Universidade Federal do Rio Grande do Norte – UFRN
Centro de Ensino Superior do Seridó – CERES
Departamento de Ciências Exatas e Aplicadas – DCEA
Bacharelado em Sistemas de Informação – BSI

Estrutura de Dados - Relatório I

Guilherme Costa de Medeiros

Orientador: Prof. Dr. João Paulo de Souza Medeiros

Relatório Técnico 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 unidade I.

Laboratório de Elementos do Processamento da Informação – LabEPI
Caicó, RN, 12 de junho de 2022

UFRN / Biblioteca Central Zila Mamede.

Catalogação da Publicação na Fonte.

Aluno, Guilherme Costa de Medeiros.

Estrutura de Dados - Relatório I. / Guilherme Costa de Medeiros. – Caicó, RN, 2022.

20 f.: il.

Orientador: Prof. Dr. João Paulo de Souza Medeiros.

Relatório Técnico – Universidade Federal do Rio Grande do Norte. Centro de Ensino Superior do Seridó. Bacharelado em Sistemas de Informação.

1. Algoritmos. 2. Ordenação. 3. Análise. 4. Gráfico. I. Professor, João Paulo de Souza Medeiros. II. Universidade Federal do Rio Grande do Norte. III. Estrutura de Dados - Relatório I.

RN/UF/BCZM CDU 004.7

Resumo

Este relatório tem como objetivo apresentar uma análise associada ao problema de ordenação, usando como base para obtenção dos resultados os seguintes algoritmos: Insertion-Sort, Merge-Sort, and Quick-Sort. No relatório foram desenvolvidas três atividades: A primeira delas foi a geração de gráficos usando o 'software' Gnuplot com estimativas práticas do tempo de execução com base no tamanho da entrada; A segunda atividade foi composta por uma análise analítica dos resultados obtidos dos três algoritmos; para finalizar, foi feita uma comparação sobre o desempenho de cada algoritmo em relação ao custo em tempo e memória.

Palavras-chave: Algoritmos; Ordenação; Gráfico; Análise.

Abstract

This report aims to present an analysis associated with the sorting problem, using the following algorithms as a basis for obtaining results: Insertion-Sort, Merge-Sort, and Quick-Sort. In the report three activities were developed: The first of them was the generation of graphs using the Gnuplot software with practical estimates of the execution time based on the input size; The second activity was composed by an analytical analysis of the results obtained from the three algorithms; to finish, a comparison was made on the performance of each algorithm regarding the cost in time and memory.

Keywords: Algorithms; Sorting; Chart; Analysis.

Sumário

1. Introdução

Na computação existe uma série de algoritmos que utilizam diferentes técnicas de ordenação para organizar um conjunto de dados. Neste relatório será feito um estudo utilizando as seguintes técnicas de soluções algorítmicas: inserção, divisão e conquista.

É importante salientar que, para gerar os gráficos de cada algoritmo, foram utilizados 100 vetores de tamanhos diferentes com entradas randômicas para cada complexidade. O primeiro vetor inicia com tamanho igual à 100 posições e vai até um vetor de 10.000 posições, pulando de 100 em 100.

1.1 Objetivos

Reunir informações sobre a execução dos algoritmos selecionados para fazer uma análise aprofundada com base nos resultados obtidos na execução de cada código utilizado.

1.2 Organização do trabalho

Este relatório está dividido em cinco capítulos: Introdução, Insertion-Sort, Merge-Sort, Quick-Sort e Resultados. A introdução visa apresentar o objetivo e o contexto de forma geral e resumida do relatório. Os capítulos dos algoritmos Insertion-Sort, Merge-Sort, Quick-Sort são divididos em dois subcapítulos, onde inicialmente há uma pequena explicação do funcionamento de cada um, seguido dos Gráficos do tempo de execução de cada complexidade, que apresenta os resultados com base em estimativas práticas obtidas, acompanhada de suas análises. O segundo subcapítulo é a Análise analítica, onde são apresentadas os cálculos das complexidades com base no tempo de execução. Por fim, nos Resultados, é feita uma comparação sobre o desempenho de cada resultado obtido dos algoritmos ao longo do relatório.

2. Insertion-Sort

O Insertion-Sort tem como rotina base a ordenação por inserção. É um método estável do qual a ideia é executar várias vezes essa rotina para ordenar um vetor, percorrendo o vetor da esquerda para a direita e à medida que avança vai ordenando os elementos à esquerda.

2.1 Gráficos do tempo de execução

2.1.1 Melhor caso

O melhor caso do Insertion-Sort é quando ele recebe um vetor já ordenado. Isto ocorre, pois, todos os elementos do vetor já estão em suas devidas posições, fazendo com que não seja necessário mover nenhum elemento de sua posição atual.

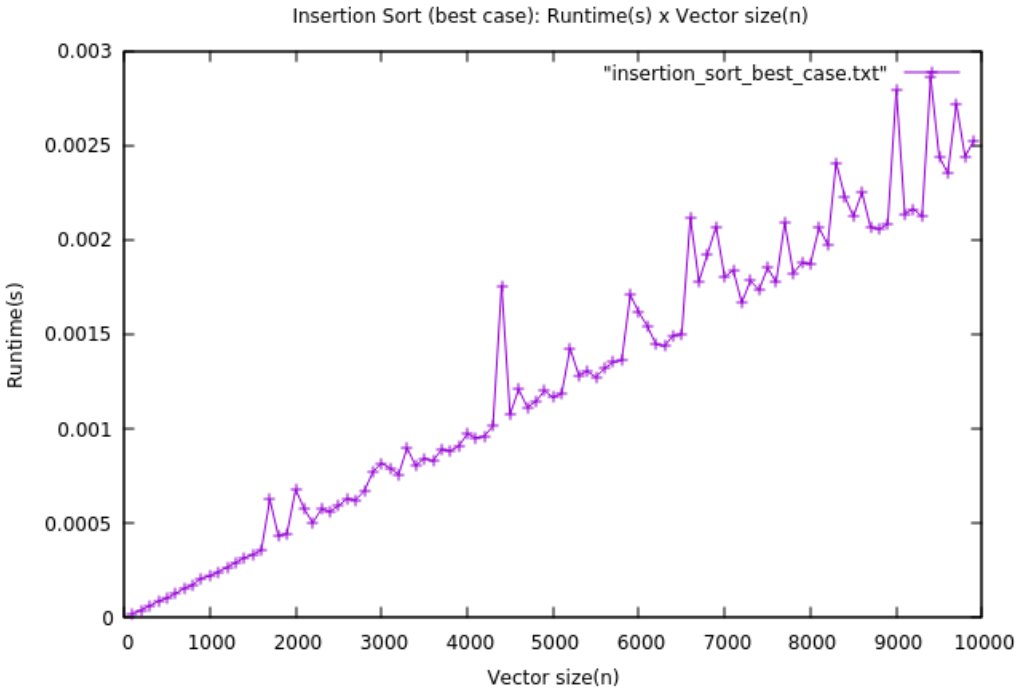

Figura 2.1: Insertion-Sort: melhor caso

Como a inserção ordenada é executada n vezes, o custo total é linear, ou O(n). Também é possível notar isso analisando o gráfico gerado, que aumenta o tempo de execução linearmente a medida que a quantidade de elementos no vetor vai aumentando.

