Universidade Estadual Júlio de Mesquita Filho

UNESP

Projeto de Análise de Algoritmos

Análise experimental e assintótica de algoritmos de ordenação

Henrique Leal Tavares

Bauru – SP

Sumário

1. Introdução

Os métodos de ordenação de dados são de fundamental importância em sistemas computacionais, escolher um método eficiente para determinada atividade gera um grande impacto no processamento e no desempenho de uma aplicação, existem diversos métodos famosos de ordenação, cada qual com seu potencial específico, sendo a melhor ou a pior opção em alguns cenários.

Este trabalho tem como objetivo testar, comparar e demonstrar como os algoritmos de ordenação: *BubbleSort*, *BubbleSort* com validação de lista ordenada, *QuickSort* com seu ponteiro no centro e no início do vetor, *InsertionSort*, *SelectionSort*, *ShellSort*, *MergeSort* e *HeapSort*, como eles atuam em diversos cenários de Caso médio (disposição aleatória dos vetores tentando evitar o melhor e o pior caso), Melhor caso (vetores ordenados, com a intenção de reduzir o número de comparações presentes no programa) e Pior caso (vetor disposto a entrar em todas as comparações e funções possíveis dentro do algoritmo).

Os valores dos vetores gerados foram de 1 até sua posição final, foram utilizados vetor de tamanho 1000, 5000, 10000, 15000, 20000 e 25000 posições, em ordem crescente, decrescente e aleatória, gerada pela função *random.sampe()*.

Os algoritmos foram executados no mínimo 10 vezes cada, a média de seu tempo foi calculada para uma análise mais confiável, todos foram executados nas mesmas configurações de máquina (Intel i3, placa de vídeo integrada 8GB RAM), a linguagem utilizada foi o *Python* e para captura do tempo foi utilizada a função *time.time()*.

2. Análise

Nos gráficos que serão demonstrados o eixo **x** representa o valor do vetor inserido no algoritmo e o eixo **y** o tempo em **ms** (milissegundos) que o mesmo levou.

2.1 Bubble Sort

O método de *Bubble Sort* é conhecido por sua fácil implementação e por suas complexidades semelhantes, em teoria o mesmo apenas realiza permutação do maior valor e menor valor, esse processo é repetido até que todos os itens da lista estejam em ordem.

Melhor caso	Caso Médio	Pior Caso
$O(n^2)$	$O(n^2)$	$O(n^2)$

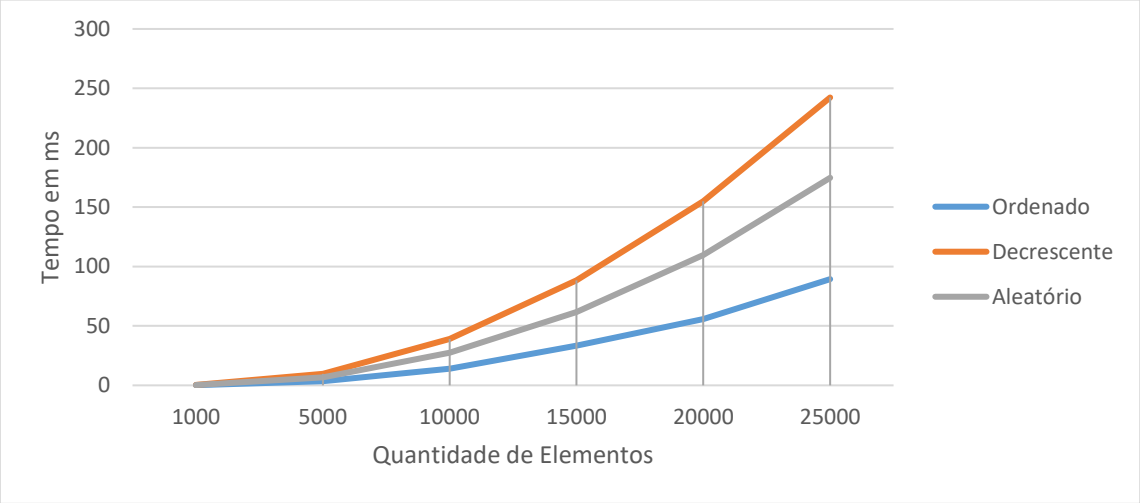

Figura 1 - Bubble Sort (Tempo)

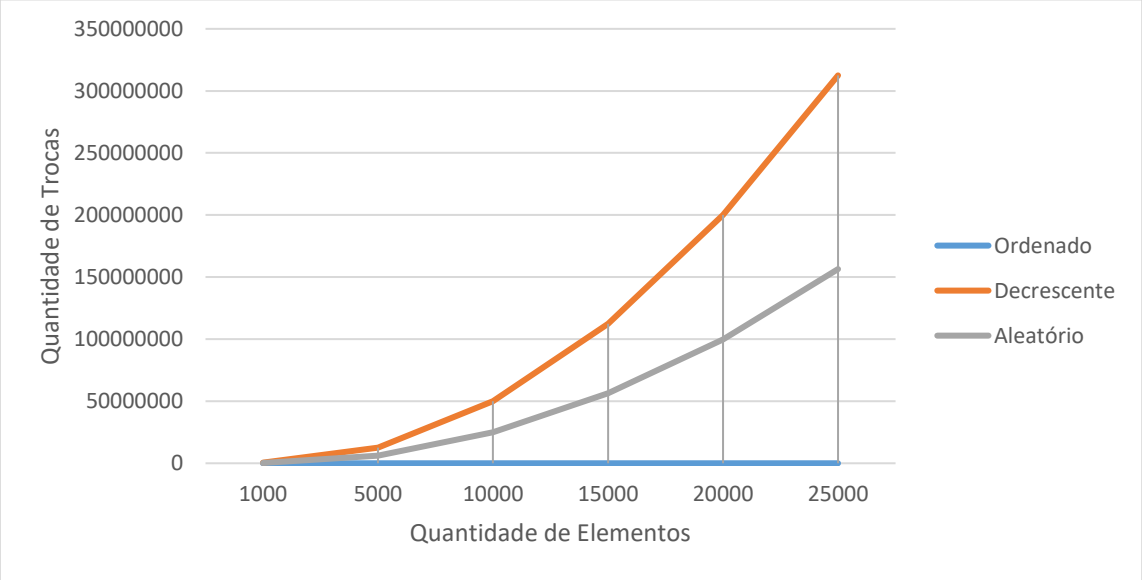

Figura 2 - Bubble Sort (Quantidade de Trocas)

2.2 Bubble Sort Otimizado

Este algoritmo é muito semelhante ao *Bubble Sort* original, porém antes de realizar as permutações é executada uma verificação se o vetor já está ordenado.

