UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE ENSINO SUPERIOR DO SERIDÓ
DEPARTAMENTO DE COMPUTAÇÃO E TECNOLOGIA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Trabalho Avaliativo para disciplina de Estrutura de Dados.

Felipe Augusto Araújo da Cunha

Caicó - RN
Julho de 2024

Felipe Augusto Araújo da Cunha

Trabalho Avaliativo para disciplina de Estrutura de Dados.

Trabalho apresentado a disciplina Estrutura de Dados do Departamento de Computação e Tecnologia da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção da nota da avaliação I.

Orientador(a): Prof. Dr. João Paulo de Souza Medeiros.

Caicó - RN
Julho de 2024

RESUMO

O trabalho apresentado se baseia na implementação dos algoritmos solicitados pelo professor João Paulo. Este documento inclui os códigos dos algoritmos, juntamente com comentários e explicações detalhadas sobre cada um deles. Além disso, são apresentados os gráficos gerados para cada algoritmo, seguidos por uma comparação e análise dos tempos de execução.

Palavras-chave: Questões, Limites, Cálculo, Resoluções.

ABSTRACT

The work presented is based on the implementation of the algorithms requested by Professor João Paulo. This document includes the codes of the algorithms, along with detailed comments and explanations about each of them. In addition, the graphs generated for each algorithm are presented, followed by a comparison and analysis of the execution times.

Keywords: Questions, Limits, Calculation, Resolutions.

SUMÁRIO

1 INTRODUÇÃO

Este trabalho tem como objetivo analisar e comparar diferentes algoritmos de ordenação, especificamente: Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Distribution Sort. Os códigos dos algoritmos foram desenvolvidos utilizando a linguagem de programação C, escolhida por sua eficiência e controle de baixo nível, o que facilita a análise de desempenho.

Para a visualização dos resultados, foram gerados gráficos utilizando a ferramenta Gnuplot. Esses gráficos mostram as estimativas práticas dos tempos de execução de cada algoritmo.

Além da geração dos gráficos, este trabalho também inclui uma análise do tempo de execução de cada algoritmo, bem como a comparação entre alguns deles. Por fim, um relatório detalhado foi preparado, consolidando todas as observações e conclusões obtidas durante o estudo.

2 RESOLUÇÃO DOS CÓDIGOS

2.1 SELECTION SORT

O Selection Sort segue uma rotina bem simples e direta: encontrar o menor elemento e colocá-lo na primeira posição. O gráfico abaixo mostra o tempo de execução em relação ao tamanho do vetor.

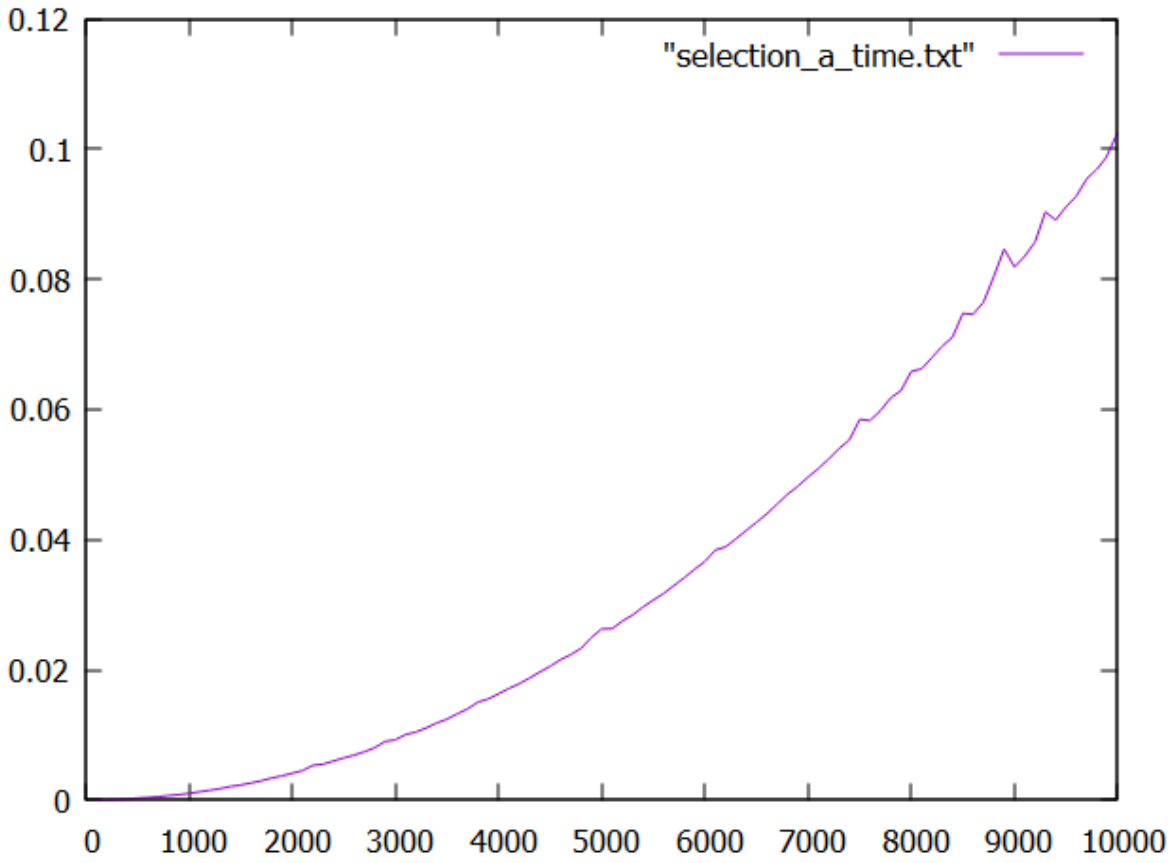

Figura 1 – Gráfico de tempo de execução para Selection Sort

ANÁLISE

O Selection Sort é um algoritmo simples de ordenação por comparação que funciona selecionando repetidamente o menor elemento da lista não ordenada e movendo-o para sua posição final. Este algoritmo possui uma complexidade de tempo de $O(n^2)$, onde n é o número de elementos na lista.

O Selection Sort opera da seguinte maneira:

- Seleção do Menor Elemento: O Selection Sort funciona encontrando o menor elemento do array e trocando-o com o primeiro elemento, depois encontrando o segundo menor e trocando-o com o segundo elemento, e assim por diante. Esse processo é repetido para cada posição no array até que o array esteja completamente ordenado.

- **Complexidade Quadrática:** O tempo de execução do Selection Sort é $O(n^2)$ em todos os casos (melhor, pior e médio), onde n é o número de elementos no array. Isso ocorre porque, para cada elemento, o algoritmo percorre o restante do array para encontrar o menor elemento, resultando em um número fixo de comparações, independentemente do estado inicial do array.

- **Impacto na Eficiência:** O Selection Sort é in-place, pois a ordenação é realizada no próprio array, sem a necessidade de estruturas auxiliares adicionais. No entanto, o Selection Sort não é estável, pois pode trocar elementos iguais, alterando a ordem relativa dos valores iguais no array.

Embora simples de implementar, o desempenho quadrático do Selection Sort torna-o ineficiente para grandes conjuntos de dados. Sua principal vantagem sobre outros algoritmos quadráticos, como o Insertion Sort, é que ele faz o número mínimo de trocas, mas isso não compensa a alta complexidade de comparação.

2.2 DISTRIBUTION SORT

O Distribution Sort, também conhecido como Bucket Sort, é um algoritmo de ordenação que divide os elementos do array em grupos chamados "buckets". Cada bucket é então ordenado separadamente, e os buckets ordenados são combinados para obter o array ordenado final.