2.1.2 Caso médio

O caso médio do Insertion-Sort é quando ele recebe um vetor embaralhado. Dessa forma o algoritmo possuirá um tempo de execução médio entre todas as entradas possíveis.

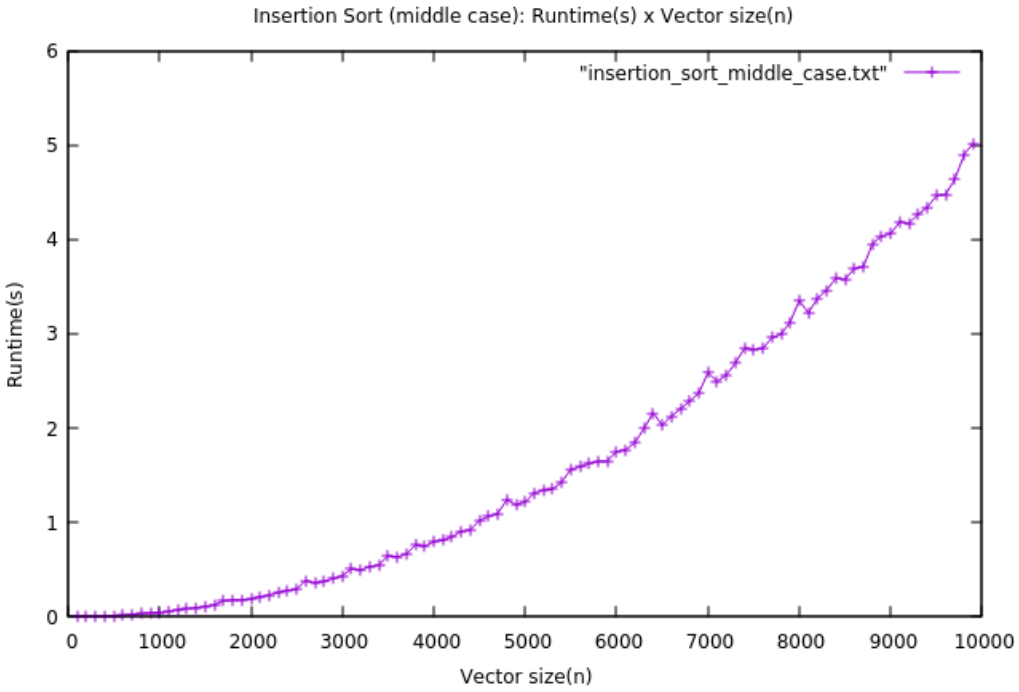

Figura 2.2: Insertion-Sort: caso médio

O custo total do caso médio será O(n^2), ou seja, será quadrático. Também é possível notar isso analisando o gráfico gerado, que aumenta o tempo de execução quadraticamente, principalmente nas últimas execuções, a medida que a quantidade de elementos no vetor vai aumentando.

2.1.3 Pior caso

O pior caso do Insertion-Sort é quando ele recebe um vetor ordenado em ordem reversa. Isto ocorre, pois, toda tentativa de inserção ordenada deve percorrer o vetor todo à esquerda trocando os elementos até encaixar o atual na primeira posição.

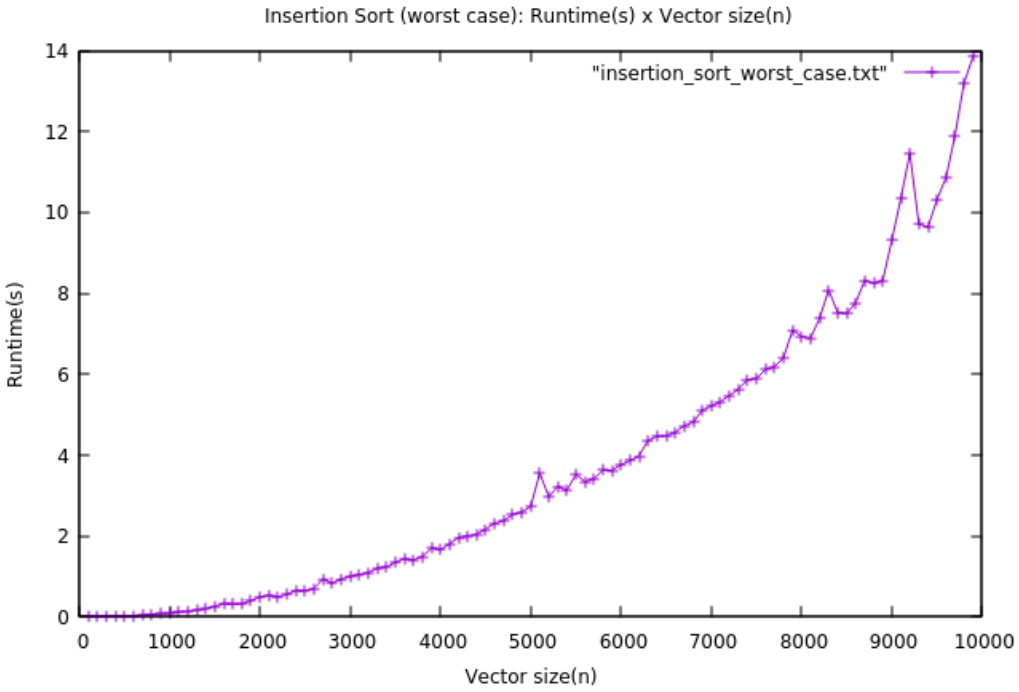

Figura 2.3: Insertion-Sort: pior caso

No pior caso o laço interno realizará a quantidade máxima de iterações, logo o custo total, assim como o caso médio, será quadrático, ou $O(n^2)$. Também é possível notar isso analisando o gráfico gerado, que aumenta o tempo de execução quadraticamente, principalmente nas últimas execuções, a medida que a quantidade de elementos no vetor vai aumentando.

2.1.4 Comparação

Por fim, iremos analisar os resultados obtidos de todas as complexidades do Insertion-Sort. Com o gráfico apresentado abaixo podemos notar a diferença do tempo de execução de cada um, onde o pior caso possui tempo de execução tão lento que o melhor caso na comparação chega a parecer uma constante.

