Comparação de Desempenho entre Algoritmos de Ordenação: Um Estudo Sobre Seus Comportamentos em Diferentes Cenários

Paulo E. R. Araujo[1], **Carlos M. Sousa**[1], **Gabriel R. Nunes**[1]

[1]Departamento de Computação - Universidade Federal do Piauí (UFPI)
Teresina - PI

pumbadeveloper@gmail.com

Abstract. *Asymptotic analysis is widely used to evaluate the performance of algorithms, however, this approach is not always able to highlight significant differences in performance between algorithms with the same complexity. In this paper, an experimental performance analysis of several sorting algorithms is presented, aiming to compare and evaluate their behaviors in different scenarios. In addition, a hybrid algorithm is proposed with the aim of obtaining a good performance in all scenarios. A simulator has been developed to receive the algorithms and the simulation parameters, generating comparative graphs with the results obtained. This experimental analysis serves as an additional criterion to evaluate and differentiate the performance of the algorithms under study, providing valuable data for the appropriate selection of algorithms in real contexts.*

Resumo. *A análise assintótica é amplamente empregada para avaliar o desempenho dos algoritmos, porém, essa abordagem nem sempre é capaz de evidenciar diferenças significativas de desempenho entre algoritmos com a mesma complexidade. Neste artigo, é apresentada uma análise experimental de desempenho de diversos algoritmos de ordenação, com o objetivo de comparar e avaliar seus comportamentos em diferentes cenários. Além disso, um algoritmo híbrido é proposto com o objetivo de obter um bom desempenho em todos os cenários. Um simulador foi desenvolvido para receber os algoritmos e os parâmetros da simulação, gerando gráficos comparativos com os resultados obtidos. Essa análise experimental serve como um critério adicional para avaliar e diferenciar o desempenho dos algoritmos em estudo, fornecendo dados valiosos para a seleção apropriada de algoritmos em contextos reais.*

1. Introdução

Os algoritmos de ordenação desempenham um papel fundamental na computação e têm sido extensivamente estudados ao longo dos anos. Este artigo apresenta uma análise comparativa de cinco algoritmos clássicos de ordenação: Insertion Sort, Bubble Sort, Merge Sort, Heap Sort e Quick Sort. Esses algoritmos foram selecionados com base no livro ”Algoritmos: Teoria e Prática” [Cormen et al. 2009], como referência principal para o estudo.

O objetivo deste trabalho é investigar o desempenho desses algoritmos em diferentes cenários, avaliando o tempo de execução e a quantidade de comparações realizadas.

Para realizar essa análise, um simulador foi implementado para executar os algoritmos em diferentes conjuntos de dados e gera gráficos com os resultados obtidos. Dessa forma, o desempenho de cada algoritmo pode ser comparado em diferentes casos de entrada.

Além disso, neste trabalho, é apresentado um algoritmo híbrido denominado AOH (Algoritmo de Ordenação Híbrido), que visa obter o melhor desempenho em todos os cenários. O AOH combina técnicas dos algoritmos estudados para explorar as vantagens de cada um deles e tentar alcançar resultados superiores. O desempenho do AOH será avaliado em comparação com os algoritmos clássicos, a fim de verificar se é possível obter um algoritmo de ordenação mais eficiente em termos de tempo de execução e número de comparações.

A estrutura restante deste trabalho é organizada da seguinte maneira: A Sessão 2 aborda os algoritmos de ordenação e suas particularidades, explorando as características distintas desses algoritmos. Em seguida, na Seção 3, é apresentado o algoritmo híbrido AOH, onde são discutidos os aspectos considerados em sua construção para alcançar um desempenho aprimorado em diferentes cenários. Os detalhes sobre a metodologia adotada e os parâmetros utilizados para simular e avaliar o desempenho dos algoritmos estudados são fornecidos na Seção 4. Na Seção 5, são apresentados os resultados obtidos durante o estudo, oferecendo análises e interpretações dos dados coletados. Por fim, na Seção 6, são apresentadas as conclusões deste estudo, destacando as principais descobertas derivadas da análise dos dados.

2. Algoritmos de Ordenação

Os algoritmos de ordenação desempenham um papel crucial na área da ciência da computação, proporcionando meios eficientes para organizar e classificar conjuntos de dados. A capacidade de ordenar informações é fundamental em uma ampla variedade de aplicações, desde a organização de bancos de dados até a solução de problemas complexos em algoritmos de busca e otimização.

Além disso, os algoritmos de ordenação têm uma gama de aplicações em várias áreas da ciência da computação. Eles são utilizados em algoritmos de busca, algoritmos de compressão de dados, algoritmos de processamento de imagens, algoritmos de aprendizado de máquina e muitos outros. A eficiência e a corretude desses algoritmos têm um impacto direto no desempenho e na precisão dessas aplicações [freeCodeCamp].

A escolha adequada do algoritmo de ordenação é de extrema importância, pois diferentes algoritmos possuem características distintas que os tornam mais adequados para certos cenários e conjuntos de dados específicos. A seleção incorreta do algoritmo de ordenação pode resultar em ineficiência computacional, desperdício de recursos e desempenho inadequado da aplicação final.

Cada algoritmo de ordenação possui sua própria complexidade de tempo, espaço e desempenho em diferentes cenários. Alguns algoritmos são mais eficientes em conjuntos de dados pequenos, enquanto outros são projetados para lidar com grandes volumes de dados de maneira mais eficiente. Além disso, a distribuição inicial dos dados, como a presença de dados parcialmente ordenados ou inversamente ordenados, pode influenciar significativamente o desempenho de determinados algoritmos.

2.1. Insertion Sort

O Insertion Sort, ou ordenação por inserção, é um algoritmo de ordenação estável amplamente estudado. Ele realiza repetidamente uma rotina de inserção para obter uma sequência ordenada. Esse algoritmo é caracterizado por realizar trocas com seus vizinhos, movendo os elementos um por um, em vez de realizar saltos de diferença entre eles. Portanto, o Insertion Sort mantém a ordem relativa dos valores iguais, tornando-o um algoritmo de ordenação estável.

