Implementação e Análise de Desempenho de Algoritmos de Ordenação

Davi Lopes Lemos[1],
Heduardo Witkoski Barcelos da Rocha[1]

[1]Campus Bagé - Universidade Federal do Pampa (UNIPAMPA)
96.413-172 – Bagé – RS – Brazil

{davilemos, heduardorocha}.aluno@unipampa.edu.br

1. Escopo do experimento

Analisar <os métodos de ordenação de dados>

para o propósito de <avaliação>

com respeito ao <tempo de execução de cada método>

do ponto de vista do(a) <programador>

no contexto de <estudos do componente de Algoritmos e Classificação de Dados>.

2. Algoritmos de Ordenação

Os algoritmos de ordenação desempenham um papel essencial na organização de dados, sendo amplamente utilizados em diversas aplicações, como buscas eficientes, análise de dados e otimização de processos. O objetivo desses métodos é reordenar elementos de uma sequência, garantindo que eles sigam uma ordem específica, como crescente ou decrescente. Com uma variedade de algoritmos disponíveis, a escolha do método mais adequado depende de fatores como o tamanho do conjunto de dados, a estrutura interna dos elementos e as limitações de tempo e espaço. Portanto, analisar a complexidade de tempo e espaço de cada algoritmo ajuda a entender sua eficiência e viabilidade em diferentes contextos.

2.1. Bubblesort

O Bubblesort é um algoritmo simples que compara repetidamente elementos adjacentes e os troca se estiverem na ordem errada. Esse processo é repetido até que a lista esteja ordenada. Embora seja fácil de implementar, o Bubblesort tem uma complexidade de tempo $O(n^2)$ no pior e no caso médio, o que o torna ineficiente para listas grandes. O algoritmo não possui um desempenho linear, já que a quantidade de comparações e trocas aumenta de forma quadrática com o número de elementos. Portanto, em casos gerais, não é considerado eficiente para grandes volumes de dados.

2.2. Insertionsort

O Insertionsort constrói a lista ordenada gradualmente, inserindo cada elemento da entrada em sua posição correta. É eficiente para listas pequenas ou quase ordenadas, com complexidade $O(n^2)$ no pior caso, mas $O(n)$ no melhor caso, quando os dados já estão ordenados. Seu desempenho melhora em listas parcialmente ordenadas devido às poucas operações de troca necessárias. Em seu melhor caso, o algoritmo pode operar de forma linear, $O(n)$, já que cada elemento é inserido em sua posição sem a necessidade de comparações adicionais, tornando-o ideal para dados já quase ordenados.

2.3. Selectionsort

O Selectionsort divide a lista em uma parte ordenada e outra desordenada. Em cada iteração, ele encontra o menor elemento da parte desordenada e o coloca na posição correta. Embora simples, o algoritmo apresenta complexidade O(n^2), independentemente do estado inicial dos dados, o que o torna pouco eficiente para listas extensas. A performance não é linear, já que o número de operações cresce quadraticamente conforme o número de elementos. Assim, o algoritmo não é indicado para listas de grandes dimensões.

2.4. Shellsort

O Shellsort é uma versão otimizada do Insertionsort que utiliza lacunas (gaps) para comparar elementos distantes. As lacunas são progressivamente reduzidas até que o algoritmo se transforme em um Insertionsort clássico. A complexidade varia de O(n^2) a O($nlog^2n$), dependendo da escolha da sequência de lacunas, com melhor desempenho para listas parcialmente ordenadas. Embora não tenha uma complexidade estritamente linear, o Shellsort pode apresentar desempenho mais próximo de O(n) dependendo das lacunas utilizadas, sendo mais eficiente em listas parcialmente ordenadas.

2.5. Heapsort

O Heapsort transforma a lista em uma estrutura de heap, o que permite a remoção eficiente do maior ou menor elemento e sua inserção em uma lista ordenada. Ele garante complexidade O($nlogn$) no pior caso e é eficiente em termos de espaço, mas não é estável, o que significa que a ordem relativa de elementos iguais pode não ser preservada. O Heapsort não é um algoritmo linear, já que a transformação da lista em uma heap e as remoções de elementos requerem operações de logaritmo, tornando a complexidade O($nlogn$).

