UNIVERSIDADE FEDERAL DO TOCANTINS
CÂMPUS UNIVERSITÁRIO DE PALMAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

RELATÓRIO DE PROJETO E ANÁLISE DE ALGORITMOS

ANALISANDO A COMPLEXIDADE DE ALGORITMOS DE ORDENAÇÃO

MATEUS ARAUJO DIAS
JOÃO PEDRO NORONHA DE MORAES SOUZA
BRUNO FERREIRA DA SILVA
GABRIEL BORGES ARAÚJO

PALMAS (TO)
2023

SUMÁRIO

1 INTRODUÇÃO

Tomando a quantidade de trocas e comparações entre valores, bem com o tempo de execução como métricas de desempenho em função do tamanho da entrada, o presente trabalho tem por objetivo expor e discutir os resultados de uma sequência de ordenação de vetores com tamanhos variados dos seguintes algoritmos: Bubble Sort, Selection Sort, Insertion Sort, Gnome Sort, Shell Sort, Heap Sort, Merge Sort e Quick Sort. Explorando, assim, suas vantagens e desvantagens e visando elucidar seus comportamentos na prática.

2 MÉTODOS

2.1 Geração dos Vetores

Para a criação e utilização dos vetores pelos algoritmos de ordenação desenvolveu-se um script em Python capaz de gerar números de 0 a 99 em ordem crescente, decrescente e aleatória, bem como de salvá-los em arquivos .json possuindo os seguintes tamanhos: 1×10^3, 1×10^4, 1×10^5, 2×10^5, 3×10^5, 4×10^5, 5×10^5, 6×10^5, 7×10^5, 8×10^5, 9×10^5 e 1×10^6.

3 RESULTADOS

3.1 Introdução aos algoritmos

3.1.1 Bubble Sort

Seu funcionamento consiste em percorrer o vetor repetidamente, comparando pares adjacentes de elementos e trocando-os se estiverem na ordem errada. Repitindo o processo até que o vetor esteja totalmente ordenado.

3.1.2 Selection Sort

Sua ideia básica centra-se em encontrar o elemento mínimo na parte não ordenada do vetor e trocá-lo pelo primeiro elemento não ordenado. Repitindo o processo até que o vetor esteja totalmente ordenado.

3.1.3 Insertion Sort

Sua ideia básica é construir uma subarray ordenada da esquerda para a direita inserindo cada novo elemento em sua posição correta na subarray. Repita até que o vetor esteja totalmente ordenado.

3.1.4 Shell Sort

O Shell Sort funciona dividindo a lista em subgrupos menores com base em um incremento chamado de lacuna (gap). A lacuna é definida inicialmente como metade do tamanho da lista, e os elementos que estão separados por essa lacuna são comparados e trocados, se necessário. Em seguida, a lacuna é reduzida e o processo é repetido até que a lacuna seja igual a 1.

A ideia por trás do Shell Sort é realizar trocas entre elementos distantes um do outro, para tentar posicionar elementos próximos um do outro na lista, a fim de reduzir o número total de comparações e trocas necessárias. Isso é diferente do algoritmo de inserção direta, que compara e troca elementos adjacentes.

3.1.5 Gnome Sort

A ideia básica por trás do Gnome Sort é percorrer uma lista de elementos a serem ordenados e, a cada passo, comparar o elemento atual com o anterior. Se a ordem estiver correta, o algoritmo avança para o próximo elemento. Caso contrário, ele troca os dois elementos de posição e retrocede um passo para verificar se o elemento anterior também está na ordem correta.

3.1.6 Heap Sort

O Heap Sort é um algoritmo de classificação (ou ordenação) baseado na estrutura de dados chamada "heap"(ou "árvore binária de mínimo/máximo"). Ele recebe esse nome porque utiliza essa estrutura para realizar a ordenação dos elementos.

3.1.7 Merge Sort

A ideia básica por trás do Merge Sort é dividir a lista não ordenada em pequenos subgrupos, ordenar cada subgrupo separadamente e, em seguida, mesclar os subgrupos ordenados para obter a lista finalmente ordenada.

3.1.8 Quick sort

A ideia básica por trás do Quick Sort é dividir a lista em subgrupos menores com base em um elemento pivô e, em seguida, recursivamente ordenar esses subgrupos. O pivô é escolhido de forma que, ao final do processo, ele esteja em sua posição correta na lista ordenada. Isso é chamado de "particionamento".

3.2 Comparações de Desempenho

3.2.1 Bubble Sort, Selection Sort, Insertion Sort, Gnome Sort e Shell Sort

O Bubble Sort possui complexidade de tempo On^2 nos piores e médios casos, $O(n)$ no melhor caso (quando o array de entrada já está ordenado) Complexidade do espaço: O(1).

Suas vantagens incluem implementação simples, funciona bem para pequenos conjuntos de dados, requer apenas espaço constante, algoritmo de classificação estável

Suas desvantagens consistem em sua ineficiente para grandes conjuntos de dados, complexidade de tempo de pior caso de $O(n^2)$, e em não ser ideal para conjuntos de dados parcialmente classificados

Selection Sort e Insertion Sort têm a mesma complexidade de espaço de $O(1)$, enquanto o Bubble Sort também tem uma complexidade de espaço de $O(1)$.

O Selection Sort é mais rápida em comparação com o Bubble Sort.

O Selection Sort possui complexidade de tempo $O(n^2)$ em todos os casos (pior, médio e melhor) e sua complexidade do espaço: $O(1)$.

Como vantagens possui implementação simples, funciona bem para pequenos conjuntos de dados, requer apenas espaço constante e é um algoritmo de classificação no local. Como desvantagens está o fato de ser ineficiente para grandes conjuntos de dados, complexidade de tempo de pior caso de $O(n^2)$, não ser ideal para conjuntos de dados parcialmente classificados e não ser um algoritmo de classificação estável

Bubble Sort e Insertion Sort são algoritmos de classificação estáveis, o que significa que eles preservam a ordem relativa de elementos iguais no vetor ordenado, enquanto o Selection Sort não é estável.

