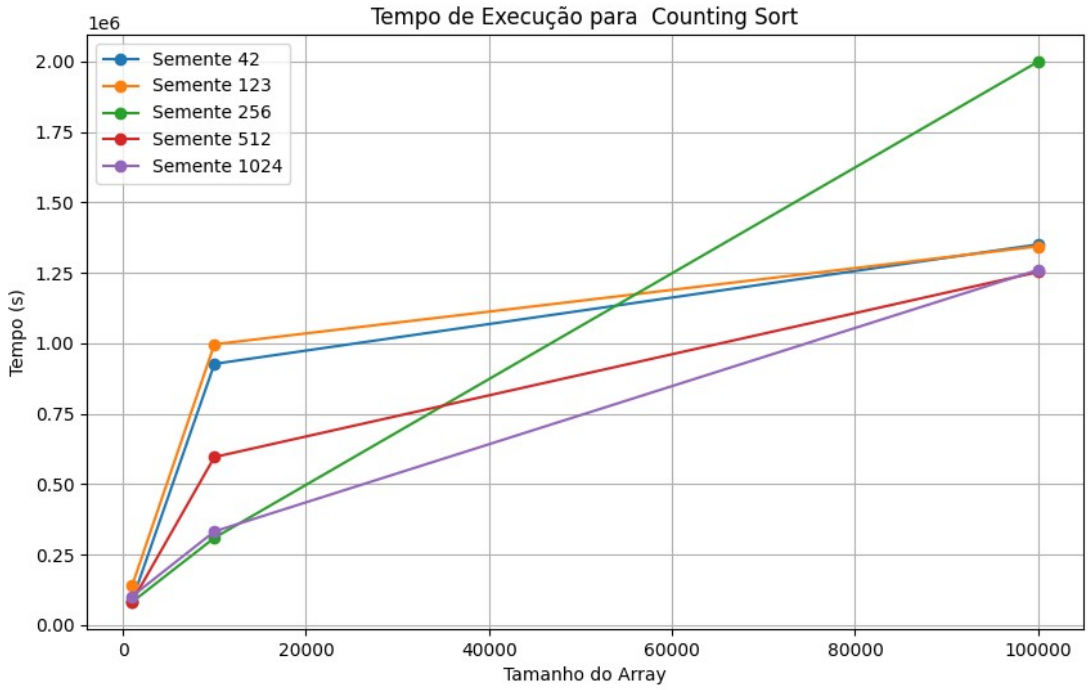
Relatório

Link do repositório no Github: https://github.com/ricaprof/Java-TDE
Analisando o caso de Counting Sort

