# Algoritmos de ordenação

**Rodrigo Duarte S. Luz[1], Matheus de Souza[2], Lucas Eduardo B. de Siqueira[3], Yasmin Karolyne A. Carvalho[4], Rodrigo de Lima Dias [5]**

Instituto de Matemática e Computação - Universidade Federal de Itajubá (UNIFEI)

Caixa Postal: 50 - CEP: 37500 903 - Itajubá – MG – Brasil

rodrigodsluz@gmail.com, matheusouza@live.com,lucasraz@hotmail.com,
yasminkarolyne@hotmail.com, rodrigolima@gmail.com

***Abstract.*** *This report consists of analyzing the performance of the following sorting algorithms: Insertion Sort, Selection Sort, Comb Sort, Bubble Sort, Merge Sort, Quick Sort, External Merge Sort and HeapSort, and making a usefulness and efficiency comparison between them.*

***Resumo.*** *Este relatório consiste em analisar o desempenho dos seguintes algoritmos de ordenação: Insertion Sort, Selection Sort, Comb Sort, Bubble Sort, Merge Sort, Quick Sort, Merge Sort Externo e HeapSort, e fazer uma comparação de utilidade e eficiência entre eles.*

## 1. Introdução

Os algoritmos de ordenação são artifícios para a resolução de tarefas utilizando computadores. Dependendo da aplicação, cada algoritmo possui alguma vantagem sobre os outros. O objetivo é compreender que existem várias formas de resolver uma mesma tarefa com um desses algoritmos, sendo que alguns são mais eficientes em certos casos, dependendo do tipo de tarefa a ser resolvida. A finalidade deste trabalho é apresentar qual a escolha do método de ordenação mais eficiente para cada caso, quando se tem diferentes tipos de conjuntos de dados a serem ordenados.

Para realizar esse trabalho foram executados sobre um mesmo hardware, algoritmos de ordenação com diferentes instâncias, ou seja, diferentes tamanhos de vetor, variando a disposição dos elementos. Foram considerados 5 tamanhos de vetor para cada um dos seguintes casos: Vetores com elementos distintos e distribuídos de forma aleatória; Vetores com elementos distintos já ordenados; Vetores com elementos distintos ordenados de forma reversa; Vetores onde os elementos estão distribuídos de forma aleatória, porém, cada elemento aparece ao menos em quantidade igual ou superior a 5% do tamanho do vetor. Os tamanhos escolhidos para o vetor foram: 50 mil, 80 mil, 120 mil, 180 mil e 270 mil elementos.

Os algoritmos foram comparados entre si considerando três métricas de desempenho: número de comparações, número de trocas e tempo gasto para completar a ordenação (dado em segundos). Cada configuração de teste foi executada ao menos 5 vezes, e o tempo expresso pela média dos valores obtidos.

Neste trabalho serão apresentados os algoritmos de ordenação Merge Sort Externo e o Heap Sort incluindo seus exemplos de funcionamento, complexidade e estabilidade. Além disso, também será apresentada a descrição do ambiente de teste com a especificação da máquina e do sistema operacional. Não obstante, também serão mostrados os resultados numéricos dos experimentos com tabelas e gráficos. Em relação aos gráficos foram gerados três para cada experimento (ordenação e busca). São eles: número de comparações x tamanho do vetor; número de trocas x tamanho do vetor e tempo x tamanho do vetor.

E ao final, terá uma  discussão sobre os resultados obtidos, e uma conclusão com indicações do uso dos algoritmos utilizados para aplicações onde eles teriam melhor desempenho.

## 2. Descrição dos algoritmos utilizados, incluindo exemplo de funcionamento, complexidade e estabilidade.

### 2.1.1 Heap Sort e seu funcionamento

O algoritmo heapsort é um algoritmo de ordenação generalista, e faz parte da família de algoritmos de ordenação por seleção. Foi desenvolvido em 1964 por Robert W. Floyd e J.W.J Williams.

A ordenação de heap é uma técnica baseada em comparação com base na estrutura de dados Binary Heap. É semelhante ao Selection Sort, onde primeiro encontramos o elemento mínimo e colocamos o elemento mínimo no início. Repetimos o mesmo processo para os elementos restantes.

Para explicar o que é Binary Heap vamos primeiro definir o que é uma árvore binária completa. Uma árvore binária completa é uma árvore binária em que todos os níveis, exceto possivelmente o último, são completamente preenchidos e todos nós estão o mais à esquerda possível.

Um heap binário é uma árvore binária completa em que os itens são armazenados em uma ordem especial de forma que o valor em um nó pai seja maior (ou menor) do que os valores em seus dois nós filhos. O primeiro é chamado de heap máximo e o último é chamado de heap mínimo. O heap pode ser representado por uma árvore binária ou matriz.

Como um heap binário é uma árvore binária completa, ele pode ser facilmente representado como uma matriz e a representação baseada em matriz é eficiente em termos de espaço. Se o nó pai for armazenado no índice I, o filho esquerdo pode ser calculado por 2 * I + 1 e o filho direito por 2 * I + 2 (assumindo que a indexação comece em 0).

O Heap Sort utiliza uma estrutura de dados chamada heap, para ordenar os elementos à medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada, lembrando-se sempre de manter a propriedade de max-heap.

A heap pode ser representada como uma árvore (uma árvore binária com propriedades especiais) ou como um vetor. Para uma ordenação decrescente, deve ser construída uma heap mínima (o menor elemento fica na raiz). Para uma ordenação crescente, deve ser construído uma heap máxima (o maior elemento fica na raiz).

Neste trabalho foi utilizado o Heap Sort para ordenar os vetores em ordem crescente, e para isso ocorreram as seguidas essas etapas:

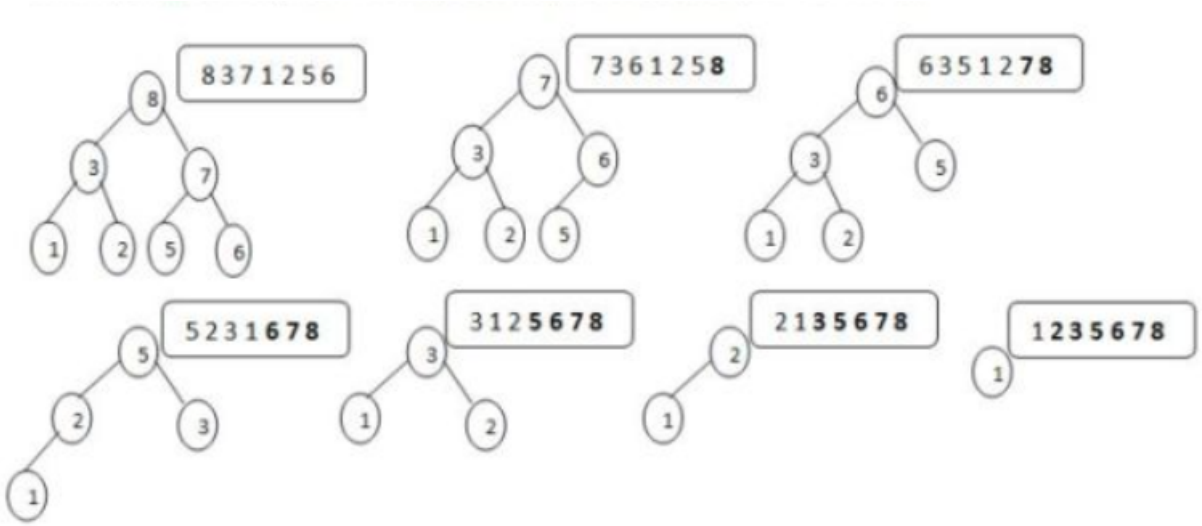
1. Construa um heap máximo a partir dos dados de entrada.

2. Nesse ponto, o maior item é armazenado na raiz do heap. Substitua-o pelo último item do heap seguido pela redução do tamanho do heap em 1. Finalmente, monte a raiz da árvore.

3. Repita a etapa 2 enquanto o tamanho do heap for maior que 1.

Como exemplo, aqui temos um vetor que foi ordenado utilizando essas etapas:

2 3 7 1 8 5 6



### 2.1.2 Complexidade

A complexidade de tempo de heapify é O (Logn). A complexidade de tempo de para criar e construir o heap é O (n) e a complexidade de tempo geral de Heapsort é O (nLogn).

### 2.1.3 Estabilidade

O heapsort não é um algoritmo de ordenação estável. Porém, é possível adaptar a estrutura a ser ordenada de forma a tornar a ordenação estável. Cada elemento da estrutura adaptada deve ficar no formato de um par (elemento original, índice original). Assim, caso dois elementos sejam iguais, o desempate ocorrerá pelo índice na estrutura original.

### 2.1.4. Características

- Heap Sort é um algoritmo in-place, o que significa que ele não precisa de vetores auxiliares para manipular os dados.
- Ele é muito eficiente para ordenar um grande número de elementos.
- O Heap Sort tem usos limitados porque o Quicksort e o Mergesort são melhores na prática. Porém, ele ainda é continuamente usado em outras aplicações, como filas prioritárias, por exemplo.

## 2.2. Mergesort Externo

### 2.2.1 Mergesort

O **mergesort**, ou ordenação por mistura, é um exemplo de algoritmo de ordenação por comparação do tipo 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.

