QUICK SORT

Disciplina: Algoritmo e Estruturas de Dados I

1st Alvaro Augusto José Silva
Engenharia de Computação
CEFET-MG / Campus V
Divinópolis, Brasil
alvaro.ajsilva@gmail.com

2nd Arthur de Oliveira Mendonça
Engenharia de Computação
CEFET-MG / Campus V
Divinópolis, Brasil
mendoncaoarthur@gmail.com

3rd Arthur Santana de Mesquita
Engenharia de Computação
CEFET-MG / Campus V
Divinópolis, Brasil
arthur.santana.mesquita@gmail.com

4th Júlia de Moura Souza
Engenharia de Computação
CEFET-MG / Campus V
Divinópolis, Brasil
juliamourasouza10@gmail.com

5th Luiz Fernando dos Santos Queiroz
Engenharia de Computação
CEFET-MG / Campus V
Divinópolis, Brasil
luizfernandosq0602@gmail.com

***Abstract*—This study aims to analyze the performance of different languages and structures during data sorting using the Quick Sort algorithm. Different versions of Quick Sort were innovative in C, C++, C, Python and Julia. The tests were performed on the timestamp column of the ratings.csv file of MovieLens 25M, applying the algorithm to list, stack and queue structures, both in linear and dynamic versions. The analyses revealed performance differences between the languages, with emphasis on performance for C and C++.**

***Key Word*—Quick Sort, Sorting, Algorithm**

***Resumo*—Este estudo tem como objetivo analisar diferentes linguagens e estruturas no desempenho durante a ordenação de dados no algoritmo Quick Sort. Foram implementadas diferentes versões do Quick Sort em C, C++, C#, Phyton e Julia. Os testes foram realizados sobre a coluna timestamp do arquivo ratings.csv do MovieLens 25M, aplicando o algoritmo nas estruturas de listas, pilhas e filas, tanto em versões lineares tanto em versões dinâmicas. As análises revelaram diferenças de desempenho entre as linguagens, com destaque em desempenho para C e C++.**

***Palavra Chave*—Quick Sort, Ordenação, Algoritmo**

I. INTRODUÇÃO

Algoritimos de ordenação

A ordenação de dados é uma das operações mais fundamentais e ubíquas na ciência da computação. O processo consiste em arranjar um conjunto de itens em uma sequência específica, geralmente numérica ou lexicográfica, a fim de otimizar buscas, facilitar a análise de dados e viabilizar a implementação de outros algoritmos mais complexos. A eficiência com que grandes volumes de dados são organizados impacta diretamente o desempenho de sistemas inteiros, desde bancos de dados e motores de busca até aplicações de bioinformática e processamento de transações financeiras.

Os algoritmos de ordenação podem ser classificados com base em diversas características, sendo as mais comuns a complexidade de tempo e o uso de memória. A complexidade, expressa pela notação Big-O, descreve como o tempo de execução do algoritmo escala com o aumento do volume de dados (n). Algoritmos mais simples, como o *Bubble Sort*, *Insertion Sort* e *Selection Sort*, geralmente possuem uma complexidade de tempo de pior caso de $O(n^2)$, tornando-os impraticáveis para grandes conjuntos de dados.

Em contraste, algoritmos mais avançados, baseados em estratégias como "dividir para conquistar", alcançam uma complexidade média de $O(n \log n)$, representando um ganho de eficiência exponencial. Outro critério importante é o uso de memória: algoritmos *in-place* (no local) reorganizam os itens dentro da estrutura de dados original, exigindo uma quantidade mínima de memória extra, enquanto algoritmos *out-of-place* (fora do local) necessitam de espaço adicional para armazenar os dados durante o processo.

Dentre os algoritmos eficientes que operam em tempo $O(n \log n)$, o Quick Sort se destaca por sua notável performance em casos práticos e sua elegante implementação recursiva. Este trabalho se aprofundará no estudo deste algoritmo, explorando sua origem, seu funcionamento e seu impacto no campo da computação.

Quick Sort

Nos anos 60, com o enorme avanço da computação e a crescente necessidade da criação de algoritmos de ordenação mais rápidos e eficientes, diversas ideias e técnicas passaram a ser analisadas. Muitos dos métodos utilizados na época, como o Bubble Sort dentre outros, apresentavam um desempenho insatisfatório em relação a grandes volumes de dados, sendo assim criando uma carência de um algoritmo que se adaptasse melhor a esses tipos de casos. Foi nesse contexto que o Quick Sort foi proposto pelo cientista da computação Tony Hoare, no ano de 1961, marcando uma significativa mudança na forma de se interpretar e abordar o problema da ordenação.

O Quick Sort surgiu em meio à Guerra Fria, período marcado pela intensa corrida tecnológica entre Estados Unidos e União Soviética, quando a computação ganhou papel de

extrema importância em áreas militares e tecnológicas. Nesse contexto, algoritmos eficientes se tornaram essenciais para processar grandes quantidade de dados com rapidez. Tony Hoare desenvolveu esse algoritmo enquanto fazia parte de um programa de intercâmbio em Moscou, em principio com o objetivo de criar um tradutor automático de idiomas. Para isso, necessitava ordenar grandes dicionários de palavras de forma eficaz, o que o levou a idealizar um algoritmo que se tornaria fundamental para o desenvolvimento da computação nas décadas seguintes. Sendo assim a primeira aplicação prática do Quick Sort foi justamente nesse projeto de tradução automática, aonde era necessário manipular e colocar em ordem milhares de termos linguísticos e estruturas da maneira mais eficiente possível.

O Quick Sort se baseia na ideia de dividir para conquistar: ele escolhe um elemento chamado pivô e reorganiza os demais elementos do vetor de forma que os menores fiquem de um lado e os maiores do outro. Esse processo vai se repetindo recursivamente, fragmentando o problema em partes menores até que a ordenação ocorra de maneira completa. Essa lógica de algoritmo, combinada com sua eficiência em casos médios e sua baixa demanda por memória adicional, fez com que o Quick Sort se tornasse uma escolha padrão em muitas bibliotecas e sistemas modernos. Permitindo que, na média, tempos de execução fossem reduzidos. Apesar de não garantir o melhor desempenho em todos os casos, sua eficiência prática e o menor uso de memória adicional fizeram com que virasse um dos algoritmos de ordenação mais utilizados na computação.

