Laboratório de Estrutura de Dados

# Segunda versão do projeto da disciplina

## Alteração dos projetos UT1

Dupla: Kaio Emanuel Rosemiro de Carvalho e Deyvid Jerônimo de Araújo Macêdo

# 1. Introdução

Neste trabalho, será realizada uma análise comparativa de algoritmos de ordenação em diferentes conjuntos de testes. Algoritmos de ordenação são empregados para organizar elementos em uma sequência específica, tal como ordenar um conjunto de números e/ou strings em ordem crescente ou decrescente. Contudo, é importante ressaltar que, diferentemente do projeto anterior, não estamos focando na criação de novos algoritmos de ordenação.

Na nova versão deste projeto, estamos adotando uma abordagem inovadora, concentrando nossos esforços na remodelação do sistema. Estamos explorando estratégias mais eficientes ao incorporar estruturas de dados como Listas, ArrayLists, Pilhas e Collections. Ao invés de criar algoritmos do zero, estamos ajustando as estratégias de ordenação existentes para melhorar a eficiência do projeto.

Os principais objetivos deste trabalho permanecem inalterados e agora incluem a consideração das estratégias de estruturas de dados adotadas. Estes objetivos são:

● Implementar e comparar o desempenho dos algoritmos de ordenação mencionados em diversos cenários de teste.

● Realizar análises estatísticas do desempenho de cada algoritmo, levando em consideração o cenário da amostra, o seu campo de ordenação e o tempo de execução.

● Investigar como os diferentes cenários de teste impactam o desempenho dos algoritmos de ordenação.

● Fornecer informações relevantes acerca da eficiência e aplicabilidade de cada algoritmo em diversas situações.

Essa abordagem visa não apenas melhorar a eficiência das ordenações, mas também ressalta a importância da escolha de estruturas de dados apropriadas para alcançar melhores resultados no contexto deste projeto remodelado.

# 2. Descrição geral sobre o método utilizado

A relevância da análise comparativa continua a ser fundamental, estando intrinsecamente ligada à disparidade de desempenho observada entre diferentes estratégias de ordenação. Contudo, nesta nova versão do projeto, nossa abordagem não se restringe apenas à comparação de algoritmos de ordenação. Em vez disso, concentramos na remodelação do projeto, otimizando as ordenações por meio da utilização de estruturas de dados mais eficientes, tais como Listas, ArrayLists, Pilhas e Collections.

Assim como na versão anterior, a escolha da estratégia de ordenação mais apropriada continua a ser um ponto crucial. Esta escolha é agora respaldada pelas peculiaridades das estruturas de dados selecionadas e de como essas estruturas são implementadas em nosso projeto remodelado.

Através desta análise comparativa, buscamos identificar as virtudes e limitações de cada estratégia, avaliando sua eficiência em diferentes cenários de teste. Os resultados obtidos não só oferecerão insights valiosos para o desenvolvimento de aplicações que requerem operações de ordenação, mas também auxiliarão na seleção da estratégia mais adequada para atender às demandas específicas de cada situação.

Dessa forma, a consideração das estratégias de estruturas de dados no âmbito desta análise comparativa proporcionará uma compreensão mais abrangente do desempenho, permitindo uma escolha mais informada e eficiente das estratégias de ordenação no contexto deste projeto remodelado.

## Descrição geral do ambiente de testes

Os insumos utilizados para a execução dos algoritmos de ordenação foram obtidos a partir das especificações do dispositivo empregado, um Desktop que abriga um processador Intel(R) Core(TM) i3-3240 CPU @ 3.40GHz, 12 GB de RAM, GT 730 GPU 4GB

Nvidia e um Sistema Operacional Windows 11 de 64 bits. Essas informações são fundamentais para contextualizar o ambiente em que os testes foram conduzidos e podem exercer influência sobre os resultados obtidos

# 3. Resultados e Análise

## 3.1 Heap Sort

O Heap Sort é um algoritmo de ordenação eficiente e versátil que oferece bom desempenho e uso eficiente de memória. É uma escolha sólida para ordenar grandes conjuntos de dados, embora possa não ser a opção mais rápida em conjuntos de dados quase ordenados ou pequenos.

