Estrutura de Dados - Atividade Prática 1: Comparação da eficiência entre os algorítmos de ordenação Insertion Sort, Merge Sort e Quick Sort

Douglas Mateus Soares Cândido da Silva* João Paulo de Souza Medeiros†

Recebido em 12 de novembro de 2020, aceito em 12 de novembro de 2020.

Resumo

Este trabalho busca demonstrar e analisar a eficiência dos algorítmos de ordenação Insertion Sort, Merge Sort e Quick Sort fazendo o uso da linguagem de programação C++ para suas implementações e testando suas performances na ordenação de arrays de diversos tamanhos. O principal atributo analisado foi o tempo de execução dos algorítmos e como ele era afetado conforme o tamanho da lista era alterado para menor e para maior. Ao final dos testes, foram gerados arquivos e gráficos contendo as medições dos tempos de execução e, a partir de sua análise, comprovou que o Insertion Sort é o algorítmo de ordenação mais eficiente entre os três quando a lista a ser ordenada é pequena, porém é o pior quando a lista é muito grande. Comprovou-se também que o Merge Sort é ineficiente para listas pequenas e bastante eficiente para listas grandes, porém menos eficiente que o Quick Sort. Percebeu-se que o Quick Sort é o melhor entre os três algorítmos de ordenação quando a lista é muito grande (Superando o Merge Sort), porém sendo um pouco menos ineficiente que o Merge Sort quando a lista é pequena. Calculou-se também a Complexidade de Tempo (Time Complexity) dos três algorítmos no que se refere às suas performances no melhor caso, pior caso e caso médio.

1 Introdução

Como destaca [3], estamos vivendo em um meio Técnico Científico e Informacional, onde a tecnologia avança com uma velocidade sem precedentes e a busca por criar tecnologia realmente de qualidade é voraz por parte dos pesquisadores e estudiosos. Dito isso, é de vital importância saber como analisar a eficiência de uma determinada solução e, com isso, saber se ela é viável para implementação ao compara-la com outras soluções. Na área da Tecnologia de Informação, esse requisito de controle de qualidade dos novos artefatos tecnológicos desenvolvidos se dá principalmente na análise da eficiência dàs soluções algorítmicas pensadas para resolver, ou otimizar soluções já encontradas, problemas do mundo real.

[2] ressalta que existem alguns campos de estudo dentro da Ciência da Computação que possuem problemas que despertam a atenção de pesquisadores e entusiastas da área, um dos problemas mais fascinantes é o desenvolvimento de algorítmos eficientes na ordenação e reorganização de itens de listas, seguindo uma relação de ordem. Essas soluções algorítmicas trazem como principal benefício a melhora no desempenho dos sistemas de busca, pois esses sistemas, majoritariamente, fazem uso de uma ordenação prévia dos registros para melhorar o desempenho da pesquisa.

*Aluno do Bacharelado em Sistemas de Informaçãoda Universidade Federal do Rio Grande do Norte. (e-mail: douglas.candido1997@gmail.com)

†Professor do Departamento de Computação e Tecnologiada Universidade Federal do Rio Grande do Norte. Coordenador do Laboratório de Elementos do Processamento da Informação. (e-mail: jpsm@dct.ufrn.br)

Dito isso, a análise da eficiência de algorítmos de ordenação com relação ao seu tempo de execução é o principal assunto abordado nesse trabalho.

2 Fundamentação

Algorítmos:

Segundo [1], um algorítmo é qualquer procedimento computacional bem definido que recebe como entrada um valor, ou um conjunto de valores, e produz um outro valor, ou conjunto de valores, como saída. O conceito de algorítmo existe há incontáveis anos e o uso de sua definição pode ser atribuído aos matemáticos que foram pioneiros na área de Ciências Exatas. Alguns algorítmos famosos e tão antigos quanto os seus criadores são, por exemplo, a Peneira de Eratóstenes para encontrar números primos e o algoritmo de Euclides para a divisão de números. Os algorítmos, de modo geral, são criados para resolver problemas e formalizar as devidas soluções em uma sequência de passos não ambíguos e finitos. Diferentes algoritmos podem realizar a mesma tarefa usando um conjunto diferenciado de instruções em mais ou menos tempo e espaço do que os seus pares. Tal diferença pode ser reflexo da complexidade computacional aplicada ao problema que deve ser resolvido e ao algorítmo em si, que depende de estruturas de dados adequadas e de uma sequência de passos tão eficiente quanto possível, por isso o verdadeiro desafio dos profissionais que atuam na área de Tecnologia da Informação é criar soluções algorítmicas de qualidade.

Complexidade de Tempo (Time Complexity):

Segundo [1], na Ciência da Computação, a complexidade do tempo é a complexidade computacional que descreve a quantidade de tempo que leva para executar um algoritmo, ou seja, sua eficiência quanto à sua velocidade. A complexidade do tempo é comumente estimada pela contagem do número de operações elementares realizadas pelo algoritmo, supondo que cada operação elementar leve um determinado tempo para ser executada. Assim, considera-se que o tempo gasto e o número de operações elementares realizadas pelo algoritmo diferem por no máximo um fator constante.

Uma vez que o tempo de execução de um algoritmo pode variar entre diferentes entradas do mesmo tamanho, normalmente se considera a complexidade de tempo do pior caso, que é a quantidade máxima de tempo necessária para entradas de um determinado tamanho. Menos comum, e geralmente especificado explicitamente, é a complexidade de caso médio, que é a média do tempo gasto em entradas de um determinado tamanho (isso faz sentido porque há apenas um número finito de entradas possíveis de um determinado tamanho). Em ambos os casos, a complexidade do tempo é geralmente expressa como uma função do tamanho da entrada. Geralmente se concentra sobre o comportamento da complexidade quando o tamanho da entrada aumenta - isto é, o comportamento assintótico da complexidade. Portanto, a complexidade do tempo é comumente expressa usando a "Notação Big-O". As complexidades algorítmicas são classificadas de acordo com o tipo de função que descreve a ordem de crescimento do tempo de execução do algorítmo, podendo ser de crescimento constante = O(c), logarítmico = O(logn), linear = O(n), polinomial = O(n elevado a c), exponencial = O(c elevado a n), fatorial = O(n!), etc.

