UNIVERSIDADE ESTADUAL DA PARAÍBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

DEPARTAMENTO DE COMPUTAÇÃO

JOÃO PAULO DE ARAUJO

ANÁLISE COMPARATIVA DE ALGORITMOS DE ORDENAÇÃO E BUSCA

CAMPINA GRANDE – PB

2025

SUMÁRIO

1. INTRODUÇÃO....3
2. METODOLOGIA DOS TESTES....4
2.1. AMBIENTE DE TESTES....4
2.2. MEDIÇÃO DE TEMPO....4
2.3. NÚMERO DE EXECUÇÕES....4
2.4. GERAÇÃO E TAMANHO DOS DADOS....5
2.5. CENÁRIOS DE TESTE....5
3. ANÁLISE DAS DIFERENÇAS DE PERFORMANCE OBSERVADAS NOS ALGORITMOS DE ORDENAÇÃO....6
3.1. ANÁLISE COMPARATIVA: BUBBLE SORT CLÁSSICO vs. BUBBLE SORT OTIMIZADO....7
3.2. ANÁLISE COMPARATIVA: SELECTION SORT CLÁSSICO vs. SELECTION SORT ESTÁVEL....8
3.3. ANÁLISE COMPARATIVA: QUICKSORT COM PIVÔ FIXO, QUICKSORT COM PIVÔ ALEATÓRIO (SHUFFLE) E A IMPLEMENTAÇÃO DA BIBLIOTECA PADRÃO DO JAVA (ARRAYS.SORT)....9
4. ANÁLISE DAS DIFERENÇAS DE PERFORMANCE OBSERVADAS NOS ALGORITMOS DE BUSCA....10
4.1. ANÁLISE COMPARATIVA: BUSCA LINEAR ITERATIVA vs. BUSCA LINEAR RECURSIVA vs. BUSCA LINEAR ITERATIVA DUAS PONTAS....11
4.2. ANÁLISE COMPARATIVA: BUSCA BINÁRIA ITERATIVA vs. BUSCA BINÁRIA RECURSIVA....12
5. RESUMO DOS PRINCIPAIS APRENDIZADOS....13

1. INTRODUÇÃO

Este projeto foi desenvolvido no contexto da disciplina Laboratório de Estrutura de Dados, com o objetivo de analisar e comparar o desempenho de diferentes algoritmos de ordenação e busca. A proposta visa compreender, de forma empírica, como a complexidade teórica de cada algoritmo se manifesta na prática, considerando distintos tamanhos de entrada e cenários de execução.

Para isso, foram implementados e testados onze algoritmos de ordenação — Bubble Sort Clássico, Bubble Sort Otimizado, Selection Sort Clássico, Selection Sort Estável, Insertion Sort, Merge Sort Clássico, Tim Sort, Quick Sort, Quick Sort com embaralhamento (shuffle), Quick Sort (implementação do Java) e Counting Sort — e cinco algoritmos de busca — Busca Linear Iterativa, Busca Linear Recursiva, Busca Binária Iterativa, Busca Binária Recursiva e Busca Linear Iterativa Duas Pontas.

A análise de desempenho foi conduzida em um ambiente controlado, buscando eliminar interferências externas e assegurar a consistência dos resultados. Para cada algoritmo, foram considerados diferentes cenários de entrada e tamanhos de vetor, de modo a avaliar o comportamento nos melhores, piores e médios casos.

Além de reforçar o entendimento sobre os princípios e estruturas dos algoritmos estudados, este trabalho também pretende evidenciar as diferenças práticas de eficiência entre abordagens simples e otimizadas, bem como entre algoritmos com diferentes paradigmas, como *dividir para conquistar* e *baseados em comparação*.

2. METODOLOGIA DOS TESTES

Para uma análise mais consistente de desempenho dos algoritmos de busca e ordenação, o ambiente de execução foi configurado para minimizar a interferência de outros processos, garantindo que o computador fosse dedicado exclusivamente à execução dos algoritmos.

