1 Introdução

Nosso objetivo é estudar a forma como o tempo de execução de um algoritmo varia em função da instância do problema sobre a qual ele opera. Também é de interesse comparar o desempenho de diferentes algoritmos quando operam sobre a mesma instância de um determinado problema.

2 Desenvolvimento

2.1 Somando os n primeiros números naturais de duas formas diferentes Um primeiro problema com que vamos lidar é um tanto simples: desejamos realizar a soma dos **n** primeiros números naturais. Uma solução um tanto simples envolve a criação de uma estrutura de repetição que faz com que uma variável "contadora" varie de 1 a n, acumulando seu valor a cada iteração. Observe.

```java
public class SomaNNaturaisV1 {
   public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        int soma = 0;
        for (int i = 1; i <= n; i++)
            soma += i;
        System.out.println(soma);
   }
}
```

Nota. Para testar este programa, você pode compilar e depois executar da seguinte forma.

javac SomaNNaturaisV1.java
java SomaNNaturaisV1 100

Professor Rodrigo Bossini <professorbossini@gmail.com>
Produção: 2025

Esse programa pode ser executado em computadores diferentes, utilizando ambientes de interpretação Java (JVM) diferentes, compilados por compiladores diferentes etc. Além disso, caso ele seja executado duas vezes no mesmo computador, o tempo de execução pode variar de acordo com a situação de momento do computador (muitos programas em execução, por exemplo).

O tipo de comparação que desejamos realizar implica em ignorar tais condições. Considere as duas execuções do mesmo algoritmo a seguir. Para simplificar, vamos contar apenas o número de execuções da instrução que realiza o acúmulo do valor.

Execução 1
Nesta execução, vamos considerar que, a cada iteração, a instrução que realiza o acúmulo do valor levou, no máximo, **10 milissegundos** para executar. Assim, o programa levou, no máximo 10 * n milissegundos para terminar.

Execução 2
Nesta execução, vamos considerar que, a cada iteração, a instrução que realiza o acúmulo do valor levou, no máximo, **7 milissegundos** para executar. Assim, o programa levou, no máximo 7 * n milissegundos para terminar.

Na prática, a Execução 2 foi mais rápida. Entretanto, para valores de n **suficientemente grandes**, essa diferença se torna irrelevante. Observe.

Valor de n	**Execução 1**	**Execução 2**
1	1 * 10	1 * 7
100	100 * 10	100 * 7
1.000	1000 * 10	1000 * 7
1.000.000	1.000.000 * 10	1.000.000 * 7

Quando olhamos para valores de n grandes o suficiente, o valor de n é justamente o fator mais importante para determinar o tempo de cada execução, tornando o tempo de execução prático de cada instrução praticamente irrelevante. Quando realizamos tais comparações, estamos interessados essencialmente na **taxa de crescimento do tempo de execução do algoritmo**.

Também podemos utilizar a seguinte propriedade matemática para escrever um algoritmo muito mais eficiente.

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

Observe que, para qualquer valor de **n**, o programa precisa essencialmente realizar uma operação de soma, uma operação de multiplicação e, por fim, uma operação de divisão.

```java
public class SomaNNaturaisV2 {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        int soma = (n * (n + 1)) / 2;
        System.out.println(soma);
    }
}
```

Novamente, vamos considerar duas execuções para esse algoritmo. Digamos que, na primeira, cada operação aritmética leva, no máximo, 15 milissegundos para ser realizada. Na segunda, cada operação leva, no máximo, 27 milissegundos para ser executada.

Valor de n	Execução 1	Execução 2
1	3 * 15	3 * 27
100	3 * 15	3 * 27
1.000	3 * 15	3 * 27
1.000.000	3 * 15	3 * 27

Na prática, a Execução 1 foi mais rápida. Entretanto, consideraremos que cada execução levou tempo constante, que não varia em função de n. Neste caso, consideramos que ambos os programas tiveram tempo de execução “igual”.

2.2 Notações O, Ômega e Teta

As notações O, Ômega e Teta nos permitem expressar o tempo de execução de nossos algoritmos de modo a ocultar as constantes - e também termos - irrelevantes, dando destaque apenas para o termo que **domina** o tempo de execução quando o tamanho da instância sobre a qual estamos operando tende ao infinito. Veja alguns exemplos. A unidade de medida utilizada neles também é irrelevante.

