

FCT/Unesp – Presidente Prudente
Departamento de Matemática e Computação

Análise Assintótica

Parte I

Prof. Danilo Medeiros Eler
danilo.eler@unesp.br

Apresentação adaptada (ver referências)

Eficiência de Algoritmos

- Algoritmos criados para resolver um mesmo problema podem diferenciar de forma quanto a sua eficiência

n	BubbleSort Tradicional	QuickSort	HeapSort	ShellSort	InsertionSort	SelectionSort	MergeSort
100	0	0.002	0	0	0	0	0
200	0.002	0.002	0.001	0	0	0.001	0
300	0.004	0.001	0.001	0.001	0.001	0.002	0.001
400	0.007	0.002	0.001	0	0.002	0.003	0.001
500	0.01	0.003	0.001	0	0.003	0.004	0.001
600	0.015	0.003	0.002	0	0.003	0.007	0.001
700	0.02	0.002	0.001	0.001	0.005	0.009	0.002
800	0.028	0.003	0.002	0.001	0.007	0.011	0.003
900	0.033	0.002	0.002	0.002	0.01	0.015	0.003
1000	0.042	0.003	0.002	0.001	0.011	0.018	0.003
2000	0.173	0.003	0.006	0.003	0.055	0.075	0.007
3000	0.449	0.006	0.009	0.006	0.095	0.155	0.01
4000	0.739	0.007	0.013	0.008	0.167	0.271	0.014
5000	1.18	0.009	0.016	0.009	0.26	0.423	0.017
7000	2.395	0.011	0.024	0.015	0.508	0.826	0.024
8000	3.17	0.012	0.028	0.017	0.658	1.075	0.027
9000	4.058	0.014	0.032	0.019	0.836	1.359	0.032
10000	5.052	0.016	0.036	0.022	1.034	1.677	0.035
20000	21.139	0.033	0.077	0.05	4.053	6.689	0.073
25000	33.122	0.041	0.099	0.065	6.306	10.446	0.092
30000	48.01	0.05	0.121	0.079	9.175	15.037	0.114
40000	86.402	0.068	0.167	0.108	16.101	26.712	0.152

Análise de Algoritmos

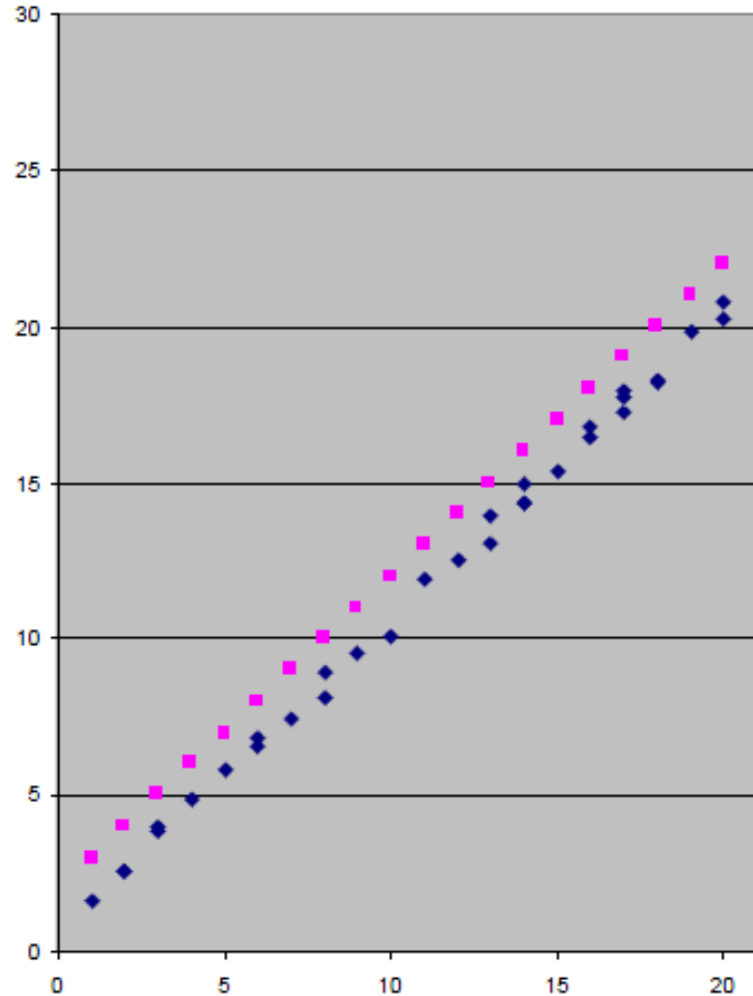
- Analisar um algoritmo significa prever os recursos que um algoritmo necessitará
- A análise nos fornece condições de
 - Comparar diversos algoritmos para o mesmo problema
 - Determinar se o algoritmo é viável para a aplicação
- Como comparar algoritmos?
- Quais são as técnicas de análise?