2.1. AMBIENTE DE TESTES

Todos os experimentos e análises apresentados neste relatório foram realizados em um computador pessoal com as seguintes especificações de hardware e software:

Hardware

- Processador (CPU): AMD Ryzen 5 5500 @ 3.6 GHz
- Número de núcleos / threads: 6 / 12
- Memória RAM: 16 GB (2×8 GB) DDR4 @ 3600 MHz
- Armazenamento: SSD SATA III 1 TB
- Placa de Vídeo: AMD Radeon RX 5500 XT 8 GB

Software

- Sistema Operacional: Windows 11 Pro 64 bits
- Versão do Java (JDK): Java 21.0.6 (javac 21.0.6)
- IDE utilizada para desenvolvimento: Eclipse IDE 2025-06 (4.36.0)

2.2. MEDIÇÃO DE TEMPO

A performance dos algoritmos foi medida em segundos, utilizando a função *System.nanoTime()* para obter uma precisão elevada. Os valores de tempo, originalmente medidos em nanossegundos, foram convertidos para segundos. Para os algoritmos de ordenação, os resultados foram apresentados com precisão de seis casas decimais, enquanto para os algoritmos de busca adotou-se uma precisão maior, de nove casas decimais, em razão do menor tempo de execução característico dessas operações.

2.3. NÚMERO DE EXECUÇÕES

Cada algoritmo, em cada cenário e com cada tamanho de vetor, foi executado 20 vezes. As 5 primeiras execuções de cada teste foram descartadas, servindo como "warm-up" para a máquina virtual Java (JVM) e minimizando a variabilidade de resultados. Para obter o tempo final, foi calculada a média das 15 execuções restantes

2.4. GERAÇÃO E TAMANHO DOS DADOS

Os testes foram realizados com três tamanhos de vetor distintos: 20k, 100k e 500k. Esses tamanhos foram escolhidos para avaliar como os algoritmos se comportam em diferentes escalas de dados.

Os dados foram gerados em Java como objetos do tipo Estudante. Os campos (nome, nota, matrícula) foram preenchidos com valores aleatórios, com notas inteiras entre 0 e 10. Para a comparação entre os QuickSorts, um experimento adicional foi realizado com arrays de inteiros.

2.5. CENÁRIOS DE TESTE

Para os algoritmos de ordenação, a análise de performance considerou três cenários de entrada:

- **Entrada Ordenada:** O vetor já se encontra na ordem natural (definida pelo método *compareTo*).
- **Entrada Aleatória:** Os elementos do vetor estão dispostos de forma totalmente aleatória.
- **Entrada Ordenada Inversamente:** O vetor está organizado de forma oposta à sua ordem natural.

Para os algoritmos de busca, a metodologia focou em avaliar o desempenho dos algoritmos em cenários que refletem o melhor, o médio e o pior caso de busca. Como os algoritmos de busca binária exigem um vetor ordenado, todos os testes de busca foram realizados sobre vetores previamente ordenados.

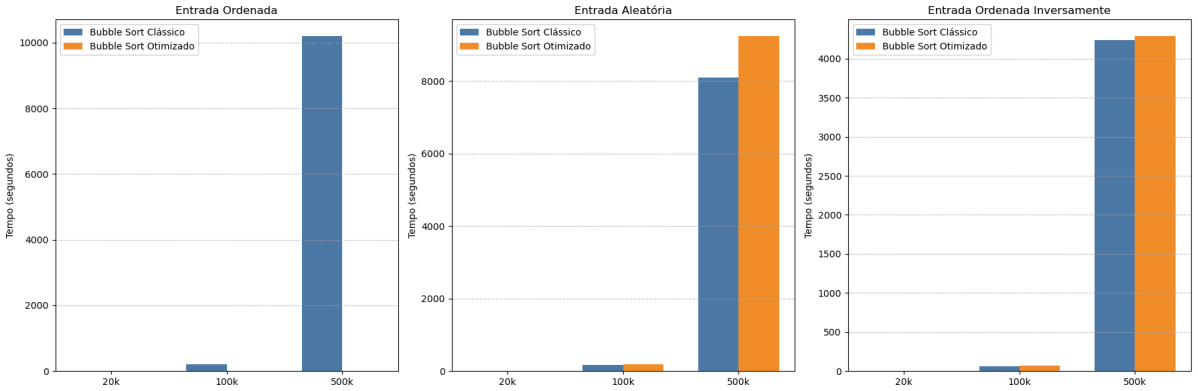
- **Melhor caso:** ocorre quando o elemento buscado é encontrado logo na primeira comparação realizada pelo algoritmo. Na busca sequencial, isso corresponde ao elemento estar na primeira posição do vetor. Já na busca binária, o melhor caso acontece quando o elemento procurado se encontra exatamente na posição central do vetor, sendo identificado na primeira iteração.
- **Caso médio:** caracteriza-se quando o elemento não está nem em uma posição imediatamente acessível (como no melhor caso) nem em uma das últimas possíveis (como no pior caso). Em geral, representa a situação em que o algoritmo realiza um número intermediário de comparações até localizar o elemento, refletindo o comportamento típico em uma execução comum.
- **Pior caso:** ocorre quando o elemento está na última posição do vetor ou não está presente nele, resultando no maior número possível de comparações antes da finalização da busca.