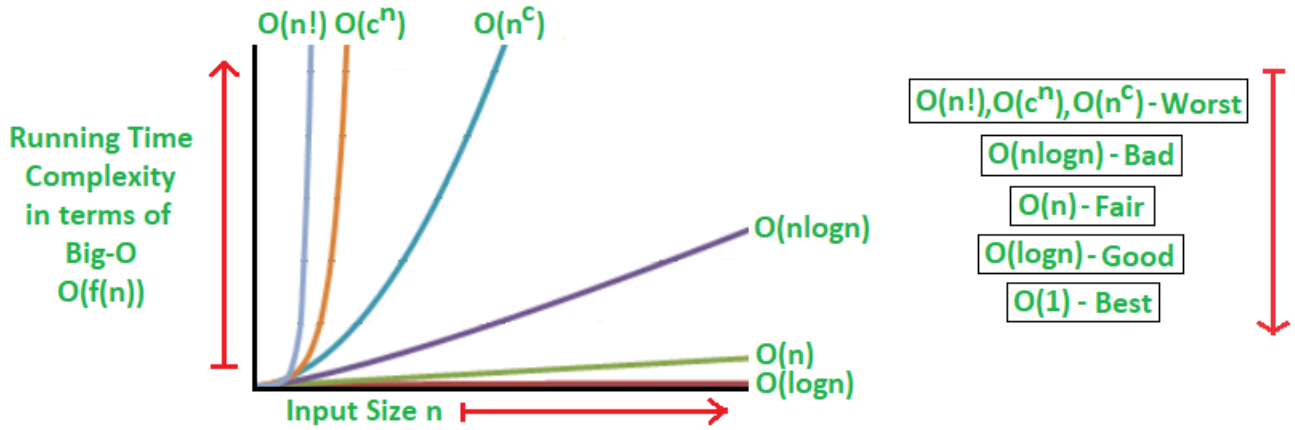

Figura 1: A Notação Big-O é utilizada para descrever e analisar o a eficiência com relação ao crescimento do tempo de execução dos algorítmos

Fonte: [1] (2020)

Algorítmos de Ordenação:

Na Ciência da Computação, um algorítmo de ordenação é um algoritmo que coloca os elementos de uma lista em uma determinada ordem. As ordens usadas com mais frequência são a ordem numérica e a ordem lexicográfica. É importante que tais algorítmos sejam pensados e implementados da maneira mais eficiente possível, pois eles servirão para otimizar a qualidade de outros algorítmos (como algorítmos de pesquisa e mesclagem) que exigem que os dados de entrada estejam em listas organizadas. A ordenação também é bastante útil para renderizar saída e informação legível (Que faça sentido do ponto de vista humano) para os usuários. Mais formalmente, a saída de qualquer algoritmo de ordenação deve satisfazer duas condições:

- A saída está em ordem crescente (O elemento anterior é menor ou igual ao elemento posterior);
- A saída é uma permutação (uma reordenação/arranjo), mas mantendo todos os elementos originais da entrada.

De acordo com [1]:
"Como muitos programas os usam como uma etapa intermediária, a classificação/ordenação é uma operação fundamental na Ciência da Computação. Como resultado, temos um grande número de bons algoritmos de classificação à nossa disposição. Qual algoritmo é melhor para um determinado aplicativo depende - entre outros fatores - do número de itens a serem classificados, até que ponto os itens já estão classificados de alguma forma, possíveis restrições sobre os valores dos itens, a arquitetura do computador e os tipos de dispositivos de armazenamentos a serem usados: memória principal, discos ou mesmo fitas."

Insertion Sort:

Um dos algorítmos de ordenação estudados nesse trabalho é conhecido pelo nome "Insertion Sort".

De acordo com [1], este algorítmo recebe como **Entrada (Input ou Instância)**: Uma sequência de n números (a1, a2, ..., an-1, an). E a **Saída (Output ou Resultado)** é: Uma permutação (Reordenação da lista) do tipo (a'1 ¡= a'2 ¡= ... ¡= a'n-' ¡= a'n). Vale ressaltar que os números que desejamos ordenar são conhecidos como chaves.
O Insertion Sort é um algoritmo eficiente para ordenar um pequeno número de elementos. A ordenação por inserção funciona da mesma forma que muitas pessoas ordenam uma mão ao jogar cartas. Começamos com a mão esquerda vazia e, em seguida, removemos uma carta de cada vez da mesa e o inserimos na posição correta na mão esquerda. Para encontrar a posição correta de uma carta, nós a comparamos com cada uma das cartas já na mão, da direita para a esquerda. Em todos os momentos, as cartas mantidas na mão esquerda são ordenadas, e essas cartas eram originalmente as cartas do topo da pilha na mesa.

Cálculo da complexidade de tempo do Insertion Sort:

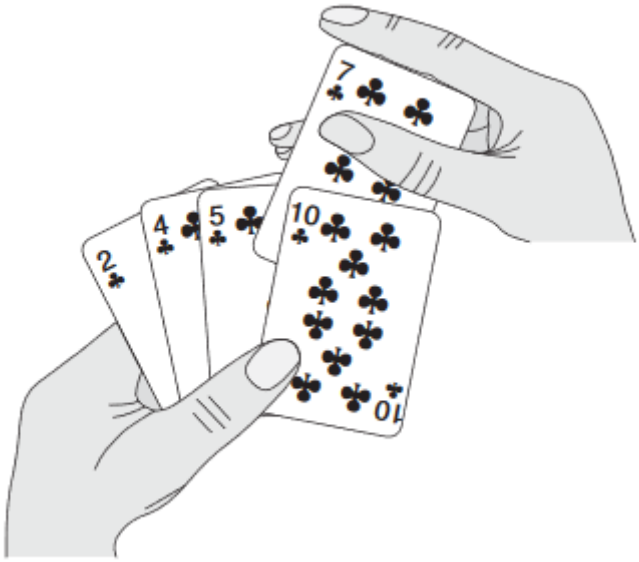

Figura 2: O funcionamento do Insertion Sort pode ser comparado ao padrão de comportamento humano de ordenar em ordem crescente as cartas em um jogo de baralho
Fonte: [1] (2020)