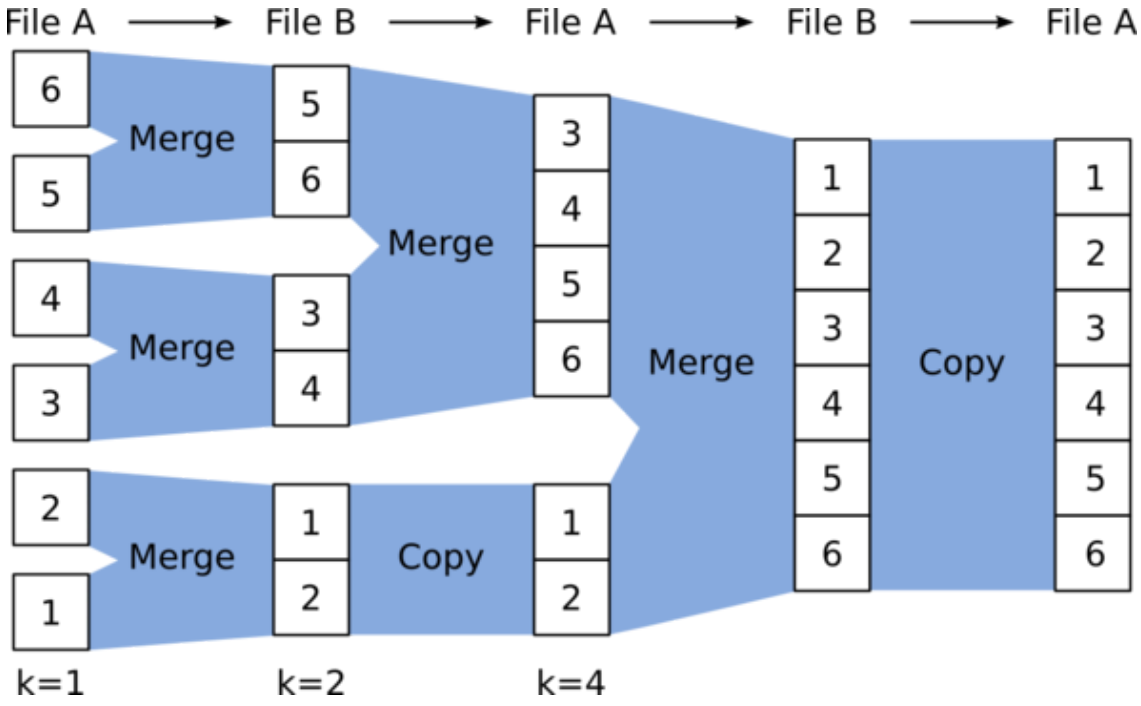
### 2.2.2. ordenação Externa

É um dos algoritmos de ordenação que podem lidar com grandes quantidades de dados. O mergesort externo é necessário quando os dados que estão sendo ordenados não cabem na memória principal. O mergesort externo normalmente usa uma estratégia híbrida de ordenação e mesclagem. Na fase de ordenação, blocos de dados pequenos o suficiente para caber na memória

principal são lidos, ordenados e gravados em um arquivo temporário. Na fase de mesclagem, os subarquivos ordenados são combinados em um único arquivo maior.

Ele funciona da seguinte forma: Primeiro dividimos o arquivo em partes de forma que o tamanho de uma execução seja pequeno o suficiente para caber na memória principal. Então executa-se cada arquivo na memória principal, utilizando o algoritmo de merge sort visto em aula. Para finalizar , junte as partições resultantes em execuções sucessivamente maiores, até que o arquivo seja ordenado.

Abaixo, temos uma imagem para demonstrar cada parte sendo ordenada e se juntando:




### 2.2.3 Eficiência

A eficiência de uma ordenação externa é medida em termos da quantidade total do tempo necessário para que os dados sejam lidos, ordenados e gravados na área de trabalho; isto corresponde à fase inicial do processo de ordenação.

Na fase seguinte as séries ordenadas são lidas e intercaladas duas a duas, três a três, etc. O número de séries intercaladas em cada passo corresponde ao número de caminhos. Quanto maior for o número de caminhos, mais complexo será o algoritmo de intercalação (merge). Como este processo é utilizado várias vezes para conseguir juntar todas as séries, em geral opta-se por trabalhar com 2 ou 3 caminhos.

Os fatores que influem na eficiência de um algoritmo de ordenação externa são os seguintes:

- Número de séries produzidas;
- Tamanho de cada série e se são do mesmo tamanho;
- Forma de distribuição das séries pelos arquivos de trabalho e
- Características dos blocos e das memórias intermediárias utilizadas na fase de ordenação

A estimativa do número de séries utilizadas, bem como seu comprimento, é função do número de registros do arquivo; isto determina o número de passadas durante a fase de ordenação. É importante que esteja bem definida a sequência na qual as séries devem ser distribuídas pelos arquivos de entrada para a fase de intercalação.

Os tipos básicos de algoritmos para distribuir as séries são equilibradas, onde são colocadas um número igual de séries iniciais 58 em cada arquivo e desequilibradas em que as séries são dispostas nos arquivos de modo a conseguir um melhor nível de intercalação.

### 2.2.4 Desempenho

Merge sort externo pode ser analisado no modelo de memória externo. Neste caso a cache, ou memória externa com o tamanho M e uma quantia limitada de memória são divididas em blocos de tamanho B, e a cada funcionamento o tempo do algoritmo é determinado pelo número de memória transferida entre memória interna e externa. Possui um um tempo de execução de:

$$O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right).$$

### 2.2.5 Estabilidade:

Por se tratar do uso do merge sort, o merge sort externo pode ser considerado um algoritmo estável, contudo ele ainda pode ser melhorado, utilizando:

- Os algoritmos para ordenação externa devem reduzir o número de passadas sobre o arquivo
- Uma boa medida de complexidade é o número de vezes que um item é lido ou escrito da/na memória externa
- Bons métodos geralmente envolvem, no total, menos do que 10 passadas sobre o arquivo.

## 3. Descrição do ambiente de teste:

Para realizarmos os testes foi utilizado um notebook com o sistema operacional Windows 10 Enterprise, com um processador Intel Core i7 da 7ª geração, e uma memória RAM com 16GB, e SSD com 256MB. O código foi compilado e executado no Code Blocks com o compilador GCC.

## 4. Resultados numéricos dos experimentos

| Quantidade de comparações dos vetores com elementos distintos e distruibuidos de forma aleatória | | | | | |
|---|---|---|---|---|---|
| **Algoritmo** | **270 000 elementos** | **180 000 elementos** | **120 000 elementos** | **80 000 elementos** | **50 000 elementos** |
| Insertion Sort | 267,635,853.00 | 269,994,704.00 | 269,215,993.00 | 264,540,626.00 | 262,049,887.00 |
| Comb sort | 8,040,068.00 | 8,400,051.00 | 5,120,058.00 | 3,413,378.00 | 1,983,397.00 |
| Bubble Sort | 36,449,865,000.00 | 16,199,910,000.00 | 7,199,940,000.00 | 3,199,960,000.00 | 1,249,975,000.00 |
| Selection Sort | 36,449,865,000.00 | 16,199,910,000.00 | 7,199,940,000.00 | 3,199,960,000.00 | 1,249,975,000.00 |
| Merge Sort | 2,622,112.00 | 1,742,879.00 | 1,202,265.00 | 842,354.00 | 586,079.00 |
| QuickSort | 385,362.00 | 339,690.00 | 250,792.00 | 196,446.00 | 175,346.00 |
| Merge Sort Externo | 2,235,969.00 | 1,485,963.00 | 1,012,552.00 | 691,065.00 | 470,695.00 |
| Heap Sort | 9,886,406.00 | 6,397,656.00 | 4,102,778.00 | 2,635,816.00 | 1,562,580.00 |

| Quantidade de comparações dos vetores com elementos distintos já ordenados. | | | | | |
|---|---|---|---|---|---|
| **Algoritmo** | **270 000 elementos** | **180 000 elementos** | **120 000 elementos** | **80 000 elementos** | **50 000 elementos** |
| Insertion Sort | 269,999.00 | 179,999.00 | 119,999.00 | 79,999.00 | 49,999.00 |
| Comb sort | 36,449,865,000.00 | 6,960,095.00 | 4,400,085.00 | 2,853,413.00 | 1,683,424.00 |
| Bubble Sort | 36,449,865,000.00 | 16,199,910,000.00 | 7,199,940,000.00 | 3,199,960,000.00 | 1,249,975,000.00 |
| Selection Sort | 36,449,865,000.00 | 16,199,910,000.00 | 7,199,940,000.00 | 3,199,960,000.00 | 1,249,975,000.00 |
| Merge Sort | 2,417,376.00 | 1,537,360.00 | 998,016.00 | 382,512.00 | 382,512.00 |
| QuickSort | 277,856.00 | 228,928.00 | 131,070.00 | 94,464.00 | 65,534.00 |
| Merge Sort Externo | 2,050,680.00 | 1,305,920.00 | 843,040.00 | 523,520.00 | 320,040.00 |
| Heap Sort | 9,908,138.00 | 6,428,162.00 | 4,137,042.00 | 2,666,302.00 | 1,596,610.00 |