Em termos de desempenho, o Insertion Sort tende a ter um desempenho melhor do que o Bubble Sort e o Selection Sort para conjuntos de dados pequenos, enquanto o Bubble Sort e o Selection Sort podem ter um desempenho melhor do que o Insertion Sort para conjuntos de dados maiores ou conjuntos de dados parcialmente classificados.

Basicamente o Gnome Sort é uma variação do Insertion Sort. Enquanto o Insertion Sort percorre um vetor inteiro de números inteiros e coloca cada elemento em sua posição adequada, o Gnome tenta ser mais eficiente e faz a mesma coisa, mas adiciona um loop de volta quando ocorre uma troca, salvando uma iteração.

O Gnome Sort executa pelo menos tantas comparações quanto o Insertion Sort e tem as mesmas características de tempo de execução assintóticas.

Shell sort é um algoritmo de ordenação altamente eficiente e baseado no algoritmo Insertion Sort. Esse algoritmo evita grandes deslocamentos, como no Insertion Sort, onde o menor valor está na extrema direita e deve ser movido para a extrema esquerda. O Shell Sort reduz sua complexidade de tempo, utilizando o fato de que usar o Insertion Sort em um vetor parcialmente classificada resulta em menos movimentos.

3.2.1.1 Gráficos

Figura 1 – Tempo de execução - Bubble Sort

n	Swaps unsorted	Swaps descending	Comparisons unsorted	Comparisons descending
...	...	...	...	...
128	8,050	16,256	8,136	8,255
256	31,854	65,280	32,893	32,895
512	128,340	261,632	130,767	131,327
1,024	528,004	1,047,552	524,475	524,799
2,048	2,111,760	4,192,256	2,097,546	2,098,175
...	...	...	...	...

Figura 2 – Trocas e comparações - Bubble Sort

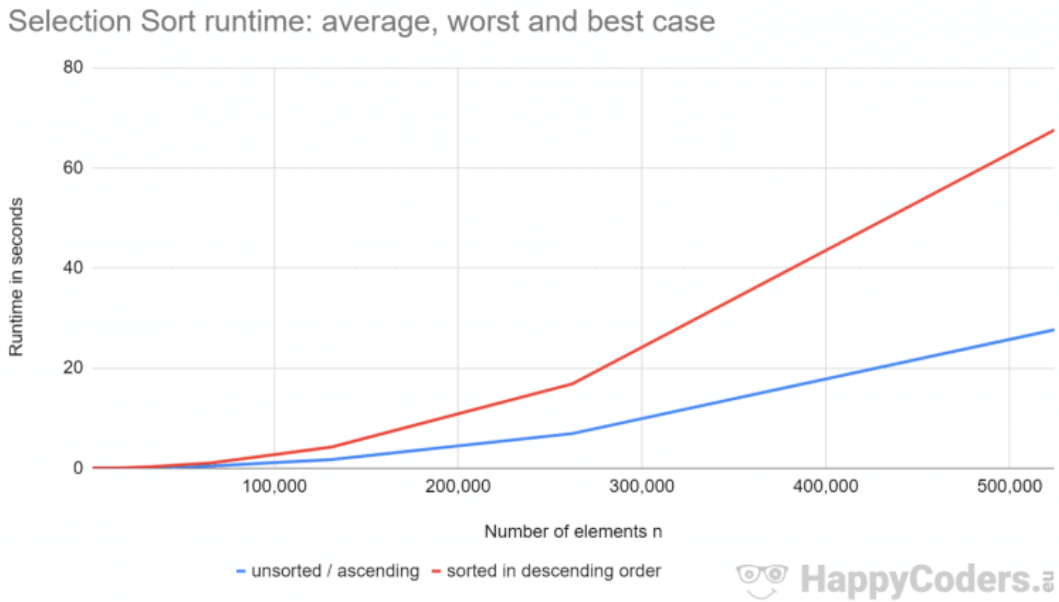

Figura 3 – Tempo de Execução - Selection Sort

n	Comparisons	Swaps unsorted	Swaps descending	minPos/min unsorted	minPos/min descending
...	...	...	...	...	...
512	130.816	504	256	2.866	66.047
1.024	523.776	1.017	512	6.439	263.167
2.048	2.096.128	2.042	1.024	14.727	1.050.623
4.096	8.386.560	4.084	2.048	30.758	4.198.399
8.192	33.550.336	8.181	4.096	69.378	16.785.407