Análise de Algoritmos

- Possíveis técnicas de análise
 - Experimentação
 - Análise assintótica
- A análise foca principalmente no
 - Tempo de execução
 - Consumo de memória
- Porque o foco nesses dois itens?

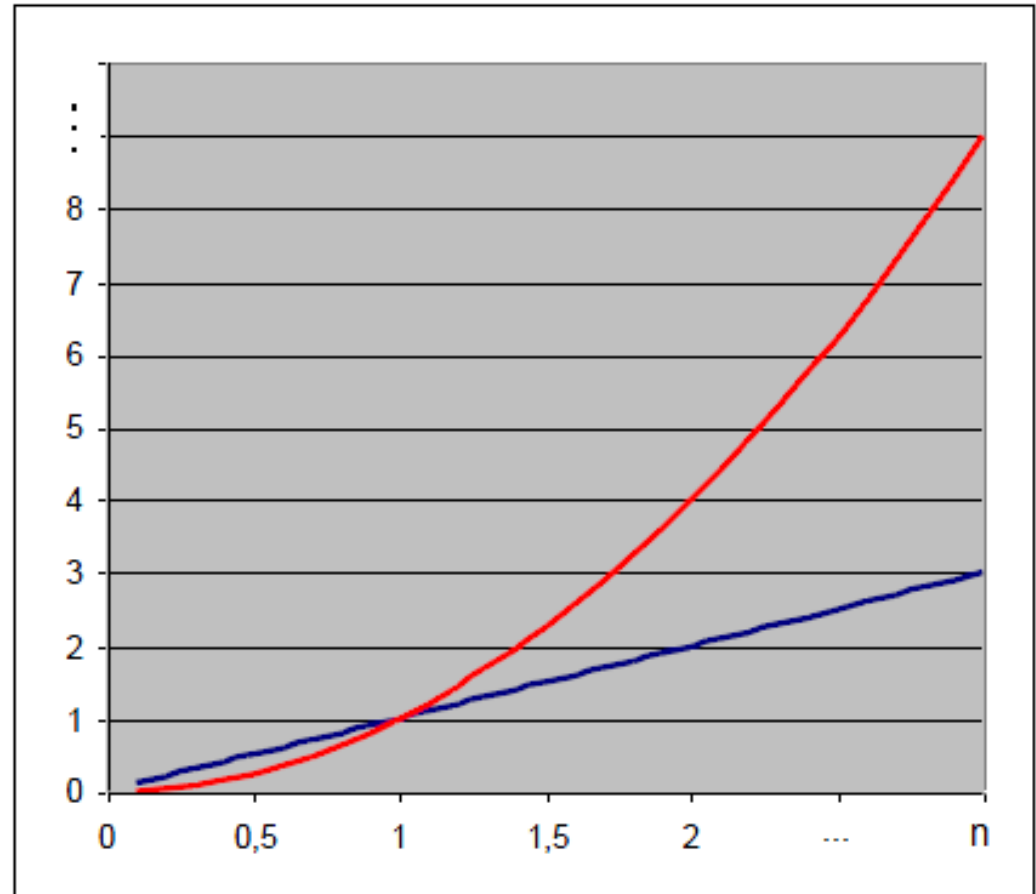
Análise de Algoritmos

- Por exemplo, o algoritmo A é executado em tempo proporcional a n
 - $f(n) = c1*n + c2$
- Variando as constantes teríamos diferentes tempos, mas o comportamento é similar



Análise de Algoritmos

- Dados dois algoritmos e suas funções de tempo de execução
 - A (azul)
 - V (vermelho)
- Qual algoritmo você prefere utilizar?



Análise de Algoritmos

- Existem vários componentes que precisamos definir antes de descrever uma metodologia de análise de algoritmos baseada em funções matemáticas
 - Uma **linguagem** para descrição de algoritmos
 - Um **modelo computacional** para execução de algoritmos
 - Uma **métrica** para medir o tempo de execução de algoritmos

Análise de Algoritmos

■ Linguagem: Pseudocódigo

- É uma mistura de linguagem natural e estruturas de programação de alto nível
- É utilizado para descrever algoritmos de forma genérica
- No entanto, não existe uma definição precisa da linguagem pseudo-código por causa de seu uso da linguagem natural
- Para aumentar sua clareza, o pseudocódigo mistura construções formais
 - Devemos tomar muito cuidado com as chamadas informais

Análise de Algoritmos

- Chamadas informais

```
int [] vetor = int[10000];
```

```
Para i = 1 ate 10000 faça
```

```
    vetor[i] = NumeroAleatorio();
```

```
OrdenaVetor(vetor);
```