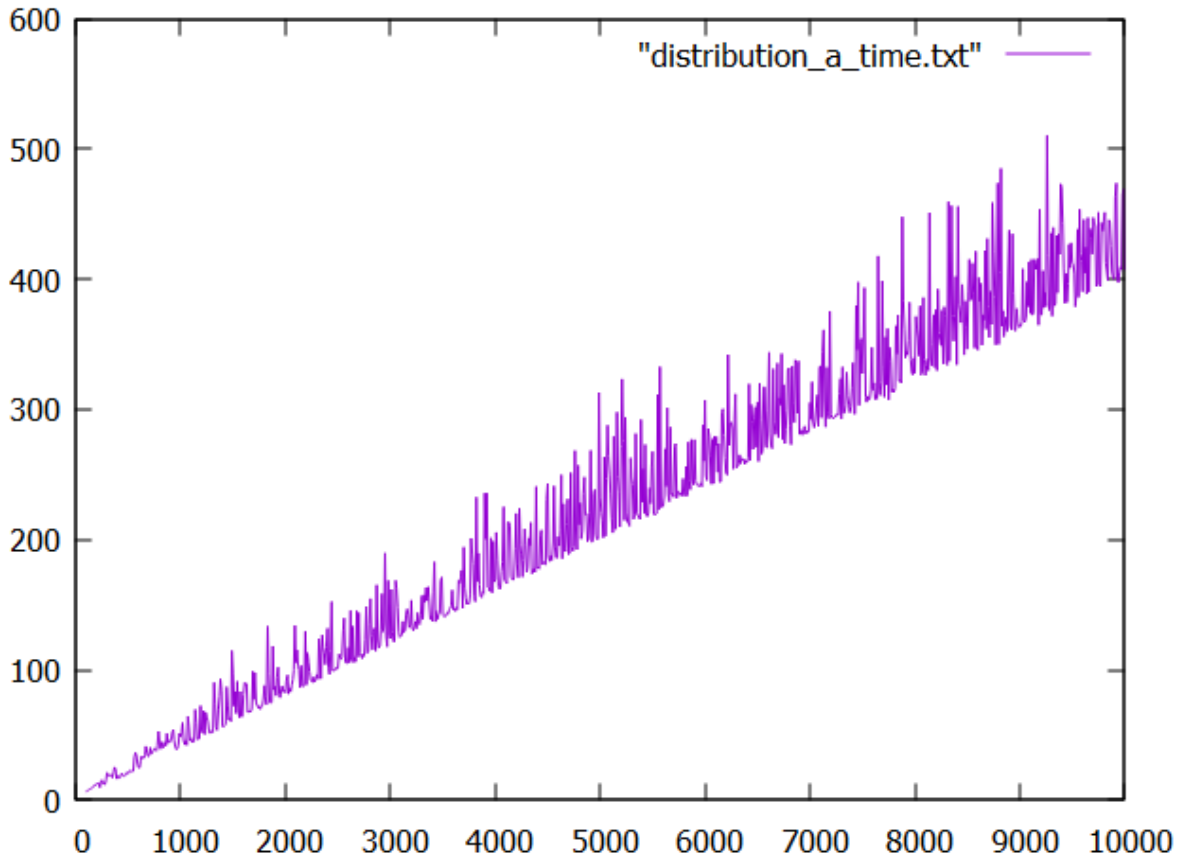

Figura 2 – Gráfico de tempo de execução para Distribution Sort

ANÁLISE

O Distribution Sort opera da seguinte maneira:

- **Identificação do Valor Máximo:** Primeiramente, o algoritmo itera pelo array para identificar o valor máximo presente.
- **Divisão em Buckets:** Com base no valor máximo encontrado, o array é dividido em vários buckets (ou recipientes). Cada bucket abrange um intervalo específico de valores.
- **Ordenação dos Buckets:** Cada bucket é ordenado individualmente. Isso pode ser feito usando um algoritmo de ordenação simples, como o Insertion Sort, devido à

sua eficiência em pequenos conjuntos de dados.

- **Concatenação dos Buckets:** Por fim, os buckets ordenados são concatenados de volta na ordem correta para formar o array ordenado final.

- **Complexidade de Tempo:** O Distribution Sort possui uma complexidade de tempo média de $O(n + k)$, onde n é o número de elementos e k é o número de "buckets". Isso o torna ideal para casos onde os dados estão distribuídos uniformemente.

- **Eficiência de Espaço:** Requer espaço adicional dependendo do número de "buckets"utilizados, mas pode ser eficientemente implementado in-place com uso mínimo de memória adicional além de estruturas de controle.

2.3 MERGE SORT

O Merge Sort é um algoritmo de ordenação eficiente que utiliza o conceito de dividir para conquistar. Ele divide o array em duas metades, ordena cada metade recursivamente e depois combina as duas metades ordenadas para obter o array completamente ordenado.

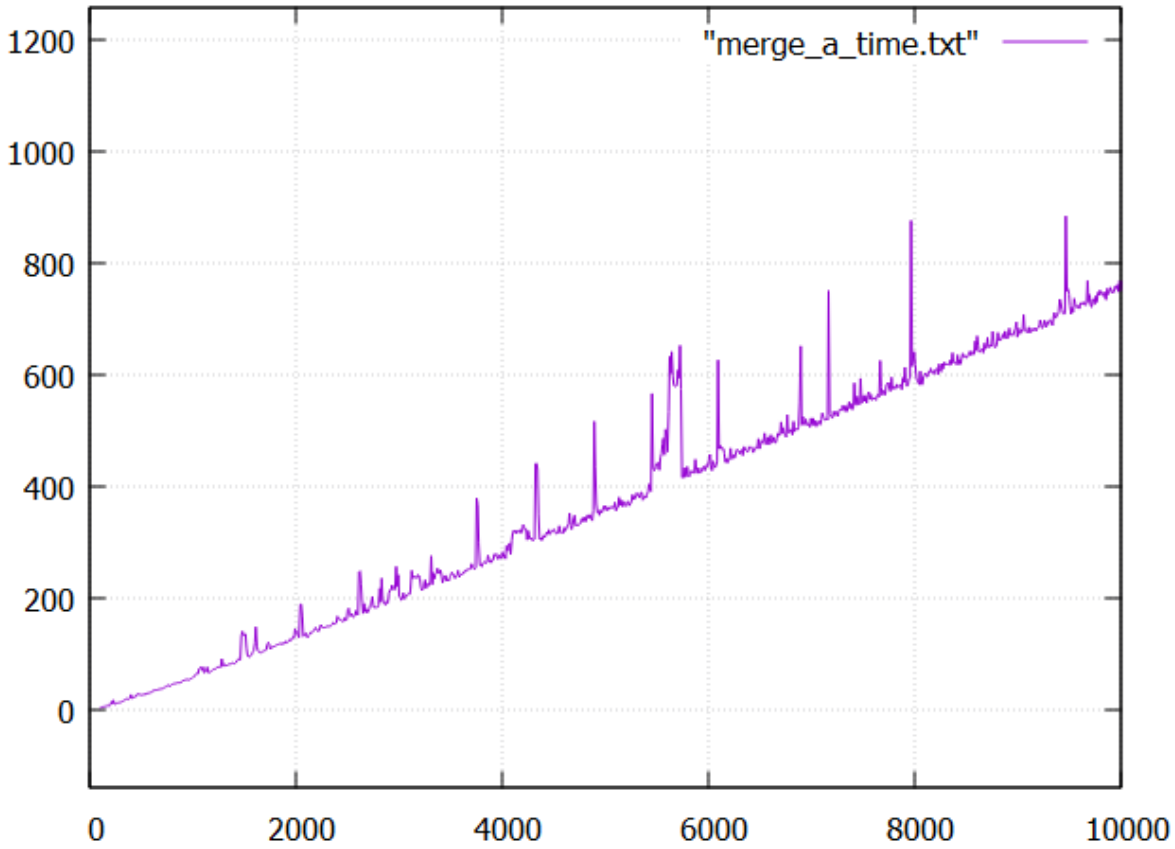

Figura 3 – Gráfico de tempo de execução para Merge Sort

ANÁLISE

O algoritmo baseia a ordenação em sucessivas execuções de merge, uma rotina que une duas partes ordenadas de um array em uma outra também ordenada.

