Trabalho de relatório de algoritmos sort

Cláudio P. T. Araújo* João P. S. Medeiros†

Recebido em 14 de agosto de 2024, aceito em 14 de agosto de 2024.

Resumo

Este relatório apresenta os resultados de experimentos realizados com diferentes algoritmos de ordenação: Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Distribution Sort. O objetivo dos experimentos foi analisar e comparar os tempos de execução desses algoritmos feitos na linguagem de programação C. Os resultados mostram que os algoritmos de ordenação variam significativamente em termos de eficiência dependendo do tipo de entrada, assim mostrando de modo geral como que cada algoritmo se comporta, com amostragem de gráficos e de explicações dos mesmos.

1 Introdução

Este relatório se concentra na análise comparativa de cinco algoritmos de ordenação: Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Distribution Sort. O objetivo principal é investigar como esses algoritmos se comportam em termos de tempo de execução. Para isso, cada algoritmo foi implementado em linguagem C e submetido a testes utilizando diferentes tamanhos de entrada e características diversas de dados.

A comparação dos tempos de execução desses algoritmos permitirá identificar suas vantagens e desvantagens, ajudando na escolha do algoritmo mais apropriado para diferentes necessidades computacionais. Além disso, a análise dos resultados obtidos oferecerá insights sobre a eficiência e escalabilidade dos algoritmos, proporcionando uma base para sistemas reais que busquem usar algum desses algoritmos de ordenação.

Também para a compreensão de algoritmos de ordenação, é de suma importância também ter o conhecimento de diferentes tipos de algoritmos, que são eles os lineares em que apenas um trabalho é feito sobre cada elemento da entrada sempre proporcional ao tempo e tamanho do array $O(n)$, quadráticos que os itens de dados são processados aos pares, muitas vezes com uma repetição dentro da outra $O(n^2)$ e logaritmos que é uma complexidade algorítmica no qual algoritmo resolve um problema com menos tempo de execução, tendo como base os transformar em partes menores $O(n \log n)$.

2 Metodologia

Para a metodologia, foi usado o conhecimento adquirido em sala de aula para a compreensão dos códigos e de como fazer para gerar os devidos gráficos que estão mais a seguir no relatório, usando as ferramentas do GCC (GNU Compiler Collection) para a compilação dos códigos escritos na linguagem de C, e também do GNU plot para poder ter a plotagem dos gráficos que foram analisados.

*Aluno do Bacharelado em Sistemas de Informaçãoda Universidade Federal do Rio Grande do Norte. (e-mail: claudio.pereira.710@ufrn.edu.br)

†Professor do Departamento de Computação e Tecnologiada Universidade Federal do Rio Grande do Norte. (e-mail: jpsm1985@gmail.com)

Para realizar esses testes foi usada um ambiente de desenvolvimento integrado online chamado replit, no qual foram colocados os códigos e rodados para ter uma comparação real de como seria em uma máquina linux, sendo que o selection-sort e o merge-sort foram feitos em máquinas de sistema operacional windows, por causa do ambiente de desenvolvimento não suportar muito estresse de CPU dado por esses algoritmos.

Sendo assim, os resultados podem serem diferentes caso for testar em uma máquina própria linux dependendo da capacidade de processamento da mesma.

3 Experimentos

3.1 Selection Sort

O selection sort é um algoritmo de ordenação que funciona como se fosse uma ordenação de que checa a lista de vetores inteira para assim descobrir o menor e ir pondo o mesmo no começo da lista ou depois do numero menor antes encontrado de forma parecida à como uma pessoa poderia ordenar manualmente uma lista de itens. A ideia central do algoritmo é dividir o array em duas partes: a parte ordenada no início do array e a parte não ordenada no restante. Os passos que esse algoritmo segue basicamente são:

Encontrar o Menor Elemento: Itera pela parte não ordenada do array para encontrar o menor elemento.

Troca de Elemento: Move o menor elemento encontrado para a primeira posição da parte não ordenada, trocando-o com o elemento que está atualmente naquela posição.

Atualizar Partições: Aumenta o tamanho da parte ordenada em uma unidade e diminui a parte não ordenada em uma unidade.

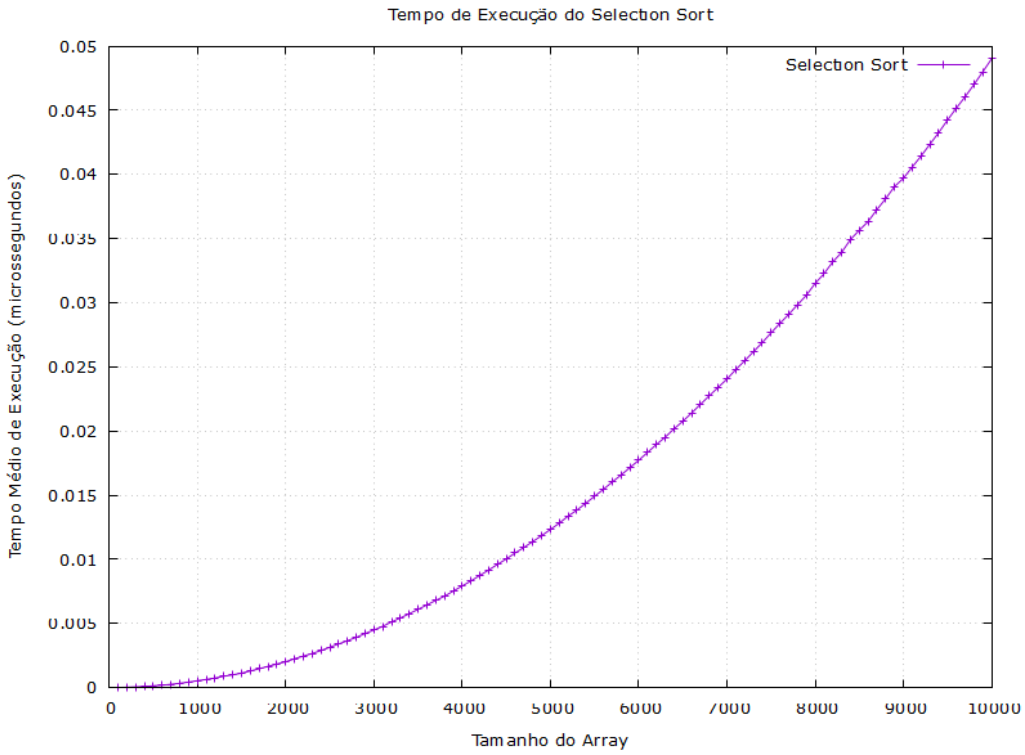

Figura 1: Gráfico do Selection Sort

Como visto no gráfico acima, pode se entender de forma clara que a complexidade do algoritmo de Selection-Sort é quadrática, ou seja, de $O(n^2)$ com isso em mente é facil deduzir que por conta da limitação de 10000 vetores tornou o Selection-Sort bem rápido no tempo de execução, mas isso levando em conta que esse numero de vetores pode não ser considerado muito em algumas situações, por isso mesmo sendo de complexidade quadrática $O(n^2)$ ela se faz muito util para listas pequenas como a demonstrada na Figura 1.

Mas se os testes tivessem sido feitos em um vetor 10 ou 100 vezes maior, o resultado do Selection-Sort seria bem diferente, podendo até ser um dos que mais consomem tempo de execução dentre outros algoritmos sort, por isso vale salientar que o experimento foi feito com um vetor considerado pequeno para programação, com isso ele teve ótimos resultados no teste executado. De modo geral é um bom algoritmo de ordenação se for utilizado com pequenos vetores, mas caso use em algum grande pode demorar muito mais causando assim sua ineficiência.

Para exemplificar algumas vantagens e desvantagens do Selection-Sort são:

Vantagens:

- Simplicidade de implementação.
- Não depende da ordem inicial dos elementos.
- Pouca utilização de memória adicional.

Desvantagens:

- Ineficiente para grandes conjuntos de dados devido à sua complexidade temporal quadrática.
- Lento em comparação com outros algoritmos de ordenação, como quicksort e mergesort, para arrays grandes.

3.2 Insertion Sort

O insertion sort é um algoritmo de ordenação que funciona muito bem para pequenas listas ou listas que já estão parcialmente ordenadas. Ele constrói a lista ordenada elemento por elemento, inserindo cada novo item na posição correta dentro da lista já ordenada. Basicamente processo é repetido até que todos os elementos estejam na posição correta, escolhendo um vetor e o fazer percorrer a lista inteira até achar sua posição. O algoritmo é descrito em 3 passos:

Iteração pelos Elementos: O algoritmo percorre o array da esquerda para a direita.

Inserção na Posição Correta: Para cada elemento, ele encontra a posição correta na parte ordenada do array e o insere nessa posição.

