Notação Assintótica: Notação O

Outro trecho de código:

```
1:  sum2 ← 0;
2:  para (i = 1; i ≤ n; i++) faça
3:      para (j = 1; j ≤ i; j++) faça
4:          sum2 ← sum2 + 1;
```

Complexidade assintótica:

$$1+2+3+\ldots+n = \sum_{j=1}^{n} j = \frac{n(n+1)}{2} = \frac{n^2+n}{2} = O(n^2), c \geq 1, m = 1.$$

Notação Assintótica: Notação O

Apenas mais um trecho de código:

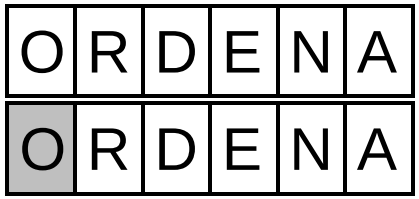
```
1:  sum3 ← 0;
2:  para (i = 1; i ≤ n; i++) faça
3:      para (j = 1; j ≤ n; j++) faça
4:          sum3 ← sum3 + 1;
```

Complexidade assintótica:

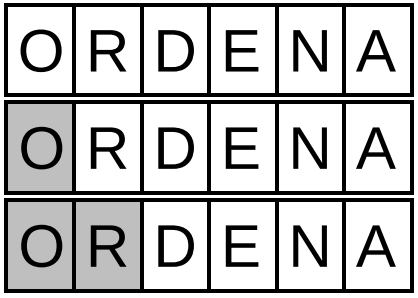
$$n+(n-1)+\ldots+1 = \sum_{j=1}^{n} j = \frac{n(n+1)}{2} = \frac{n^2+n}{2} = O(n^2), c \geq 1, m = 1.$$