Figura 4 – Trocas e comparações - Selection Sort

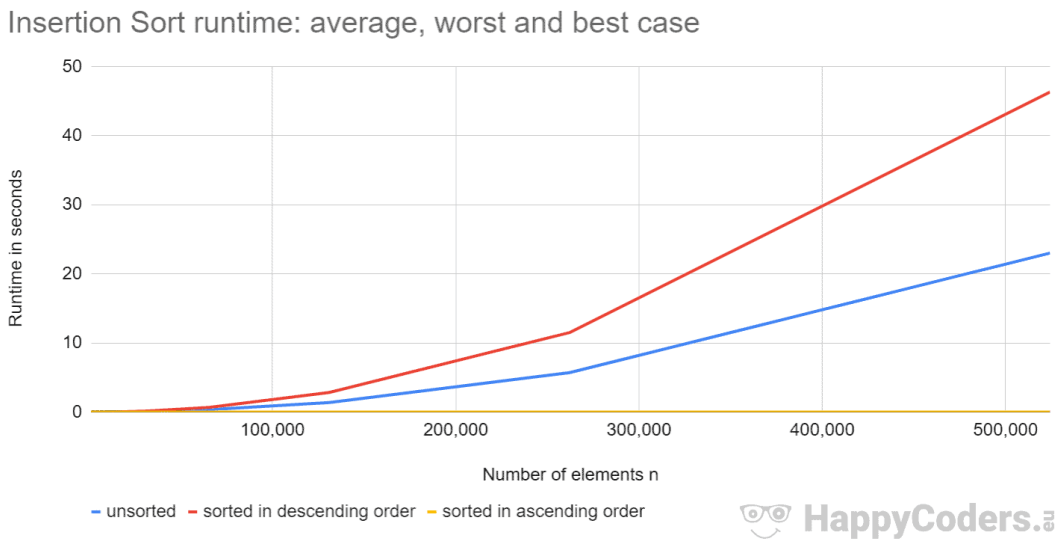

Figura 5 – Tempo de Execução - Insertion Sort

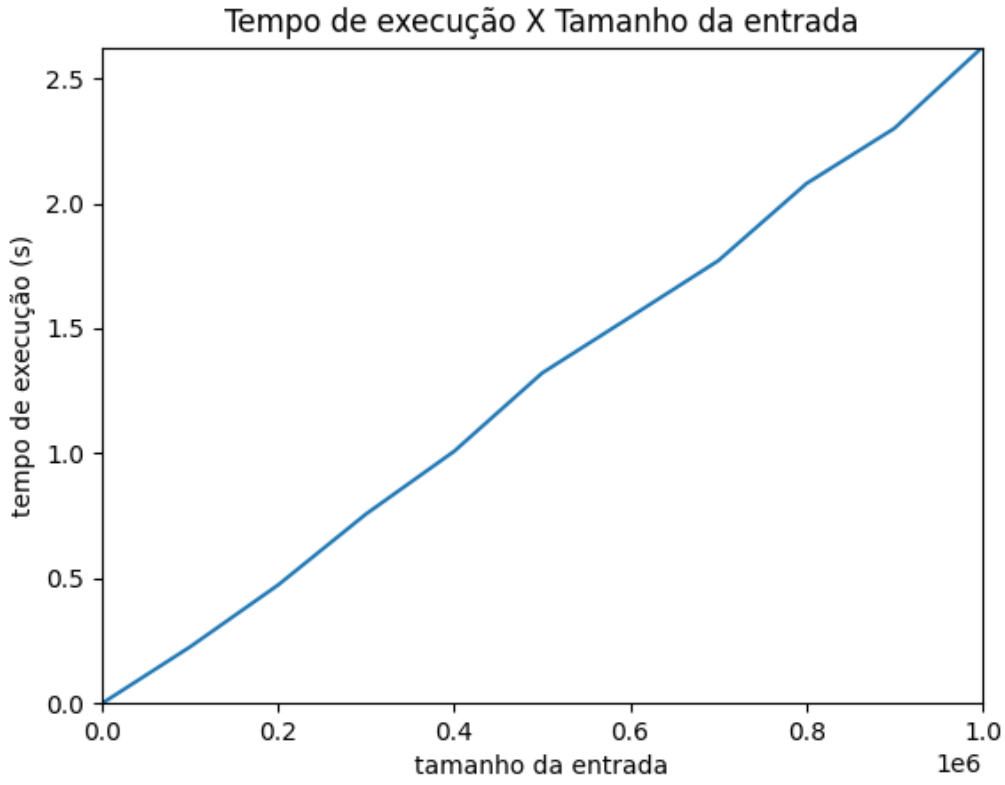

Figura 6 – Tempo de Execução(Caso médio) - Shell Sort

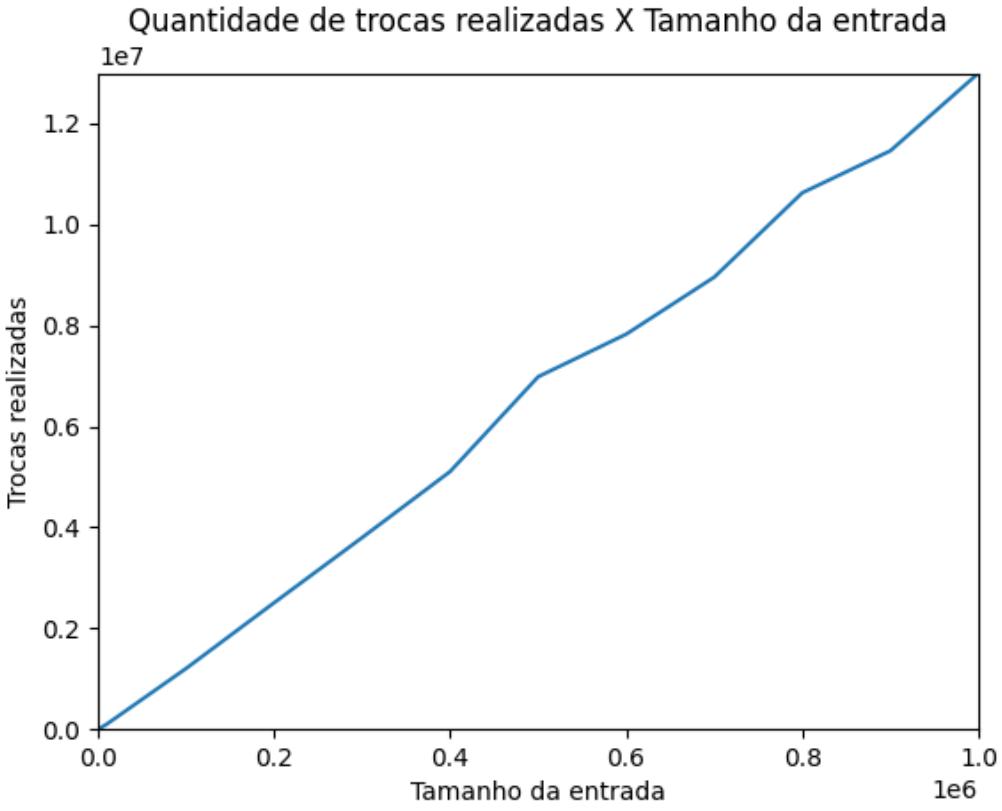

Figura 7 – Trocas (Caso médio) - Shell Sort

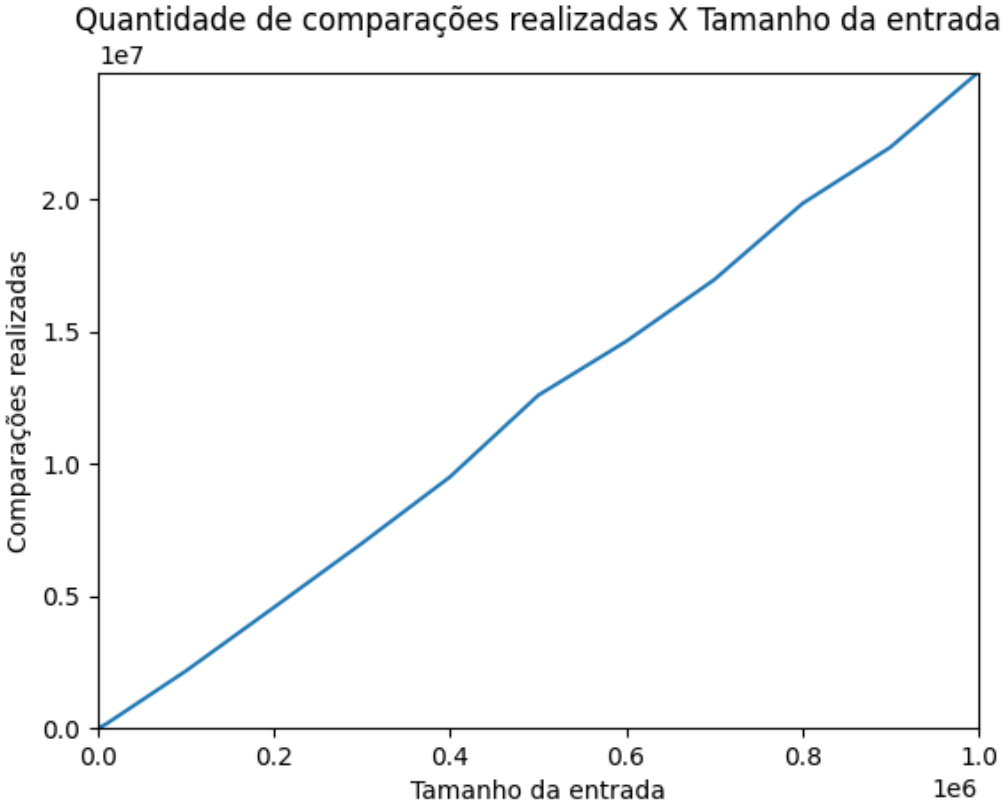

Figura 8 – Comparações (Caso médio) - Shell Sort

3.2.2 Merge Sort, Heap Sort e Quick Sort

As diferênças entre Merge Sort e Quick Sort consistem em:

Partição de elementos no vetor: No merge sort, o vetor é dividida em apenas 2 metades (ou seja, n/2). considerando que, no caso do Quick Sort, o vetor é dividida em qualquer proporção. Não há compulsão de dividir o vetor de elementos em partes iguais no Quick Sort.

Complexidade de pior caso: A complexidade de pior caso do Quick Sort é $O(n^2)$, pois há necessidade de muitas comparações na pior condição. Enquanto que no Merge Sort, o pior caso e o caso médio têm as mesmas complexidades $O(nlogn)$.

Uso com conjuntos de dados: Merge Sort pode funcionar bem em qualquer tipo de conjunto de dados, independentemente de seu tamanho (grande ou pequeno). Enquanto que o Quick Sort não pode funcionar bem com grandes conjuntos de dados.

Requisito de espaço de armazenamento adicional: Merge Sort não é in-place porque requer espaço de memória adicional para armazenar os vetores auxiliares. Enquanto o Quick Sort é in-place, pois não requer nenhum armazenamento adicional.

Eficiência: Merge Sort é mais eficiente e funciona mais rápido do que o Quick Sort no caso de conjuntos de dados ou vetores maiores. Enquanto o Quick Sort é mais eficiente e funciona mais rápido que o Merge Sort no caso de tamanho de vetores ou conjuntos de dados menores.

Método de classificação: Quick Sort é um método interno de classificação em que os dados são classificados na memória principal. Enquanto que o merge sort é um método de classificação externo no qual os dados a serem classificados não podem ser acomodados na memória e é necessária memória auxiliar para classificação.

