Laboratório de Estrutura de Dados

Primeira versão do projeto da disciplina

Comparação entre os algoritmos de ordenação elementar

IAN DINIZ DE OLIVEIRA.
LUKAS CHRISTOPHER DE SOUZA SANTANA.
NICOLAS MARTINS DE MOURA.

1. Introdução

Este projeto da disciplina de laboratório de estruturas de dados foi realizado com o objetivo de colocar em prática os conhecimentos abordados em sala de aula. Para isso, foi escolhido o projeto “Los Angeles Metro Bike Share”, um sistema de compartilhamento de bicicletas na cidade de Los Angeles, Califórnia e área metropolitana. O sistema usa uma frota de cerca de 1400 bicicletas e inclui mais de 93 estações na região denotada de trabalho do sistema. Os dados referentes ao registro de locações das bicicletas controladas pelo sistema entre os anos de 2016 e 2021, são disponibilizados em um arquivo no formato CSV.

A primeira etapa do projeto consiste em realizar algumas transformações nos arquivos *stations.csv* e *LA_Metro_BikeSharing_CLEANED_2016quater3-2021q3.csv.* Essas transformações são:

1. Gerar um arquivo único chamado LAMetroTrips.csv substituindo o id das estações pelo nome (campo station_name) que está contido no arquivo (stations.csv).

2. Considerando o arquivo gerado no Item 1 (LAMetroTrips.csv), filtrar apenas as viagens que estão nas estações de Pasadena. Gerar um arquivo chamado LAMetroTrips_F1.csv

3. Considerando, o arquivo gerado no Item 1 (LAMetroTrips.csv), filtra apenas as viagens que possuem duração maior que a média geral.

Para as transformações, foram utilizadas algumas importações da biblioteca padrão do java. Para a leitura e escrita dos arquivos utilizamos as classes BufferedReader, FileReader. FileWriter e File. Todas necessitam de apenas alguns segundos para carregar o arquivo no *array* principal de forma que possamos realizar as transformações e criar os novos arquivos. As classes responsáveis pela modificação do arquivo base podem ser encontradas no arquivo do projeto. São elas as classes: P0ConsertarDatas, P1SubstituirId, P2FiltrarStationName e P3FiltrarMedia.

Seguindo com a segunda parte do projeto, foi requisitada a ordenação dos dados do arquivo gerado no item 1 - LAMetroTrips.csv. Os algoritmos usados foram: Selection Sort, Insertion Sort, Merge Sort, Quick Sort, QuickSort com Mediana de 3, Counting Sort e HeapSort. Por fim, deve-se gerar um arquivo para cada algoritmo de ordenação e o tipo de caso(melhor, pior e médio).

As ordenações exigidas são as seguintes:

1. Ordenar o arquivo completo pelo nomes das estações (campo station_name/ station_id) em ordem alfabética.

2. Ordenar o arquivo LAMetroTrips.csv pelo campo de duração da viagem (campo duration) do menor para o maior

3. Ordenar o arquivo LAMetroTrips.csv pela data de início da viagem (campo Start_time) mais recente para o mais antigo.

Os resultados sobre tempo de execução e uso de memória dos algoritmos podem ser visualizados nos seguintes gráficos:

2. Descrição geral sobre o método utilizado

Para o projeto foi necessário utilizarmos três computadores pessoais referentes a cada integrante do grupo. Cada integrante do grupo ficou responsável por realizar uma das tarefas de manipulação do arquivo principal bem como cada um ficou com a tarefa de implementar sua própria versão dos algoritmos de ordenação já citados. Como tivemos que ordenar o arquivo várias vezes em relação diferentes colunas do arquivo principal *duration, starta time e start station* tivemos alguns problemas referentes aos tipos de dados que cada coluna possui e tomamos a decisão de criar algoritmos diferentes para cada tipo de dado a ser ordenado, no caso Int (duração), LocalDateTime (início da viagem) e String. (nome da estação)

O programa desenvolvido para o projeto abre, altera e cria novos arquivos. Foi construída a classe “AbrirArquivoCsv.java”, responsável pela função de abrir e ler todos os arquivos e colocar seus dados em um *array*. Primeiro lemos a quantidade de linhas do arquivo desejado para que possamos alocar a memória necessária para ler o arquivo no *array*. Em seguida abrimos o arquivo através das classes da biblioteca java e salvamos linha a linha no *array* principal.

