Universidade Federal
de São João del-Rei

Bianca Rodrigues Guedes, João Lucas Vilas Boas Faria e João Victor Ottoni Garcia

TRABALHO PRÁTICO II

São João Del Rei - MG
2023

Bianca Rodrigues Guedes, João Lucas Vilas Boas Faria e João Victor Ottoni Garcia

TRABALHO PRÁTICO II

Trabalho de Algoritmos e Estrutura de Dados II relacionado a uma análise crítica sobre a execução dos métodos de ordenação em linguagem *C* ao receber entradas de diferentes tamanhos.

Professor: Rafael Sachetto Oliveira

São João Del Rei - MG
2023

SUMÁRIO

1. INTRODUÇÃO

Este trabalho tem como objetivo apresentar uma análise crítica sobre o funcionamento e tempo de execução dos métodos de ordenação em linguagem *C*, sendo eles: *Insertion Sort, Selection Sort, Merge Sort, Heap Sort* e *Shell Sort*. Os vetores de entrada foram 20, 500, 5.000, 10.000 e 200.000, e, para cada um, foram realizados dez testes para avaliação e comparação. Dessa forma, obteve-se a média do tempo de execução, número de comparações e trocas.

Serão apresentados, neste documento, os piores e melhores casos encontrados em cada teste desenvolvido no trabalho prático. Ademais, será tratada a eficácia de cada algoritmo quando a ordenação inicial do vetor de entrada estiver em ordem aleatória, crescente ou decrescente.

Foram utilizados registros e funções existentes nas bibliotecas disponíveis na linguagem *C*. Além disso, os códigos disponibilizados nos *slides* do professor foram usados como base para cumprir as especificações do trabalho. Foi desenvolvida uma função auxiliar para o menu de opções da *main* a fim de melhorar a visualização e entendimento ao colocar as entradas. Sendo assim, esta documentação é fundamental para a compreensão dos resultados obtidos ao observar o comportamento de cada método de organização.

2. CONCEITOS

2.1 ALGORITMOS DE ORDENAÇÃO EM LINGUAGEM *C*

Os algoritmos de ordenação nada mais são que movimentações lógicas para organizar certa estrutura linear. Esses algoritmos atuam sobre as *structs* de um arquivo, as quais contém uma chave para controlar a ordenação.

Portanto, será apresentada uma análise crítica da eficiência dos métodos de ordenação utilizados, que foram *Insertion Sort, Quick Sort, Selection Sort, Merge Sort, Heap Sort* e *Shellsort,* quando recebem diferentes tamanhos de entradas e sua forma de pré organização, sendo elas aleatoriamente, crescentemente ou decrescentemente.

3. IMPLEMENTAÇÃO

Nesta seção serão brevemente apresentados os registros utilizados na implementação do trabalho prático.

3.1 *STRUCTS* UTILIZADAS

3.1.1 ITEM_LEVE

```
1. typedef struct {
2. int chave;
3. }item_leve;
```

Esta *struct* recebe apenas um *int chave*, ou seja, a entrada a ser ordenada, sem peso algum. Logo, o registro apresentado será utilizado para análise dos algoritmos quando a entrada é um item leve.

3.1.2 ITEM_PESADO

```
1. typedef struct {
2.  int chave;
3.  char peso[50][50];
4. }item_pesado;
```

A segunda *struct* implementada recebe, assim como a primeira, um *int chave* com os elementos de entrada. No entanto, apresenta também um *char peso[50][50]*, um elemento com 50 *strings* que podem receber até 50 caracteres, possibilitando, então, um estudo do comportamento dos algoritmos quando trabalham com itens pesados.

4. ANÁLISE

4.1 ELEMENTOS LEVES

Esta seção analisará a eficiência e escalabilidade dos algoritmos ao receber elementos na *struct* apresentada inicialmente, de nome “item_leve”, e os resultados obtidos após a realização de dez testes.

4.1.1 ORDEM ALEATÓRIA

4.1.1.1 20 ELEMENTOS

Ao receber 20 elementos de forma aleatória, todos os algoritmos apresentaram um tempo médio de execução muito baixo, próximo de 0.0 segundos.

Quanto às médias de comparação e troca, e considerando o tamanho de entrada, o algoritmo que apresentou melhor resultado foi o *Shellsort*, com 81 comparações e 39 trocas em 0.000005 segundos. No entanto, o *QuickSort* também apresentou ótimo desempenho, com 88 comparações e 178 trocas, de forma tão rápida que a máquina contabilizou 0.000000 segundos.

O algoritmo com pior desempenho nos testes realizados foi o *Heap Sort*, com média de 158 comparações e 207 trocas.

4.1.1.2 500 ELEMENTOS

Ao receber 500 elementos de forma aleatória, os algoritmos também apresentaram um tempo médio de execução baixo. O *Heap Sort* foi o algoritmo com maior tempo, com 0.000762 segundos.

O algoritmo que apresentou mais eficiência foi o *Merge Sort*, com média de 3.853 comparações e 8.976 trocas em média de 0.000010 segundos. Nessa fase de testes, o *Quick Sort* também mostrou bons resultados, com média de 5.142 comparações e 8.962 trocas em tempo médio de 0.000005 segundos.