Tempo de execução "real"	Tempo de execução considerado
10	Constante
27	Constante
10000000	Constante
$2n$	n
$10n + 3$	n
$1000n + 500000$	n
$10n^2$	n^2
$10n^2 + 2n$	n^2
$10n^2 + 2n + 10000$	n^2
$n^3 + 1000n^2 + 10n + 5$	n^3
$\log n + 5$	$\log n$
$3 \log n + 10$	$\log n$
$n \log n$	$n \log n$
$2n \log n + 100$	$n \log n$
$5n \log n + n^2$	n^2
2^n	2^n
$3 \cdot 2^n + 10n$	2^n
$2^{n+1} + n^3$	2^n

O tempo de execução de um algoritmo será expresso como uma função que representaremos da seguinte forma.

$$T(n)$$

A forma como $T(n)$ se comporta **conforme o valor de n tende ao infinito** é conhecido como seu **comportamento assintótico**.

Na prática, **o valor de n simboliza algo inerente às características do problema que estamos estudando**. Consideramos que n é o tamanho da instância do problema sendo tratada. Por exemplo, n pode ser o tamanho de um vetor, o número de vértices ou arestas de um grafo e assim por diante. Assim, **uma instância de um problema pode ter tamanho menor, igual ou maior ao tamanho de outra instância do mesmo problema**, por exemplo. Eles são "comparáveis".

As notações

$$O(g(n)) \quad \Omega(g(n)) \quad e \quad \Theta(g(n))$$

representam **conjuntos de funções**. Vamos defini-los formalmente.

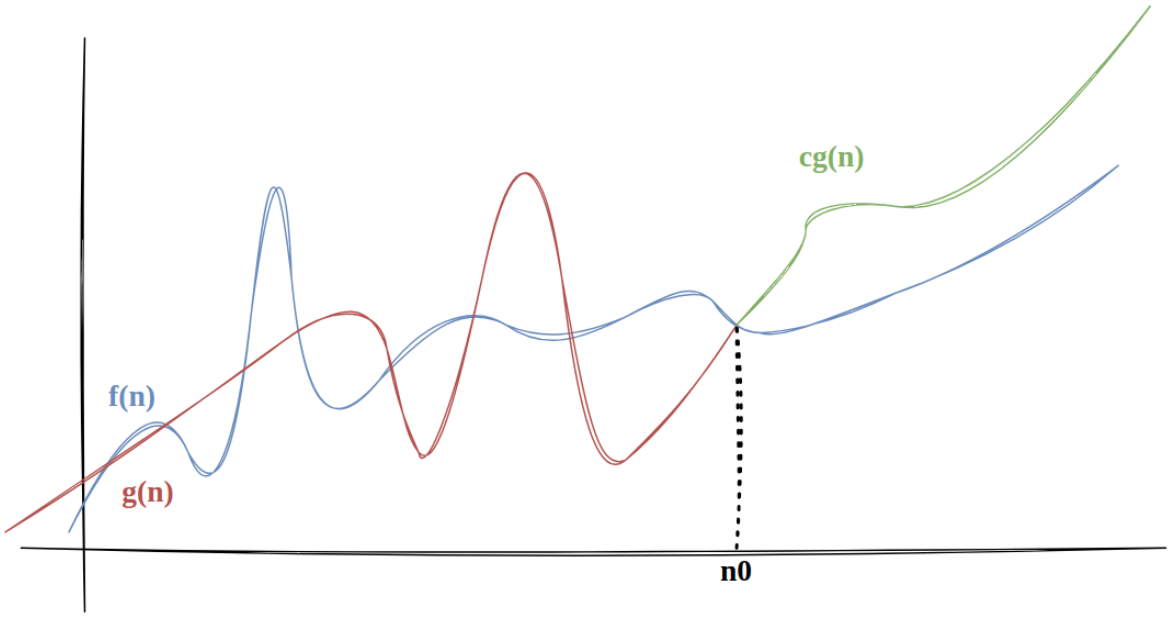
$O(g(n))$ Eis a definição de $O(g(n))$.

$$O(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 \geq 0 \text{ tal que } 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0\}$$

Observe que para que uma função $f(n)$ possa "morar" neste conjunto, $g(n)$ deve servir de **limite assintótico superior** para $f(n)$. Ou seja,

a partir de um determinado valor de n, deve ser possível multiplicar g por uma constante, fazendo com que $f(n)$ não mais a ultrapasse.