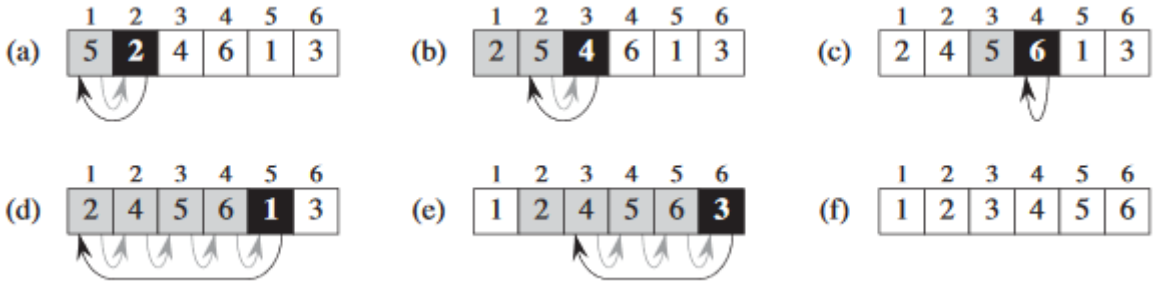

Figura 3: Fluxo de ordenação de uma lista de números pelo Insertion Sort
Fonte: [1] (2020)

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Figura 4: Pseudocódigo do Insertion Sort
Fonte: [1] (2020)

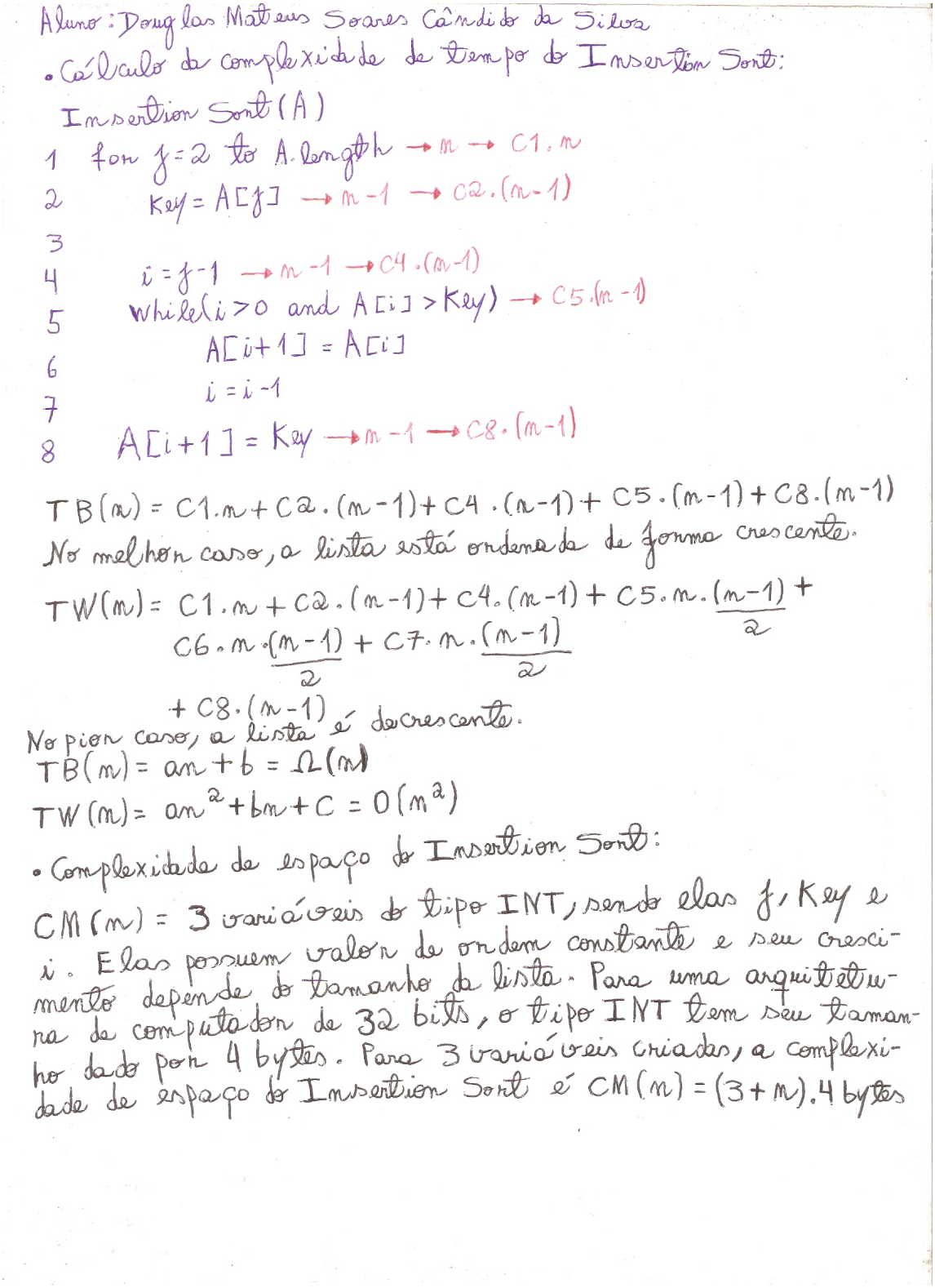

Figura 5: Cálculo da Complexidade de Tempo do Insertion Sort
Fonte: O próprio autor (2020)

Merge Sort:

O merge sort, ou ordenação por mistura, é um exemplo de algoritmo de ordenação por comparação que utiliza a estratégia conhecida como "dividir-para-conquistar". Sua ideia básica consiste em Dividir (o problema em vários subproblemas e resolver esses subproblemas através da recursividade) e Conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas). Como o algoritmo Merge Sort usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas. Ele divide a matriz de entrada em duas metades, chama a si mesmo (Recursivamente) para as duas metades e, em seguida, mescla as duas metades ordenadas. A função merge() é usada para mesclar duas metades. O array é dividido recursivamente em duas metades até que o tamanho se torne 1. Uma vez que o tamanho se torna 1, os processos de ordenação e mesclagem entram em ação e começam a mesclar os arrays novamente até que o array completo seja mesclado e ordenado.

