



**FAETERJ - Paracambi**

**Curso: Análise e Desenvolvimento de Sistemas**

**Disciplina: Estrutura de Dados      Prof: Carlos Eduardo**

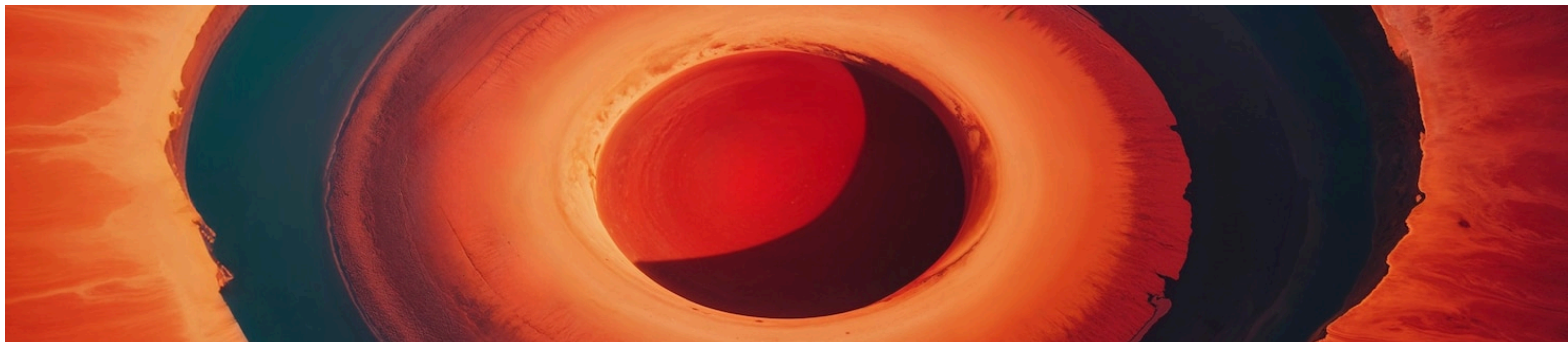
# **Complexidade de Algoritmos**

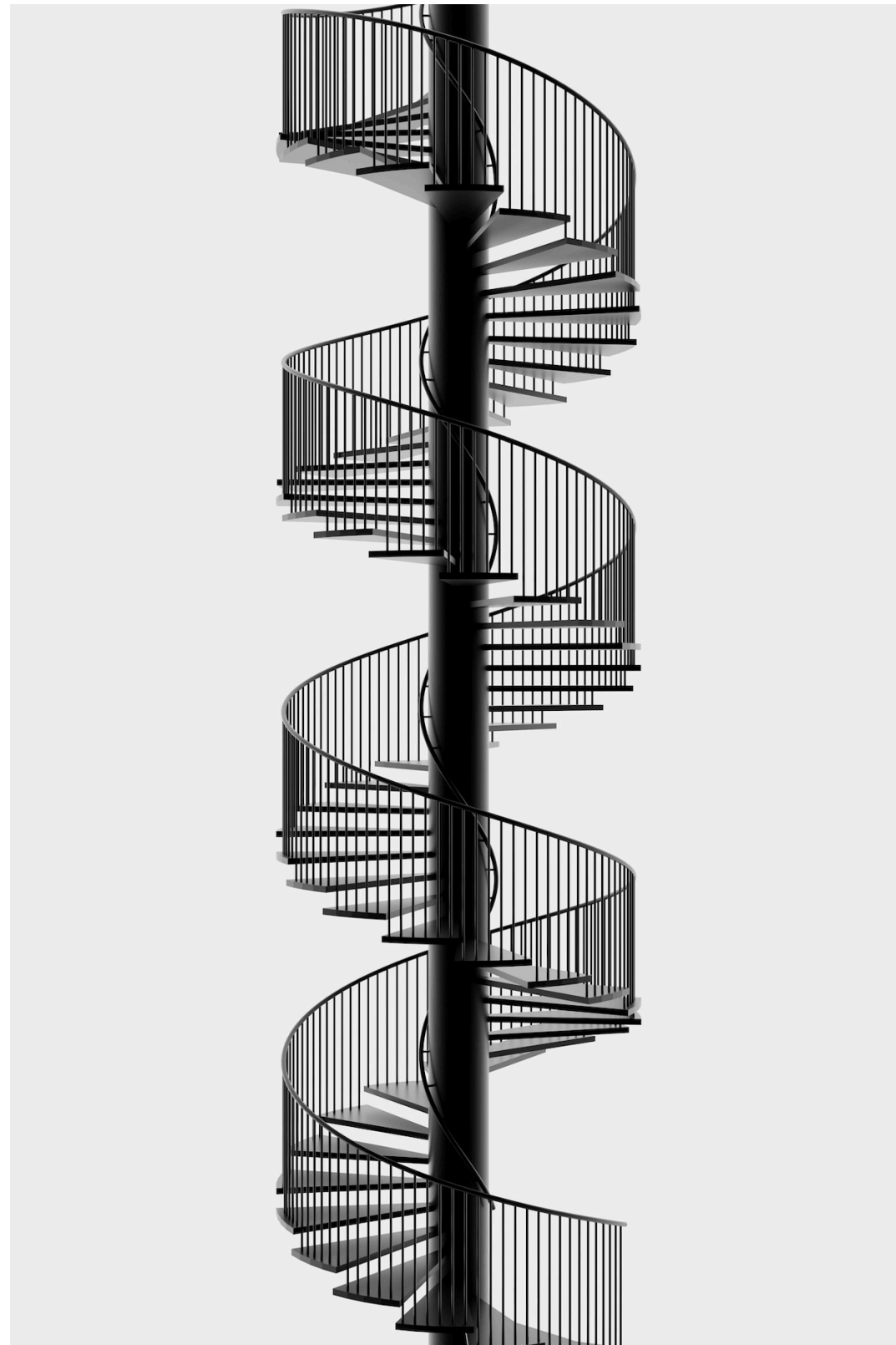
**Alunos: Isabelle Gonçalves da Silva**

**Vinícius Aniceto dos Santos**



- Definição da Sequência de Fibonacci
- Aplicações da Sequência de Fibonacci
- Algoritmos implementados com explicação e respectivas complexidades
- Comparações do tempo de processamento em tabelas e gráficos
- Conclusão
- Referências Bibliográficas





# Definição da Sequência de Fibonacci

A sequência de Fibonacci começa com os números 0 e 1. O terceiro número na sequência são os dois primeiros números somados ( $0 + 1 = 1$ ). O quarto número na sequência é o segundo e o terceiro números somados ( $1 + 1 = 2$ ). Cada número sucessivo é a adição (a soma) dos dois números anteriores na sequência.

A sequência acaba ficando assim:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...



# Aplicações da Sequência de Fibonacci

É aplicada em análises financeiras e na informática, sendo utilizada por Da Vinci, que chamou a sequência de Divina Proporção, para fazer desenhos perfeitos.

A arte, o design e a arquitetura se inspiram para criar desde obras de arte a construções.

No campo da matemática e computação, ela contribui com aplicações na teoria dos números, estudos de sequências, algoritmos e ciência da computação.

Na Biologia a sequência descreve padrões de crescimento em plantas e animais.



```
#include <stdio.h>
#include <time.h> // Biblioteca para funções de tempo
long int Fibl(int n) { // Função para calcular o número de Fibonacci usando recursão
    if (n < 2) // Caso base: se n é 0 ou 1, retorna n
        return n;
    else // Caso recursivo: retorna a soma dos dois números anteriores
        return Fibl(n - 1) + Fibl(n - 2);
}
int main() {
    int n;