3. ANÁLISE DAS DIFERENÇAS DE PERFORMANCE OBSERVADAS NOS ALGORITMOS DE ORDENAÇÃO

Médias de tempo de execução em segundos dos algoritmos de ordenação									
	Entrada ordenada			Entrada aleatória			Entrada ordenada inversamente		
	n = 20k	n = 100k	n = 500k	n = 20k	n = 100k	n = 500k	n = 20k	n = 100k	n = 500k
Bubble sort Clássico	4,684737	209,07582	10192,28988	4,592464	165,795836	8101,88904	1,171494	66,347846	4240,78402
Bubble sort Otimizado	0,001422	0,006617	0,027852	4,174295	196,07257	9235,029864	1,144495	67,231682	4285,21035
Selection sort clássico	0,383071	10,331525	1618,505994	0,50827	16,674911	2595,16798	0,451423	14,12153	2301,158436
Selection Sort estável	0,276231	12,574647	1752,317941	0,403284	16,740321	2457,641692	0,515791	16,517403	2536,014792
Insertion Sort	0,001394	0,006711	0,033224	0,30002	9,758094	1323,406251	0,585896	21,157591	2704,176405
Merge Sort Clássico	0,002472	0,016468	0,118511	0,00604	0,037744	0,341523	0,002439	0,016391	0,099229
Tim Sort	0,001638	0,00765	0,035225	0,014086	0,047505	0,322791	0,00239	0,009407	0,047848
Quick Sort (versão do slide)	StackOverflowError	StackOverflowError	StackOverflowError	0,001124	0,006386	0,035673	StackOverflowError	StackOverflowError	StackOverflowError
Quick Sort + shuffle	0,00096	0,004044	0,020606	0,001531	0,008502	0,04632	0,000961	0,004258	0,021089
Quick Sort versão do java	0,000348	0,000456	0,000172	0,001383	0,00476	0,025643	0,000588	0,00048	0,000312
Counting Sort	0,000552	0,001825	0,020223	0,000469	0,002253	0,021169	0,000325	0,001983	0,018168

(acessar tabela)

