

CURSO TÉCNICO EM INFORMÁTICA

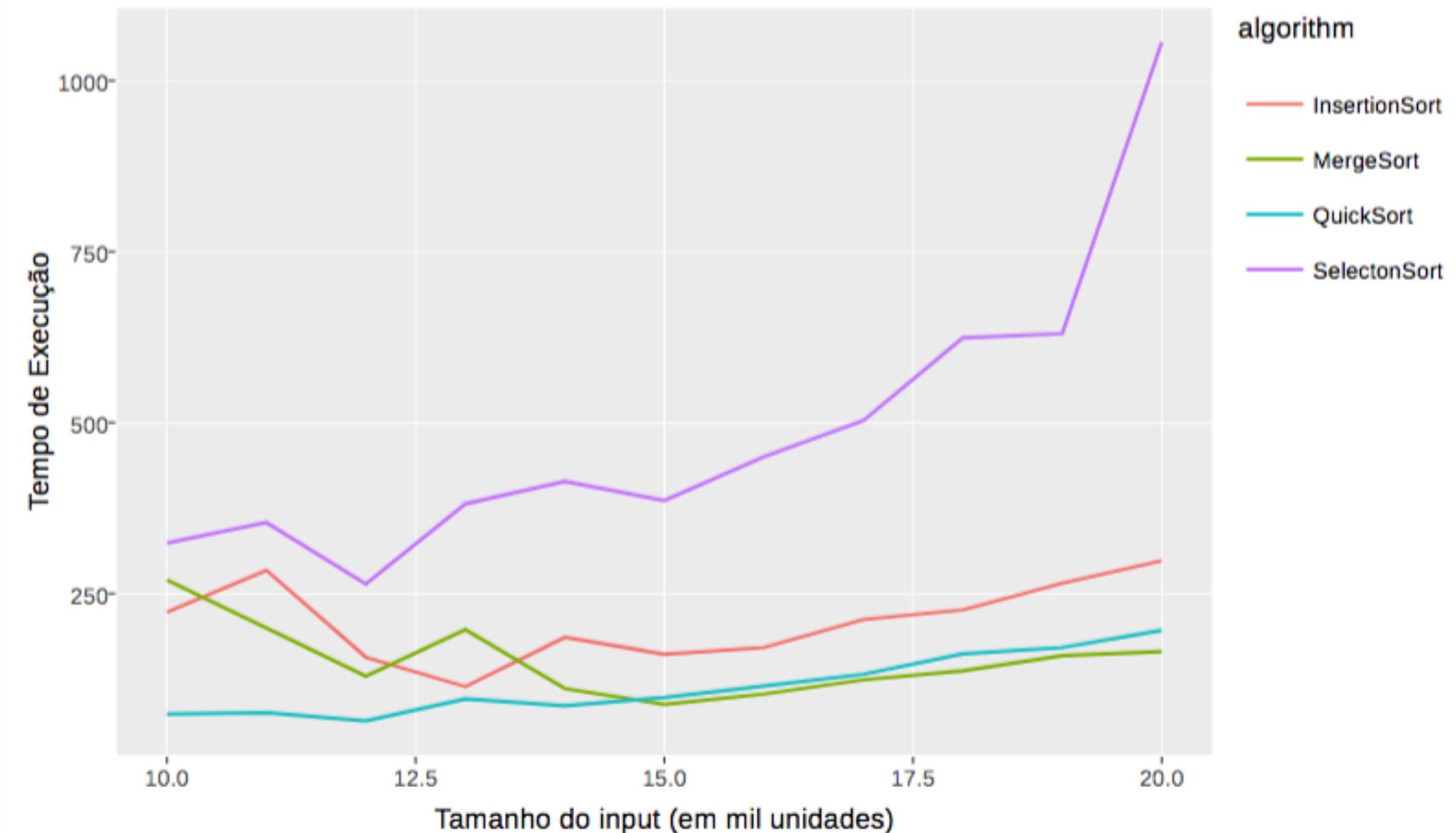
Análise complexa de Algoritmos



Introdução à Análise de Algoritmos

A análise de desempenho é essencial para escolher o **algoritmo** mais adequado, avaliando recursos consumidos, especialmente o tempo de execução. Entre as abordagens, a empírica mede o tempo gasto em função do tamanho da entrada, em ambiente controlado, comparando diferentes soluções.

