

CENTRO UNIVERSITÁRIO HERMINIO OMETTO  
UNIARARAS

**GUILHERME CAVENAGHI**

**Avaliação Comparativa de Algoritmos em Diferentes  
Linguagens de Programação: Impacto no  
Desempenho e Eficiência Computacional**

Projeto de Trabalho de Conclusão de Curso apresentado ao curso de Engenharia da Computação da Uniararas, como parte dos requisitos para obtenção do grau de Bacharel.

Orientador: Prof. Renato Luciano Cagnin

ARARAS

2025

# Resumo

Este trabalho tem como objetivo realizar uma análise comparativa de desempenho de algoritmos clássicos implementados em diferentes linguagens de programação. Com base nas linguagens mais utilizadas na indústria e na comunidade acadêmica, serão avaliadas métricas como tempo de execução, uso de memória e complexidade de implementação. A proposta visa identificar como a escolha da linguagem pode influenciar na eficiência computacional de uma aplicação.

**Palavras-chave:** algoritmos, linguagens de programação, desempenho, eficiência computacional.

# 1 Introdução

O desenvolvimento de software eficiente depende de diversos fatores, e a escolha da linguagem de programação é um dos elementos mais determinantes. Embora a lógica dos algoritmos seja, em essência, independente da linguagem, a forma como são implementados e executados pode variar substancialmente entre diferentes ambientes. Essa variação influencia diretamente o desempenho e a eficiência computacional, afetando aspectos como tempo de execução e uso de memória.

A seleção da linguagem de programação, portanto, não é apenas uma questão de preferência ou familiaridade: trata-se de uma decisão técnica que pode ter impacto significativo em aplicações reais. Este trabalho propõe uma análise comparativa entre linguagens de programação, com foco em algoritmos clássicos de diferentes classes de complexidade (P, NP, NP-completo e NP-difícil). Assume-se que, embora o algoritmo permaneça o mesmo, fatores como o modelo de compilação, o gerenciamento de memória e o runtime variam entre linguagens e podem alterar de forma significativa os resultados observados.

Apesar da existência de benchmarks genéricos, como o TIOBE Index [tiobe \[2024\]](#), que indicam a popularidade das linguagens, poucos estudos empíricos abordam de forma prática a comparação de desempenho em algoritmos específicos, considerando dados objetivos como tempo e consumo de memória. Essa lacuna na literatura motiva a realização desta pesquisa, que busca preencher a necessidade de evidências concretas sobre o comportamento de diferentes linguagens em cenários reais.

A justificativa para este estudo reside na crescente demanda por aplicações de alto desempenho e na necessidade de tomar decisões mais embasadas sobre a escolha da linguagem, tanto na academia quanto na indústria. Uma compreensão clara dessas diferenças pode beneficiar desenvolvedores, arquitetos de software e pesquisadores, ajudando-os a selecionar ferramentas alinhadas às necessidades de seus projetos.

A motivação para esta investigação advém da curiosidade de verificar se linguagens tidas como modernas ou populares oferecem, de fato, vantagens práticas em termos de desempenho, ou se tais percepções estão mais ligadas a aspectos culturais e de mercado do que a fatores técnicos.

Os objetivos deste trabalho incluem:

- Avaliar o desempenho de algoritmos clássicos implementados em diferentes linguagens;
- Identificar fatores técnicos que afetam esse desempenho, como gerenciamento de memória, modelo de execução e overhead de runtime;
- Oferecer uma síntese clara de quais linguagens se destacam em cada aspecto analisado.

Ao final, será apresentado um resumo dos resultados obtidos nos testes comparativos, destacando pontos fortes e limitações de cada linguagem no contexto dos algoritmos utilizados.

## 2 Referencial Teórico

### 2.1 Algoritmos Clássicos

Algoritmos são sequências finitas de instruções bem definidas que, partindo de um estado inicial, visam resolver um problema ou realizar uma tarefa específica Cormen et al. [2022]. Entre os algoritmos clássicos, destacam-se:

- **MergeSort**: um algoritmo de ordenação eficiente baseado no paradigma "dividir para conquistar", com complexidade  $O(n \log n)$ , pertencente à classe P Cormen et al. [2022], Knuth [1998].
- **Algoritmos de verificação para SAT**: utilizados para validar se uma atribuição satisfaz uma fórmula booleana, caracterizando um problema típico da classe NP Garey and Johnson [1979].
- **Algoritmos para o Problema da Mochila**: voltados à resolução (exata ou aproximada) de uma seleção ótima de itens, sendo este um problema NP-completo Garey and Johnson [1979].
- **Modelos de decisão de parada**: que se relacionam com o Problema da Parada, um problema NP-difícil e indecidível, importante na teoria da computação Sipser [2012].