    long int resultadol; // Variável para armazenar o resultado de Fibonacci
    clock_t inicio, fim; // Variáveis para armazenar o tempo inicial e final
    double tempo_gasto; // Variável para armazenar o tempo gasto

    printf("Digite um número para calcular o Fibonacci: ");
    scanf("%d", &n);

    inicio = clock(); // Registra o tempo inicial
    resultadol = Fibl(n); // Calcula o número de Fibonacci
    fim = clock(); // Registra o tempo final
    tempo_gasto = (double)(fim - inicio) / CLOCKS_PER_SEC; // Calcula o tempo gasto em segundos

    printf("Fibonacci de %d é (Complexidade O(2^n)): %ld\n", n, resultadol);
    printf("Tempo de Processamento (Complexidade O(2^n)): %lf segundos\n", tempo_gasto);
    return 0;
}
```



```
#include <stdio.h>
#include <time.h> // Biblioteca para manipulação de tempo
long int Fib2 (int n) { // Função para calcular o n-ésimo número de Fibonacci de forma iterativa
    int i = 1, j = 0, k; // Inicializa as variáveis i e j
    for (k = 1; k <= n; k++) { // Loop para calcular o n-ésimo número de Fibonacci
        i = i + j; // Atualiza i com a soma de i e j
        j = i - j; // Atualiza j com o valor anterior de i
    }
    return j; // Retorna o n-ésimo número de Fibonacci
}

int main() {
    int n;
    long int resultado2; // Variável para armazenar o resultado do cálculo de Fibonacci
    clock_t inicio, fim; // Variáveis para medir o tempo de execução
    double tempo_gasto; // Variável para armazenar o tempo gasto

    printf("Digite um número para calcular o Fibonacci: ");
    scanf("%d", &n);
    inicio = clock(); // Marca o tempo de início
    resultado2 = Fib2(n); // Calcula o n-ésimo número de Fibonacci
    fim = clock(); // Marca o tempo de fim
    tempo_gasto = (double)(fim - inicio) / CLOCKS_PER_SEC; // Calcula o tempo gasto em segundos
    printf("Fibonacci de %d é (Complexidade O(n)): %ld\n", n, resultado2);
    printf("Tempo de processamento (Complexidade O(n)): %f segundos\n", tempo_gasto);
    return 0;
}
```