Melhor caso	Caso Médio	Pior Caso
O(n)	$O(n^2)$	$O(n^2)$

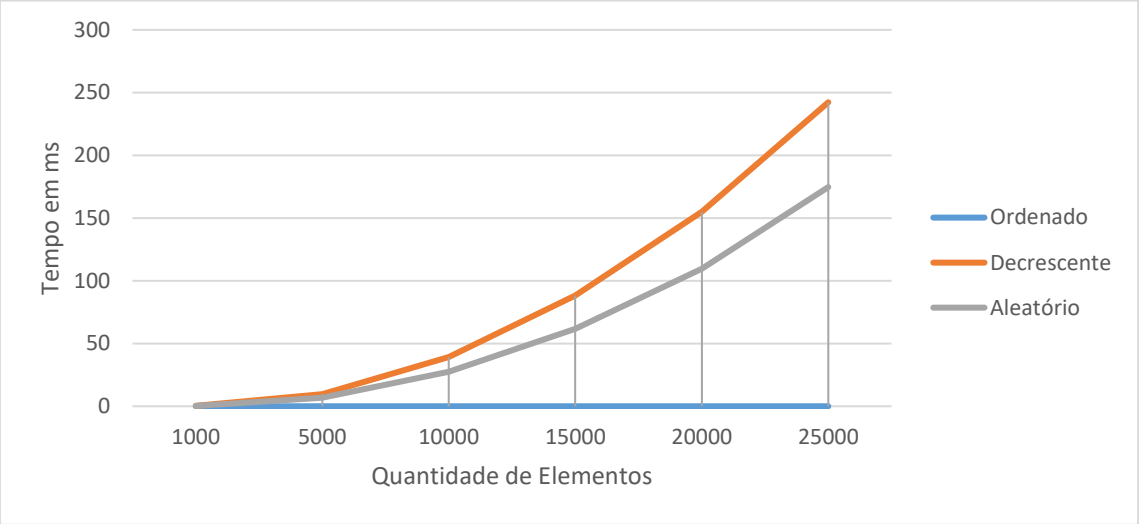

Figura 3 - Bubble Sort Otimizado (Tempo)

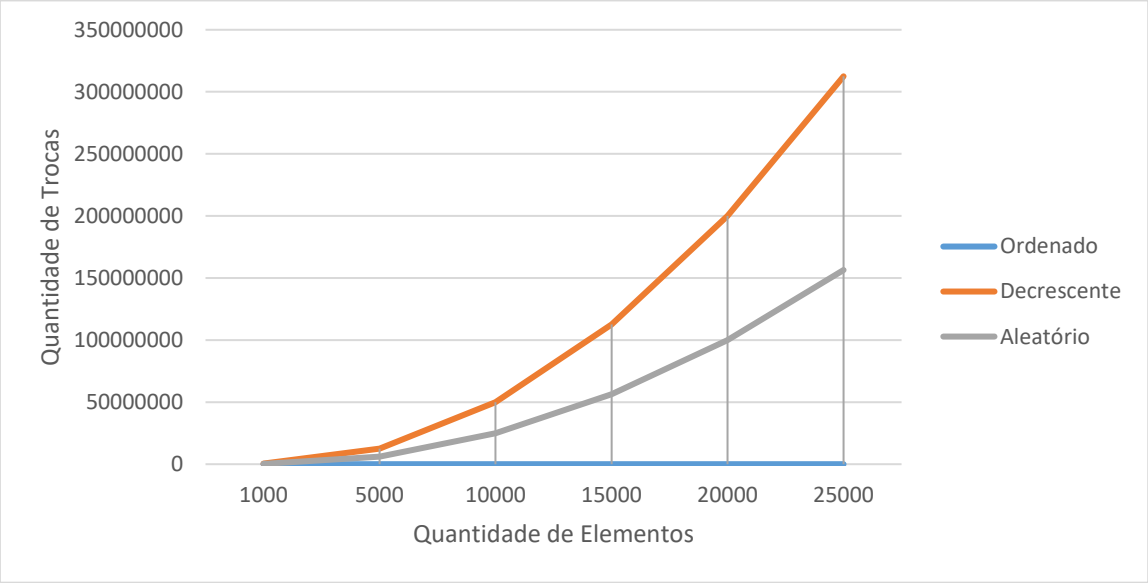

Figura 4 - Bubble Sort Otimizado (Quantidade de Trocas)

2.3 Quick Sort

O *Quick Sort* utiliza a estratégia de divisão e conquista, onde é escolhi um elemento dentro da lista (pivô) e a partir dele são permutados os maiores e melhores valores, a lista é particionada para resolução de problemas menores e a partir disto as listas são combinadas posteriormente à ordenação, para que um maior problema seja resolvido.

2.3.1 Pivô Inicial

Neste caso o pivô é escolhido pelo primeiro elemento da lista.

Melhor caso	Caso Médio	Pior Caso
O(n log (n))	**O(n log (n))**	**O(n^2)**

Figura 5 – Quick Sort Pivô Inicial (Tempo)

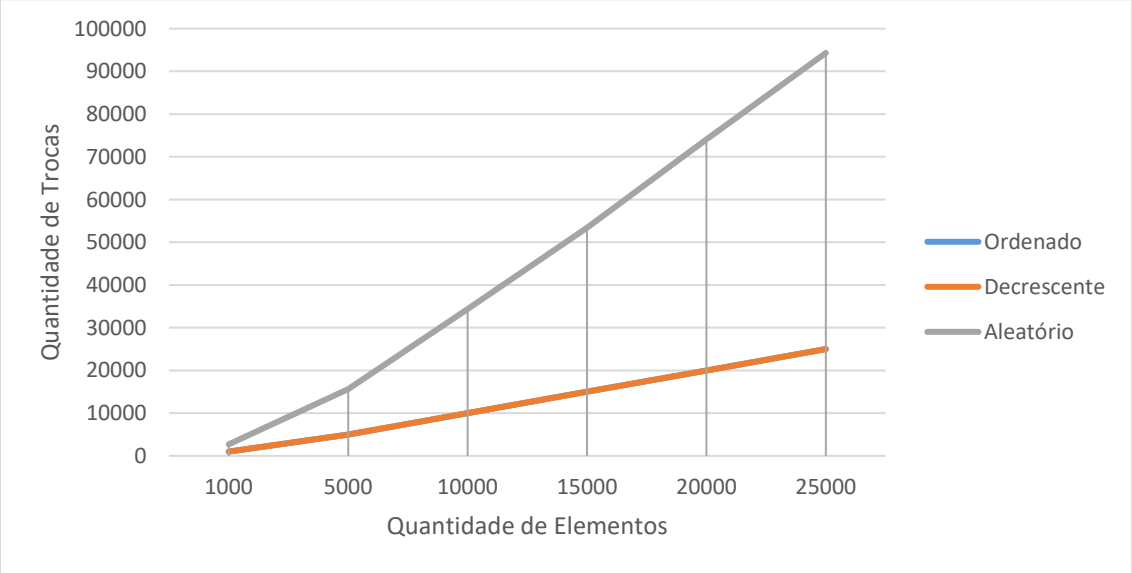

Figura 6 – Quick Sort Pivô Inicial (Quantidade de Trocas)

2.3.2 Pivô Central

Uma maneira eficaz de se evitar o pior caso do Quick Sort é mudando a metodologia de escolha de pivôs do algoritmo. Ao invés de escolher o primeiro (ou último) elemento, como o Quick Sort padrão, pode-se escolher o elemento central do conjunto de dados como pivô (assegurando que, na maioria dos casos, não há a geração de subproblemas de tamanho *0*).