2.6. Mergesort

O Mergesort é um algoritmo de ordenação baseado na divisão e conquista. Ele divide a lista em partes menores até que cada parte tenha um único elemento e, em seguida, as mescla de forma ordenada. Sua complexidade é O($nlogn$) no pior caso, e ele é estável, mas consome espaço adicional proporcional ao tamanho da lista, o que pode ser um inconveniente. Como o Mergesort se baseia na divisão da lista em várias partes, seu desempenho não é linear, e a complexidade O($nlogn$) reflete a combinação de operações de divisão e fusão.

2.7. Quicksort

O Quicksort também utiliza a técnica de divisão e conquista, escolhendo um pivô e partindo a lista em elementos menores e maiores que o pivô. Ele é eficiente na prática, com complexidade O($nlogn$) no caso médio, mas pode ter desempenho O(n^2) no pior caso se os pivôs forem mal escolhidos. Técnicas como a escolha aleatória de pivô podem mitigar esse problema. Embora não tenha um desempenho linear, o Quicksort é um dos algoritmos mais eficientes para listas grandes devido à sua complexidade média O($nlogn$).

2.8. Countingsort

O Countingsort é um algoritmo não comparativo que usa a contagem de ocorrências para ordenar elementos. É eficiente quando o intervalo de valores é relativamente pequeno,

apresentando complexidade O(n + k), onde k é o maior valor na lista. Ele requer espaço adicional proporcional ao intervalo dos valores e não é adequado para listas com dados dispersos. Quando o intervalo de valores não é muito grande, o Countingsort pode se aproximar de um desempenho linear O(n), tornando-o extremamente eficiente nesse tipo de caso.

2.9. Radixsort

O Radixsort ordena números digitando os dígitos, do menos ao mais significativo, usando um algoritmo estável de ordenação por contagem em cada dígito. Ele é eficiente para ordenar grandes conjuntos de dados numéricos e possui complexidade O(nk), onde k é o número de dígitos. Funciona bem para dados com representações numéricas limitadas. Embora o Radixsort dependa de k, que pode ser considerado uma constante em certos cenários, ele pode ser visto como linear, O(n), quando k é pequeno ou fixo, sendo eficiente para grandes volumes de dados numéricos.

2.10. Bucketsort

O Bucketsort distribui os elementos em várias listas chamadas baldes (buckets) e, em seguida, ordena cada balde individualmente (geralmente com Insertionsort). Ele é eficiente para entradas uniformemente distribuídas, com complexidade média O(n + k), onde k é o número de baldes. No entanto, o desempenho pode se degradar se os dados não forem distribuídos de forma uniforme. Quando os dados são uniformemente distribuídos, o Bucketsort pode ter um desempenho linear O(n), tornando-o uma escolha viável em cenários específicos de dados bem distribuídos.

3. Metodologia

Esta pesquisa experimental se propõe a analisar o desempenho (tempo de execução) e eficácia dos diferentes algoritmos de ordenação apresentados na Seção 2. Para tanto, todos eles foram implementados na linguagem de alto nível de programação `Java`, utilizando recursos como encapsulamento de lógica em classes e métodos, estruturas de controle for e while para iterar sobre elementos, e a manipulação de vetores para armazenamento e ordenação dos dados. A medição do tempo de execução foi realizada com a utilização da classe `System` e seu método `nanoTime()`, permitindo capturar o tempo inicial e final de cada algoritmo e calcular a duração com precisão nanosegundos, garantindo uma análise detalhada da eficiência temporal de cada método de ordenação.

A fim de mitigar eventuais medições de tempo de execução aberrantes, isto é, que destoam da normalidade do método, executou-se cada algoritmo dez vezes para cada um dos diferentes tamanhos de entrada, que por sua vez variam entre 10^2, 10^3, 10^4 e 10^5. É necessário salientar que dada a complexidade de alguns métodos, não foi possível executar todos eles com o tamanho de entrada 10^6, excluíndo-o da análise por este motivo. O ambiente de execução foi padronizado a todos os experimentos e conta com o microprocessador *multicore* AMD Ryzen 7 5700U com frequência de até 4,3 GHz e 8 núcleos com 16 threads, placa de vídeo integrada AMD Radeon Graphics 8, 12 GB de memória DDR4 e sistema operacional `Ubuntu 24.04 LTS`.