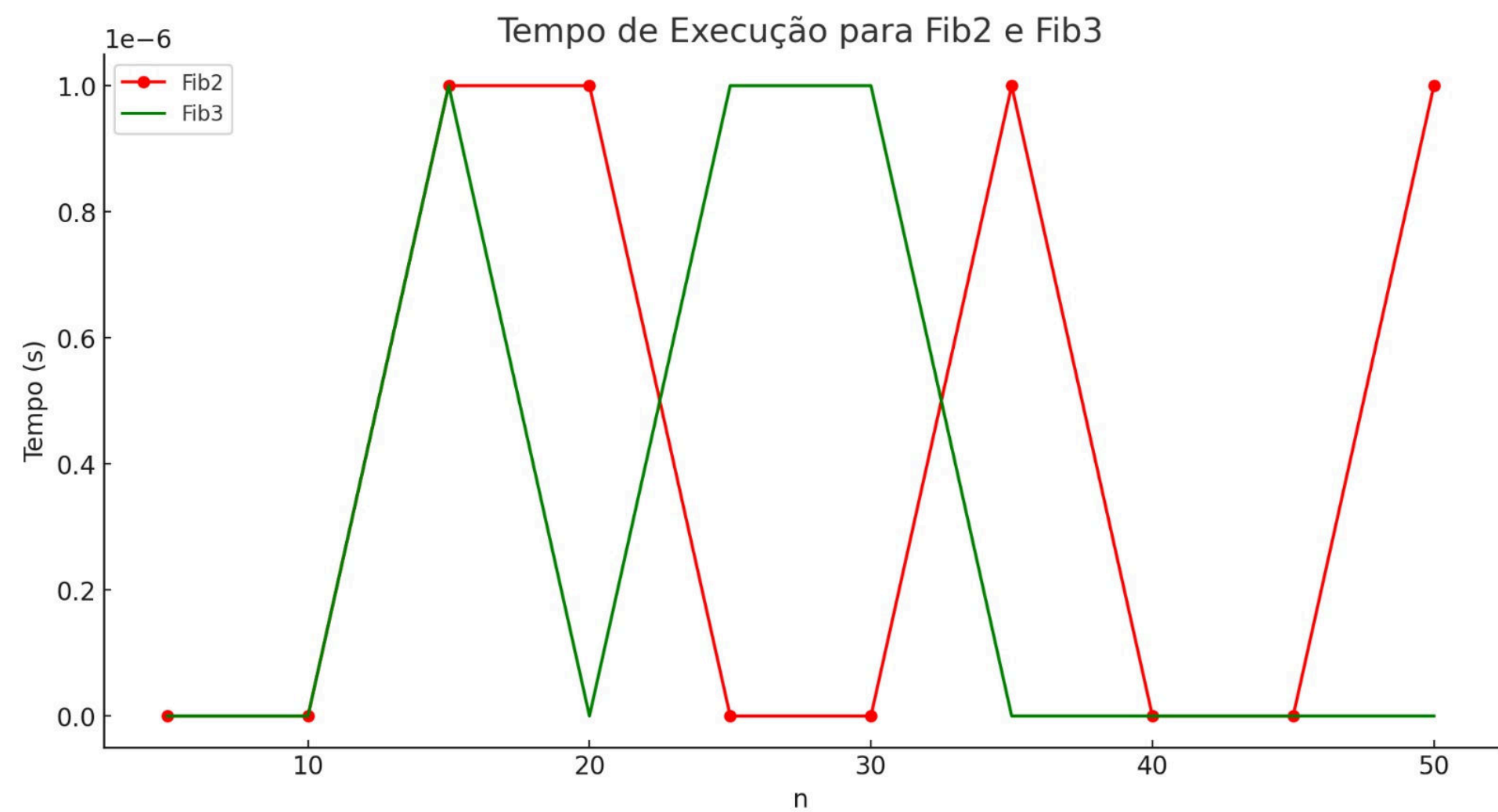
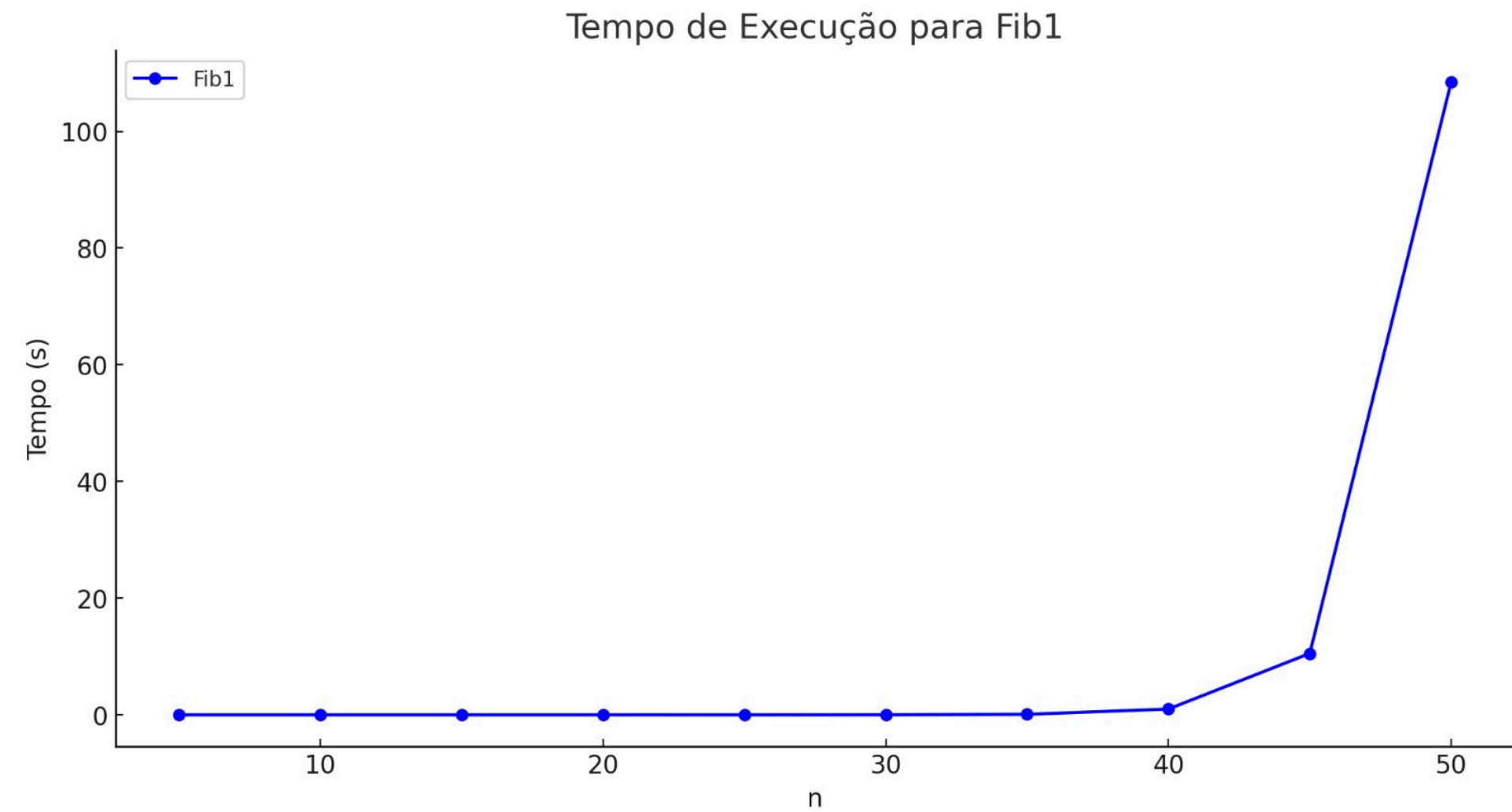
```

#include <stdio.h>
#include <time.h> // Biblioteca para manipulação de tempo
long int Fib3(int n) // Função que calcula o n-ésimo número de Fibonacci usando um método iterativo
    int i = 1, j = 0, k = 0, h = 1, t; // Declaração e inicialização das variáveis
    while (n > 0) { // Enquanto n for maior que 0
        if (n % 2 == 0) { // Se n é par
            t = 2 * h * j + k * k; // Calcula t com base nos valores atuais de h e k
            h = h * h + k * k; // Atualiza h para o próximo valor na sequência
            k = t; // Atualiza k para o próximo valor na sequência
            n = n / 2; // Divide n por 2
        } else { // Se n é ímpar
            t = j * h + i * k; // Calcula t com base nos valores atuais de j, h e k
            j = i * h + j * k + j * t; // Atualiza j para o próximo valor na sequência
            i = i * k + j * t; // Atualiza i para o próximo valor na sequência
            n = n - 1; // Decrementa n
        } return j; // Retorna o n-ésimo número de Fibonacci
    }
}

int main() {
    int n;
    long int resultado3; // Variável para armazenar o resultado do cálculo de Fibonacci
    clock_t inicio, fim; // Variáveis para armazenar o tempo de início e fim
    double tempo_gasto; // Variável para armazenar o tempo gasto
    printf("Digite um numero para calcular o Fibonacci: ");
    scanf("%d", &n);
    inicio = clock(); // Marca o tempo de início
    resultado3 = Fib3(n); // Calcula o n-ésimo número de Fibonacci
    fim = clock(); // Marca o tempo de fim
    tempo_gasto = (double)(fim - inicio) / CLOCKS_PER_SEC; // Calcula o tempo gasto em segundos
    printf("Fibonacci de %d é (Complexidade O(log(n))): %ld\n", n, resultado3);
    printf("Tempo de PROCESSAMENTO (Complexidade O(log(n))): %lf segundos\n", tempo_gasto);
    return 0; }