Melhor caso	Caso Médio	Pior Caso
O(n log (n))	**O(n log (n))**	**O(n²)**

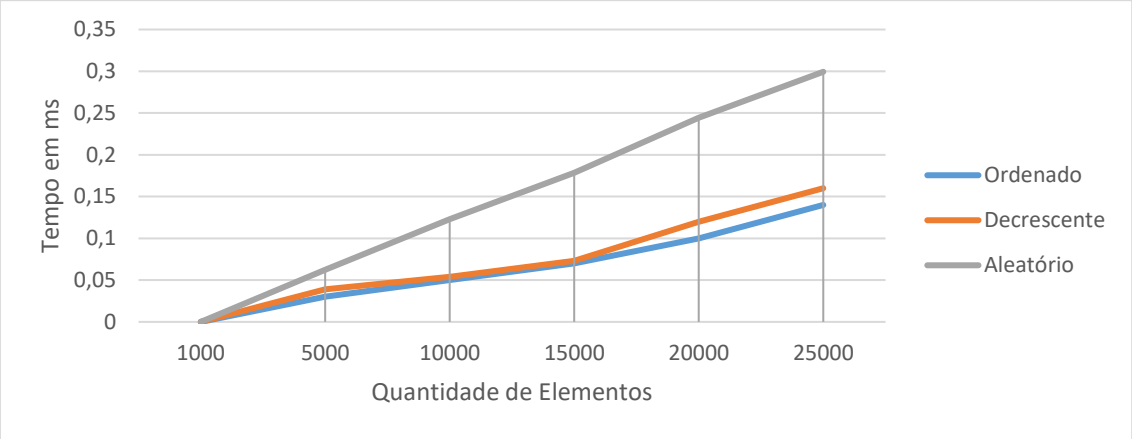

Figura 7 – Quick Sort Pivô Central (Tempo)

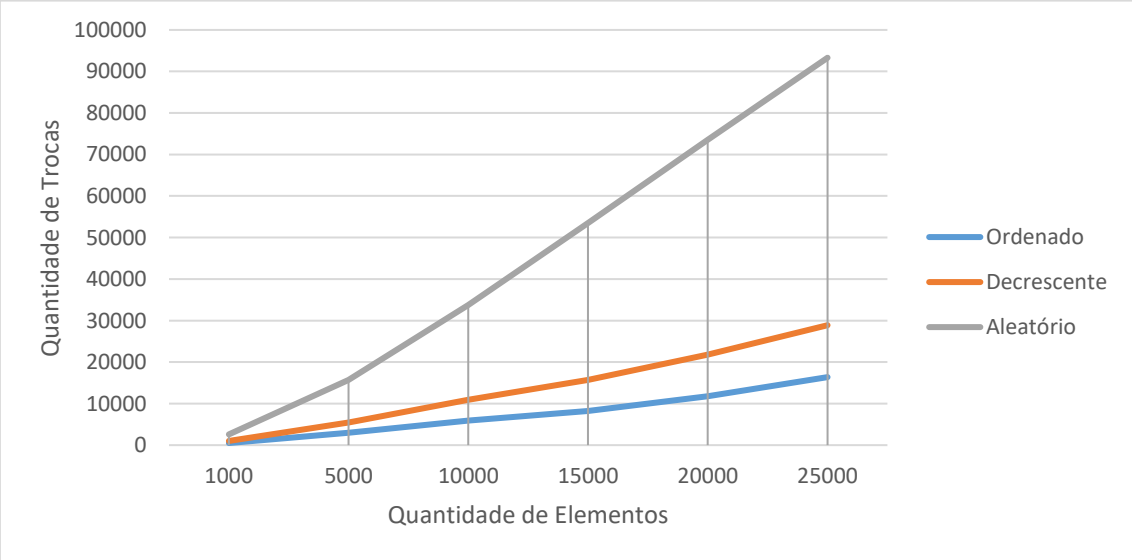

Figura 8 – Quick Sort Pivô Central (Quantidade de Trocas)

Podemos perceber que com o pivô concentrando-se no meio do vetor o desempenho deste algoritmo para conjuntos crescentes ou decrescentes melhorou de n^2 para $n\ log(n)$

e também na configuração aleatória o tempo foi drasticamente menor em relação ao algoritmo que toma o primeiro ou último elemento como pivô.

2.4 Insert Sort

Insertion Sort utiliza ordenação por inserção, onde dada uma lista é construída e realizada uma inserção por vez onde o objeto se encaixa da melhor forma. Pertence aos algoritmos de ordenação quadrática, descantando-se assim para organização de vetores com pequenas entradas.

Melhor caso	Caso Médio	Pior Caso
O(n)	**O(n^2)**	**O(n^2)**

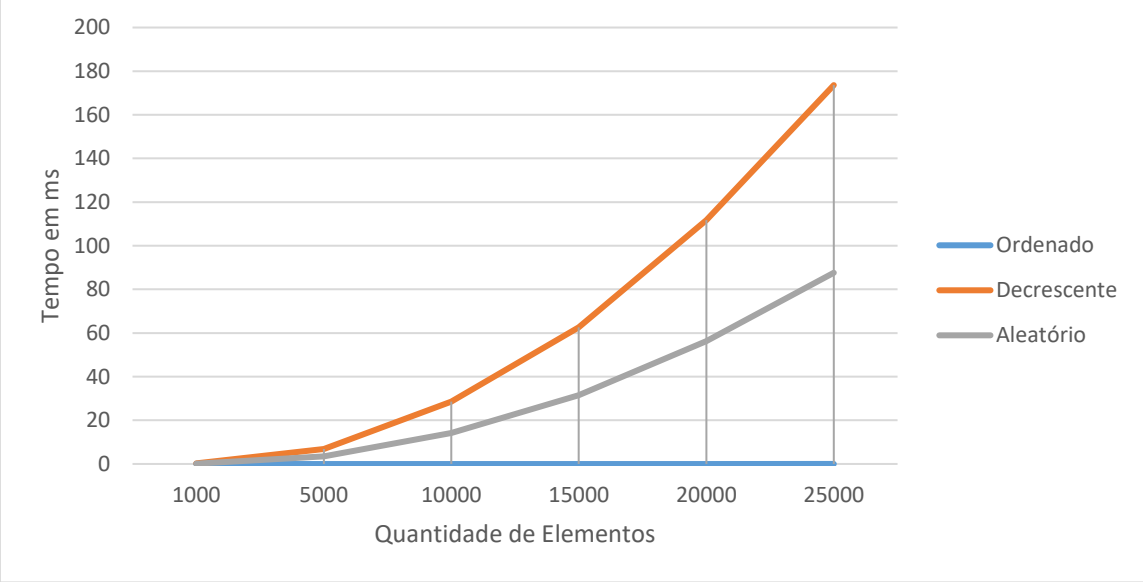

Figura 9 – Insert Sort (Tempo)