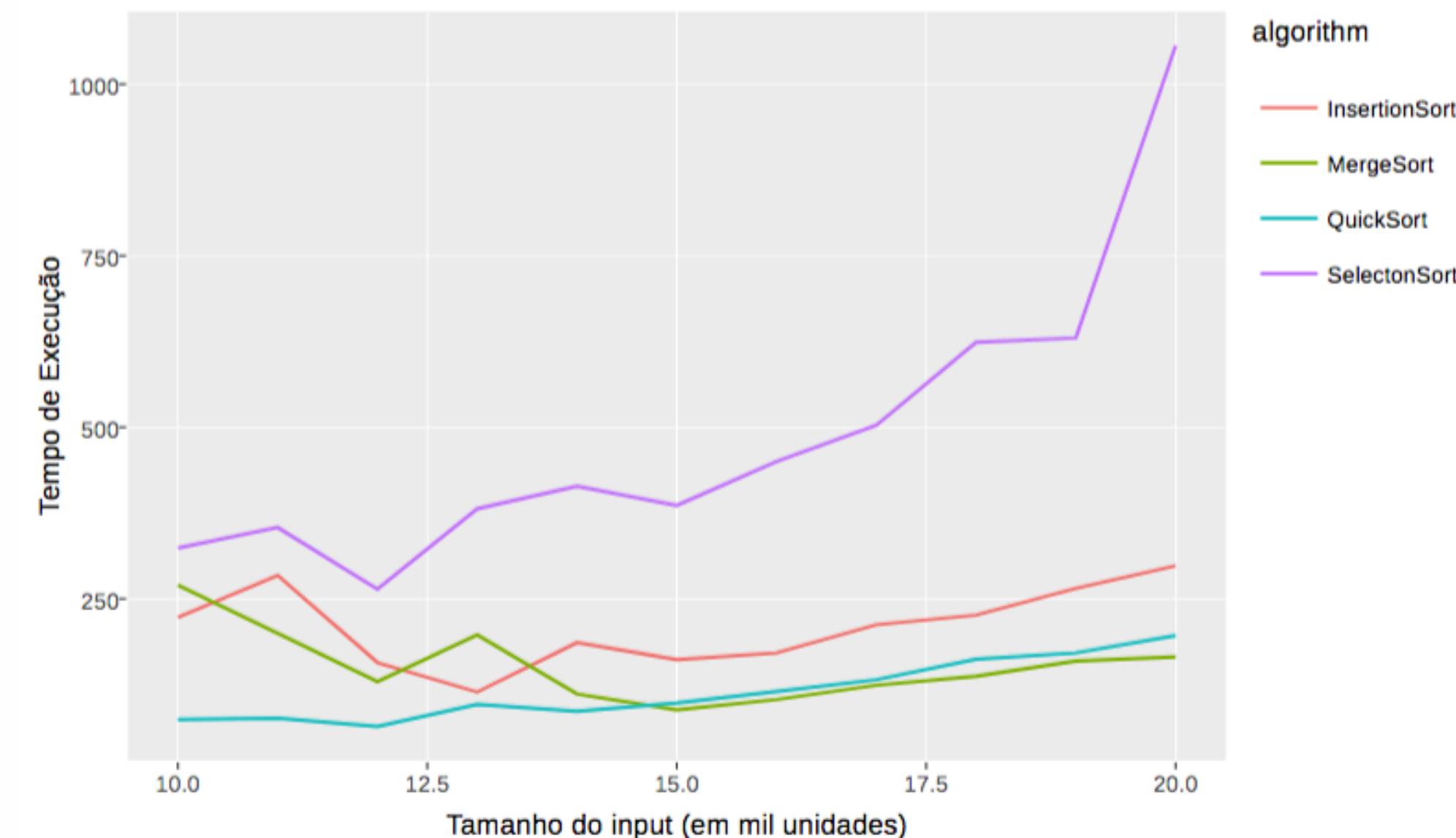
Em experimentos, variam-se condições (como dados já ordenados ou aleatórios) para observar o comportamento do algoritmo.