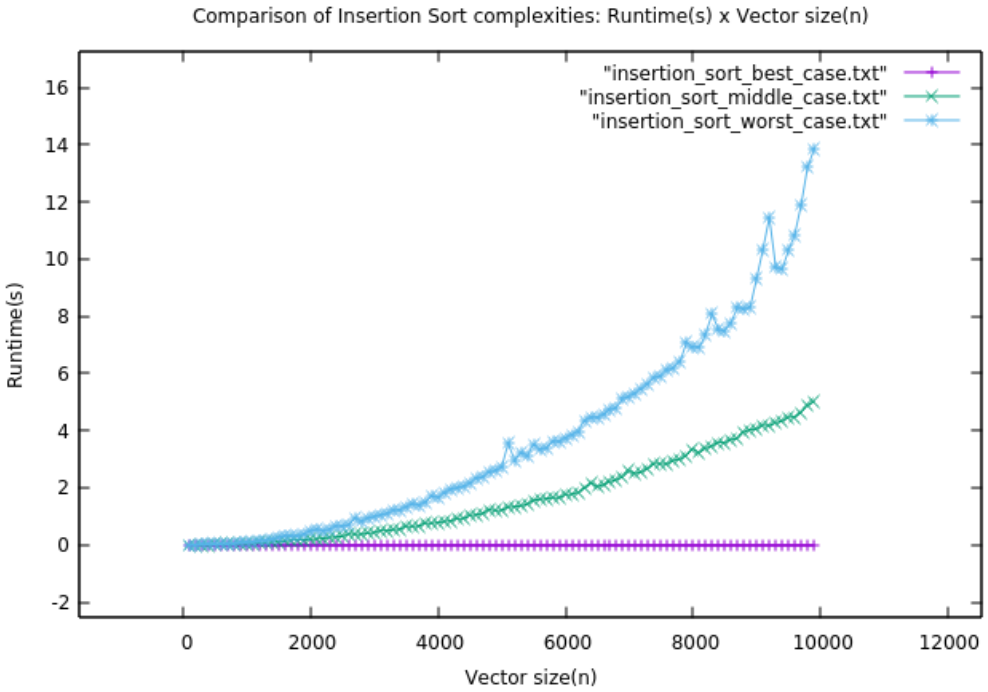

Figura 2.4: Insertion-Sort: comparação das complexidades

2.2 Análise analítica

2.2.1 Melhor caso

A análise do cálculo de complexidade do melhor caso do algoritmo Insertion-Sort nos permitiu observar mais uma vez que o tempo de execução é linear, ou O(n), sendo dado por uma função f(n) = an + b.

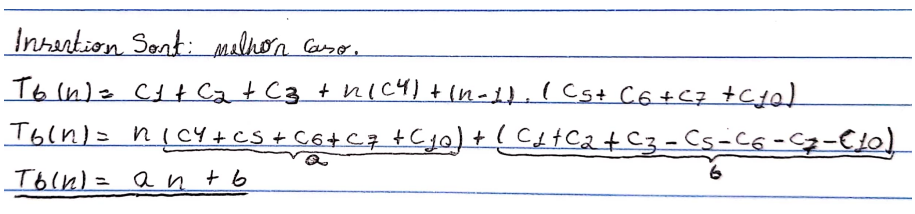

Figura 2.5: Insertion-Sort: cálculo do melhor caso

2.2.2 Pior caso

A análise do cálculo de complexidade do pior caso do algoritmo Insertion-Sort permitiu observar que o tempo de execução no pior caso é quadrático, ou O(n^2), sendo dado por uma função f(n) = an^2 + bn + c.

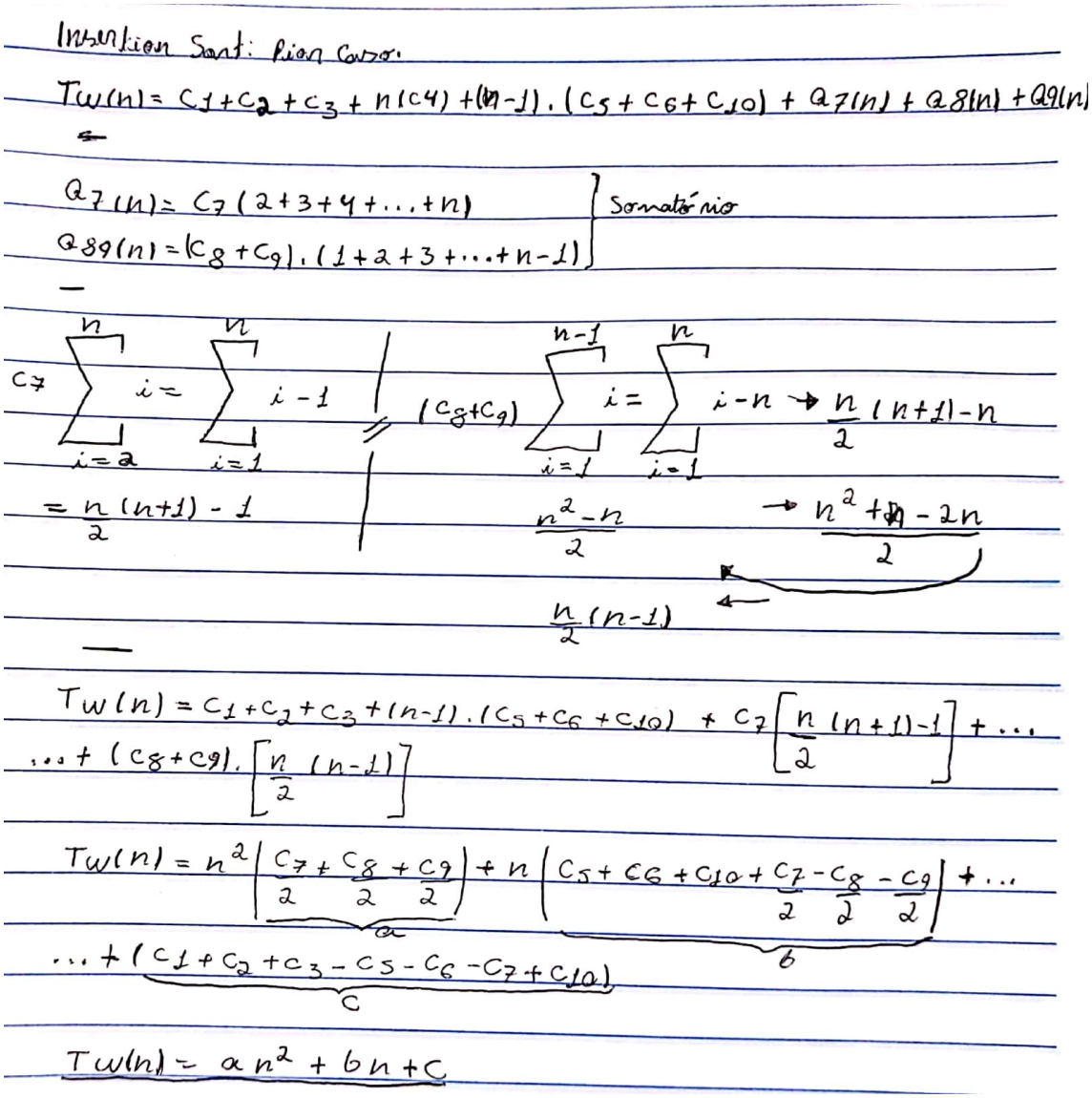

Figura 2.6: Insertion-Sort: cálculo do pior caso

3. Merge-Sort

O Merge-Sort é um algoritmo eficiente de ordenação por divisão e conquista. É um método estável que age em três etapas, que seria dividir o problema em pedaços menores (subproblemas), conquistar cada pedaço resolvendo-os recursivamente e depois combinar (merge) as soluções dos subproblemas em uma solução para o problema original.