| Quantidade de comparações dos vetores com elementos distintos ordenados de forma reversa. | | | | | |
|---|---|---|---|---|---|
| **Algoritmo** | **270 000 elementos** | **180 000 elementos** | **120 000 elementos** | **80 000 elementos** | **50 000 elementos** |
| Insertion Sort | 36,449,865,000.00 | 16,199,910,000.00 | 7,199,940,000.00 | 3,199,960,000.00 | 1,249,975,000.00 |
| Comb sort | 11,250,070.00 | 7,140,085.00 | 4,520,070.00 | 2,933,397.00 | 1,733,410.00 |
| Bubble Sort | 36,449,865,000.00 | 16,199,910,000.00 | 7,199,940,000.00 | 3,199,960,000.00 | 1,249,975,000.00 |
| Selection Sort | 36,449,865,000.00 | 16,199,910,000.00 | 7,199,940,000.00 | 3,199,960,000.00 | 1,249,975,000.00 |
| Merge Sort | 2,458,336.00 | 1,620,496.00 | 1,030,912.00 | 672,064.00 | 401,952.00 |
| QuickSort | 547,854.00 | 408,926.00 | 251,068.00 | 174,462.00 | 115,532.00 |
| Merge Sort Externo | 1,941,640.00 | 1,246,400.00 | 793,120.00 | 514,560.00 | 298,040.00 |
| Heap Sort | 9,130,778.00 | 5,880,182.00 | 3,782,110.00 | 2,428,150.00 | 1,447,786.00 |

| Qtd de comparações de vetores com valores distribuídos aleatoriamente, e cada elemento com qtd >= a 5% do tamanho | | | | | |
|---|---|---|---|---|---|
| **Algoritmo** | **270 000 elementos** | **180 000 elementos** | **120 000 elementos** | **80 000 elementos** | **50 000 elementos** |
| Insertion Sort | 14,579,507,422.00 | 6,479,521,059.00 | 2,881,382,611.00 | 1,280,941,665.00 | 499,743,396.00 |
| Comb sort | 11,250,070.00 | 6,960,087.00 | 4,520,070.00 | 2,933,397.00 | 1,683,412.00 |
| Bubble Sort | 36,449,865,000.00 | 16,199,910,000.00 | 7,199,940,000.00 | 3,199,960,000.00 | 1,249,975,000.00 |
| Selection Sort | 36,449,865,000.00 | 16,199,910,000.00 | 7,199,940,000.00 | 3,199,960,000.00 | 1,249,975,000.00 |
| Merge Sort | 4,182,454.00 | 2,688,080.00 | 1,731,812.00 | 1,108,310.00 | 662,571.00 |
| QuickSort | 3,481,322.00 | 2,193,260.00 | 1,433,312.00 | 901,392.00 | 542,884.00 |
| Merge Sort Externo | 3,401,700.00 | 2,162,318.00 | 1,384,885.00 | 874,721.00 | 517,305.00 |
| Heap Sort | 7,658,764.00 | 4,963,716.00 | 3,184,684.00 | 2,063,002.00 | 1,228,000.00 |

| Quantidade de trocas dos vetores com elementos distintos e distruibuidos de forma aleatória | | | | | |
|---|---|---|---|---|---|
| **Algoritmo** | **270 000 elementos** | **180 000 elementos** | **120 000 elementos** | **80 000 elementos** | **50 000 elementos** |
| Insertion Sort | 267,365,866.00 | 269,814,718.00 | 269,096,004.00 | 264,460,640.00 | 261,999,899.00 |
| Comb sort | 240,379.00 | 240,515.00 | 242,600.00 | 243,227.00 | 238,920.00 |
| Bubble Sort | 267,973,858.00 | 267,477,909.00 | 267,899,308.00 | 268,017,208.00 | 258,916,965.00 |
| Selection Sort | 269,999.00 | 179,999.00 | 119,999.00 | 79,999.00 | 49,999.00 |
| Merge Sort | 9,751,424.00 | 6,315,712.00 | 4,057,856.00 | 2,617,856.00 | 1,568,928.00 |
| QuickSort | 187,394.00 | 163,990.00 | 119,970.00 | 92,547.00 | 81,785.00 |
| Merge Sort Externo | 7,984,640.00 | 5,104,640.00 | 3,272,320.00 | 2,076,160.00 | 1,236,160.00 |
| Heap Sort | 4,808,203.00 | 3,108,828.00 | 1,991,389.00 | 1,277,908.00 | 756,290.00 |

| Quantidade de trocas dos vetores com elementos distintos e ordenados | | | | | |
|---|---|---|---|---|---|
| **Algoritmo** | **270 000 elementos** | **180 000 elementos** | **120 000 elementos** | **80 000 elementos** | **50 000 elementos** |
| Insertion Sort | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Comb sort | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Bubble Sort | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Selection Sort | 269,999.00 | 179,999.00 | 119,999.00 | 79,999.00 | 49,999.00 |
| Merge Sort | 9,751,424.00 | 6,315,712.00 | 4,057,856.00 | 1,568,928.00 | 1,568,928.00 |
| QuickSort | 138,928.00 | 114,464.00 | 65,535.00 | 47,232.00 | 32,767.00 |
| Merge Sort Externo | 7,984,640.00 | 5,104,640.00 | 3,272,320.00 | 2,076,160.00 | 1,236,160.00 |
| Heap Sort | 4,819,069.00 | 3,124,081.00 | 2,008,521.00 | 1,293,151.00 | 773,305.00 |

| Quantidade de trocas dos vetores com elementos distintos e ordenados de forma reversa | | | | | |
|---|---|---|---|---|---|
| **Algoritmo** | **270 000 elementos** | **180 000 elementos** | **120 000 elementos** | **80 000 elementos** | **50 000 elementos** |
| Insertion Sort | 36,449,865,000.00 | 16,199,910,000.00 | 7,199,940,000.00 | 3,199,960,000.00 | 1,249,975,000.00 |
| Comb sort | 709,494.00 | 459,846.00 | 302,248.00 | 192,662.00 | 116,838.00 |
| Bubble Sort | 36,449,865,000.00 | 16,199,910,000.00 | 7,199,940,000.00 | 3,199,960,000.00 | 1,249,975,000.00 |
| Selection Sort | 269,999.00 | 179,999.00 | 119,999.00 | 79,999.00 | 49,999.00 |
| Merge Sort | 9,751,424.00 | 6,315,712.00 | 4,057,856.00 | 2,617,856.00 | 1,568,928.00 |
| QuickSort | 273,927.00 | 204,463.00 | 125,534.00 | 87,231.00 | 57,766.00 |
| Merge Sort Externo | 7,984,640.00 | 5,104,640.00 | 3,272,320.00 | 2,076,160.00 | 1,236,160.00 |
| Heap Sort | 4,430,389.00 | 2,850,091.00 | 1,831,055.00 | 1,174,075.00 | 698,893.00 |

| Qtd de trocas de vetores com valores distribuídos aleatoriamente, e cada elemento com qtd >= a 5% do tamanho | | | | | |
|---|---|---|---|---|---|
| **Algoritmo** | **270 000 elementos** | **180 000 elementos** | **120 000 elementos** | **80 000 elementos** | **50 000 elementos** |
| Insertion Sort | 14,579,237,424.00 | 6,479,341,062.00 | 2,881,262,613.00 | 1,280,861,667.00 | 499,693,401.00 |
| Comb sort | 187,066.00 | 129,316.00 | 111,835.00 | 63,160.00 | 42,230.00 |
| Bubble Sort | 14,579,930,612.00 | 6,480,110,102.00 | 2,880,334,072.00 | 1,279,150,498.00 | 499,437,727.00 |
| Selection Sort | 269,999.00 | 179,999.00 | 119,999.00 | 79,999.00 | 49,999.00 |
| Merge Sort | 9,751,424.00 | 6,315,712.00 | 4,057,856.00 | 2,617,856.00 | 1,568,928.00 |
| QuickSort | 1,739,810.00 | 1,095,775.00 | 716,026.00 | 450,216.00 | 271,110.00 |
| Merge Sort Externo | 7,984,640.00 | 5,104,640.00 | 3,272,320.00 | 2,076,160.00 | 1,236,160.00 |
| Heap Sort | 3,694,382.00 | 2,391,858.00 | 1,532,342.00 | 991,501.00 | 589,000.00 |

| Tempo de execução dos vetores com elementos distintos e distruibuidos de forma aleatória | | | | | |
|---|---|---|---|---|---|
| **Algoritmo** | **270 000 elementos** | **180 000 elementos** | **120 000 elementos** | **80 000 elementos** | **50 000 elementos** |
| Insertion Sort | 2.05 | 2.10 | 2.07 | 2.04 | 2.04 |
| Comb sort | 0.05 | 0.05 | 0.03 | 0.02 | 0.02 |
| Bubble Sort | 180.03 | 77.73 | 42.42 | 18.18 | 9.02 |
| Selection Sort | 92.64 | 41.09 | 18.14 | 14.82 | 5.76 |
| Merge Sort | 0.10 | 0.07 | 0.05 | 0.03 | 0.02 |
| QuickSort | 0.02 | 0.01 | 0.01 | 0.01 | 0.01 |
| Merge Sort Externo | 0.50 | 0.43 | 0.28 | 0.21 | 0.13 |
| Heap Sort | 0.12 | 0.08 | 0.05 | 0.03 | 0.04 |

| Tempo de execução dos vetores com elementos distintos e ordenados | | | | | |
|---|---|---|---|---|---|
| **Algoritmo** | **270 000 elementos** | **180 000 elementos** | **120 000 elementos** | **80 000 elementos** | **50 000 elementos** |
| Insertion Sort | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Comb sort | 166.91 | 0.04 | 34.12 | 0.02 | 0.01 |
| Bubble Sort | 169.22 | 74.42 | 32.91 | 14.55 | 5.41 |
| Selection Sort | 174.26 | 77.42 | 0.03 | 15.09 | 6.48 |
| Merge Sort | 0.10 | 0.07 | 0.04 | 0.02 | 0.02 |
| QuickSort | 0.02 | 0.01 | 0.01 | 0.01 | 0.00 |
| Merge Sort Externo | 0.58 | 0.40 | 0.33 | 0.26 | 0.19 |
| Heap Sort | 0.13 | 0.09 | 0.05 | 0.03 | 0.02 |