Análise de Algoritmos

- **Modelo computacional:** máquina de acesso aleatório (RAM – *Random-Access Machine*)
 - **RAM** define uma máquina abstrata
 - Não deve ser confundido com “memória de acesso a aleatório”
 - Este modelo vê o computador com uma Unidade Central de Processamento conectada a uma memória
 - Cada posição da memória armazena uma palavra, que pode ser
 - Um número
 - Uma cadeia de caracteres
 - Ou um endereço
 - Ou seja, algum tipo básico da linguagem

Análise de Algoritmos

- **Modelo computacional:** máquina de acesso aleatório (RAM)
 - O termo “acesso aleatório” refere-se à capacidade da **CPU acessar uma posição arbitrária de memória em apenas uma operação primitiva**
 - Presumimos também que a **CPU** do modelo RAM pode **realizar qualquer operação primitiva em um número constante de passos** que não depende do tamanho da entrada

Análise de Algoritmos

- Operações realizadas em número constante de passos

```
int soma;
```

```
int [] vetor = int[10000];
```

```
para i = 1 ate 10000 faça
```

```
    soma = soma + vetor[i];
```

```
exibir(soma);
```

Análise de Algoritmos

- Faremos agora, depois de definido o modelo computacional, a contagem de operações primitivas de algoritmos
- Geralmente consideramos apenas algumas operações elementares
 - Comparações
 - Atribuições
- Este fator varia de autor para autor na literatura
 - Na prática, as comparações são mais demoradas que atribuições
 - Alguns analistas consideraram apenas comparações na análise de algoritmos

Análise de Algoritmos

- Qual é a função que definiria o custo computacional para a execução do algoritmo abaixo?

```
int soma;
```

```
int [] vetor = int[10000];
```

```
para i = 1 ate 10000 faça
```

```
    soma = soma + vetor[i];
```

```
exibir(soma);
```

Análise de Algoritmos

- Qual é a função que definiria o custo computacional para a execução do algoritmo abaixo?

```
int soma; _____→ 1
int [] vetor = int[10000]; _____→ 1
para i = 1 ate 10000 faça
    soma = soma + vetor[i]; _____→ 1
exibir(soma); _____→ 1
```

Análise de Algoritmos

- Qual é a função que definiria o custo computacional para a execução do algoritmo abaixo?

```
int soma; _____→ 1
int [] vetor = int[10000]; _____→ 1
para i = 1 ate 10000 faça _____→
    soma = soma + vetor[i]; _____→ 1
exibir(soma); _____→ 1
```

x 10000

Análise de Algoritmos

- Qual é a função que definiria o custo computacional para a execução do algoritmo abaixo?

```
int soma; _____→ 1
int [] vetor = int[10000]; _____→ 1
para i = 1 ate 10000 faça _____→ 1
    soma = soma + vetor[i]; _____→ 1
exibir(soma); _____→ 1
```

x 10000

Número de Instruções: $1 + 1 + 10000 + 1 = 10003$

Análise de Algoritmos

```
int somar(int vetor[], int n){  
    int soma;  
    para i = 1 ate n faça  
        soma = soma + vetor[i];  
    exibir(soma);  
    retorna(soma);  
}
```

Análise de Algoritmos

```
int somar(int vetor[], int n){  
    int soma; _____→ 1  
    para i = 1 ate n faça  
        soma = soma + vetor[i]; _____→ 1  
    exibir(soma); _____→ 1  
    retorna(soma); _____→ 1  
}
```

Análise de Algoritmos

```
int somar(int vetor[], int n){  
    int soma; —————→ 1  
    para i = 1 ate n faça —————→  
        soma = soma + vetor[i]; —————→ 1 } x n  
    exibir(soma); —————→ 1  
    retorna(soma); —————→ 1  
}
```

Análise de Algoritmos

```
int somar(int vetor[], int n){  
    int soma; —————→ 1  
    para i = 1 ate n faça —————→  
        soma = soma + vetor[i]; —————→ 1 } x n  
    exibir(soma); —————→ 1  
    retorna(soma); —————→ 1  
}
```

Número de Instruções: $1 + 1*n + 1 + 1 = n + 3$

Análise de Algoritmos

```
int somar(int vetor[], int n){  
    int soma;  
    para i = 1 ate n faça  
        soma = soma + vetor[i];  
    exibir(soma);  
    exibir(vetor, n);  
    retorna(soma);  
}
```

Análise de Algoritmos

```
int somar(int vetor[], int n){  
    int soma; _____> 1  
    para i = 1 ate n faça  
        soma = soma + vetor[i]; _____> 1  
    exibir(soma); _____> 1  
    exibir(vetor, n); _____> 1  
    retorna(soma); _____> 1  
}
```