Deslocamento de Elementos: Os elementos na parte ordenada que são maiores que o elemento a ser inserido são deslocados para a direita para abrir espaço para o novo elemento.

Agora para se falar sobre a complexidade do Insertion-Sort é preciso saber que ele não possui apenas um tipo de complexidade, e sim 2 o de complexidade linear $O(n)$ e a quadrática $O(n^2)$ sendo o de complexidade linear o melhor caso desse algoritmo e a quadrática o médio e pior caso do mesmo que será exemplificado com os graficos a seguir.

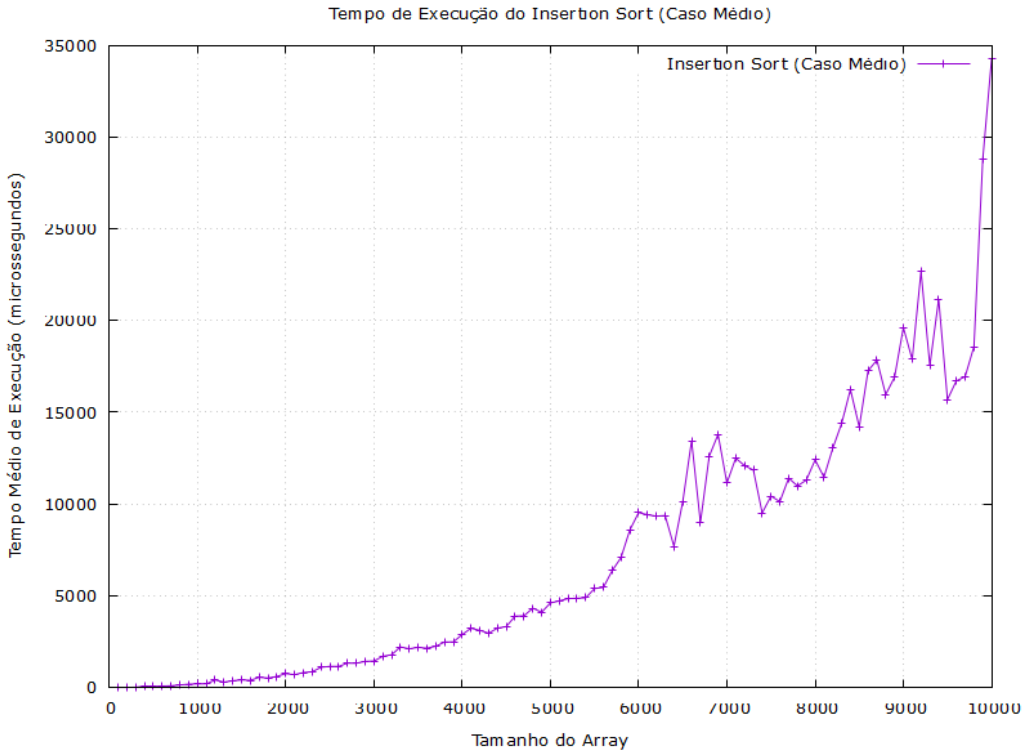

Figura 2: Gráfico do caso médio Insertion Sort

Caso Médio $O(n^2)$: Ocorre quando o array contém elementos em ordem aleatória. Cada novo elemento a ser inserido pode precisar ser deslocado através de vários elementos da parte ordenada, em média, o algoritmo precisa deslocar metade dos elementos da parte ordenada para cada inserção. O tempo de execução cresce de forma quadrática, refletindo a inserção de elementos em posições diversas dentro da parte ordenada.

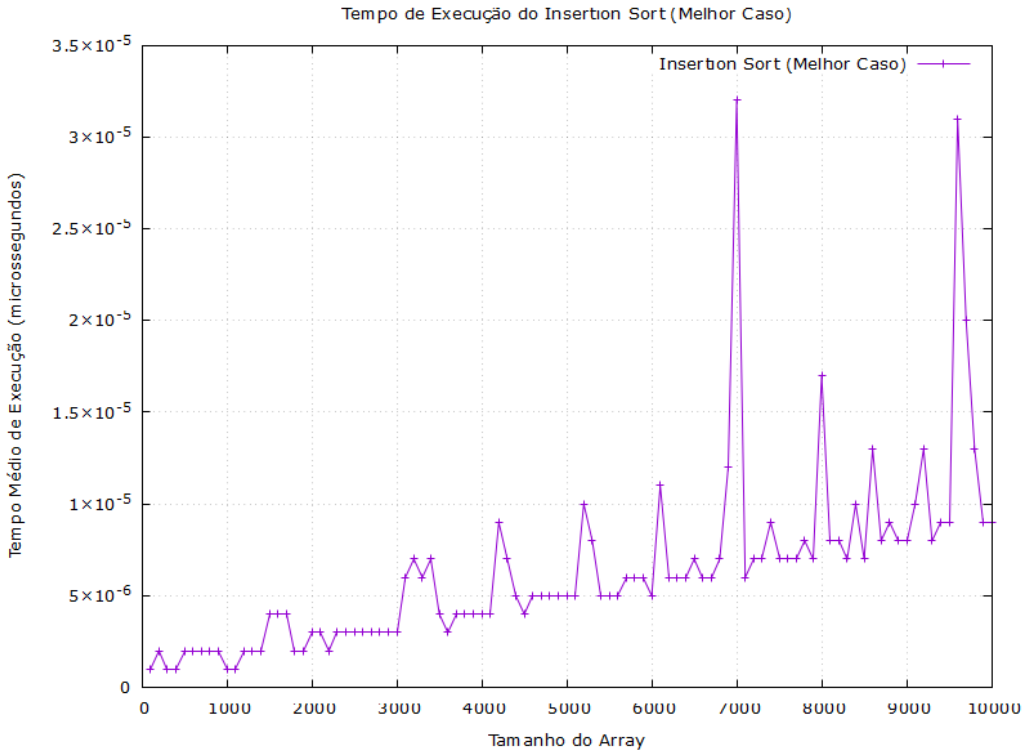

Figura 3: Gráfico do melhor caso Insertion Sort

Melhor Caso $O(n)$**:** Ocorre quando o array já está ordenado. O algoritmo apenas percorre o array sem necessidade de deslocar elementos, em cada iteração, o algoritmo apenas compara o elemento atual com o último elemento da parte ordenada, sem necessidade de deslocar elementos, com isso esse cenário indica o melhor desempenho do Insertion-Sort para o qual o numero de operações é reduzido, com isso, o tempo de execução cresce linearmente, demonstrando a eficiência do algoritmo em listas já ordenadas.

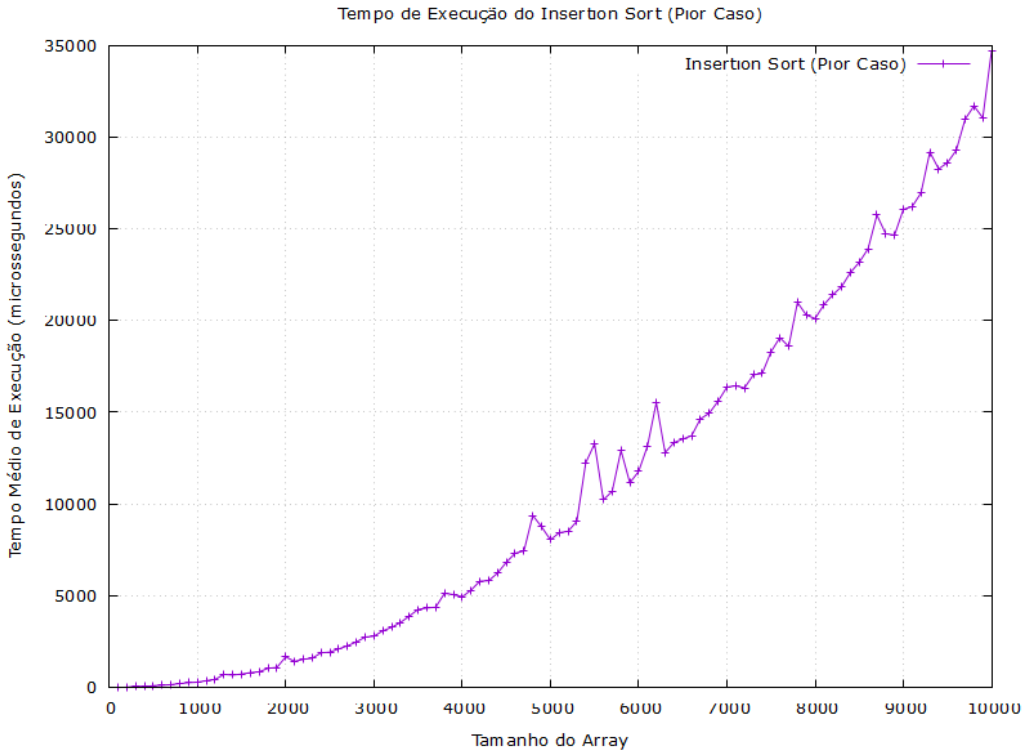

Figura 4: Gráfico do pior caso Insertion Sort

Pior Caso $O(n^2)$: Ocorre quando o array está em ordem inversa. Cada novo elemento a ser inserido é menor que todos os elementos na parte ordenada, exigindo deslocamento máximo, com isso, esse cenário resulta no maior número possível de operações, demonstrando a ineficiência do algoritmo para listas grandes e completamente desordenadas, por isso quando vemos a Figura 4 o tempo de execução também cresce quadraticamente, mas de forma mais pronunciada devido ao deslocamento de todos os elementos da parte ordenada para cada inserção.