Ordenação por inserção - Exemplo

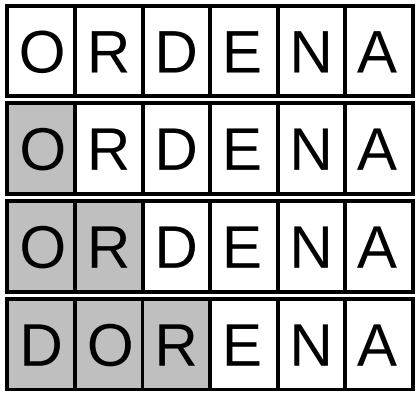
Ordenação por inserção - Exemplo



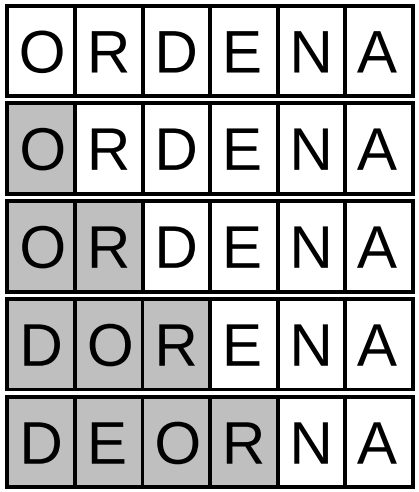
Ordenação por inserção - Exemplo



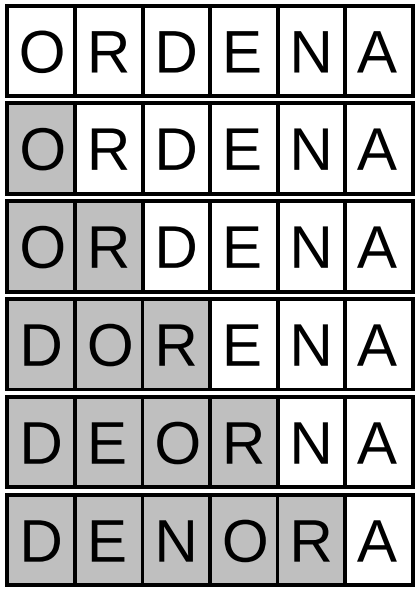
Ordenação por inserção - Exemplo



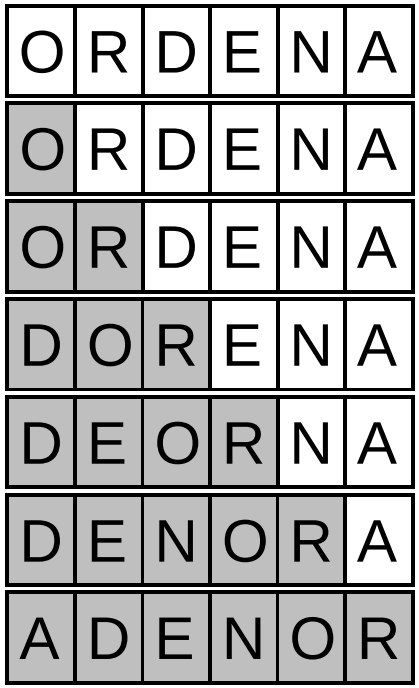
Ordenação por inserção - Exemplo



Ordenação por inserção - Exemplo



Ordenação por inserção - Exemplo



Ordenação por Inserção - Algoritmo

```
1:  Ordenação-Inserção(A);
2:      for (j = 2; j ≤ n; j++) do
3:          key ← A[j];
            // Insere A[j] na sequência ordenada A[1..j − 1]
4:          i ← (j − 1);
5:          while (i > 0) and (A[i] > chave) do
6:              A[i + 1] ← A[i];
7:              i ← (i − 1);
8:          A[i + 1] ← chave;
```

Ordenação por Inserção - Complexidade Local

1: **Ordenação-Inserção(**A**);** **Frequência**
2: **for** $(j = 2; j \leq n; j++)$ **do**

Ordenação por Inserção - Complexidade Local

		Frequência
1:	**Ordenação-Inserção(**A**);**	
2:	**for** $(j = 2; j \leq n; j++)$ **do**	$(n-1)+1$
3:	$key \leftarrow A[j]$;	

Ordenação por Inserção - Complexidade Local

		Frequência
1:	**Ordenação-Inserção(**A**);**	
2:	**for** $(j = 2; j \leq n; j++)$ **do**	$(n-1)+1$
3:	$key \leftarrow A[j]$;	$n-1$
4:	$i \leftarrow (j-1)$;	

Ordenação por Inserção - Complexidade Local

		Frequência
1:	**Ordenação-Inserção(**A**);**	
2:	**for** $(j = 2; j \leq n; j++)$ **do**	$(n-1)+1$
3:	$key \leftarrow A[j]$;	$n-1$
4:	$i \leftarrow (j-1)$;	$n-1$
5:	**while** $(i > 0)$ **and** $(A[i] > chave)$ **do**	

Ordenação por Inserção - Complexidade Local

		Frequência
1:	**Ordenação-Inserção(**A**);**	
2:	**for** $(j = 2; j \leq n; j++)$ **do**	$(n-1)+1$
3:	$key \leftarrow A[j];$	$n-1$
4:	$i \leftarrow (j-1);$	$n-1$
5:	**while** $(i > 0)$ **and** $(A[i] > chave)$ **do**	$\sum_{j=2}^{n} t_j$
6:	$A[i+1] \leftarrow A[i];$	

Ordenação por Inserção - Complexidade Local

		Frequência
1:	**Ordenação-Inserção(A);**	
2:	**for** $(j = 2; j \leq n; j++)$ **do**	$(n-1)+1$
3:	$key \leftarrow A[j]$;	$n-1$
4:	$i \leftarrow (j-1)$;	$n-1$
5:	**while** $(i > 0)$ **and** $(A[i] > chave)$ **do**	$\sum_{j=2}^{n} t_j$
6:	$A[i+1] \leftarrow A[i]$;	$\sum_{j=2}^{n} (t_j - 1)$
7:	$i \leftarrow (i-1)$;	

Ordenação por Inserção - Complexidade Local

		Frequência
1:	**Ordenação-Inserção(**A**);**	
2:	**for** $(j = 2; j \leq n; j++)$ **do**	$(n-1)+1$
3:	$key \leftarrow A[j];$	$n-1$
4:	$i \leftarrow (j-1);$	$n-1$
5:	**while** $(i > 0)$ **and** $(A[i] > chave)$ **do**	$\sum_{j=2}^{n} t_j$
6:	$A[i+1] \leftarrow A[i];$	$\sum_{j=2}^{n}(t_j - 1)$
7:	$i \leftarrow (i-1);$	$\sum_{j=2}^{n}(t_j - 1)$
8:	$A[i+1] \leftarrow chave;$	

Ordenação por Inserção - Complexidade Local

		Frequência
1:	**Ordenação-Inserção(**A**);**	
2:	**for** $(j = 2; j \leq n; j++)$ **do**	$(n-1)+1$
3:	$key \leftarrow A[j]$;	$n-1$
4:	$i \leftarrow (j-1)$;	$n-1$
5:	**while** $(i > 0)$ **and** $(A[i] > chave)$ **do**	$\sum_{j=2}^{n} t_j$
6:	$A[i+1] \leftarrow A[i]$;	$\sum_{j=2}^{n}(t_j - 1)$
7:	$i \leftarrow (i-1)$;	$\sum_{j=2}^{n}(t_j - 1)$
8:	$A[i+1] \leftarrow chave$;	$n-1$

onde t_j é o número de vezes que o teste do laço **while** é executado para cada valor de j.