A fim de demonstrar a aplicação prática e avaliar a eficiência do Quick Sort em um cenário de grande volume de dados, este trabalho utilizará o conjunto de dados público *MovieLens 25M Dataset*. Esta base, disponível na plataforma Kaggle (https://www.kaggle.com/datasets/garymk/movielens-25m-dataset), contém 25 milhões de avaliações de filmes fornecidas por usuários, constituindo um ambiente robusto e relevante para testar o desempenho do algoritmo.

II. METODOLOGIA

O Quick Sort é um algoritmo de ordenação que se baseia na estratégia de *dividir para conquistar*. Sua popularidade se deve à sua alta eficiência em casos médios, resultando em uma complexidade de tempo de $O(n \log n)$, e ao fato de ser um algoritmo *in-place*, que demanda uma quantidade mínima de memória auxiliar.

Contudo, o desempenho prático do Quick Sort é fortemente dependente da distribuição dos dados de entrada. Enquanto o caso médio é excelente, uma sequência de escolhas de pivô desfavoráveis — que ocorre tipicamente em vetores já ordenados ou com baixa entropia quando se utiliza uma estratégia de pivoteamento ingênua — pode degradar sua performance para o pior caso.

Para mitigar esse risco e garantir uma performance robusta na maioria dos cenários, a escolha da implementação é de suma importância. A abordagem adotada neste trabalho utiliza a técnica da **mediana de três** para a seleção do pivô, que visa justamente evitar os piores casos mais comuns. Para formalizar esta implementação e detalhar sua lógica de funcionamento, o pseudocódigo a seguir é apresentado. Ele servirá de base para toda a análise de desempenho subsequente.

```
1: algoritmo QuickSort_MedianaDeTres(vetor, inicio, fim)
2: if inicio < fim then
3:     pivô ← MedianaDeTres(vetor, inicio, fim)
4:     pos_pivô ← Particionar(vetor, inicio, fim, pivô)
5:     QuickSort_MedianaDeTres(vetor, inicio, pos_pivô - 1)
6:     QuickSort_MedianaDeTres(vetor, pos_pivô + 1, fim)
7: end if
```

```
8: função MedianaDeTres(vetor, inicio, fim)
9: meio ← ⌊(inicio + fim)/2⌋
10: {Ordena os três elementos: inicio, meio, fim}
11: if vetor[inicio] > vetor[meio] then
12:     Trocar(vetor[inicio], vetor[meio])
13: end if
14: if vetor[meio] > vetor[fim] then
15:     Trocar(vetor[meio], vetor[fim])
16: end if
17: if vetor[inicio] > vetor[meio] then
18:     Trocar(vetor[inicio], vetor[meio])
19: end if
20: {A mediana está agora em vetor[meio]}
21: retorna vetor[meio]
```

```
22: função Particionar(vetor, inicio, fim, pivô)
23: i ← inicio
24: j ← fim
25: while i ≤ j do
26:     while vetor[i] < pivô do
27:         i ← i + 1
28:     end while
29:     while vetor[j] > pivô do
30:         j ← j − 1
31:     end while
32:     if i ≤ j then
33:         Trocar(vetor[i], vetor[j])
34:         i ← i + 1
35:         j ← j − 1
36:     end if
37: end while
38: retorna i
```

```
39: procedimento Trocar(ref a, ref b)
40: temp ← a
41: a ← b
42: b ← temp
```

Estratégias para a Escolha do Pivô

A etapa de particionamento é o coração do Quick Sort, e sua eficiência é quase inteiramente ditada pela qualidade da escolha do pivô. Uma escolha inadequada pode levar a partições

desbalanceadas, degradando o desempenho do algoritmo para sua complexidade de pior caso, $O(n^2)$. A escolha ideal do pivô depende fortemente da distribuição dos dados de entrada.

- **Primeiro ou Último Elemento:** A abordagem mais simples.
  - *Como funciona:* Escolhe o primeiro ou o último elemento do sub-vetor como pivô.
  - *Melhor para:* Conjuntos de dados pequenos ou onde se tem certeza de que não há ordenação prévia (distribuição verdadeiramente aleatória). Sua simplicidade o torna útil em contextos didáticos.
  - *Pior para:* Distribuições ordenadas, inversamente ordenadas ou com baixa entropia (quase ordenadas), onde causa o pior caso $O(n^2)$ de forma consistente.
- **Elemento Aleatório:** A abordagem mais robusta contra o pior caso.
  - *Como funciona:* Escolhe um elemento de uma posição aleatória dentro do sub-vetor.
  - *Melhor para:* Qualquer tipo de distribuição, especialmente quando não se tem conhecimento prévio sobre os dados. É a escolha mais segura para garantir o desempenho $O(n \log n)$ probabilisticamente.
  - *Pior para:* Não há uma distribuição em que seja "ruim", mas introduz um pequeno custo computacional para a geração de números aleatórios em cada passo recursivo.
- **Mediana de Três:** O método adotado neste trabalho, buscando um balanço prático.
  - *Como funciona:* Seleciona como pivô o valor mediano entre três elementos: o primeiro, o do meio e o último do sub-vetor.
  - *Melhor para:* Distribuições de propósito geral. Oferece excelente proteção contra dados ordenados ou quase ordenados, que são comuns em cenários do mundo real, com um custo adicional muito baixo.
  - *Pior para:* Ainda pode ser vulnerável a distribuições maliciosamente construídas para enganar a heurística (embora sejam raras na prática).

Considerando o balanço entre eficiência, simplicidade de implementação e proteção contra os cenários de pior caso mais comuns, a técnica da **mediana de três** foi a escolhida para a implementação e análise de desempenho neste estudo.

O Particionamento para Chaves Duplicadas (Problema da Bandeira Holandesa)

Além da seleção do pivô, o método de **particionamento** também afeta o desempenho, especialmente em vetores com muitos elementos repetidos. O esquema de particionamento padrão (como o de Lomuto ou Hoare) divide o vetor em duas seções: $<$ pivô e $\geq$ pivô. Se o vetor contém muitas chaves iguais ao pivô, elas podem acabar todas em uma das partições, levando a recursões desbalanceadas.