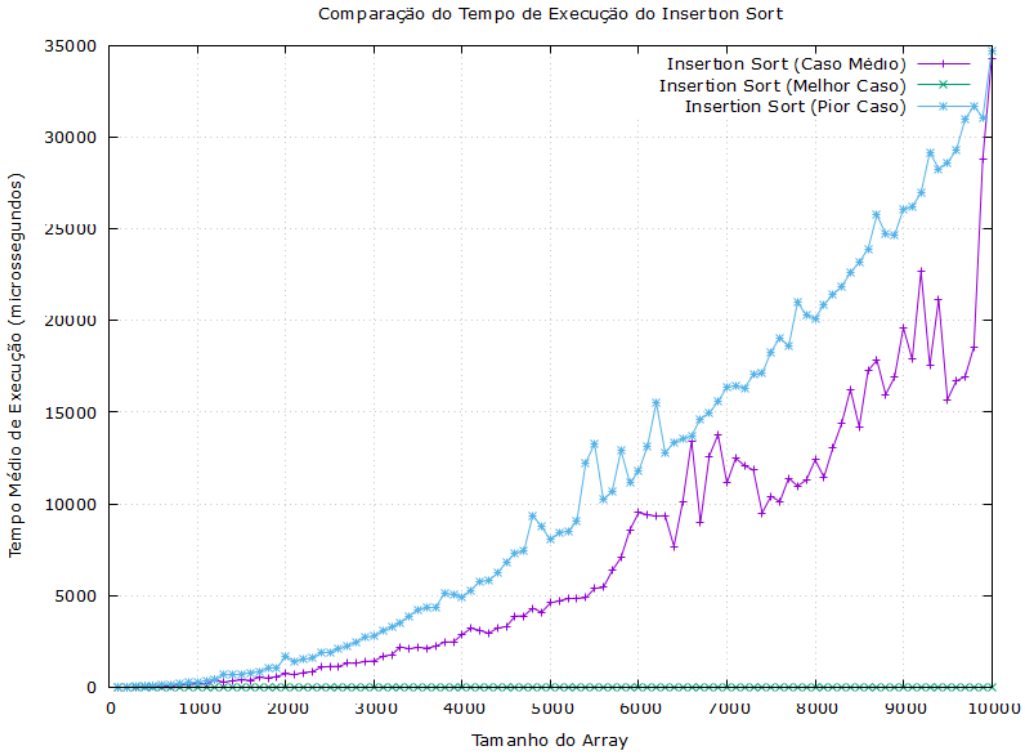

Figura 5: Gráfico de comparação dos Insertion Sort

Com isso vemos que ao comparar todos os gráficos o Insertion-Sort pode ser bem custoso no seu tempo de execução dependendo da ordenação do array em expecífico, o pior e médio caso não se diferem tanto dos seus resultados por serem em suma de mesma complexidade $O(n^2)$, diferente do seu melhor caso que é linear $O(n)$ que não custou quase nada de tempo de execução e vale salientar que esse algoritmo não é eficiente para ordenar grandes quantidades de dados devido ao seu crescimento quadrático em tempo de execução, sendo mais amplamente utilizado apenas para checagens caso necessário já que possui uma varredura rápida e caso precise possa-se organizar poucos dados que estejam desordenados, mas para arrays sem ordenação e que são grandes não se faz muito util a utilização desse algoritmo, dado isso esse algoritmo possui suas peculiaridades em ter suas vantagens e desvantagens que são:

Vantagens:

- Simplicidade de implementação.
- Eficiência em listas pequenas ou quase ordenadas.
- Algoritmo estável (não altera a ordem relativa de elementos iguais).
- Pouca utilização de memória adicional.

Desvantagens:

- Muito demorado para grandes conjuntos de dados que estejam aleatórios ou ordenados inversamente devido à sua complexidade temporal quadrática.
- Performance significativamente pior em listas grandes comparada a algoritmos como quicksort e mergesort.

3.3 Merge Sort

O Merge Sort é um algoritmo de ordenação que utiliza a técnica de "dividir para conquistar"lembrando um pouco o Quick-Sort por conta da recursividade presente também no codigo. Ele é útil para ordenar grandes conjuntos de dados, sendo feito com que o algoritmo divida repetidamente o array em subarrays menores até que cada subarray contenha um único elemento e, em seguida, combina esses subarrays em ordem crescente para formar o array ordenado final. O funcionamento do algoritmo pode ser descrito em três passos principais:

Divisão do Array: O array é recursivamente dividido em duas metades até que cada subarray tenha um único elemento.

Conquista: Os subarrays são ordenados recursivamente.

Combinação: Os subarrays ordenados são combinados para formar um array ordenado maior.

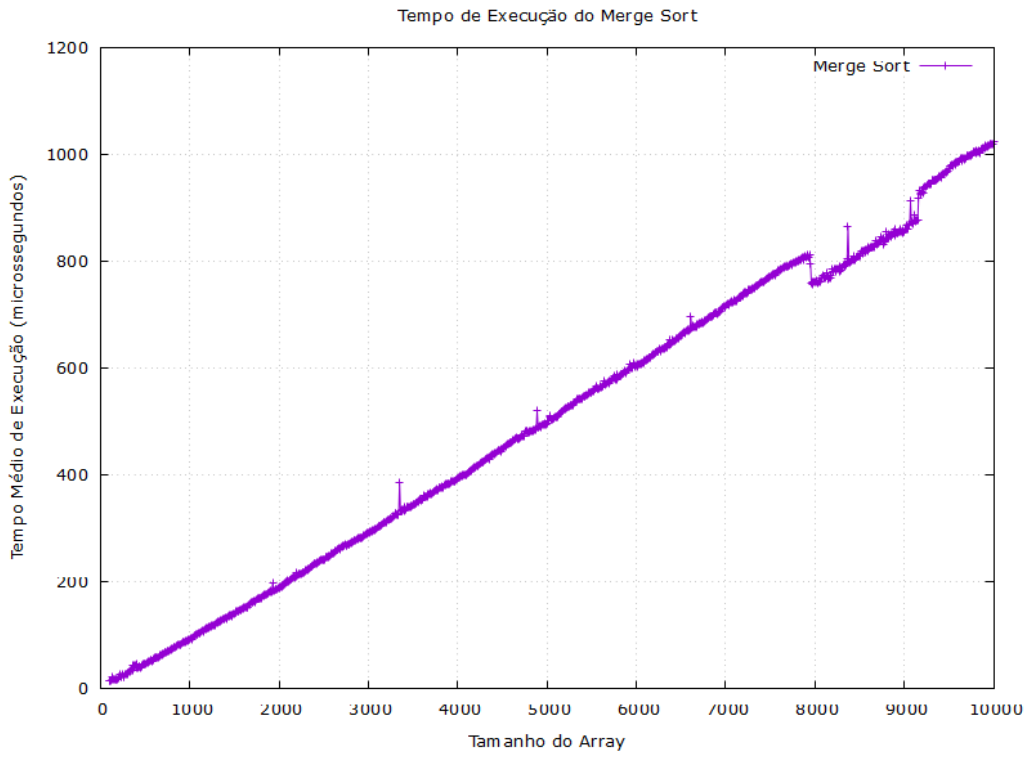

Figura 6: Gráfico do Merge Sort

Como o Merge Sort tem a mesma complexidade para todos os casos, não há distinção entre melhor, médio e pior caso em termos de complexidade temporal sendo todos os casos pertencentes a complexidade logaritma $O(n \log n)$. Por isso não se teve necessidade de implementar médio, melhor e pior caso distintos, e devido a limitação do array possuir apenas 10000 vetores não foi possivel visualizar de forma clara a questão logaritma do algoritmo em si, pois como dito anteriormente ele é para ser um tipo de algoritmo de ordenação mais estável e que tem um uso melhor em arrays grandes por causa da sua complexidade de execução, sendo assim para arrays menores não muito útil por consumir a maior parte do seu tempo de execução mais no começo doque pro final que ele vai está mais estável devido a complexidade $O(n \log n)$ e que para sistemas com memória restrita, outras opções podem ser melhor consideradas. Dado isso as vantagens e desvantagens de se usar o Merge-Sort são:

Vantagens:

- Complexidade de tempo garantida de $O(n \log n)$ para todos os casos.
- Estabilidade, preservando a ordem relativa de elementos iguais.
- Bom desempenho em conjuntos de dados grandes e em dados que não cabem na memória principal (útil em algoritmos de ordenação externa).