São utilizados diferentes métodos para experimentação dos algoritmos através da entrada de diferentes configurações de vetores. De forma mais recorrente, neste caso

aplicado para todos os tamanhos de entrada analisados, foram utilizados vetores com elementos aleatórios variando de um a 9999, padronizados para todos os algoritmos. Além deste, também analisou-se outros três vetores de teste de correção, com enfoque apenas no tamanho de entrada intermediário, – 10^4: o primeiro possui elementos de um a 9999 em ordem crescente, o segundo os mesmos valores em ordem decrescente e, por fim, um terceiro com vários valores repetidos entre um e 2000 (repetindo assim cinco vezes cada elemento).

Adicionalmente, considerando a implementação dos algoritmos, foi realizada uma alteração nas variáveis do tipo Comparable, utilizada para permitir a comparação entre objetos na linguagem Java, para o tipo int, uma variável primitiva, buscando avaliar o impacto dessa mudança na execução, uma vez que, ao utilizar o tipo primitivo, a comparação torna-se mais simples e rápida, sem a necessidade de chamadas a métodos como o compareTo(), presente em objetos que implementam Comparable.

Neste trabalho, não se faz uma comparação direta entre os métodos lineares (radix sort, counting sort e bucket sort) e os demais métodos. A razão para isso é que os algoritmos lineares possuem características e aplicações distintas, não sendo apropriado compará-los diretamente com os métodos baseados em comparação de elementos, já que não são dependentes disto mas utilizam informações adicionais, como contagens ou distribuições dos dados, o que os torna mais eficientes sob determinadas condições. Dessa forma, sua análise deve ser feita em contextos em que essas características sejam relevantes, o que difere dos métodos tradicionais, que se baseiam em comparações diretas para realizar a ordenação.

Ademais, para sintetização dos dados e visualização gráfica dos resultados, utilizou-se de um programa em linguagem Python, capaz de processar e visualizar os tempos de execução de diferentes algoritmos de ordenação. Através de bibliotecas como `pandas` e `matplotlib`, o código realiza a leitura e processamento dos dados, agrupando-os por tamanho de entrada e algoritmo. Para a visualização, foram empregados gráficos como boxplots, que ajudam a ilustrar a distribuição dos tempos de execução, e gráficos de barras, que comparam as médias dos tempos de execução de forma clara. Além disso, gráficos de linha com escala logarítmica foram utilizados para mostrar as linhas de tendência dos tempos de execução conforme o aumento do tamanho de entrada, permitindo uma análise visual da complexidade dos algoritmos.

4. Resultados

4.1. Gráficos de caixa e *outliers*

Nos gráficos de caixa (*boxplots*) ilustrados na Figura 1, é possível notar a presença de *outliers* em quase todos os algoritmos de ordenação, especialmente para tamanhos de entrada maiores. Esses outliers indicam casos isolados em que o tempo de execução do algoritmo foi significativamente mais lento ou mais rápido do que o esperado, possivelmente devido a variações na organização inicial dos dados ou a particularidades no gerenciamento de recursos computacionais. Além disso, análises preliminares indicaram que a primeira execução foi a mais demorada em todos os experimentos, possivelmente por conta do processo inicial de carregamento e preparação dos dados ou da alocação de recursos, o que impactou o desempenho dos algoritmos.

