# Aula Prática 3 – Velocidade de Algoritmos Recursivos e Iterativos

**Dante Junqueira Pedrosa**

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
Belo Horizonte – MG – Brasil

dantepedrosa@ufmg.br

## 1. Introdução

Este relatório tem como objetivo apresentar resultados da comparação de tempos de execução entre algoritimos que realizam as mesmas operações, mas implentações distintas. Dentro do escopo deste relatório, foram consideradas duas implementações diferentes para cada uma das seguintes operações:

- **Tarefa 1:** Cáclulo de Fatorial – Implementação Recursiva e Iterativa
- **Tarefa 2:** Cálculo de Fibonacci – Implementação Recursiva e Iterativa

Nas seções a seguir, serão apresentados um contexto teórico, tal como a metodologia utilizada e os resultados de tempo de execução encontrados nas simulações.

## 2. Contexto Teórico

Como descrito acima, a comparação será entre algoritmos similares, sendo comparada uma versão iterativa com outra recursiva.

**Implementação Recursiva:** Uma implementação de método recursivo é uma técnica comumente usada para melhor representação visual do código, que permite melhor entendimento. Se baseia no princípio de uma função ou método que chama a si mesmo dentro de seu código, com parâmetros diferentes, invocando uma nova instância, que também irá chamar a si mesmo. Para que esta implementação seja possível, devem ser estabelecida uma condição base que caso seja alcançada, não irá abrir uma nova recursão e irá encerrar a função. Isto impede que erros como *stack-overflow* (atingir limite de utilização de memória de pilha) ocorram.

**Implementação Iterativa:** Por outro lado, uma implementação de método iterativo é uma abordagem onde não são chamadas múltiplas instâncias de uma mesma função, mas as operações ocorrem em um loop finito dentro da própria função. Este método, apesar ser no geral de mais difícil visualização e compreensão do código, evita o uso excessivo da memória pelo programa.

Apesar da grande diferença no desempenho das implementações serem no quesito da memória utilizada, o foco desta atividade é avaliar a velocidade de execução dos códigos, sem análises extras de uso de memória.

# 3. Metodologia

## 3.1. Descrição de Hardware e Software

Para a prática, foi definido a utilização da linguagem C em sua versão “gcc (Ubuntu 13.2.0-23ubuntu4) 13.2.0”. O computador e sistema operacional usado nos experimentos possuem as seguintes especificações:

| | |
|---|---|
| Fabricante/Modelo | Acer Aspire 3820 |
| Processador | 4 × Intel® Core™ i3 CPU M 380 @ 2.53GHz |
| Memória RAM | 7,4 GiB DDR3 |
| Placa de Vídeo | Mesa Intel® HD Graphics |
| Sistema Operacional | Kubuntu 24.04 (Linux) |
| Versão do Kernel | 6.8.0-48-generic (64-bit) |

De forma a estar de acordo com a metodologia científica, apenas o algoritmo usado na função de cálculo foi alterado, sendo mantido em sua forma mais eficiente e simples possível. Todas as outras operações como leitura e escrita de linha de comando e arquivo foram mantidas de forma a não interferir a medição de tempo de execução.

## 3.2. Algoritmos Usados

### 3.2.1. Algoritimo de Fibonacci – Recursivo e Iterativo

Para encontrar o n-ésimo número da sequeência Fibonacci, foram utilizadas as seguintes versões de código:

```c
int rec_fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    return rec_fibonacci(n - 1) + rec_fibonacci(n - 2);
}
```

Figura 3.2.1 – Código recursivo da sequência Fibonacci

```c
int it_fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    int a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

Figura 3.2.2 – Código iterativo da sequência Fibonacci

### 3.2.2. Algoritmo de Fatorial – Recursivo e Iterativo

Para o algoritmo para calcular o fatorial de um número, foram utilizadas as seguintes versões:

```c
int rec_factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * rec_factorial(n - 1);
}
```

Figura 3.2.2 – Código recursivo da sequência Fibonacci

```c
int it_factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

Figura 3.2.4 – Código iterativo para cáculo de fatorial

## 3.3. Cáclulo de Tempo

O calculo de tempo foi realizado de forma simples, sendo registrado o tempo da CPU imediatamente antes da chamada da função e comparado com o tempo da CPU imediatamente após o retorno para a main. Para o teste foi realizado um loop de valores de entrada de 1 a 30, tanto para o Fibonacci quanto para fatorial e seus resultados foram salvos em um arquivo de texto.

Para que houvesse menor interferência possível de outros fatores, o computador foi reiniciado e cada rodada era realizada imediatamente após a outra, sem outros softwares abertos, diretamente do terminal. Os 3 resultados mais próximos foram selecionados para compor uma média de tempo gasto.