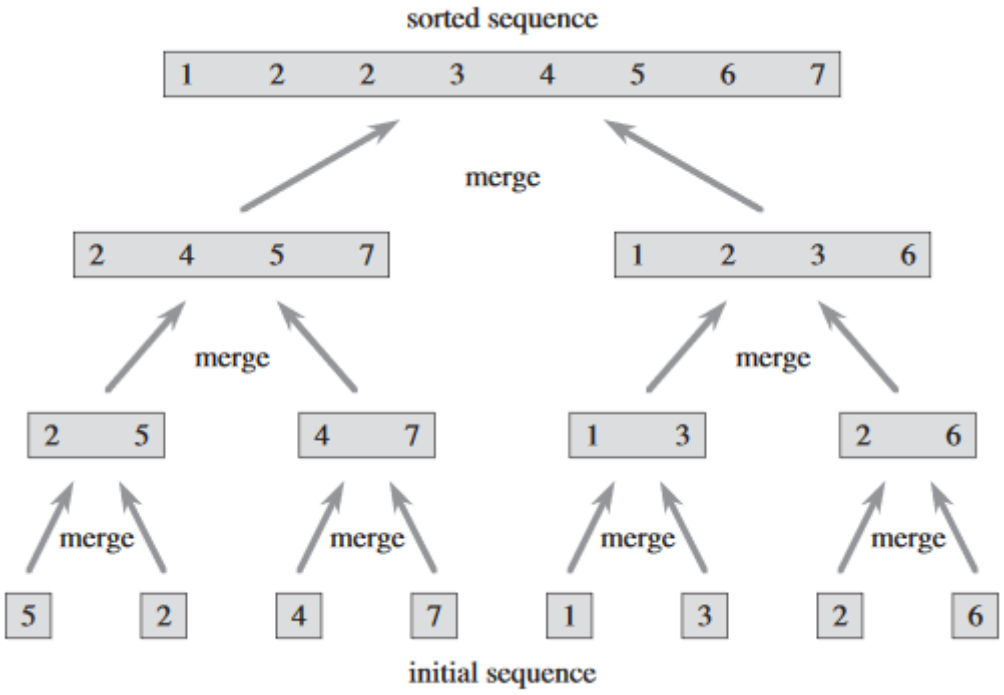

Figura 6: O funcionamento do Merge Sort se dá através da divisão da lista em lista menores e na ordenação destas sublistas para, logo após este processo de ordenação das listas menores, serem unificadas e formar uma lista ordenada contendo todos os elementos originais

Fonte: [1] (2020)

Entrada: Uma sequência de n números (a1, a2, ..., an-1, an). **Saída:** é: Uma permutação (Reordenação da lista) do tipo (a'1 ¡= a'2 ¡= ... ¡= a'n-' ¡= a'n).

Cálculo da complexidade de tempo do Merge Sort:

```
algorithm  merge-sort(A, s, e)
 1 if s < e then
 2     m ← ⌊(s + e)/2⌋
 3     merge-sort(A, s, m)
 4     merge-sort(A, m + 1, e)
 5     merge(A, s, m, e)

algorithm  merge(A, s, m, e)
 1 i ← s
 2 j ← m + 1
 3 for k ← 1 to e − s + 1 do
 4     if A[i] < A[j] and i ≤ m or j > e then
 5         B[k] ← A[i]
 6         i ← i + 1
 7     else
 8         B[k] ← A[j]
 9         j ← j + 1
10 for k ← 1 to e − s + 1 do
11     A[s + k] ← B[k]
```

Figura 7: Pseudocódigo do Merge Sort
Fonte: [1] (2020)

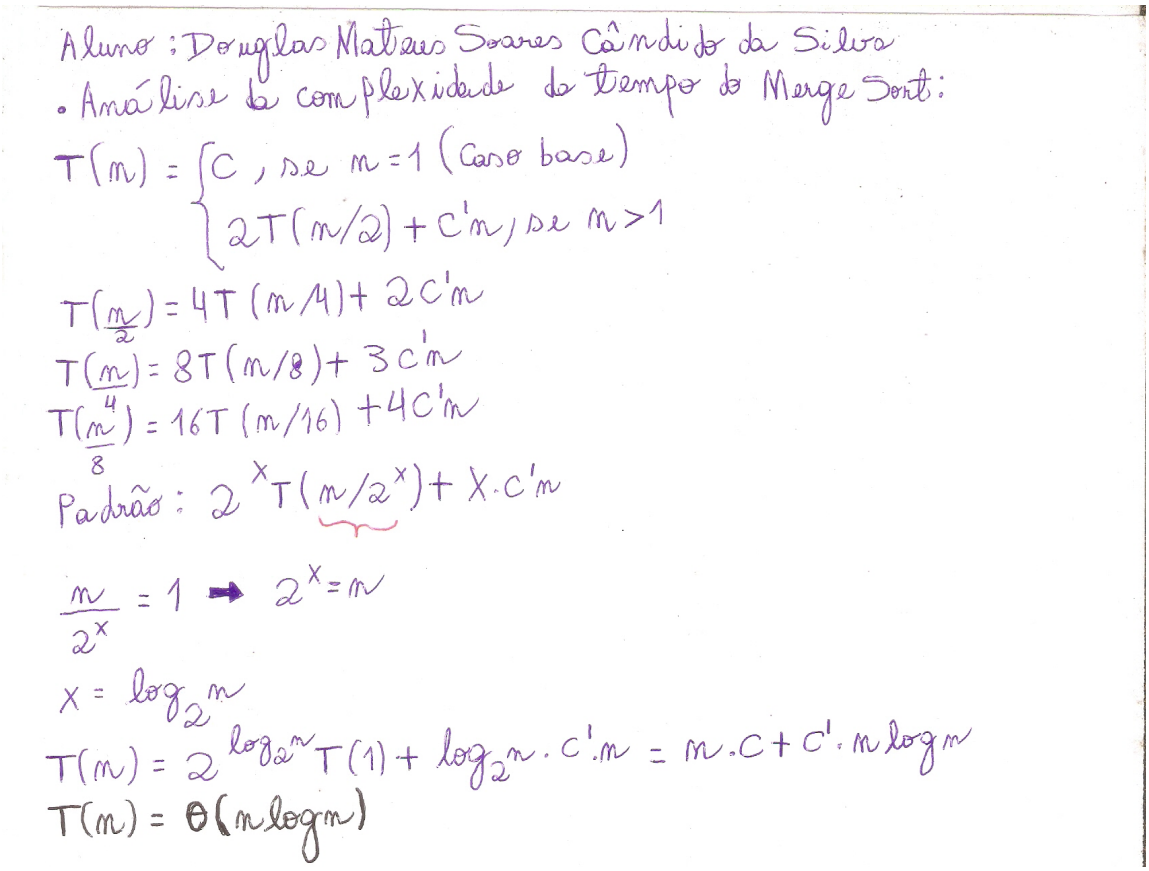

Figura 8: Cálculo da Complexidade de Tempo do Merge Sort
Fonte: O próprio autor (2020)

Quick Sort:

O Quick Sort adota a estratégia de divisão e conquista. Inicialmente, o Quick Sort divide o problema em problemas menores (Divide a lista em duas listas menores) e, em seguida, ele rearranja as chaves de modo que as chaves "menores"precedam as chaves "maiores". Em seguida o Quick Sort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada.