Com o arquivo carregado no *array* podemos utilizar a classe “CsvMetod.java” para obter uma coluna específica do arquivo principal, também podemos utilizar essa classe para podermos alterar toda uma coluna do arquivo principal.

Agora que temos como abrir e selecionar uma coluna específica do arquivo podemos focar no processo de ordenação. Cada um dos três tipos de dados a serem ordenados tem seu respectivo tipo de algoritmo para ordenação, uma vez que estamos comparando diferentes tipos de dados, inteiros, Strings e datas. Para ordenar selecionamos o algoritmo desejado e passamos o *array* com os dados do arquivo junto com outro *array* contendo uma coluna do *array* principal, o *array* coluna é ordenada e à medida que seus dados trocam de posição alteramos também o array do arquivo principal. Dessa forma, à medida em que ordenamos uma coluna, também estamos ordenando o *array* principal que contém os dados do arquivo csv. Após a ordenação criamos um novo arquivo com dados do csv ordenados.

Descrição geral do ambiente de testes

- Especificações do dispositivo utilizado na ordenação por Data:
  CPU: Intel(R) Core(TM) i3-2310M CPU @ 2.10GHz
  Ram: 6GB DDR3
  Sistema: Ubuntu 23.04

- Especificações do dispositivo utilizado na ordenação por Nome da estação:
  CPU: Athlon 3000g 3.50 GHz
  Ram: 8GB DDR4
  Sistema: Windows 10

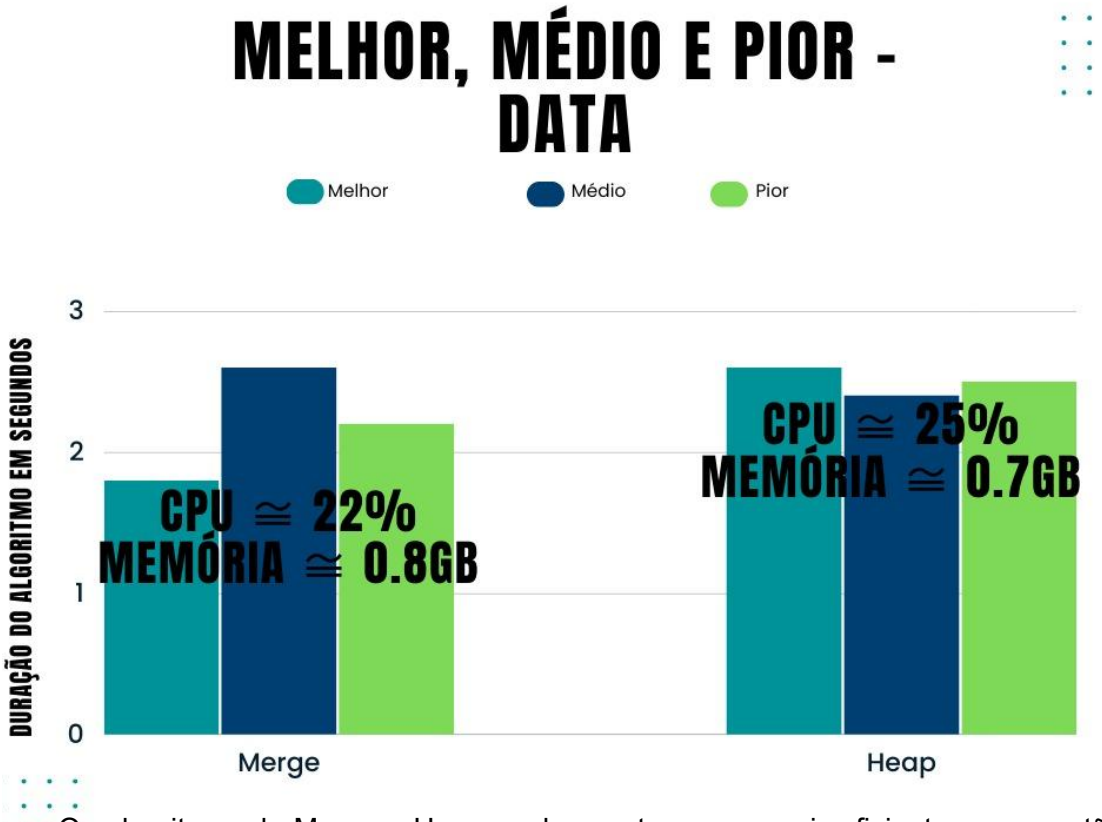
- Especificações do dispositivo utilizado na ordenação por Nome da estação:
  CPU: Xeon E5-2650 V2
  Ram: 16GB DDR3
  Sistema: Windows 10