Ordenação por Inserção - Análise Assintótica

Operação relevante: comparações entre chaves

```
1:  Ordenação-Inserção(A);
2:      for (j = 2; j ≤ n; j++) do
3:          key ← A[j];
            // Insere A[j] na sequência ordenada A[1..j − 1]
4:          i ← (j − 1);
5:          while (i > 0) and (A[i] > chave) do
6:              A[i + 1] ← A[i];
7:              i ← (i − 1);
8:          A[i + 1] ← chave;
```

Ordenação por Inserção - Análise Assintótica

1: **Ordenação-Inserção(**A**);**
2: **for** $(j = 2; j \leq n; j++)$ **do**
3: $key \leftarrow A[j]$;
// *Insere* $A[j]$ *na sequência ordenada* $A[1..j-1]$
4: $i \leftarrow (j-1)$;
5: **while** $(i > 0)$ **and** $(A[i] > chave)$ **faça**
6: $A[i+1] \leftarrow A[i]$;
7: $i \leftarrow (i-1)$;
8: $A[i+1] \leftarrow chave$;

Melhor caso: sequência já ordenada;
Linha 2 executa $n-1$ vezes: $O(n)$!

Ordenação por Inserção - Análise Assintótica

```
1:  Ordenação-Inserção(A);
2:      for (j = 2; j ≤ n; j++) do
3:          key ← A[j];
            // Insere A[j] na sequência ordenada A[1..j − 1]
4:          i ← (j − 1);
5:          while (i > 0) and (A[i] > chave) faça
6:              A[i + 1] ← A[i];
7:              i ← (i − 1);
8:          A[i + 1] ← chave;
```

Pior caso: sequência invertida;

Linha 2 executa $n-1$ vezes e linha 5 executa:

$$2+3+\ldots+n=\sum_{j=2}^{n} j=\frac{(n-1)(n+2)}{2}=\frac{n^2+n-2}{2}=O(n^2).$$

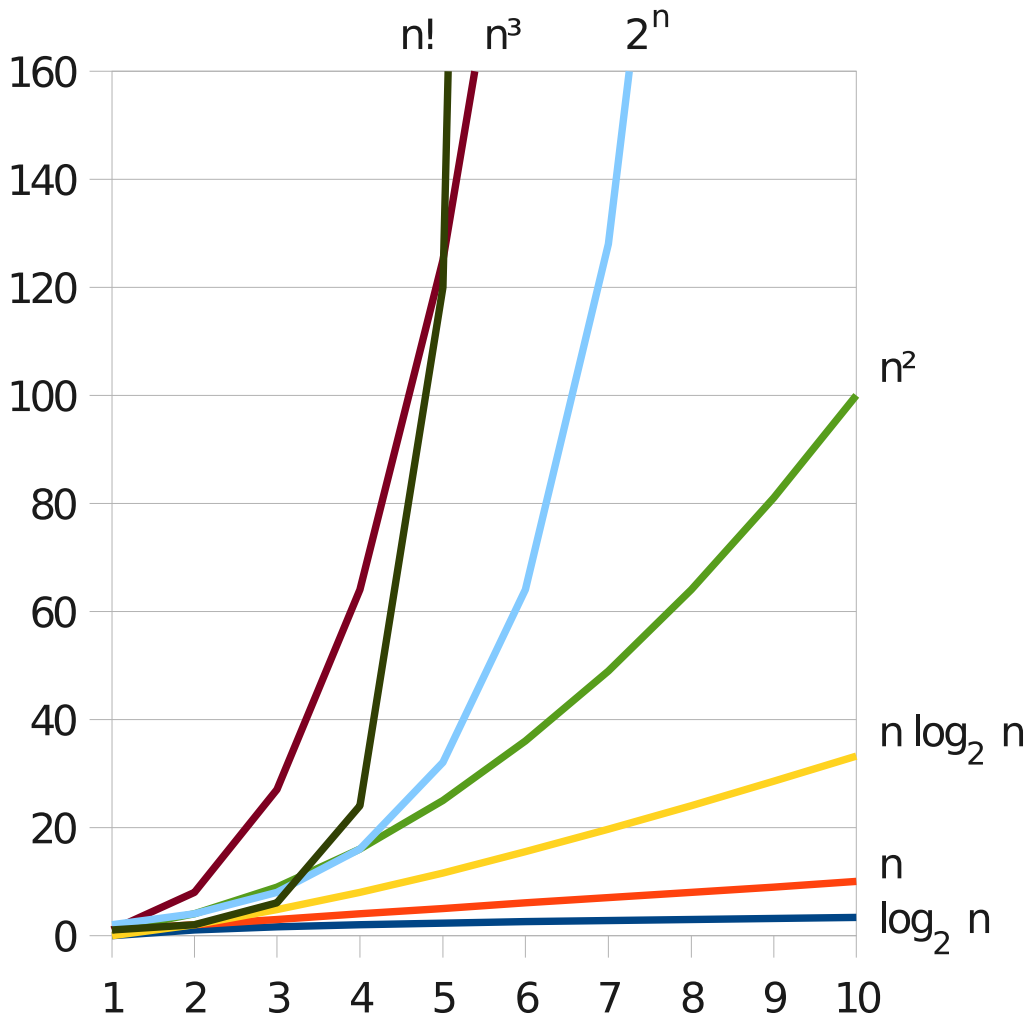
Ordenação por Inserção - Análise Assintótica