```

n	T(Fib1)	T(Fib2)	T(Fib3)
5	0.000000	0.000000	0.000000
10	0.000002	0.000000	0.000000
15	0.000008	0.000001	0.000001
20	0.000077	0.000001	0.000000
25	0.001067	0.000000	0.000001
30	0.008381	0.000000	0.000001
35	0.096364	0.000001	0.000000
40	1.002.202	0.000000	0.000000
45	10.533.197	0.000000	0.000000
50	108.439.362	0.000001	0.000000







# Conclusão

A análise dos tempos de processamento confirmou que a complexidade do algoritmo impacta diretamente a eficiência:


Complexidade Exponencial ( $O(2^n)$ ): Tempo de execução cresce rapidamente, tornando-se inviável para entradas maiores.

Complexidade Linear ( $O(n)$ ): Crescimento proporcional ao tamanho da entrada, apresentando boa escalabilidade.

Complexidade Logarítmica ( $O(\log n)$ ): Tempo de execução aumenta lentamente, mesmo com grandes entradas, sendo altamente eficiente.

A escolha do algoritmo certo, com uma complexidade adequada ao problema, é essencial para garantir um desempenho eficiente.

Os testes evidenciam a importância de considerar a complexidade ao desenvolver soluções para problemas computacionais.





## Referências Bibliográficas



- TODA MATÉRIA. Sequência de Fibonacci. Disponível em: <https://www.todamateria.com.br/sequencia-de-fibonacci/>. Acesso em: 20 out. 2024.
- ESTUDYANDO. Sequência de Fibonacci: exemplos, proporção áurea e natureza. Disponível em: <https://pt.studyando.com/sequencia-de-fibonacci-exemplos-proporcao-aurea-e-natureza/>. Acesso em: 20 out. 2024.