3.1. ANÁLISE COMPARATIVA: BUBBLE SORT CLÁSSICO vs. BUBBLE SORT OTIMIZADO

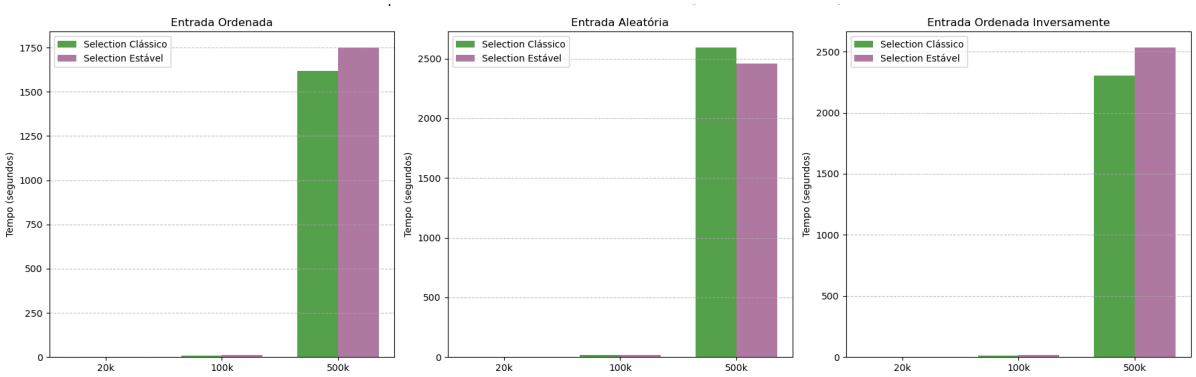


A principal divergência de desempenho entre as versões reside no tratamento de vetores previamente ordenados. A implementação otimizada reduz a complexidade quadrática ($O(n^2)$) para linear ($O(n)$) ao utilizar uma *flag* de verificação de trocas, interrompendo a execução prematuramente quando o vetor já está ordenado. Isso explica a diferença drástica de ~0,02s (otimizado) contra ~10.192s (clássico) para N=500k.

No entanto, em cenários de entrada aleatória ou inversamente ordenada, a otimização se mostra ineficaz. Como a ordenação exige iterações contínuas, a verificação constante da *flag* insere um custo computacional extra (*overhead*). Por isso, observa-se que a versão otimizada teve um desempenho ligeiramente inferior à clássica nestes casos (ex: 9.235s vs 8.101s no cenário aleatório).

Conclusão: A versão otimizada é vantajosa exclusivamente para dados quase ordenados; para dados completamente desordenados, o custo da verificação adicional anula seus benefícios.

3.2. ANÁLISE COMPARATIVA: SELECTION SORT CLÁSSICO vs. SELECTION SORT ESTÁVEL

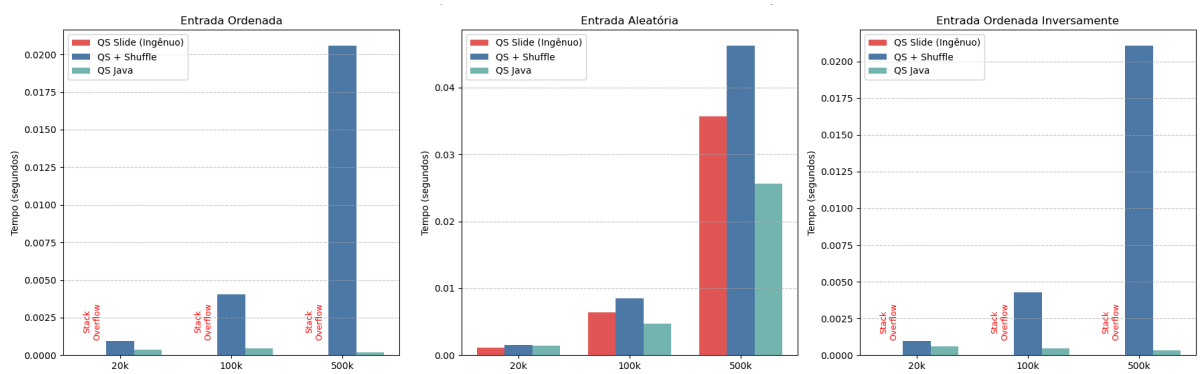


Ambas as versões do Selection Sort apresentam complexidade $O(n^2)$ e tempos de execução da mesma ordem de grandeza para grandes entradas (N=500 000). Contudo, a versão estável tende a ser mais lenta, especialmente nos cenários ordenado e inversamente ordenado, devido ao custo adicional necessário para preservar a ordem relativa de elementos com chaves iguais.

Esse overhead ocorre porque o algoritmo estável substitui trocas diretas por deslocamentos sucessivos de elementos no vetor, aumentando o número de operações de escrita. Embora no cenário aleatório a versão estável tenha apresentado desempenho pontualmente melhor, o comportamento geral indica que a estabilidade implica perda de eficiência.

Conclui-se que o Selection Sort clássico é preferível quando a estabilidade não é necessária, enquanto a versão estável deve ser utilizada apenas quando a preservação da ordem relativa dos dados é um requisito.