O Merge Sort opera da seguinte maneira:

- **Divisão do Array:** Primeiramente, o array é dividido ao meio repetidamente até que cada subarray tenha um único elemento.
- **Ordenação Recursiva:** Cada par de subarrays é ordenado recursivamente utilizando o Merge Sort.

- **Combinação (Merge):** Os subarrays ordenados são mesclados (merged) de volta em ordem crescente. Isso é feito de forma eficiente, garantindo que o array resultante esteja ordenado.

- **Complexidade de Tempo:** Independente do caso (melhor, pior ou médio) o Merge Sort sempre será $O(n \log n)$. Isso ocorre porque a divisão do problema sempre gera dois sub-problemas com a metade do tamanho do problema original.

- **Eficiência de Espaço:** O Merge-Sort não é in-place, pois requer espaço adicional para armazenar os subarrays durante o processo de ordenação, mas pode ser implementado de forma estável (stable sort) e com uso mínimo de memória adicional além das estruturas de controle.

O Merge Sort é amplamente utilizado devido à sua eficiência e estabilidade na ordenação de grandes conjuntos de dados. Ele é particularmente eficaz quando é necessário garantir a ordem dos elementos e quando o desempenho em termos de tempo é crucial.

2.4 QUICK SORT - MELHOR CASO

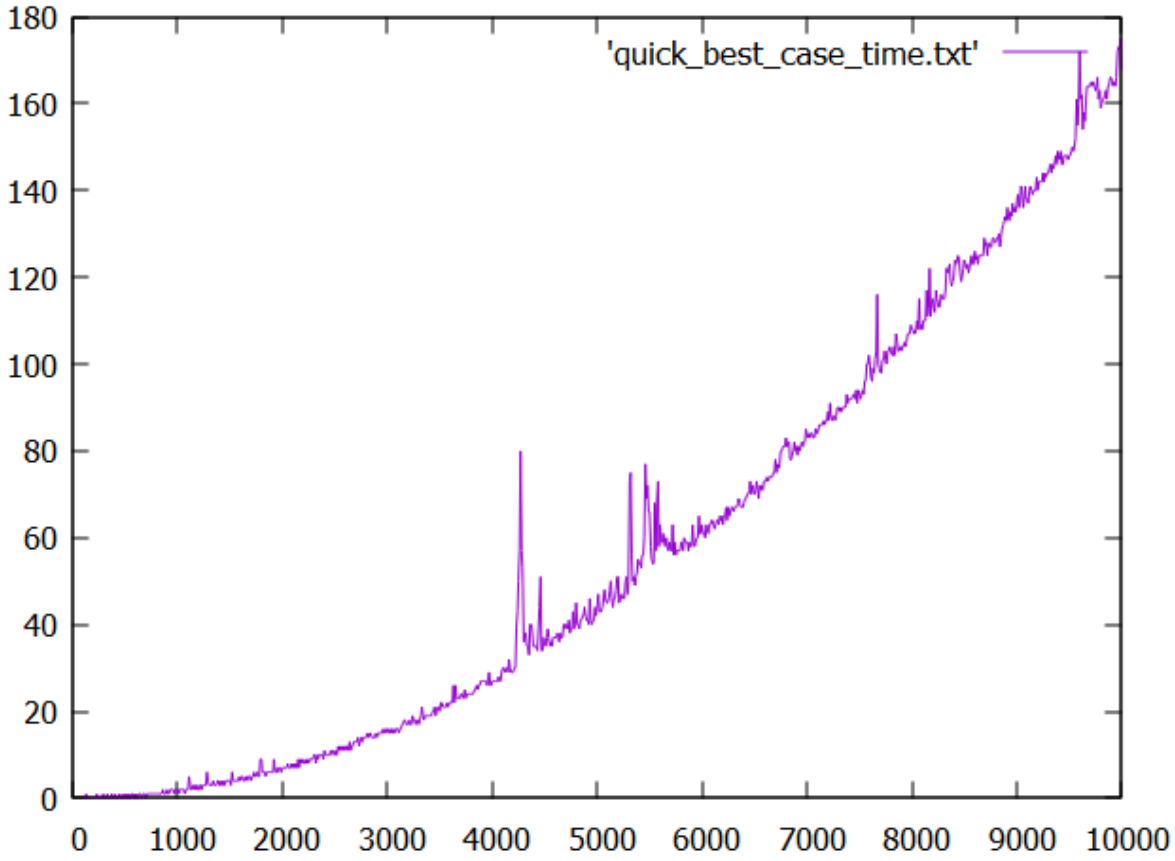

Figura 4 – Gráfico de tempo de execução para Quick Sort no melhor caso

ANÁLISE

O Quick Sort é um algoritmo de ordenação que utiliza a estratégia de dividir para conquistar. No melhor caso, o desempenho do Quick Sort se aproxima de $O(n \log n)$, onde n é o número de elementos na lista a ser ordenada. Isso ocorre quando o pivô escolhido divide o array de forma balanceada, dividindo-o aproximadamente pela metade a cada recursão.

No melhor caso do Quick Sort:

- **Escolha Balanceada do Pivô:** O desempenho ideal do Quick Sort depende crucialmente da escolha do pivô. No melhor cenário, o pivô é escolhido de forma a dividir o array em duas partes aproximadamente iguais a cada passo recursivo. Isso minimiza o número de comparações e trocas necessárias para ordenar o array.
- **Complexidade $O(n \log n)$:** Devido à escolha balanceada do pivô, o Quick Sort alcança uma complexidade de tempo médio de $O(n \log n)$. Isso é muito eficiente

para grandes conjuntos de dados e é uma das razões pelas quais o Quick Sort é amplamente utilizado em aplicações práticas.

- **Eficiência de Espaço:** O Quick Sort é in-place, o que significa que não requer espaço adicional significativo além da pilha de recursão. Isso o torna eficiente em termos de uso de memória comparado a algoritmos como o Merge Sort, que requer espaço adicional para fusão.