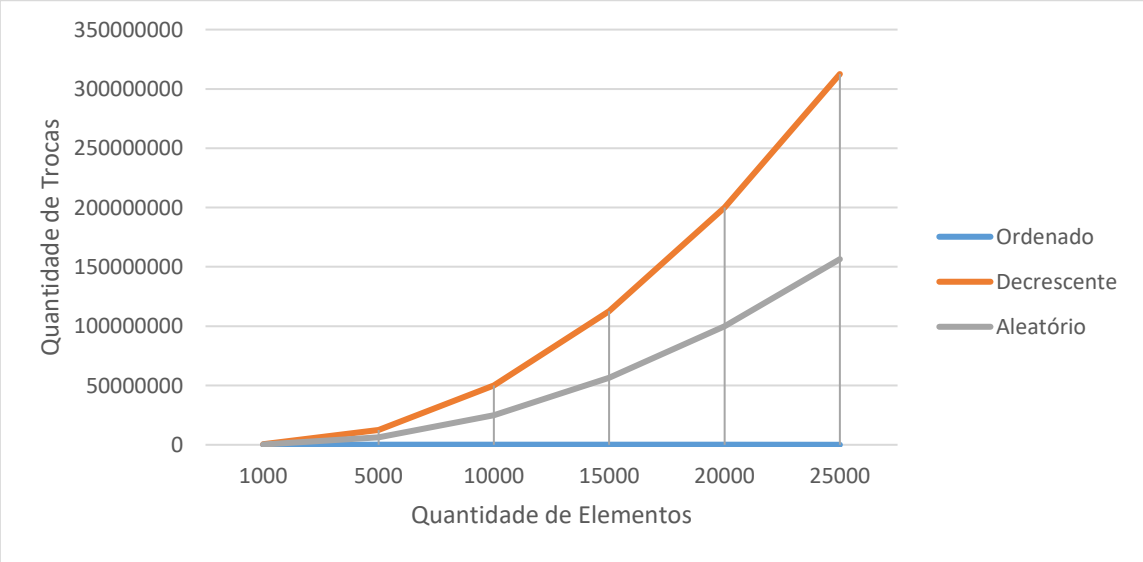

Figura 10 – Insert Sort (Quantidade de Trocas)

2.5 Shell Sort

O método *ShellSort* pode ser considerado o refinamento do método *Insertion Sort*. O que diferencia pelo fato de no lugar de considerar o vetor ordenado, como único segmento, considera vários segmentos sendo aplicado o algoritmo de *Insertion Sort* em cada um desses segmentos.

O *ShellSort* permite trocar de registros que estão distantes um do outro e o tempo de execução do algoritmo é sensível à ordem inicial do arquivo.

Melhor caso	Caso Médio	Pior Caso
O(n)	**O(n^2)**	**O(n^2)**

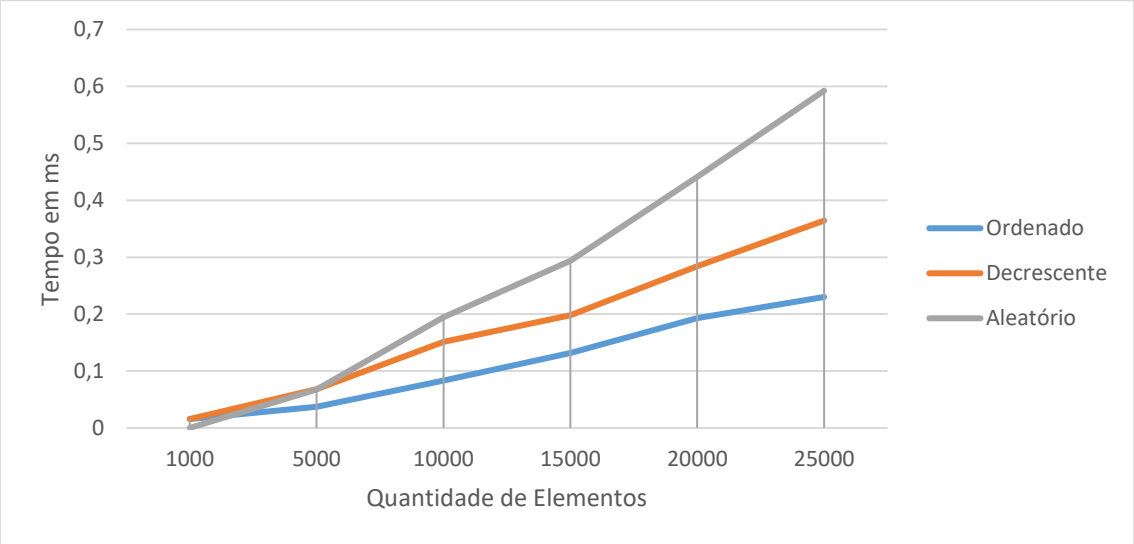

Figura 11 – Shell Sort (Tempo)

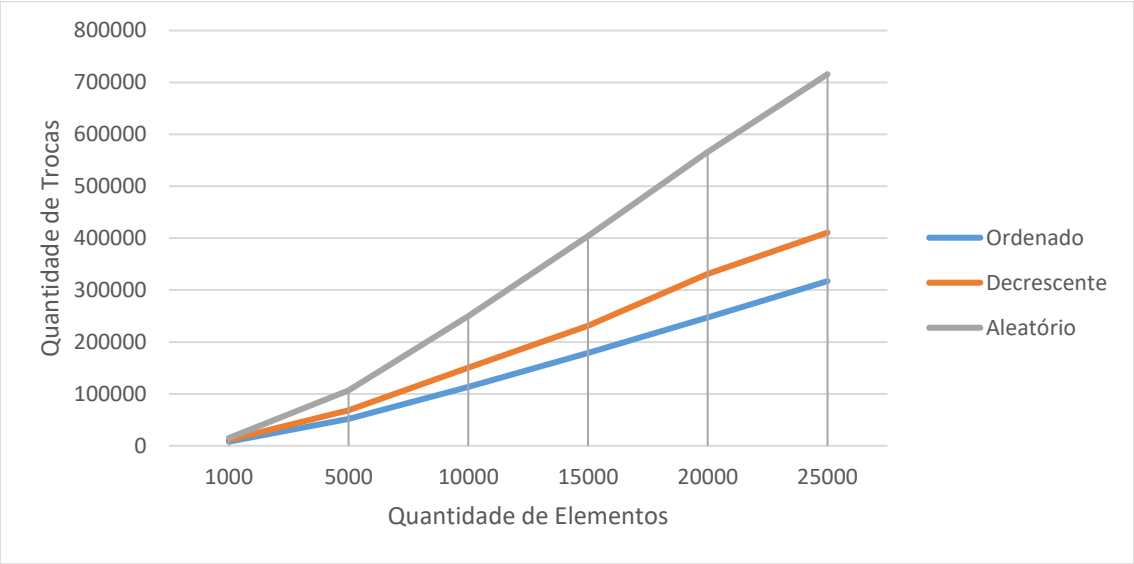

Figura 12 – Shell Sort (Quantidade de Trocas)

2.6 Selection Sort

O Selection Sort por sua vez tem a intenção de selecionar o menor item da lista e colocá-lo em sua posição correta, o algoritmo faz comparações de n – 1, n – 2 e assim por diante.