Estabilidade: Merge Sort é estável, pois dois elementos com valor igual aparecem na mesma ordem na saída ordenada como estavam no vetor não ordenado de entrada. Enquanto o Quick Sort é instável neste cenário. Mas pode se tornar estável usando algumas mudanças no código.

Preferido para: O Quick Sort é preferido para vetores. Enquanto que o Merge Sort é preferido para listas encadeadas.

Localidade de referência: Quick Sort exibe boa localidade de cache e isso torna o Quick Sort mais rápido do que o Merge sort (em muitos casos, como no ambiente de memória virtual).

Para efeito de comparação entre Quick Sort, Insertion Sort e Selection Sort, uma vez que o Quick Sort funciona a partir da posição e possui tempo de execução no pior caso tão ruim quanto o do Insert Sort e Selection Sort :(n^2) e ainda um tempo de execução médio tão bom quanto o do merge sort: $(nlog2n)$ temos que mesmo Merge Sort sendo no mínimo tão bom quanto o quick sort, devido ao fator constante oculto na notação big theta ser muito bom, na prática o Merge Sort supera significativamente o Selection Sort

e o Insertion Sort.

Na comparação entre Heap Sort e Quick Sort, o primeiro é um algoritmo de ordenação eficiente e instável com uma complexidade de tempo média, melhor e pior caso de $O(nlogn)$. É também significativamente mais lento que o Quick Sort e o Merge Sort, portanto, o Heapsort é menos comumente encontrado na prática.

O Heapsort é mais lento que o Quick Sort e mais lento que o Merge Sort para dados de entrada distribuídos aleatoriamente. Para dados ordenados, o Heap Sort é oito a nove vezes mais lento que o Quick Sort e duas vezes mais lento que o Merge Sort.

Na comparação Heap Sort vs. Quick Sort o Quicksort geralmente é muito mais rápido que o heapsort.

Devido à complexidade de tempo de pior caso $O(n^2)$ do Quick Sort, o Heap Sort às vezes é preferido ao Quicksort na prática. Para o Quick Sort, se o elemento pivô for escolhido de forma adequada, é improvável que o pior caso ocorra. No entanto, existe um certo risco de que um invasor em potencial com conhecimento suficiente da implementação do Quick Sort usada possa explorar esse conhecimento para travar ou congelar um aplicativo com dados de entrada preparados adequadamente.