2.5 QUICK SORT - PIOR CASO

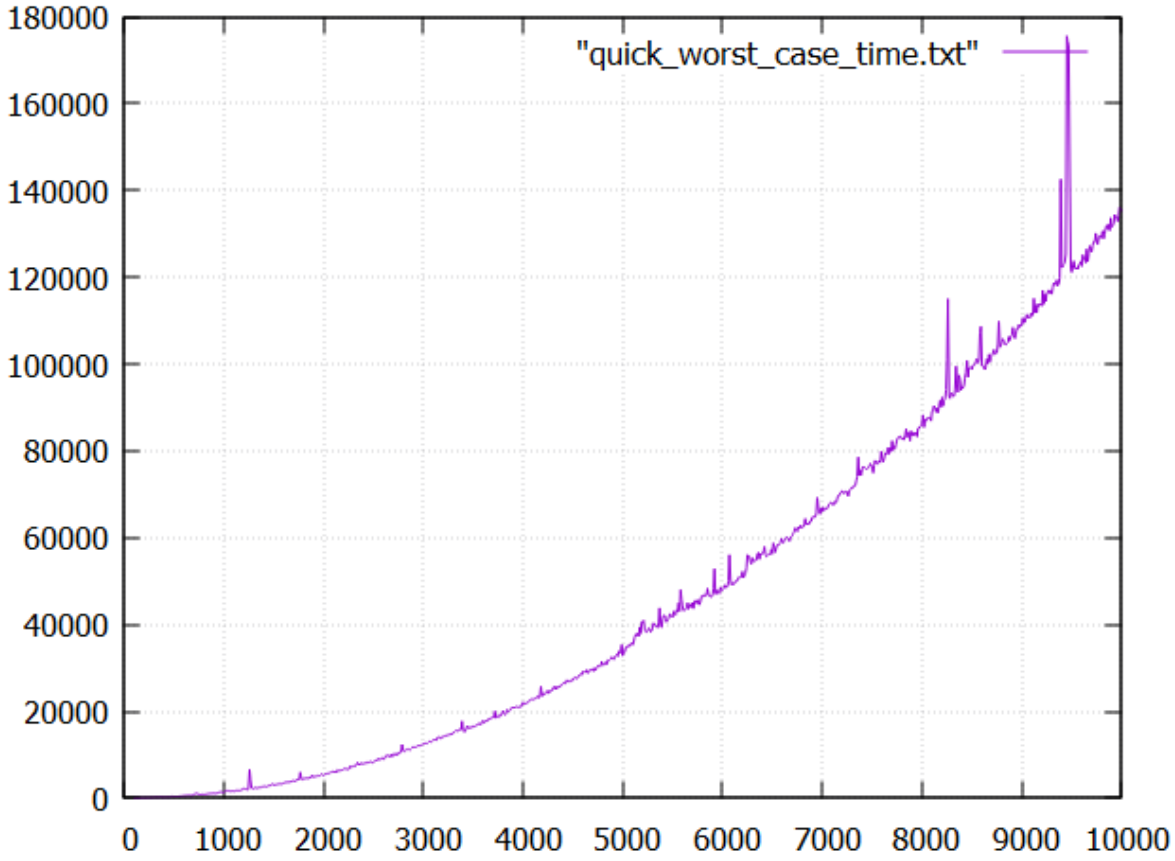

Figura 5 – Gráfico de tempo de execução para Quick Sort no pior caso

ANÁLISE

No pior caso, o desempenho do Quick Sort pode degradar para uma complexidade de $O(n^2)$, onde n é o número de elementos na lista. Isso ocorre quando o pivô escolhido não divide o array de forma balanceada, resultando em partições muito desiguais. Um exemplo típico é quando o pivô é escolhido como o maior ou o menor elemento do array em cada iteração.

No pior caso do Quick Sort:

- **Escolha Desbalanceada do Pivô:** Se o pivô é sempre escolhido como o maior ou o menor elemento do array, o Quick Sort pode degenerar para um desempenho quadrático ($O(n^2)$). Isso ocorre porque a partição não divide o array de forma equilibrada, levando a muitas iterações desnecessárias.
- **Complexidade $O(n^2)$:** Quando o pivô não divide o array de forma balanceada, cada partição pode deixar apenas um elemento à esquerda ou à direita, resultando

em n partições para um array de tamanho n. Isso resulta em uma complexidade de tempo quadrática no pior caso.

- **Impacto na Eficiência:** O desempenho do Quick Sort no pior caso pode ser severamente afetado em cenários onde os dados estão quase ou completamente ordenados, já que a escolha inadequada do pivô pode não aproveitar a estratégia de dividir para conquistar de forma eficiente.

Para mitigar o pior caso do Quick Sort, estratégias como a escolha do pivô mediano ou aleatório podem ser adotadas para tentar equilibrar melhor as partições. Essas estratégias ajudam a reduzir a probabilidade de ocorrência de partições desbalanceadas, melhorando assim o desempenho geral do algoritmo.

2.6 QUICK SORT - CASO MÉDIO

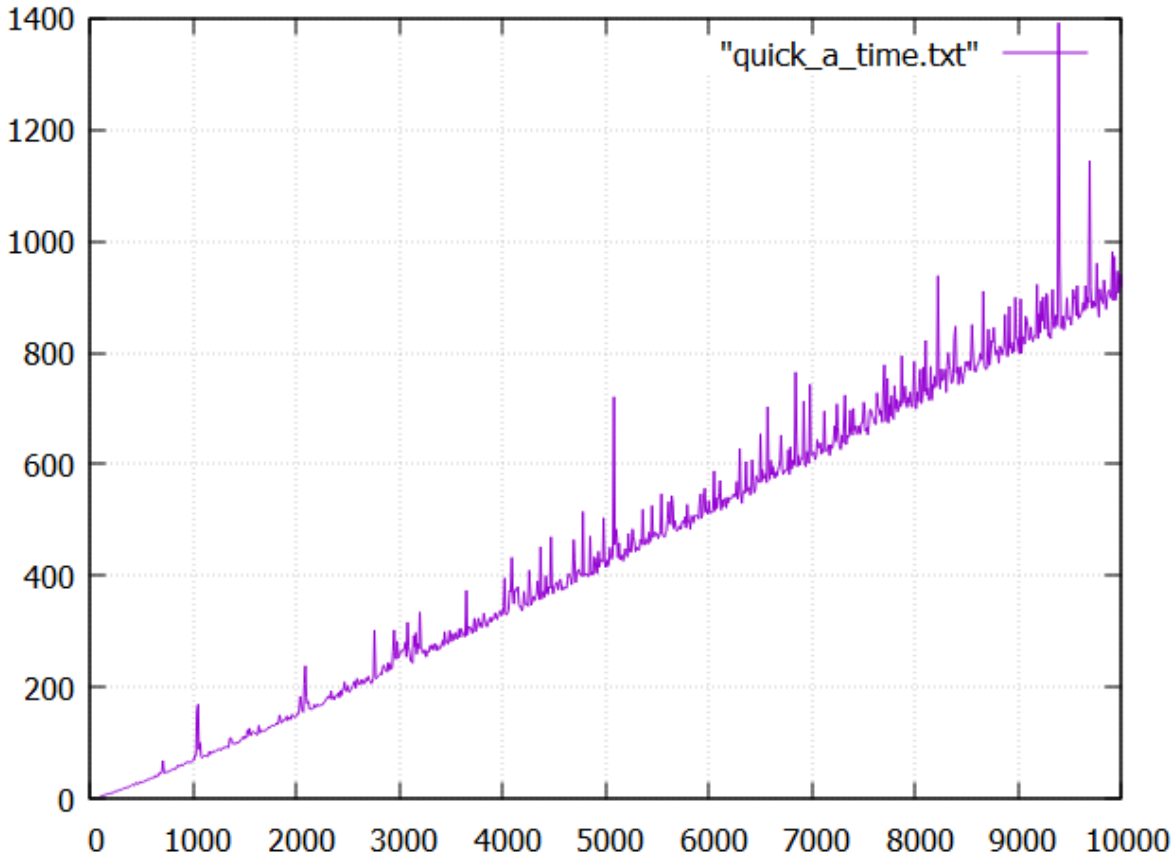

Figura 6 – Gráfico de tempo de execução para Quick Sort no caso médio

ANÁLISE

No caso médio, o Quick Sort possui um desempenho médio de $O(n \log n)$, onde n é o número de elementos na lista a ser ordenada. Este representa a situação mais comum de desempenho eficiente, onde a estratégia de escolha do pivô resulta em partições balanceadas na maioria dos casos. Essa eficiência torna-o uma escolha popular para aplicações práticas de ordenação.