Uma das principais vantagens do Insertion Sort é sua simplicidade de implementação. Além disso, ele opera no modo *in-place*, ou seja, a ordenação é realizada diretamente no vetor de elementos desordenados, sem a necessidade de vetores ou estruturas auxiliares. Essa característica torna o algoritmo eficiente em termos de espaço de memória.

Embora o Insertion Sort seja menos eficiente do que algoritmos como o Merge Sort e o Quick Sort, que possuem uma complexidade assintótica de $O(nlogn)$, ele possui algumas características notáveis. O algoritmo é particularmente útil para conjuntos de dados pequenos ou com um número considerável de trocas, pois realiza menos comparações em relação a outros algoritmos de complexidade semelhante.

A complexidade do Insertion Sort varia de acordo com o vetor de entrada. No melhor caso, com vetor de entrada já está ordenado, o algoritmo precisa realizar apenas uma passagem pelo vetor para verificar que ele está ordenado corretamente. Portanto, a complexidade do melhor caso é de $O(n)$, onde *n* é o tamanho do vetor. O caso médio do é uma média entre todas as possíveis entradas. Estatisticamente falando, o caso médio terá uma complexidade de tempo quadrática, ou seja, $O(n^2)$. No pior caso, o vetor de entrada está ordenado em ordem decrescente. Nesse cenário, o algoritmo precisa realizar uma comparação e uma troca para cada elemento do vetor, resultando em um total de $\sum_{i=1}^{n-1} i$ comparações e trocas. Simplificando essa soma, obtemos $\frac{n^2}{2} - \frac{n}{2}$. Portanto, a complexidade do pior caso é de $O(n^2)$ [Wikipedia (Insertion)].

2.2. Bubble Sort

O Bubble Sort, ou ordenação por flutuação, é outro algoritmo clássico de ordenação amplamente estudado. Ele percorre repetidamente o vetor de entrada, comparando elementos adjacentes e realizando trocas caso necessário, até que todo o vetor esteja ordenado.

Uma das principais características do Bubble Sort é a sua simplicidade de implementação. O algoritmo percorre o vetor várias vezes, movendo elementos em direção à posição correta, até que nenhuma troca seja necessária. Dessa forma, os elementos ”flutuam” gradualmente para suas posições finais, daí o nome do algoritmo. Assim como o Insertion Sort, o Bubble Sort é um algoritmo estável, preservando a ordem relativa dos valores iguais durante as trocas. Ele também opera *in-place*, não requerendo estruturas de dados auxiliares além do próprio vetor a ser ordenado.

No entanto, o Bubble Sort é conhecido por ter um desempenho menos eficiente em relação a outros algoritmos de ordenação, especialmente para grandes conjuntos de dados. No melhor caso, quando o vetor de entrada já está ordenado, o Bubble Sort precisa fazer apenas uma passagem pelo vetor para verificar que ele está ordenado corretamente. Portanto, a complexidade do melhor caso é de $O(n)$. O caso médio do Bubble Sort

é uma média entre todas as possíveis entradas. Então, o caso médio tende a ter uma complexidade de tempo quadrática, ou seja, $O(n^2)$. No pior caso, o vetor está em ordem decrescente. Nesse cenário, o Bubble Sort precisa realizar uma comparação e uma troca para cada par de elementos adjacentes do vetor em cada passagem. Isso resulta em $\sum_{i=1}^{n-1} i$ comparações e trocas, que é equivalente a $\frac{n^2}{2} - \frac{n}{2}$. Portanto, a complexidade do pior caso é de $O(n^2)$ [Wikipedia (Bubble)].

Apesar de suas limitações, o Bubble Sort pode ser útil em algumas situações específicas, como quando o vetor de entrada já está quase ordenado ou possui um número muito pequeno de elementos. Além disso, devido à sua simplicidade, o Bubble Sort pode ser usado como um ponto de partida para entender os conceitos básicos de ordenação antes de explorar algoritmos mais avançados.

2.3. Merge Sort

O Merge Sort opera seguindo a abordagem de ”dividir para conquistar”, dividindo repetidamente o conjunto de dados em subconjuntos menores, até que cada subconjunto contenha apenas um elemento. Em seguida, ele combina e mescla gradualmente os subconjuntos, aplicando a operação de fusão (*merge*) para obter uma sequência ordenada. Esse processo ocorre recursivamente até que todos os subconjuntos sejam mesclados em um único conjunto ordenado.

Uma das principais vantagens do Merge Sort é sua estabilidade. Isso se deve ao fato de que, durante a fase de *merge*, quando dois elementos são considerados iguais, o Merge Sort prioriza a seleção do elemento do subconjunto original em que ele estava localizado, mantendo a ordem original.

O Merge Sort também se destaca pela sua eficiência em lidar com grandes conjuntos de dados. Devido à sua abordagem, ele é capaz de dividir o problema em subproblemas menores, facilitando o processamento paralelo e reduzindo o tempo de execução global. Além disso, o Merge Sort requer um espaço de memória adicional para armazenar os subconjuntos durante o processo de *merge*, tornando-o menos adequado para conjuntos de dados com limitações de memória.

No que diz respeito à complexidade de tempo, o Merge Sort possui uma complexidade assintótica garantida de *O(n log n)*. Isso significa que, à medida que o tamanho do conjunto de dados aumenta, o tempo de execução do algoritmo cresce proporcionalmente ao logaritmo do tamanho do conjunto de dados. Essa eficiência torna o Merge Sort uma escolha popular em situações em que o desempenho é uma consideração crítica. [Wikipedia (Merge)]

No entanto, é importante destacar que o Merge Sort possui uma necessidade de armazenamento adicional de *O(n)*, devido à necessidade de armazenar os subconjuntos durante o processo de fusão. Portanto, em casos de conjuntos de dados extremamente grandes, é fundamental considerar a disponibilidade de memória ao escolher o Merge Sort como algoritmo de ordenação.