- Especificações do dispositivo utilizado na ordenação por Duração da viagem:

3. Resultados e Análise

Data - Tempo de execução

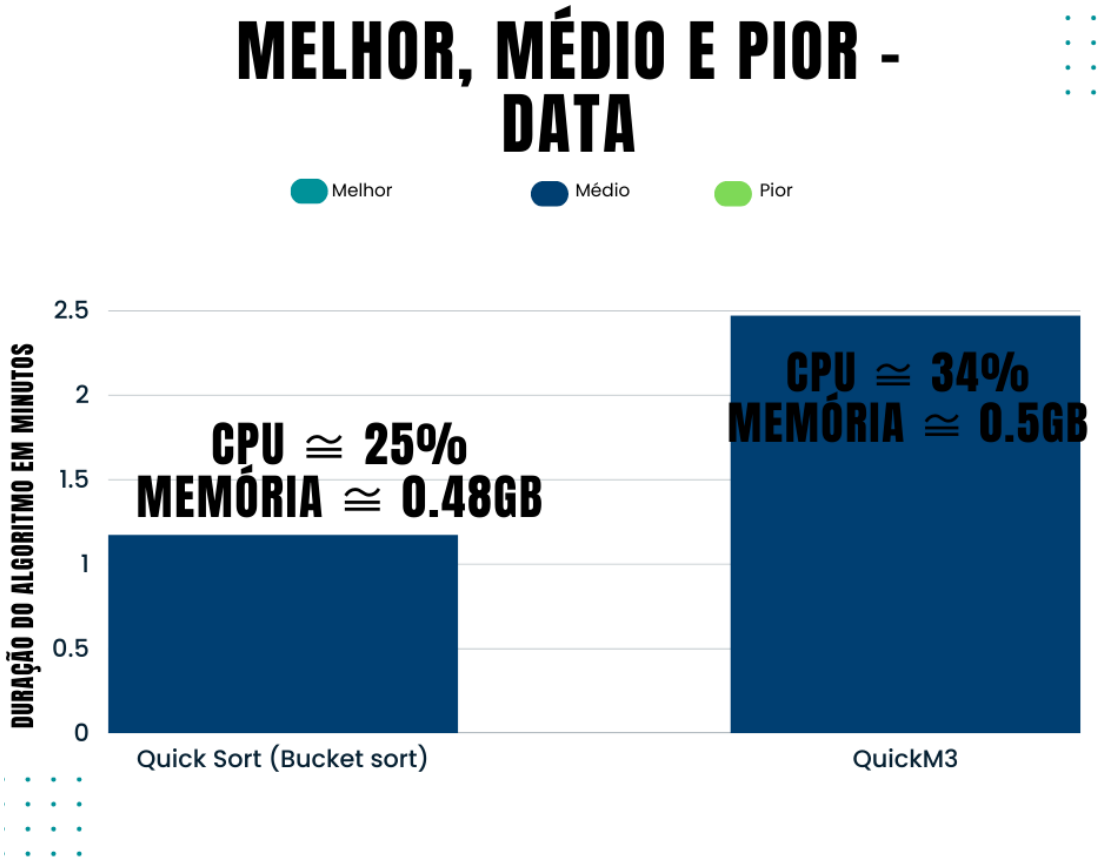
Para as ordenações com os algoritmos Merge Sort e Heapsort obtemos os seguintes resultados



Os algoritmos de Merge e Heap se demonstraram os mais eficientes em questão de tempo e utilização de recurçêos computacionais, ambos são O(log n) e ordenam o arquivo numa faixa de tempo de dois a três segundos. Embora que ambos gastam mais memória do que outros algoritmos sua utilização extra de memória vale a pena, uma vez que algoritmos como Selection Sort e Insertion Sort por mais que não gastem tanta memória extra são extremamente lentos levando em média 3 horas para ordenarem o arquivo no médio caso. Claro, em situações em que o tempo é relevante para a aplicação principal e o número de dados é baixo, algoritmos como o inserto e o selection podem ser utilizados.