Análise de Algoritmos

```
void exibir(int vetor[], int n){  
    para i = 1 ate n faça  
        exibir(vetor[i]);  
}
```


Análise de Algoritmos

```
void exibir(int vetor[], int n){  
    para i = 1 ate n faça  
        exibir(vetor[i]);  $\longrightarrow$  1  
}
```

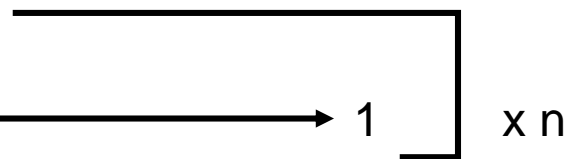
Análise de Algoritmos

```
void exibir(int vetor[], int n){  
    para i = 1 ate n faça  
        exibir(vetor[i]);  
}
```

Diagram illustrating the loop structure: a horizontal line above "faça" connects to a vertical line on the right, which then connects to a horizontal line below "exibir(vetor[i]);". An arrow points from "exibir(vetor[i]);" to the number "1", and the text "x n" is positioned to the right of the vertical line, indicating the loop repeats n times.

Análise de Algoritmos

```
void exibir(int vetor[], int n){  
    para i = 1 ate n faça  
        exibir(vetor[i]);  
}
```



Número de Instruções: $1 * n = n$

Análise de Algoritmos

```
int somar(int vetor[], int n){  
    int soma; _____→ 1  
    para i = 1 ate n faça  
        soma = soma + vetor[i]; _____→ 1  
    exibir(soma); _____→ 1  
    exibir(vetor, n); _____→ n  
    retorna(soma); _____→ 1  
}
```

Análise de Algoritmos



```
int somar(int vetor[], int n){  
    int soma; —————→ 1  
    para i = 1 ate n faça —————→  
        soma = soma + vetor[i]; —————→ 1 } x n  
    exibir(soma); —————→ 1  
    exibir(vetor, n); —————→ n  
    retorna(soma); —————→ 1  
}
```

Análise de Algoritmos

```
int somar(int vetor[], int n){  
    int soma;  $\xrightarrow{\hspace{10em}}$  1  
    para i = 1 ate n faça  $\xrightarrow{\hspace{10em}}$   $\left. \begin{array}{l} \text{soma} = \text{soma} + \text{vetor}[i]; \xrightarrow{\hspace{10em}} 1 \\ \hspace{10em} \end{array} \right\} \times n$   
    exibir(soma);  $\xrightarrow{\hspace{10em}}$  1  
    exibir(vetor, n);  $\xrightarrow{\hspace{10em}}$  n  
    retorna(soma);  $\xrightarrow{\hspace{10em}}$  1  
}
```