3.3. ANÁLISE COMPARATIVA: QUICKSORT COM PIVÔ FIXO, QUICKSORT COM PIVÔ ALEATÓRIO (SHUFFLE) E A IMPLEMENTAÇÃO DA BIBLIOTECA PADRÃO DO JAVA (ARRAYS.SORT)



Os experimentos realizados evidenciaram diferenças significativas de comportamento entre as variações do algoritmo QuickSort, especialmente em função da estratégia de seleção do pivô. Observou-se que, a partir de aproximadamente 14.000 elementos nos cenários ordenado e inversamente ordenado, a versão do QuickSort com pivô fixo (conforme apresentada nos slides) resultou em ***StackOverflowError*** na máquina de teste utilizada.

Na implementação com pivô fixo, o primeiro elemento do subvetor é sempre escolhido como pivô. Embora essa abordagem funcione adequadamente em entradas aleatórias, ela apresenta comportamento degenerado quando aplicada a vetores previamente ordenados. Nesses casos, a etapa de partição não consegue dividir o vetor em subconjuntos equilibrados, pois todos os elementos são classificados de forma uniforme em relação ao pivô. Como consequência, o índice retornado pela partição não reduz significativamente o tamanho do problema, fazendo com que o algoritmo realize chamadas recursivas sobre intervalos praticamente inalterados. Esse processo leva a um crescimento excessivo da profundidade da recursão, culminando no estouro da pilha.

Para mitigar esse problema, foi adotada uma versão do QuickSort com pivô aleatório, combinada com a partição de Hoare. A seleção aleatória do pivô reduz drasticamente a probabilidade de partições degeneradas, mesmo em vetores ordenados ou inversamente ordenados, ao eliminar padrões determinísticos na escolha do elemento de referência. A partição de Hoare, por sua vez, promove uma divisão mais robusta do vetor ao garantir o avanço contínuo dos ponteiros de varredura, independentemente da distribuição dos valores. Dessa forma, os subproblemas diminuem progressivamente de tamanho, assegurando a terminação da recursão e prevenindo o erro de StackOverflow, sem comprometer a complexidade média do algoritmo.

Comparando as três versões analisadas, observa-se que a versão com pivô fixo é a menos robusta, apresentando falhas graves em cenários adversos. A versão com pivô aleatório mostrou-se significativamente mais estável e confiável, mantendo tempos de execução consistentes mesmo para grandes volumes de dados e diferentes distribuições de entradas.

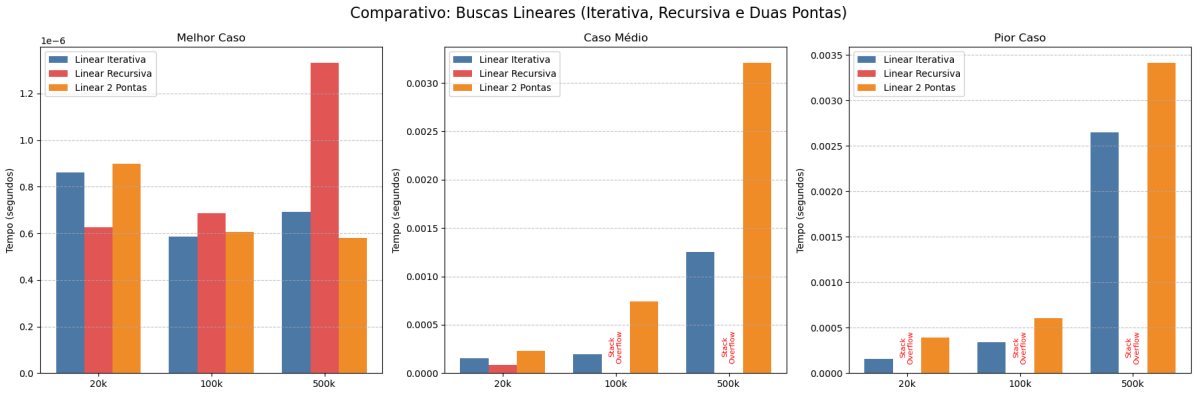
Por fim, a versão do Java, acessível por meio do método ***Arrays.sort***, apresentou o melhor desempenho geral. Para arrays de tipos primitivos, o Java utiliza o Dual-Pivot QuickSort, uma variação otimizada do QuickSort clássico que reduz a profundidade da recursão e melhora o balanceamento das partições. Essa implementação é altamente otimizada, robusta contra casos degenerados e adequada para uso prático, justificando seu desempenho superior em relação às implementações manuais analisadas.