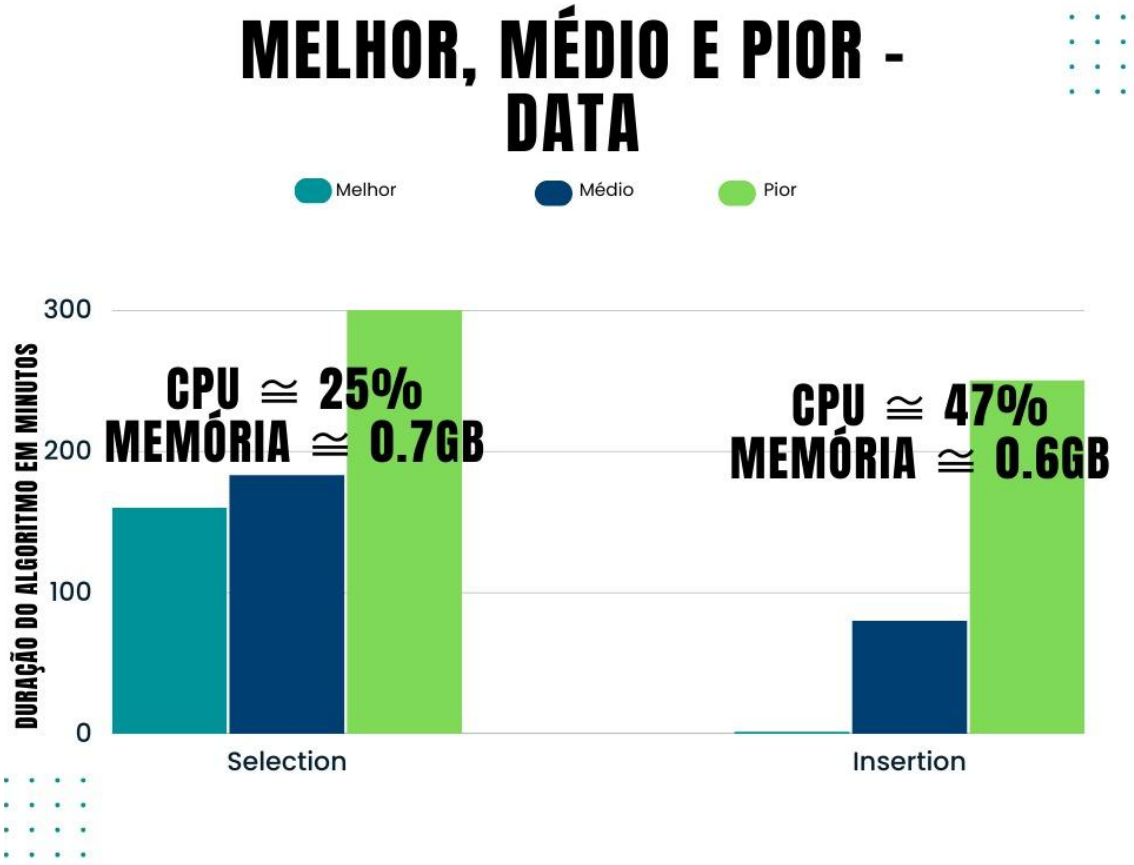
Durante o desenvolvimento dos nossos algoritmos tivemos um grande problema com os algoritmos de Quicksort. Graças a natureza recursiva do Quicksort tivemos estouros na memória da Stack do programa, isso ocorre principalmente quando o algoritmo escolhe um

pivô ruim, oque ocorre no melhor e pior caso. No caso médio não tivemos grandes problemas com o Quicksort com mediana de 3, contudo se tentarmos rodar o Quicksort sem a mediana de 3 iremos nos deparar com o problema de Stack overflow. Para resolver isso implementei um algoritmo arcaico de Bucket Sort para as datas e divide as datas em “baldes” referentes ao anos de 2016 a 2021, com isso o algoritmo de Quicksort passou a funcionar no médio caos, tendo inclusive um resultado melhor que o algoritmo com mediana de 3.



O Quicksort no geral é um bom algoritmo quando estamos trabalhando com dados que sabemos que estão desorganizados, pois assim ele consegue obter um bom pivô a cada chamada recursiva. Para o Quicksort obter um bom pivô é algo essencial, já que um bom pivô evita chamadas recursivas desnecessárias oque garante melhor uso da memória e evita o estouro da Stack. No geral ambos os algoritmos de Quicksort entregam o resultado

em um bom tempo na casa dos minutos, mas sua maior vantagem em relação ao Merge e ao Heapsort é o pouco uso de memória, que a depender da situação pode ser algo a ser levado em conta na hora de escolher um algoritmo de ordenação.