Uma característica importante do Merge-Sort é que sua eficiência é a mesma ($n \log_n$) para o melhor, pior e para o caso médio. Isso nos dá uma garantia de que, independente da disposição dos dados em um vetor, a ordenação será eficiente.

3.1 Gráficos do tempo de execução

3.1.1 Caso base

Como foi citado, o Merge-Sort garante eficiência $n \log_n$, independente do caso (melhor, pior ou médio). Isso ocorre porque a divisão do problema sempre gera dois subproblemas com a metade do tamanho do problema original. Ou seja, independente dos dados, estamos sempre dividindo o vetor na metade. Portanto, a relação de recorrência é única e, quando resolvida, sempre nos fornece um custo, $n \log_n$.

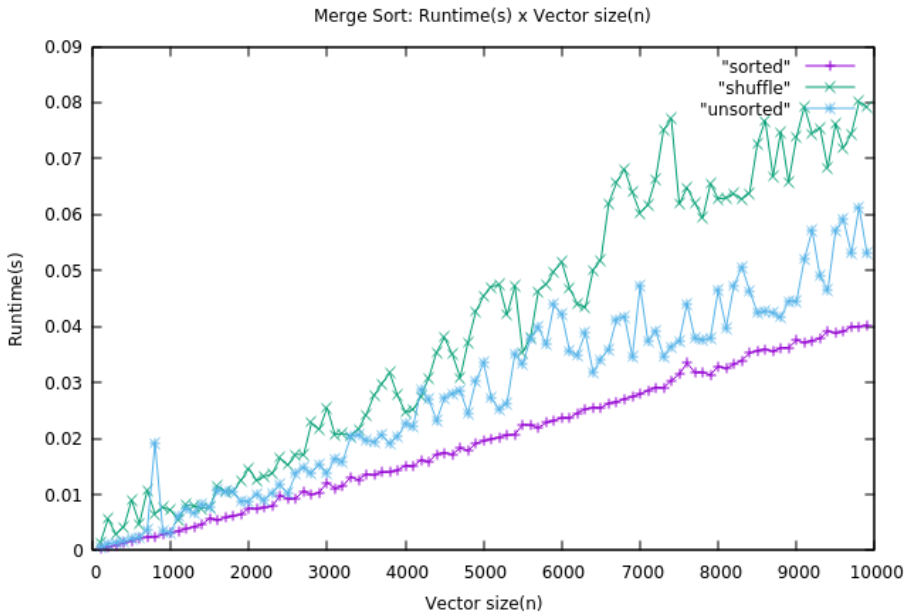

Figura 3.1: Merge-Sort: caso base

O caso base da Merge-Sort será linear, ou O(n). Também é possível notar isso analisando o gráfico gerado, que aumenta o tempo de execução linearmente a medida que a quantidade de elementos no vetor vai aumentando. Para obter os resultados do gráfico foram passados uma série de vetores ordenados, embaralhados e ordenados de forma reversa, e apesar de variar o tempo de execução é possível notar que possui pouca oscilação.

3.2 Análise analítica

3.2.1 Caso base

A análise do cálculo de complexidade do algoritmo Merge-Sort nos permitiu observar mais uma vez que o tempo de execução é linear, ou O(n), sendo dado por uma função f(n) $= n \log_{2^n} + n$.

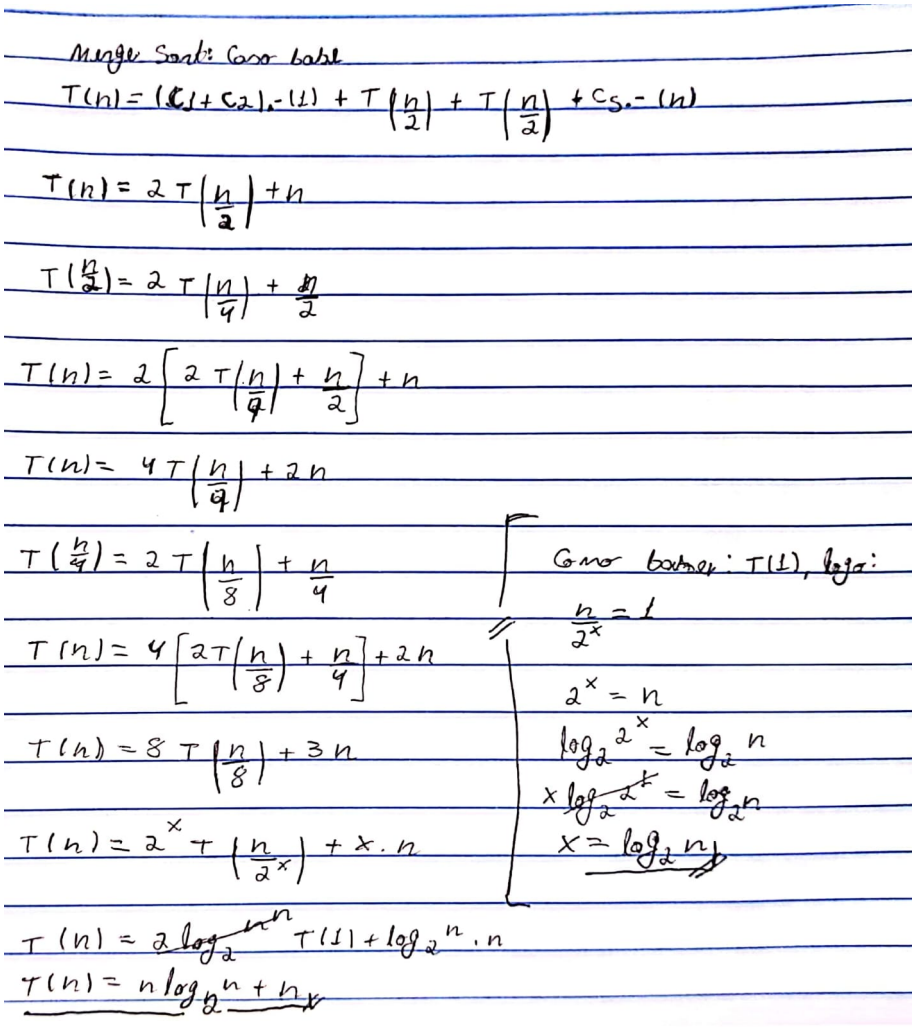

Figura 3.2: Merge-Sort: cálculo do caso base

4. Quick-Sort

O Quick-Sort é, assim como o Merge-Sort, um algoritmo recursivo e eficiente de ordenação por divisão e conquista, porém, ele não é estável. O funcionamento do Quick-Sort baseia-se em uma rotina fundamental cujo nome é particionamento. Particionar significa escolher um número qualquer presente no vetor, chamado de pivot, e colocá-lo em uma posição tal que todos os elementos à esquerda são menores ou iguais e todos os elementos à direita são maiores.