Os passos são:

- Selecionar: Escolher um elemento da lista, denominado pivô;
- Particionar: Rearranjar a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele. Ao fim do processo o pivô estará em sua posição final e haverá duas sub listas não ordenadas. Essa operação é denominada partição;
- Recursivamente ordene a sub lista dos elementos menores e a sub lista dos elementos maiores;
- Ao final de todo o processo, a lista estará ordenada.

A escolha do pivô e os passos do particionamento podem ser feitos de diferentes formas e a escolha de uma implementação específica afeta fortemente a performance do algoritmo.

Entrada: Uma sequência de n números (a1, a2, ..., an-1, an). **Saída:** é: Uma permutação (Reordenação da lista) do tipo (a'1 ¡= a'2 ¡= ... ¡= a'n-' ¡= a'n).

Cálculo da complexidade de tempo do Quick Sort:

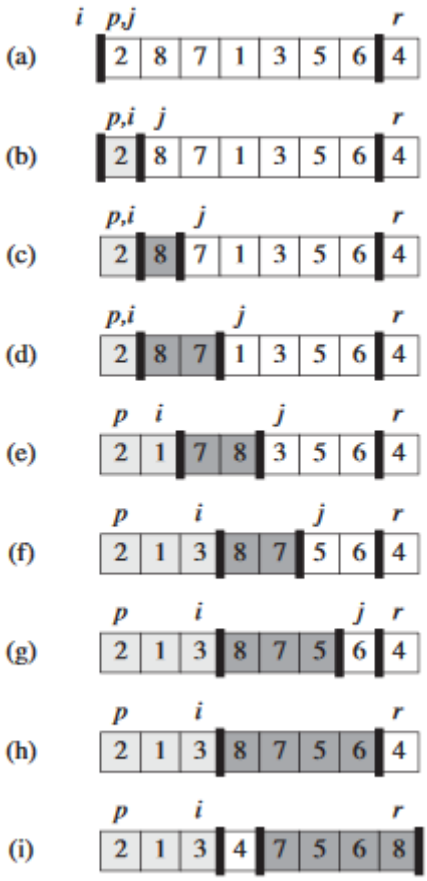

Figura 9: O funcionamento do Quick Sort se dá através da escolha de um pivô que servirá como base para a criação de duas sublistas, uma contendo todos os elementos menores ou iguais ao pivô e a outra contendo todos os elementos maiores ou iguais ao pivô. Essas duas sublistas serão ordenadas e, ao término de todo o processo, a lista estará completamente unificada e ordenada

Fonte: [1] (2020)

```
algorithm  quick-sort(A, s, e)
 1 if s < e then
 2     p ← partition(A, s, e)
 3     quick-sort(A, s, p − 1)
 4     quick-sort(A, p + 1, e)

algorithm  partition(A, s, e)
 1 k ← A[e]
 2 i ← s − 1
 3 for j ← s to e − 1 do
 4     if A[j] ≤ k then
 5         i ← i + 1
 6         A[i] ↔ A[j]
 7 A[i + 1] ↔ A[e]
 8 return  i + 1
```

Figura 10: Pseudocódigo do Quick Sort
Fonte: [1] (2020)