# 4. Resultados

## 4.1. Fatorial

A seguir se encontram os resultados dos testes realizados para Fatorial.

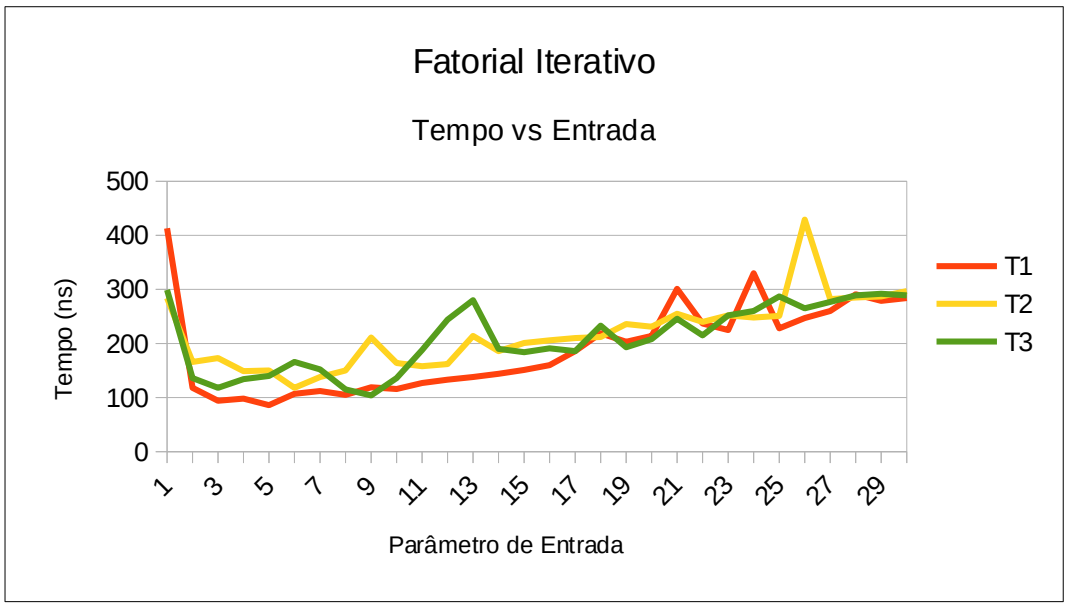


Gráfico 4.1 – Tempo gasto na Versão Iterativa de Fatorial

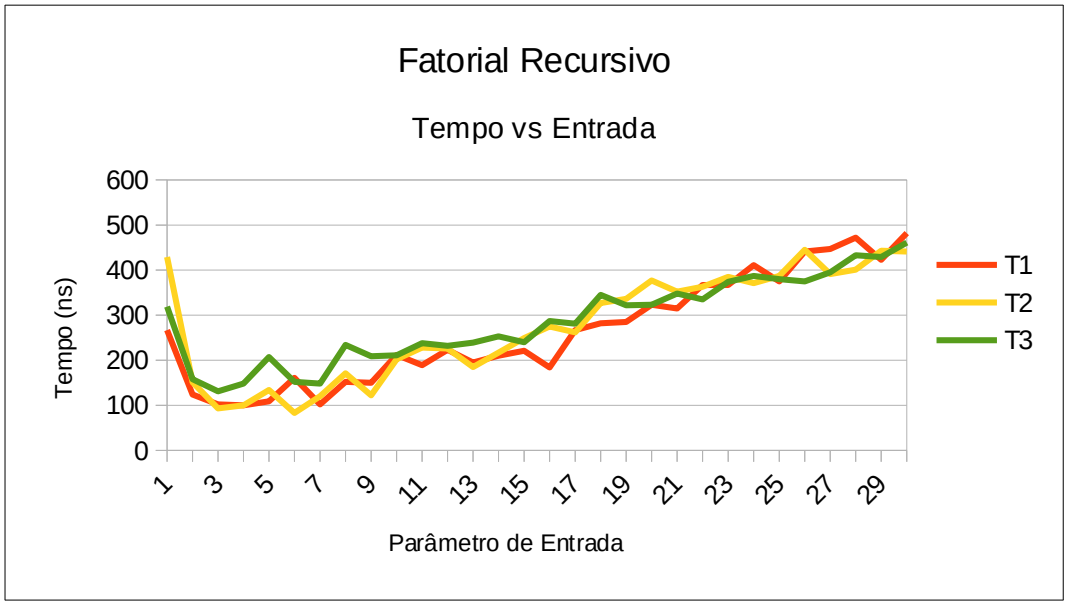


Gráfico 4.2 – Tempo gasto na Versão Recursiva de Fatorial

A partir do Gráfico 4.3, é possível perceber que o tempo de execução do Recursivo foi levemente maior para valores mais altos. Apesar disso, pode-se deduzir, empiricamente, que ambas as formas possuem complexidade de ordem linear, ou O(n).