4.1 Gráficos do tempo de execução

4.1.1 Melhor caso

O melhor caso do Quick-Sort depende de seu particionamento, que seria quando o pivot sempre estivesse no meio do vetor. Dessa forma, obteremos uma árvore binária na recursão em que a esquerda tem metade do tamanho do array e a direita também tem a metade do tamanho.

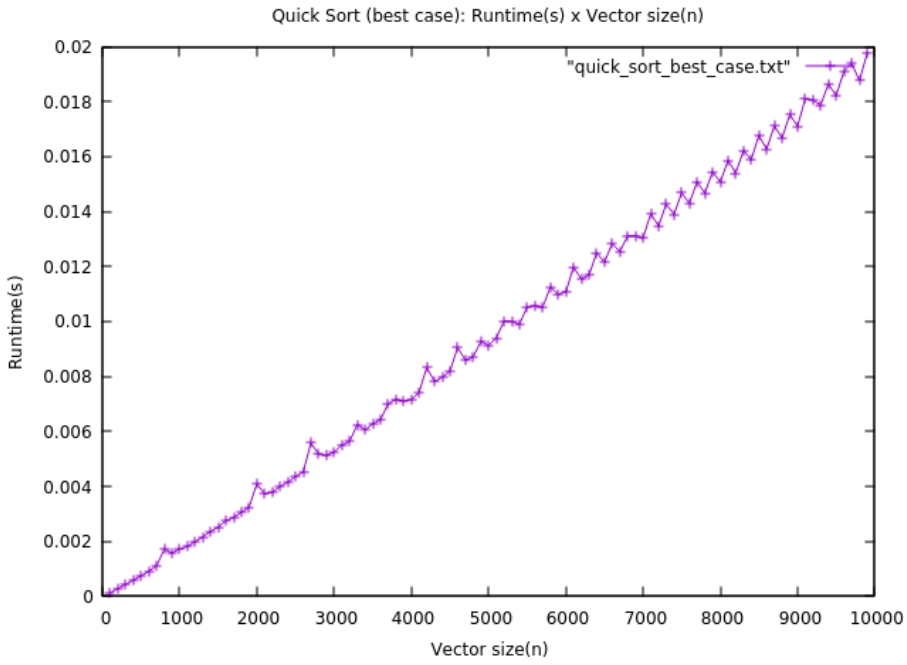

Figura 4.1: Quick-Sort: melhor caso

Como foi dito, o custo do melhor caso seria $\log_n$ execuções do particionamento, que é O(n). Como resultado, o melhor caso do Quick-Sort é $O(n \log_n)$, ou linearitmica. Também é possível notar isso analisando o gráfico gerado, que aumenta o tempo de execução a medida que a quantidade de elementos no vetor vai aumentando.

4.1.2 Caso médio

O caso médio do Quick-Sort é quando ele recebe um vetor embaralhado. Dessa forma o algoritmo possuirá um tempo de execução médio entre todas as entradas possíveis.

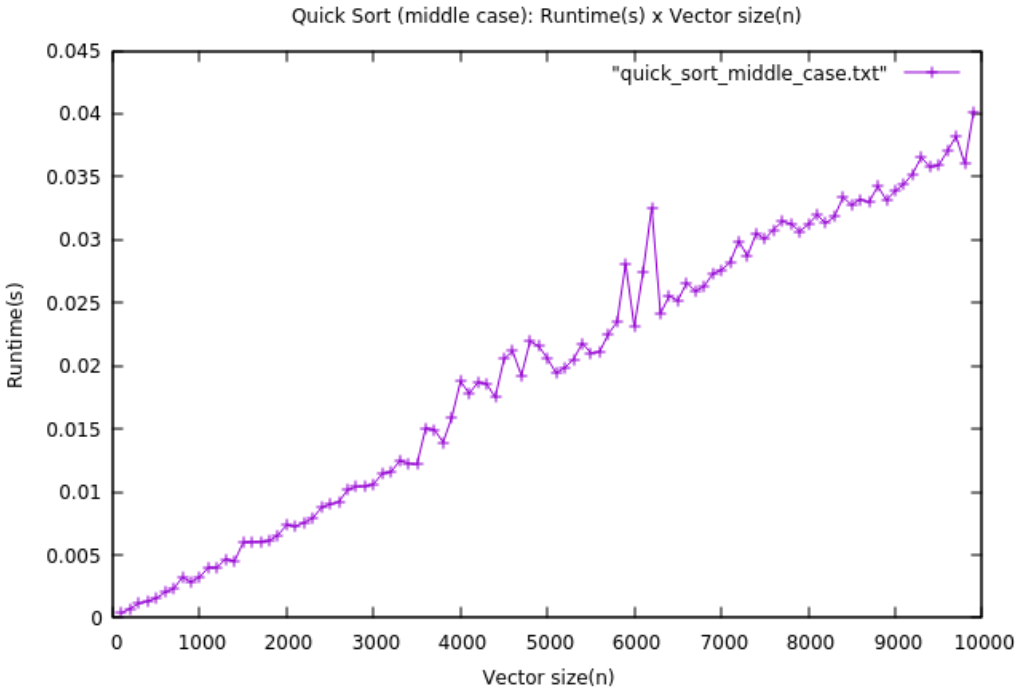

Figura 4.2: Quick-Sort: caso médio

Assim como o melhor caso, o custo do caso médio do Quick-Sort é $O(n \log_n)$, ou linearitmica. Também é possível notar isso analisando o gráfico gerado, que aumenta o tempo de execução a medida que a quantidade de elementos no vetor vai aumentando.

É importante notar que o caso médio do Quick-Sort é muito próvavel de acontecer. A probabilidade de não escolher recorrentemente péssimos pivots quando temos um vetor embaralhado é alta. Por meio da teoria desse caso, podemos saber que mesmo alternando péssimos particionamentos (pivot longe da metade do vetor) e bons particionamentos (pivot próximo à metade do vetor), o algoritmo ainda seria $O(n \log_n)$.