O pior caso, considerando a média de comparações, foi do algoritmo *Selection Sort*, com 123.750 comparações, 1.497 trocas e média de 0.000048 segundos.

4.1.1.3 5.000 ELEMENTOS

O tempo médio de execução foi maior ao receber 5.000 elementos, apesar de ainda ser muito baixo. O algoritmo mais lento nessa etapa foi o *Selection Sort,* obtendo resultado média de 0.022349 segundos.

O *Merge Sort* foi o algoritmo mais proveitoso, considerando a média de comparações, realizando 55.218 comparações e 123.613 trocas, com tempo médio de 0.000623 segundos. Posteriormente, o *Quick Sort* também foi eficiente nessa observação, tendo média 74.596 comparações e 124.590 trocas, em tempo médio de 0.000450 segundos.

Considerando a média de comparações e tempo, o pior algoritmo foi novamente o *Selection Sort*, tendo média 12.497.500 comparações, 14.997 trocas, em 0.022349 segundos.

4.1.1.4 10.000 ELEMENTOS

Com 10000 elementos de entrada, o tempo médio de execução também foi baixo, sendo o maior deles 0.085363 segundos, novamente do algoritmo *Selection Sort*.

O melhor caso, nessa fase de testes, foi obtido do *Merge Sort*, realizando em média 120.455 comparações e 267.232 trocas, em 0.001316 segundos. Outra vez, o *Quick Sort* apresentou bom desempenho, executando em média 166.146 comparações e 271.412 trocas, em 0.001035 segundos.

Ademais, o *Selection Sort* apresentou mais uma vez o pior resultado, em torno de 49.995.000 comparações e 29.997 trocas em 0.085363 segundos.

4.1.1.5 200.000 ELEMENTOS

Com entrada de 200.000 elementos, o *Selection Sort* foi o método mais devagar, com tempo médio de execução de 32.560372 segundos e o mais rápido foi o *Quick Sort*, obtendo 0.028062 segundos.

Levando em conta o número de comparações, o melhor caso foi do *Merge Sort*, tendo média de 3.272.734 comparações e 7.075.712 trocas, em média de 0.033802 segundos.

Como mencionado anteriormente, o *Selection Sort* foi o mais lento, apresentando o pior caso, com média de 19.999.900.000 comparações e 599.997 trocas.

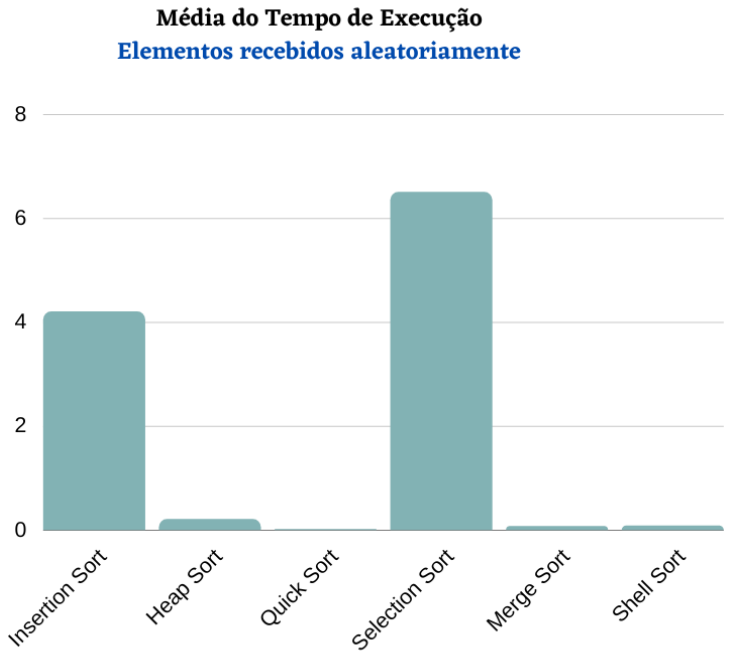
4.1.1.6 COMENTÁRIO

Conclui-se, então, que o algoritmo *Merge Sort* apresentou melhores resultados para ordenar itens de forma aleatória, independente do tamanho. O algoritmo tem complexidade O(n log n) e trabalha recursivamente dividindo os elementos em sublistas cada vez menores a serem ordenadas, e, por fim, une as sublistas em uma única lista ordenada.

Além disso, constata-se que o *Selection Sort* não trabalhou tão bem em comparação aos outros algoritmos recebendo elementos aleatoriamente. Isso se deve ao fato de que, independente do tamanho ou ordenação inicial da lista de itens, o algoritmo precisa percorrer toda a lista para encontrar o elemento mínimo, colocando-o em sua posição correta.

Abaixo segue um gráfico comparativo dos algoritmos, levando em conta a média do tempo de execução. Em geral, o *Quick Sort* foi o algoritmo mais rápido a ordenar os elementos pré dispostos aleatoriamente. Os dados apresentados estão denotados em segundos.