Para resolver isso, Edsger W. Dijkstra popularizou o **esquema de particionamento de 3 vias**, também conhecido como o Problema da Bandeira Holandesa.

- *Como funciona:* Ele particiona o vetor em três seções em um único passo: uma com elementos **menores** que o pivô, uma com elementos **iguais** ao pivô, e uma com elementos **maiores** que o pivô.
- *Vantagem:* Após o particionamento, os elementos na seção do meio (iguais ao pivô) já estão em sua posição final e são excluídos das chamadas recursivas. A recursão ocorre apenas nas seções "menor que"e "maior que", que são estritamente menores.
- *Melhor para:* Distribuições com **baixa cardinalidade**, ou seja, onde um número pequeno de valores únicos se repete muitas vezes. Exemplos incluem ordenar um grande vetor de idades de uma população, notas de alunos em uma escala de 0 a 10, ou o status de pedidos em um sistema (ex: PENDENTE, APROVADO, ENVIADO). Nesses casos, o particionamento de 3 vias pode transformar o desempenho de $O(n \log n)$ para quase linear, $O(n)$.

Embora a implementação neste trabalho utilize um particionamento padrão combinado com a seleção por mediana de três, é fundamental conhecer as diferentes técnicas de escolha de pivô.

Representação visual do Quick Sort

A fim de ilustrar de maneira inequívoca a dinâmica do algoritmo QuickSort com a estratégia de mediana de três, uma completa representação visual foi desenvolvida. Esta sequência detalha cada etapa do processo, desde a seleção do pivô até a conclusão do particionamento em todos os níveis recursivos.

Devido à sua natureza exaustiva e para não interromper o fluxo principal do texto, a demonstração completa foi alocada ao final do trabalho. Convidamos o leitor a consultar a execução passo a passo no Apêndice na página 9.

Metodologia Experimental

Para avaliar empiricamente a performance de um processo de ordenação completo, foi desenhado um experimento que vai além da simples execução do algoritmo Quick Sort. A metodologia avalia o tempo total necessário para ordenar dados armazenados em diferentes estruturas de dados fundamentais, considerando o custo de manipulação e transferência desses dados.

Fonte e Estrutura dos Dados: A fonte de dados para todos os testes foi o arquivo `ratings.csv` do conjunto de dados *MovieLens 25M Dataset*. A Tabela I ilustra o formato dos registros contidos no arquivo.

Tabela I: Exemplo de registros do arquivo `ratings.csv`.

userId	movieId	rating	timestamp
1	296	5.0	1147880044
1	306	3.5	1147868817
1	307	5.0	1147868828
...	...	...	...
162541	56176	2.0	1240950697
162541	58559	4.0	1240953434
162541	63876	5.0	1240952515

Para manipular esses dados em memória, cada linha do arquivo foi mapeada para uma estrutura customizada (ou classe), denominada "Registro de Avaliação", contendo os quatro campos: `userId` (inteiro), `movieId` (inteiro), `rating` (ponto flutuante) e `timestamp` (inteiro longo).

1) Estruturas de Dados Avaliadas: O núcleo do experimento consistiu em avaliar o processo de ordenação a partir de seis diferentes estruturas de dados, implementadas para armazenar os "Registros de Avaliação":

Vetor Estático: Um array com tamanho fixo, alocado em tempo de compilação.
Vetor Dinâmico: Um array cujo tamanho pode ser ajustado em tempo de execução (como `std::vector` em C++ ou `list` em Python).
Lista Estática: Uma estrutura de lista implementada sobre um vetor estático.
Lista Dinâmica (Encadeada): Uma lista clássica baseada em nós, onde cada nó contém um registro e um ponteiro para o próximo nó.
Pilha Estática: Estrutura LIFO (Last-In, First-Out) implementada sobre um vetor estático.
Pilha Dinâmica (Encadeada): Estrutura LIFO implementada com nós encadeados.

2) Pipeline de Ordenação e Procedimento de Teste: A métrica de desempenho avaliada foi o **tempo total do pipeline de ordenação**. O cronômetro foi acionado para medir a execução completa da seguinte sequência de passos:

1) **Início do Cronômetro.**
2) **Extração para Vetor Auxiliar:** Os dados são transferidos da estrutura de dados de origem (ex: de uma lista encadeada) para um vetor auxiliar em memória.
3) **Ordenação:** O algoritmo Quick Sort, com pivô por mediana de três, é executado sobre este vetor auxiliar. A chave de ordenação utilizada foi o campo **`timestamp`** de cada registro.
4) **Reinserção na Estrutura Original:** Os dados agora ordenados no vetor auxiliar são transferidos de volta para a estrutura de dados de origem (que é previamente limpa).
5) **Fim do Cronômetro.**

Este procedimento garante que o tempo medido reflete não apenas a eficiência do Quick Sort em si, mas também o custo computacional de manipular e reestruturar os dados, que varia significativamente entre as estruturas avaliadas.

3) Casos de Teste e Volumes de Dados: Para cada uma das seis estruturas, o pipeline de ordenação foi executado sob três cenários de ordem inicial de dados (Original, Pré-Ordenado por Timestamp e Inversamente Ordenado por Timestamp). Os testes foram conduzidos com os seguintes volumes de dados (N), correspondentes às primeiras N linhas lidas do arquivo `ratings.csv`:

- 100 registros
- 1.000 registros
- 10.000 registros
- 100.000 registros
- 1.000.000 de registros

4) Ambiente de Teste e Confiabilidade: Todos os testes foram executados em um único ambiente de hardware e software para garantir a consistência.

- **Processador:** Intel® Core™ i7-8550U × 8
- **Memória RAM:** 16.0GiB DDR4 3200 MHz
- **Sistema Operacional:** Ubuntu 24.04.1 LTS
- **Linguagens:** C, C++, C#, Python, Julia

Para assegurar a confiabilidade estatística, cada teste (combinação de estrutura, cenário e volume) foi **executado 10 vezes**, e o resultado final reportado é a **média aritmética** dos tempos coletados.