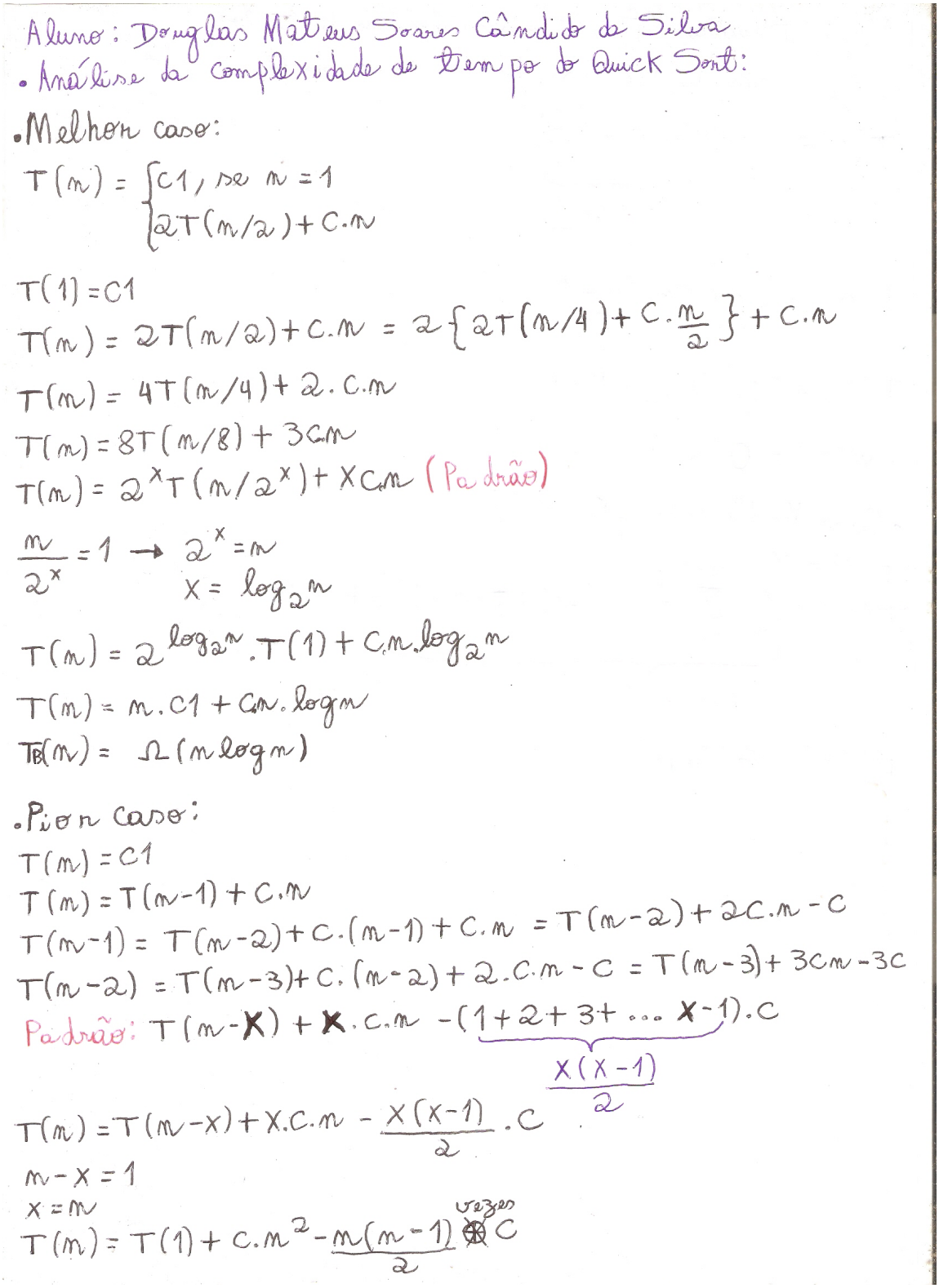

Figura 11: Cálculo da Complexidade de Tempo do Quick Sort (1)
Fonte: O próprio autor (2020)

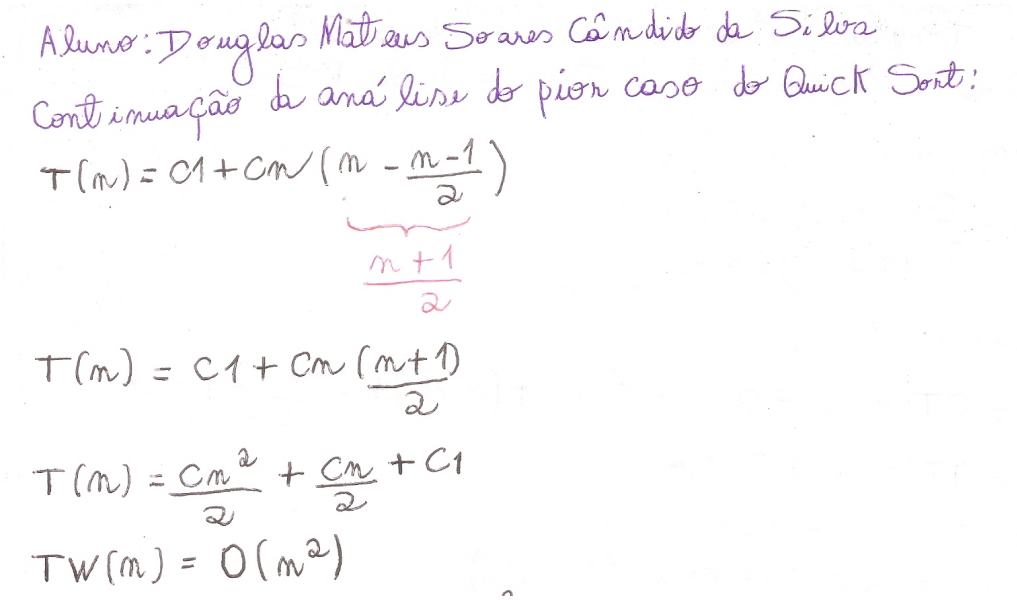

Figura 12: Cálculo da Complexidade de Tempo do Quick Sort (2)
Fonte: O próprio autor (2020)

3 Metodologia

Para analisar a eficiência dos três algorítmos de ordenação de modo quantitativo, é necessário medir os seus tempos de execução. Vale ressaltar que o valor do tempo de execução de um programa pode ser afetado positivamente ou negativamente pelas configurações da máquina em que seu cálculo foi realizado, isso ocorre devido ao processador gastar recursos processando múltiplas aplicações executando em paralelo.

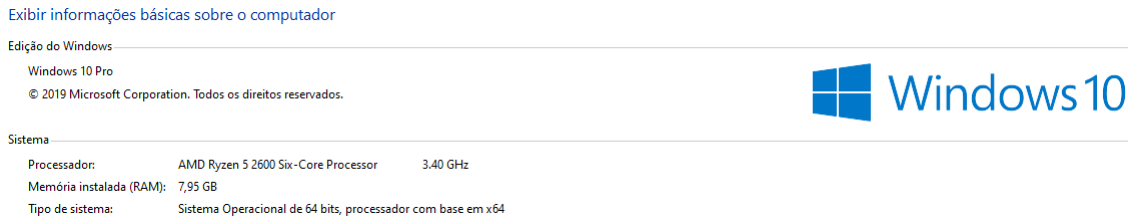

Figura 13: Especificações do computador usado nas medições dos tempos de execução
Fonte: O próprio autor (2020)

Foi necessário implementar os algorítmos com o uso de uma linguagem de programação para atingir tal objetivo. A linguagem de programação escolhida foi o C++, que é uma linguagem de programação moderna, de médio nível e bastante rápida.

Figura 14: Software desenvolvido
Fonte: O próprio autor (2020)

Os algorítmos de ordenação, na maioria de suas implementações, devem ordenar um array (Lista) informado como entrada (Instância). Para saber quanto tempo de execução cada função (Implementação do algorítmo de ordenação) levou para ordenar a lista foi necessário calcular a quantidade de clocks, ou Ciclos de CPU, (Quantidade de Cíclos que o processador gastou para executar o programa) durante a chamada da função e depois que ela parar de executar e, finalmente, calcular a diferença entre a quantidade de clocks do fim e do início. Essa diferença foi então dividida pelo valor de cíclos de clocks por segundo que o processador é capaz de executar. O resultado final é o tempo em segundos (s) que o determinado algorítmo de ordenação levou para ordenar a lista informada como instância.

Após isso, os tempos de execução são armazenados em um arquivo texto para então serem utilizados para gerar gráficos para fins de comparação.

4 Conclusão

Os gráficos abaixo ilustram as diferenças entre os tempos de execução e performance dos três algorítmos de ordenação estudados neste trabalho.