Figura 1. Média do tempo de execução dos algoritmos leves em ordem aleatória.



Fonte: elaboração própria.

4.1.2 ORDEM CRESCENTE

Nesta seção serão analisados os elementos recebidos em ordem crescente.

4.1.2.1 20 ELEMENTOS

Com a entrada de 20 elementos pré-ordenados crescentemente, o tempo médio de execução de todos os algoritmos foi muito baixo, com o pior tempo de 0.000096 segundos do *Heap Sort*.

O *Insertion Sort* foi o melhor algoritmo nesse caso, realizando em média 19 comparações e 0 trocas, levando 0.000003 segundos.

O pior caso, levando em conta a média do número de comparações, foi o *Quick Sort*, executando em média 192 comparações, 570 trocas e 0.000005 segundos.

4.1.2.2 500 ELEMENTOS

Nessa fase de testes, a média de execução também foi baixa, sendo o algoritmo mais demorado o *Heap Sort*, com tempo médio de 0.003325 segundos.

O melhor caso, mais uma vez, foi do algoritmo *Insertion Sort*, mediando 499 comparações e 0 trocas, em 0.000003 segundos.

O *Quick Sort* foi novamente o menos eficiente, fazendo, em média 124.800 comparações, 338.322 trocas, em 0.000781 segundos.

4.1.2.3 5.000 ELEMENTOS

O tempo de execução segue pequeno com a entrada de 5000 elementos, sendo o *Quick Sort* o método mais lento, com média de 0.053779 segundos.

Levando em consideração a média de correspondências, o *Insertion Sort* novamente apresentou bons resultados. Realizado cerca de 4999 comparações, 0 trocas, em 0.000012 segundos.

O pior caso foi o algoritmo *Quick Sort*, mediando 12.498.000 associações e 33.758.247 trocas.

4.1.2.4 10.000 ELEMENTOS

Ao receber 10.000 elementos, o pior tempo médio de execução foi do *Heap Sort*, com 0.36760 segundos, e o melhor foi do *Insertion Sort*, com 0.000009 segundos.

O algoritmo com mais eficiência foi, mais uma vez, o *Insertion Sort*, com 9999 comparações e 0 trocas, em média.

O *Quick Sort* segue sendo o algoritmo com o pior caso dessa fase de testes, executando em média 49.996.000 comparações e 135.016.497 trocas.

4.1.2.5 200.000 ELEMENTOS

Nessa fase de testes, aconteceu algo incomum comparado aos testes anteriores. O algoritmo *Quick Sort* apresentou estouro de pilha, e, assim, não foi possível obter os dados para análise de sua eficácia com tal quantia de elementos. No entanto, é notório que o algoritmo não é uma boa opção para entradas leves em ordem crescente, conforme observado nos estudos anteriores.

O melhor algoritmo continua sendo o *Insertion Sort*, realizando, em média, 199.999 comparações e 0 trocas, em 0.000546 segundos.

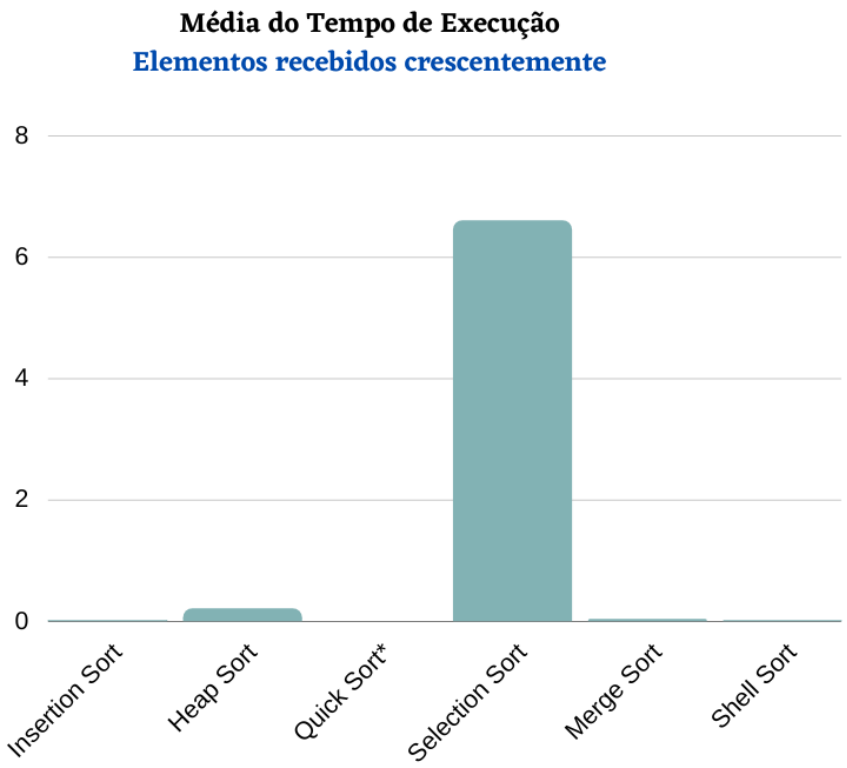
4.1.2.6 COMENTÁRIO

Portanto, observa-se que o *Insertion Sort* é o melhor algoritmo para listas ordenadas crescentemente, assim como para listas quase ordenadas. Isso acontece porque o algoritmo percorre a lista de elementos comparando cada elemento com seu anterior, e, como eles já estão em sua posição correta, funcionando competentemente.