III. MODELOS DE APLICAÇÃO

A. Onde eu Encontro o Quick Sort?

O Quick Sort é considerado um algoritmo in-place, o que significa que ele realiza a ordenação modificando os próprios elementos do vetor original, sem precisar criar um novo vetor auxiliar para guardar cópias ou partes dos dados costumando ser encontrados em estruturas de bancos de dados, especialmente em operações que envolvem a ordenação de registros, como nas fases de sort-merge join ou durante a construção de índices. Nesses casos, o fato de ser um algoritmo um algoritmo in-place, que não requer uso de memória adicional significativa, faz com que ele se torne uma alternativa com melhor desempenho que algoritmos como o Merge Sort, que são algoritmos que tendem a exigir espaço extra proporcional ao tamanho da entrada e do vetor a ser organizado. Em ambientes onde a ordenação precisa ter alto desempenho, o Quick Sort se destaca por sua performance prática superior.

Em sistemas embarcados, onde memória é bastante limitada, o Quick Sort é preferido justamente por sua baixa demanda por recursos. Como não realiza alocações adicionais de memória, ele se adapta bem a dispositivos que precisam executar algoritmos com agilidade, sem sobrecarregar o hardware.

Em compiladores, pelo fato de sua implementação simples, combinada com um bom desempenho na maioria dos casos, o torna uma escolha eficiente para organizar grandes volumes de dados intermediários durante o processo de tradução do código.

B. Quando Utilizar o Quick Sort

- **Grandes Conjuntos de Dados:**
  O Quick Sort é um algoritmo de ordenação in-place, o que significa que ele requer uma quantidade constante de memória adicional. Isso é útil quando a memória é crucial e você não pode se dar ao luxo de usar algoritmos que requerem espaço adicional significativo.
- **Conjuntos de Valores Dispersos:**
  O Quick Sort é indicado para ocasiões na qual os dados apresentam ampla variação em seus valores, ou seja,

estão mais dispersos, pois aumenta as chances de divisões equilibradas e evita os piores cenários de desempenho. Isso faz com que o algoritmo seja ainda mais eficiente em situações práticas com dados diversos e heterogêneos.

C. Vantagens E Desvantagens do Algoritmo

- Algortimo In-Place:
  Não requer memória adicional significativa além do espaço usado pelo array original.
- Implementação simples e versátil:
  A lógica do algoritmo é fácil de se implementar e pode ser facilmente adaptada com diferentes estratégias de escolha de pivô para melhorar o desempenho. Podendo ter melhor desempenho a partir da escolha do melhor pivô.
- Desempenho em grande Conjuntos de Dados:
  Devido à sua maneira de dividir para conquistar para a ordenação e baixa sobrecarga, o Quick Sort se lida muito bem em grandes quantidade de dados ordenados em memória.
- Sensível a escolha do pivô:
  O desempenho do Quick Sort está fortemente ligado a escolha do pivô, em um caso de escolha de um pivô ruim, tende a ter um desempenho inferior. Estratégias como ”mediana de três”são tomadas para evitar o pior caso, consistindo essa estratégia em escolher o primeiro, o último , e o elemento central do vetor e fazer a mediana desse três valores, podendo apartir dessa estratégia evitar casos de O(n^2).
- Casos de escolha de um pivô ruim:
  Quando o pivô é escolhido de forma inadequada, o algoritmo pode ter desempenho de O(n^2). Isso ocorre, por exemplo, em listas já ordenadas, fazendo com que um pivô incorreto faça com que haja um número de comparações excessivas, comprometendo seu desempenho.

IV. Resultados e Discussão

Inicialmente escolheu-se a coluna **`rating`** para a ordenação, porém, imediatamente observou-se uma grande demora na execução do algoritmo, e, após optar-se pela coluna **`timestamp`** os novos testes tiveram uma diminuição considerável no tempo (Fig. 1)

A razão dessa diferença está na natureza do algoritmo e na organização do conjunto de dados. A coluna **`rating`** varia entre *0* e *5* em incrementos de *0.5*, tendo portanto apenas 10 possíveis valores diferentes. Verifica-se que em cada iteração do Quick Sort há pouca diminuição no número de termos desordenados, com a maior parte da execução sendo movendo termos idênticos ao pivô para uma das partições devido ao agrupamento de valores iguais.

Isso é especialmente detrimental para a linguagem **Julia**, que inicialmente possuiu o dobro do tempo de execução de python, e após a troca do conjunto de dados a ser ordenado, teve desempenho comparável a *C* e *C++*, de forma que Julia é *3200* vezes mais lento na base de dados com alta repetição.

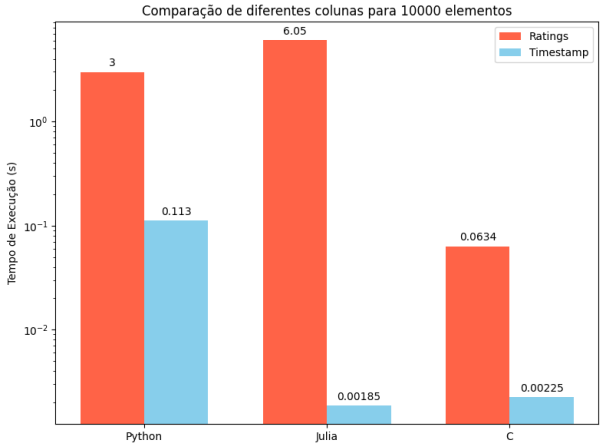


Figura 1: Gráfico em escala logarítmica dos tempos de execução nas duas colunas

Assim, demonstra-se a grande dependência do Quick Sort na organização do conjunto de dados escolhido. Sendo este substancialmente mais eficiente quando baseado em dados com **poucas repetições**, como é o caso da coluna **`timestamp`**, onde é altamente improvável que muitos usuários lancem a nota ao sistema no exato mesmo segundo. Também é importante pontuar que o tempo de execução do algoritmo pela coluna **`rating`** poderia ser ainda maior caso não fosse utilizado o pivô de mediana de três pois haveria uma maior probabilidade de divisões desbalanceadas da lista, ocasionando no pior caso do Quicksort.