| Tempo de execução dos vetores com elementos distintos e ordenados de forma reversa | | | | | |
|---|---|---|---|---|---|
| **Algoritmo** | **270 000 elementos** | **180 000 elementos** | **120 000 elementos** | **80 000 elementos** | **50 000 elementos** |
| Insertion Sort | 253.29 | 118.26 | 55.28 | 23.83 | 12.28 |
| Comb sort | 0.03 | 0.02 | 0.01 | 0.01 | 0.01 |
| Bubble Sort | 169.18 | 84.29 | 37.28 | 16.18 | 8.40 |
| Selection Sort | 83.86 | 37.62 | 19.61 | 8.58 | 4.00 |
| Merge Sort | 0.05 | 0.04 | 0.03 | 0.02 | 0.01 |
| QuickSort | 0.01 | 0.01 | 0.01 | 0.00 | 0.00 |
| Merge Sort Externo | 0.30 | 0.26 | 0.19 | 0.16 | 0.14 |
| Heap Sort | 0.06 | 0.04 | 0.03 | 0.02 | 0.01 |

| Tempo de execução dos vetores com valores distribuídos aleatoriamente, e cada elemento com qtd >= a 5% do tamanho | | | | | |
|---|---|---|---|---|---|
| **Algoritmo** | **270 000 elementos** | **180 000 elementos** | **120 000 elementos** | **80 000 elementos** | **50 000 elementos** |
| Insertion Sort | 107.60 | 48.00 | 21.10 | 9.97 | 4.18 |
| Comb sort | 0.04 | 0.02 | 0.01 | 0.01 | 0.01 |
| Bubble Sort | 147.87 | 77.17 | 36.51 | 32.36 | 8.62 |
| Selection Sort | 85.23 | 42.83 | 18.62 | 16.84 | 3.49 |
| Merge Sort | 0.06 | 0.04 | 0.03 | 0.02 | 0.01 |
| QuickSort | 0.02 | 0.01 | 0.01 | 0.01 | 0.00 |
| Merge Sort Externo | 0.25 | 0.20 | 0.15 | 0.18 | 0.09 |
| Heap Sort | 0.05 | 0.03 | 0.02 | 0.02 | 0.01 |

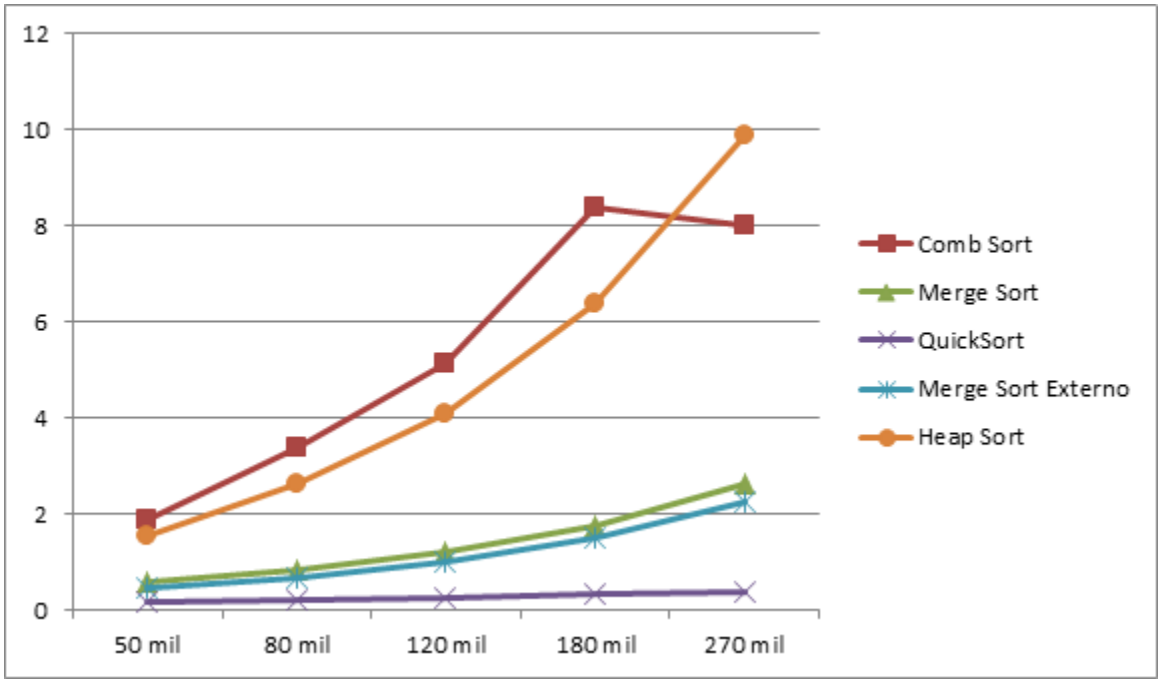
## 5. Resultados gráficos dos experimentos

### 5.1 Relação Comparações X Tamanho do Vetor

**Vetor Aleatório**

Foram removidos os algoritmos bubble sort, Selection e Insertion Sort para uma melhor visualização do gráfico, pois os valores dos três algoritmos destoam do restante.

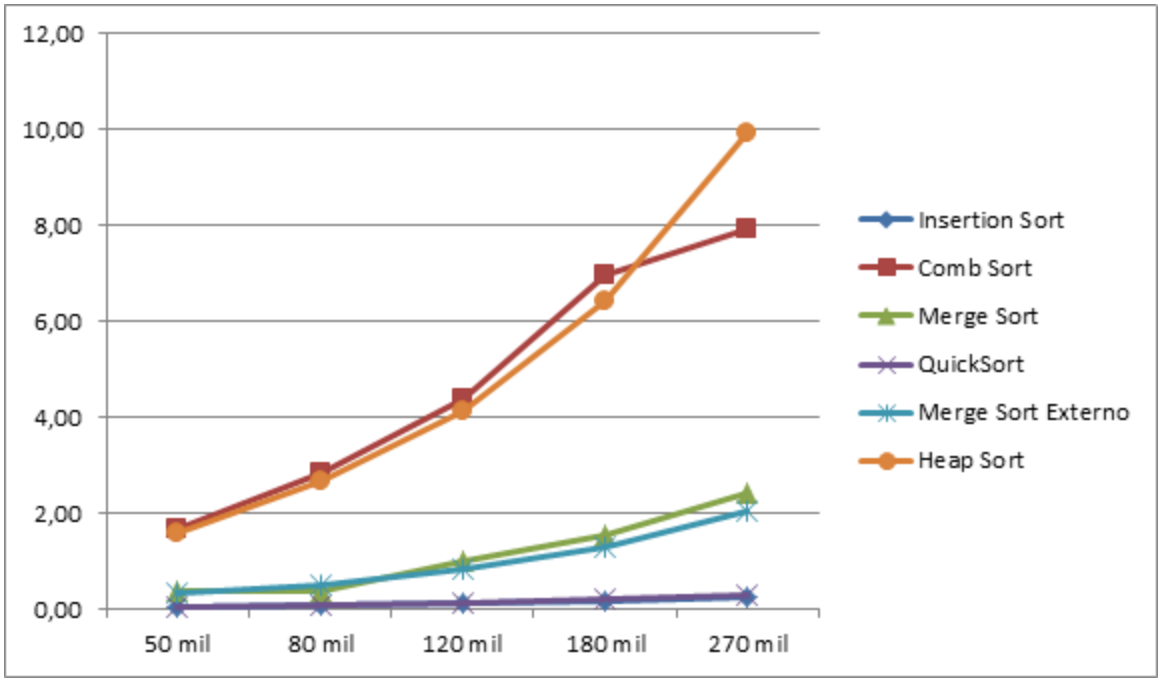
Valores em Milhões




**Vetor Ordenado**

Valores em Milhões

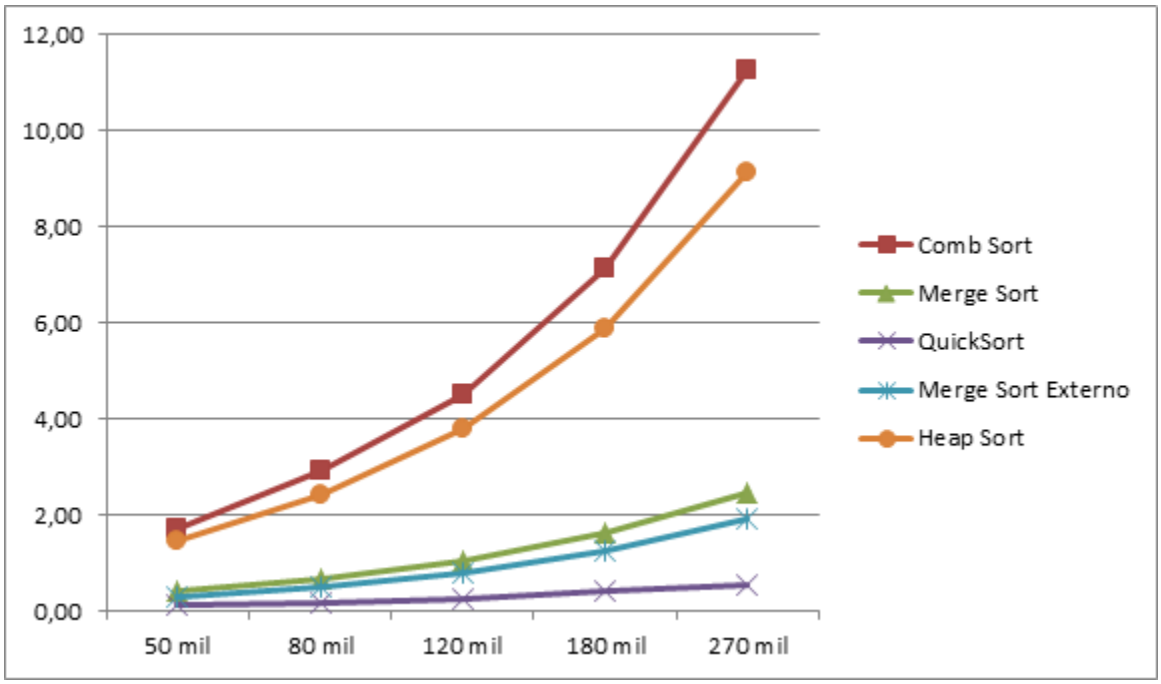
Foram removidos os algoritmos bubble sort, Selection Sort para uma melhor visualização do gráfico, pois os valores dos três algoritmos destoam do restante.