Os algoritmos de Selection e Insertion sort são algoritmos o(n^2) que os tornam extremamente lentos já que ele tem que executar a quantidade de itens da entrada ao quadrado, claro em entradas pequenas eles ordenam os dados rapidamente, contudo o projeto proposto trata de arquivos com entradas enormes, levando horas para concluir a ordenação. Vale ressaltar que o insertion sort no melhor caso(quando os dados já estão ordenados) é extremamente rápido sendo executado por completo em alguns milissegundos. Isso se deve ao fato de que internamente o algoritmo verifica se os dados já estão ordenados, e caso esteja ele sai do laço while.

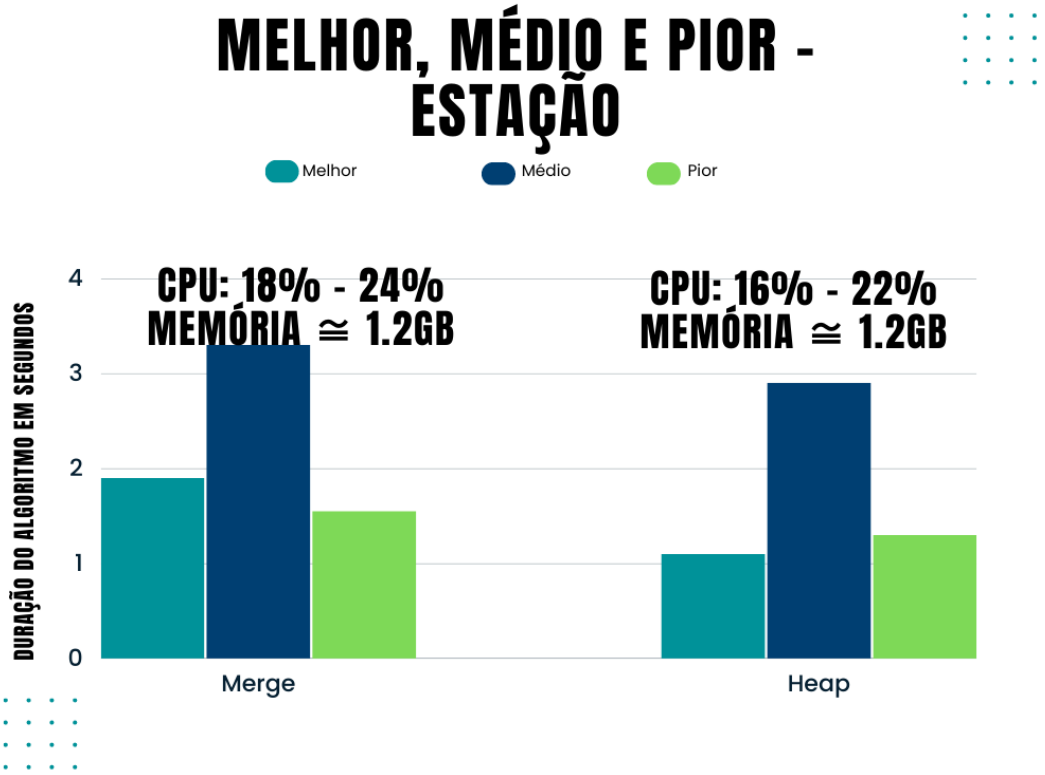
Vale ressaltar que pela natureza dos dados a serem tratados ficou inviável a utilização do counting sort para a data.

Estação - Tempo de execução

Para as ordenações baseadas na ordem alfabética do campo "station_name" do arquivo "LAMetroTrips.csv", os resultados foram medidos de duas formas distintas. Devido à grande discrepância nos resultados obtidos pelos cinco algoritmos avaliados, no primeiro gráfico o tempo de execução foi medido em segundos, enquanto no segundo gráfico foi medido em minutos.

Ao analisar os tempos de execução das ordenações, fica claro que tanto o Merge Sort quanto o Heapsort apresentaram resultados superiores em todos os casos avaliados,

mostrando-se mais eficientes em termos de tempo de execução. No entanto, esses dois algoritmos consomem mais memória durante o processo de ordenação dos valores em relação aos demais algoritmos avaliados.