Na análise de desempenho entre Heapsort vs. Merge Sort o Merge Sort também costuma ser mais rápido que o Heapsort. Além disso, ao contrário do Heapsort, o Merge Sort é estável.

O Heapsort tem uma vantagem sobre o Merge Sort porque não requer memória adicional, enquanto o Merge Sort requer memória adicional na ordem de $O(n)$.

Na comparação entre Heap Sort e Merge Sort, o segundo algoritmo é um pouco mais rápida do que a primeira para conjuntos maiores, mas requer o dobro da memória do Heap Sort devido à segunda array.

3.2.2.1 Gráficos

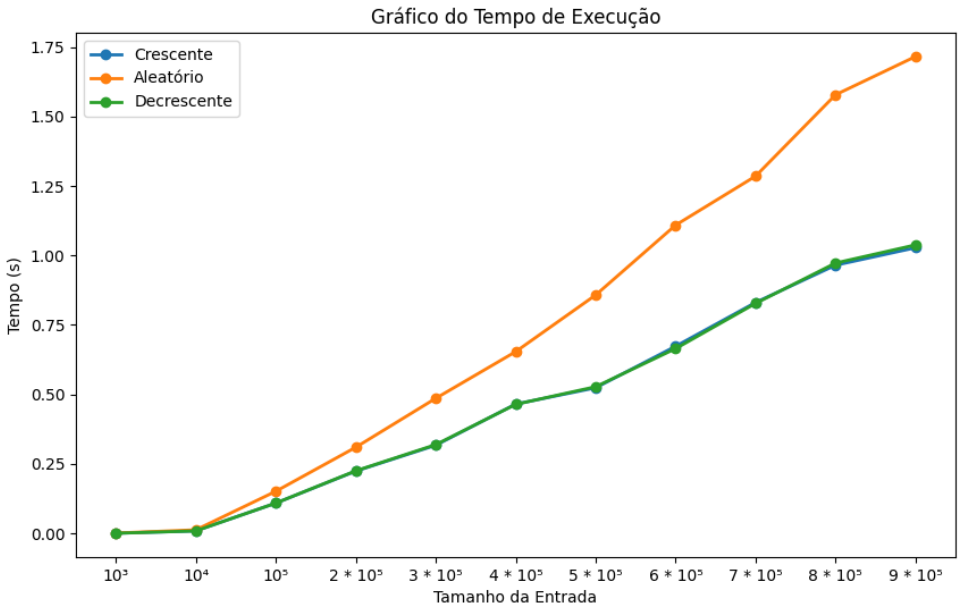

Figura 9 – Tempo de Execução - Quick Sort

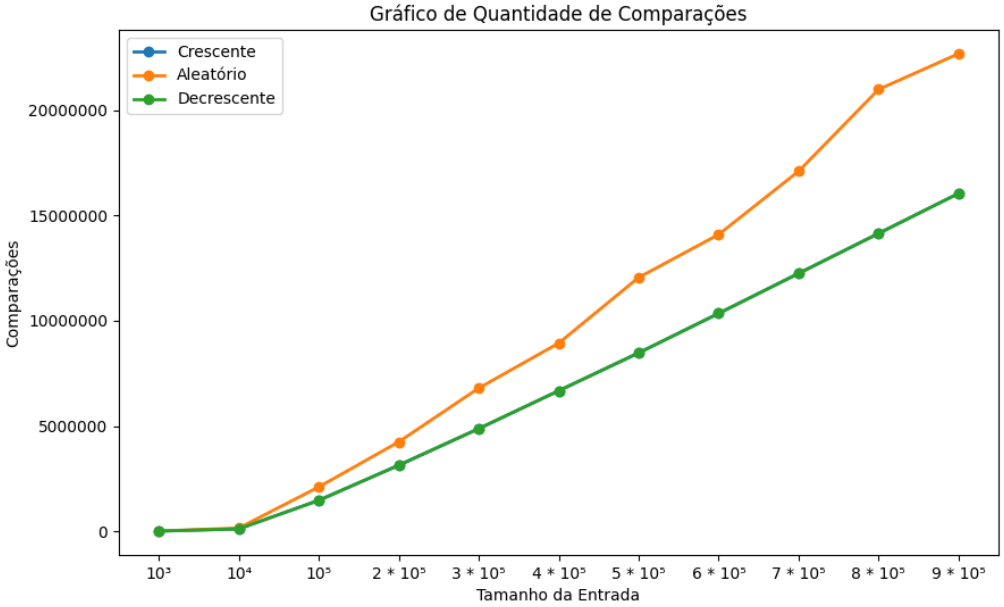

Figura 10 – Comparações - Quick Sort

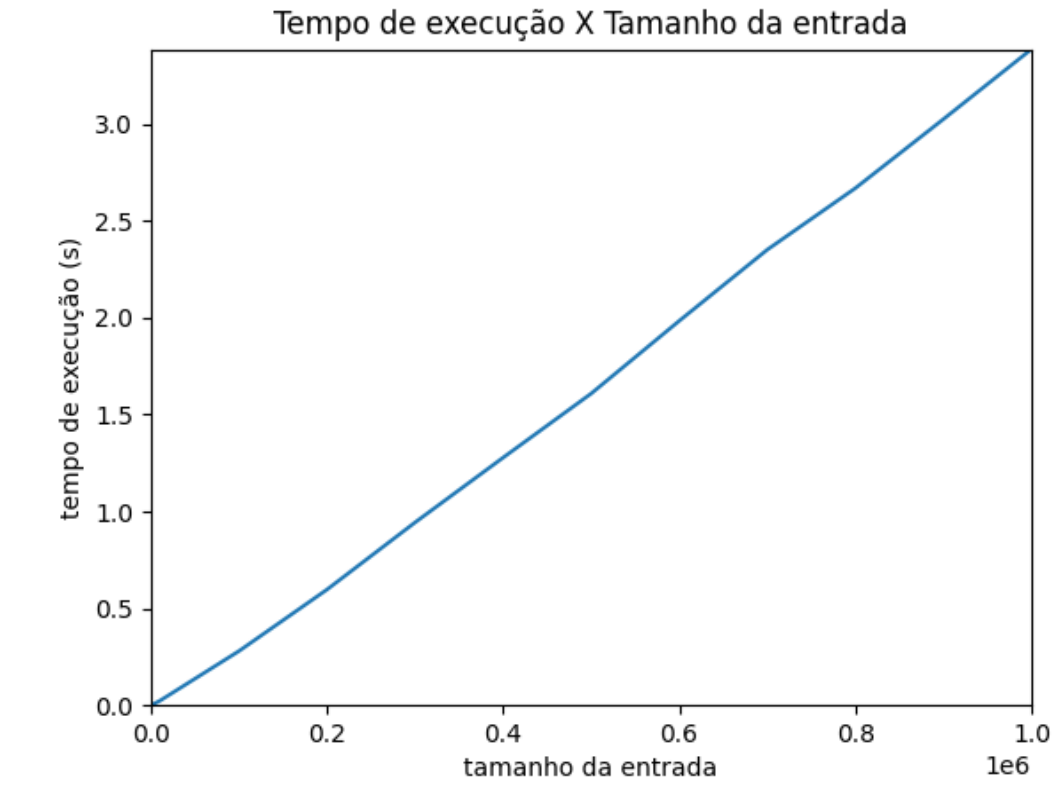

Figura 11 – Tempo de Execução(Caso médio) - Merge Sort

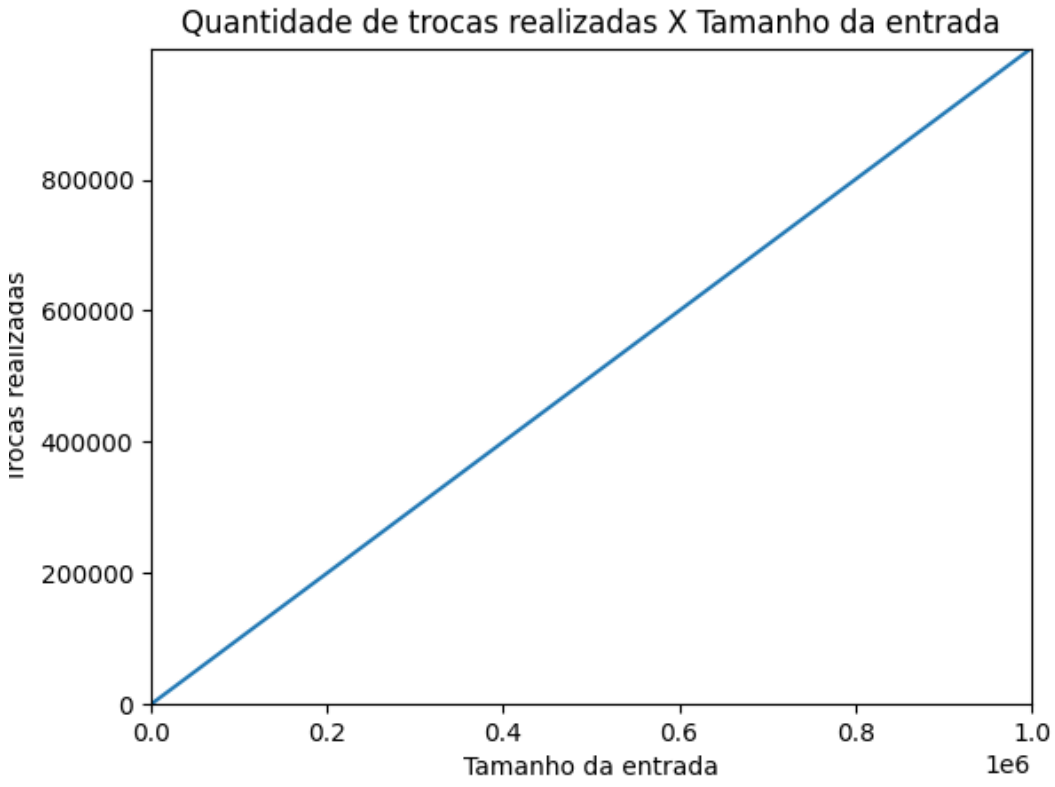

Figura 12 – Trocas(Caso médio) - Merge Sort

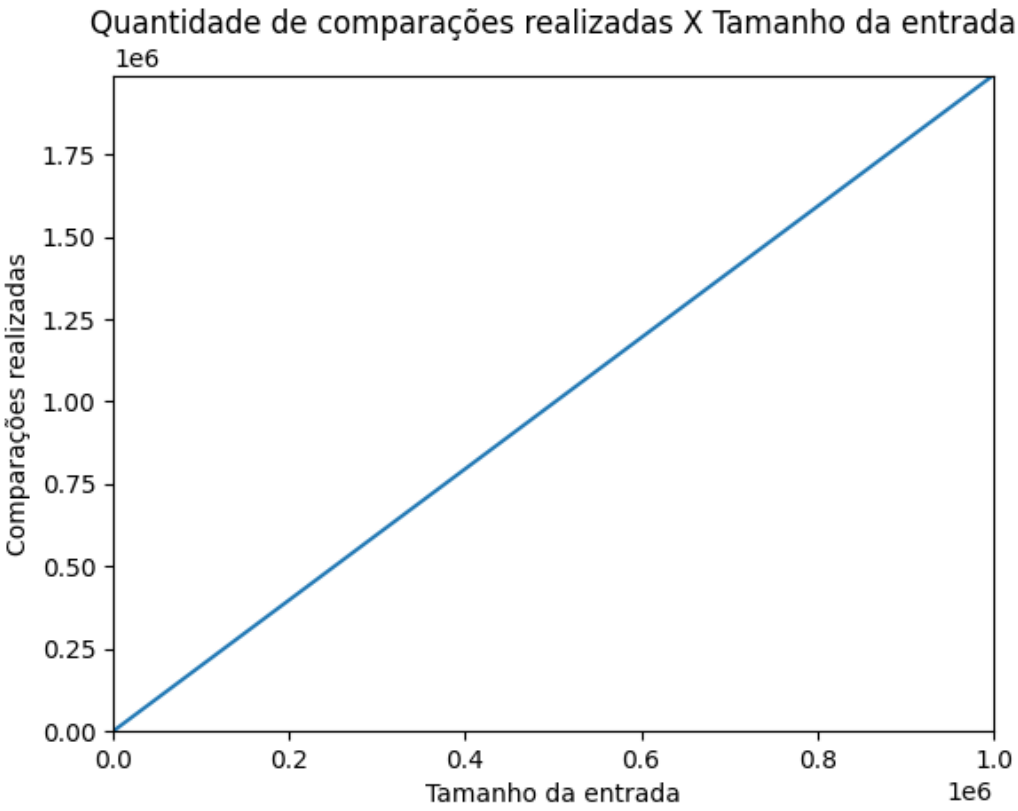