Ademais, é evidente que o *Quick Sort* não é uma boa escolha para itens em ordem crescente. O algoritmo seleciona um elemento para ser o "pivô" e divide a lista em duas partes: uma com todos os elementos menores que o pivô e a outra com os maiores, repetindo esse processo recursivamente até que a ordenação esteja completa. No entanto, como a lista já está ordenada de modo crescente, o pivô escolhido inicialmente provavelmente será um dos maiores, resultando em uma sublista quase ou inteiramente vazia, e a outra com o restante dos elementos. Portanto, acontece uma quantidade excessiva e desnecessária de comparações e iterações, tornando-o um algoritmo ruim para receber elementos ordenados crescentemente.

Abaixo, gráfico de comparação entre a média do tempo de execução de cada algoritmo, em segundos. Nesse gráfico, o algoritmo *Quick Sort* não foi levado em conta devido ao erro de estouro de pilha.

Figura 2. Média do tempo de execução dos algoritmos leves em ordem crescente.



Fonte: elaboração própria.

4.1.3 ORDEM DECRESCENTE

Nesta seção serão estudados os algoritmos ao receber algoritmos pré-ordenados em ordem decrescente.

4.1.3.1 20 ELEMENTOS

Ao receber 20 itens decrescentemente, o tempo médio de execução de todos os algoritmos foi muito baixo, próximo de 0 segundos.

O melhor caso foi com o algoritmo *Merge Sort*, com média de 48 comparações e 176 trocas. O tempo médio de execução foi tão rápido que a máquina retornou 0.000000 segundos.

O *Insertion Sort* obteve pior resultado, com média de 209 comparações e 190 trocas. O algoritmo teve tempo médeio de 0.000002 segundos para executar.

4.1.3.2 500 ELEMENTOS

Com um conjunto de entrada de 500 elementos, o tempo de execução ainda foi baixo, sendo o mais devagar o que levou 0.001744 segundos, *Heap Sort*.

Conseguinte, o *Merge Sort* foi o mais eficiente, com média de 2.272 comparações, 8.976 trocas, em 0.000063 segundos.

Nos testes realizados, o pior caso foi com o *Insertion Sort*, efetuando 125.249 comparações e 124.750 trocas, em média de 0.000327 segundos.

4.1.3.3 5.000 ELEMENTOS

Ao receber 5000 elementos, o algoritmo mais lento foi o *Selection Sort*, que executou 0.024065 segundos.

O melhor caso, levando em consideração a média de comparações, foi do algoritmo *Heap Sort*, que realizou 7.854 comparações e 11.031 trocas, apesar de não ter sido o mais rápido, executou em 0.001744 segundos.

O *Selection Sort* apresentou os piores resultados, com médias de 12.497.500 comparações e 14.997 trocas.

4.1.3.4 10.000 ELEMENTOS

Com entrada de 10.000 elementos, o algoritmo mais lento foi o *Quick Sort*, com tempo médio de execução em torno de 0.166257 segundos.

O algoritmo mais eficiente foi o *Merge Sort*, o qual efetuou em média 69.008 comparações e 267.232 trocas em um tempo de 0.000735 segundos.

O *Insertion Sort* foi o pior método. Realizou em média 50.004.999 comparações e 49.995.000 trocas, em um tempo médio de 0.108529 segundos.

4.1.3.5 200.000 ELEMENTOS

Com a entrada de 200.000 elementos em ordem decrescente, houve novamente estouro de pilha com o algoritmo *Quick Sort* e, portanto, não foi possível analisar seus dados detalhadamente, mas sabe-se que ele não é a melhor escolha para a situação.

Nessa etapa de testes, o *Merge Sort* destacou-se como algoritmo mais adequado. Foram executadas, em média, 1.807.808 comparações e 7.075.712 trocas.

O método com piores resultados foi, mais uma vez, o *Insertion Sort*, que realizou em média 2.000.099.999 comparações e 19.999.900.000 trocas em tempo médio de execução próximo de 42 segundos.