Assim, após a finalização dos testes para Lista Estática na ordenação dos timestamps, gerou-se a seguinte tabela comparativa dos tempos de execução entre as linguagens escolhidas.

Tabela II: Média dos Tempos Para Lista Estática (s)

Tamanho	*C*	*C++*	*C#*	*PY*	*JL*
100	0.0000450	0.000063	0.000039	0.001200	0.0000882
1.000	0.0001000	0.000237	0.000387	0.010700	0.0002041
10.000	0.0022465	0.001479	0.005492	0.112600	0.0018548
100.000	0.0108480	0.013609	0.058507	1.372500	0.0256452
1.000.000	0.1273460	0.153992	0.747585	15.691050	0.3560414

Para melhor observação gerou-se também um gráfico em escala logaritmica (Fig. 2) onde observa-se que, como esperado, as linhas permanecem aproximadamente paralelas no gráfico, demonstrando que o crescimento assintótico da função permanece o mesmo entre as diferentes linguagens. O fator de diferença de performance entre as linguagens pode ser aproximado, portanto, como uma constante multiplicativa. A linguagem mais lenta observada, **Python**, apresentou tempo de execução aproximadamente *100* vezes maior que **C++** nas entradas maiores que *10000*, por exemplo.

Quanto a essa discrepância entre linguagens, pode ser inicialmente associada à diferença entre linguagens interpretadas como *Julia*, *Python* e *C#*, e compiladas como *C* e *C++*, já que, de fato, *C* e *C++* foram as linguagens mais rápidas, com **Julia** demorando *60-135%* mais em sua execução que **C++**.

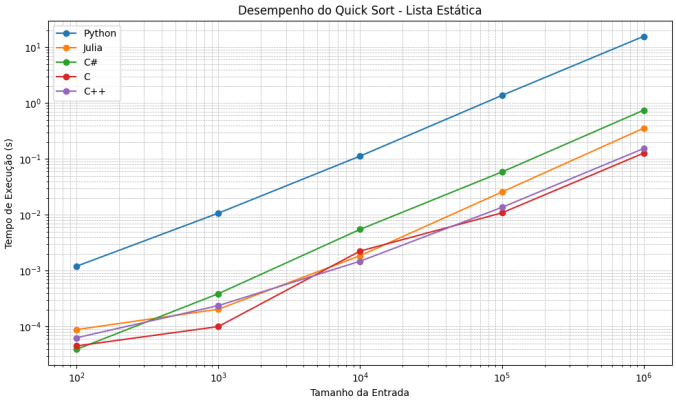


Figura 2: Tempos de execução para Lista Estática

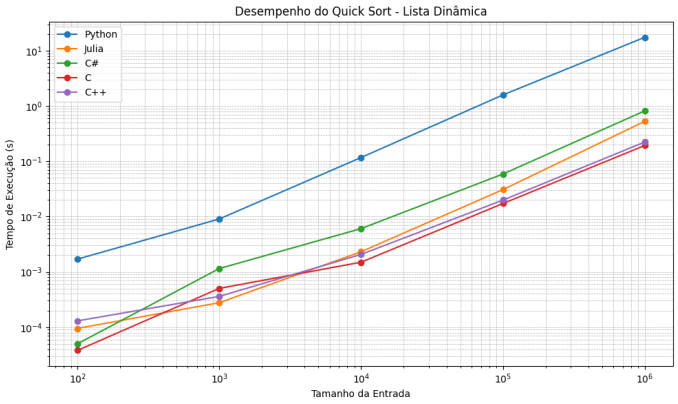


Figura 3: Tempos de execução para Lista Dinâmica

Porém a diferença ainda maior identificada entre as três linguagens interpretadas escolhidas exige posterior diferenciação entre estas linguagens. **Python** em especial é *2000%* mais lenta que **C#** enquanto **C#** é *50-110%* mais lenta que **Julia**. Primeiramente, é importante diferenciar *Julia* e *C#* devido ao seu **compilador JIT**, que converte código em linguagem de máquina durante a execução do programa, permitindo diversas otimizações de código que não são permitidas em linguagens interpretadas típicas, como é o caso de *Python*, esse código pode depois ser reutilizado em tarefas repetidas, replicando o comportamento de uma linguagem compilada, uma característica especialmente efetiva em um algoritmo altamente recursivo como o Quicksort.

Em seguida, deve-se considerar a diferença na forma como diferentes tipos de dados são salvos e manipulados em cada linguagem. No caso de *Julia*, seu JIT é baseado na tecnologia **LLVM**, o que permite otimizações avançadas como *vetorizações* e *inlining*. Isso faz com que a linguagem seja extremamente eficiente para cálculos numéricos e manipulação de grandes volumes de dados, já *C#* , uma linguagem de uso geral, não possui o JIT especializado para o tipo de dados inferido, sem funções específicas para comparação numérica, o que causa a diferença de execução entre as duas linguagens. Por fim, é importante pontuar que *Python*, como uma linguagem completamente dinâmica, realiza verificações constantes de tipos e manipula estruturas de dados altamente flexíveis, o que adiciona uma sobrecarga significativa.

A segunda sequência de testes, utilizando Lista Dinâmica e algumas **outras estruturas de dados** resultaram nas seguintes tabelas e gráficos.