4. ANÁLISE DAS DIFERENÇAS DE PERFORMANCE OBSERVADAS NOS ALGORITMOS DE BUSCA

Médias de tempo de execução em segundos dos algoritmos de busca									
	Melhor caso			Caso médio			Pior caso		
	n = 20k	n = 100k	n = 500k	n = 20k	n = 100k	n = 500k	n = 20k	n = 100k	n = 500k
Busca Linear Iterativa	0,00000086	0,000000587	0,000000693	0,00015214	0,000192187	0,001249453	0,00015382	0,00033 6893	0,00265 1133
Busca Linear Recursiva	0,00000 0627	0,00000 0687	0,00000 1333	0,00008 3253	StackOverflowError	StackOverflowError	StackOverflowError	StackOverflowError	StackOverflowError
Busca Binária Iterativa	0,00000 26	0,00000 2513	0,00000 286	0,00000 3007	0,00000 434	0,00000 3167	0,00000 1713	0,00000 1853	0,00000 1947
Busca Binária Recursiva	0,00000 356	0,00000 5247	0,00000 334	0,00000 2687	0,00000 5973	0,00000 3567	0,00000 4293	0,00000 2173	0,00000 3433
Busca Linear Iterativa Duas Pontas	0,00000 09	0,00000 0607	0,00000 058	0,00023 1853	0,00074 3953	0,00321 088	0,00038 9493	0,00060 4547	0,00341 4993

(acessar tabela)

4.1. ANÁLISE COMPARATIVA: BUSCA LINEAR ITERATIVA vs. BUSCA LINEAR RECURSIVA vs. BUSCA LINEAR ITERATIVA DUAS PONTAS



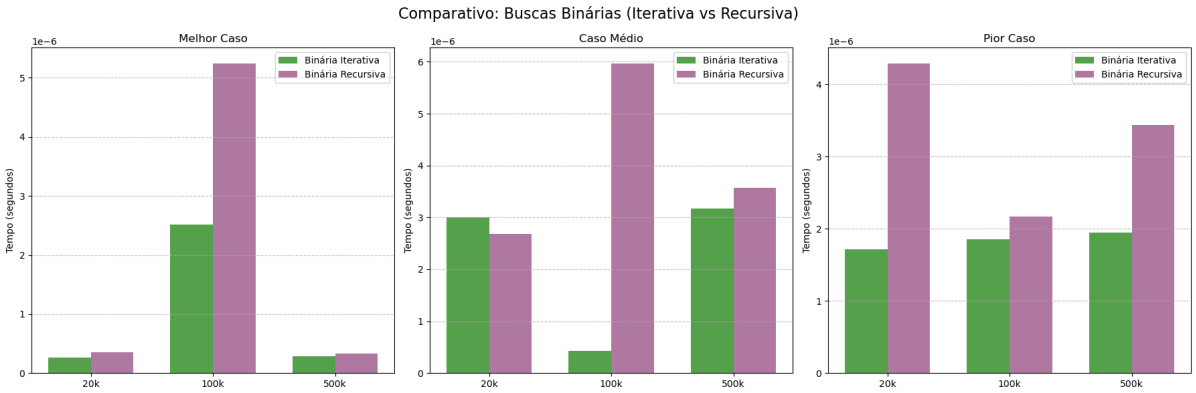
A análise dos resultados evidencia diferenças relevantes entre as variações da busca linear, especialmente quanto à robustez e ao custo da recursão. A busca linear iterativa apresentou comportamento estável em todos os cenários e tamanhos de entrada analisados, com crescimento gradual do tempo de execução conforme o aumento de n, como esperado para um algoritmo de complexidade O(n).