1: **Ordenação-Inserção(A);**
2: **for** $(j = 2; j \leq n; j++)$ **do**
3: $key \leftarrow A[j]$;
$// \ Insere\ A[j]\ na\ sequência\ ordenada\ A[1..j-1]$
4: $i \leftarrow (j-1)$;
5: **while** $(i > 0)$ **and** $(A[i] > chave)$ **faça**
6: $A[i+1] \leftarrow A[i]$;
7: $i \leftarrow (i-1)$;
8: $A[i+1] \leftarrow chave$;

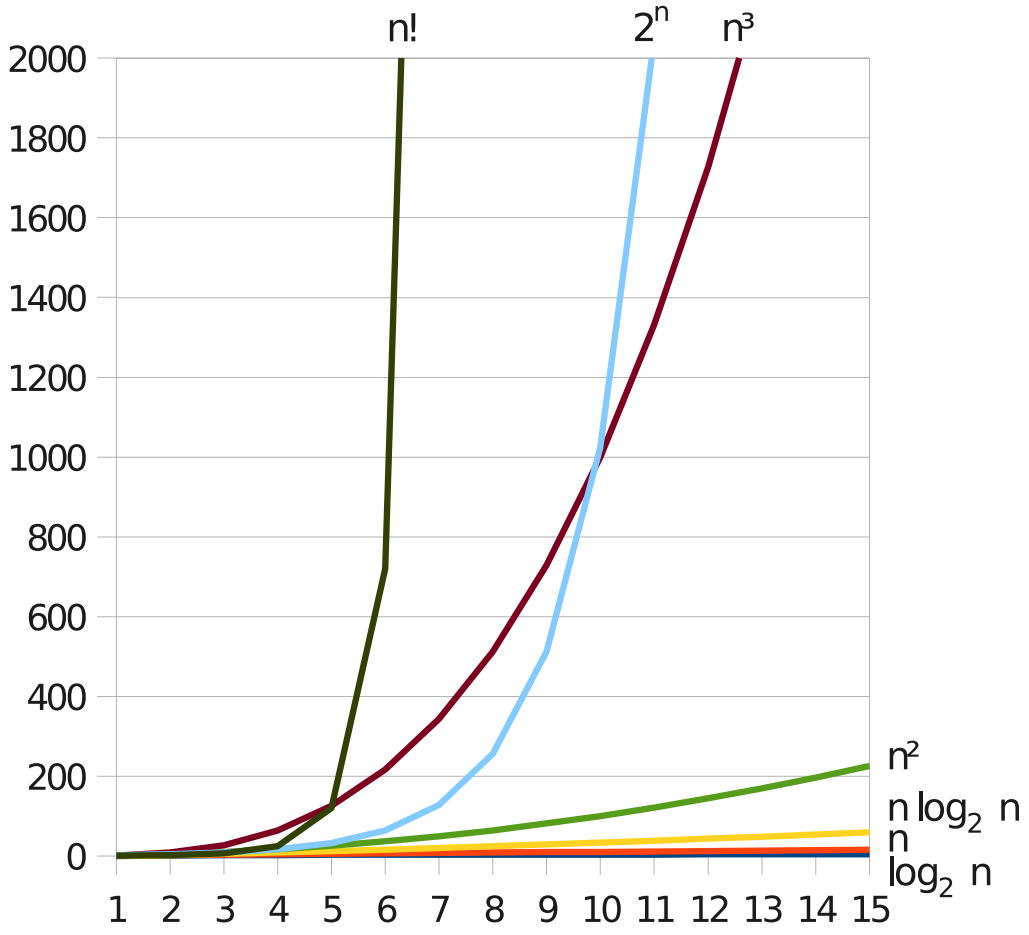
Pior caso: sequência invertida;

$$\frac{n^2+n-2}{2} = O(n^2), c \geq 1, m = 2.$$

Funções de complexidade frequentes



Funções de complexidade frequentes



Bibliografia

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. e STEIN, C. *Introduction to Algorithms*, 3ª edição, MIT Press, 2009.

SZWARCFITER, J. L. e MARKENZON, L. Estruturas de Dados e seus Algoritmos, LTC, 1994.

ZIVIANI, N. Projeto de Algoritmos: com implementações em Pascal e C, 2ª edição, Cengage Learning, 2009.