Tabela III: Média dos Tempos Para Lista Dinâmica (s)

Nº de Entradas	*C*	*C++*	*C#*	*PY*	*JL*
1.000	0.000038	0.00013	0.00005	0.00170	0.000095
10.000	0.000503	0.00036	0.00115	0.00905	0.000277
100.000	0.001496	0.00207	0.00606	0.11710	0.002316
500.000	0.017228	0.01989	0.05881	1.58545	0.030721
1.000.000	0.192995	0.22331	0.81629	17.47410	0.528310

Tabela IV: Média dos Tempos Para Pilha Estática (s)

Nº de Entradas	*C*	*C++*	*C#*	*PY*	*JL*
1.000	0.000017	0.000082	0.000116	0.001800	0.000088
10.000	0.000122	0.000124	0.003671	0.014150	0.000214
100.000	0.002457	0.001118	0.006607	0.131000	0.002305
500.000	0.010808	0.013696	0.060055	1.689400	0.024462
1.000.000	0.124157	0.155643	0.777399	16.785150	0.344365

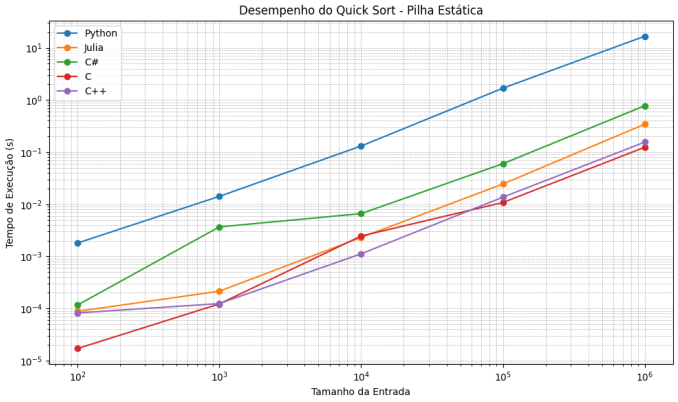


Figura 4: Tempos de execução para Pilha Estática

Tabela V: Média dos Tempos Para Pilha Dinâmica (s)

Nº de Entradas	*C*	*C++*	*C#*	*PY*	*JL*
1.000	0.000026	0.00012	0.00003	0.00320	0.000099
10.000	0.000418	0.00055	0.00101	0.01075	0.000309
100.000	0.004460	0.00184	0.00575	0.14085	0.003359
500.000	0.016554	0.01978	0.05753	1.82170	0.031429
1.000.000	0.187110	0.22237	0.77498	20.36265	0.493027

Tabela VI: Média dos Tempos Para Fila Estática (s)

Nº de Entradas	*C*	*C++*	*C#*	*PY*	*JL*
1.000	0.000046	0.00008	0.00004	0.00280	0.00008
10.000	0.000300	0.00040	0.00094	0.01095	0.00022
100.000	0.001620	0.00197	0.01029	0.13615	0.00151
500.000	0.015470	0.01982	0.05975	1.63270	0.02336
1.000.000	0.170340	0.21705	0.78017	18.49560	0.34703

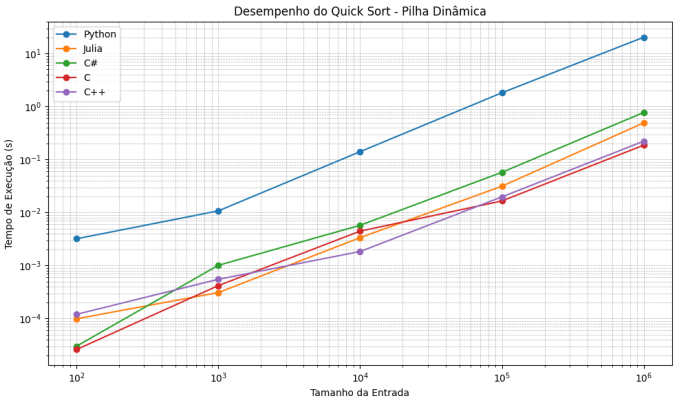


Figura 5: Tempos de execução para Pilha Dinâmica

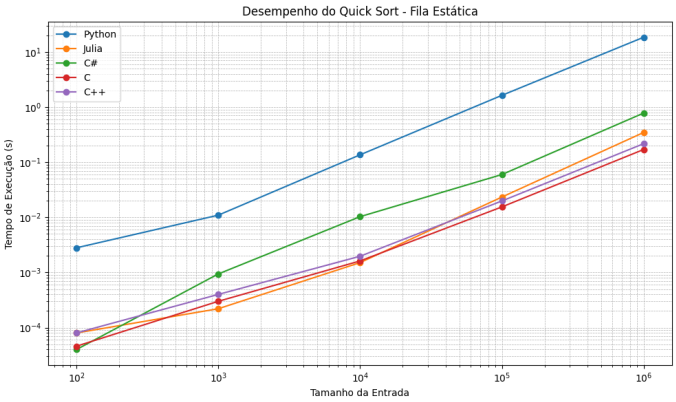


Figura 6: Tempos de execução para Fila Estática

Tabela VII: Média dos Tempos Para Fila Dinâmica (s)

Nº de Entradas	*C*	*C++*	*C#*	*PY*	*JL*
1.000	0.000072	0.00011	0.00003	0.00130	0.00011
10.000	0.000050	0.00043	0.00108	0.01195	0.00033
100.000	0.001600	0.00187	0.00889	0.14210	0.00232
500.000	0.018350	0.01993	0.06029	1.83800	0.03382
1.000.000	0.200040	0.22343	0.76934	20.43675	0.62261

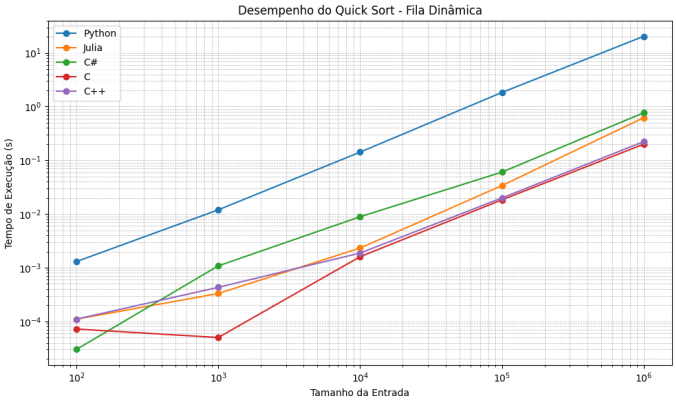


Figura 7: Tempos de execução para Fila Dinâmica

A principal tendência que se observa é que **estruturas estáticas superam as dinâmicas** em desempenho. Isso se deve à busca das medianas para determinação do pivô e à permutação de elementos, que em estruturas estáticas possuem custo constante. Essa diferença de desempenho porém varia conforme a linguagem, com **Julia** sendo a linguagem que sofreu o maior aumento percentual - 57% - no tempo de execução em estruturas dinâmicas para grandes volumes de dados, novamente devido a alta especialização da linguagem para operações puramente numéricas. *C* e *C++* também tiveram um grande aumento, *40%* e *30%* respectivamente, devido a seu baixo nível, que torna o custo adicional das abstrações e manipulações de memória das estruturas dinâmicas um gargalo maior, enquanto isso, *Python* e *C#*, que já executam alta manipulação de memória, incluindo *garbage colletion*, tem seu desempenho pouco afetado pela mudança de estruturação, *15%* e *3%* respectivamente.