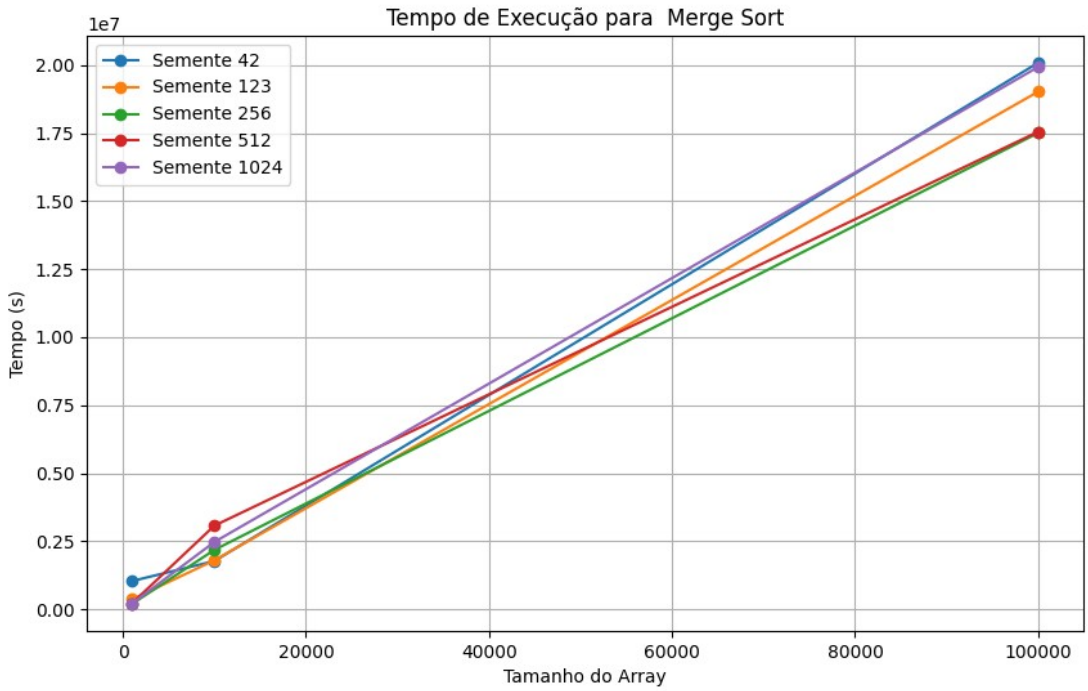


Inicialmente, para um número de elementos até cerca de 10.000, o algoritmo mostra um tempo de execução mais elevado. Esse aumento pode estar relacionado à forma como o Count Sort lida com a alocação do array auxiliar, que, em pequenas quantidades de dados, pode ser mais custoso proporcionalmente. Além disso, a contagem de valores dispersos exige mais operações de acesso e modificação do array auxiliar, o que pode resultar em uma execução mais lenta nos primeiros pontos do gráfico.

Após atingir a marca de 10.000 elementos, o algoritmo se torna mais eficiente. Isso ocorre porque, com um maior número de elementos, o custo da inicialização do array auxiliar se dilui em relação ao volume total de dados. Nesse ponto, o Count Sort consegue processar os elementos mais rapidamente, possivelmente porque os valores estão mais concentrados ou o intervalo é mais bem aproveitado, otimizando o uso do array de contagem.

Esse comportamento pode sugerir que o Count Sort possui uma espécie de "ponto de equilíbrio" em que o aumento do número de elementos compensa o custo da inicialização e manipulação do array auxiliar. Em outras palavras, o crescimento de elementos começa a favorecer o algoritmo em termos de desempenho, tornando-o mais rápido depois de ultrapassar a fase inicial de ajuste do array.

Analisando o caso de Merge sort:

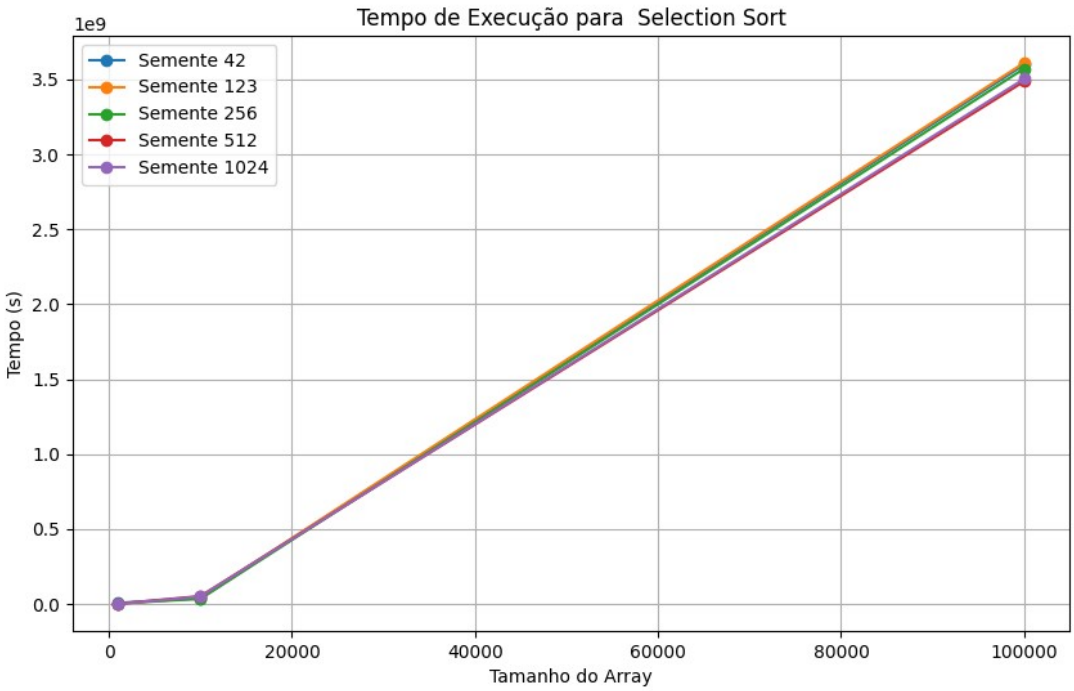


O Merge Sort é um algoritmo altamente eficiente que segue o princípio de divisão e conquista, dividindo repetidamente os dados em sublistas menores até restarem pares ou elementos individuais. Uma vez divididos, esses elementos são ordenados e combinados recursivamente, até que a lista original esteja completamente ordenada. Com isso, o Merge Sort é teórico e experimentalmente muito estável e possui uma complexidade de tempo O(nlogn), tornando-o ideal para grandes conjuntos de dados, independente de sua organização inicial.

No gráfico mostra um comportamento de tempo de execução quase linear, o que pode parecer inesperado, já que o Merge Sort possui uma complexidade teórica que depende do termo nlogn. Esse comportamento linear aparente pode ser explicado pelo crescimento lento da função logn em relação ao número de elementos n. Conforme o número de dados aumenta, o termo logn cresce de forma tão lenta que se torna praticamente imperceptível em comparação ao crescimento linear de n, especialmente quando os dados são representados graficamente.

Em resumo, embora a complexidade teórica do Merge Sort seja O(nlogn), o comportamento observado no gráfico sugere que, na prática, ele lida de forma excepcional com grandes volumes de dados, de modo que a curva de crescimento do tempo de execução parece praticamente linear. Isso reflete a eficiência do algoritmo em particionar e combinar dados, fazendo com que ele mantenha uma performance robusta e ágil mesmo em cenários de alta demanda.

Analisando o caso de Selection Sort:



,

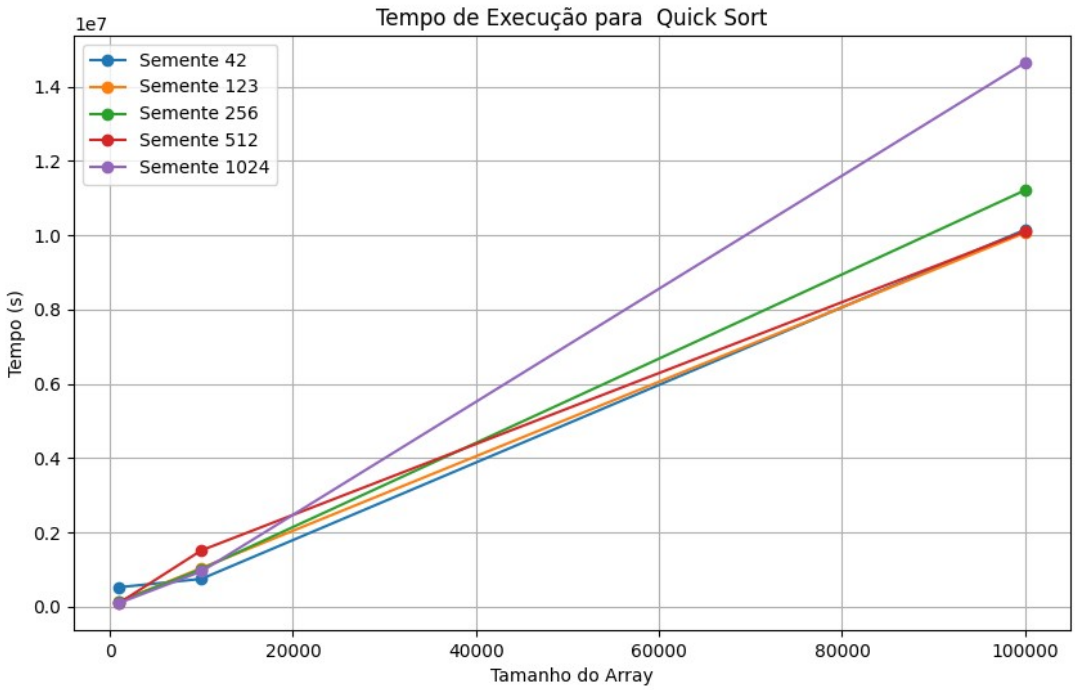
O Selection Sort é um algoritmo de ordenação simples que, como o próprio nome sugere, funciona selecionando repetidamente o menor (ou maior) elemento de uma lista e movendo-o para a posição correta. A cada iteração, o algoritmo percorre a lista, localiza o elemento que deve ser colocado na próxima posição ordenada e, em seguida, realiza a troca. Sua complexidade de tempo é O(n2), o que significa que o tempo de execução aumenta quadraticamente em relação ao número de elementos na lista.