Principais Características:

● É um algoritmo 'in-place', ou seja, ele ordena os elementos no próprio array de entrada.

● Possui uma complexidade de tempo média e pior caso de O(n log n)

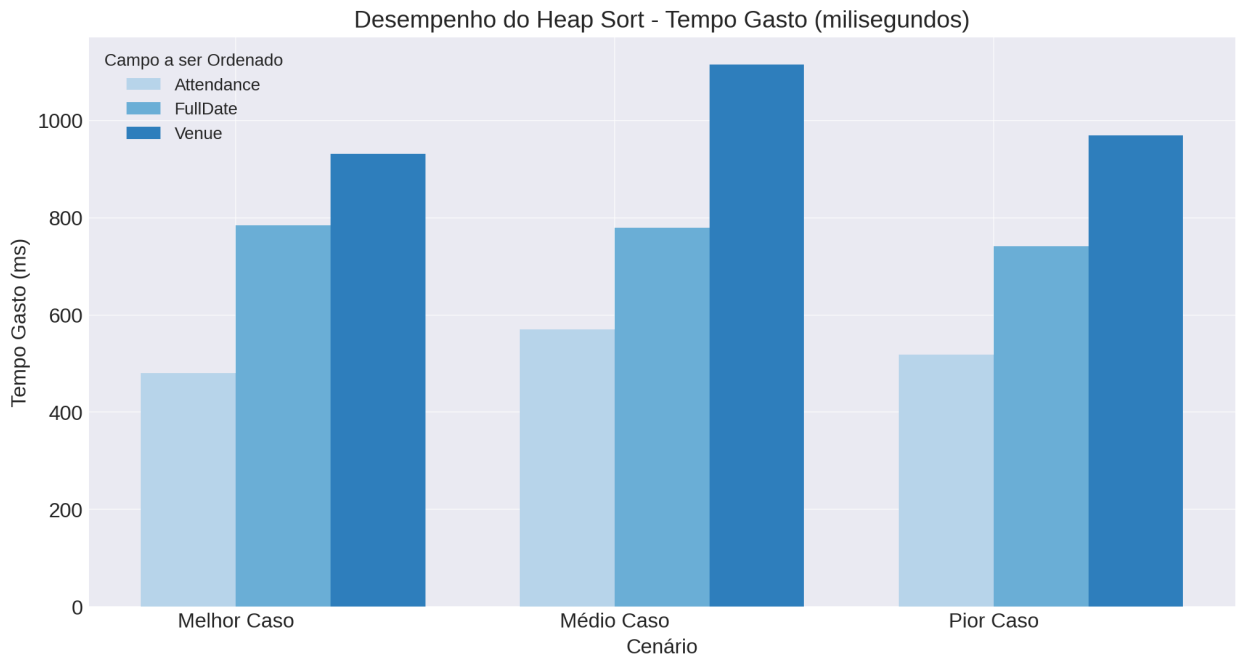
● Não é um algoritmo de ordenação estável, ou seja, ele pode alterar a ordem relativa de elementos com chaves iguais durante o processo de ordenação.