Melhor caso	Caso Médio	Pior Caso
$O(n^2)$	$O(n^2)$	$O(n^2)$

Figura 13 – Select Sort (Tempo)

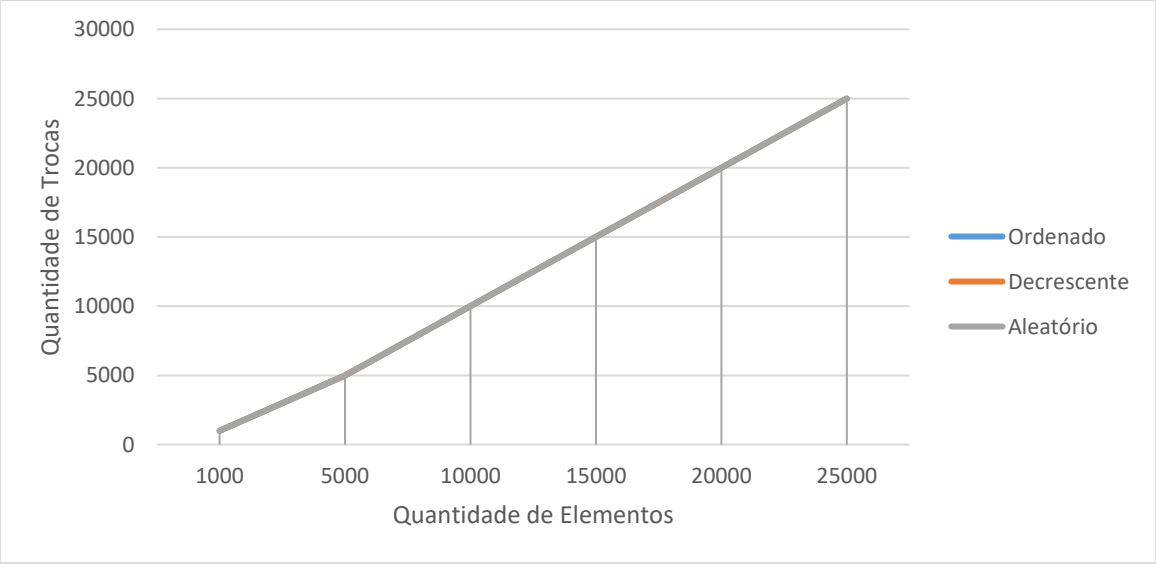

Figura 14 – Select Sort (Quantidade de Trocas)

2.7 Heap Sort

Utiliza metodologia de ordenação por seleção, criado por Robert W. Floyd em 1964 tem seus melhores resultados quando inseridos conjunto de dados ordenados de maneira aleatória.

Melhor caso	Caso Médio	Pior Caso
O(n log (n))	O(n log (n))	O(n log (n))

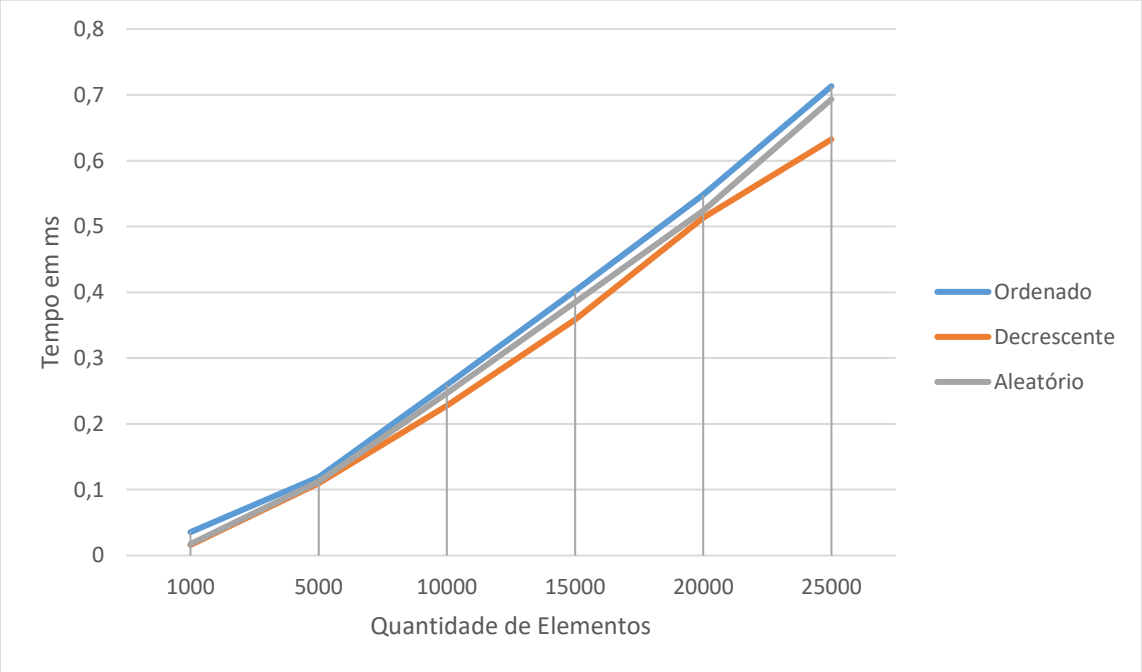

Figura 15 – Heap Sort (Tempo)

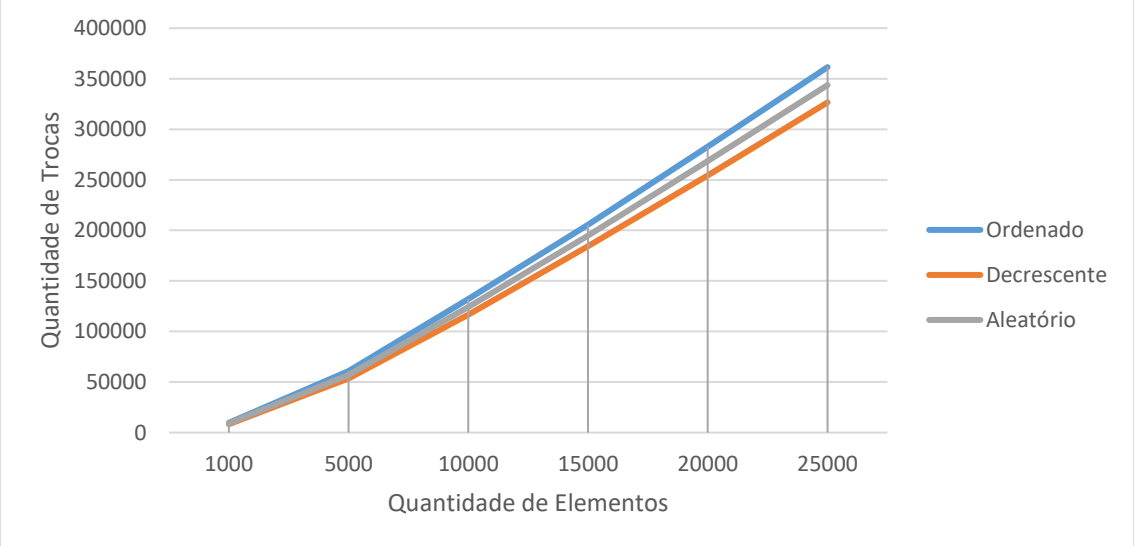

Figura 16 – Heap Sort (Quantidade de Trocas)

2.8 Merge Sort

O Merge Sort é outro algoritmo de ordenação que utiliza o método de divisão e conquista, divide a lista em sub-lista, realiza a ordenação das mesmas e executa a unificação destas, porém este algoritmo utiliza recursividade, o que demanda um consumo de processamento maior em comparação a outros.