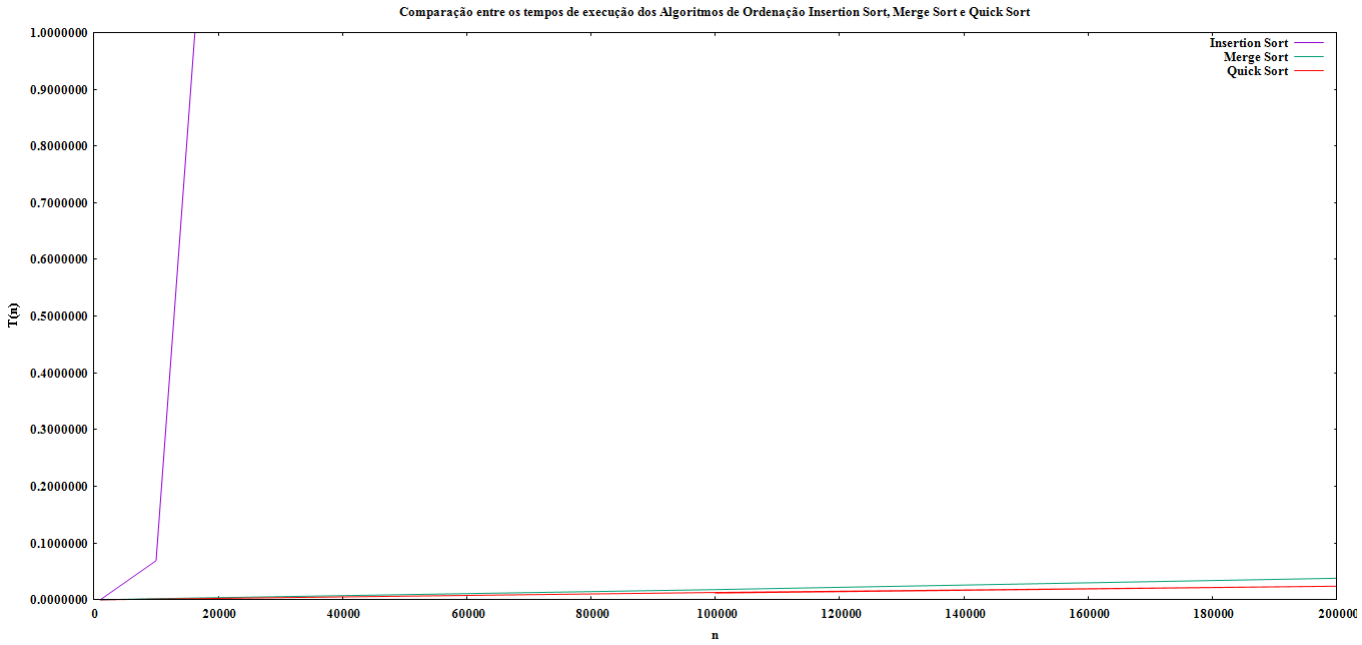

Figura 15: Gráfico de comparação do tempo de execução entre os algorítmos
Fonte: O próprio autor (2020)

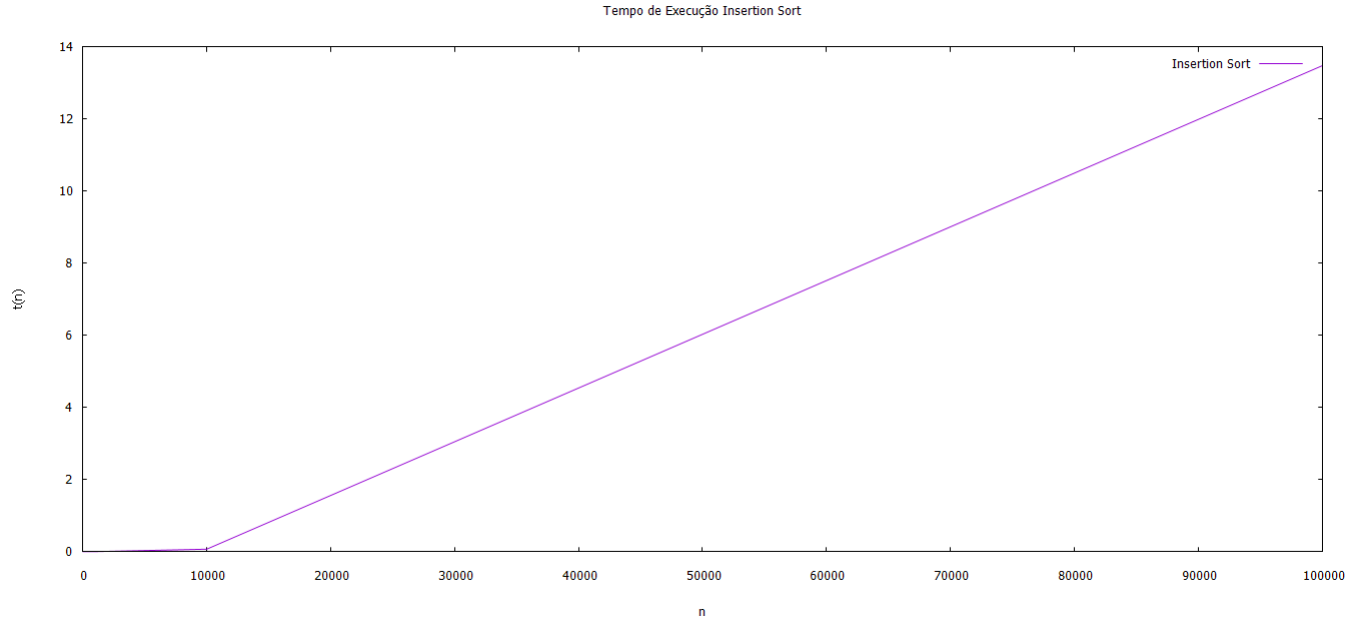

Figura 16: Gráfico do Insertion Sort
Fonte: O próprio autor (2020)