Desvantagens:

- Utiliza memória adicional proporcional ao tamanho do array a ser ordenado.
- Pode ser menos eficiente que o Quicksort para arrays pequenos devido à sobrecarga de combinar subarrays.

3.4 Quick Sort

O Quicksort é um dos algoritmos de ordenação mais eficientes e mais comum de verem sendo utilizados. Ele utiliza a estratégia de "dividir para conquistar"para ordenar elementos no qual faz uma recurssão no código chamando ele mesmo duas vezes para poder ordenar um mesmo vetor o dividindo em partes para que no final fique totalmente ordenado. O algoritmo pode ser descrito em três passos principais:

Escolha do Pivô: Seleciona um elemento do array como pivô. Existem várias estratégias para a escolha do pivô, como o primeiro elemento, o último, o meio, ou até mesmo um valor aleatório.

Particionamento: Reorganiza o array de forma que todos os elementos menores que o pivô fiquem à esquerda do pivô e todos os elementos maiores fiquem à direita.

Recursão: Aplica o mesmo processo recursivamente nas sublistas à esquerda e à direita do pivô até que todas as sublistas estejam ordenadas.

E parecido com o Insertion-Sort, o Quick-Sort possui diferenças no seus casos de tempo de execução sendo o médio e o melhor caso possuindo uma complexidade de execução de $O(n \log n)$ que não cresce muito a medida que o vetor também cresce, e para o pior caso possui complexidade quadrática $O(n^2)$ que dependendo do tamanho do vetor pode ser prejudicial para o tempo de execução. Com isso destacado, ademais segue os gráficos de seus respectivos casos e a explicação dos mesmos:

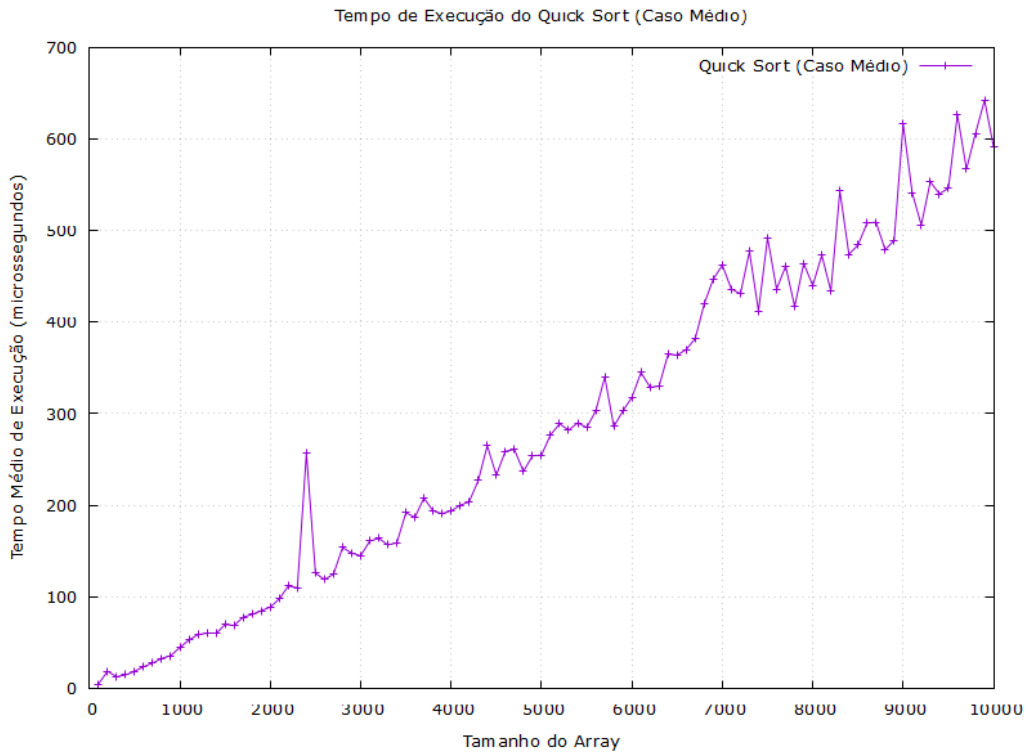

Figura 7: Gráfico do caso médio Quick Sort

Caso Médio $O(n \log n)$ **:** Ocorre quando em média, a escolha do pivô é tal que as partições são relativamente balanceadas. Este é o cenário mais comum na prática, onde a escolha do pivô é relativamente boa e o algoritmo realiza um número de operações proporcional a $n \log n$ refletindo a performance típica do Quicksort em situações práticas.

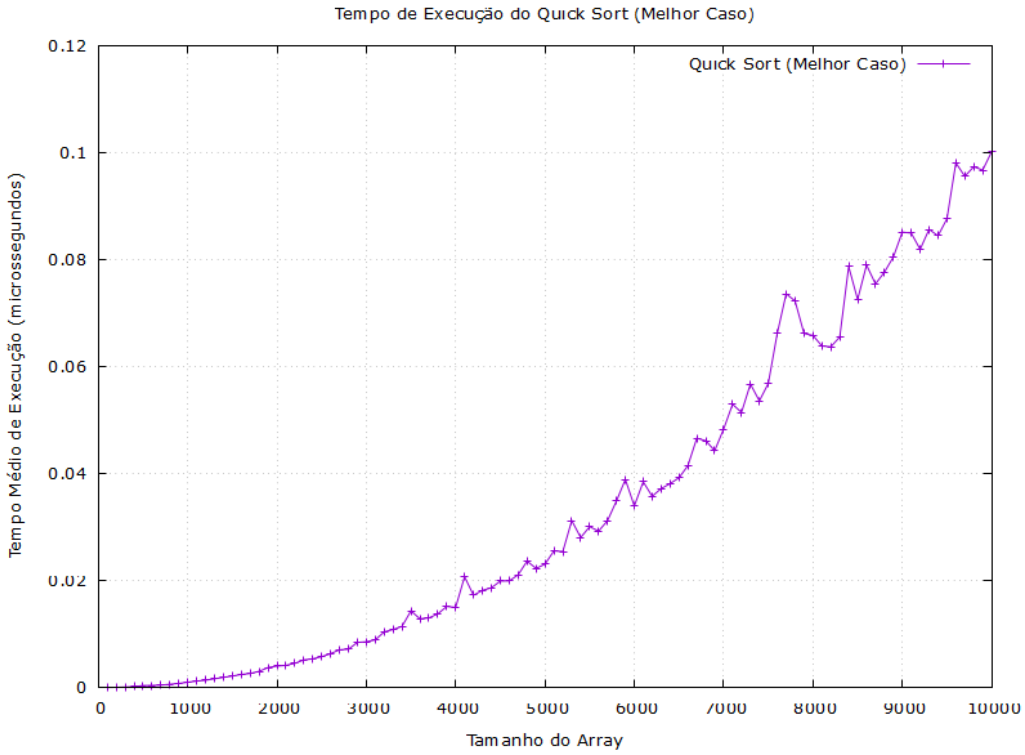

Figura 8: Gráfico do melhor caso Quick Sort

Melhor Caso $O(n \log n)$ **:** Ocorre quando o pivô divide o array em duas partes aproximadamente iguais a cada iteração. Cada divisão reduz o problema pela metade, resultando em um tempo de execução proporcional a $n \log n$ refletindo a eficiência do algoritmo quando o pivô divide o array de forma equilibrada.