Ao analisar o desempenho do Selection Sort, especialmente em um gráfico que mostra seu tempo de execução, é possível notar um comportamento interessante. Inicialmente, o algoritmo tende a executar muito rapidamente, especialmente em listas pequenas ou em listas que já estão quase ordenadas. Isso ocorre porque o número de comparações e trocas necessárias para organizar os primeiros elementos é baixo, o que resulta em um tempo de execução eficiente no início.

No entanto, à medida que o número de elementos aumenta ou a desordem na lista se torna mais significativa, o desempenho do Selection Sort começa a deteriorar. A razão para isso está na sua abordagem ingênua: para cada elemento na lista, o algoritmo precisa percorrer a lista restante para encontrar o próximo menor elemento. Esse processo se torna cada vez mais custoso em termos de tempo à medida que a lista cresce, pois o número de comparações se eleva rapidamente.

Portanto, o gráfico do Selection Sort reflete um desempenho rápido no início, onde o número reduzido de elementos e a quase ordenação favorecem o algoritmo. No entanto, à medida que o número de elementos aumenta, o custo de execução aumenta rapidamente, resultando em um tempo de ordenação que se torna significativamente mais longo, tornando-o inadequado para conjuntos de dados maiores ou desordenados(tanto é que ele leva mais tempo que os outros). Isso destaca as limitações do Selection Sort, que, embora seja fácil de implementar e entender, não é uma escolha prática para situações que exigem alta eficiência.

Tempo de execução Quick Sort:



O Quick Sort é um dos algoritmos de ordenação mais populares e eficientes, utilizando o método de divisão e conquista para organizar os elementos. Ele funciona selecionando um elemento chamado "pivô" e particionando os dados em torno desse pivô, de modo que todos os elementos menores que o pivô fiquem à esquerda e todos os elementos maiores fiquem à direita. Em seguida, o algoritmo aplica recursivamente o mesmo processo nas sublistas resultantes até que todas as partes estejam ordenadas.

O Quick Sort possui uma complexidade de tempo média de O(nlogn), o que o torna bastante eficiente para grandes conjuntos de dados. No entanto, seu desempenho pode variar bastante dependendo da escolha do pivô e da organização inicial dos dados. Em um cenário ideal, onde o pivô é escolhido de forma a dividir a lista em partes aproximadamente iguais, o Quick Sort se comporta muito bem, mantendo a eficiência esperada.

Ao analisar o gráfico do Quick Sort, é comum observar um desempenho muito rápido em comparação com outros algoritmos de ordenação, especialmente em listas de tamanho moderado a grande. No início, para um número relativamente pequeno de elementos, o algoritmo se destaca pela rapidez nas operações de particionamento e pela capacidade de lidar eficientemente com dados já quase ordenados. Essa característica resulta em um tempo de execução que pode parecer linear em alguns casos, especialmente quando as listas são bem distribuídas.

No entanto, à medida que o número de elementos aumenta ou quando a escolha do pivô não é ideal (por exemplo, quando o pivô é o menor ou maior elemento da lista), o desempenho do Quick Sort pode piorar significativamente. Em situações desfavoráveis, como listas já ordenadas ou com muitos elementos repetidos, o algoritmo pode degenera para uma complexidade de O(n2)O(n2), já que as divisões se tornam muito desequilibradas, exigindo mais comparações e trocas.

Assim, o gráfico do Quick Sort reflete essa dualidade de comportamento: um desempenho rápido e eficiente na maioria das situações, mas com a possibilidade de queda abrupta de desempenho em casos específicos como na semente 1024. Isso torna o Quick Sort uma escolha poderosa e popular em aplicações práticas, mas também exige atenção na seleção do pivô e na estrutura inicial dos dados para garantir que ele funcione em sua eficiência ideal.