Veja isso graficamente:



Observe como, antes do valor $n0$, os comportamentos de f e g podem ser quaisquer entre si. Para que f possa morar no conjunto $O(g(n))$, no entanto, deve ser verdade que, a partir de $n0$, basta multiplicar $g(n)$ por uma constante para que $f(n)$ não a ultrapasse.

Neste caso, podemos dizer que

$$f(n) \in O(g(n))$$

Nota. Tornou-se comum, na literatura, escrever

$$f(n) = O(g(n))$$

em vez de

$$f(n) \in O(g(n))$$

Esta convenção será também adotada neste material.

Nota. O conjunto em que moram todas - e apenas - as funções constantes é comumente denotado por

$$O(1)$$

Veja alguns exemplos.

Exemplo 1 - Big Oh

Sejam as funções.

$$f(n) = 5, \quad g(n) = 1$$

Tomando

$$c = 5, \quad n_0 = 1$$

Temos

$$5 \leq 5 \cdot 1, \quad \forall n \geq 1$$

Exemplo 2 - Big Oh

Sejam as funções.

$$f(n) = 2, \quad g(n) = n$$

Tomando

$$c = 2, \quad n_0 = 1$$

Temos

$$2 \leq 2 \cdot n, \quad \forall n \geq 1$$

Exemplo 3 - Big Oh

Sejam as funções.

$$f(n) = 2n + 10, \quad g(n) = n$$

Tomando

$$c = 3, \quad n_0 = 10$$

Temos

$$2n + 10 \leq 3n, \quad \forall n \geq 10$$

Exemplo 4 - Big Oh

Sejam as funções.

$$f(n) = 3n + 15, \quad g(n) = n^2$$

Tomando

$$c = 1, \quad n_0 = 6$$

Temos

$$3n + 15 \leq 1 \cdot n^2, \quad \forall n \geq 6$$

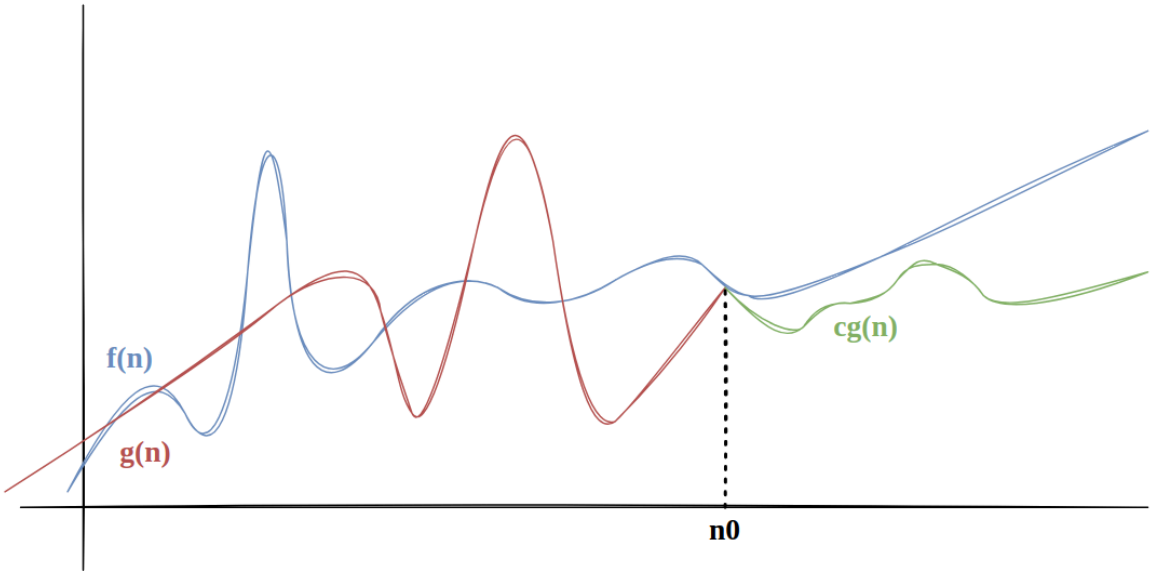
$\Omega(g(n))$ Eis a definição de $\Omega(g(n))$.