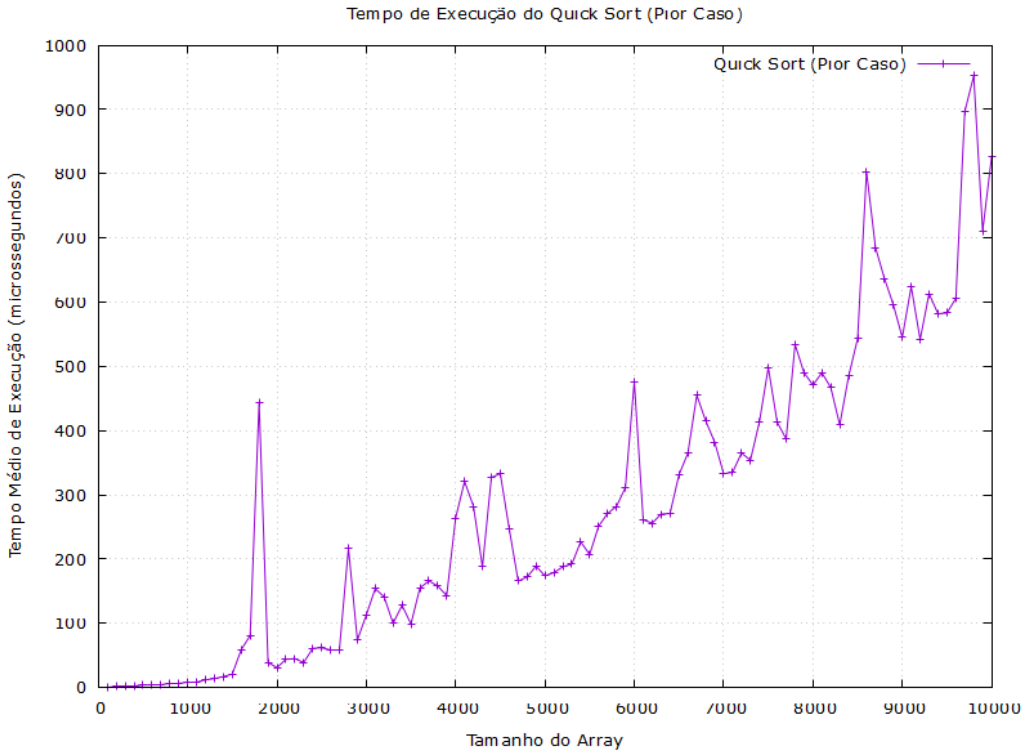

Figura 9: Gráfico do pior caso Quick Sort

Pior Caso $O(n^2)$: Ocorre quando o pivô é o menor ou o maior elemento a cada iteração, resultando em partições muito desiguais (por exemplo, se o array estiver já ordenado ou ordenado em ordem inversa). Cada divisão reduz o array apenas em um elemento, resultando em uma série de divisões que é linear em número, com isso o tempo de execução cresce de forma quadrática n^2, demonstrando o impacto negativo de uma escolha ruim de pivô em grandes conjuntos de dados.

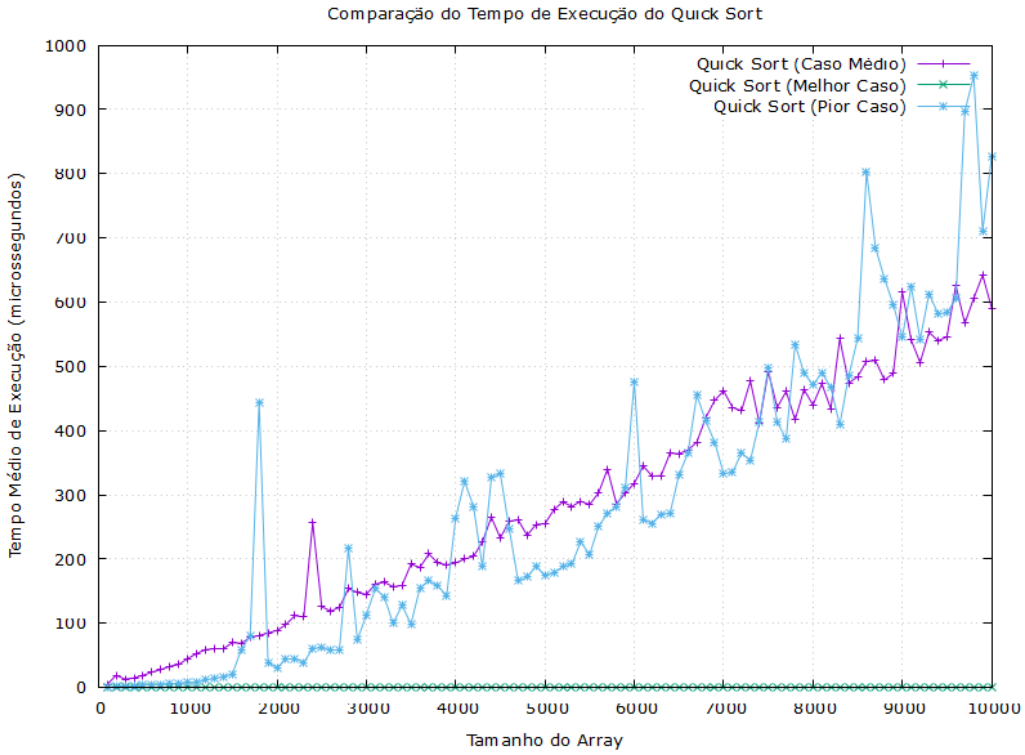

Figura 10: Gráfico de comparação do Quick Sort

Com isso, ao comparar os resultados dos gráficos podemos ver que devido a limitação de 10000 vetores no array o caso médio e pior caso ficaram quase que empatados mas isso se deve exclusivamente pelo tamanho do array que foi escolhido para fazer esses experimentos e análises, pois se levar em conta a complexidade de tempo de execução de cada caso pode-se ter uma noção que ao aumentar o vetor 10 ou 100 vezes pode ter resultados completamente diferentes, com base que o pior caso do Quick-Sort que é quadrático $O(n^2)$ iria crescer muito mais enquanto o caso médio que é logaritmo $O(n \log n)$ iria continuar crescendo muito menos doque o do pior caso, com isso em conta, da para imaginar o por que esse é o algoritmo mais amplamente utilizado para ordenação, já que o pior caso é bem dificil de se acontecer e por ser logaritmo, mesmo com um número de array maior, ele ainda assim iria conseguir entregar uma perfomace decente, mas que geralmente é mais utilizado em arrays de menor tamanho por sua praticidade de implementação já que arrays maiores são mais recomendadas outros tipos de algoritmos de ordenação, dado isso, algumas vantagens e desvantagens de se usar o Quick-Sort são:

Vantagens:

- Alta eficiência para grandes conjuntos de dados devido à sua complexidade média de $O(n \log n)$.
- Algoritmo in-place, requerendo pouca memória adicional.
- Flexível, podendo ser adaptado com diferentes estratégias de escolha de pivô.

Desvantagens:

- Complexidade no pior caso pode chegar a $O(n^2)$ se o pivô não for escolhido adequadamente.
- Não é um algoritmo estável (não preserva a ordem relativa de elementos iguais).

3.5 Distribution Sort

O Distribution Sort, que utilizou da estrategia do Counting Sort(que conta quantas vezes cada valor ocorre no array original), é um algoritmo de ordenação que se baseia na distribuição dos elementos em um array auxiliar de contagem, pois o Distribution-Sort tem que derivar de um algoritmo auxiliar para ajudar na ordenação, e o mesmo possui de modo geral 3 métodos conhecidos que são o Counting Sort, o Radix Sort e o Bucket Sort, mas que para essa análise e experimento foi utilizado o Counting Sort. Esse método pode ser bem mais eficiente quando o intervalo de valores dos elementos é relativamente pequeno, já que Se o intervalo de valores for muito grande, o tamanho do array auxiliar de contagem também será grande. Isso não apenas aumenta o uso de memória, mas também pode tornar a fase de inicialização do array auxiliar e a fase de reconstrução do array ordenado mais custosas, com isso o funcionamento do algoritmo pode ser descrito em três passos principais:

Contagem dos Elementos: Os elementos do array são distribuídos em um array de contagem onde cada índice representa um valor possível no array de entrada, e o valor em cada índice representa a contagem de ocorrências daquele valor.

Construção do Array de Contagem: Para cada elemento do array original, incrementa-se a posição correspondente no array de contagem.

Reconstrução do Array Ordenado: O array original é reconstruído utilizando as contagens armazenadas, colocando os elementos na ordem correta.