Introdução à Análise de Algoritmos

Os resultados permitem identificar quais soluções escalam melhor, como no caso do **SelectionSort**, que se torna mais lento com entradas maiores em relação a outros métodos

A análise empírica fornece medições precisas do tempo de execução, mas exige alto custo de implementação, é limitada ao conjunto de entradas testado e depende do hardware. Por isso, busca-se uma abordagem independente de hardware, abrangente e simples, com foco na comparação entre algoritmos e no comportamento para grandes entradas — a análise assintótica.



Análise de Algoritmos

A análise de algoritmos assume que **operações primitivas** — como cálculos aritméticos, acessos a vetores, atribuições, retornos e comparações — têm custo constante, $O(1)$.

Embora esse tempo varie na prática conforme hardware ou linguagem, essa variação é irrelevante para a **análise assintótica**.

Operações Primitivas

- * Avaliação de expressões booleanas (`i >= 2`; `i == 2`, etc);
- * Operações matemáticas (*, -, +, %, etc);
- * Retorno de métodos (`return x;`);
- * Atribuição (`i = 2`);
- * Acesso à variáveis e posições arbitrárias de um array (`v[i]`).



Análise de Algoritmos

Nesse contexto, o tempo de execução de um algoritmo é a soma do custo das operações primitivas. Por exemplo, considere o algoritmo que **multiplica o resto da divisão** de dois inteiros pela parte inteira da mesma divisão:

```
int multiplicaRestoPorParteInteira(int i, int j) {  
    int resto = i % j;          // Calcula o resto da divisão  
    int pInteira = i / j;        // Calcula a parte inteira da divisão  
    int resultado = resto * pInteira; // Multiplica os dois valores  
    return resultado;  
}
```



Análise de Algoritmos

Passo 1: Identificar primitivas. O primeiro passo para determinar de modo analítico o tempo de execução de qualquer algoritmo é identificar todas as operações primitivas. Cada uma, como discutido anteriormente, tem um custo constante. Para o algoritmo acima temos:

1. **atribuição (resto =) -> c₁**
2. **operação aritmética (i % j) -> c₂**
3. **atribuição (plnteira =) -> c₃**
4. **operação aritmética (i / j) -> c₄**
5. **atribuição (resultado =) -> c₅**
6. **operação aritmética (resto * plnteira) -> c₆**
7. **retorno de método (return resultado) -> c₇**



Análise de Algoritmos

```
int multiplicaRestoPorParteInteira(int i, int j) {  
    int resto = i % j;          // Calcula o resto da divisão  
    int pInteira = i / j;        // Calcula a parte inteira da divisão  
    int resultado = resto * pInteira; // Multiplica os dois valores  
    return resultado;  
}
```

Passo 2: Identificar a **quantidade de vezes** que cada uma das primitivas é executada.
Para o algoritmo acima, **todas as primitivas** são executadas apenas uma vez.



CURSO TÉCNICO EM INFORMÁTICA

Análise de Algoritmos

Passo 3: Somar o **custo total**. O tempo de execução do algoritmo é a soma das execuções das **operações primitivas**. Nesse caso temos que a função que descreve o tempo de execução é:

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7$$

Lembrando estamos interessados em uma **função** que nos diga o tempo de execução em relação ao tamanho da entrada. Nesse caso, escolhemos **n** para representar o tamanho da entrada.

Como pode ser visto na função detalhada, o custo não depende de **n** de maneira alguma. Independente dos números passados como parâmetro, o custo será sempre o mesmo. Por isso dizemos que essa **função**, e portanto o algoritmo que é descrito por ela, tem custo constante, ou seja, independe do **tamanho da entrada**.



CURSO TÉCNICO EM INFORMÁTICA

Análise de Algoritmos

Dizer que um algoritmo tem custo constante significa dizer que o seu tempo de execução independe do tamanho da entrada.

Outro fator de destaque é que podemos considerar que todas as **constantes** possuem o mesmo valor **c**. Assim, podemos simplificar a função para:

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7$$

$$f(n) = 7c$$



Análise de Algoritmos: comandos condicionais

O uso de comandos condicionais (como o **if** e o **else**) é muito comum nos algoritmos.