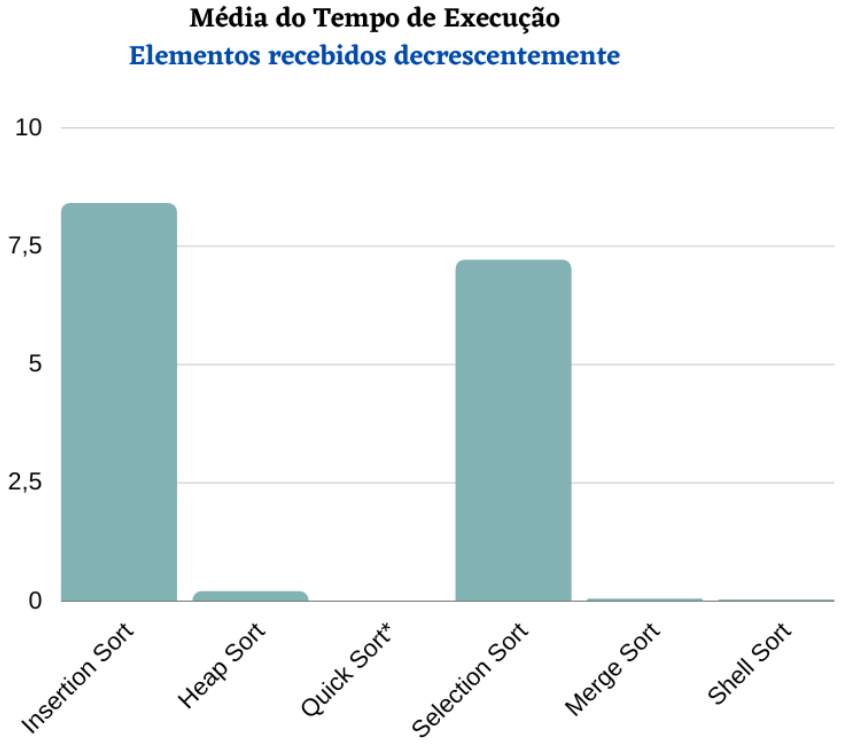
4.1.3.6 COMENTÁRIO

Dessa forma, o algoritmo que mais se destacou positivamente nessa fase de testes foi o *Merge Sort* que, como previamente explicado, trabalha recursivamente com sublistas, ordenando-as, unindo-as e, por fim, resultando em uma única lista ordenada.

Ademais, constata-se que o pior algoritmo encontrado foi o *Insertion Sort*. Esse algoritmo percorre a lista de elementos da esquerda para a direita, fazendo comparações de cada elemento com seu anterior e posicionando-o na ordem correta. Porém, quando os itens estão pré-ordenados decrescentemente, cada novo elemento a ser inserido sempre será menor que os elementos anteriores. Portanto, o elemento a ser inserido precisará percorrer toda a lista até encontrar a posição correta, resultando em um grande número de comparações e deslocamentos.

Abaixo, gráfico de comparação entre a média do tempo de execução em segundos de cada algoritmo recebendo itens em ordem decrescente. Neste gráfico, o algoritmo *Quick Sort* não foi levado em conta devido ao erro de estouro de pilha.

Figura 3. Média do tempo de execução dos algoritmos leves em ordem decrescente.



Fonte: elaboração própria.

4.2 ELEMENTOS PESADOS

Por fim, esta seção apresentará a eficiência e escalabilidade dos algoritmos ao receber elementos na *struct* de nome “item_pesado” e os resultados obtidos após a realização de dez testes.

4.2.1 ORDEM ALEATÓRIA

Os algoritmos seguintes serão analisados ao receber diferentes entradas em ordem aleatória.

4.2.1.1 20 ELEMENTOS

Ao testar 10 vezes as entradas de 20 elementos, a média do tempo de execução foi muito próxima de zero, sendo *Heap Sort* o algoritmo que apresentou resultado mais lento, com 0.000276 segundos.

Levando em consideração a média de comparações, o método *Merge Sort* apontou melhores resultados, realizando em média de 63 comparações e 176 trocas. O *Selection Sort* teve o pior caso nos testes, apontando média de 190 comparações e 57 trocas.

4.2.1.2 500 ELEMENTOS

Ao receber 500 elementos, o melhor resultado foi indicado, mais uma vez, pelo algoritmo *Merge Sort*, tendo média 3.853 comparações e 8.976 trocas, em 0.000077 segundos. Ademais, novamente houve contraste com o algoritmo *Selection Sort*, que realizou em média 124.750 comparações e 1.497 trocas em 0.000247 segundos.

4.2.1.3 5.000 ELEMENTOS

Com entrada de 5.000 elementos, o algoritmo *Merge Sort* foi o mais eficiente levando em conta o índice de comparações, que foi 55.229 e 123.616 trocas, em média de 0.000728 segundos. Outra vez, o algoritmo *Selection Sort* realizou um número muito grande de associações, 12.497.500, porém, com menor número de trocas, 14.997.

Nessa etapa de testes, o algoritmo mais rápido foi o *Quick Sort*, com tempo médio de execução em 0.000259 segundos. Em contraste, o mais lento foi o *Insertion Sort*, com tempo médio em 0.470879 segundos.

4.2.1.4 10.000 ELEMENTOS

O algoritmo que exibiu melhores resultados ao receber 10.000 elementos foi o *Merge Sort*, indicando média de 120.431 comparações, 267.232 trocas, em 0.002358 segundos. Os piores dados resultaram do *Selection Sort*, mediando 49.995.000 comparações e 29.997 trocas, em 0.151351 segundos.

O algoritmo mais rápido foi, mais uma vez, o *Quick Sort*, com tempo médio de 0.000766 segundos. O mais demorado foi o *Insertion Sort*, em 2.7 segundos.