No caso médio do Quick Sort:

- **Divisão Balanceada do Pivô:** O desempenho médio $O(n \log n)$ é alcançado quando, em média, o pivô divide o array em duas partes quase iguais em cada iteração recursiva. Isso minimiza o número de comparações e trocas necessárias para ordenar o array.
- **Complexidade de Tempo:** A complexidade de tempo média $O(n \log n)$ do Quick Sort é ideal para ordenação eficiente de grandes conjuntos de dados. Ele supera muitos outros algoritmos de ordenação devido à sua eficiência na estratégia de dividir para conquistar.

- **Eficiência de Espaço:** O Quick Sort opera in-place, o que significa que não requer espaço adicional significativo além da pilha de recursão. Isso o torna eficiente em termos de uso de memória comparado a algoritmos como o Merge Sort.

O Quick Sort não é um algoritmo estável. O modo como o particiona é implementado permite que elementos iguais troquem de posições relativas durante a sua execução.

É possível implementar uma versão estável do Quick Sort, mas para isso, ao invés de trocar o pivot diretamente com a primeira posição, o algoritmo deve fazer sucessivas trocas para "afastar" o pivot até a posição desejada – um processo semelhante à inserção ordenada. Essa implementação, contudo, piora significativamente o desempenho do Quick Sort.

O Quick Sort é in-place. O uso de memória auxiliar é constante em relação ao tamanho do array.

2.7 COMPARAÇÃO DOS CASOS DO QUICK SORT

Nesta seção, serão comparados os casos do Quick Sort: Pior caso, caso médio e melhor caso. O gráfico abaixo mostra o tempo de execução em relação ao tamanho do vetor para cada caso.

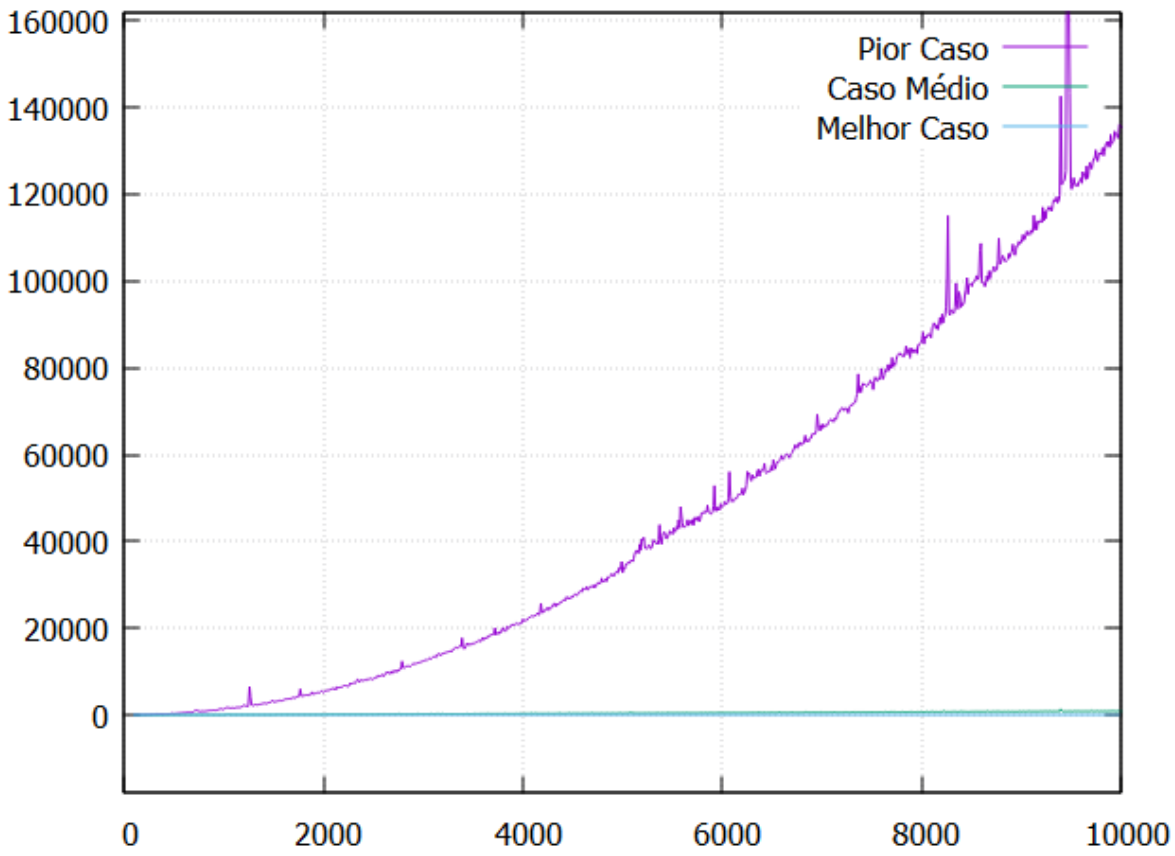

Figura 7 – Comparação de desempenho entre os casos do Quick Sort

2.7.1 ANÁLISE COMPARATIVA

A análise dos casos do Quick Sort destaca a importância da escolha do pivô e suas consequências diretas no desempenho do algoritmo. Enquanto o melhor e o caso médio são geralmente eficientes para a maioria dos conjuntos de dados, o pior caso pode levar a um desempenho muito inferior, sendo crucial considerar estratégias para evitar este cenário em aplicações práticas.

Em resumo, o Quick Sort é um algoritmo poderoso quando implementado corretamente, oferecendo desempenho eficiente na maioria dos casos, mas requer atenção especial para evitar degradações significativas de desempenho no pior caso.

2.8 COMPARAÇÃO DO MELHOR E MÉDIO CASO DO QUICK SORT

Nesta seção, serão comparados os casos do Quick Sort: Melhor caso e caso médio. O gráfico abaixo mostra uma visão mais aproximada da distinção entre os casos apesar de ambos serem $O(n \log n)$. O tempo de execução em relação ao tamanho do vetor.