Figura 13 – Comparações(Caso médio) - Merge Sort

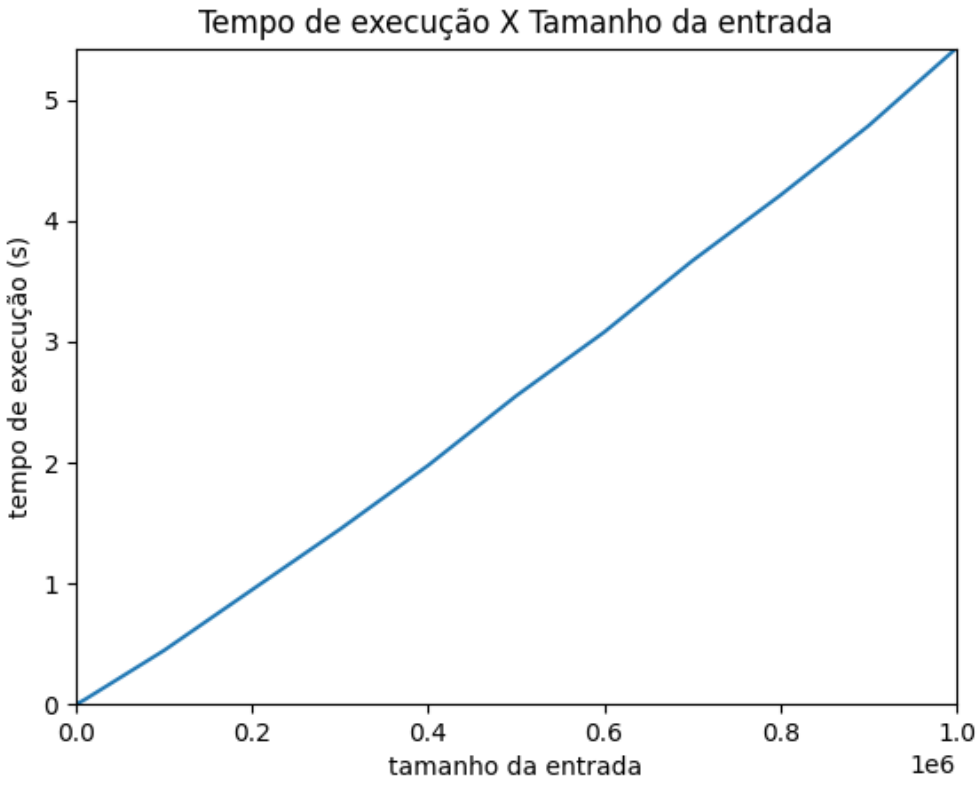

Figura 14 – Tempo de Execução(Caso médio) - Heap Sort

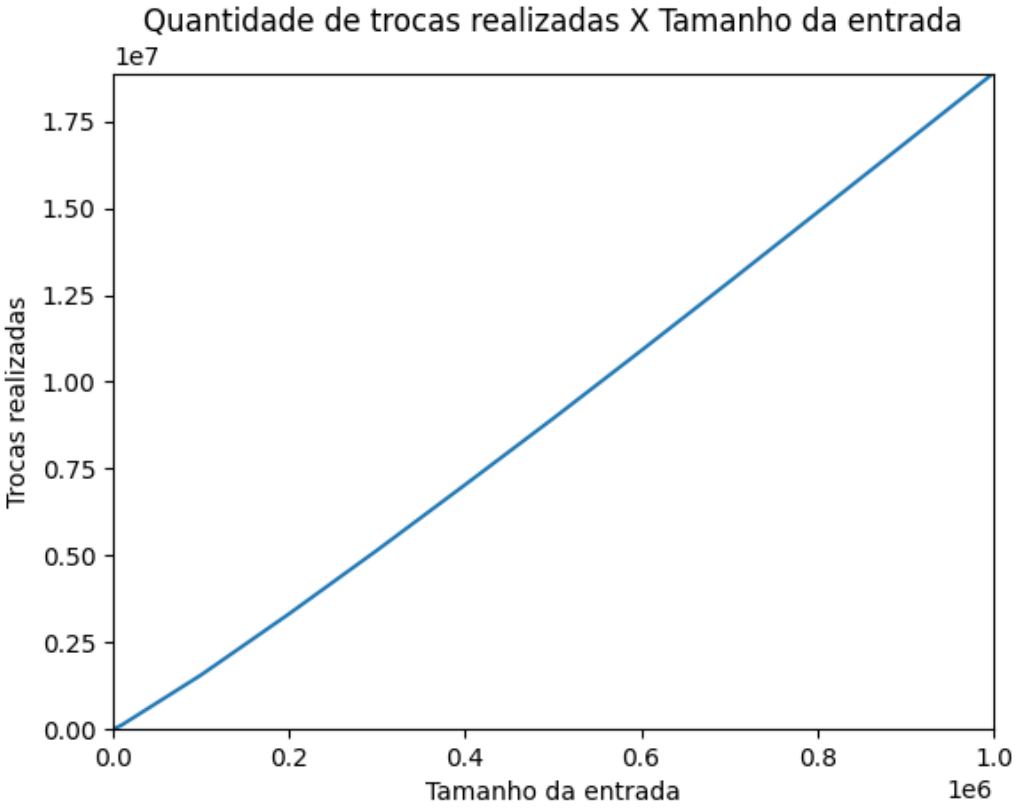

Figura 15 – Trocas(Caso médio) - Heap Sort

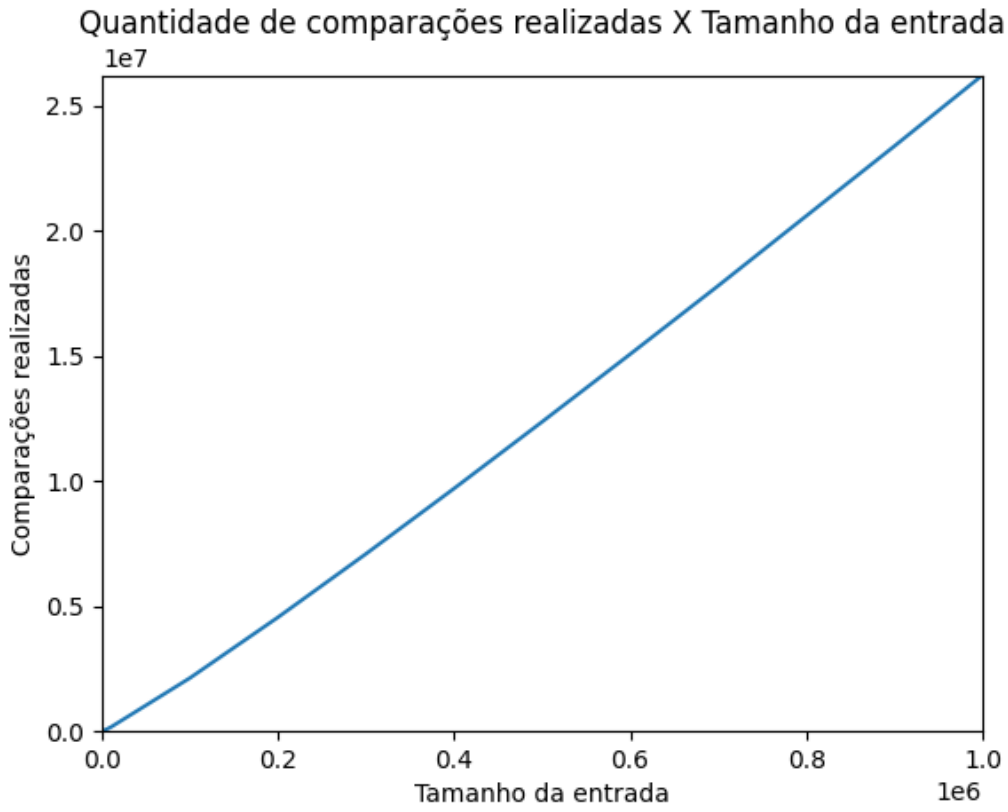

Figura 16 – Comparações(Caso médio) - Heap Sort

4 CONCLUSÃO

Existem diversos algoritmos já estudados que realizam a operação de ordenação. Todos esses utilizam metodologias variadas, com objetivos diferentes. Sendo assim podemos imaginar cenários distindos onde algum algoritmo seja melhor para a determinada situação.

Para ensino, por exemplo, o Bubble Sort e o Gnome Sort podem ser muito úteis. São fáceis de se explicar e ótimos para fazer a introdução a esse tipo de algoritmo. Para grandes valores o Heap Sort pode ser mais eficiente.

Por fim, cabe ao programador decidir qual o melhor algoritmo para o caso específico.

5 REFERÊNCIA BIBLIOGRÁFICA