2.4. Heap Sort

O algoritmo de ordenação Heap Sort, assim como o Merge Sort, segue a abordagem de ”dividir para conquistar”, dividindo o conjunto de dados em subconjuntos menores

e aplicando operações de reorganização afim de se obter uma sequência ordenada. No Heap Sort, é utilizado uma estrutura auxiliar chamada de Heap, que consiste em uma árvore binária onde o pai deve ser maior que seus filhos para garantir a integridade da estrutura. Caso contrário, a árvore é reorganizada, trocando o pai com o filho, até que a condição seja satisfeita. Dessa forma, o maior número da árvore estará sempre na raiz. No final, a raiz é trocada com a última folha e o tamanho do vetor é reduzido até que o vetor tenha tamanho 0, resultando em um vetor ordenado do menor para o maior.

Uma das características notáveis do Heap Sort é sua eficiência na ordenação de conjuntos de dados de grande porte. A construção inicial do Heap tem uma complexidade de tempo de *O(n)*, onde *n* representa o tamanho do conjunto de dados. A ordenação em si ocorre em duas etapas: a troca de elementos, que tem uma complexidade de tempo constante (*O(1)*), e o ajuste da raiz, que possui uma complexidade de tempo logarítmica (*O(log n)*). Essas duas etapas são executadas um total de *(n-1)* vezes, resultando em uma complexidade total de *O((n-1) log n)* [Wikipedia (Heap)].

No que diz respeito aos diferentes cenários de aplicação, a complexidade do Heap Sort permanece a mesma. Tanto no melhor caso, onde o vetor já está parcialmente ordenado, quanto no caso médio, com entradas aleatórias, e no pior caso, com entradas inversamente ordenadas, a complexidade do Heap Sort é $\Theta(n \log n)$. Isso significa que o desempenho do algoritmo é garantido em qualquer cenário de uso, independentemente do estado inicial do conjunto de dados.

No entanto, o Heap Sort requer uma estrutura de dados adicional, o Heap, para realizar a ordenação. Isso resulta em um consumo de memória adicional, o que pode ser uma consideração relevante em sistemas com recursos limitados.

2.5. Quick Sort

O algoritmo Quick Sort, assim como os anteriores, segue a abordagem de ”dividir para conquistar”, onde divide recursivamente o array em subarrays menores, ordenando-os e, em seguida, combinando-os para obter uma ordenação completa. É um dos algoritmos de ordenação mais amplamente utilizados devido à sua eficiência e desempenho. Essas características são melhores evidenciadas quando aplicada a arrays desordenados.

O Quick Sort opera selecionando um elemento como pivô e particionando o array em duas partes: uma contendo elementos menores que o pivô e outra contendo elementos maiores. Em seguida, ele recursivamente aplica a mesma estratégia nos subarrays resultantes até que todo o array esteja ordenado. Com isso, a escolha do pivô é de suma importância e pode afetar o desempenho do algoritmo.

Um dos problemas do Quick Sort é que, em determinadas situações, pode ocorrer um desequilíbrio na partição, resultando em uma execução ineficiente. Isso acontece quando o pivô escolhido é um elemento extremamente grande ou pequeno em relação aos demais elementos do array. Nesses casos, o Quick Sort pode ter uma complexidade de tempo de execução quadrática, tornando-o menos eficiente.

Para mitigar esse problema, foi desenvolvido o Quick Sort randomizado. Nesse algoritmo, o pivô é escolhido de forma aleatória a cada chamada recursiva. Essa abordagem elimina o viés na escolha do pivô, garantindo uma distribuição mais uniforme dos elementos nos subarrays. Isso reduz significativamente a probabilidade de ocorrer um

desempenho ruim no Quick Sort.

No caso médio e no melhor caso, o Quick Sort tem uma complexidade de tempo de execução média de $O(nlogn)$. No pior caso, quando ocorre um desequilíbrio na partição, a complexidade pode chegar a $O(n^2)$. No entanto, com a randomização no Quick Sort, a complexidade média permanece em $O(nlogn)$, enquanto a probabilidade de ocorrer o pior caso é consideravelmente reduzida [IME USP].

3. Algoritmo de Ordenação Híbrido

Levando em consideração que os diferentes algoritmos de ordenação estudados possuem desempenhos variados dependendo do cenário em que são aplicados, este artigo apresenta o algoritmo de ordenação híbrido (AOH). O objetivo do AOH é obter um desempenho otimizado em todos os cenários de entrada, considerando tanto arrays pequenos quanto grandes, bem como casos em que o array já está parcial ou totalmente ordenado.

O algoritmo AOH faz uma análise prévia do vetor a ser ordenado e escolhe o algoritmo mais adequado para lidar com o cenário específico. Para arrays de tamanho reduzido, o AOH utiliza o algoritmo Insertion Sort, devido ao seu bom desempenho em cenários com poucos elementos. Em contrapartida, quando o tamanho do array ultrapassa um determinado limiar, o AOH verifica se o array já está ordenado utilizando um algoritmo linear de verificação. Se o array não estiver ordenado, o algoritmo Quick Sort Randomizado é selecionado devido à sua eficiência em ordenar listas desordenadas de grande porte. Por outro lado, se o array já estiver ordenado, não é necessário utilizar nenhum algoritmo de ordenação.

O limiar escolhido para definir o tamanho pequeno ou grande do array foi baseado nos experimentos realizados com os algoritmos estudados. Observou-se que o Insertion Sort perde seu desempenho para arrays com mais de 500 elementos. Portanto, esse valor de limiar foi estabelecido para extrair o melhor desempenho de cada algoritmo, aplicando-os aos cenários em que se destacam.

O algoritmo AOH será comparado com os outros algoritmos estudados para avaliar sua eficácia e desempenho. Os resultados dessas comparações serão apresentados na Seção 5, fornecendo uma análise abrangente sobre o desempenho do AOH em relação aos demais algoritmos de ordenação.