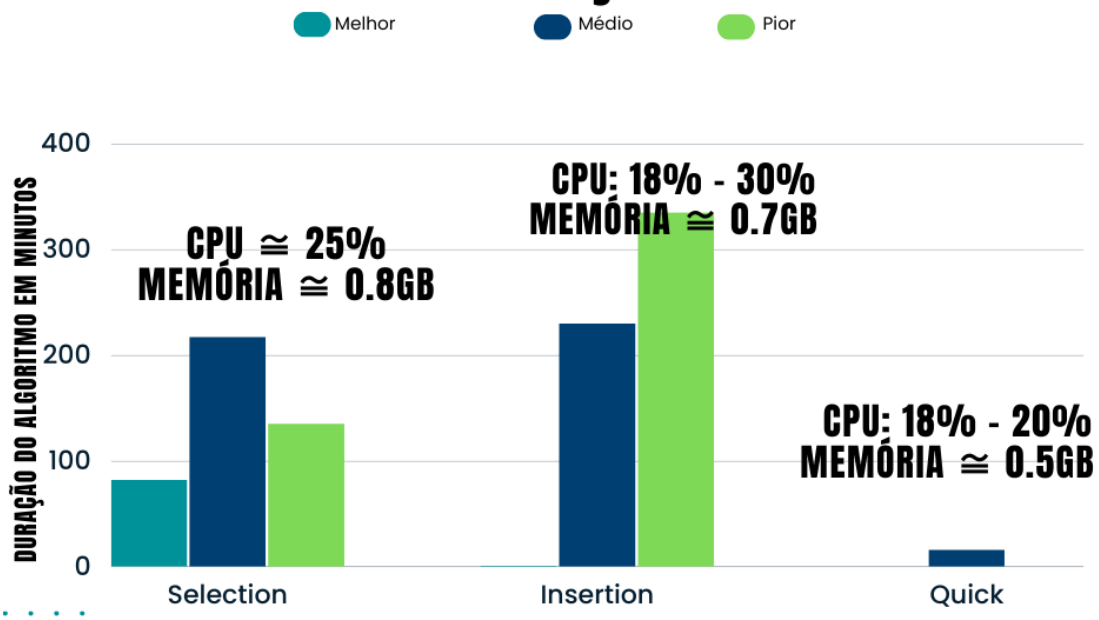


Continuando a análise dos resultados, observou-se que o algoritmo Selection Sort apresentou um tempo de execução consideravelmente maior em relação aos algoritmos anteriores. Entretanto, em termos de consumo de memória durante a ordenação, ele mostrou-se mais eficiente, utilizando cerca de 66% menos memória que o Merge Sort e Heap Sort.

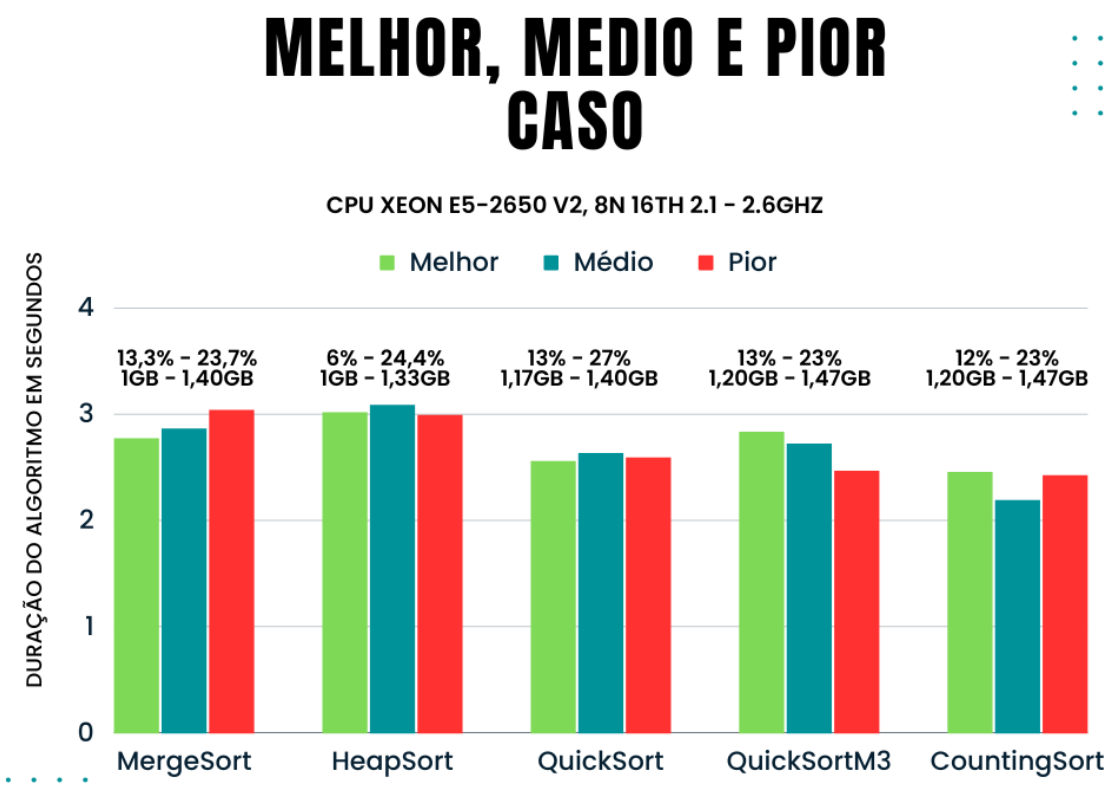
No caso do Insertion Sort, os resultados variaram bastante dependendo do cenário. No melhor caso, onde os valores já se encontravam em ordem alfabética, o algoritmo executa em apenas alguns segundos. Porém, nos outros dois cenários, o tempo de execução aumentou significativamente, chegando a levar horas para ordenar os dados.