4.1.3 Pior caso

O pior caso do Quick-Sort é quando ele recebe um vetor já ordenado, tanto em ordem crescente quando decrescente. Isso ocorre porque o pivot sempre vai dividir o vetor em duas porções de tamanho 0 e n-1, respectivamente.

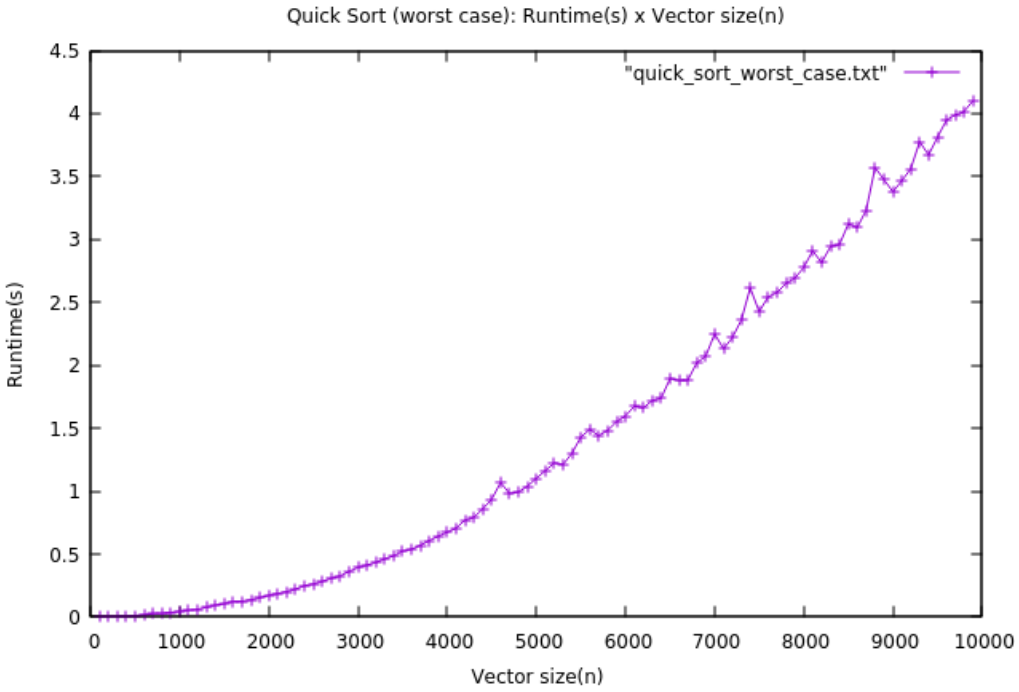

Figura 4.3: Quick-Sort: pior caso

O custo do pior caso do Quick-Sort é O(n^2), ou quadrático. Também é possível notar isso analisando o gráfico gerado, que aumenta o tempo de execução quadraticamente a medida que a quantidade de elementos no vetor vai aumentando.

4.1.4 Comparação

Por fim, iremos analisar os resultados obtidos de todas as complexidades do Quick-Sort. Com o gráfico apresentado abaixo podemos notar a diferença do tempo de execução de cada um, onde o pior caso possui tempo de execução tão lento que o melhor caso e o caso médio chegam a parecer uma constante.

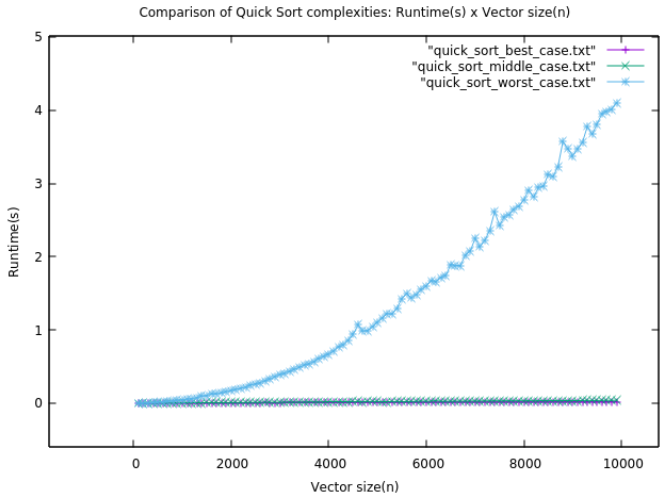

Figura 4.4: Quick-Sort: comparação das complexidades

Então, gerei um segundo gráfico dessa comparação apenas com o melhor e o pior caso para obter uma melhor visualização destes resultados. Como é possível observar, os tempos de execução deles ficaram bastante parecidos, pois ambos são $O(n \log_n)$.

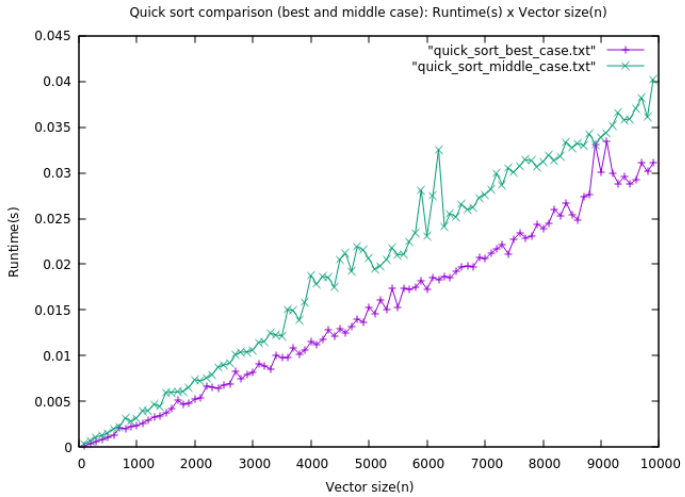

Figura 4.5: Quick-Sort: comparação do caso médio e do melhor caso

4.2 Análise analítica

4.2.1 Melhor caso

A análise do cálculo de complexidade do melhor caso do algoritmo Quick-Sort nos permitiu observar mais uma vez que o tempo de execução é linearitmica, ou $O(n \log_n)$, sendo dado por uma função $f(n) = n \log_{2^n}$.