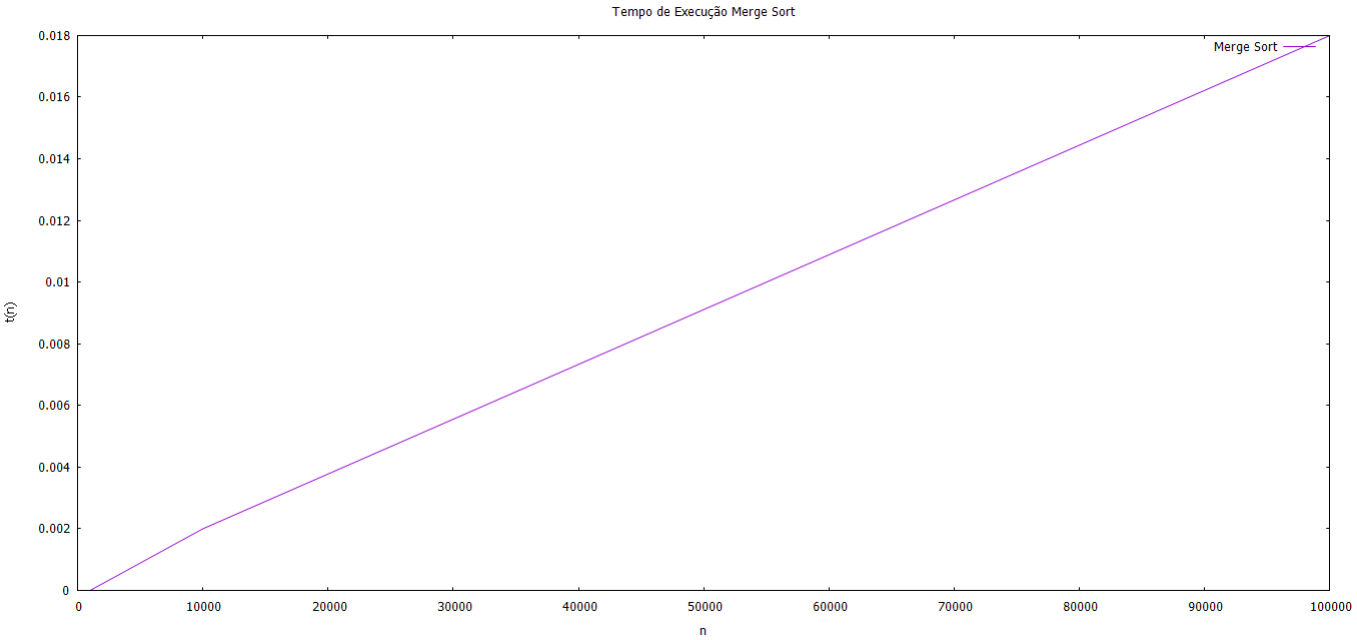

Figura 17: Gráfico do Merge Sort
Fonte: O próprio autor (2020)

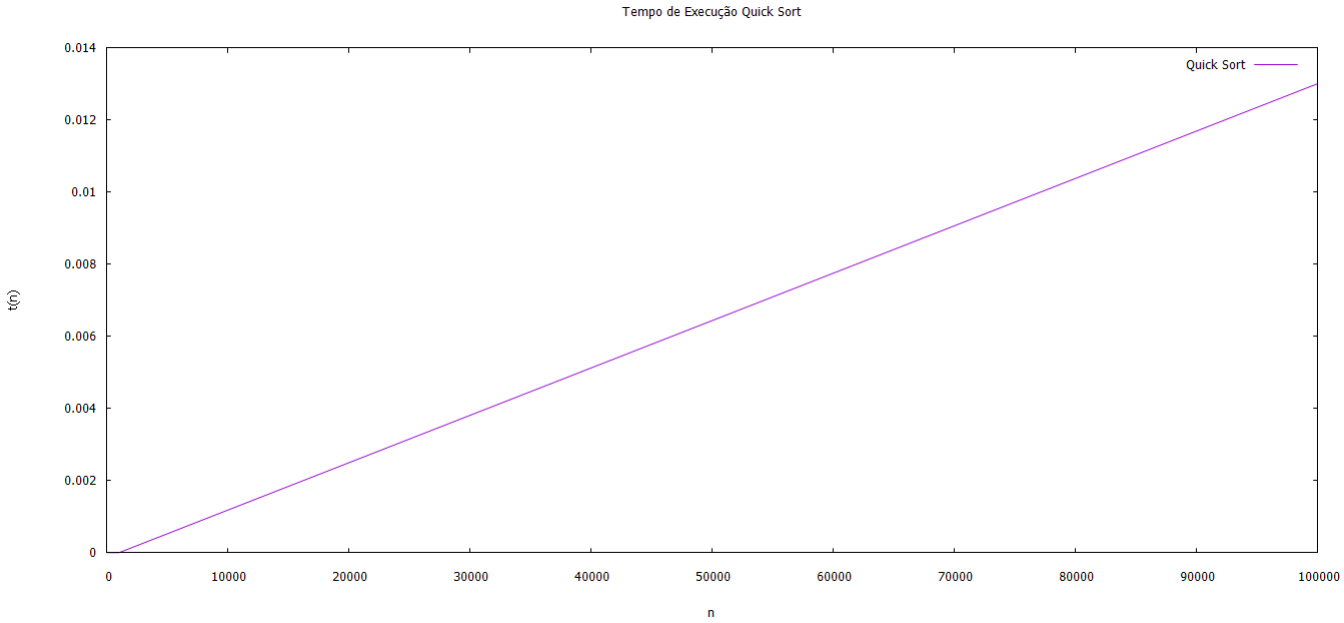

Figura 18: Gráfico do Quick Sort
Fonte: O próprio autor (2020)

A partir da análise dos gráficos percebeu-se que o algorítmo Insertion Sort é mais eficiente para arrays com tamanho inferior a 10000 números, ou seja, para listas menores; o Insertion Sort perde em velocidade para o Merge Sort e o Quick Sort quando a lista possui um tamanho muito grande. Descobriu-se também que o Merge Sort tem um tempo de execução muito lento para valores menores que 10000, tornando-o ineficiente para vetores menores e destacando sua ineficiência quando comparado ao Insertion Sort na ordenação de listas pequenas, porém ele tem um tempo de execução bastante rápido para arrays com uma quantidade de números superior a 10000, destacando sua grande eficiência na ordenação de listas grandes, mas sendo inferior ao Quick Sorte nesse sentido. Comprovou-se também que o Quick Sort é um pouco mais eficiente que o Merge Sort na ordenação de listas com tamanho inferior a 10000, porém ele perde em desempenho para o Insertion Sort nesse sentido. Por fim, comprovou-se que, para a ordenação de listas muito grandes, o Quick Sort é o melhor algorítmo de ordenação entre os três.

Uma possível sugestão para trabalhos futuros seria analisar a eficiência de outros algorítmos de ordenação, bem como analisar sua complexidade de espaço, pois este trabalho não conseguiu atender a essa demanda.

Referências

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[2] Anany Levitin. *Introduction to the design & analysis of algorithms*. Boston: Pearson,, 2012.

[3] Milton Santos. A natureza do espaço: técnica e tempo, razão e emoção. 2. reimpr. *São Paulo: Editora da Universidade de São Paulo*, 1, 2006.