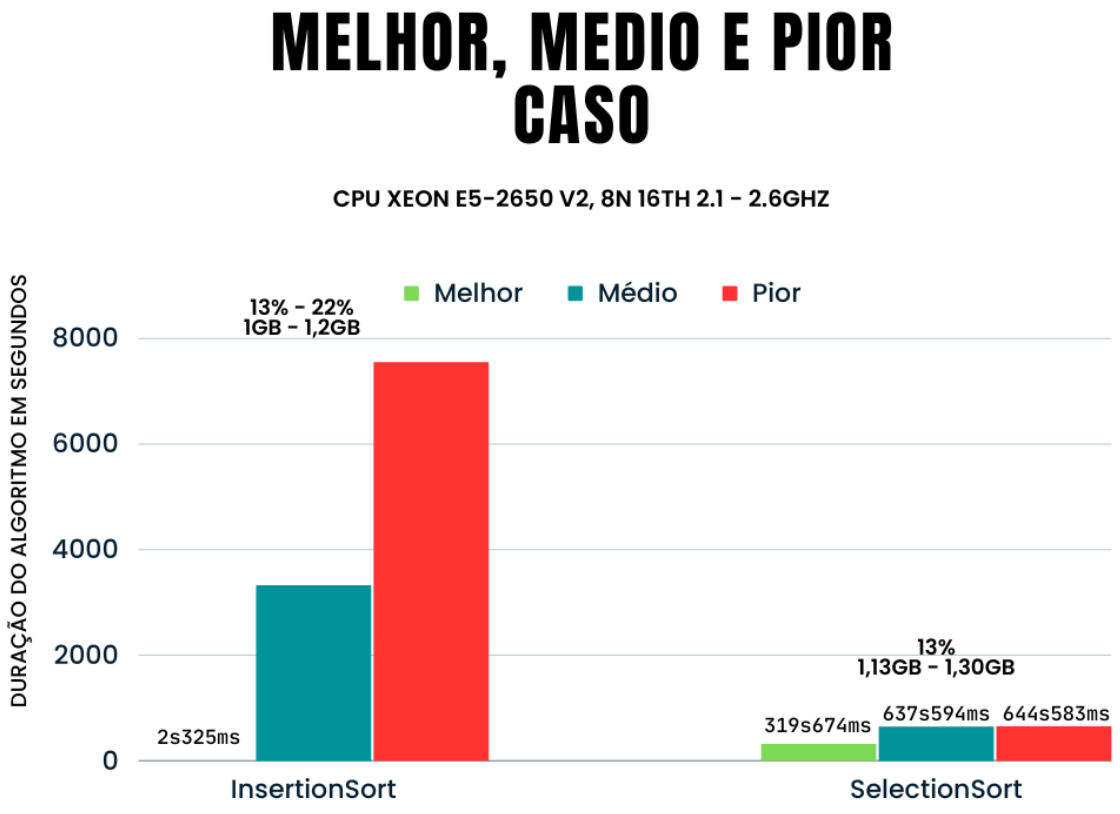
Já o Quicksort, apresentou um tempo de execução razoável para o caso médio, sendo executado em 15 minutos e utilizando menos memória que os algoritmos anteriores. No entanto, para os casos melhor e pior, a ordenação não foi realizada.