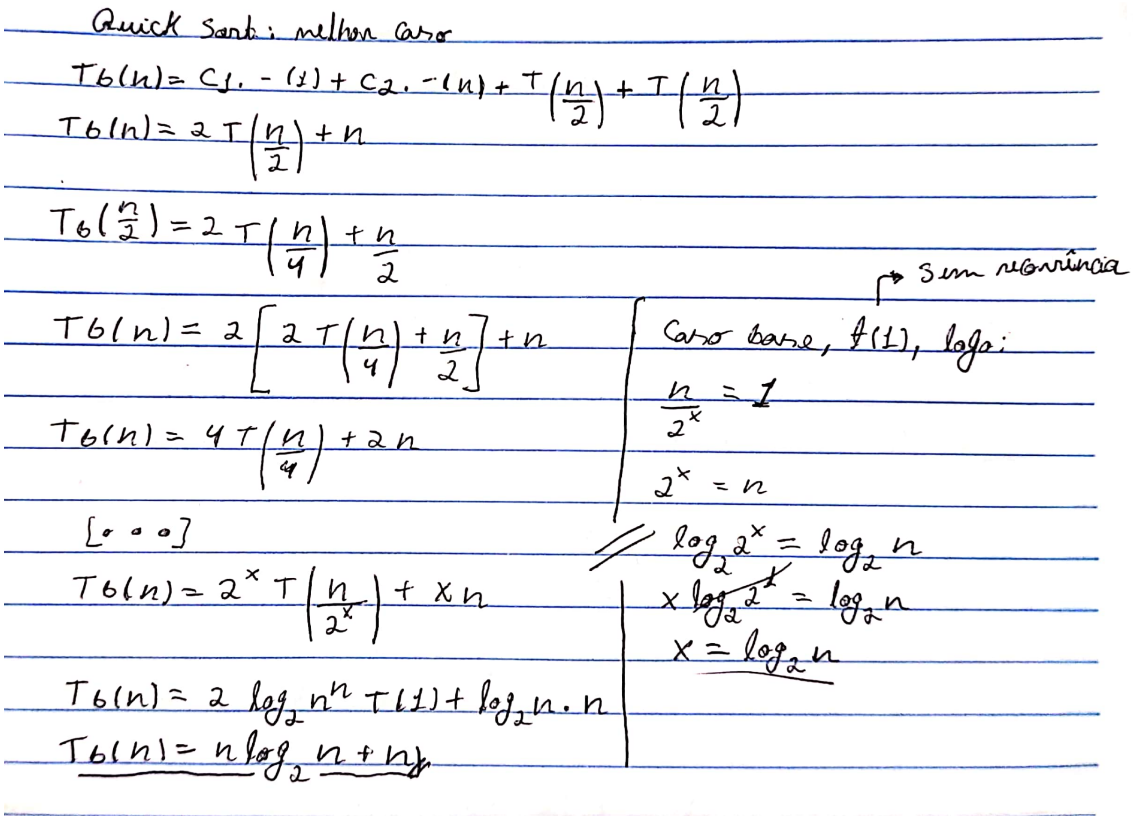

Figura 4.6: Quick-Sort: cálculo do melhor caso

4.2.2 Pior caso

A análise do cálculo de complexidade do pior caso do algoritmo Quick-Sort nos permitiu observar mais uma vez que o tempo de execução é quadrático, ou O(n^2), sendo dado por uma função f(n) = an^2 + bn + c.

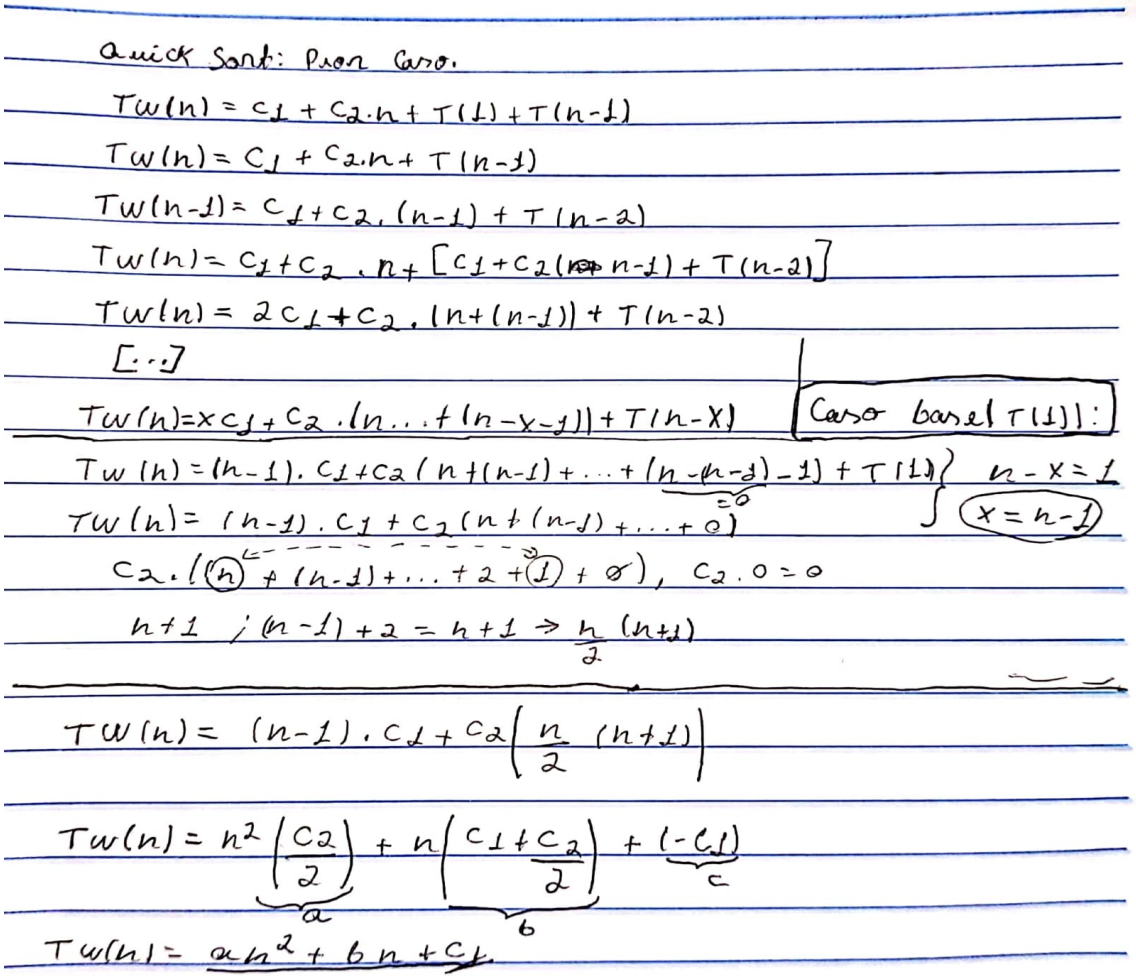

Figura 4.7: Quick-Sort: cálculo do pior caso

5. Resultados

Agora faremos uma análise dos resultados obtidos de cada algoritmo ao longo do relatório para fazer uma comparação geral sobre o desempenho do tempo de execução e memória. Para esse passo, usarei os resultados obtidos dos *casos médios* dos algoritmos estudados.

De primeira já é possível observar que o algoritmo Insertion-Sort é menos eficiente do que o Merge-Sort e o Quick-Sort. Isso é possível de observar, pois, o Insertion-Sort levou mais tempo para realizar a ordenação, mesmo nas situações em que seus vetores eram de tamanho inferior aos utilizados nos testes dos demais algoritmos. Assim, já sabemos que o Insertion-Sort por demandar mais tempo, também gasta mais memória.