Eles tornam a **análise do tempo** de execução um pouco mais complexa, pois, dependendo da situação, apenas parte do código será executada.

Isso nos leva a uma pergunta importante: qual caminho considerar na análise?

A resposta é: **analisamos o pior caso**.

Neste caso, estamos interessados em entender como o algoritmo se comporta na sua pior situação possível.

Essa escolha é estratégica, pois nos oferece uma visão clara e segura sobre o desempenho.

```
#include <stdio.h>

double precisaNaFinal(double nota1, double nota2, double nota3) {
    double media = (nota1 + nota2 + nota3) / 3.0;

    if (media >= 7.0 || media < 4.0) {
        return 0.0;
    } else {
        double mediaFinal = 5.0;
        double pesoFinal = 0.4;
        double pesoMedia = 0.6;
        double precisa = (mediaFinal - pesoMedia * media) / pesoFinal;

        return precisa;
    }
}
```



CURSO TÉCNICO EM INFORMÁTICA

Análise de Algoritmos: comandos condicionais

```
#include <stdio.h>

double precisaNaFinal(double nota1, double nota2, double nota3) {
    double media = (nota1 + nota2 + nota3) / 3.0;

    if (media >= 7.0 || media < 4.0) {
        return 0.0;
    } else {
        double mediaFinal = 5.0;
        double pesoFinal = 0.4;
        double pesoMedia = 0.6;
        double precisa = (mediaFinal - pesoMedia * media) / pesoFinal;

        return precisa;
    }
}
```

Passo 1. Identificar primitivas.

1. atribuição (media =) -> c_1
2. operação aritmética (nota1 + nota2 + nota3) -> c_2
3. operação aritmética (... / 3) -> c_3
4. avaliação de expressão booleana (media >=7 || media < 4) -> c_4
5. retorno de método (return 0) -> c_5
6. atribuição (mediaFinal =) -> c_6
7. atribuição (pesoFinal =) -> c_7
8. atribuição (pesoMedia =) -> c_8
9. atribuição (precisa =) -> c_9
10. operação aritmética (pesoMedia * media) -> c_{10}
11. operação aritmética (mediaFinal - ...) -> c_{11}
12. operação aritmética (... / pesoFinal) -> c_{12}
13. retorno de método (return precisa) -> c_{13}



Análise de Algoritmos: comandos condicionais

```
#include <stdio.h>

double precisaNaFinal(double nota1, double nota2, double nota3) {
    double media = (nota1 + nota2 + nota3) / 3.0;

    if (media >= 7.0 || media < 4.0) {
        return 0.0;
    } else {
        double mediaFinal = 5.0;
        double pesoFinal = 0.4;
        double pesoMedia = 0.6;
        double precisa = (mediaFinal - pesoMedia * media) / pesoFinal;

        return precisa;
    }
}
```

Passo 2: Identificar a quantidade de vezes que cada uma das **operações primitivas** é executada.

Aqui vem a grande diferença. Como estamos interessados no pior caso, nós vamos descartar a constante **c5**, pois, no **pior caso**, o bloco do **else** será executado, uma vez que é mais custoso que o bloco do **if**. As outras primitivas são executadas apenas uma vez.

Passo 3: Somar o custo total.

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_6 + c_7 + c_8 + c_9 + c_{10} + c_{11} + c_{12} + c_{13}$$

Note que **c5** é desconsiderada.



Análise de Algoritmos: E quando houver iteração?

```
#include <stdio.h>
#include <stdbool.h> // Para usar 'bool', 'true' e 'false'

bool contains(int v[], int tamanho, int n) {
    int i;
    for (i = 0; i < tamanho; i++) {
        if (v[i] == n) {
            return true;
        }
    }
    return false;
}
```

Passo 1: Identificar **operações primitivas**.

1. Atribuição (**int i = 0**) -> c1
2. Avaliação de expressão booleana (**i < v.length**) -> c2
3. Operação aritmética (**i++**) -> c3
4. Avaliação de expressão booleana (**v[i] == n**) -> c4
5. Retorno de método (**return true**) -> c5
6. Retorno de método (**return false**) -> c6



Análise de Algoritmos: E quando houver iteração?