**Vetor Ordenado Reverso**

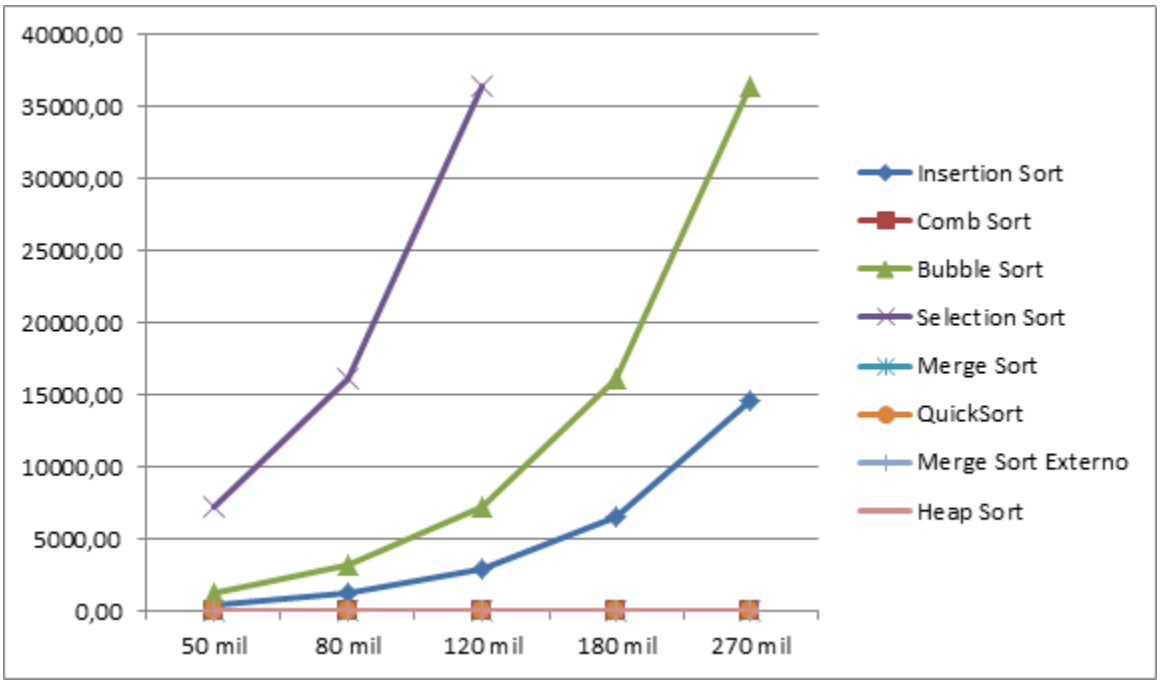
Foram removidos os algoritmos bubble sort, Selection Sort e Insertion Sort para uma melhor visualização do gráfico, pois os valores dos três algoritmos destoam do restante.