Melhor caso	Caso Médio	Pior Caso
O(n log (n))	O(n log (n))	O(n log (n))

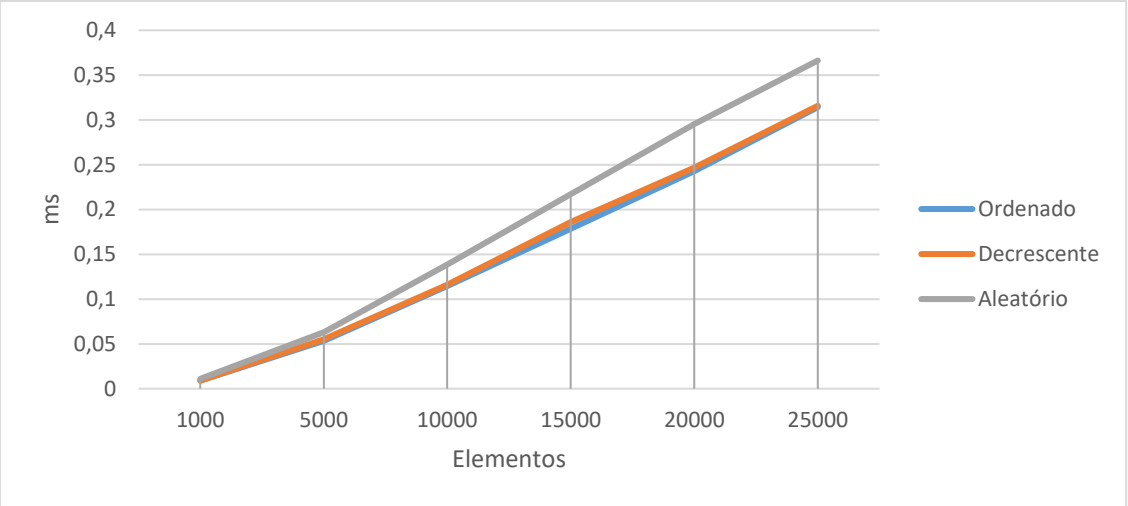

Figura 17 – Merge Sort (Tempo)

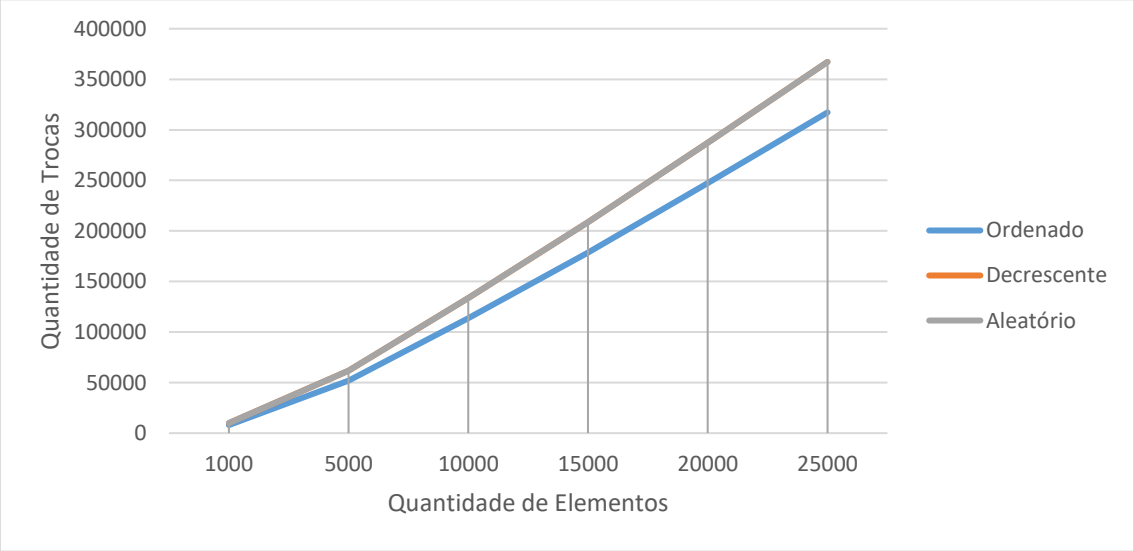

Figura 18 – Merge Sort (Quantidade de Trocas)

3. Comparações

Analisando gráfico a gráfico, é possível ver nitidamente a dificuldade que alguns algoritmos enfrentam conforme a entrada de dados cresce ou pelo menos muda de ordenação (crescente, aleatória e decrescente) e a diferença no processamento entre eles.

3.1 Ordenação Crescente

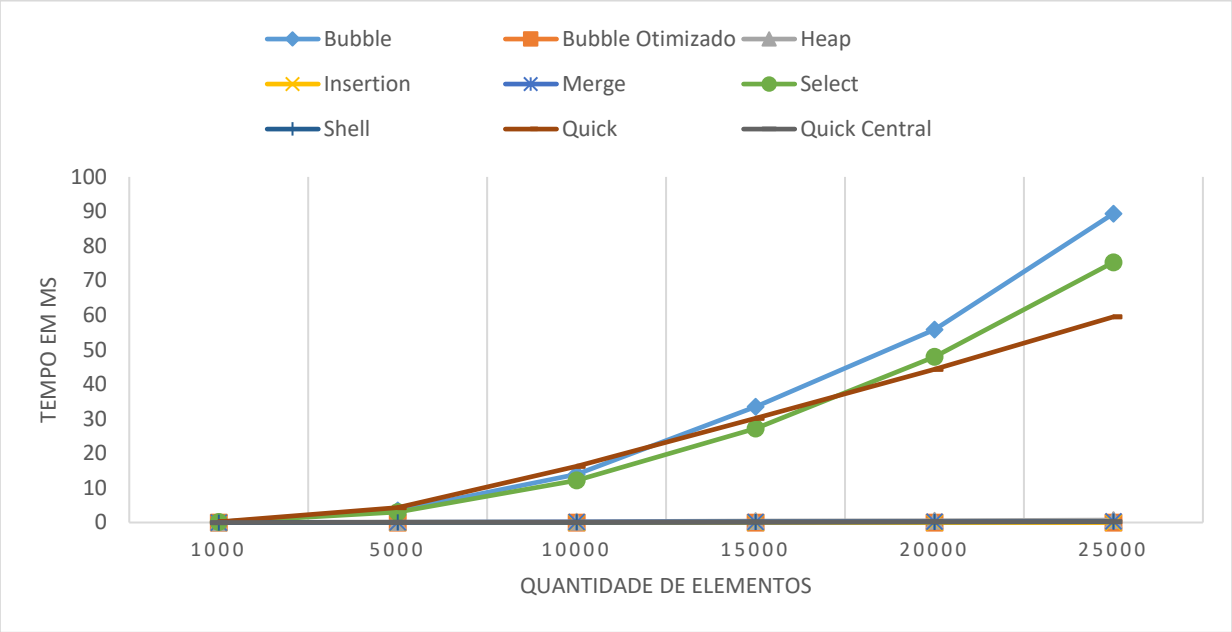

Figura 19 – Vetor Ordenado (Tempo)