### 2.2 Complexidade Computacional

A análise dos algoritmos considerados neste trabalho será baseada nas classes de complexidade da Teoria da Computação Sipser [2012]. Estas classes descrevem o comportamento dos algoritmos em termos de tempo e espaço exigidos conforme o tamanho da entrada cresce.

A classe **P** representa os problemas que podem ser resolvidos em tempo polinomial por uma máquina determinística Cormen et al. [2022]. Já a classe **NP** representa os

problemas cujas soluções podem ser verificadas em tempo polinomial, mesmo que não saibamos resolvê-los de forma eficiente Sipser [2012].

Problemas **NP-completos** são os mais difíceis dentro da classe NP, e um algoritmo eficiente para qualquer um deles implicaria em uma solução eficiente para todos os problemas NP Garey and Johnson [1979]. Já os **NP-difíceis** não pertencem necessariamente à classe NP, mas são ao menos tão difíceis quanto os problemas NP-completos Garey and Johnson [1979], Sipser [2012].

As definições e análises dos algoritmos implementados neste trabalho serão fundamentadas nessas categorias de complexidade, permitindo compreender não apenas a eficiência empírica, mas também os limites teóricos esperados para cada abordagem em diferentes linguagens de programação.

### 2.2.1 Exemplo de Algoritmo da Classe P

Um exemplo clássico de algoritmo da classe **P** é o **MergeSort**, um algoritmo de ordenação baseado no paradigma "dividir para conquistar", cuja complexidade é  $O(n \log n)$ . Este algoritmo sempre resolve o problema de ordenar uma lista em tempo polinomial Cormen et al. [2022], Knuth [1998].

### 2.2.2 Exemplo de Problema da Classe NP

Um exemplo de problema da classe **NP** é o **Problema de Satisfatibilidade Booleana (SAT)**. Dada uma fórmula booleana composta por variáveis e operadores lógicos, o objetivo é determinar se existe uma atribuição de valores que satisfaça todas as cláusulas Garey and Johnson [1979]. Embora verificar uma solução seja rápido, encontrar uma solução não possui, até o momento, algoritmo determinístico em tempo polinomial.

### 2.2.3 Exemplo de Problema NP-Completo

O **Problema da Mochila (Knapsack Problem)** é um exemplo clássico de problema **NP-completo** Garey and Johnson [1979]. Nele, dado um conjunto de itens, cada um com um peso e um valor, e uma capacidade máxima que a mochila pode suportar, busca-se selecionar os itens de forma que o valor total seja maximizado, sem ultrapassar o limite de peso. Apesar de ser fácil verificar se uma combinação de itens respeita a restrição de peso e calcular seu valor total, encontrar a melhor combinação possível é um desafio computacional ainda sem solução eficiente conhecida.

## 2.2.4 Exemplo de Problema NP-Difícil

O **Problema da Parada (Halting Problem)** é um exemplo clássico de problema **NP-difícil**. Este problema consiste em determinar se um programa de computador, dado um input específico, terminará sua execução ou continuará indefinidamente Sipser [2012]. Alan Turing demonstrou que não existe algoritmo capaz de resolver esse problema para todos os casos possíveis, caracterizando-o como indecidível.