Número de Instruções: $1 + 1*n + 1 + n + 1 = 2n + 3$

Análise de Algoritmos



	$n + 3$
	$2n + 3$

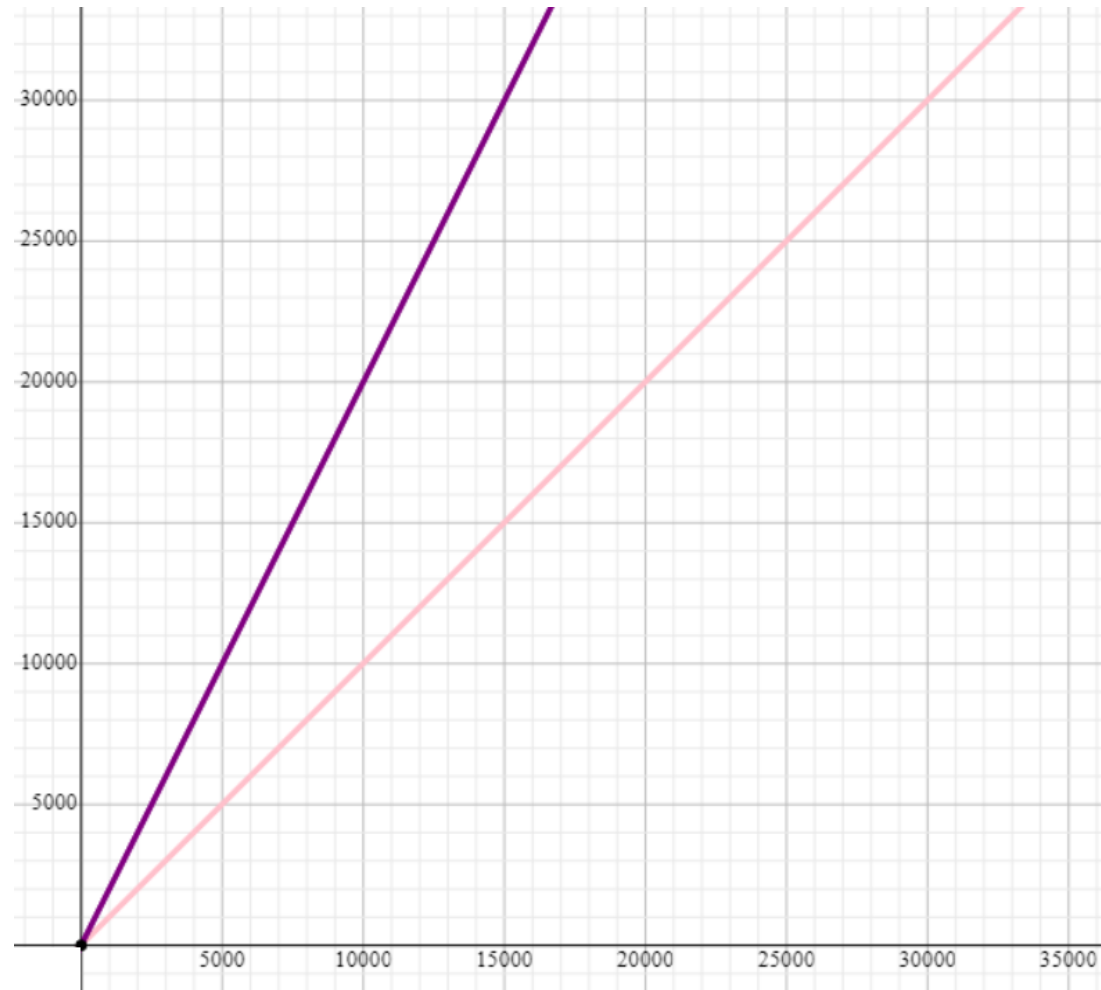


Análise de Algoritmos

■ Comportamento de uma função linear

□ n

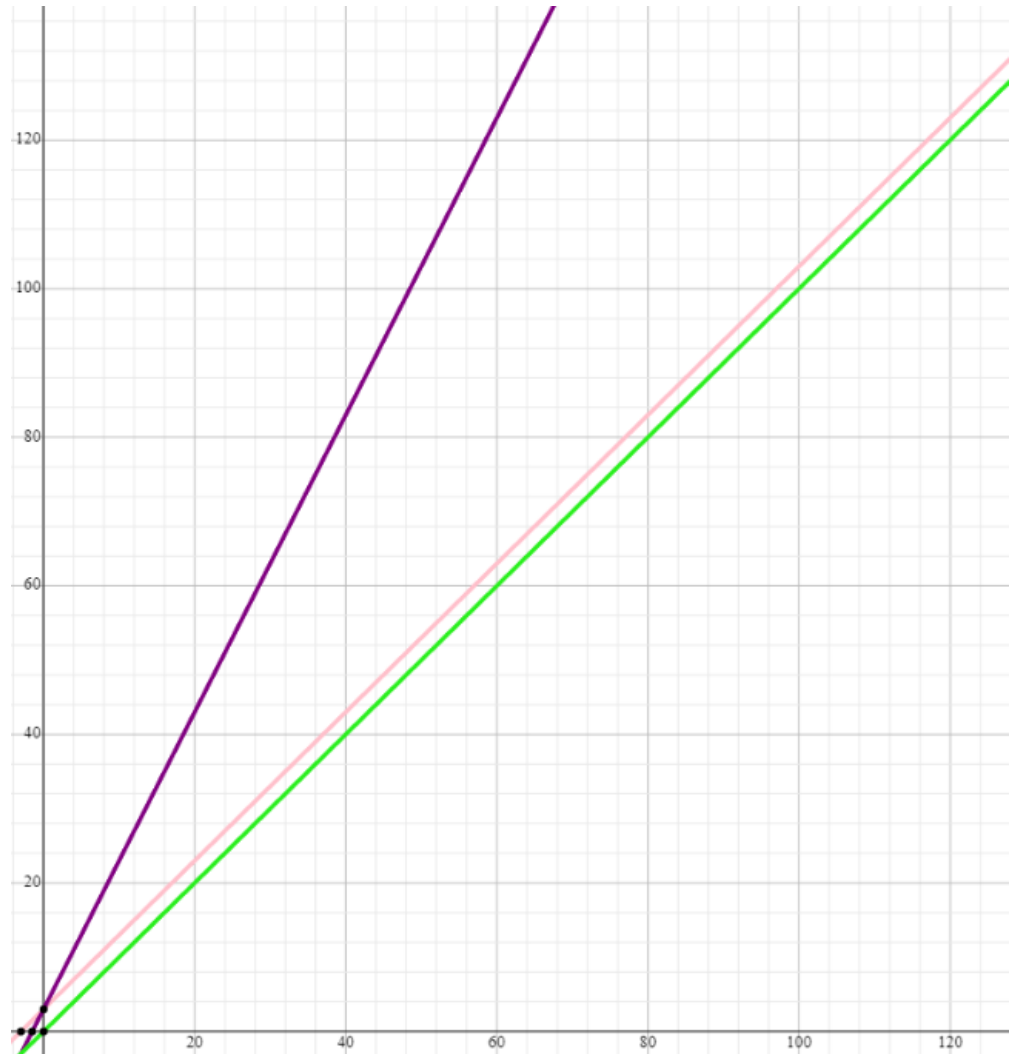
	$n + 3$
	$2n + 3$



Análise de Algoritmos

- Comportamento de uma função linear

	$n + 3$
	$2n + 3$
	n



Análise de Algoritmos

```
int menor(int vetor[], int n){
    int menor = MAX_INT;
    para i=1 ate n faça
        se (vetor[i] < menor)
            menor = vetor[i];
    retorna(menor);
}
```

Análise de Algoritmos

```
int menor(int vetor[], int n){  
    int menor = MAX_INT; → 1  
    para i=1 ate n faça  
        se (vetor[i] < menor) → 1  
            menor = vetor[i]; → 1  
    retorna(menor); → 1  
}
```

Análise de Algoritmos

```
int menor(int vetor[], int n){  
    int menor = MAX_INT;  $\longrightarrow$  1  
    para i=1 ate n faça  $\longrightarrow$   $\left. \begin{array}{l} \text{se (vetor[i] < menor) } \longrightarrow 1 \\ \text{menor = vetor[i]; } \longrightarrow 1 \end{array} \right\} \text{ x n}$   
    retorna(menor);  $\longrightarrow$  1  
}
```

Análise de Algoritmos

```
int menor(int vetor[], int n){  
    int menor = MAX_INT;  $\longrightarrow$  1  
    para i=1 ate n faça  $\longrightarrow$   $\left. \begin{array}{l} \text{se (vetor[i] < menor) } \longrightarrow 1 \\ \text{menor = vetor[i]; } \longrightarrow 1 \end{array} \right\} \times n$   
    retorna(menor);  $\longrightarrow$  1  
}
```

Número de Instruções: $1 + 2*n + 1 = 2n + 2$

Análise de Algoritmos

- É comum verificarmos o **melhor** e o **pior caso** de execução de um algoritmos
 - Eles dependem da natureza dos dados de entrada
- O **melhor caso** é aquele em que o algoritmo recebe uma entrada que o faz executar da maneira mais eficiente (mais rápida)