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 \geq 0 \text{ tal que } 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0\}$$

Observe que para que uma função $f(n)$ possa "morar" neste conjunto, $g(n)$ deve servir de limite assintótico inferior para $f(n)$. Ou seja,

a partir de um determinado valor de n, deve ser possível multiplicar g por uma constante, fazendo com que seu valor seja, no máximo, igual ao de $f(n)$.

Veja graficamente.



Neste caso, podemos dizer que

$$f(n) \in \Omega(g(n))$$

Ou, ainda, como é mais comum

$$f(n) = \Omega(g(n))$$

Veja alguns exemplos.

Exemplo 1 - Omega

Sejam as funções.

$$f(n) = n, \quad g(n) = 1$$

Tomando

$$c = 1, \quad n_0 = 1$$

Temos

$$1 \cdot 1 \leq n, \quad \forall n \geq 1$$

Exemplo 2 - Omega

Sejam as funções.

$$f(n) = n^2, \quad g(n) = n$$

Tomando

$$c = 1, \quad n_0 = 1$$

Temos

$$1 \cdot n \leq n^2, \quad \forall n \geq 1$$

Exemplo 3 - Omega

Sejam as funções.

$$f(n) = n^2 - 2n, \quad g(n) = n^2$$

Tomando

$$c = \frac{1}{2}, \quad n_0 = 4$$

Temos

$$\frac{1}{2} \cdot n^2 \leq n^2 - 2n, \quad \forall n \geq 4$$

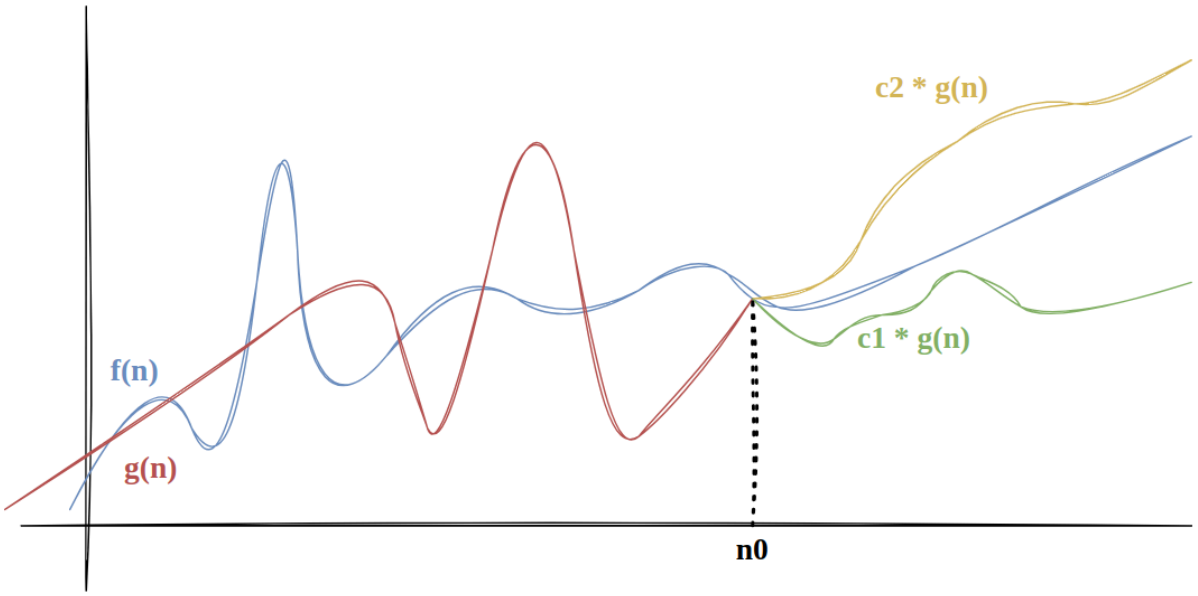
$\Theta(g(n))$ Eis a definição de $\Theta(g(n))$.

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, \exists n_0 \geq 0 \text{ tal que } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0\}$$

Observe que para que uma função $f(n)$ possa "morar" neste conjunto, $g(n)$ deve servir de **limite assintótico inferior e superior** para $f(n)$. Ou seja,

a partir de um determinado valor de n, deve ser possível multiplicar g por duas constantes, c_1 e c_2 **fazendo com que** $c_1 g(n) \leq f(n) \leq c_2 g(n)$**.**

Veja graficamente.



Neste caso, podemos dizer que

$$f(n) \in \Theta(g(n))$$

Ou, ainda, como é mais comum

$$f(n) = \Theta(g(n))$$

Veja alguns exemplos.