MDAMIRCODER. **Comparison among Bubble Sort, Selection Sort and Insertion Sort**. 2023. Disponível em: https://www.geeksforgeeks.org/comparison-among-bubble-sort-selection-sort-and-insertion-sort/. Acesso em: 30 maio 2023.

WIKIPEDIA, a enciclopédia livre. **Comb sort**. 2023. Disponível em: https://en.wikipedia.org/wiki/Comb$_s ort. Acessoem$: 01jun.2023.

TAHIR, Hamza. **Difference between bubble sort and gnome sort**. 2012. Disponível em: https://stackoverflow.com/questions/9552224/difference-between-bubble-sort-and-gnome-sort. Acesso em: 29 maio 2023.

ANKIT_BISHT. **Quick Sort vs Merge Sort**. 2023. Disponível em: https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/. Acesso em: 29 maio 2023.

CORMEN, Thomas; BALKCOM, Devin. **Visão geral do quicksort**. Disponível em: https://pt.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort. Acesso em: 28 maio 2023.

WOLTMANN, Sven. **Heapsort – Algorithm, Source Code, Time Complexity**. 2020. Disponível em: https://www.happycoders.eu/algorithms/heapsort/. Acesso em: 24 maio 2023.

BAELDUNG. **Quicksort vs. Heapsort**. 2023. Disponível em: https://www.baeldung.com/cs/quicksort-vs-heapsort. Acesso em: 24 maio 2023.

PELI, Marcos. **Quicksort vs heapsort**. 2015. Disponível em: https://stackoverflow.com/questions/2467751/quicksort-vs-heapsort. Acesso em: 24 maio 2023.

RAJ, Rashmi. **Analysis of Algorithms**. Disponível em: http://www-cs-students.stanford.edu/ rashmi/projects/Sorting: :text=HeapSort

SIMPLILEARN. **Shell Sort Algorithm: Everything You Need to Know**. 2023. Disponível em: https://www.simplilearn.com/tutorials /data-structure-tutorial/shell-sort: :text=She

WIKIPEDIA, a enciclopédia livre. **Gnome sort**. Disponível em: https://en.wikipedia.org/wiki/Gnome$_s ort. Acessoem$: 01jun.2023.

HAPPYCODERS, BECAME A BETTER PROGRAMMER. **Figuras - (1, 2, 3, 4, 5)**. Disponível em: https://www.happycoders.eu/. Acesso em: 30 mai. 2023.