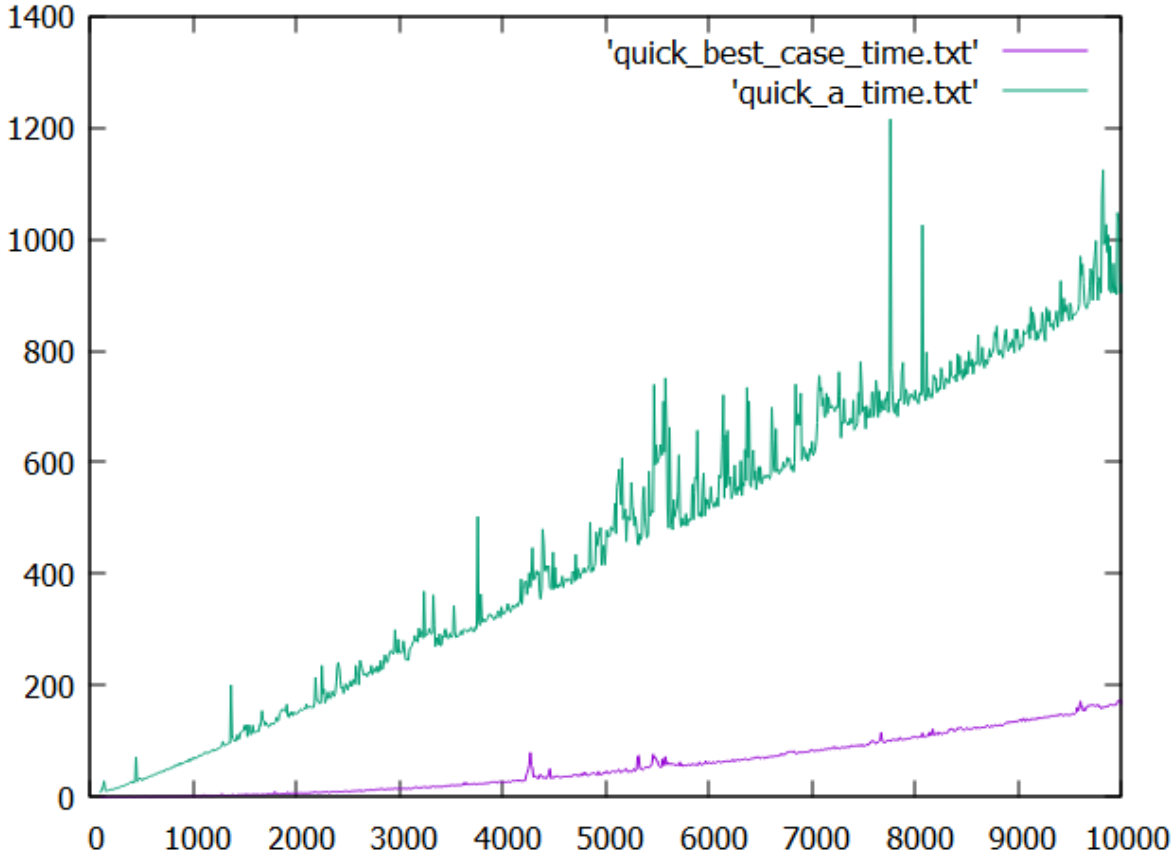

Figura 8 – Comparação de desempenho entre os casos B e W do Quick Sort

2.9 INSERTION SORT - MELHOR CASO

Nesta seção, será apresentada a análise do melhor caso do Insertion Sort, seguida pelo gráfico correspondente ao tempo de execução em relação ao tamanho do vetor.

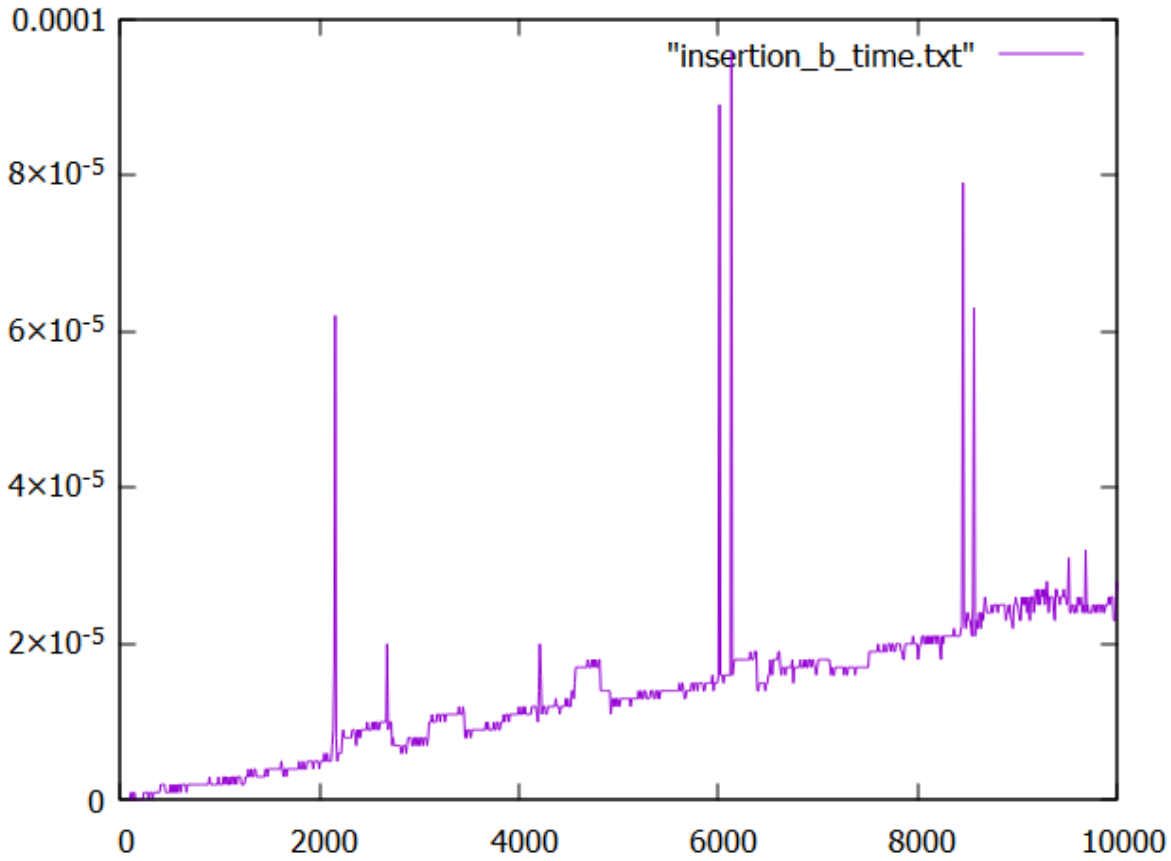

Figura 9 – Tempo de execução do Insertion Sort no melhor caso

ANÁLISE

No melhor caso, o Insertion Sort demonstra um desempenho otimizado quando o array está inicialmente ordenado ou quase ordenado. Graficamente, o algoritmo requer menos operações de comparação e troca de elementos, já que cada elemento novo é inserido na posição correta, deslocando-se apenas o necessário para manter a ordenação.

No melhor caso do Insertion Sort:

- Array Ordenado em Ordem Crescente: Quando o array já está ordenado de forma crescente, o Insertion Sort executa da maneira mais eficiente possível. Neste caso, cada elemento é comparado com o anterior e, como todos já estão na posição correta, nenhum deslocamento adicional é necessário.
- **Complexidade Linear:** O tempo de execução do Insertion Sort no melhor caso é $O(n)$, onde n é o número de elementos no array. Isso ocorre porque, em cada

iteração, apenas uma comparação é necessária para verificar que o elemento já está na posição correta.

- **Impacto na Eficiência:** O Insertion Sort permanece estável no melhor caso, assim como no pior caso, preservando a ordem relativa dos valores iguais. Não há trocas desnecessárias, o que contribui para a eficiência neste cenário.

  O Insertion Sort também continua sendo in-place, pois a ordenação é realizada dentro do próprio array sem a necessidade de estruturas auxiliares.

  No melhor caso, o Insertion Sort é extremamente eficiente, especialmente para arrays pequenos ou quase ordenados, pois a complexidade linear permite uma execução rápida.