- O **pior caso** é aquele em que o algoritmo recebe uma entrada que o faz executar da maneira menos eficiente (mais lenta)

Análise de Algoritmos

Pior Caso de Execução

```
int menor(int vetor[], int n){
```

```
    int menor = MAX_INT;  $\longrightarrow$  1
```

```
    para i=1 ate n faça  $\longrightarrow$ 
```

```
        se (vetor[i] < menor)  $\longrightarrow$  1
```

```
            menor = vetor[i];  $\longrightarrow$  1
```

```
    retorna(menor);  $\longrightarrow$  1
```

```
}
```

x n

Número de Instruções: $1 + 2*n + 1 = 2n + 2$

Análise de Algoritmos

Pior Caso de Execução: entrada em ordem decrescente

```
int menor(int vetor[], int n){
```

vetor = {5, 4, 3, 2, 1}

```
    int menor = MAX_INT; → 1
```

```
    para i=1 ate n faça
```

```
        se (vetor[i] < menor) → 1
```

```
            menor = vetor[i]; → 1
```

x n

```
    retorna(menor); → 1
```

```
}
```

Número de Instruções: $1 + 2*n + 1 = 2n + 2$

Análise de Algoritmos

Melhor Caso de Execução?

```
int menor(int vetor[], int n){
```

```
    int menor = MAX_INT;  $\longrightarrow$  1
```

```
    para i=1 ate n faça  $\longrightarrow$ 
```

```
        se (vetor[i] < menor)  $\longrightarrow$  1
```

```
            menor = vetor[i];  $\longrightarrow$  1
```

```
    retorna(menor);  $\longrightarrow$  1
```

```
}
```

x n

Número de Instruções: $1 + 2*n + 1 = 2n + 2$

Análise de Algoritmos

Melhor Caso de Execução: entrada em ordem crescente

```
int menor(int vetor[], int n){
```

vetor = {1, 2, 3, 4, 5}

```
    int menor = MAX_INT; → 1
```

```
    para i=1 ate n faça
```

```
        se (vetor[i] < menor) → 1
```

```
            menor = vetor[i]; → 1
```

x n

```
    retorna(menor); → 1
```

```
}
```

Número de Instruções: $1 + 2*n + 1 = 2n + 2$

Análise de Algoritmos

Melhor Caso de Execução: entrada em ordem crescente

```
int menor(int vetor[], int n){
```