4. Simulação

Para realizar uma análise abrangente e comparativa dos algoritmos de ordenação estudados e o algoritmo proposto, foi desenvolvido um simulador utilizando a linguagem JavaScript e o ambiente Node.js [SAPS]. O simulador tem como objetivo gerar cenários de teste variados, que incluem vetores ordenados, vetores inversamente ordenados e vetores aleatórios. Esses cenários são utilizados para executar os algoritmos de ordenação e coletar informações sobre o desempenho de cada algoritmo.

A fim de evitar resultados extremos e garantir uma avaliação mais precisa, o simulador realiza múltiplas replicatas da execução de cada algoritmo em um mesmo cenário e calcula a média dos tempos de execução. Além disso, é feita uma validação das respostas obtidas pelos algoritmos para garantir a corretude dos resultados.

A avaliação de desempenho dos algoritmos ocorre em duas etapas, considerando o tamanho das entradas. A primeira etapa abrange conjuntos de dados menores que 2.500 elementos, enquanto a segunda etapa aborda conjuntos maiores, variando de 10.000 a 250.000 elementos. Os algoritmos Insertion Sort e Bubble Sort foram excluídos da análise devido ao seu desempenho inferior em relação aos demais algoritmos para tamanhos consideráveis. Durante a avaliação, são consideradas duas métricas principais: o tempo de execução em milissegundos e a quantidade de comparações realizadas pelos algoritmos.

Durante os testes realizados para este artigo, foram executadas 10 replicações para cada cenário, visando obter uma média confiável dos resultados. Os testes foram conduzidos utilizando as seguintes configurações de hardware:

Arquitetura	x64
Sistema Operacional	Windows 11
Processador	Intel(R) Core(TM) i5-10400 CPU @ 2.90GHz
Núcleos do Processador	12
Memória RAM Total	16 GB
Velocidade da Memória RAM	2666 MHz

Table 1. Especificações do hardware utilizado para avaliação de desempenho

5. Resultados

5.1. Avaliação dos Algoritmos para Entradas Pequenas

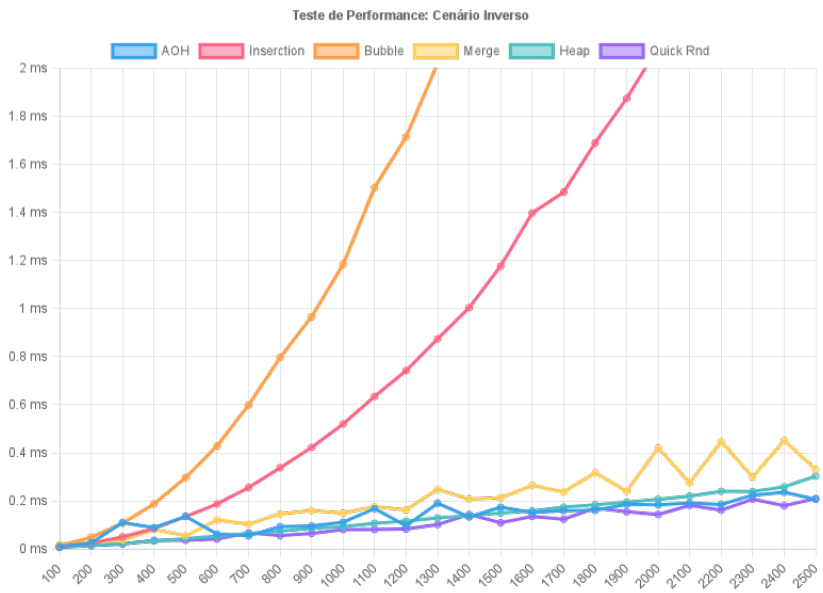

Figure 1. Tempo de execução para entradas pequenas e inversas.

Para tamanhos de entradas relativamente pequenos e inversamente ordenados, o algoritmo proposto apresenta o desempenho esperado para o seu funcionamento. A Figura 1 evidencia o uso do Insertion Sort no algoritmo híbrido AOH, onde os comportamentos dos dois algoritmos são semelhantes até o ponto definido como limite. Após esse ponto,

o comportamento do AOH passa a se assemelhar ao do Quick Sort Randomizado, uma vez que a partir desse ponto o AOH delega ao Quick Sort a responsabilidade de ordenar o vetor.

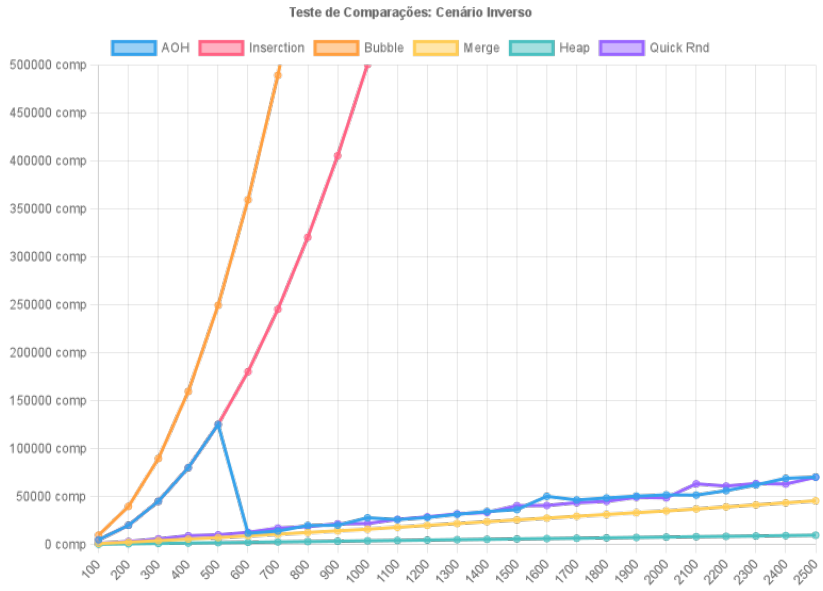

Figure 2. Quantidade de comparações para entradas pequenas e inversas.