2.10 INSERTION SORT - PIOR CASO

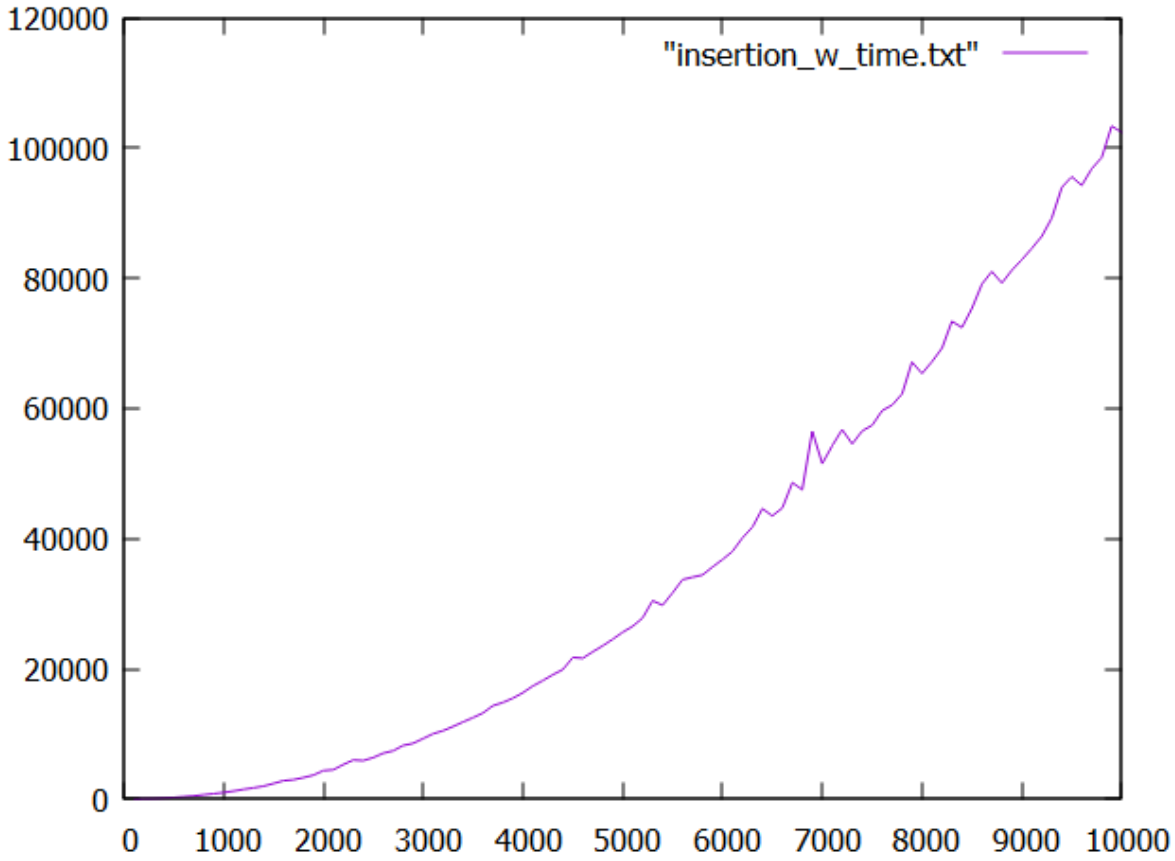

Figura 10 – Gráfico de tempo de execução para Insertion Sort no pior caso

ANÁLISE

No pior caso, o desempenho do Insertion Sort pode ser bastante impactado, quando o array está ordenado em ordem decrescente. Nesse cenário, cada elemento precisa ser movido para a posição correta no array ordenado, resultando em um número significativo de comparações e trocas.

No pior caso do Insertion Sort:

- **Array Ordenado em Ordem Decrescente:** Quando o array está ordenado de forma decrescente, cada elemento precisa ser movido para a sua posição correta no array ordenado. Isso resulta em muitas comparações e deslocamentos de elementos para trás.
- **Complexidade Quadrática:** O tempo de execução do Insertion Sort no pior caso é $O(n^2)$, onde n é o número de elementos no array. Isso ocorre porque, para cada elemento no array, pode ser necessário percorrer todos os elementos anteriores no pior cenário.

- **Impacto na Eficiência:** O Insertion Sort é estável porque mantém a ordem relativa dos valores iguais. Isso ocorre porque as trocas são feitas sempre com vizinhos. Os valores vão sendo "afastados" um a um, e não dando saltos. Por isso, um elemento qualquer nunca trocará de posição com elementos de mesmo valor.

  O Insertion Sort é in-place porque a ordenação é feita rearranjando os elementos no próprio array, ao invés de usar arrays ou outras estruturas auxiliares

  O desempenho do Insertion Sort no pior caso pode ser impraticável para grandes conjuntos de dados devido à sua complexidade quadrática.

2.11 INSERTION SORT - CASO MÉDIO

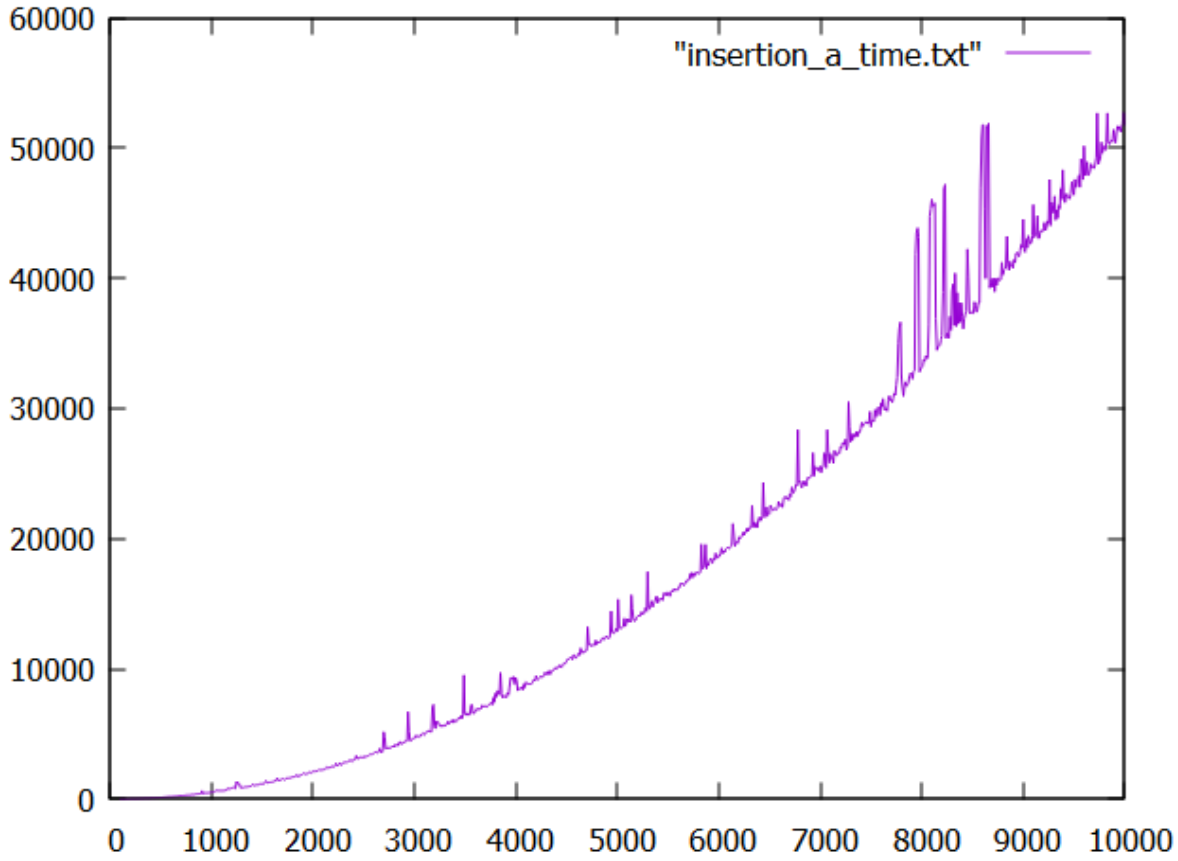

Figura 11 – Gráfico de tempo de execução para Insertion Sort no caso médio

ANÁLISE

No caso médio, o Insertion Sort demonstra um desempenho melhor em comparação com o pior caso, especialmente quando o array é aleatorizado. Nesse cenário, o tempo de execução do Insertion Sort é influenciado pelo número de inversões no array, ou seja, quantos elementos estão fora de ordem relativa.