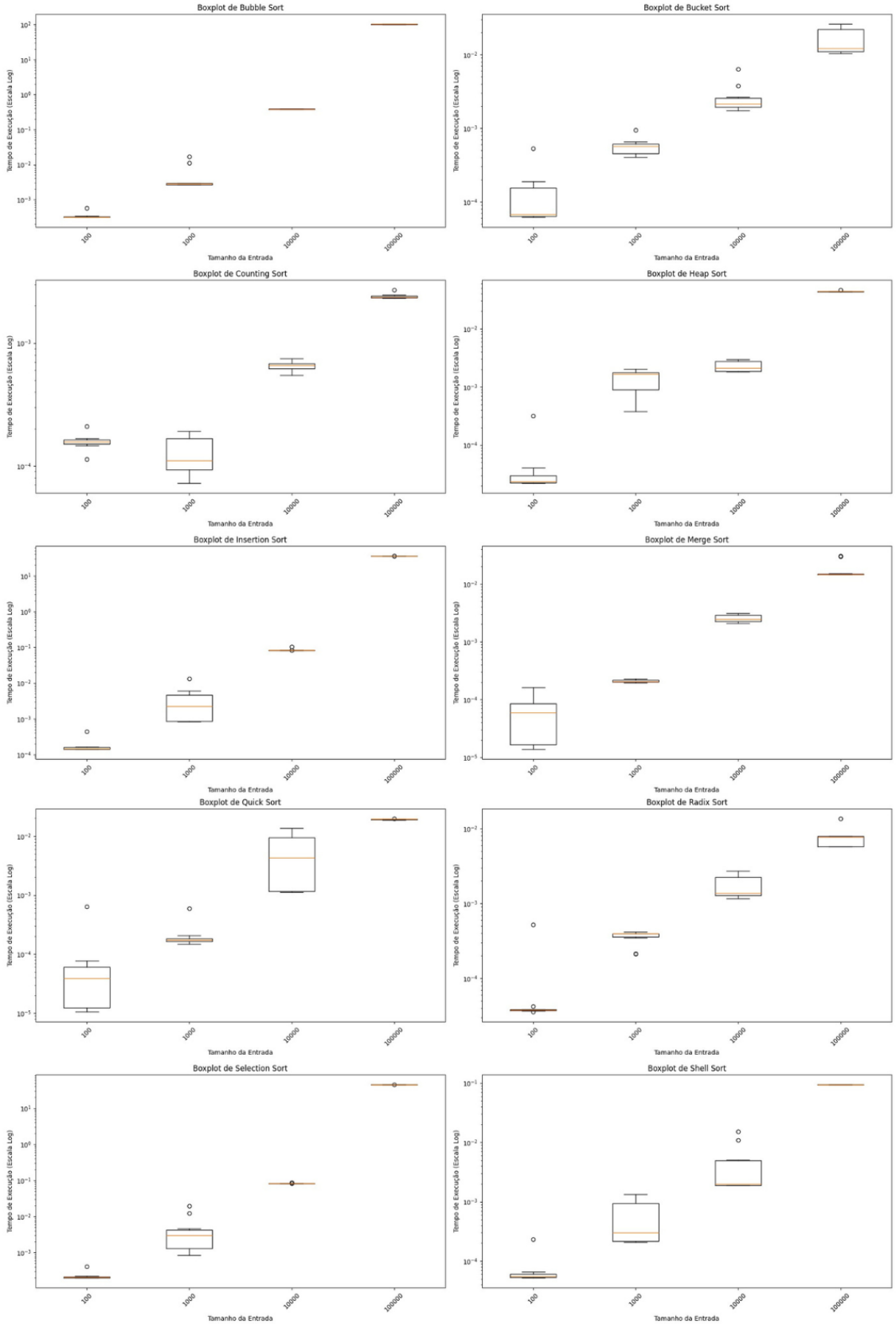

Figura 1. Gráficos de caixa dos tempos de execução de cada algoritmo

4.2. Análise dos algoritmos não-lineares

Nos gráficos de barras, observa-se que, conforme esperado, os algoritmos com complexidade O(n^2) — Bubble Sort, Selection Sort e Insertion Sort — apresentaram os maiores tempos de execução em todos os casos. Dentre esses três, o Bubble Sort foi, com maior frequência, o que demandou mais tempo. Um fator que pode contribuir para esse comportamento é o maior número de trocas de elementos realizadas pelo Bubble Sort em comparação com o Insertion Sort e o Selection Sort, que realizam menos trocas. Além disso, o Bubble Sort percorre o vetor mais vezes, o que pode ter influenciado os resultados observados. Em contraste, os algoritmos com complexidade O($nlogn$) — Heap Sort, Merge Sort, Shell Sort e Quick Sort — apresentaram, como esperado, tempos de execução significativamente menores que os algoritmos de complexidade O(n^2). Essa diferença entre os algoritmos de complexidade O($nlogn$) e os de complexidade O(n^2) aumentou à medida que o tamanho da entrada cresceu, o que é consistente com a teoria, pois o crescimento de uma função quadrática é mais rápido do que o de uma função logarítmica. Entre os algoritmos de complexidade O($nlogn$), não houve consenso claro, mas observou-se que, para o maior tamanho de entrada testado, os algoritmos que apresentaram o melhor desempenho foram o Quick Sort e o Merge Sort, embora com diferenças mínimas em relação aos outros algoritmos dessa classe.