Quando **analisamos um algoritmo** com mais atenção, percebemos que, em algumas expressões, estamos simplificando a contagem das operações básicas (as chamadas primitivas).

No caso da expressão booleana **v[i] == n**, poderíamos pensar que ela é uma única primitiva.

Mas, se formos bem detalhistas, na verdade ela envolve:

1. Acesso ao elemento **v[i]**

2. Acesso à variável **n**

Comparação (**==**) entre os
dois valores Ou seja: **três primitivas.**

```
#include <stdio.h>
#include <stdbool.h> // Para usar 'bool', 'true' e 'false'

bool contains(int v[], int tamanho, int n) {
    int i;
    for (i = 0; i < tamanho; i++) {
        if (v[i] == n) {
            return true;
        }
    }
    return false;
}
```



CURSO TÉCNICO EM INFORMÁTICA

Análise de Algoritmos: E quando houver iteração?

O mesmo vale para uma operação aritmética como $i + j$:

1. Acesso à variável i
2. Acesso à variável j
3. Soma dos dois valores ($+$)

Se fôssemos contar dessa forma para todas as expressões, nossa análise ficaria muito carregada e difícil de acompanhar.

Por isso, por **fins didáticos**, vamos simplificar:

- Toda **expressão booleana** será contada como uma **única primitiva**
- Toda **expressão aritmética** também será contada como uma **única primitiva**

Essa abordagem mantém a análise mais clara, sem “poluir” com detalhes que, nesse momento, não vão mudar a compreensão geral.



Análise de Algoritmos: E quando houver iteração?

Passo 2: Identificar a quantidade de vezes que cada uma das primitivas é executada.

Primeiro, nem todas as **primitivas** serão executadas apenas uma vez.

Depois, precisamos lembrar que estamos considerando o pior caso.

No nosso contexto, o pior caso acontece quando o **array** não contém o número procurado.

Nesse cenário:

- O **algoritmo** precisa percorrer todo o array
- Nenhuma comparação encontra o número
- O retorno **false** só acontece no final

Agora, se o número estivesse presente no array, a execução poderia terminar bem antes, encerrando o laço assim que encontrasse o valor.

Por isso, na nossa análise do pior caso, vamos desconsiderar a primitiva **c5**, pois ela nunca é executada nesse cenário específico.



Análise de Algoritmos: E quando houver iteração?

Passo 2: Identificar a quantidade de vezes que cada uma das primitivas é executada.

Sabendo que o tamanho do vetor é **tamanho**, podemos contar quantas vezes cada primitiva é executada no pior caso:

- $c_1 \rightarrow$ Executada 1 vez
- Exemplo: Inicialização da variável de controle.
- $c_2 \rightarrow$ Executada $(n + 1)$ vezes
 - **Por quê?** A verificação da condição do loop ocorre uma vez a mais do que a quantidade de iterações.
- Exemplo: Se $n = 5$, as verificações são:
 - **Total:** 6 verificações.

```
0 < 5
1 < 5
2 < 5
3 < 5
4 < 5
5 < 5 ← Última verificação, que encerra o loop
```



CURSO TÉCNICO EM INFORMÁTICA

Análise de Algoritmos: E quando houver iteração?

Passo 3: Somar o custo total.

O tempo de execução do algoritmo é a soma das execuções das operações primitivas. Nesse caso temos que a função que descreve o tempo de execução é:

$$f(n) = c_1 + c_2 \cdot (n+1) + c_3 \cdot n + c_4 \cdot n + c_6$$

Se considerarmos que todas as **primitivas** têm o mesmo custo **c** e simplificarmos, teremos:

$$f(n) = 3 \cdot c \cdot n + 3 \cdot c$$

Isso significa que o **tempo de execução** está diretamente ligado ao tamanho do array **n**.

Quanto maior for **n**, maior será o tempo de execução do pior caso.