Valores em milhões



**Vetor com valores distribuídos aleatoriamente, e cada elemento com qtd >= 5% do tamanho**

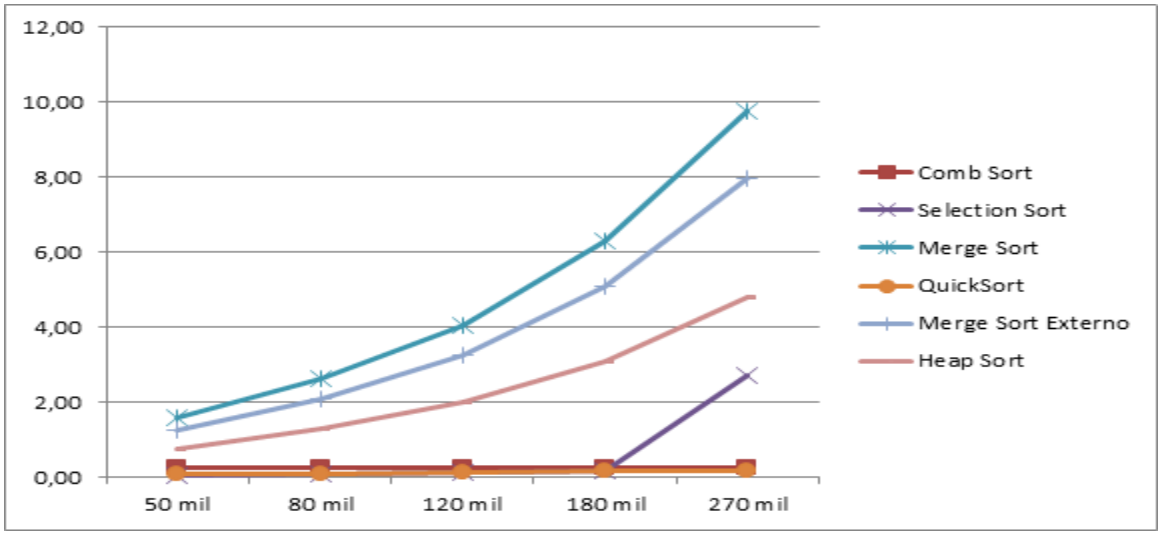
Valores em milhões




## 5.2 Número de Trocas x Tamanho do Vetor

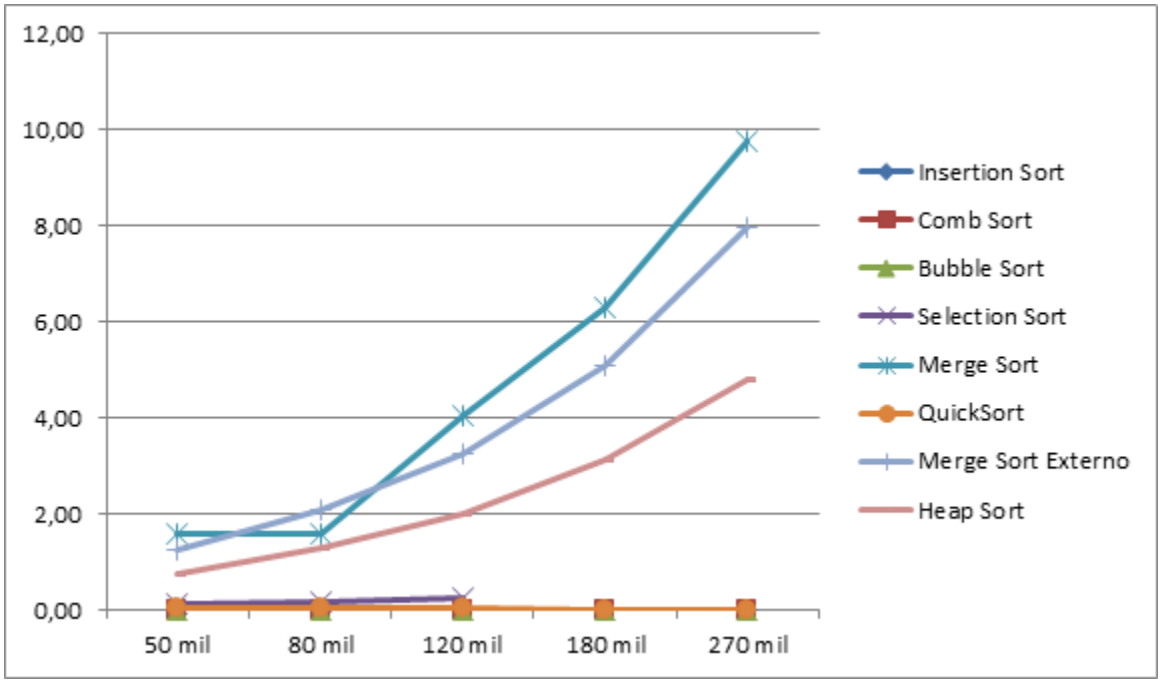
### Vetor Aleatório

Valores em Milhões



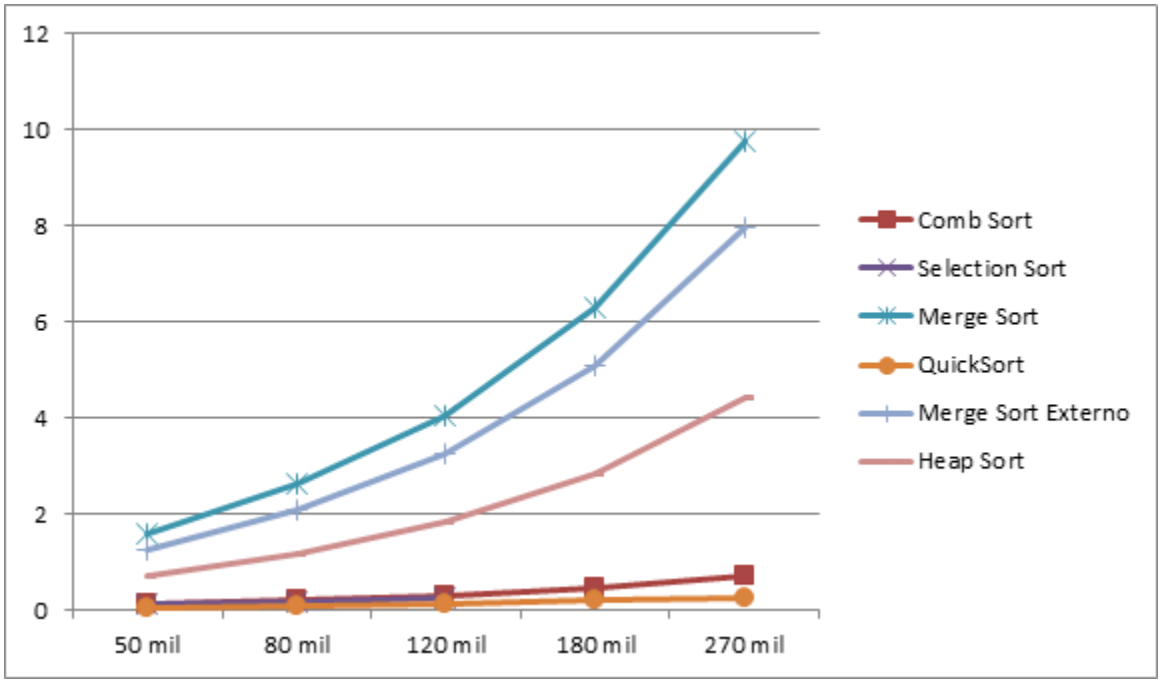
**Vetor ordenado**

Valores em Milhões



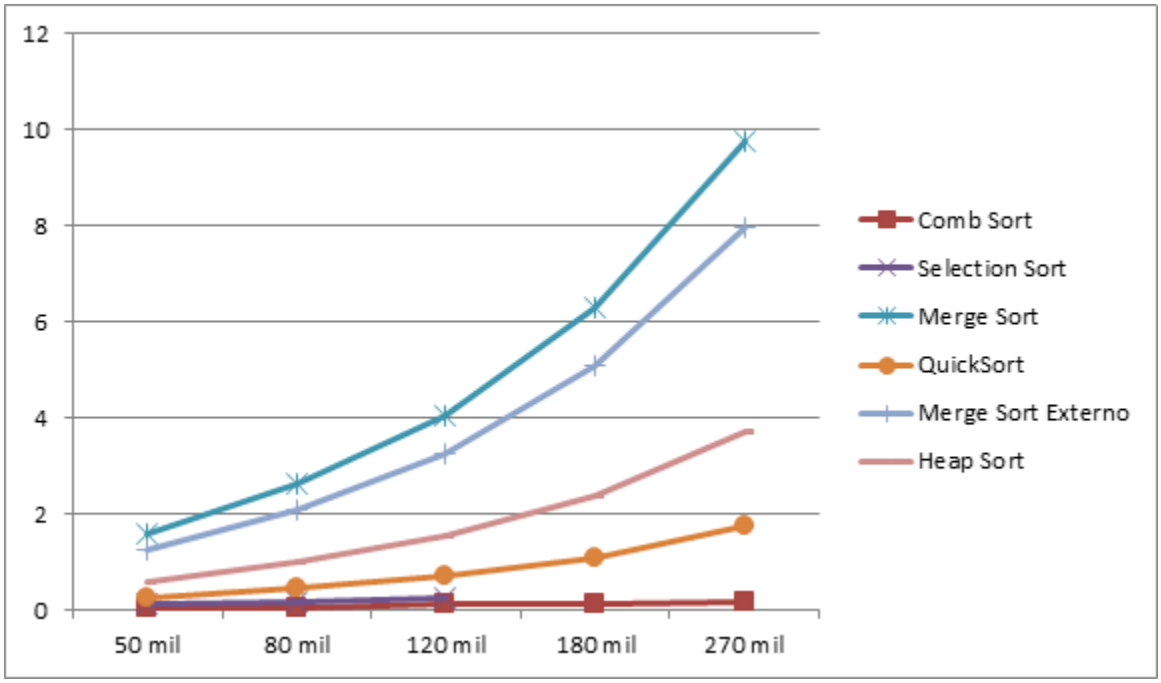

**Vetor Ordenado de Forma Reversa**

Valores em milhões



**Vetor distribuído aleatoriamente, e cada elemento com qtd >= 5% do tamanho**

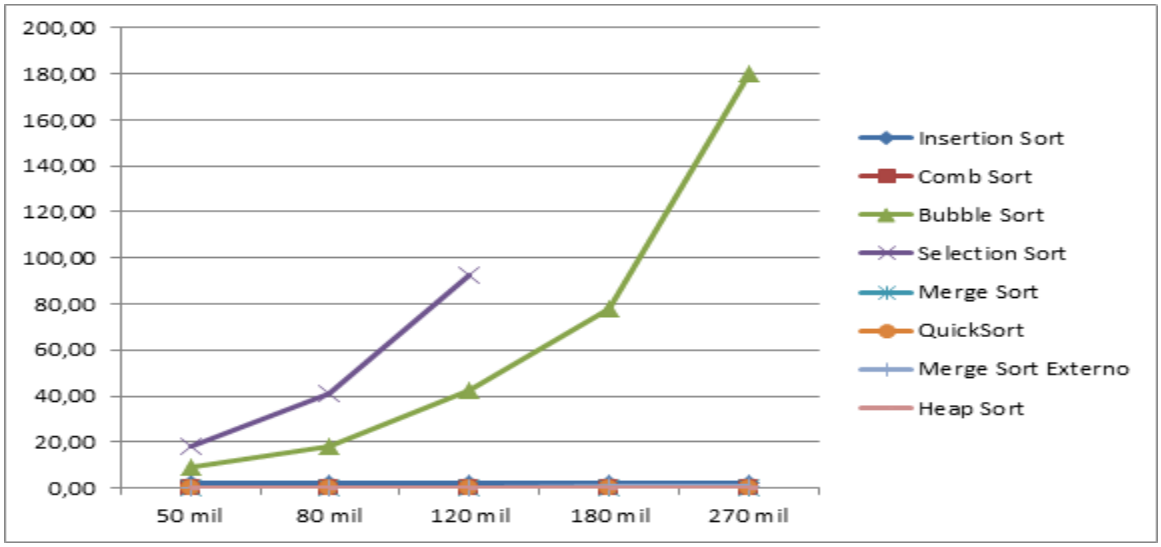
Valores em milhões




## 5.3 Tempo x Tamanho do Vetor

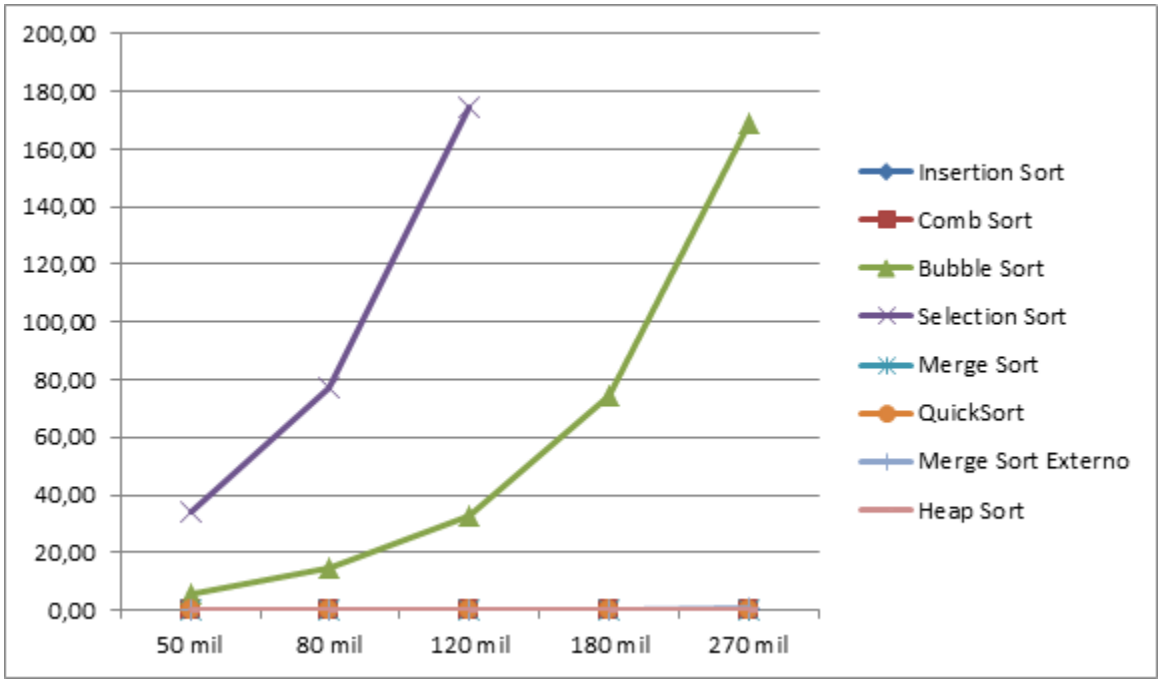
**Vetor Aleatório**

Valores em segundos



**Vetor Ordenado**

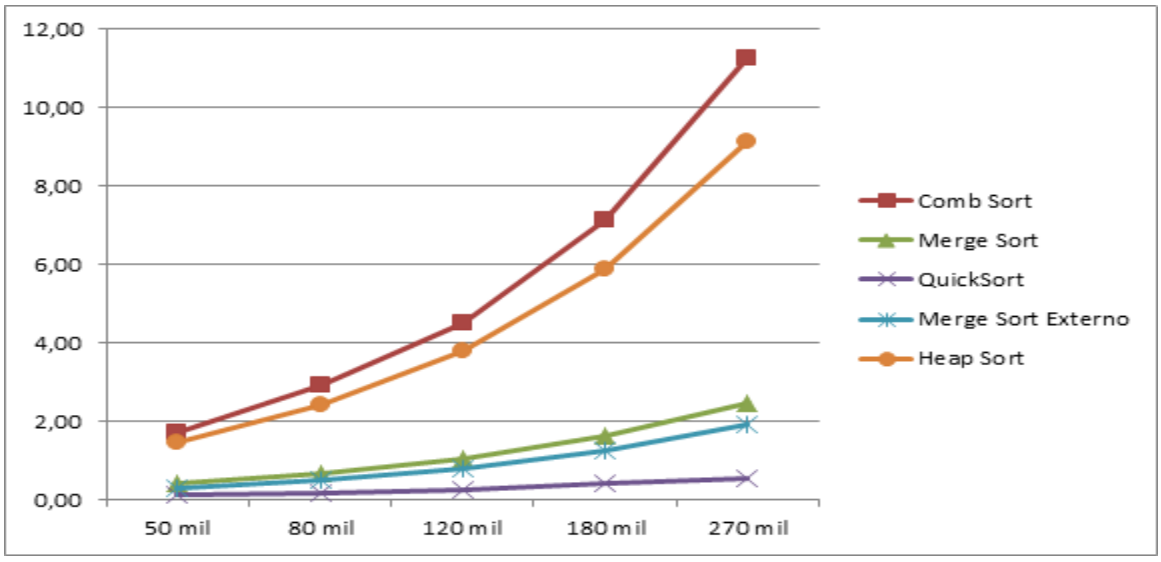
Valores em segundos




**Vetor Ordenado de Forma Reversa**

Foram removidos os algoritmos bubble sort, Selection Sort e Insertion Sort para uma melhor visualização do gráfico, pois os valores dos três algoritmos destoam do restante.

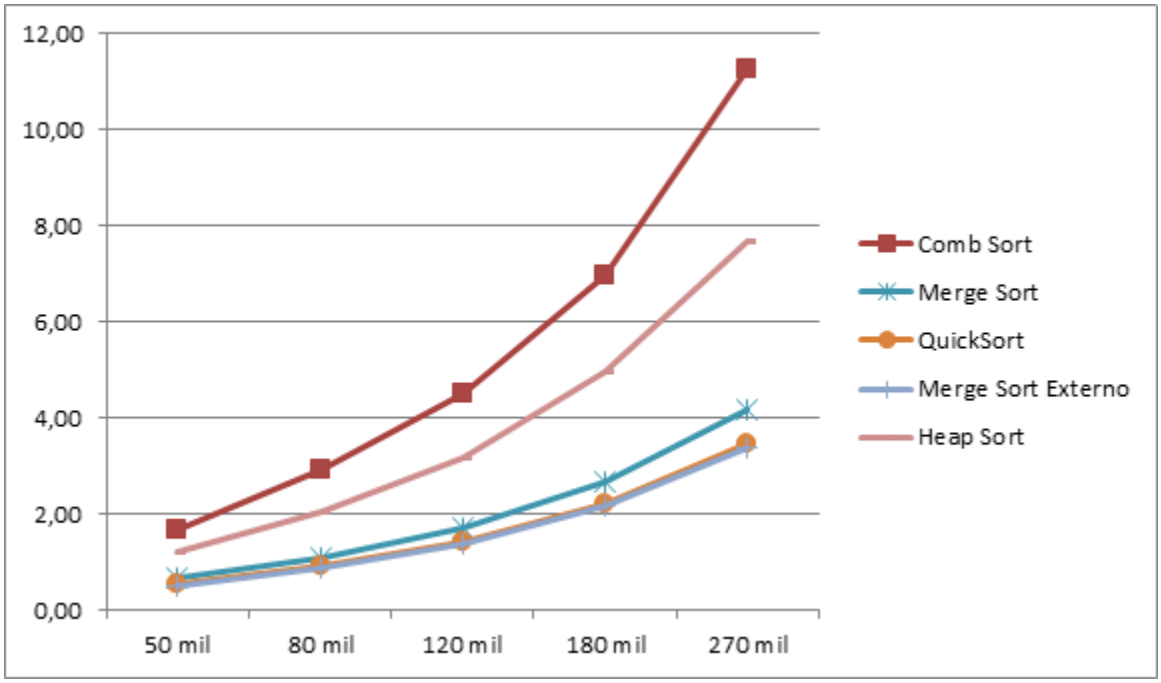
Valores em Segundos



**Vetor distribuído aleatoriamente, e cada elemento com qtd >= 5% do tamanho**

Foram removidos os algoritmos bubble sort, Selection Sort e Insertion Sort para uma melhor visualização do gráfico, pois os valores dos três algoritmos destoam do restante.

Valores em segundos




## 6. Discussão sobre os resultados obtidos.

Os resultados obtidos foram bem interessantes, alguns algoritmos diferem bastante um dos outros, já outros tiveram resultados próximos. Claro que de acordo com o tipo que se encontra o vetor, os tipos de troca, os tipos de comparações, os resultados variam e muito.

Vamos dar início analisando as comparações feita nos 8 algoritmos, com vetores de 50.000, 80.000, 120.000, 180.000 e 270.000 elementos.

De início temos um vetor com elementos distribuídos de forma aleatória. Começando por 50.000 elementos, os algoritmos Bubble Sort e Selection Sort tiveram a mesma quantidade de comparações (1.249.975.000 de comparações para ambos). O algoritmo Insertion Sort também teve uma grande quantidade de comparações (262.049.887). O melhor algoritmo para realizar comparações com um vetor aleatório de 50.000 elementos é o Quick Sort com incríveis 175.346 comparações, muito mais versátil que os primeiros algoritmos citados. Merge Sort Externo também teve um bom rendimento com 470.695 trocas e, o Heap Sort teve um desempenho pior com 1.562.580 comparações. Aumentando para um vetor aleatório de 120.000 elementos, temos ainda Bubble Sort e Selection Sort como os piores algoritmos com 7.199.940.000 trocas cada um, e o QuickSort assim como no vetor de 50.000 elementos, temos ele como o melhor