Essa transição entre os algoritmos também pode ser observada na Figura 2, na qual o número de comparações do AOH é exatamente igual ao do Insertion Sort. À medida que o limite de tamanho é ultrapassado, o numero de comparações do AOH se aproxima do Quick Sort Randomizado, porém com variações decorrentes da escolha do pivô.

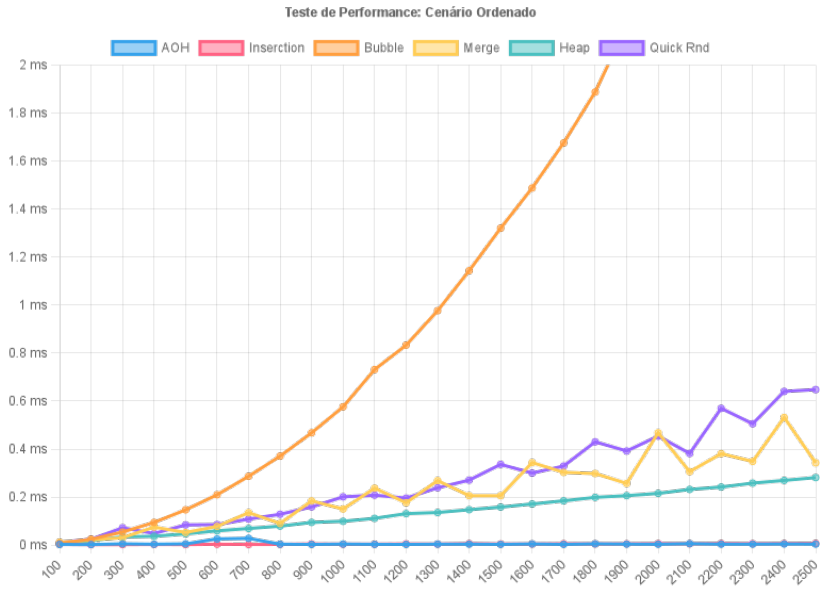

Figure 3. Tempo de execução para entradas pequenas e ordenadas.

Na Figura 3, observa-se que o comportamento do algoritmo híbrido AOH é semelhante ao comportamento do Insertion Sort. Isso ocorre devido ao fato de que, assim como o AOH, quando o Insertion Sort é executado em um cenário já ordenado, ele simplesmente itera sobre os elementos, verificando se o elemento subsequente é maior que o anterior. Essa semelhança entre os algoritmos também pode ser constatada na Figura 4, na qual o número de comparações realizadas pelos dois algoritmos é exatamente igual.

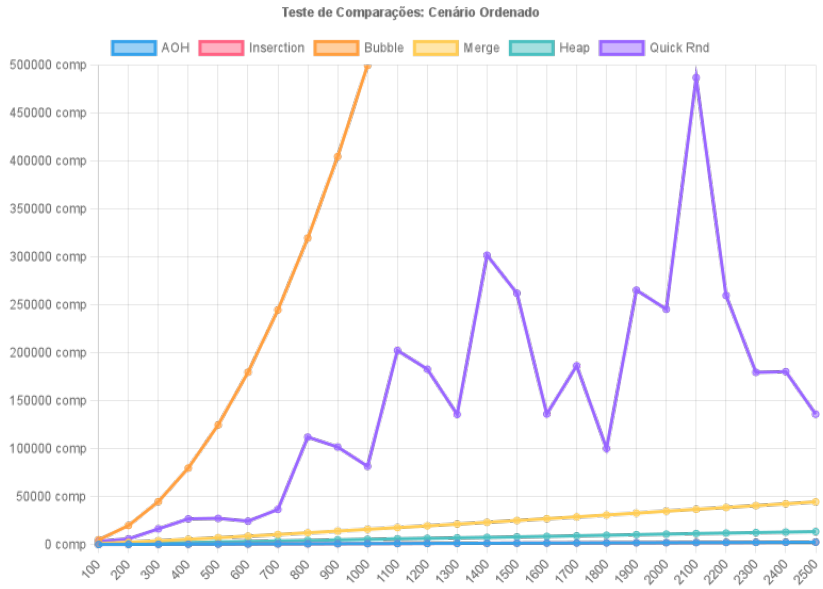

Figure 4. Quantidade de comparações para entradas pequenas e ordenadas.

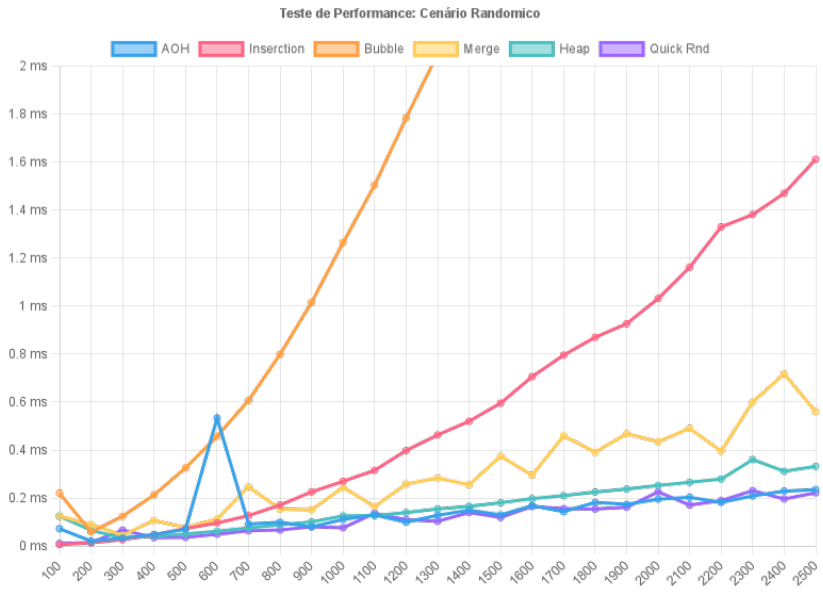

Figure 5. Tempo de execução para entradas pequenas e aleatórias.