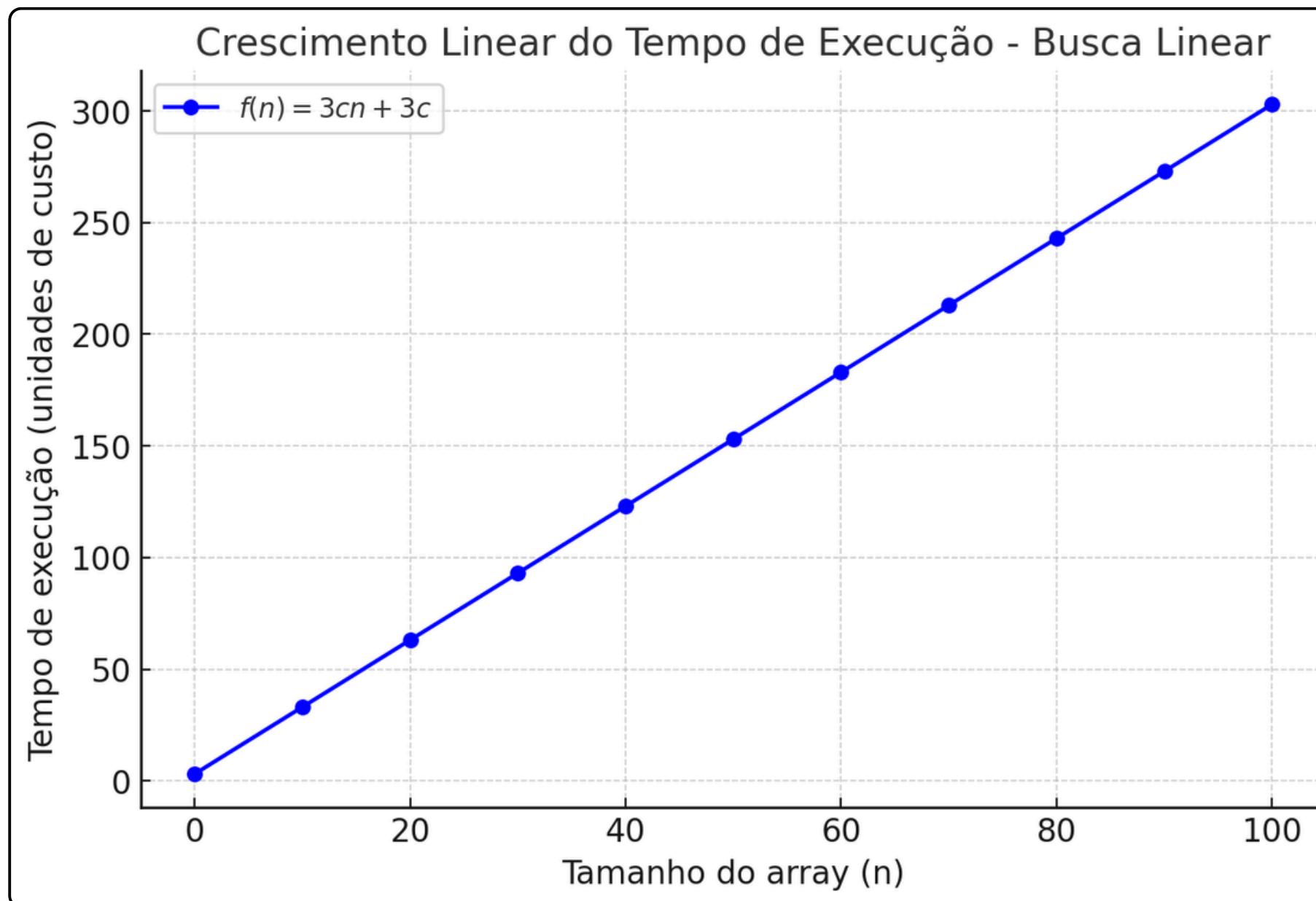
Essa relação é **linear**, pois a função é linear.

Por exemplo: percorrer um array com **100** elementos leva, aproximadamente, 10 vezes mais tempo do que percorrer um **array** com **10** elementos.



Análise de Algoritmos: E quando houver iteração?