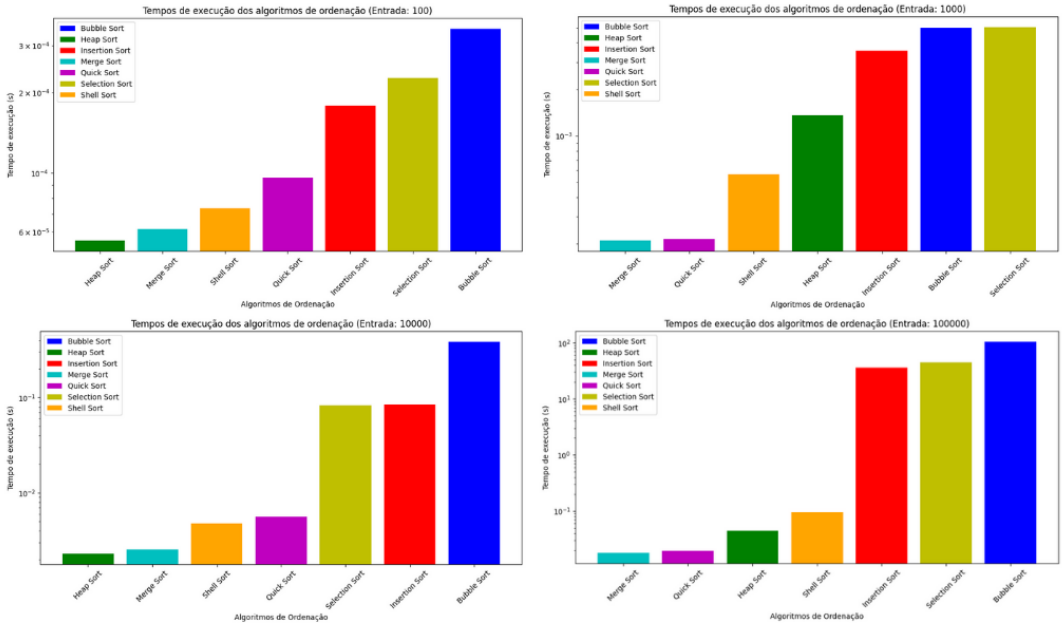

Figura 2. Comparação geral entre os algoritmos não-lineares

4.2.1. Gráfico de dispersão

No gráfico de dispersão, ao plotar os valores de tempo de execução, os algoritmos com complexidade O(n^2) exibem o comportamento esperado, enquanto os algoritmos com complexidade O($nlogn$) mostram um aumento mais gradual à medida que o tamanho da entrada cresce, o que também é esperado.

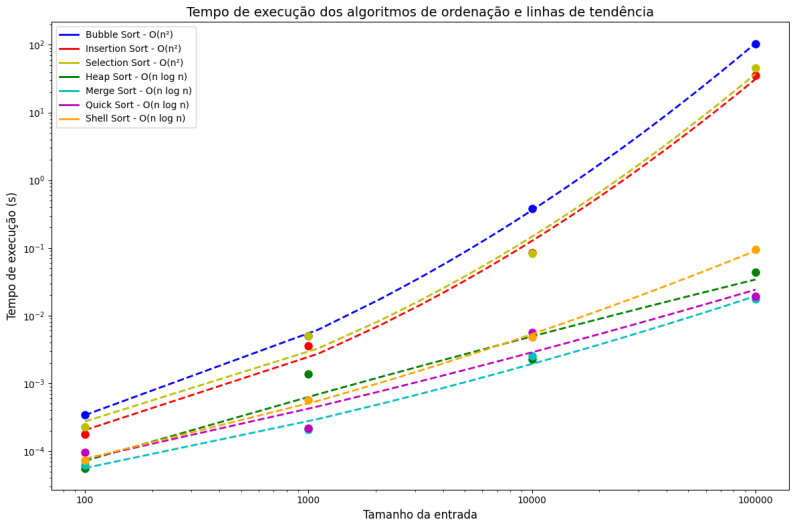

Figura 3. Gráfico de dispersão dos algoritmos não-lineares

4.2.2. Uso de diferentes configurações para vetor

Ao comparar o desempenho dos algoritmos com vetores aleatórios, crescentes, decrescentes e com valores repetidos, todos com 10.000 elementos, observou-se que o padrão geral se manteve. Os algoritmos de complexidade O(n^2) continuaram a ser os mais lentos, enquanto os algoritmos de complexidade O($nlogn$) se mantiveram mais rápidos em todos os casos. No entanto, notou-se que a configuração inicial do vetor teve menor impacto no desempenho dos algoritmos de complexidade O(n^2) do que nos algoritmos de complexidade O($nlogn$), nos quais foram observadas diferenças significativas dependendo da configuração. Em particular, nos algoritmos de complexidade O($nlogn$), o tempo de execução foi menor nos vetores com valores repetidos. Esse comportamento pode ser explicado pelo fato de que algoritmos como Merge Sort, Quick Sort, Heap Sort e Shell Sort realizam menos trocas e comparações quando os elementos são repetidos, o que reduz o trabalho necessário para ordenar o vetor e, consequentemente, melhora o desempenho.