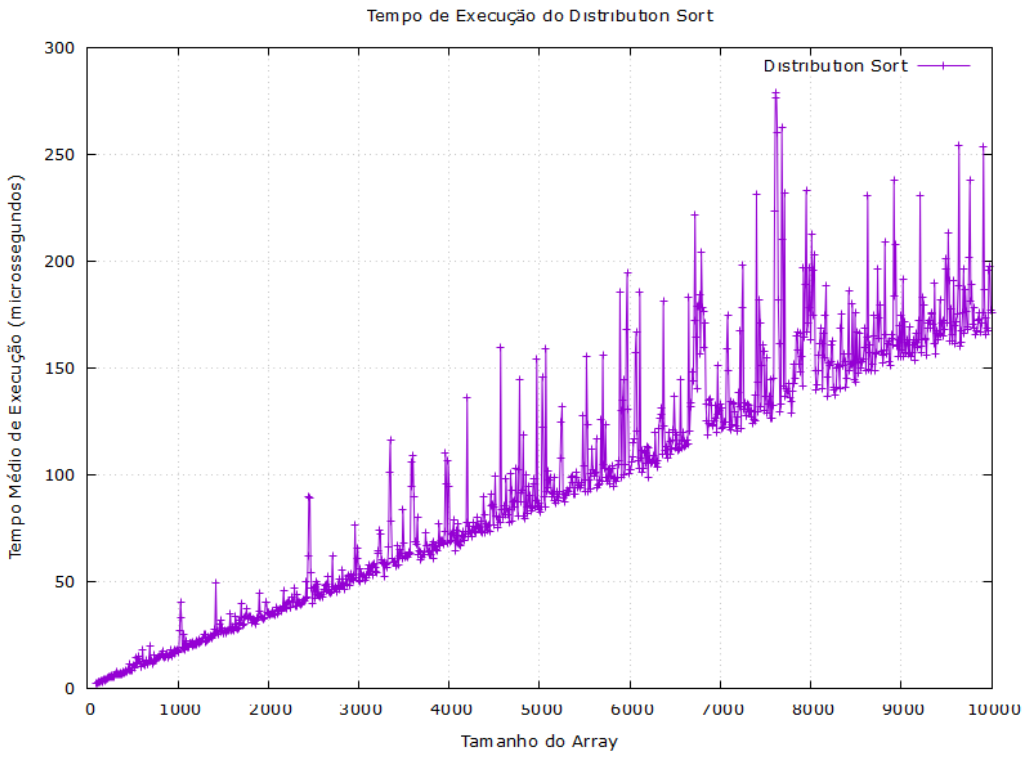

Figura 11: Gráfico do Distribution Sort

Para a complexidade do Distribution-Sort é considerada a complexidade linear em relação ao número de elementos n e ao intervalo de valores k sendo assim de complexidade $O(n + k)$ no qual o k seria o valor máximo do array, e mesmo com a limitação de 10000 vetores da para perceber se

desconsiderar as falhas por conta do processamento do sistema que de fato segue uma complexidade de certa forma linear, sendo altamente eficiente para intervalos de valores restritos e bem definidos, com isso podendo ser usado tanto para arrays grandes ou pequenas já que o seu tempo de execução não oferece muito custo, apesar de que o uso da memória possa ser mais prejudicial se for com grandes arrays, mas se isso não for uma limitação é um ótimo uso por ser linear e ter como prever um pouco o tempo de execução dependendo do tamanho. Com isso algumas vantagens e desvantagens desse algoritmo são:

Vantagens:

- Com o uso do Counting Sort ele se torna muito eficiente para arrays onde o intervalo de valores k é pequeno em comparação ao número de elementos n.
- Alta eficiência em casos específicos onde os valores estão em um intervalo restrito ou possuem características que facilitam a distribuição.
- Pode ser mais rápido que algoritmos de ordenação por comparação (como Quicksort ou Merge Sort) em certos cenários.

Desvantagens:

- Requer conhecimento prévio das propriedades dos dados (como o intervalo de valores).
- Menos flexível para conjuntos de dados grandes ou não uniformes.
- Requer memória adicional proporcional ao intervalo de valores, o que pode ser um fator limitante para grandes intervalos.

3.6 Tempo de Execução Geral

Ao analisar todos os gráficos e dados gerados do tempo de execução de cada algoritmo de ordenação foi possivel juntar todos eles em um só gráfico para poder se ter uma métrica de comparação entre os diferentes tipos de algoritmos sort que de modo geral todos ficaram como está a figura adiante:

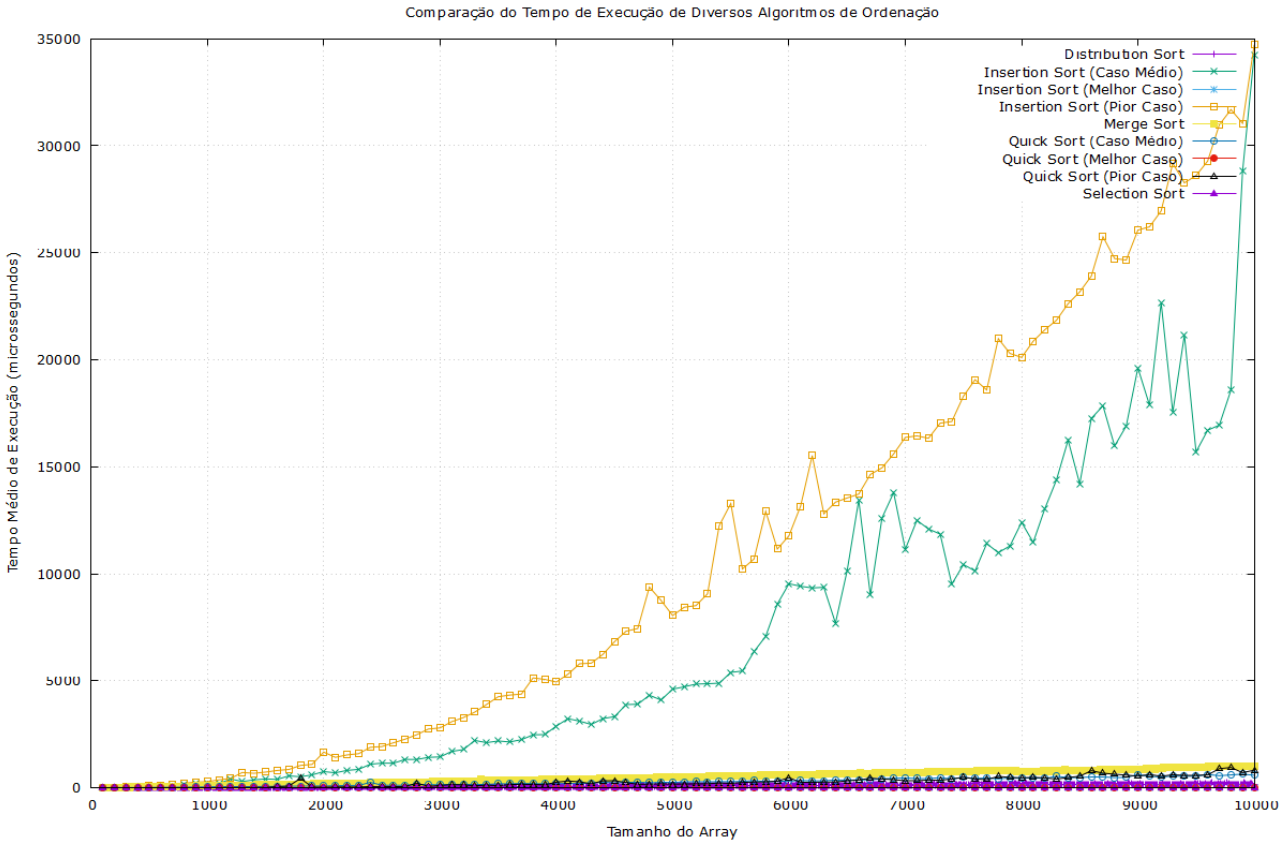

Figura 12: Gráfico de comparação entre todos os algoritmos

Como previsto, o gráfico ainda não consegue mostrar com exatidão todos ao mesmo tempo por existir o pior e médio caso do Insertion-Sort que foram muito custosos no tempo de execução já que são de complexidade quadrática $O(n^2)$ e que mesmo o array sendo considerado pequeno ainda tiveram muita dificuldade para ordenar o mesmo, sendo assim, ele se torna um algoritmo muito custoso com o tempo de execução, não sendo recomendado ao menos com base nesses testes para nenhum tipo de ordenação muito complexa ou que tenha que ser executada com pouco tempo para manter estável, e para conseguir visualizar melhor os resultados foi feito outro gráfico sem a presença do Insertion-Sort no médio e pior caso logo adiante.