Outra comparação interessante é entre *listas*, *filas* e *pilhas*. Ao comparar os resultados entre estas 3 estruturas, percebe-se que a estrutura de **Lista** é a que possui **melhor desempenho** no Quicksort. Isto se deve à natureza dos algoritmos de divisão e conquista, onde se faz necessário dividir a estrutura em várias partes. A Lista, como uma estrutura com alta flexibilidade de adição e remoção de elementos, é necessária para estes algoritmos, de forma que o custo adicional das outras duas estruturas decorre da realocação dos dados para uma estrutura que possa ser **dividida**. A exceção deste resultado é *C*, onde o custo da estrutura de pilha se mostrou menor. É evidente porém, que por este caso possuir uma diferença pequena de *2%*, se encontra na margem de erro dos testes e que o gargalo da leitura da estrutura original é tão pequeno que pode ser considerado irrelevante no contexto geral do tempo de execução.

Já a **pior** estrutura foi a **Fila**, que foi, até *40%* mais lenta que Lista e até *37%* mais lenta que Pilha dentre as linguagens escolhidas, algo que pode ser associado à estrutura de **FIFO** que demora mais a ser reorganizada sem inversão da ordem original dos dados, um problema similar se apresenta na Pilha, porém apenas quando ocorre a devolução dos dados à estrutura original, tarefa que não foi considerada no tempo de execução neste teste. Uma forma de mitigar esse tempo seria iniciar ou finalizar respectivamente a ordenação dos dados na ordem *inversa* para estas estruturas, de forma que menos tempo seja gasto para transferir os dados entre estruturas.

Por fim, foi gerada uma tabela comparando a **quantidade de memória** utilizada por cada linguagem para a execução do algoritmo.

Observa-se não só que as **linguagens interpretadas** tiveram um gasto de memória **substancialmente mais alto** que as compiladas, mas também que as linguagens interpretadas com compilação **JIT** tiveram **mais** gasto de memória que *Python*, com *C#* gastando *228%* mais e *Julia 654%* mais. Assim, mostra-se a **consequência** do armazenamento das funções compiladas que proporcionam o impressionante desempenho destas linguagens. O desempenho comparável ao de uma linguagem compilada com a versatilidade de uma linguagem

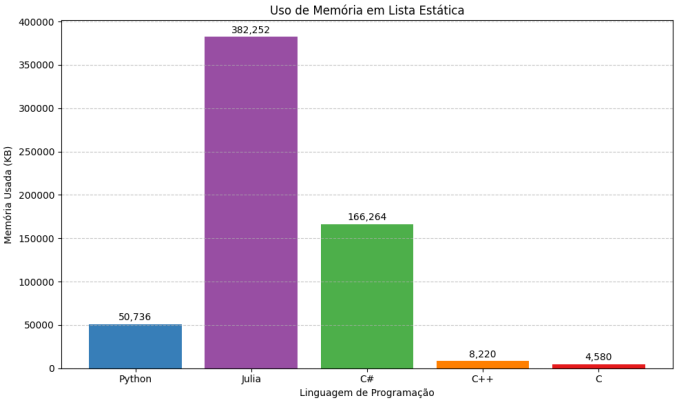


Figura 8: Comparação do uso de memória na execução do Quicksort com 100.000 elementos baseado na linguagem escolhida

interpretada tem como custo, neste caso, um aumento extremo na quantidade de memória necessária para a execução do algoritmo.

V. Conclusão

Assim, definimos que para esses casos de teste que a **Lista Estática** é a melhor estrutura para a execução do método de ordenação *Quicksort*, com este algoritmo se beneficiando substancialmente também do uso de linguagens de baixo nível como *C*, especialmente quando utilizado para ordenar um conjunto de dados pouco abstratos como os números inteiros. Cabe porém a cada programador selecionar, quando maior flexibilidade se faz necessária, entre o menor uso de memória de uma linguagem interpretada pura como **Python**, ou o maior desempenho de uma linguagem com *compilador JIT* como **Julia** e **C#**. Também conclui-se que o Quicksort deve ser evitado para bases de dados com alto indíce de repetição de valores, abrindo a possibilidade para que testes com outros algoritmos de ordenação encontrem formas mais eficientes de ordenar a *MovieLens 25M* utilizando a coluna *rating*.

.

Referências

[1] SILVA, A. A. *Algoritmo Quick Sort*. Acessado em: 29 de junho de 2025. 2024. URL: https://github.com/alvaroajs/ordenacaoAEDS.

[2] HOARE, C. A. R. "Algorithm 64: Quicksort". Em: *Communications of the ACM* 4.7 (1961), p. 321.

[3] CORMEN, T. H. *Algoritmos - Teoria e Prática*. MIT Press, 1989.

[4] KNUTH, D. E. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd. Addison-Wesley Professional, 1998.

[5] BENTLEY, J. L. and MCILROY, M. D. "Engineering a Sort Function". Em: *Software: Practice and Experience* 23.11 (1993), pp. 1249–1265.

[6] MUSSER, D. R. "Introspective Sorting and Selection Algorithms". Em: *Software: Practice and Experience* 27.8 (1997), pp. 983–993.

[7] ISO/IEC. *C Standard*. https://en.cppreference.com/w/c. Acessado em: 29 de junho de 2025. 2024.

[8] The Julia Language. *Julia Documentation*. https://docs.julialang.org/. Acessado em: 29 de junho de 2025. 2024.

[9] Python Software Foundation. *Python Documentation*. https://docs.python.org/3/. Acessado em: 29 de junho de 2025. 2024.

[10] Microsoft Corporation. *C Language Specification*. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/specifications. Acessado em: 29 de junho de 2025. 2024.