A Figura 5 apresenta o tempo de execução dos algoritmos para entradas aleatórias

de pequeno porte, na qual pode-se observar um desempenho favorável do algoritmo híbrido AOH. Durante um momento específico da simulação, com o valor de n igual a 600, ocorre um pico no tempo de execução do algoritmo proposto logo após o limiar estabelecido para a troca dos algoritmos de ordenação, seguido de uma rápida normalização nos instantes subsequentes. Suspeita-se que essa alteração nos resultados seja decorrente de algum overhead presente no ambiente, tanto em termos de software quanto de hardware, necessário para a transição entre conjuntos de instruções distintos utilizados pelo sistema.

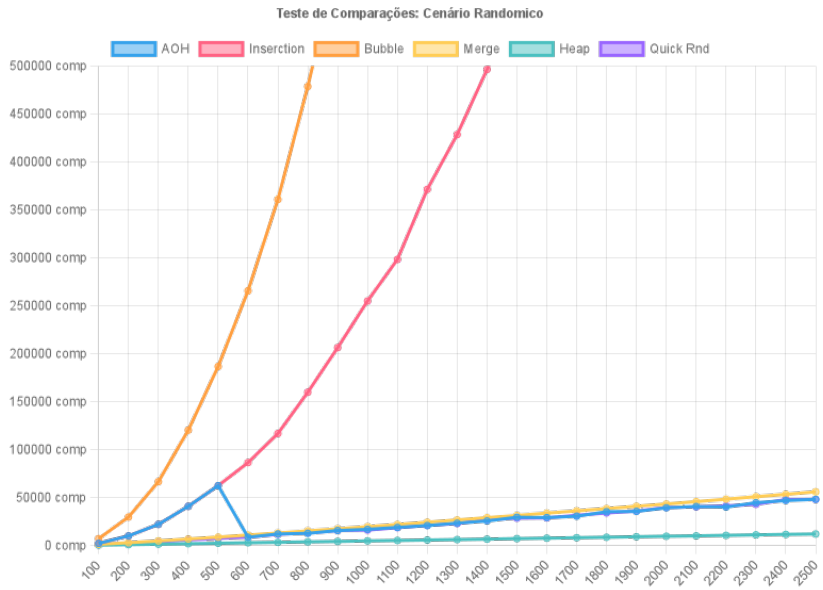

Figure 6. Quantidade de comparações para entradas pequenas e aleatórias.

A Figura 6 ilustra a quantidade de comparações realizadas pelos algoritmos no caso médio. Nessa figura, é possível novamente observar o funcionamento do algoritmo híbrido proposto, no qual o número de comparações se mantém igual ao do Inserction Sort até atingir o limite estabelecido. Após esse ponto, o comportamento das comparações assemelha-se ao do Quick Sort Randomizado, com variações decorrentes da escolha do pivô. Esses resultados evidenciam a capacidade do algoritmo híbrido em adaptar-se aos diferentes cenários de ordenação, proporcionando um desempenho adequado para cada um deles.

5.2. Avaliação dos Algoritmos para Entradas Grandes

Para tamanhos de entrada suficientemente grandes, os algoritmos de ordenação Insertion Sort e Bubble Sort foram descartados devido ao alto crescimento no tempo de processamento em relação aos demais algoritmos analisados. No entanto, é importante ressaltar que o Insertion Sort apresenta um desempenho favorável em cenários de entradas já ordenadas, especialmente em casos de grande porte. Nesses cenários específicos, o Insertion Sort demonstra ser uma opção eficiente para a ordenação, proporcionando um tempo de processamento mais rápido em comparação com outros algoritmos.

A Figura 7 apresenta o tempo de execução dos algoritmos no pior caso para

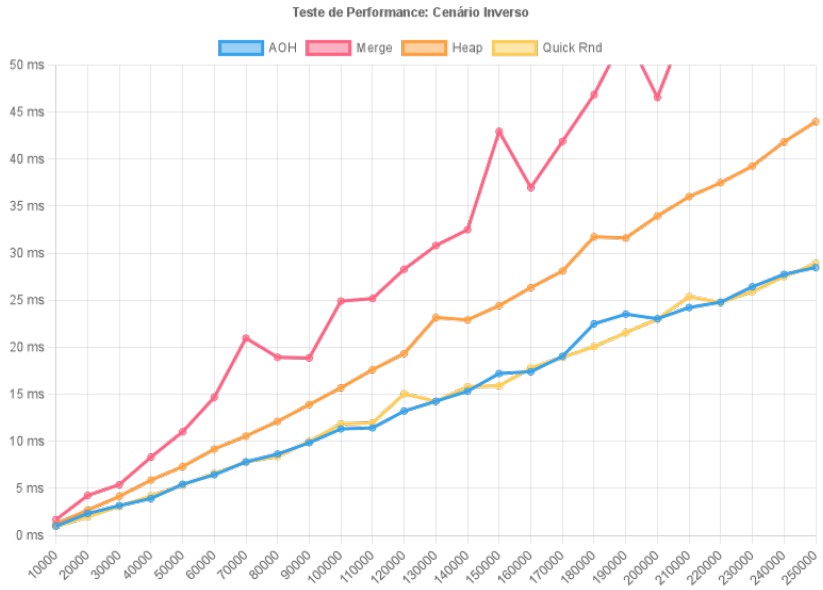

Figure 7. Tempo de execução para entradas grandes e inversas.