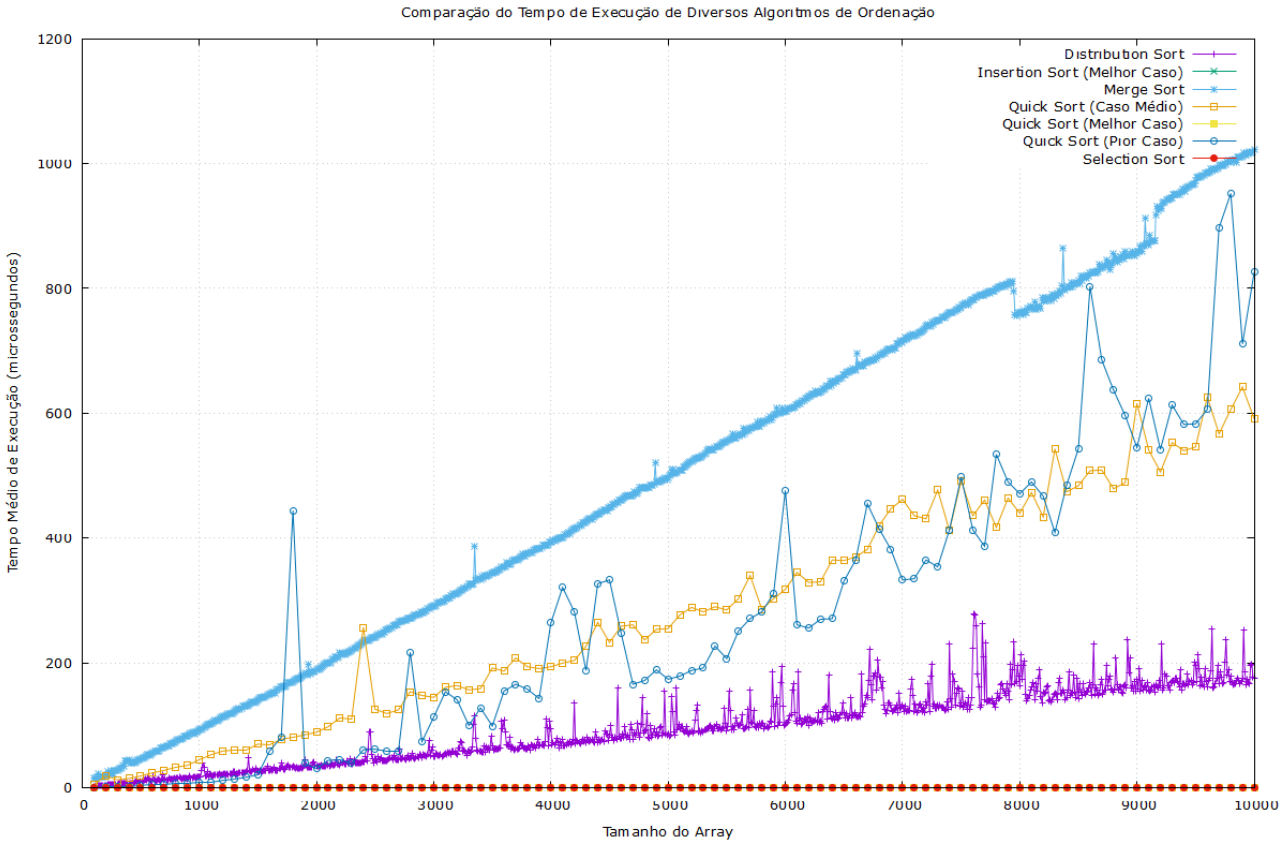

Figura 13: Gráfico de comparação entre todos os algoritmos sem o pior e médio caso do Insertion Sort

Com esse gráfico fica mais visível como cada algoritmo se comportou e mais facil ainda de poder comparálos com os demais, que oque pode-se notar é que dentre todos (tirando o Insertion-Sort no caso médio e pior caso) é que o merge foi oque mais custou tempo de execução mesmo sendo $O(n \log n)$ mas isso se deve ao fator que sempre foi deixado bem claro durante todo o relatório, que foi a limitação de ser um array de 10000 vetores, que mesmo que não seja um numero grande para se testar ainda assim da para tirar uma noção de como todos se comportam, e o merge não é diferente, pois com essa limitação da para ter uma noção de que o merge não é o ideal para arrays pequenas, já que sua função tende a ser mais estável com grandes vetores e não custar muito tempo de execução, sendo assim o mais util dentre todos para banco de dados extensos ou sistemas do tipo.

Logo abaixo tem a presença do pior e médio caso do quick que mesmo com a complexidades diferentes ainda se mantem semelhantes em um array pequeno, coisa que mudaria drasticamente caso fosse testado com um array 10 ou 100 vezes maior, pois com isso, o médio caso do quick iria continuar custando relativamente o mesmo custo de tempo de execução enquanto o pior caso teria já aumentado drasticamente, causando assim uma descrepância muito grande de uma da outra.

E por ultimo desse gráfico que ainda é possivel analisar se tem o Distribution-Sort, que como sua complexidade linear em relação ao número de elementos $O(n + k)$ já demonstra é que como já dito anteriormente o array possuir essa limitação, o Distribution-Sort teve sua execução bastante rápida devido ao maior numero do vetor não ter um intervalo tão diferente dos outros valores, dado a esse quesito foi possivel ter um bom resultado com o tempo de execução, mesmo sendo de complexidade linear que dependendo do maior valor poderia facilmente ultrapassar o caso médio do Quick-Sort ou até do Merge-Sort, possuindo essa limitação de que tem que sempre declarar a constante do maior valor do vetor para ter uma boa funcionalidade, caso contrário pode ser um pouco difícil a implementação do mesmo, além de que ele é o que menos exige da memória para rodar no sistema, chegando a ser um

limitador em alguns casos por essa causa, sendo assim não muito indicado para arrays no qual não se conhece o vetor máximo e que também o sistema tem um empecilho em gastar muita memória com esse algoritmo.

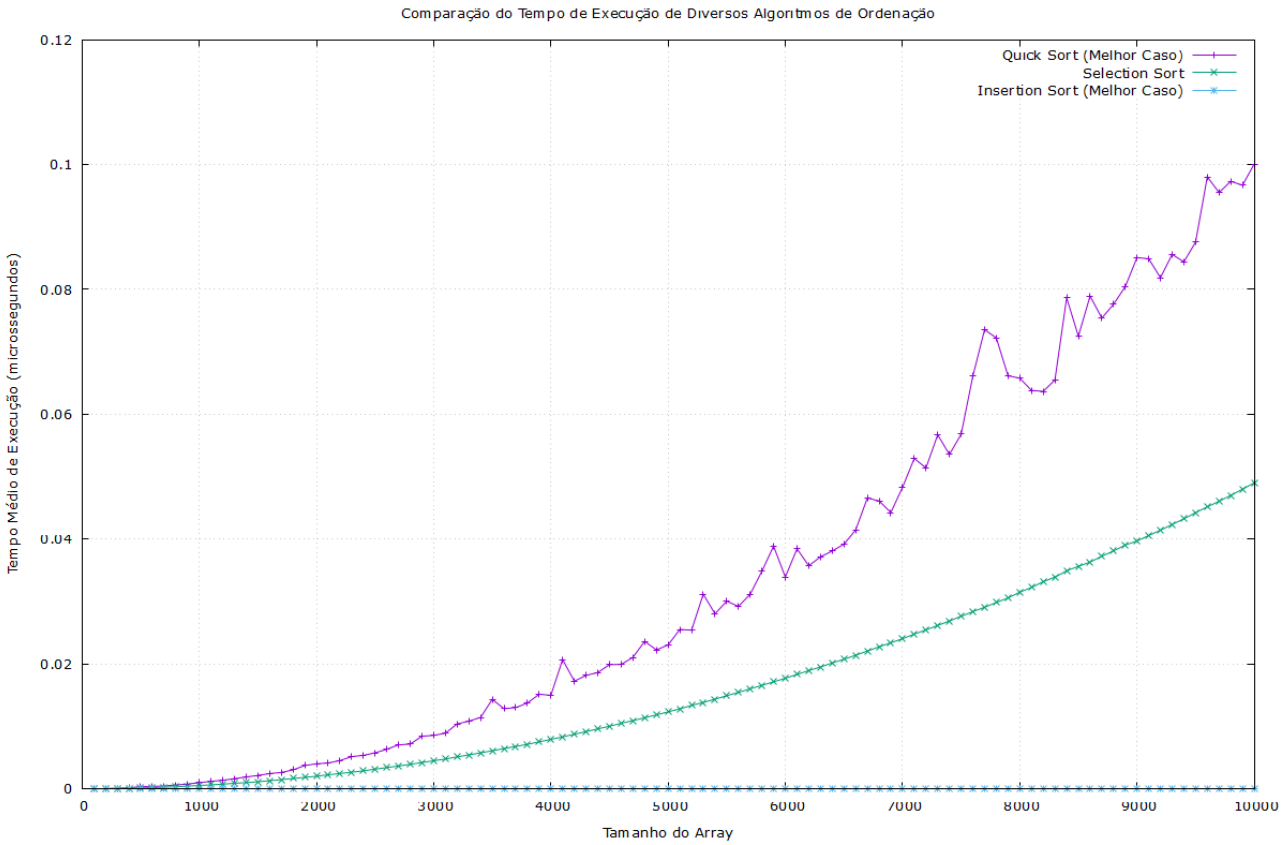

Figura 14: Gráfico de comparação entre os menores tempos de execução

E ao Fim das comparações em gráfico como não se pôde visualizar o melhor caso do Quick-Sort nem do Insertion-Sort e também não dava para visualizar o Selection-Sort, foi feito mais esse gráfico para poder ter uma análise completa de todos eles e poder ver a olho nu como que seria o tempo de cada em array de 10000 vetores, que como pode se observar na Figura 14 o Quick-Sort no melhor caso está como sendo o que mais demora dentre os outros dois, mas isso se tem que levar em consideração que por ser de complexidade logaritma $O(n \log n)$ ele vai ser possível se manter nesse tempo de execução ou não aumentar muito mais por isso.