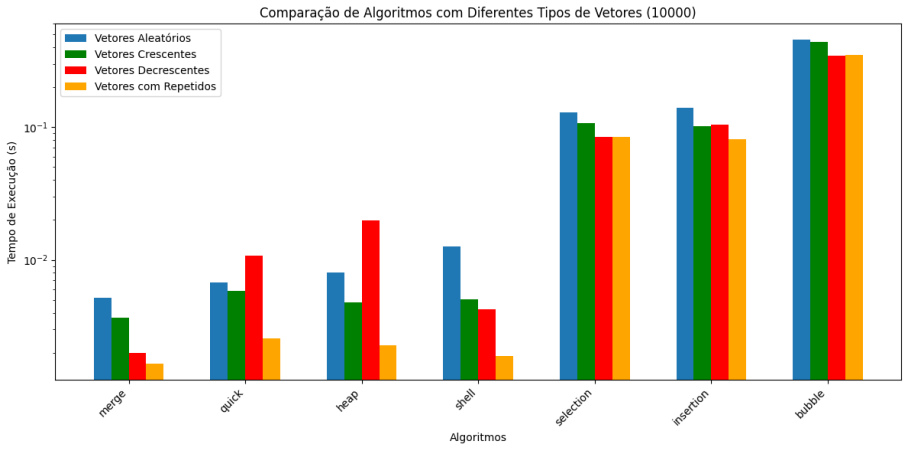

Figura 4. Comparação entre os algoritmos não-lineares com vetores configurados

4.2.3. Impactos pelo tipo de dado

Na comparação entre o uso de dados do tipo int e Comparable, com vetores de tamanho 10.000, não houve diferença significativa no desempenho dos algoritmos nas duas configurações. O padrão observado foi o mesmo em ambos os casos. Embora diferenças possam surgir com vetores de maior tamanho, com entradas de 10.000 elementos, não foi observada nenhuma variação relevante entre os tipos de dados utilizados.