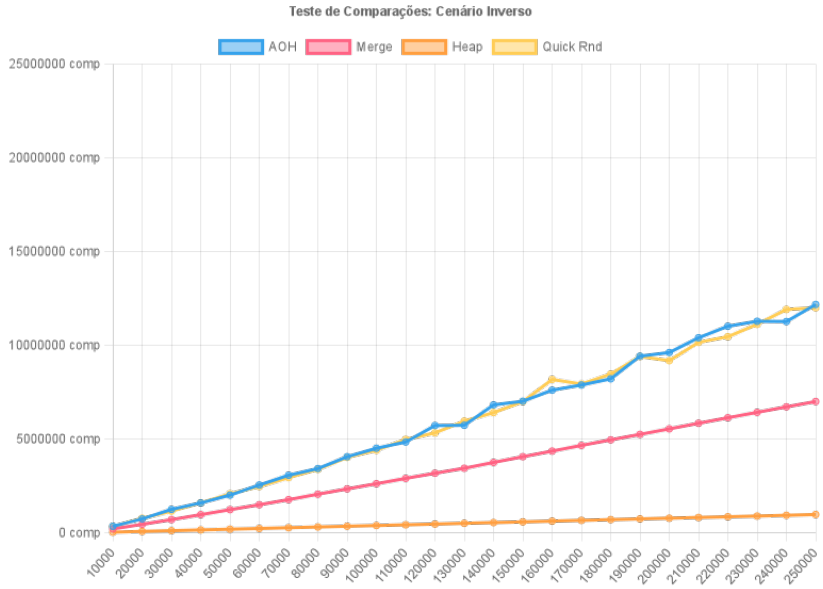

Figure 8. Quantidade de comparações para entradas grandes e inversas.

cenários de grande porte. Nesse contexto, o algoritmo híbrido AOH demonstra um desempenho semelhante ao Quick Sort Randomizado, uma vez que é o próprio Quick Sort que realiza a ordenação do vetor. Esse comportamento é corroborado pela Figura 8, que exibe a quantidade de comparações realizadas nesse cenário.

As Figuras 9 e 10 ilustram o comportamento dos algoritmos para entradas grandes já ordenadas. Nesse cenário, o algoritmo híbrido proposto utiliza um algoritmo linear de verificação, o que resulta em um tempo de processamento semelhante ao observado no

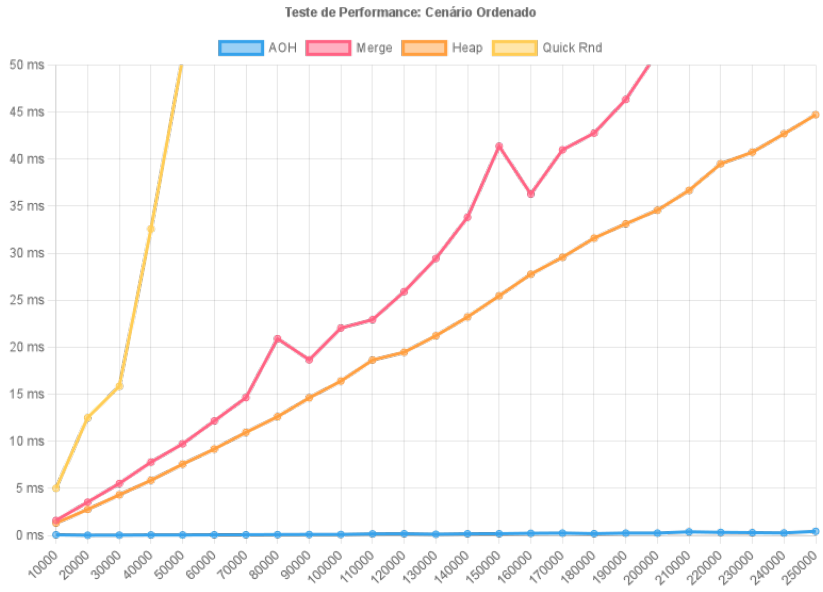

Figure 9. Tempo de execução para entradas grandes e ordenadas.

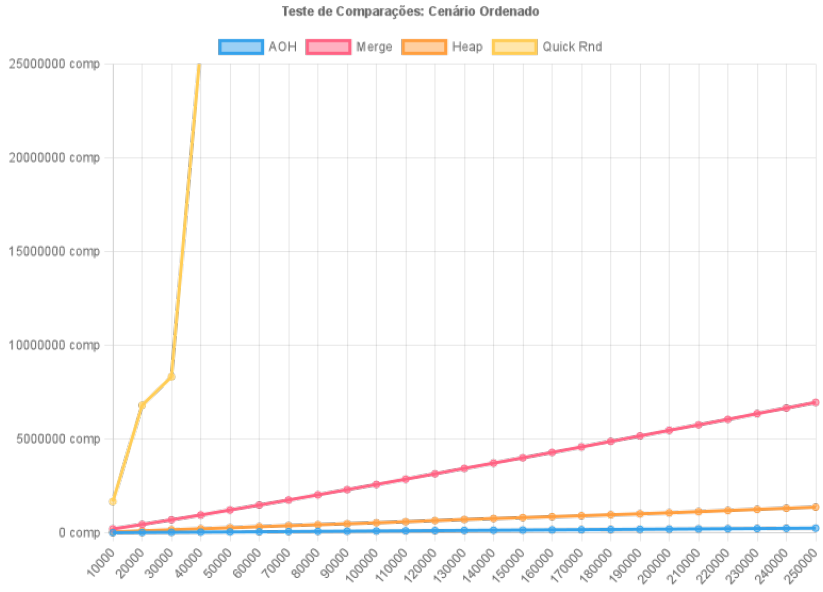

Figure 10. Quantidade de comparações para entradas grandes e ordenadas.

Insertion Sort. Essa abordagem permite otimizar o tempo de execução, uma vez que o vetor já está ordenado e não é necessário realizar operações de ordenação mais complexas.