Selection-Sort, que por causa do tamanho do array ser pequeno ele teve um ótimo desempenho mesmo sendo de complexidade quadrática $O(n^2)$, que da para tirar de conclusão que esse algoritmo é uma ótima adição para arrays do mesmo tipo, diferente se fosse maior pois ele poderia ser um dos que mais consomem tempo de execução junto do pior caso do Quick-Sort por eles serem de mesma complexidade, com isso é mais util usar-lo em pequenos arrays que vai ter uma boa eficiência como mostra o experimento.

Já o melhor caso do Insertion-Sort, quase não pode se chamar de algoritmo de ordenação já que oque foi o motivo dele ficar com um tempo de execução tão baixo foi devido que para se ter o melhor caso do Insertion-Sort é necessário o vetor já estar ordenado, pois ele vai apenas fazer uma varredura checando se todos estão nos lugares corretos, assim quase nem existindo um tempo de execução para o mesmo, mas com o teste é possivel se notar que um ótimo uso do mesmo é para checar arrays que já estão ordenadas ou possuem poucos elementos desordenados pois vai ter uma rápida varredura e

mal vai custar tempo de execução, pois, mesmo o melhor caso sendo ele já estar ordenado ainda assim tem um uso muito util para sistemas que so precisam fazer verificações e pequenos ajustes rápidos.

Colocação	Algoritmos de Ordenação	Maior Tempo de Execução
1°	Insertion-Sort(Melhor Caso)	0.000009
2°	Selection-Sort	0.049010
3°	Quick-Sort (Melhor Caso)	0.100087
4°	Distribution-Sort	175.800000
5°	Quick-Sort(Caso Médio)	591.000000
6°	Quick-Sort(Pior Caso)	826.082609
7°	Merge-Sort	1023.000000
8°	Insertion-Sort(Caso Médio)	34253.000000
9°	Insertion-Sort(Pior Caso)	34714.000000

Tabela 1: Resultados do tempo médio de algoritmos de ordenação do menor para o maior

Por fim, uma tabela mostrando o resultado final de cada algoritmo de ordenação presente neste relatório, vale ressaltar novamente que os experimentos foram feitos apenas com arrays de tamanho 10000, por isso mesmo o Selection-Sort estar com um tempo tão baixo, não é recomendável usar-lo em arrays grandes ou gigantes pois pode-se ter um atraso muito grande para a resposta da ordenação, e também para o Merge-Sort por mesmo tendo ficado em uma classificação baixa, vale salientar que ele pode ser um dos que menos consome tempo de execução em arrays maiores devido sua complexidade logaritma $O(n \log n)$ por isso deve-se levar em consideração todos esses fatores doque levar os resultados desse relatório como verdades absolutas.

3.7 Memória Usada

Para a parte de memória usada, foi usado vários testes mas nenhum em suma saiu como esperado, provavelmente pela falta de conhecimento da linguagem não foi possível fazer um algoritmo no qual fosse possível ver com precisão a memória usada do sistema para os algoritmos de ordenação, pois todos estavam acusando ser 40000 bytes de memória usados no array em todos eles, com isso não houve necessidade de por esses resultados aqui neste relatório já que todas acusavam o mesmo tamanho de memória usado, podendo ser verdade ou não.

Mas de modo geral, com base nos graficos, mesmo sendo apenas com o tempo de execução de cada um, tem como ter uma certa noção de qual algoritmo consome mais ou menos memória do sistema, por exemplo o menor tempo de execução nos testes foi o selection-sort, mesmo possuindo uma complexidade quadrática $O(n^2)$ com um numero pequeno de vetores que foi o caso desse relatório o uso de memória pode ser considerado pequeno ou indiferente, igualmente ao melhor caso do insertion sort que tem complexidade linear $O(n)$ já que já está ordenado e apenas vai fazer uma verificação, esses exemplos dão a entender que o uso de memória não vai ser nenhum problema nesses casos.

Ademais, para conjunto de dados grandes o selection-sort pode ser um problema, por ser de complexidade $O(n^2)$ ele pode chegar a ser o algoritmo que mais vai ter uso da memória do sistema, então para vetores grandes o que pode-se tirar de conclusão seria o uso do Merge-Sort já que é de complexidade linear $O(n)$ então teria menos uso de memória do sistema além de também ser mais viável para essa situação de grandes conjuntos de dados.

4 Conclusão

Para concluir o relatório cabe um veredito de qual realmente é o melhor algoritmo de ordenação dentre todos os apresentados, levando em conta claro o tamanho do array, pois, mesmo constando que o melhor caso do Insertion-Sort e o Selection-Sort sejam os classificados "melhores"no tempo de execução mesmo assim ainda não pode se considerar isso devido a limitação do array em si, pois de modo geral para usos mais constantes e sistemas que queiram se manterem mais estáveis com algoritmos de ordenação, o melhores mesmo são os que possuem complexidade de execução logaritma $O(n \log n)$, pois, além de que com o maior numero de vetores ele não vai aumentar drasticamente o tempo de execução, eles também não são instaveis e também não possuem um grande uso de memória, sendo possivel se adequar em muitos sistemas, e até mesmo com o array de 10000 que foi o caso dos testes já apresentados, eles ainda assim não gastam muito tempo de execução se for comparar com o pior e médio caso do Insertion-Sort já que todos foram medidos em microsegundos, que para padrões humanos não duram quase nada, dado isso, o veredito é que o Quick-Sort melhor/médio caso e o Merge-Sort são os melhores algoritmos de ordenação e também os mais sustentáveis com o tempo e tamanho de array, mas caso não for possível manter um melhor ou médio caso do Quick-Sort, então de modo geral o Merge-Sort consegue ser o algoritmo mais recomendável e melhor de se implementar em sistemas pela sua complexidade logaritma $O(n \log n)$ e sustentabilidade.

A Informações Complementares

```c
void selectionSort(int arr[], int n) {
    int i, j, minIndex, temp;
    for (i = 0; i < n - 1; i++) {
        minIndex = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex])
                minIndex = j;
        }
        if (minIndex != i) {
            temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}
```

Figura 15: Código do Selection Sort

```c
void insertionSort(int arr[], int size) {
    int i, key, j;
    for (i = 1; i < size; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Figura 16: Código do Insertion Sort

```c
void merge(int vetor[], int comeco, int meio, int fim) {
    int com1 = comeco, com2 = meio + 1, comAux = 0, tam = fim - comeco + 1;
    int *vetAux;
    vetAux = (int*)malloc(tam * sizeof(int));

    while(com1 <= meio && com2 <= fim) {
        if(vetor[com1] < vetor[com2]) {
            vetAux[comAux] = vetor[com1];
            com1++;
        } else {
            vetAux[comAux] = vetor[com2];
            com2++;
        }
        comAux++;
    }

    while(com1 <= meio) {
        vetAux[comAux] = vetor[com1];
        comAux++;
        com1++;
    }

    while(com2 <= fim) {
        vetAux[comAux] = vetor[com2];
        comAux++;
        com2++;
    }

    for(comAux = comeco; comAux <= fim; comAux++) {
        vetor[comAux] = vetAux[comAux - comeco];
    }

    free(vetAux);
}

void mergeSort(int vetor[], int comeco, int fim) {
    if (comeco < fim) {
        int meio = (fim + comeco) / 2;

        mergeSort(vetor, comeco, meio);
        mergeSort(vetor, meio + 1, fim);
        merge(vetor, comeco, meio, fim);
    }
}
```

Figura 17: Código do Merge Sort

```c
// Função para trocar dois elementos
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

// Função para particionar o vetor para o QuickSort
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Função para realizar o QuickSort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Figura 18: Código do Quick Sort

```c
// Função para realizar o Distribution Sort
void distributionSort(int array[], int tamanho) {
    // Encontrar o valor máximo no array
    int max = array[0];
    for (int i = 1; i < tamanho; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }

    // Criar um array auxiliar para contar as ocorrências de cada valor
    int *count = (int *)malloc((max + 1) * sizeof(int));
    if (count == NULL) {
        printf("Erro na alocação de memória.\n");
        return;
    }

    // Inicializar o array count com zeros
    for (int i = 0; i <= max; i++) {
        count[i] = 0;
    }

    // Contar as ocorrências de cada valor
    for (int i = 0; i < tamanho; i++) {
        count[array[i]]++;
    }

    // Atualizar o array original com os valores ordenados
    int j = 0;
    for (int i = 0; i <= max; i++) {
        while (count[i] > 0) {
            array[j] = i;
            j++;
            count[i]--;
        }
    }

    free(count);
}
```

Figura 19: Código do Distribution Sort