algoritmo de ordenação nessa ocasião com 250.792 comparações. Por último teste temos um vetor de 270.000 elementos, destacando novamente os algoritmos Bubble Sort e Selection Sort como os piores para se realizar comparações, ambos com 3.6449.865.000 comparações. E sem ser novidade nesse caso de teste o algoritmo Quick Sort se sobressai como o melhor, com 385 362 comparações. Um ponto curioso é que a partir do vetor com 120.000 elementos, todos os algoritmos de ordenação chegam na casa do milhão de comparações, exceto o Quick Sort, e também o Bubble Sort e Selection Sort que já se encontram na casa dos milhões.

Agora vamos analisar casos de comparações com um vetor ordenado, começando com 50.000 elementos temos novamente Selection Sort e Bubble Sort com um maior número de comparações em relação aos outros algoritmos, 1.249.975.000 para ambos, mesma quantidade do caso do vetor com elementos aleatórios. Já o insertion Sort cai nas comparações para 49.999 com um vetor ordenado, já com um vetor aleatório são 262.049.887 comparações, muita diferença. O Quick Sort nesse caso não é o melhor com 65.534 comparações. Aumentando para um vetor ordenado de 120.000 elementos somente os algoritmos Comb Sort, Bubble Sort, Selection Sort e Heap Sort, chegam na casa dos milhões de comparação, tendo ainda Insertion Sort como o melhor algoritmo com 119.999 de comparações, bem diferente do Bubble e Selection com 7.199 940.000 comparações. Passando para um vetor com 180.000 elementos, exceto o Quick Sort e Selection Sort chegam na casa dos milhões de comparações, mesma ocasião para um vetor com 270.000 elementos onde o Quick Sort com 277.856 comparações se aproxima do Insertion com 269.999 comparações, caso fosse testado em um vetor com mais de 300.000 elementos ordenados, provavelmente o Quick Sort se tornaria o melhor em questões de comparações em um vetor ordenado.

Já em um caso de um vetor ordenado de forma reversa, já com 50.000 elementos todos os algoritmos, exceto 3 (Merge Sort, Quick Sort, Merge Sort Externo), chegam na casa dos milhões de comparações, todos os outros chegam. Já com um vetor de 180.000 elementos todos os algoritmos menos o Quick Sort, estão na casa dos milhões de comparações, ressaltando Insertion Sort, Bubble Sort e Selection Sort, com 16.199.910.000 comparações. Já com um vetor ordenado de forma reversa com 270.000 elementos o Quick Sort segue sendo o melhor com 547.854 comparações.