Exemplo 1 - Theta

Sejam as funções.

$$f(n) = 5, \quad g(n) = 1$$

Tomando

$$c_1 = 4, \quad c_2 = 6, \quad n_0 = 1$$

Temos

$$4 \cdot 1 \leq 5 \leq 6 \cdot 1, \quad \forall n \geq 1$$

Exemplo 2 - Theta

Sejam as funções.

$$f(n) = 3n + 7, \quad g(n) = n$$

Tomando

$$c_1 = 2, \quad c_2 = 4, \quad n_0 = 1$$

Temos

$$2n \leq 3n + 7 \leq 4n, \quad \forall n \geq 7$$

Exemplo 3
Sejam as funções. $$f(n) = 2n^2 + 5n + 10, \quad g(n) = n^2$$ Tomando $$c_1 = 2, \quad c_2 = 3, \quad n_0 = 1$$ Temos $$2n^2 \leq 2n^2 + 5n + 10 \leq 3n^2, \quad \forall n \geq 7$$

2.3 Analisando o tempo de execução dos algoritmos de soma de naturais Nesta seção, vamos determinar o tempo de execução dos algoritmos de soma dos primeiros n números naturais apresentados anteriormente.

O primeiro algoritmo faz a soma utilizando uma estrutura de repetição. Veja, a seguir, uma análise mais precisa de seu tempo de execução.

Linha	Vezes que executa	Tempo de execução
int n = Integer.parseInt(args[0]);	1	c1
int soma = 0;	1	c2
for (int i = 1; i <= n; i++)	n + 1	c3 + c4 + c5
soma += i;	n	c6
System.out.println(soma);	1	c7

O total é

$$T(n) = 1 \cdot c_1 + 1 \cdot c_2 + c_3 \cdot n + c_3 \cdot 1 + c_4 \cdot n + c_4 \cdot 1 + c_5 \cdot n + c_5 \cdot 1 + n \cdot c_6 + 1 \cdot c_7$$

Observe que estamos diante de uma equação linear que pode ser expressa como

$$T(n) = an + b$$

Assim, é verdade que

$$T(n) = \Theta(n)$$

Agora vamos analisar o tempo de execução do segundo algoritmo.

Linha	Vezes que executa	Tempo de execução
int n = Integer.parseInt(args[0]);	1	c_1
int soma = (n * (n + 1)) / 2;	1	c_2
System.out.println(soma);	1	c_3

Assim,

$$T(n) = c_1 + c_2 + c_3$$

E, portanto

$$T(n) = \Theta(1)$$

2.4 Analisando o tempo de execução de algoritmos diversos Nesta seção, vamos analisar o tempo de execução de mais alguns algoritmos.

Cálculo de juros simples O programa a seguir faz o cálculo de juros simples.

```
public class JurosSimples {
   public static void main(String[] args) {
       double capitalInicial = Double.parseDouble(args[0]);
       double taxaJurosAnual = Double.parseDouble(args[1]);
       int tempoAnos = Integer.parseInt(args[2]);
       double montanteFinal = capitalInicial + (capitalInicial * taxaJurosAnual * tempoAnos);
       System.out.println("Montante final após " + tempoAnos + " anos: R$" + montanteFinal);
   }
}
```

Observe que, cada linha, executa uma única vez. Não há variação alguma em função do tamanho da entrada. Poderíamos imaginar que cada instância tem seu tamanho caracterizado por três variáveis: o valor de capital inicial, a taxa de juros inicial e o tempo em anos. Assim,

$$T(c, tx, tp) = O(1)$$

Valor máximo de um vetor O programa a seguir encontra o valor máximo de um vetor. Observe que ele utiliza duas estruturas de repetição **sem aninhamento**.

```java
public class MaiorValor {
    public static void main(String[] args) {
        int n = args.length;
        int[] valores = new int[n];

        for (int i = 0; i < n; i++) {
            valores[i] = Integer.parseInt(args[i]);
        }

        int maior = valores[0];

        for (int i = 1; i < n; i++) {
            if (valores[i] > maior) {
                maior = valores[i];
            }
        }

        System.out.println("Maior valor: " + maior);
    }
}
```

Qual a sua complexidade?