No caso médio do Insertion Sort:

- **Array Aleatorizado:** Quando o array é aleatorizado, o tempo de execução do Insertion Sort é influenciado pelo número de inversões no array. Inversões referem-se ao número de pares de elementos que estão fora de ordem relativa.
- **Complexidade Quadrática:** Assim como no pior caso, o tempo de execução do Insertion Sort no caso médio é $O(n^2)$, onde n é o número de elementos no array. Isso ocorre porque o algoritmo pode precisar percorrer todos os elementos anteriores para inserir corretamente cada elemento no seu lugar.

- **Impacto na Eficiência:** O Insertion Sort é estável porque mantém a ordem relativa dos valores iguais. Isso ocorre porque as trocas são feitas sempre com vizinhos. Os valores vão sendo "afastados" um a um, e não dando saltos. Por isso, um elemento qualquer nunca trocará de posição com elementos de mesmo valor.

  O Insertion Sort é in-place porque a ordenação é feita rearranjando os elementos no próprio array, ao invés de usar arrays ou outras estruturas auxiliares

2.12 COMPARAÇÃO DO QUICK SORT COM MERGE SORT

Nesta seção, serão comparados os casos (Melhor, Médio) do Quick Sort com o Merge Sort. Os gráfico abaixo mostra a comparação entre eles, apesar de ambos serem $O(n \log n)$, o Melhor caso o Quick Sort chega a ser melhor que o Merge Sort. Contudo, a seu favor, o Merge Sort garante $O(n \log n)$ para qualquer caso, enquanto o Quick Sort pode ter ordenação $O(n^2)$ no pior caso, embora raro.. O tempo de execução em relação ao tamanho do vetor.

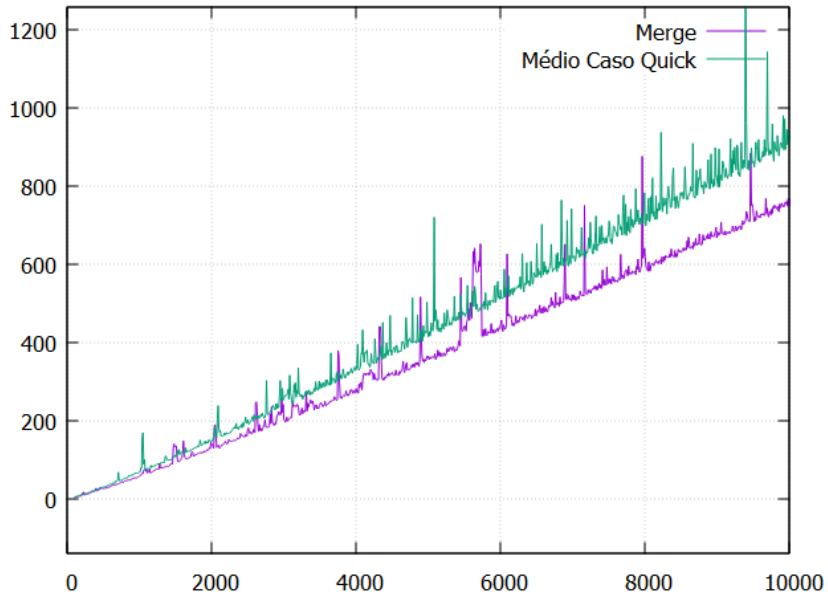

Figura 12 – Comparação de desempenho entre o Merge e Médio Caso Quick Sort

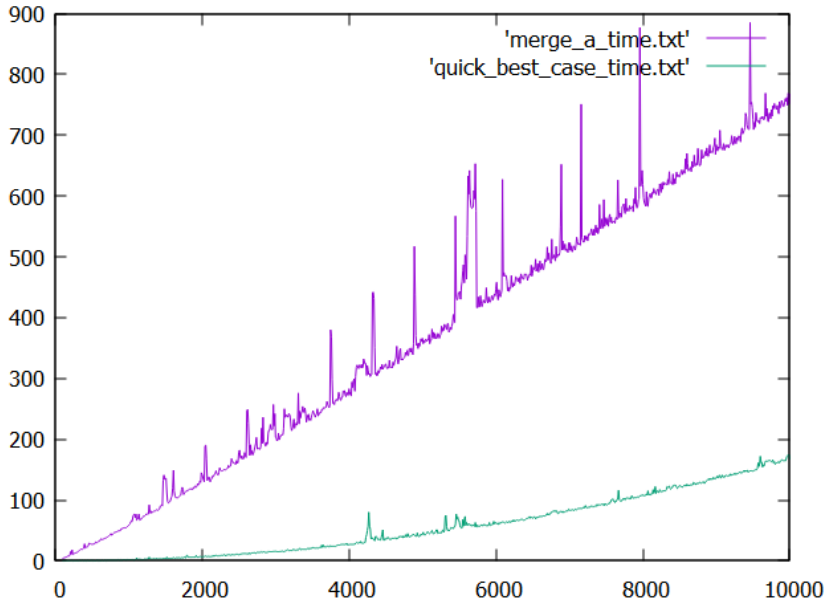

Figura 13 – Comparação de desempenho entre o Merge e Melhor Caso Quick Sort

3 CONCLUSÃO

Ao longo deste estudo, analisamos diversos algoritmos de ordenação, cada um com suas particularidades em relação ao desempenho em diferentes cenários. Abaixo, apresentamos um resumo das principais conclusões:

SELECTION SORT

O Selection Sort demonstrou ser um algoritmo simples e intuitivo, porém com um desempenho constante de $O(n^2)$ em todos os casos, o que o torna menos eficiente para grandes conjuntos de dados.

INSERTION SORT

O Insertion Sort, apesar de também possuir complexidade $O(n^2)$ no pior caso, mostra-se eficiente para pequenos conjuntos de dados ou em situações onde os elementos já estão quase ordenados, devido ao seu comportamento de inserção direta.

QUICK SORT

O Quick Sort mostrou-se altamente eficiente em média, com complexidade de tempo médio $O(n \log n)$. Seu desempenho é notável em comparação com outros algoritmos de ordenação, especialmente para grandes conjuntos de dados. No entanto, seu pior caso pode degradar para $O(n^2)$, tornando-o menos desejável em cenários onde a distribuição dos elementos é adversa.

MERGE SORT

O Merge Sort, com complexidade de tempo $O(n \log n)$ em todos os casos, demonstrou ser estável e eficiente para grandes volumes de dados, especialmente em situações onde a memória adicional para a mesclagem não é uma limitação.

DISTRIBUTION SORT

O Distribution Sort, utilizando buckets para distribuir os elementos, é eficiente quando os dados estão uniformemente distribuídos. No entanto, requer espaço adicional e pode não ser ideal para conjuntos de dados muito grandes ou com distribuições não uniformes.

COMPARAÇÃO E PREFERÊNCIAS

Para aplicações onde o desempenho médio é crucial e a distribuição dos dados não é conhecida antecipadamente, o Quick Sort e o Merge Sort são escolhas sólidas devido à sua complexidade média de $O(n \log n)$. Se a estabilidade é um requisito importante, o Merge Sort é preferível. Por outro lado, para pequenos conjuntos de dados ou situações onde a simplicidade e o espaço são críticos, o Insertion Sort pode ser mais adequado, especialmente se os dados já estão quase ordenados.

Em conclusão, a escolha do algoritmo de ordenação depende fortemente das características específicas do problema em mãos, como o tamanho do conjunto de dados, a distribuição dos elementos e as restrições de espaço e tempo. Cada algoritmo apresentado possui vantagens e desvantagens distintas, sendo essencial considerar esses fatores ao selecionar o mais adequado para uma aplicação específica.