vetor = {1, 2, 3, 4, 5}

```
    int menor = MAX_INT; → 1
```

```
    para i=1 ate n faça
```

```
        se (vetor[i] < menor) → 1
```

```
            menor = vetor[i]; → 1
```

```
    retorna(menor); → 1
```

```
}
```

x n

executado uma vez

Número de Instruções: $1 + 2*n + 1 = 2n + 2$

Análise de Algoritmos

Melhor Caso de Execução: entrada em ordem crescente

```
int menor(int vetor[], int n){
```

vetor = {1, 2, 3, 4, 5}

```
    int menor = MAX_INT; → 1
```

```
    para i=1 ate n faça → 1
```

```
        se (vetor[i] < menor) → 1
```

```
            menor = vetor[i]; → 1
```




```
    retorna(menor); → 1
```

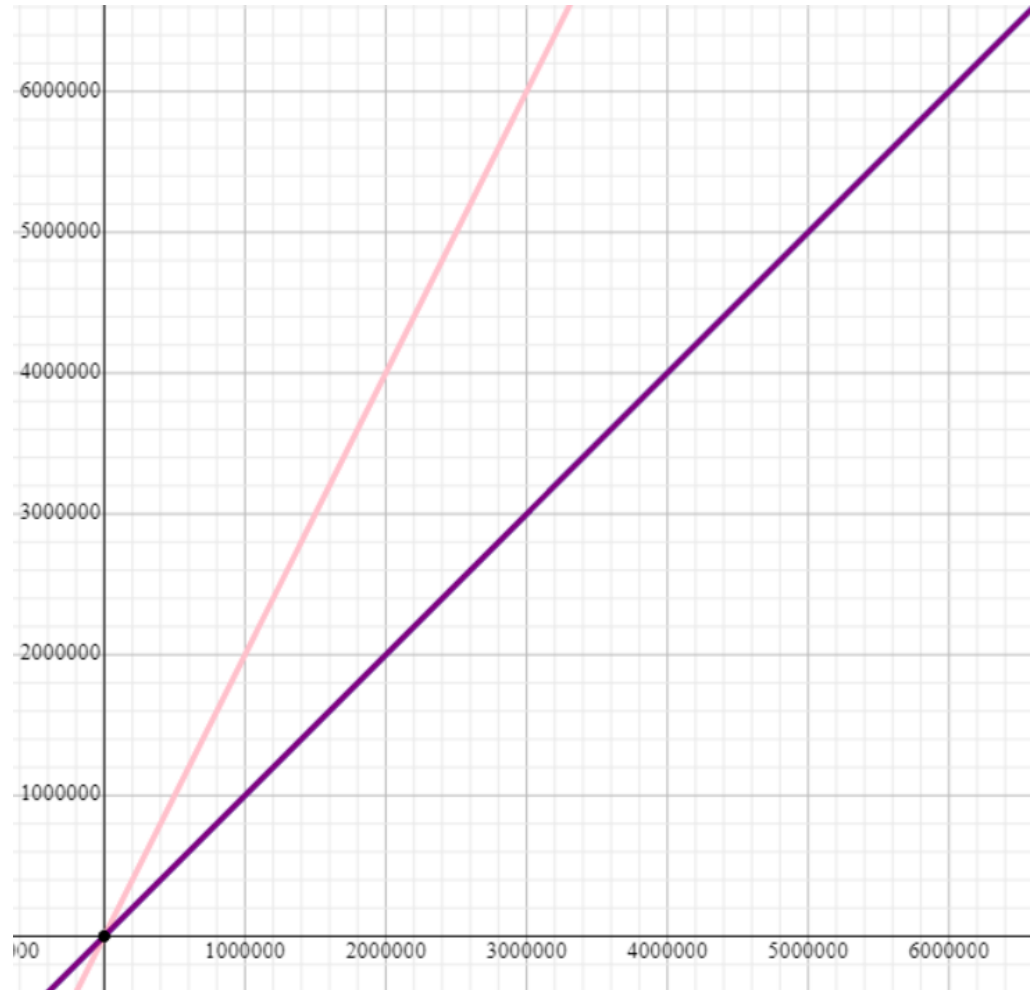
```
}
```

Número de Instruções: $1 + 1*n + 1 + 1 = n + 3$

Análise de Algoritmos




■ Comportamento de uma função linear

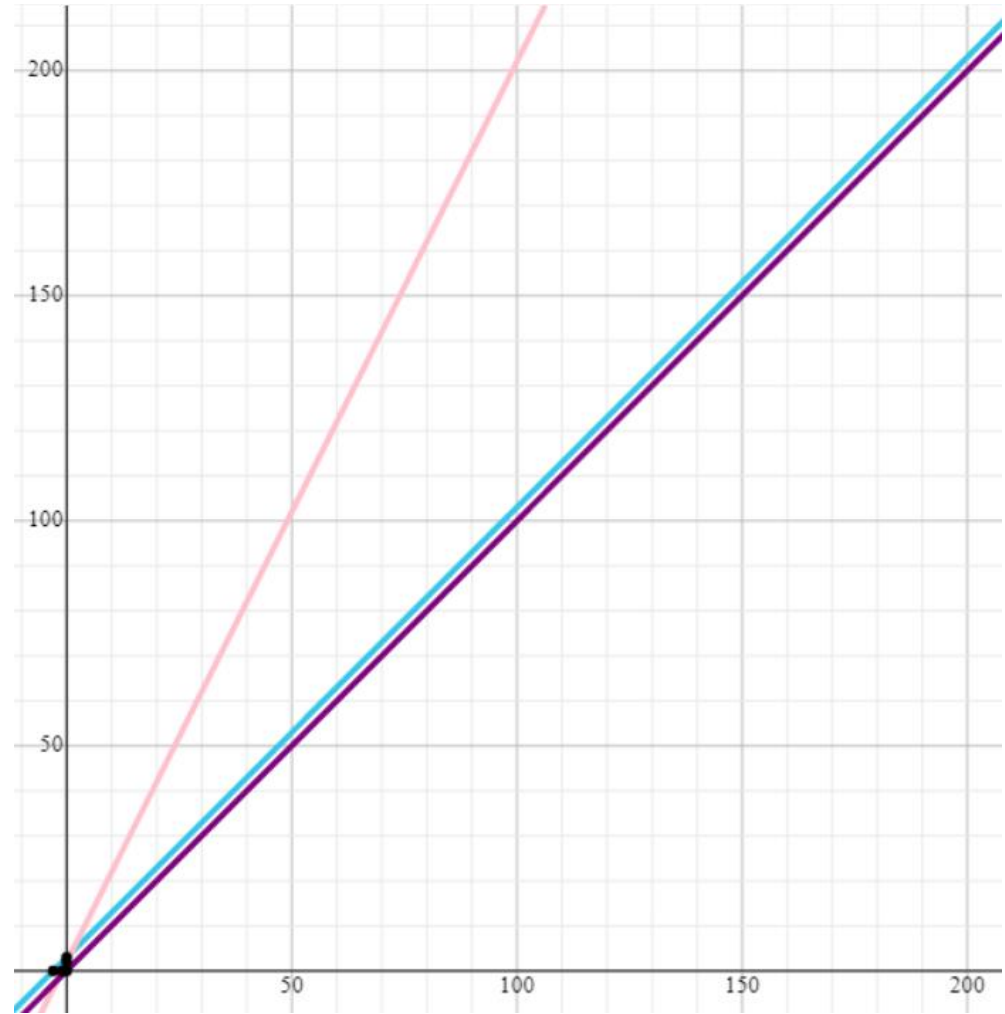
	$n + 3$	Melhor Caso
	$2n + 2$	Pior Caso
	n	



Análise de Algoritmos

■ Comportamento de uma função linear

	$n + 3$	Melhor Caso
	$2n + 2$	Pior Caso
	n	



Análise de Algoritmos

```
int somar(int vetor[], int n){  
    int soma; —————→ 1  
    para i = 1 ate n faça —————→  