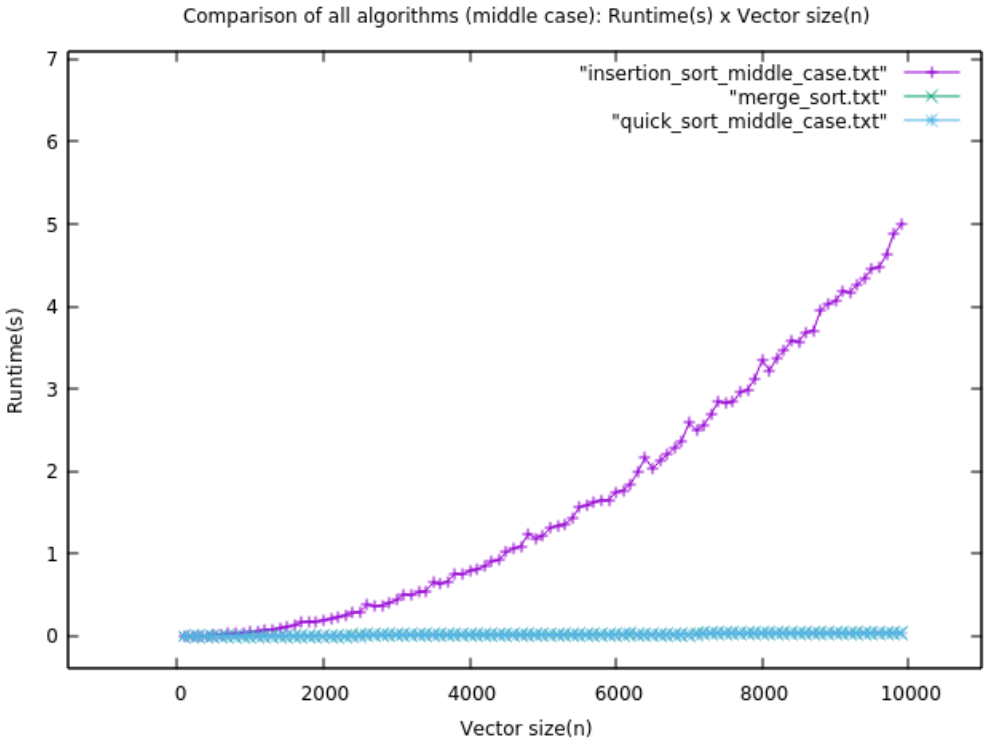

Figura 5.1: Comparação de todos os algoritmos: caso médio

Comparando o Merge-Sort e o Quick-Sort obtemos uma variação de milissegundos, porém, podemos visualizar que o Quick-Sort é ligeiramente mais rápido que o Merge-Sort em algumas situações. Isto ficará mais evidente logo a frente, que será apresentado uma tabela com os valores definitivos de todos os algoritmos.

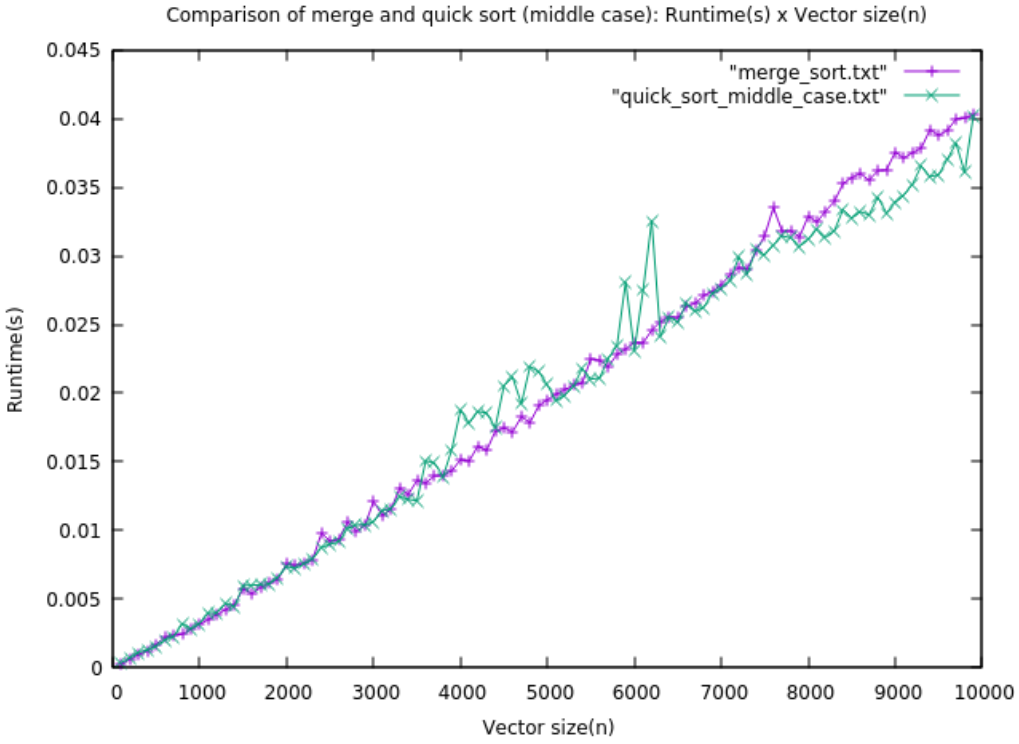

Figura 5.2: Comparação do Merge-Sort e Quick-Sort: caso médio

Dessa forma, podemos estimar que apesar do Quick-Sort possuir o pior caso mais demorado dos demais algoritmos analisados, seu fator constante oculto na notação Big-O é muito bom. Na prática, e também com base nos resultados obtidos ao longo de todo o relatório, o Quick-Sort tem desempenho melhor que o Merge-Sort, e é significativamente melhor do que o Insertion-Sort. Já o Insertion-Sort é o algoritmo que apresenta maior custo em tempo e memória e o Merge-Sort apresenta custo médio.

Na tabela abaixo é possível visualizar o tempo de cada algoritmo para tamanhos diferentes de vetores, onde podemos comprovar cada observação feita neste capítulo.

Tamanho do vetor	Insertion-Sort	Merge-Sort	Quick-Sort
1.000	0.048716783523559	0.00319004058837	0.00323152542114
2.000	0.19151401519775	0.00757980346679	0.00738215446472
3.000	0.43672108650207	0.01212382316589	0.01061868667602
4.000	0.79663133621215	0.01516461372375	0.01877379417419
5.000	1.21783590316772	0.01951575279235	0.02058529853820
6.000	1.75249648094177	0.02370047569274	0.02310919761657
7.000	2.58825945854187	0.02788519859313	0.02759790420532
8.000	3.35251259803771	0.03288054466247	0.03120470046997
9.000	4.06099629402160	0.03754830360412	0.03387904167175