As Figuras 11 e 12 apresentam o comportamento dos algoritmos em relação a entradas grandes e aleatórias. É notável o funcionamento do algoritmo híbrido proposto, que adota o Quick Sort Randomizado como estratégia para ordenar os vetores nesse contexto. Essa abordagem destaca a capacidade do algoritmo híbrido em selecionar o método mais adequado para lidar com tamanhos maiores de entrada, resultando em um desempenho

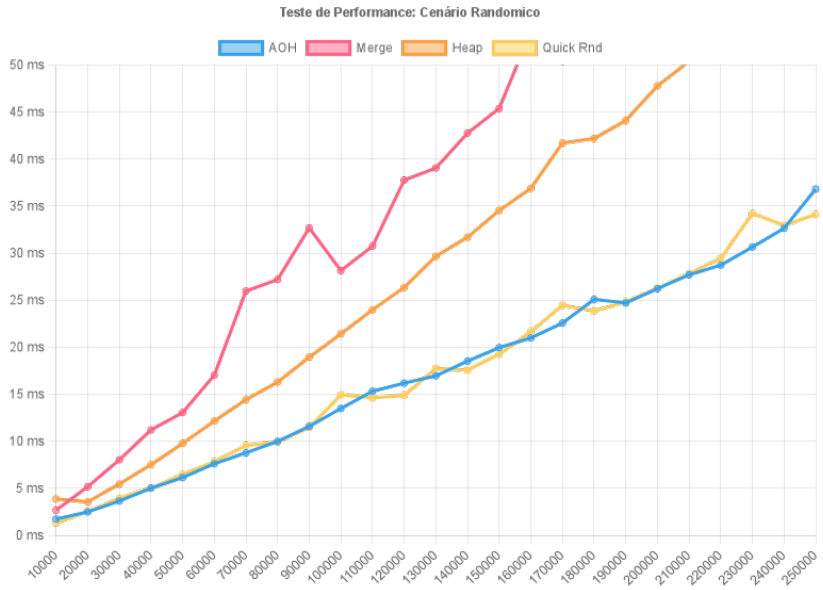

Figure 11. Tempo de execução para entradas grandes e aleatórias.

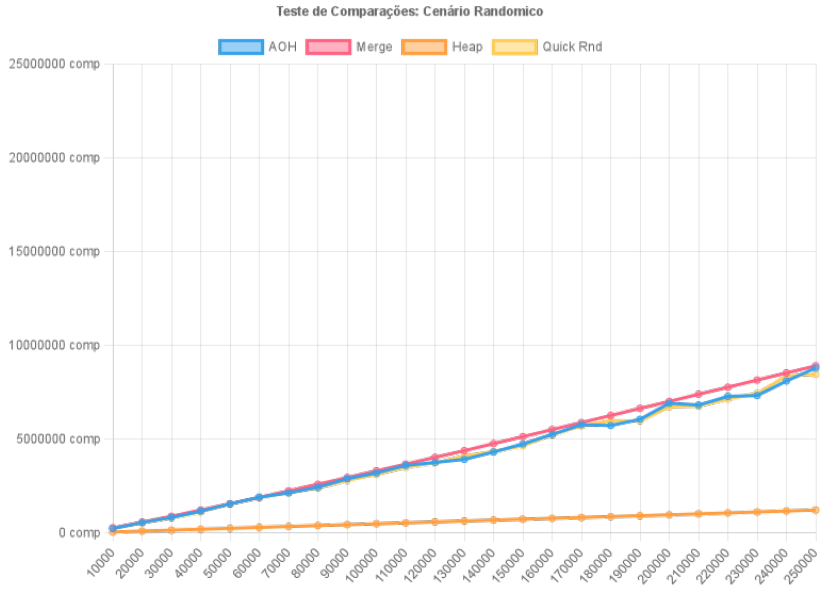

Figure 12. Quantidade de comparações para entradas grandes e aleatórias.

eficiente e satisfatório na ordenação dos dados.

O algoritmo híbrido proposto apresentou resultados consistentes e satisfatórios, graças à sua capacidade de adaptação aos diversos cenários enfrentados. Foi observado que o algoritmo obteve um desempenho excelente em todos os casos de teste, demonstrando sua eficiência e eficácia em lidar com diferentes conjuntos de dados. Além disso, o comportamento do algoritmo híbrido mostrou-se similar aos algoritmos que o compõem, o que fortalece a confiança em seu funcionamento consistente e confiável.

6. Conclusão

Em conclusão, este artigo apresentou uma análise comparativa de desempenho de diferentes algoritmos de ordenação em diversos cenários. Foi demonstrado que a escolha do algoritmo adequado para cada situação pode ter um impacto significativo no tempo de execução e no número de comparações realizadas. Além disso, foi proposto um algoritmo híbrido, o AOH, que busca combinar as melhores características dos algoritmos estudados, visando obter um desempenho satisfatório em todos os cenários.

Os resultados obtidos mostraram que o algoritmo híbrido AOH demonstrou um desempenho consistente e eficiente em diferentes tamanhos de entrada e configurações. Ele se adaptou de forma inteligente aos diferentes cenários, utilizando algoritmos específicos para obter resultados ótimos. Essa abordagem híbrida demonstrou ser uma solução promissora para lidar com problemas de ordenação em contextos reais, onde os conjuntos de dados podem variar em tamanho e características.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Algoritmos: Teoria e Prática*. Campus, Rio de Janeiro, RJ, 3ª ed. edition.

freeCodeCamp. Algoritmos de ordenação explicados com exemplos em python, java e c. Acesso em 13 de julho de 2023. Disponível em: https://www.freecodecamp.org/portuguese/news/algoritmos-de-ordenacao-explicados-com-exemplos-em-python-java-e-c/.

IME USP. Notas de aula: Quicksort. Acesso em 13 de julho de 2023. Disponível em: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/quick.html.

SAPS. Sorting algorithms performance simulator. https://github.com/pumba-dev/sorting-algorithms-comparison. Acesso em 13 de julho de 2023.

Wikipedia (Bubble). Bubble sort. Acesso em 13 de julho de 2023. https://pt.wikipedia.org/wiki/Bubble_sort.

Wikipedia (Heap). Heapsort. Acesso em 13 de julho de 2023. https://pt.wikipedia.org/wiki/Heapsort.

Wikipedia (Insertion). Insertion sort. Acesso em 13 de julho de 2023. https://pt.wikipedia.org/wiki/Insertion_sort.

Wikipedia (Merge). Merge sort. Acesso em 13 de julho de 2023. https://pt.wikipedia.org/wiki/Merge_sort.