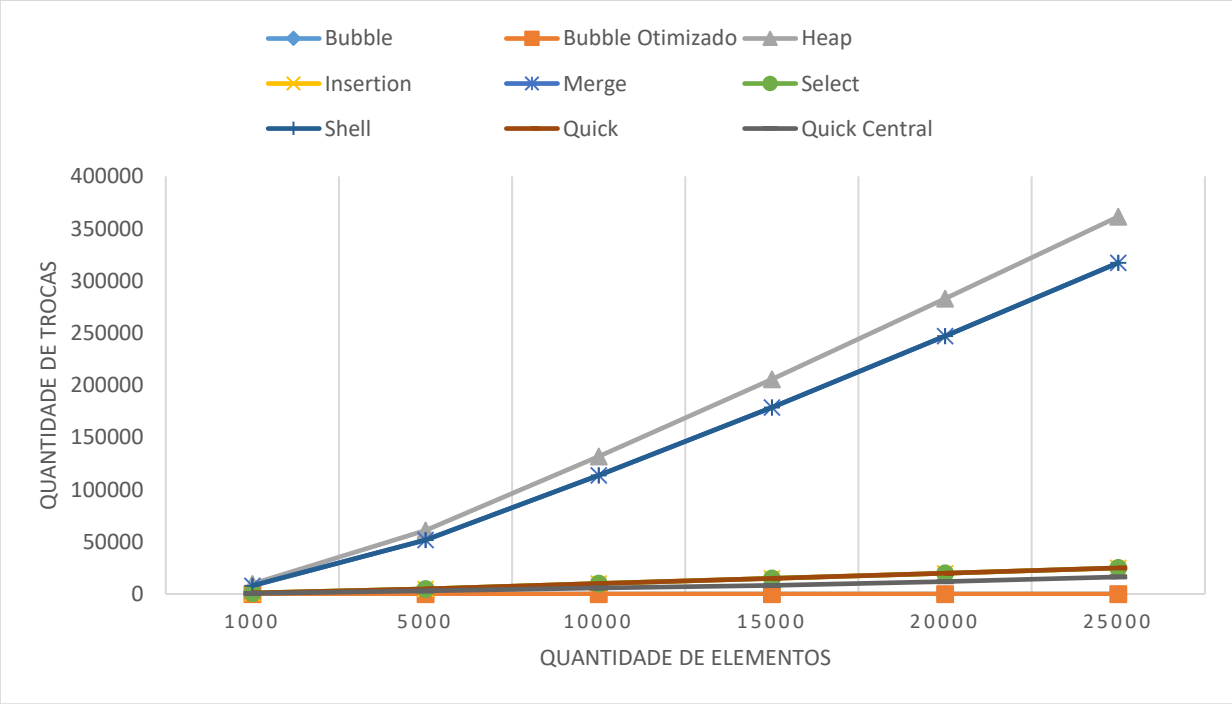

Figura 20 – Vetor Ordenado (Quantidade de Trocas)

Com vetores ordenados, Bubble Sort, Quick (pivô inicial), Selection tiveram os piores resultados de maneira discrepante com relação aos outros, os demais algoritmos mostraram-se eficientes neste cenário, o Bubble Sort otimizado quase não executou instruções.

3.2 Ordenação Decrescente

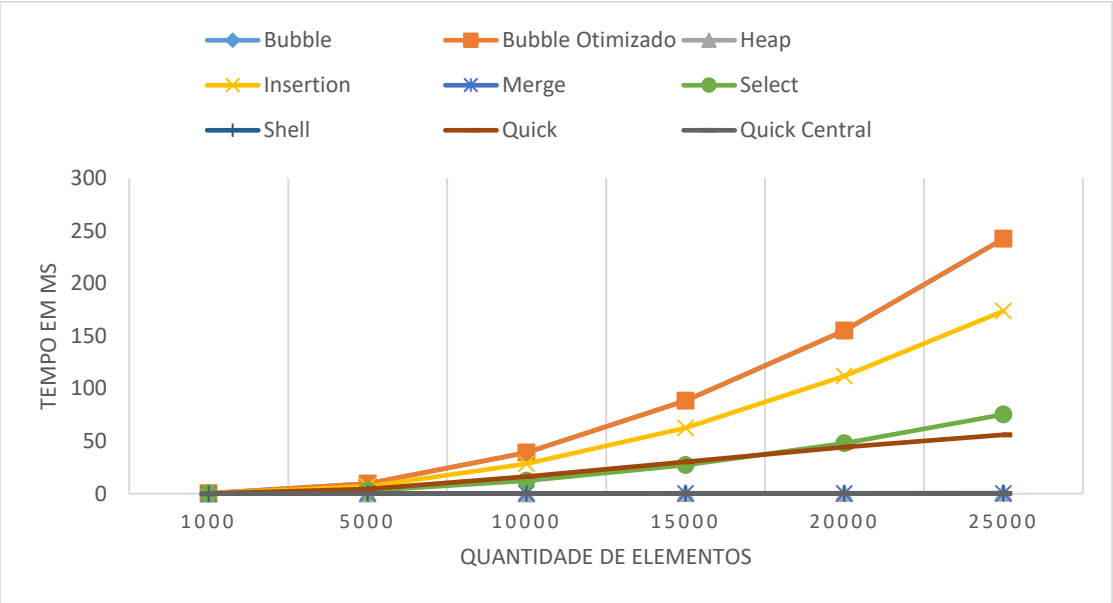

Figura 21 – Vetor Decrescente (Tempo)