4.2.1.5 200.000 ELEMENTOS

Nessa etapa, foi realizado apenas um teste com o algoritmo *Insertion Sort*, pois este levou mais de 30 minutos para ser executado, tornando-o inviável para realização de 10 testes. Além disso, o algoritmo fez 10.016.431.142 comparações e 10.016.231.143 trocas.

O melhor algoritmo encontrado foi o *Merge Sort*, que realizou, em média, 3.272.767 comparações e 7.075.712 trocas, em 0.260654 segundos. O pior algoritmo, considerando seu grande número médio de comparações, foi o *Selection Sort*, com 19.999.900.000 comparações e 599.991 trocas em 2 minutos.

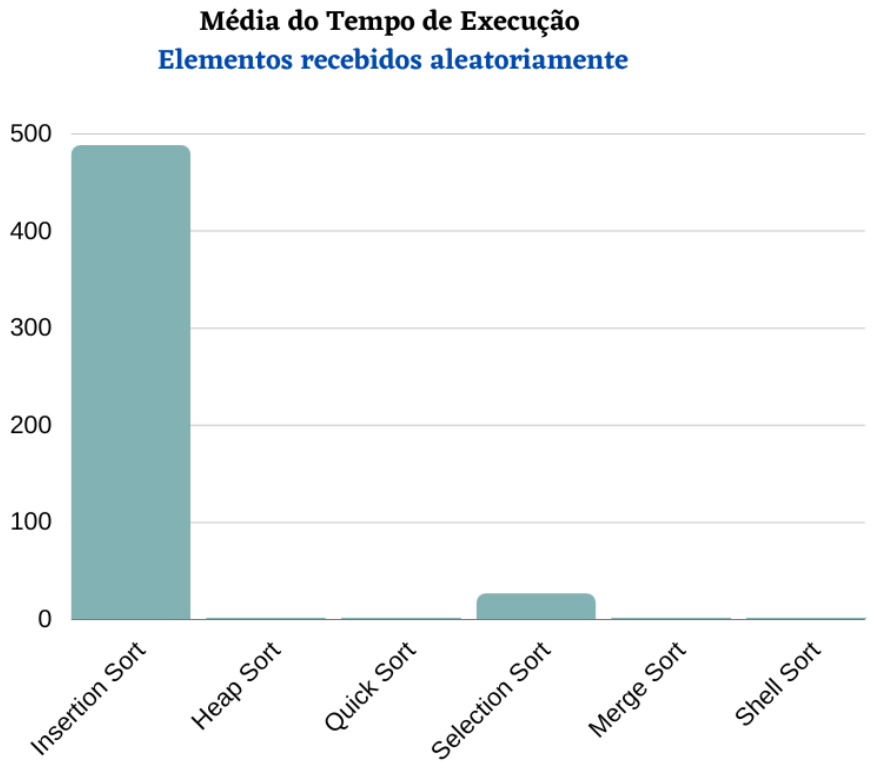
4.2.1.6 COMENTÁRIO

Conclui-se que o melhor algoritmo para executar itens pesados, inicialmente ordenados aleatoriamente, foi o *Merge Sort*. Esse algoritmo apresenta bons resultados também em seu pior caso, que tem complexidade O(n log n), ou seja, aponta resultados razoáveis até mesmo com a lista completamente desordenada. Em seguida, o *Quick Sort* apresentou bons resultados, próximos do *Merge*, mas com maior velocidade. Isso se deve ao fato de que o algoritmo, apesar de em seu pior caso apresentar complexidade $O(n^2)$, quando o pivô não é escolhido adequadamente, em média costuma ser muito eficiente, colocando o *Quick Sort* em uma boa posição para ordenar itens recebidos aleatoriamente.

O pior algoritmo encontrado foi o *Selection Sort*, pois esse precisa sempre percorrer a lista completa de elementos para achar o menor e colocá-lo em sua posição correta, o que ocasiona em um número exagerado de comparações.

Abaixo, gráfico para comparação do tempo médio de execução de cada algoritmo de ordenação, em segundos.

Figura 4. Média do tempo de execução dos algoritmos pesados em ordem aleatória.



Fonte: elaboração própria.

4.2.2 ORDEM CRESCENTE

Comparado à seção 4.1.2, a média de comparações e trocas são iguais, pois os itens foram recebidos em ordem crescente, mas dessa vez com peso maior. Dessa forma, nessa seção será analisada a média do tempo de execução de cada algoritmo.

4.2.2.1 20 ELEMENTOS

A média de tempo de execução dos algoritmos foi muito baixa, com *Insertion Sort, Heap Sort* e *Quick Sort* resultando em 0.00000 segundos. O algoritmo que apresentou maior tempo de execução foi o *Merge Sort*, em 0.000017 segundos.

4.2.2.2 500 ELEMENTOS

Com entrada de 500 elementos em ordem crescente, o algoritmo com melhor resultado foi o *Insertion Sort*, realizando tempo de 0.000002 segundos. O algoritmo mais lento foi o *Heap Sort*, tempo média de execução em 0.002722 segundos.