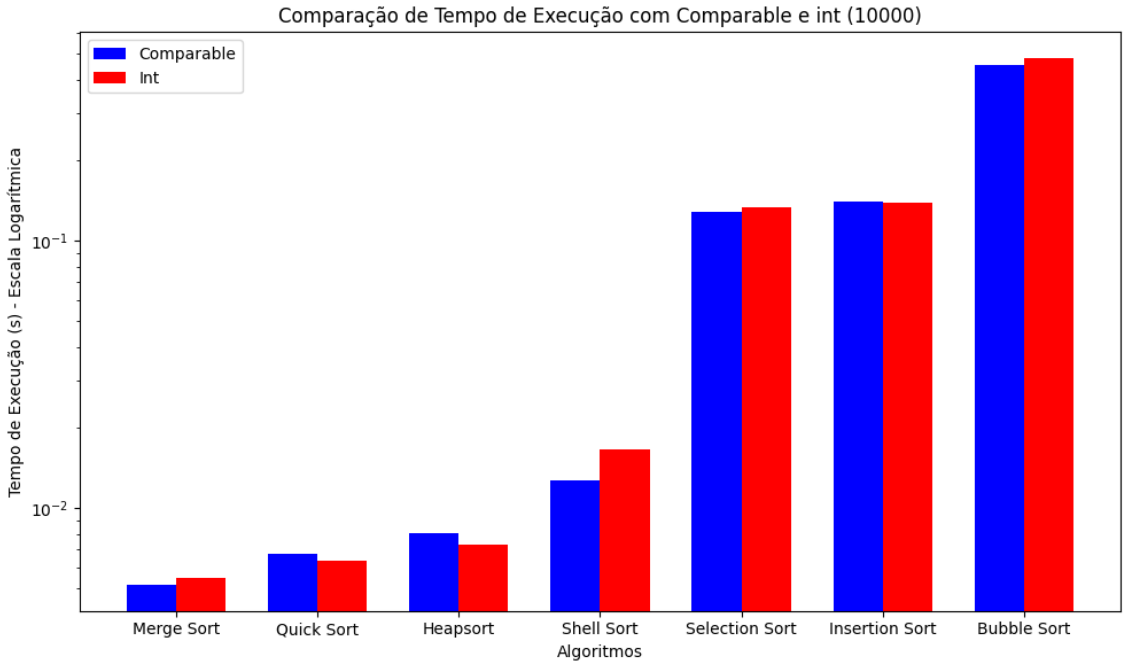

Figura 5. Impactos pelo tipo de dado

4.3. Análise dos algoritmos lineares

4.3.1. Uso de diferentes configurações para vetor

Na comparação entre os algoritmos lineares — Counting Sort, Radix Sort e Bucket Sort — com vetores aleatórios, repetidos, crescentes e decrescentes, foi possível observar que, em geral, os três algoritmos apresentaram melhor desempenho com vetores contendo valores aleatórios (bem distribuídos). Por outro lado, os piores desempenhos foram observados com vetores contendo valores repetidos. Esse comportamento pode ser explicado pelo fato de que, no caso do Bucket Sort, a presença de valores repetidos torna o algoritmo menos eficiente, pois os elementos são agrupados em poucos "buckets". Já no caso do Radix Sort, a variação nos dígitos é reduzida com valores repetidos, o que pode levar o algoritmo a realizar mais passos, afetando negativamente sua performance.

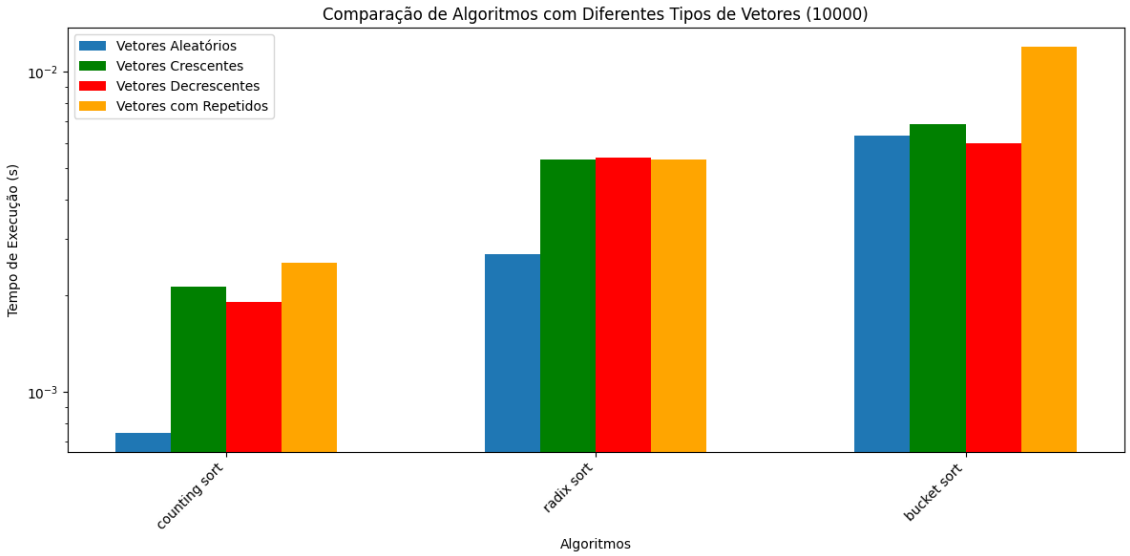

Figura 6. Análise dos algoritmos lineares

4.3.2. Impactos pelo tipo de dado

Na comparação entre o uso de dados do tipo int e Comparable, com vetores de 10.000 elementos, não foi observada diferença significativa no desempenho dos algoritmos lineares, assim como ocorreu nas comparações anteriores.