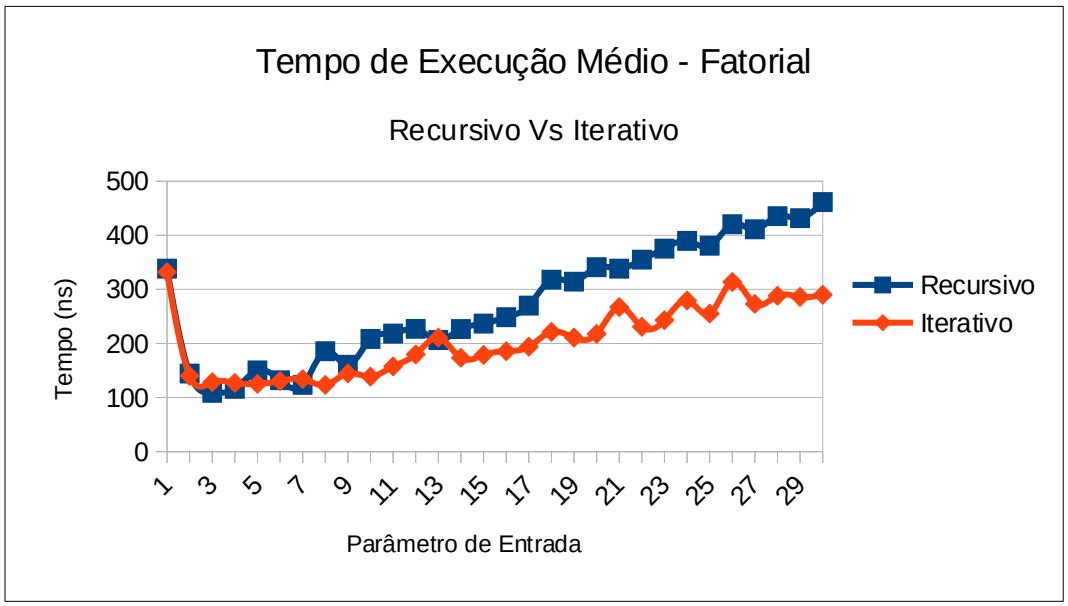


Gráfico 4.3 – Comparação de tempo entre métodos recursivo e iterativo para o cálculo de Fatorial

## 4.2. Fibonacci

A seguir se encontram os resultados das dos experimentos na sequência de Fibonacci. Diferentemente do cáculo de fatorial, o uso da recursividade aumenta exponencialmente o tempo de execução, como é possível ver no Gráfico 4.6.