Passo 3: Somar o custo total.



$$f(n) = 3 \cdot c \cdot n + 3 \cdot c$$

```
// Função de busca linear
bool contains(int v[], int tamanho, int n) {
    int i;
    for (i = 0; i < tamanho; i++) {
        if (v[i] == n) {
            return true; // c5 (não executa no pior caso)
        }
    }
    return false; // c6
}
```



Análise de Algoritmos: loops aninhados

```
bool contemDuplicacao(int v[], int tamanho) {  
    for (int i = 0; i < tamanho; i++) {  
        for (int j = i + 1; j < tamanho; j++) {  
            if (v[i] == v[j]) {  
                return true; // encontrou duplicação  
            }  
        }  
    }  
    return false; // não encontrou duplicações  
}
```



Análise de Algoritmos: loops aninhados

```
bool contemDuplicacao(int v[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        for (int j = i + 1; j < tamanho; j++) {
            if (v[i] == v[j]) {
                return true; // encontrou dupli
            }
        }
    }
    return false; // não encontrou duplicações
}
```

- c1 → atribuição **int i = 0**
- c2 → teste da condição do primeiro **for (i < v.length)**
- c3 → incremento **i++**
- c4 → declaração **int j** (ligada ao for interno)
- c5 → inicialização de **j = i + 1**
- c6 → teste da condição do segundo for (**j < v.length**)
- c7 → incremento **j++**
- c8 → comparação **v[i] == v[j]**
- c9 → **return true** caso encontre duplicação
- c10 → **return false** se não encontrar nenhuma duplicação



CURSO TÉCNICO EM INFORMÁTICA

Análise do pior caso de execução

O **pior caso** do algoritmo ocorre quando não há duplicação no vetor. Nesse cenário, os dois laços são percorridos até o final, e por isso a primitiva c8 (comparação $v[i] == v[j]$) nunca resulta em verdadeiro e o c9 (retorno true) nunca é executado.

Considerando que o vetor possui tamanho **n**, temos:

- **c1 (atribuição int i = 0)**: Executada 1 vez apenas, na inicialização do laço externo.
- **c2 (condição i < n)**: Executada $n + 1$ vezes.
- Exemplo:

Para **n = 5**, temos as verificações: $0 < 5, 1 < 5, 2 < 5, 3 < 5, 4 < 5$ e $5 < 5 \rightarrow$ total de 6.

- **c3 (incremento i++)**:
Executada n vezes (um incremento ao final de cada iteração do laço externo).



Análise do pior caso de execução

Laço interno

- c4 (declaração de j) e c5 (atribuição $j = i + 1$):
Executadas n vezes, uma vez a cada execução do laço externo.
- c6 (condição $j < n$):
A quantidade de vezes depende do valor de i.
Para cada i, j varia de $i+1$ até $n-1$.
Assim, o número total de execuções de **c6** é dado pela soma:

$$(n) + (n-1) + (n-2) + \dots + 1$$



Análise do pior caso de execução

Laço interno

- c4 (declaração de j) e c5 (atribuição $j = i + 1$):
Executadas n vezes, uma vez a cada execução do laço externo.
- c6 (condição $j < n$):
A quantidade de vezes depende do valor de i.
Para cada i, j varia de $i+1$ até $n-1$.
Assim, o número total de execuções de **c6** é dado pela soma:

$$(n) + (n-1) + (n-2) + \dots + 1$$



Análise do pior caso de execução

Laço interno

No caso:

$$c_6 = 1 + 2 + 3 + \dots + n$$

Essa é uma PA crescente com:

- Primeiro termo: $a_1 = 1$
- Último termo: $a_n = n$
- Número de termos: n

A fórmula da soma de uma PA é:

$$S = \frac{n \cdot (a_1 + a_n)}{2}$$

$$c_6 = \frac{n \cdot (1 + n)}{2} \stackrel{e}{=} \frac{n^2 + n}{2}$$



CURSO TÉCNICO EM INFORMÁTICA

Análise do pior caso de execução

Somar o custo total

O tempo de execução do algoritmo é a soma das execuções das operações primitivas. Nesse caso temos que a função que descreve o tempo de execução é:

$$f(n) = 1 + (n + 1) + n + n + n + \frac{n^2 + n}{2} + \frac{n^2 - n}{2} + \frac{n^2 - n}{2} + 0 + 1 = \frac{3}{2}n^2 + \frac{7}{2}n + 3$$