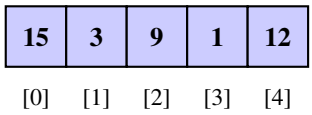
[11] ISO/IEC. *C++ Standard*. https://en.cppreference.com/w/cpp. Acessado em: 29 de junho de 2025. 2024.

[12] TORVALDS, L. *Linux Kernel - lib/sort.c*. Código-fonte do Linux Kernel, implementado com Heapsort. 2024. URL: https://github.com/torvalds/linux/blob/master/lib/sort.c.

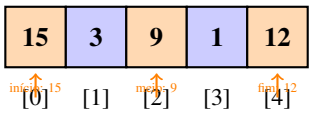
.

APÊNDICE

Este documento mostra todos os passos da execução do QuickSort com mediana de três em uma única visualização sequencial.

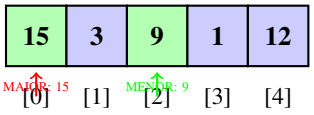


Passo 1: Array Inicial

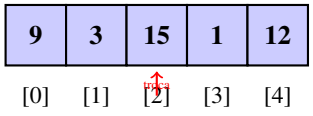


Passo 2: QuickSort – Nível 0: [0..4]

H

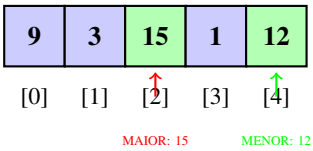


Passo 3: Mediana de Três – Candidatos

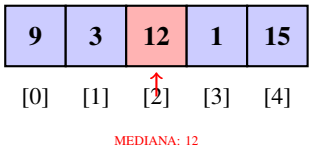


Passo 4: Troca Inicial – arr[0] & arr[2]

H

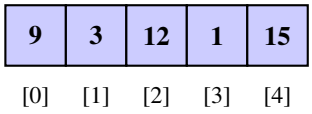


Passo 5: Passo 2: $arr[2] > arr[4] \rightarrow$ Trocar

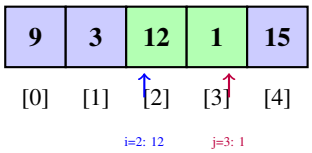


Passo 6: Mediana de Três Concluída

H

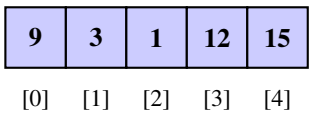


Passo 7: Início do Particionamento - Pivô: 12

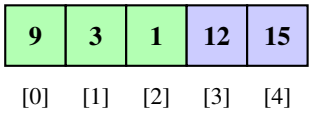


Passo 8: Particionamento: Trocar i=2 com j=3

H

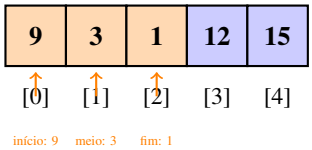


Passo 9: Particionamento Concluído

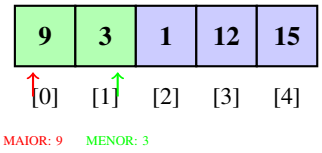


Passo 10: QuickSort - Nível 1: [0..2]

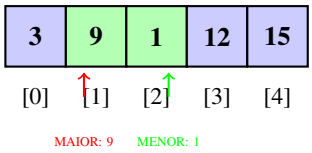
H



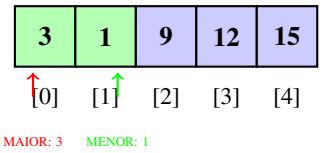
Passo 11: Mediana de Três - Candidatos



Passo 12: Passo 1: $arr[0] > arr[1] \rightarrow$ Trocar

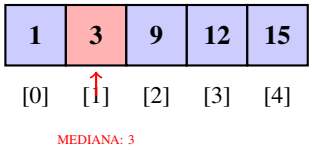


Passo 13: Passo 2: $arr[1] > arr[2] \rightarrow$ Trocar

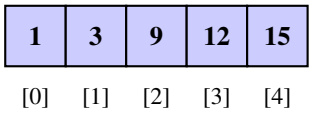


Passo 14: Passo 3: $arr[0] > arr[1] \rightarrow$ Trocar

H

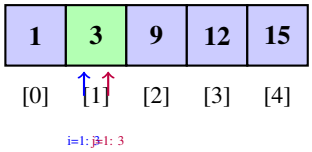


Passo 15: Mediana de Três Concluída

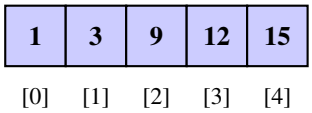


Passo 16: Início do Particionamento - Pivô: 3

H



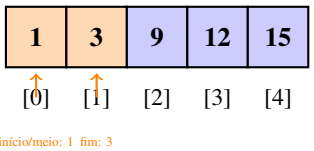
Passo 17: Particionamento: Trocar i=1 com j=1



Passo 18: Particionamento Concluído

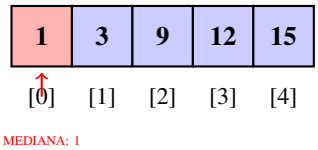
H

Passo 19: QuickSort - Nível 2: [0..1]

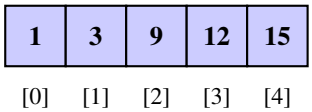


Passo 20: Mediana de Três - Candidatos

H

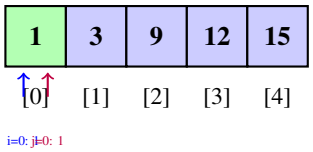


Passo 21: Mediana de Três Concluída

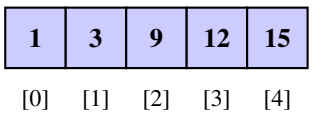


Passo 22: Início do Particionamento - Pivô: 1

H

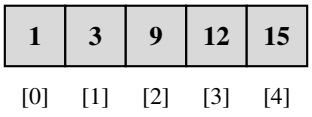


Passo 23: Particionamento: Trocar i=0 com j=0



Passo 24: Particionamento Concluído

H



Passo 25: Array Totalmente Ordenado