4.2.2.3 5.000 ELEMENTOS

Ao receber 5.000 elementos, o *Insertion Sort* destaca-se mais uma vez, com tempo de execução médio em 0.000032 segundos. O algoritmo que apontou pior lapso temporal de execução foi o *Quick Sort*, levando em média de 0.078059 segundos.

4.2.2.4 10.000 ELEMENTOS

Com 10.000 elementos pesados, o *Insertion Sort* executou a ordenação em menos tempo, 0.000120 segundos. Outra vez, o *Quick Sort* foi o mais lento, com 0.335713 segundos em média.

4.2.2.5 200.000 ELEMENTOS

O algoritmo *Insertion Sort* foi o mais rápido até mesmo ao receber 200.000 elementos, com média de tempo de execução em apenas 0.003155 segundos. O algoritmo mais lento encontrado foi o *Selection Sort*, apontando aproximadamente 2 minutos.

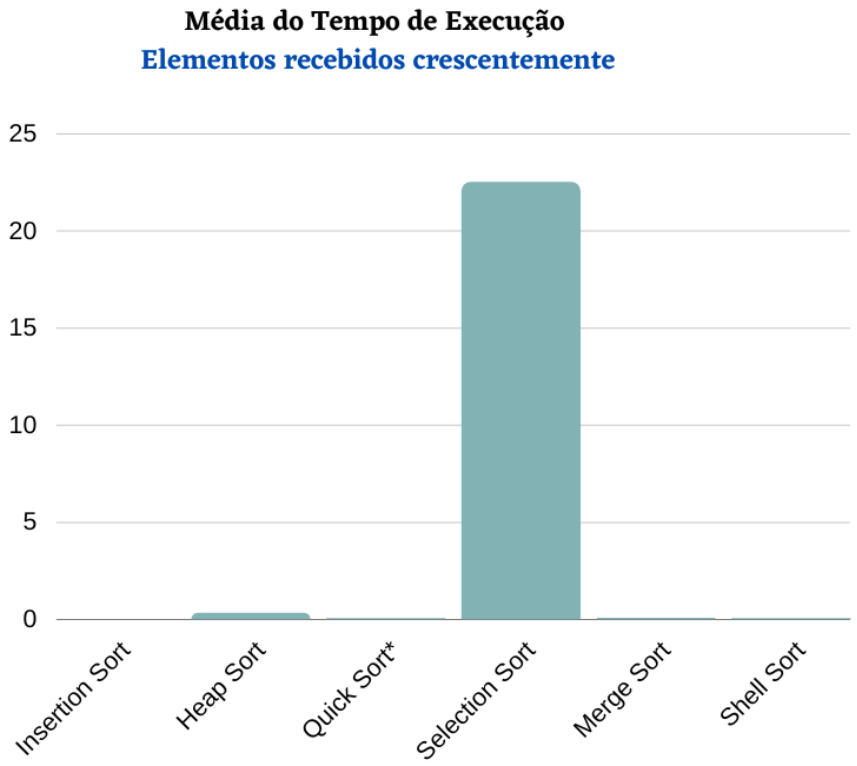
Nessa fase de testes, o algoritmo *Quick Sort* apresentou estouro de pilha, tornando impossível analisar o tempo médio de execução. Logo, é notório que não é o algoritmo adequado para a situação.

4.2.2.6 COMENTÁRIO

Portanto, é inegável a eficiência do algoritmo *Insertion Sort* para listas ordenadas ou quase ordenadas. Como os itens já se encontravam em ordem crescente, o número de comparações realizadas é baixo, tornando o algoritmo mais rápido e eficaz em comparação com os outros algoritmos de ordenação.

Abaixo, um gráfico que compara a média do tempo de execução de cada algoritmo em segundos. O algoritmo *Quick Sort* não foi levado em consideração, visto que deu erro por estouro de pilha, tornando impossível coletar seus dados nessa etapa de testes.

Figura 5. Média do tempo de execução dos algoritmos pesados em ordem crescente.



Fonte: elaboração própria.

4.2.3 ORDEM DECRESCENTE

Analisando a seção 4.1.3 com a média de comparações e trocas percebe-se que são iguais, pois os itens foram recebidos em ordem decrescente, mas dessa vez, com maior peso. Assim, nessa seção será analisada a média do tempo de execução de cada algoritmo.

4.2.3.1 20 ELEMENTOS

Ao receber 20 elementos em ordem decrescente, os algoritmos *Selection Sort, Shell Sort* e *Merge Sort* foram tão rápidos que a máquina registrou tempo médio de execução em 0.000000 segundos. O algoritmo mais lento foi o *Heap Sort*, com média de 0.000288 segundos.

4.2.3.2 500 ELEMENTOS

Outra vez, os algoritmos *Selection Sort, Shell Sort* e *Merge Sort* apontaram 0.000000 segundos na máquina. No entanto, o algoritmo *Quick Sort* também foi eficiente, com

0.000402 segundos. O algoritmo que levou mais tempo para ordenar foi o *Insertion Sort*, com tempo médio de 0.008270 segundos.