MELHOR, MÉDIO E PIOR - ESTAÇÃO



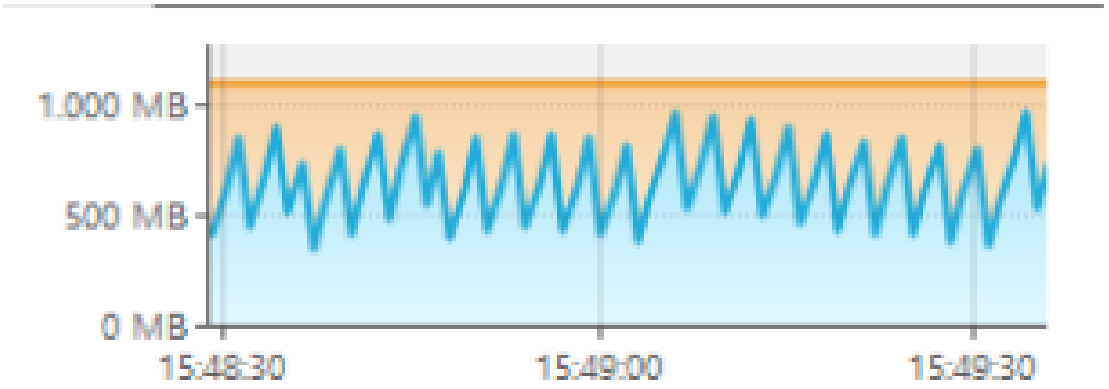
Duração - Tempo de Execução





O algoritmo counting sort se mostrou mais eficiente nesse tipo de caso, pois ele é usado para fazer a contagem dos números de forma linear, guardar as informações e depois escrever em um novo array a quantidade de números, e isso acaba ordenando os números, como a duração é feita em minutos, ele se mostrou bastante eficiente por não fazer nenhuma comparação complexa.

O segundo algoritmo mais eficiente foi o quickSort, mesmo que ele esteja na mesma complexidade dos algoritmos da imagem 1, e isso se dá por conta que ele tem constante menores e utiliza menos comparações, usufruindo do pivô escolhido, porém, esse algoritmo tem um problema, ele é ineficiente em casos de muitos dados por conta da utilização recursiva do Quicksort que tem efeito colateral como o estouro da memória Stack do programa, isso ocorre principalmente quando o algoritmo escolhe um pivô ruim, o'que ocorre, então, foi necessário gerar um algoritmo de pilha ao invés utilizar recursividade, e isso fez o ficar mais otimizado e livre de erros.



Utilização de memória QuickSort em pilha

Diferente de todos os métodos de ordenação existentes aqui, o quickSort em pilha é bem diferente de seus companheiros, pois enquanto todos os outros 6 algoritmos têm uma utilização linear da memória, ou seja, a utilização é contínua e nunca muda até o fim da ordenação, o quickSort muda a utilização da memória a cada ciclo da pilha e isso é evidente na imagem acima.

Analisando o resultado de ambas as prints, percebe-se que em relação ao uso de CPU e memória RAM, todos os algoritmos estão muito próximos, porém nos algoritmos que são mais rápidos, não é possível calcular com exatidão a utilização média de CPU, mas nos sorts insertion e selection por ser mais demorados, geralmente eles seguiam em 13% de utilização e após 40 minutos seguiam em 22% de utilização até o fim da ordenação.

Os sorts da imagem dois foram separados dos demais por conta da discrepância de valores, e isso acontece pois a complexidade do algoritmo Insertion e selection é de $C(n) = O(n^2)$ e isso quer dizer que o tempo aumenta em relação a quantidade de dados, quanto maior dados existir, pior fica o tempo do algoritmo. É importante comentar melhor caso do insertion Sort que ficou na média de tempo dos algoritmos de ordem superior, e isso só acontece pois o insertion percorre todos os elementos e vai ordenando a esquerda toda vez que achar um novo maior, ou seja, no melhor caso todos os elementos estão ordenados, então ele apenas faz as comparações, resumindo ele apenas percorre os elementos, porém isso vai piorando a maneira que os elementos vão ficando cada vez mais desordenados.

Já a maioria dos demais algoritmos utilizam complexidade $C(n) = O(n \log n)$, isso quer dizer que a quantidade de elementos não interferem negativamente no tempo de execução, pois a quantidade de números nunca será maior.