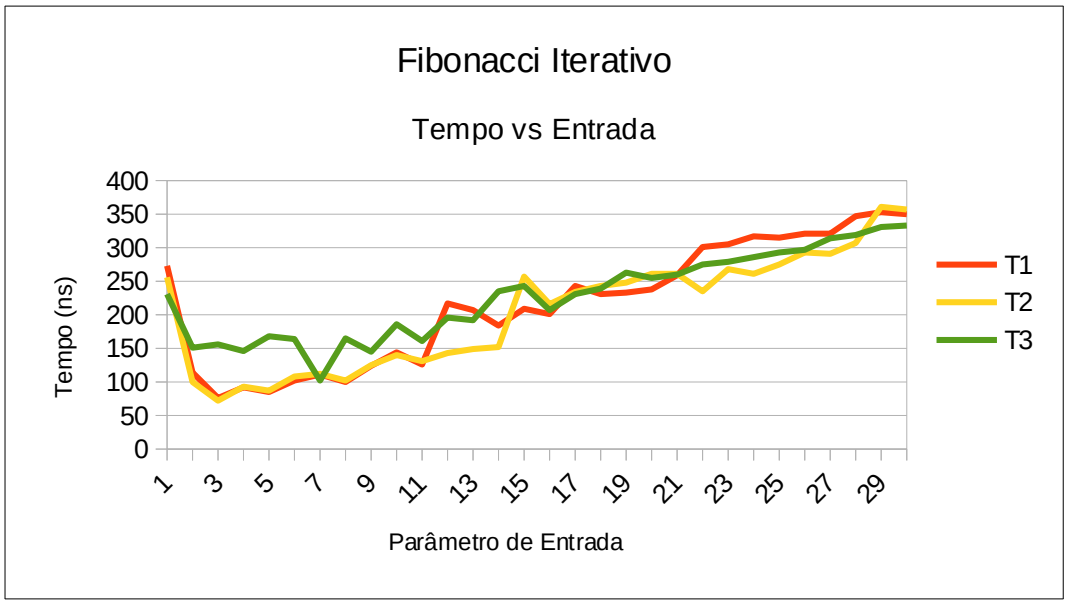


Gráfico 4.4 – Tempo gasto na Versão Iterativa de Fibonacci


Gráfico 4.5 – Tempo gasto na versão recursiva de Fibonacci

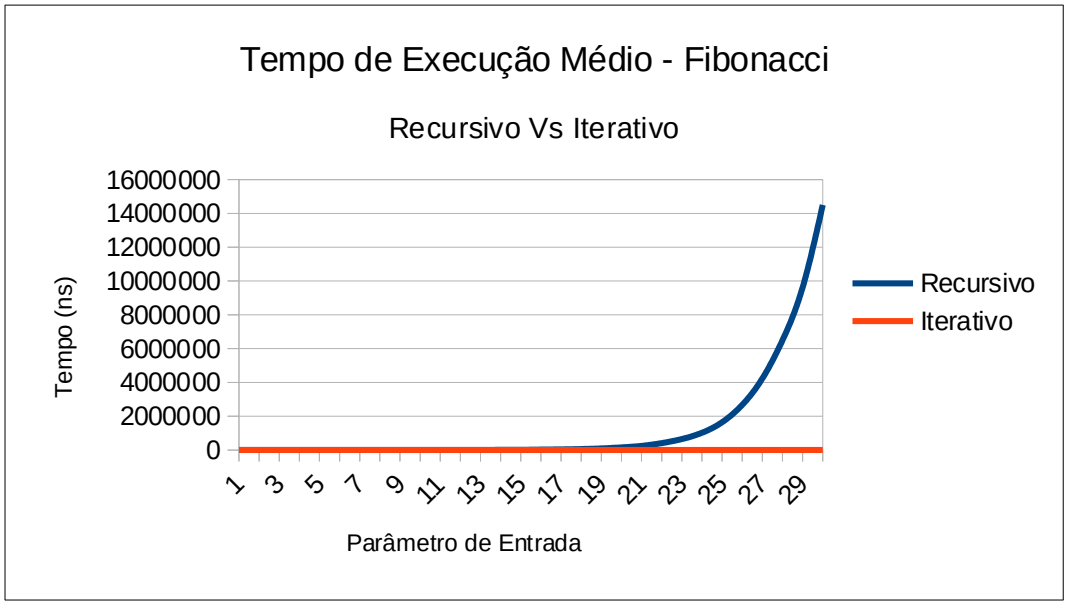


Gráfico 4.6 – Comparação de tempo entre métodos recursivo e iterativo para o cálculo de Fibonacci

Como no exemplo do fatorial, foram retiradas a média de 3 rodadas similares. No gráfico de comparação é claro que a função que dita o tempo de execução do código recursivo é de ordem de grandeza maior que pelo método iterativo. Isso pode ser confirmado pelo código que, ao realizar os cálculos, resulta em ordem de complexidade de $O(2^n)$ para o recursivo e $O(n)$ para o iterativo.]

## 4.3 gprof

Os resultados da simulação em gprof se encontram a seguir:

| Index | % Time | Self | Children | Called | Name |
|---|---|---|---|---|---|
| 1 | 1 | 0,08 | 0 | 30+7049092 | rec_fibonacci |
| | | | 7049092 | | rec_fibonacci |
| 2 | 1 | 0,08 | 0,08 | 30/30 | rec_fibonacci |
| | | | 0,08 | | rec_fibonacci |

| Index | % Time | Self | Children | Called | Name |
|---|---|---|---|---|---|
| | | | | 435 | rec_factorial [1] |
| | 0 | 0 | 0 | 30/30 | main [10] |
| 1 | 0 | 0 | 0 | 30+435 | rec_factorial [1] |
| | | | | 435 | rec_factorial [1] |

## 5. Conclusões

Com os experimentos realizados, foi possível observar como códigos que implementam a mesma função podem apresentar desempenhos de tempo significativamente distintos. Notou-se que, apesar dos códigos recursivos oferecerem, muitas vezes, maior legibilidade e simplicidade na estrutura, seu uso deve ser ponderado, pois tendem a exigir mais tempo e memória, especialmente em problemas que se expandem exponencialmente. A escolha entre recursão e iteração, portanto, é fundamental para otimizar a eficiência do código e reduzir o consumo de recursos.