4.2.3.3 5.000 ELEMENTOS

O algoritmo apresentando maior velocidade ao receber 5.000 elementos foi o *Shell Sort*, com tempo médio em 0.000286 segundos. O mais lento foi o *Insertion Sort*, que executou em 1 segundo.

4.2.3.4 10.000 ELEMENTOS

Ao receber 10.000 elementos, o algoritmo *Shell Sort* apresentou melhor resultado, com tempo médio de 0.000672 segundos. O pior caso foi, novamente, o *Insertion Sort*, executando em média de 6 segundos.

4.2.3.5 200.000 ELEMENTOS

Assim como no caso anterior de 200.000 elementos em ordem decrescente, o algoritmo *Quick Sort* apresentou erro de estouro de pilha. Portanto, seus dados não foram considerados.

O algoritmo mais rápido analisado foi o *Shell Sort*, com média de tempo de execução em 0.052596 segundos. Em contraste, o algoritmo mais lento encontrado, *Insertion Sort*, levou 1 hora e 15 minutos para executar apenas um teste, tornando-o inviável para realização de 10 testes, mas é notório seu mau desempenho ao receber itens em ordem decrescente.

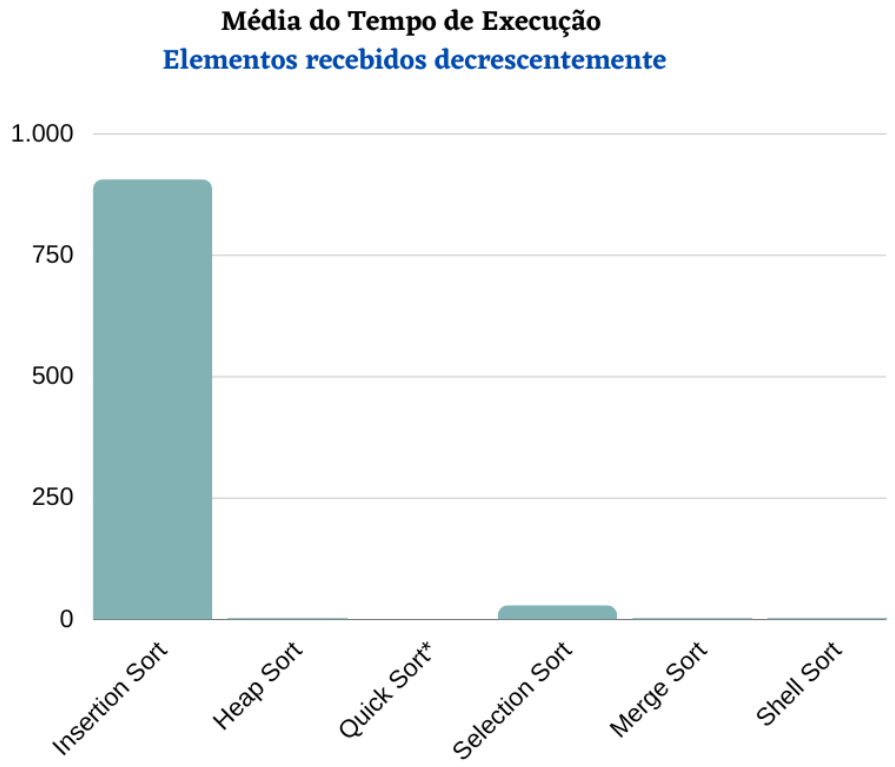
4.2.3.6 COMENTÁRIO

Sendo assim, o melhor algoritmo encontrado para ordenar listas pesadas em ordem decrescente foi o *Shell Sort*. Mesmo assim, é válido considerar que os algoritmos *Heap Sort* e *Merge Sort* também apresentaram ótimos resultados na fase de testes.

É evidente, também, que o algoritmo *Insertion Sort* não é uma boa opção quando se trata de ordenar listas decrescentes.

Abaixo, um gráfico representando a comparação entre o tempo de execução médio, em segundos, dos algoritmos de ordenação. Novamente, o algoritmo *Quick Sort* não está incluso por conta do erro de estouro de pilha.

Figura 6. Média do tempo de execução dos algoritmos pesados em ordem decrescente.



Fonte: elaboração própria.

5. CONCLUSÃO

Conclui-se, então, que cada algoritmo de ordenação apresenta diferentes comportamentos de acordo com os requisitos sugeridos. Dessa forma, é evidente conhecer a necessidade de cada algoritmo para saber quando apresenta seu melhor caso e, assim, desenvolver ordenações mais rápidas e eficientes.

6. BIBLIOGRAFIA

SACHETTO, Rafael. Algoritmos e Estruturas de Dados II - Ordenação. Slides. Universidade Federal de São João del Rei, São João del Rei. Disponível em: Portal Didático. Acesso em: Maio de 2023.

SORT VISUALIZER. Sort Visualizer, 2023. Página Inicial. Disponível em <https://www.sortvisualizer.com/>. Acesso em: Maio de 2023.