Em contrapartida, a busca linear recursiva mostrou-se inadequada para entradas maiores. Embora apresente tempos semelhantes à versão iterativa no melhor caso e para tamanhos reduzidos, essa abordagem resultou em StackOverflowError para valores superiores a aproximadamente 20.000 elementos, tanto no caso médio quanto no pior caso. Esse comportamento ocorre devido à profundidade excessiva da recursão, uma vez que cada chamada recursiva consome espaço adicional na pilha, tornando a implementação inviável para grandes volumes de dados.

Embora a busca linear com duas pontas reduza o número de iterações ao verificar simultaneamente elementos nas extremidades do vetor, os resultados experimentais indicam que essa abordagem apresentou maior tempo de execução em comparação à busca linear iterativa tradicional, tanto no caso médio quanto no pior caso.

Esse comportamento pode ser atribuído ao maior custo por iteração, decorrente do aumento no número de comparações e acessos à memória, o que anula o ganho teórico de reduzir o número de iterações. Assim, apesar de manter complexidade assintótica O(n), a versão com duas pontas mostrou-se menos eficiente na prática para o ambiente específico de teste considerado.

4.2. ANÁLISE COMPARATIVA: BUSCA BINÁRIA ITERATIVA vs. BUSCA BINÁRIA RECURSIVA



Os resultados obtidos indicam que tanto a busca binária iterativa quanto a recursiva apresentaram excelente desempenho em todos os cenários analisados, com tempos de execução significativamente inferiores aos das buscas lineares. Esse comportamento é consistente com a complexidade O(log n) do algoritmo, que garante crescimento muito lento do tempo de execução mesmo para grandes valores de n.

Observa-se que as diferenças de tempo entre as versões iterativa e recursiva são mínimas e pouco significativas do ponto de vista prático. No entanto, a versão iterativa tende a apresentar tempos ligeiramente menores, especialmente para entradas maiores, devido à ausência do overhead associado às chamadas recursivas.

Diferentemente do observado na busca linear recursiva, a busca binária recursiva não apresentou problemas de StackOverflow. Isso ocorre porque a profundidade da recursão cresce de forma logarítmica, permanecendo suficientemente pequena mesmo para grandes vetores.

Dessa forma, ambas as implementações são adequadas do ponto de vista funcional. Contudo, a busca binária iterativa é geralmente preferida por ser mais eficiente em termos de uso de memória e por evitar o custo adicional da recursão, enquanto a versão recursiva pode ser utilizada quando se busca maior clareza conceitual ou didática.

5. RESUMO DOS PRINCIPAIS APRENDIZADOS

A realização deste estudo e experimento possibilitou uma compreensão mais aprofundada sobre o comportamento prático de algoritmos clássicos de ordenação e busca. Observou-se que escolhas de implementação, como a estratégia de seleção de pivô no QuickSort ou o uso de recursão, exercem impacto significativo tanto no tempo de execução quanto no consumo de recursos, podendo levar a comportamentos indesejados, como o aumento excessivo da profundidade da recursão e a ocorrência de erros de estouro de pilha. Em particular, verificou-se que estratégias ingênuas, como o uso de pivô fixo em vetores ordenados, tornam o algoritmo suscetível ao pior caso com alta frequência, enquanto abordagens mais robustas, como pivô aleatório e partição de Hoare, mitigam esses problemas.

No contexto dos algoritmos de busca, ficou evidente que implementações iterativas tendem a ser mais estáveis e seguras para grandes volumes de dados quando comparadas às versões recursivas, especialmente em algoritmos de complexidade linear. Além disso, confirmou-se a superioridade das buscas binárias sobre as lineares em vetores ordenados, tanto em termos de desempenho quanto de escalabilidade.

De modo geral, o trabalho reforçou a importância de considerar não apenas a complexidade assintótica, mas também fatores como estabilidade, profundidade da recursão, overhead de implementação e características da entrada de dados. Esses aspectos são determinantes para a escolha adequada de algoritmos em aplicações reais, destacando a necessidade de decisões fundamentadas tanto em teoria quanto em experimentação prática.