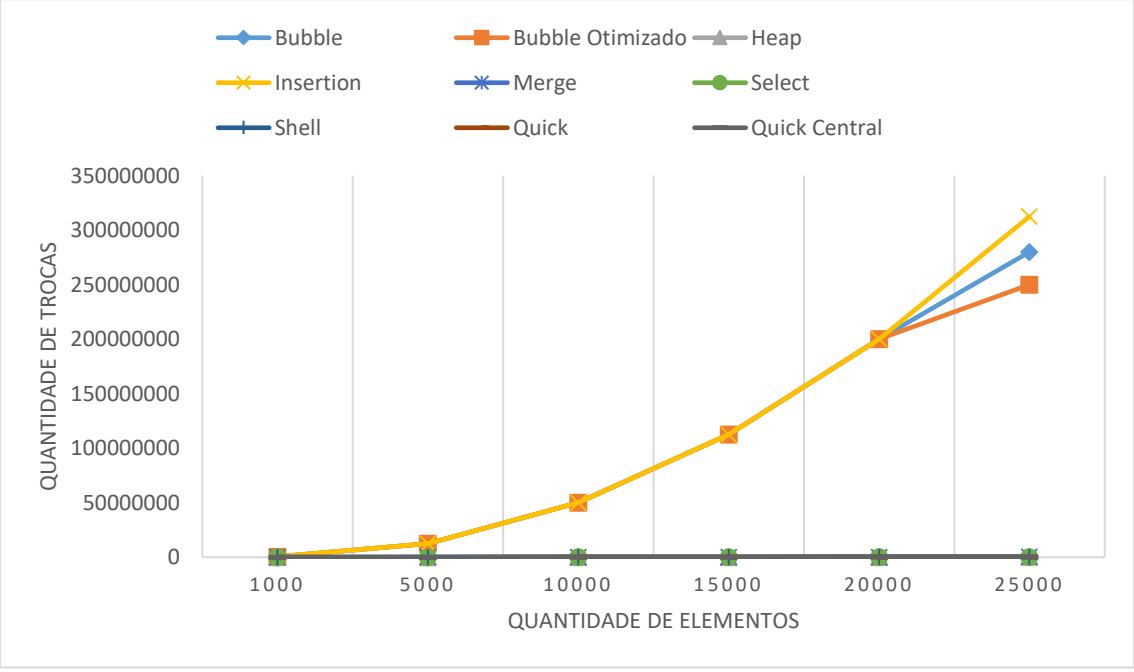

Figura 22 – Vetor Decrescente (Quantidade de Trocas)

Para os vetores que estão em ordem decrescente os algoritmos de Bubble Sort, Bubble Sort Otimizado e Insertion Sort, apresentam os piores resultados. Quick sort com pivô central demonstrou o melhor resultado.

3.2 Ordenação Aleatória

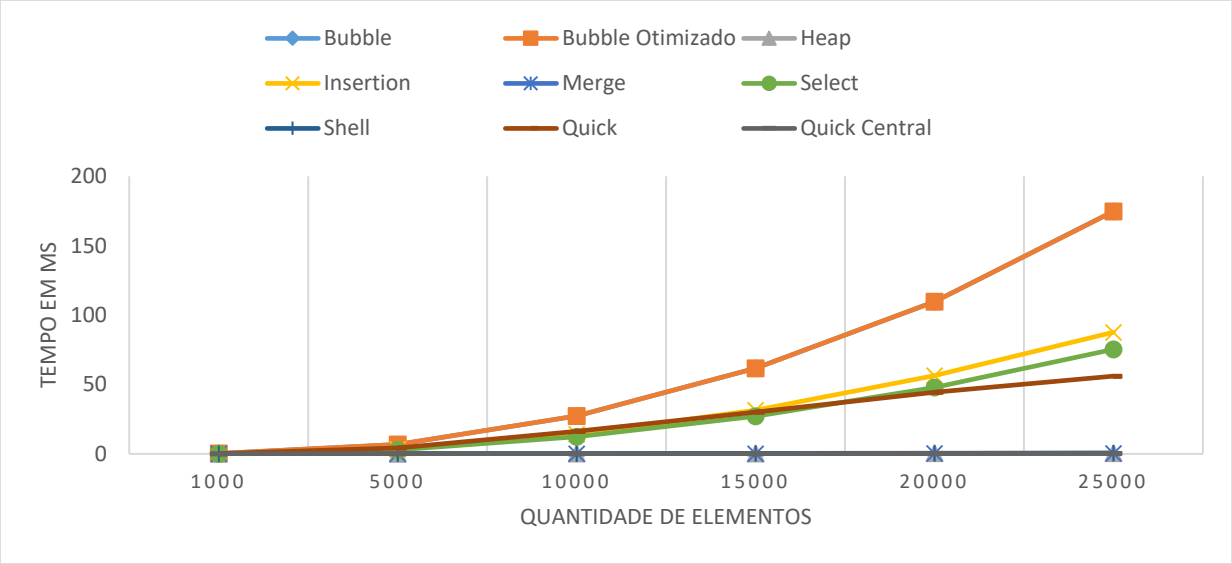

Figura 23 – Vetor Aleatório (Tempo)

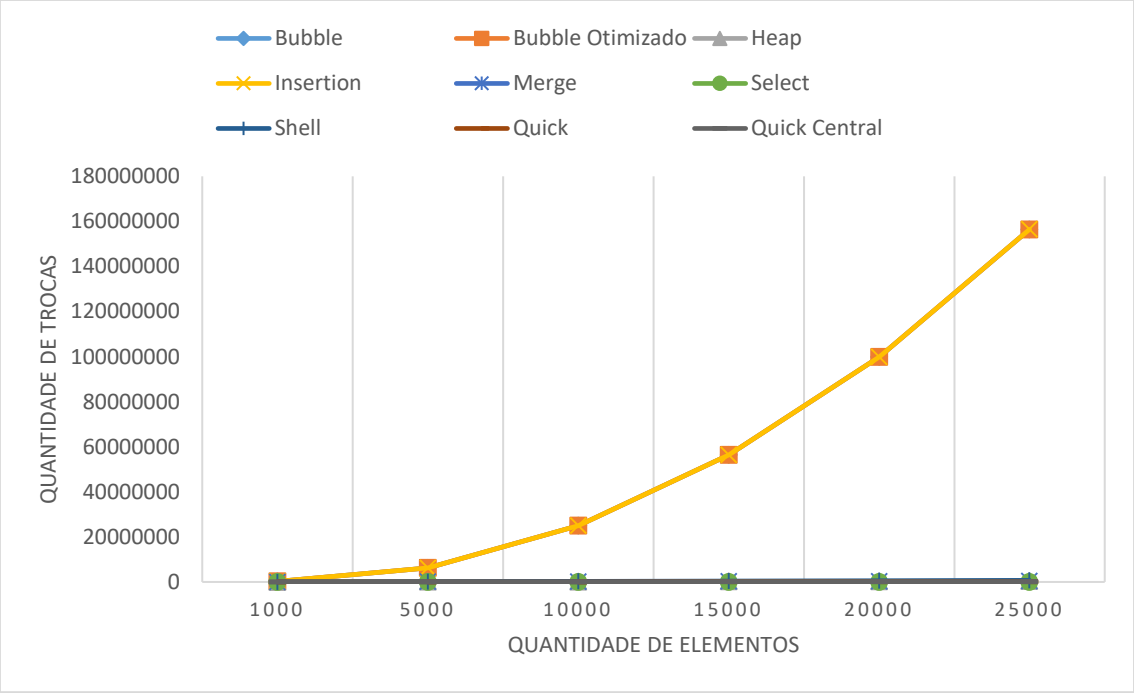

Figura 24 – Vetor Aleatório (Quantidade de Trocas)

A semelhança de processamento é quase a mesma em relação a vetores aleatórios e vetores decrescentes. Bubble Sort, Bubble Sort Otimizado, Insertion Sort, Select Sort e Quick Sort apresentaram os piores resultados.

4. Referências

[1] Materiais dados em aula.

[2] J. E. Souza, J. V. G. Ricarte, and N. C. de Almeida Lima. Algoritmos de ordenação: Um estudo comparativo. Anais do Encontro de Computação do Oeste Potiguar ECOP/UFERSA, 1, 2017.

[3] N. Ziviani et al. *Projeto de algoritmos: com implementações em Pascal e C*, volume 2. Thomson, 2004.

[4] Pratt, Vaughan R. *Shellsorg and Sorting Networks*. No. STAN-CS-72-260. STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE, 1972.