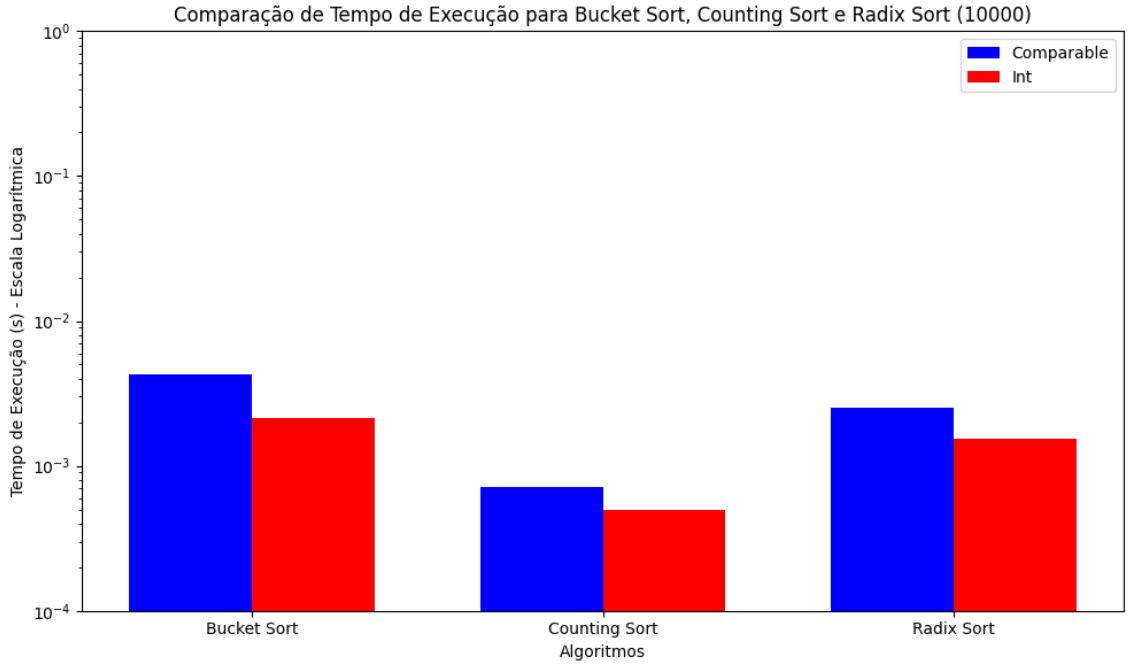

Figura 7. Impactos pelo tipo de dado

5. Conclusão

A análise dos sete algoritmos não-lineares (Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Merge Sort, Quick Sort e Shell Sort) revelou comportamentos alinhados com as expectativas teóricas. Os algoritmos de complexidade O(n^2) — Bubble Sort, Selection Sort e Insertion Sort — apresentaram tempos de execução significativamente maiores em comparação com os de complexidade O($nlogn$) (Heap Sort, Merge Sort, Quick Sort e Shell Sort). O Bubble Sort, em particular, foi consistentemente o mais lento entre os três algoritmos quadráticos, possivelmente devido ao maior número de trocas e iterações realizadas durante o processo de ordenação.

Por outro lado, os algoritmos de complexidade O($nlogn$) demonstraram um desempenho superior, com o Quick Sort e o Merge Sort destacando-se no caso de entradas maiores, embora as diferenças entre eles tenham sido mínimas. Isso confirma que o crescimento mais lento da função logarítmica resulta em um desempenho mais eficiente à medida que o tamanho da entrada aumenta, corroborando a teoria sobre a complexidade assintótica.

Quando analisado o impacto da configuração do vetor, observou-se que os algoritmos de complexidade O($nlogn$) foram mais sensíveis a diferentes tipos de entrada. Vetores com valores repetidos, por exemplo, resultaram em um desempenho superior devido à redução no número de trocas e comparações necessárias. Isso foi particularmente notável no Merge Sort, Quick Sort, Heap Sort e Shell Sort. Em contraste, os algoritmos quadráticos não apresentaram grandes variações em seu desempenho com diferentes configurações de entrada.

Na comparação entre o uso de dados do tipo int e Comparable, não houve diferenças significativas no desempenho dos algoritmos, tanto nos casos lineares quanto nos não-lineares, quando o tamanho da entrada era de 10.000 elementos.

Finalmente, ao considerar os algoritmos lineares (Counting Sort, Radix Sort e Bucket Sort), verificou-se que seu desempenho era melhor com vetores de valores aleatórios, enquanto o desempenho deteriorava com vetores contendo valores repetidos, especialmente no caso do Bucket Sort e Radix Sort. Este comportamento pode ser atribuído à forma como esses algoritmos manipulam a distribuição dos elementos.

Em suma, os resultados obtidos reforçam as expectativas teóricas sobre a eficiência dos algoritmos de ordenação, com algoritmos O($nlogn$) sendo mais adequados para entradas grandes e estruturas de dados mais complexas, enquanto os algoritmos lineares são eficazes quando o tipo de entrada é mais controlado ou especializado.