## 2.2.5 Resumo das Classes e Exemplos

A seguir, são apresentados exemplos representativos de algoritmos e problemas relacionados a cada uma das principais classes de complexidade computacional:

- **Classe P:** O **MergeSort** é um algoritmo de ordenação com tempo polinomial  $O(n \log n)$  Cormen et al. [2022], Knuth [1998]. Ele resolve de forma eficiente o problema de ordenar uma lista de elementos.
- **Classe NP:** O **Problema de Satisfatibilidade Booleana (SAT)** consiste em determinar se existe uma atribuição de valores que satisfaça todas as cláusulas de uma fórmula booleana Garey and Johnson [1979]. Embora a verificação de uma solução seja rápida, encontrar a solução pode ser computacionalmente difícil.
- **Classe NP-completo:** O **Problema da Mochila (Knapsack Problem)** consiste em selecionar, de um conjunto de itens com pesos e valores, aqueles que maximizam o valor total sem ultrapassar a capacidade da mochila Garey and Johnson [1979]. Verificar uma solução é simples, mas encontrar a combinação ótima é computacionalmente complexo.
- **Classe NP-difícil:** O **Problema da Parada (Halting Problem)** questiona se, dado um programa e uma entrada, é possível determinar se o programa terminará ou continuará executando indefinidamente Sipser [2012]. Alan Turing provou que este problema é indecidível, ou seja, não existe um algoritmo geral que resolva essa questão para todos os casos.

## 2.3 Linguagens de Programação

As linguagens de programação selecionadas para este estudo foram escolhidas com base em sua ampla adoção e relevância prática, sendo fundamentadas em dados de pesquisas de satisfação e popularidade realizadas por fontes consolidadas da área de tecnologia,

como o TIOBE Index tiobe [2024], o GitHub Octoverse octoverse [2022] e o Stack Overflow Developer Survey stackoverflow [2024]. Esses levantamentos refletem o uso real das linguagens por desenvolvedores ao redor do mundo, bem como sua aceitação, preferência e aplicabilidade em diversos contextos, tanto industriais quanto acadêmicos.

As linguagens consideradas neste trabalho são:

- **Python:** muito utilizada em ciência de dados, inteligência artificial e ensino, destaca-se por sua simplicidade e clareza sintática Sebesta [2016].
- **C:** uma linguagem de baixo nível com grande controle sobre recursos computacionais, amplamente utilizada em sistemas embarcados e aplicações de alto desempenho Scott [2015].
- **C++:** oferece recursos da linguagem C com suporte adicional à programação orientada a objetos, sendo comum em softwares de engenharia e jogos Sebesta [2016].
- **Java:** consolidada em ambientes corporativos e no desenvolvimento Android, reconhecida por sua portabilidade e estabilidade Sebesta [2016].
- **JavaScript:** essencial no desenvolvimento web, permite também o desenvolvimento no lado servidor por meio do Node.js Sebesta [2016], Scott [2015].
- **Go:** criada pelo Google, é uma linguagem moderna que combina simplicidade, alto desempenho e excelente suporte à concorrência Scott [2015].
- **Rust:** projetada para segurança e performance, tem se destacado em sistemas que exigem controle de memória sem comprometer a robustez Scott [2015].
- **TypeScript:** superconjunto de JavaScript com tipagem estática, oferecendo maior previsibilidade e escalabilidade no desenvolvimento de aplicações Scott [2015].
- **C#:** utilizada em aplicações Windows, web e jogos, é fortemente tipada e orientada a objetos, com boa integração no ecossistema .NET Sebesta [2016].
- **Kotlin:** moderna, concisa e interoperável com Java, tem sido amplamente adotada no desenvolvimento de aplicações Android Scott [2015].

A diversidade entre essas linguagens — em termos de paradigmas, modelos de execução, sistemas de tipos e comunidade de suporte — proporciona uma base sólida e representativa para a análise comparativa do desempenho de algoritmos clássicos, considerando diferentes abordagens de implementação Sebesta [2016], Scott [2015].



## 3 Materiais e Métodos

### 3.1 Materiais

Para a realização deste estudo, foram utilizados os seguintes recursos:

- **Hardware:** Computador pessoal com processador multi-core, arquitetura x64, 16GB de memória RAM, e sistema operacional Linux Ubuntu 22.04.
- **Ambientes de desenvolvimento:**
  - Compiladores e interpretadores específicos de cada linguagem: *GCC* para C e C++, *OpenJDK* para Java, *Python 3.x*, *Node.js* para JavaScript e TypeScript, *Go Compiler*, *Rust Compiler*, *.NET SDK* para C# e *Kotlin Compiler*.
  - IDEs e editores: Visual Studio Code, IntelliJ IDEA e terminal de linha de comando.
- **Ferramentas de medição:**
  - Ferramentas internas das linguagens para medição de tempo (por exemplo, `time`, `System.nanoTime`, `performance.now`).
  - Monitoramento de uso de memória via `/usr/bin/time` e ferramentas nativas do sistema.