O número de iterações do primeiro for é proporcional a n. Assim como acontece com o segundo. Assim,

$$T(n) = \Theta(n) + \Theta(n) = \Theta(n)$$

Matriz identidade O programa a seguir lê a ordem de uma matriz quadrada e exibe uma matriz identidade.

```
public class MatrizIdentidade {
   public static void main(String[] args) {
       int n = Integer.parseInt(args[0]);

       for (int i = 0; i < n; i++) {
           for (int j = 0; j < n; j++) {
               if (i == j) {
                   System.out.print("1 ");
               } else {
                   System.out.print("0 ");
               }
           }
           System.out.println();
       }
   }
}
```

Observe que o número de iterações do for externo é proporcional a n. A cada iteração, o for interno também executa uma quantidade de iterações proporcional a n. Temos n iterações externas e cada uma custa n iterações. Assim,

$$T(n) = \underbrace{\Theta(n) + \Theta(n) + \cdots + \Theta(n)}_{\text{n vezes}} = \Theta(n^2)$$

Barra de progresso O programa a seguir lê n valores inteiros e produz uma sequência de barras de progresso. Para a coleção

2 3 4 1 5

por exemplo, a sua saída é

```
[1] **
[2] ***
[3] ****
[4] *
[5] *****
```

```java
public class BarraDeProgresso {
   public static void main(String[] args) {
        int n = args.length;
        int[] valores = new int[n];

        for (int i = 0; i < n; i++) {
            valores[i] = Integer.parseInt(args[i]);
        }


        for (int i = 0; i < n; i++) {
            System.out.print("[" + (i + 1) + "] ");
            for (int j = 0; j < valores[i]; j++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Determinar a complexidade deste algoritmo depende da forma como definimos as instâncias do problema e seu tamanho.

Por exemplo, podemos considerar que o tamanho de uma instância é dado em função apenas de **n**.

Neste caso, os valores contidos na coleção são todos constantes.

Assim, se a constante **k** é a maior entre aquelas contidas na coleção, o for interno executa, no máximo, **k** iterações.

O for externo executa **n** iterações. Cada uma, portanto, custa no máximo $k = O(1)$. A ideia descrita aqui se assemelha à definição dada para uma técnica chamada **loop unrolling**.

Há, também, um for que apenas lê os valores. Assim,

$$T(n) = O(n) + \underbrace{O(1) + O(1) + \cdots + O(1)}_{\text{n vezes}} = O(n)$$

Observe, entretanto, que a definição do problema e a forma como o tamanho de suas instâncias é caracterizado é fundamental. Os valores contidos na coleção são arbitrários, podendo, inclusive, serem maiores do que n. Assim, uma definição que envolva tais valores pode ser mais interessante. Suponha que **k** é o maior valor contido na coleção. Sendo **k** um valor que varia de maneira independente de **n**, sabemos novamente que cada iteração do loop externo custa **k** iterações do loop interno. Assim,

$$T(n,k) = O(n) + \underbrace{O(k) + O(k) + \cdots + O(k)}_{\text{n vezes}} = O(kn)$$

Contagem de divisões por 2 O programa a seguir divide um número por 2 sucessivas vezes, até "exauri-lo", ou seja, enquanto ele for maior do que 1.

```
public class DivisaoPorDois {
   public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        int passos = 0;

        while (n > 1) {
            n /= 2;
            passos++;
        }

        System.out.println(passos);
   }
}
```

Quantas vezes podemos dividir um número por 2 até "esgotá-lo". Veja a tabela a seguir, em que utilizamos potências de para simplificar, muito embora o raciocínio se aplique a valores quaisquer.

n	Quantidade de divisões
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10

Por inspeção, concluímos que o número procurado é

$$\log_2 n$$

Nota. Dado que

$$\log_b x = \frac{\log_c x}{\log_c b}$$

Veja um exemplo

$$\log_2 x = \frac{\log_{10} x}{\log_{10} 2}$$

consideramos que a base de logaritmos é irrelevante na análise de complexidade de algoritmos.

Assim,

$$T(n) = O(\log n)$$

Referências

CORMEN, Thomas H. et al. **Introduction to Algorithms**. 3. ed. Cambridge: MIT Press, 2009.

FEOFILOFF, Paulo. **Análise de Algoritmos**. Disponível em: https://www.ime.usp.br/~pf/analise_de_algoritmos/. Acesso em: março de 2025.

KLEINBERG, Jon; TARDOS, Éva. **Algorithm Design**. Boston: Pearson, 2006.