        soma = soma + vetor[i]; —————→ 1 } x n  
    exibir(soma); —————→ 1  
    exibir(vetor, n); —————→ n  
    retorna(soma); —————→ 1  
}
```

Número de Instruções: $1 + 1*n + 1 + n + 1 = 2n + 3$

Referências Bibliográficas

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; (2002). Algoritmos –Teoria e Prática. Tradução da 2ª edição americana. Rio de Janeiro. Editora Campus
- TAMASSIA, ROBERTO; GOODRICH, MICHAEL T. (2004). Projeto de Algoritmos -Fundamentos, Análise e Exemplos da Internet
- ZIVIANI, N. (2007). Projeto e Algoritmos com implementações em Java e C++. São Paulo. Editora Thomson
- Symbolab (utilizados para gerar alguns gráficos)
 - <https://pt.symbolab.com/graphing-calculator>

Referências de Material

- Adaptado do material de
 - Professor Alessandro L. Koerich da *Pontifícia Universidade Católica do Paraná (PUCPR)*
 - Professor Humberto Brandão da *Universidade Federal de Alfenas (Unifal-MG)*
 - Professor Ricardo Linden da *Faculdade Salesiana Maria Auxiliadora (FSMA)*
 - Professor Antonio Alfredo Ferreira Loureiro da *Universidade Federal de Minas Gerais (UFMG)*