Os algoritmos selecionados para este estudo foram implementados de forma equivalente em cada linguagem de programação, respeitando suas características sintáticas, mas preservando a lógica algorítmica original.

## 3.2 Métodos

### 3.2.1 Seleção dos Algoritmos

Neste trabalho, foram selecionados algoritmos e problemas clássicos representativos das principais classes de complexidade computacional: P, NP, NP-completo e NP-difícil. A escolha visa permitir uma análise comparativa entre linguagens de programação em diferentes níveis de dificuldade computacional.

Os algoritmos e problemas analisados são:

- **Ordenação (Classe P):** *MergeSort*.
- **Satisfatibilidade Booleana (Classe NP):** *SAT*.
- **Problema da Mochila (Classe NP-completo):** *Knapsack*.
- **Problema da Parada (Classe NP-difícil):** *Halting Problem*.

### 3.2.2 Implementação

Cada algoritmo foi implementado nas seguintes linguagens de programação: Python, C, C++, Java, JavaScript, Go, Rust, TypeScript, C# e Kotlin. As implementações buscaram manter a estrutura algorítmica o mais fiel possível, evitando otimizações específicas ou uso de bibliotecas externas que pudessem alterar significativamente o desempenho.

### 3.2.3 Procedimentos de Teste

Os algoritmos foram executados sobre conjuntos de dados sintéticos, com tamanhos variados para simular diferentes níveis de carga computacional:

- Pequeno: 1.000 elementos.
- Médio: 10.000 elementos.
- Grande: 100.000 elementos.

Cada teste foi repetido 30 vezes para reduzir o impacto de variações esporádicas do sistema. Os seguintes parâmetros foram registrados:

- **Tempo de execução:** medido em milissegundos.

- **Uso de memória:** medido em megabytes.
- **Facilidade de implementação:** avaliada qualitativamente com base na extensão e complexidade do código fonte.

### 3.2.4 Análise dos Resultados

Os dados coletados foram organizados em tabelas e gráficos para facilitar a análise comparativa. A avaliação focou em:

- Identificação da linguagem com melhor desempenho em cada categoria.
- Comparação da eficiência entre linguagens compiladas e interpretadas.
- Discussão sobre a relação entre características da linguagem (paradigma, tipagem, compilação) e o desempenho obtido.

Os resultados serviram de base para as conclusões acerca do impacto da escolha da linguagem de programação na eficiência computacional dos algoritmos clássicos.

## 4 Resultados Esperados

Espera-se identificar quais linguagens oferecem melhor desempenho para tarefas específicas e como características como paradigma, tipagem e compilação influenciam o comportamento dos algoritmos. Esses dados poderão auxiliar desenvolvedores e pesquisadores na escolha mais adequada de linguagem para projetos com requisitos de desempenho.

## 5 Justificativa

Com a crescente demanda por sistemas eficientes e responsivos, a escolha da linguagem de programação tornou-se um fator crítico no desenvolvimento de software. Embora existam inúmeros estudos sobre algoritmos, poucos se concentram em comparações práticas entre linguagens populares no contexto da eficiência computacional. Este trabalho se justifica pela necessidade de fornecer uma base empírica que auxilie desenvolvedores, estudantes e pesquisadores na tomada de decisões mais conscientes, considerando o impacto direto da linguagem no desempenho final das aplicações.

# Referências Bibliográficas

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 4th edition, 2022. ISBN 9780262046305.

Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN 9780716710455.

Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998. ISBN 9780201896855.

octoverse. GitHub Octoverse 2022. <https://octoverse.github.com/2022/top-programming-languages>, 2022. Acesso em: abril 2025.

Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 4th edition, 2015. ISBN 9780124104099.

Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 11th edition, 2016. ISBN 9780133943023.

Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012. ISBN 9781133187790.

stackoverflow. Stack Overflow Developer Survey 2024. <https://survey.stackoverflow.co/2024/technology>, 2024. Acesso em: abril 2025.

tiobe. TIOBE Index for April 2025. <https://www.tiobe.com/tiobe-index/>, 2024. Acesso em: abril 2025.