Testando um ambiente com vetor com elementos aleatórios onde cada elemento é maior ou igual a 5% do tamanho do mesmo. Para um vetor de 50.000 elementos, já temos o Bubble Sort e Selection Sort “de cara” como o pior algoritmo, com 1.249.975.000 comparações, e Quick Sort e Merge Sort Externo bem pertos um do outro com 542.884 e 517.305 comparações respectivamente. Com um vetor de 120.000 elementos, o Merge Sort chega perto do Quick Sort e Merge Sort Externo com 1.73.812; 1.433.312; 1.384 885 comparações respectivas. Já com um vetor de 270.000 elementos o Merge Sort se distancia com 4.182.454 comparações e o Merge Sort Externo é o melhor com 3.401.700 comparações e novamente Bubble Sort e Selection Sort são os piores em comparação com 3.6449.865.000 comparações ambos.

Observando de maneira ampla onde se diz respeito a comparações em todas ocasiões analisadas, Bubble Sort e Selection Sort se destacam em todas, como o pior para se comparar, com altos números de comparações. O Insertion Sort varia bastante em comparação com base no tipo de vetor, e em exceto uma ocasião o Quick Sort se destaca em todas, como o algoritmo que realiza menos comparações.

Observando agora as situações de troca, com vetores de 50.000, 80.000, 120.000, 180.000, 270.000 elementos.

De início vamos observar um vetor com elementos dispostos de forma aleatória. Para um vetor com 50.000 elementos Bubble Sort e Selection Sort se destacam como os algoritmos que mais realizam trocas, 25.8916.965 e 261.999.899 trocas. Selection Sort faz menos trocas com 50.000 elementos. À medida que avança o tamanho do vetor, os algoritmos Selection Sort e Quick Sort vão se equiparando. Com um vetor de 120.000 elementos aleatórios, Selection Sort com 119.999 trocas e Quick Sort com 119.970 trocas chegam bem próximos. Selection Sort passa o Quick Sort em números de trocas com um vetor de 180.000 elementos, e quanto mais aumenta o tamanho do vetor mais o Selection Sort se distancia do Quick Sort, com 270.000 elementos Selection Sort faz 269.999 trocas e Quick Sort 187.394, sendo mais versátil. E Bubble Sort é o que mais faz troca, com 267.973.858 trocas.

Para um vetor ordenado, Insertion Sort, Comb Sort e Bubble Sort são mais versáteis, pois como o vetor já está ordenado não há a necessidade de trocar elementos, por isso os 3 algoritmos fazem 0 trocas. Ressaltando Merge Sort e Merge Sort Externo que são os que mais fazem trocas em um vetor ordenado com 9.751.424 e 7.984.640 trocas respectivamente com um vetor ordenado de 270.000 elementos.

Com um vetor ordenado de forma reversa, Insertion Sort e Bubble Sort são os que mais fazem trocas, possuindo o mesmo número de trocas em todos os tamanhos do vetor chegando no vetor de 270.000 elementos realizando 36.449.865.000 trocas ambos, e o Selection Sort passa o Quick Sort em números de trocas, onde até no momento que o vetor tem 180.000 elementos o Quick Sort faz mais trocas que o Selection, quando chega em 270.000 elementos Selection Sort faz 269.999 trocas e Quick Sort faz 273.927 trocas. Sendo o Selection Sort mais versátil quando o vetor chega a 270.000 elementos.

Quando temos um vetor aleatório onde cada elemento é maior ou igual a 5% do tamanho do mesmo, novamente Insertion Sort e Bubble Sort fazem mais trocas que os outros algoritmos, mas o Bubble Sort após passar o vetor de 120.000 elementos acaba por fazer mais trocas, por mais que sejam parecidos o Bubble Sort com 270.000 elementos faz 14.579.930.612 e o Insertion Sort 14.579.237.424 trocas. Quando temos o vetor de 180.000 elementos todos os algoritmos estão na casa do milhão de trocas, menos o Selection Sort (179.999 trocas) e o Comb Sort (129.316 trocas). Por fim o Comb Sort se destaca em todos casos de teste de tamanho do vetor como melhor algoritmo de troca (no vetor onde cada elemento é maior ou igual a 5% do tamanho do mesmo), chegando no vetor de 270.000 elementos com 187.066 trocas.

De maneira geral Insertion e Bubble Sort são os algoritmos que mais realizam trocas, exceto quando o vetor já se encontra ordenado, e o Quick Sort acaba por ser o algoritmo que menos faz trocas, exceto quando o vetor tem elementos onde o elemento é maior ou igual a 5% do tamanho do mesmo, nessa ocasião o Comb Sort predomina.

Já foi observado trocas, comparações e por fim vamos dar uma atenção ao tempo de execução em relação aos algoritmos de ordenação com vetor de 50.000, 80.000, 120.000, 180.000, 270.000 elementos.

Quando temos um vetor com elementos de forma aleatória, o tempo de execução do Quick Sort é predominantemente o melhor, com 0.01 segundos em todos os casos, e para o vetor com 270.000 elementos esse valor é 0.02 segundos. Os outros algoritmos não fogem muito desse tempo do Quick, exceto Selection Sort e Bubble Sort, onde com um algoritmo de 270.000 elementos Bubble Sort faz um tempo de 180.03 segundos.

Para um vetor ordenado o Quick Sort segue sendo o melhor em relação a tempo, possuindo os mesmos tempos do vetor aleatório. O Comb Sort, Bubble Sort e Selection Sort possuem o maior tempo de execução, com valores bem aproximados, ressaltando que de 180.000 elementos para 270.000 elementos o tempo aumenta em praticamente 100 a mais, onde o Selection é o que mais consome tempo com 174.26 segundos.

Para vetores ordenados de forma reversa o Quick Sort novamente se destaca com menor tempo de execução, a partir de 120.000 elementos temos um tempo de 0.1 segundos. E o Insertion Sort é o algoritmo que mais gasta tempo, desde o algoritmo com 50.000 elementos, chegando no 270.000 elementos com um tempo de 253.29 segundos.

Observando os vetores onde cada elemento é maior ou igual a 5% do tamanho do mesmo o Quick Sort segue sendo o que executa em menor tempo, com 0.1 e 0.2 segundos. Merge Sort também se aproxima do tempo do Quick Sort diferindo em alguns milésimos apenas. Os algoritmos Insertion Sort, Bubble Sort e Selection Sort tem o maior tempo de execução com o Bubble Sort ganhando em todos os tamanhos do vetor com 147.87 segundos no vetor com 270.000 elementos.

Observando de maneira geral o Quick Sort é o algoritmo de ordenação que executou em menor tempo em todos os casos, já o Bubble Sort é o algoritmo que leva maior tempo para executar, exceto quando o vetor está ordenado de forma reversa.

## 7. Conclusão

Concluímos que dependendo da aplicação cada algoritmo possui alguma vantagem sobre os outros tendo melhor desempenhos em certos casos, dependendo do tipo de tarefa a ser resolvida.

Observamos que o Insertion Sort é usado quando o número de elementos é pequeno, ele também pode ser útil quando o vetor de entrada está quase ordenado, dessa forma apenas alguns elementos são colocados no lugar errado em um grande vetor completo.

O Selection Sort pode ser bom para verificar se tudo já está ordenado, ele também é bom usar quando o espaço de memória é limitado. Isso ocorre porque, ao contrário de outros algoritmos de ordenação, ele não troca coisas até o final, resultando em menos espaço de armazenamento temporário usado.

O Bubble Sort é um dos mais lentos para ordenar grandes vetores, ele deve ser usado para vetores com poucos elementos.

O Comb Sort se demonstrou mais eficiente que o Bubble Sort, podendo ser usado com vetores de diferentes tamanhos, e ele é mais eficiente com vetores de diferentes finais quando os dados estão quase ordenados.

O algoritmo Merge sort é o algoritmo mais estável e é usado em aplicações que necessitam de um tempo de execução fixo.

O Quick Sort é o algoritmo mais rápido e usado para ordenar qualquer tipo de vetor de maneira rápida e eficiente.

Heap Sort também é um algoritmo bem rápido e eficiente para diversos tipos de vetores.

O Merge Sort Externo deve ser usado para ordenar arquivos muito grandes para caber na memória.

Portanto, com os dados estudados, pode-se concluir que o algoritmo mais rápido é o Quick Sort, e os algoritmos Bubble Sort, Insertion Sort e Selection Sort são opções inviáveis para ordenação em larga escala. Os Algoritmos Heap Sort e Merge Sort Externo são algoritmos rápidos porém possuem uma complexidade maior na hora da implementação, como o Quick Sort se mostra, na maioria dos casos, mais rápido que estes dois é mais viável utilizá-lo, com exceção a arquivos muitos grandes para caber na memória, podendo usar o Merge Sort Externo. O algoritmo Bubble Sort e Selection Sort são algoritmos de fácil implementação e são usados somente em aplicações mais simples.

## 8. Referências


https://www.researchgate.net/profile/Gf-Cintra/publication/279708678_Pesquisa_e_Ordenacao_de_Dados/links/5597f0e908ae793d137dfa16/Pesquisa-e-Ordenacao-de-Dados.pdf

https://www.geeksforgeeks.org/external-sorting/

https://www.geeksforgeeks.org/heap-sort/

https://pt.wikipedia.org/wiki/Merge_sort

https://en.wikipedia.org/wiki/External_sorting

https://pt.wikipedia.org/wiki/Heapsort

http://www.ijcim.th.org/past_editions/2005V13N3/ijcimv13n3_article5.pdf