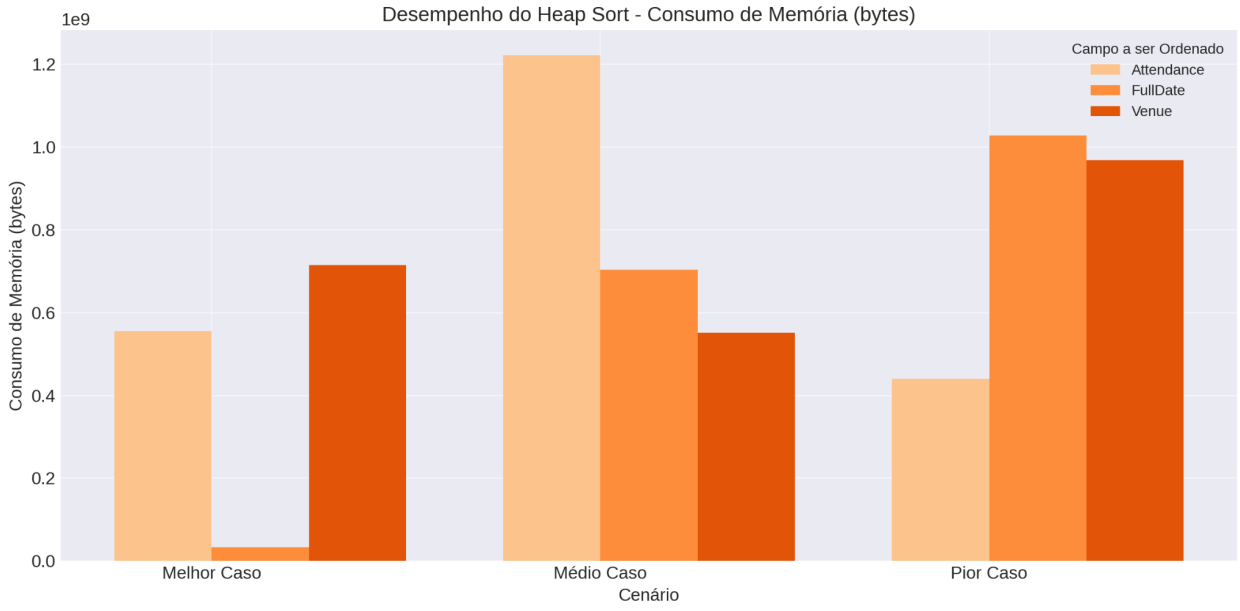
● Possui um desempenho relativamente constante em diferentes cenários de entrada, tornando-o uma escolha sólida para muitas aplicações.

Aqui estão os resultados obtidos na tabela:

| Método de Ordenação Heap Sort | | | | |
|---|---|---|---|---|
| Campo a ser Ordenado | Cenário | Tamanho da Amostra | Tempo Gasto (milisegundos) | Consumo de Memória (bytes) |
| Venue | Melhor Caso | 25725 | 932 | 714681072 |
| Venue | Médio Caso | 25725 | 1115 | 551955224 |
| Venue | Pior Caso | 25725 | 969 | 965891992 |
| Attendance | Melhor Caso | 25725 | 481 | 554873576 |
| Attendance | Médio Caso | 25725 | 570 | 1222286808 |
| Attendance | Pior Caso | 25725 | 519 | 440668584 |
| FullDate | Melhor Caso | 25725 | 784 | 327757832 |
| FullDate | Médio Caso | 25725 | 779 | 703224816 |
| FullDate | Pior Caso | 25725 | 742 | 1028018088 |

E sua representação gráfica:




Estratégias abordadas:

1. ArrayLists: Usados para armazenar dados de arquivos CSV devido ao seu acesso rápido e redimensionamento dinâmico.
2. Collections.swap(): Empregado para trocas eficientes durante a ordenação, simplificando o código.

3. Cenários de Ordenação: Implementados para testar o desempenho do algoritmo em diferentes conjuntos de dados.
4. Medição de Desempenho: Tempo de execução e uso de memória são medidos para avaliar a eficiência do algoritmo.
5. Função heapify Adaptada: Mantém a propriedade do heap, adaptada para ordenar com base em uma coluna específica dos dados CSV.
6. Tratamento de Cabeçalho de CSV: O cabeçalho é ignorado durante a leitura para evitar confusão com dados reais.
7. Limpeza e Comparação de Strings: As strings são normalizadas para uma comparação consistente, removendo variações de formatação.

## 3.2 Merge Sort

O Merge Sort é um algoritmo de ordenação baseado na técnica de divisão e conquista. Ele divide a lista não ordenada em sub-listas menores, ordena cada sub-lista e, em seguida, combina essas sub-listas ordenadas para obter uma lista ordenada completa.

Principais Características:

● Divide a lista original em duas metades iguais, recursivamente.

● Ordena cada metade separadamente.

● Combina as duas metades ordenadas em uma única lista ordenada.

● Tem uma complexidade de tempo média e no pior caso de O(n log n), tornando-o eficiente para grandes conjuntos de dados.

● É estável, o que significa que mantém a ordem relativa de elementos iguais.

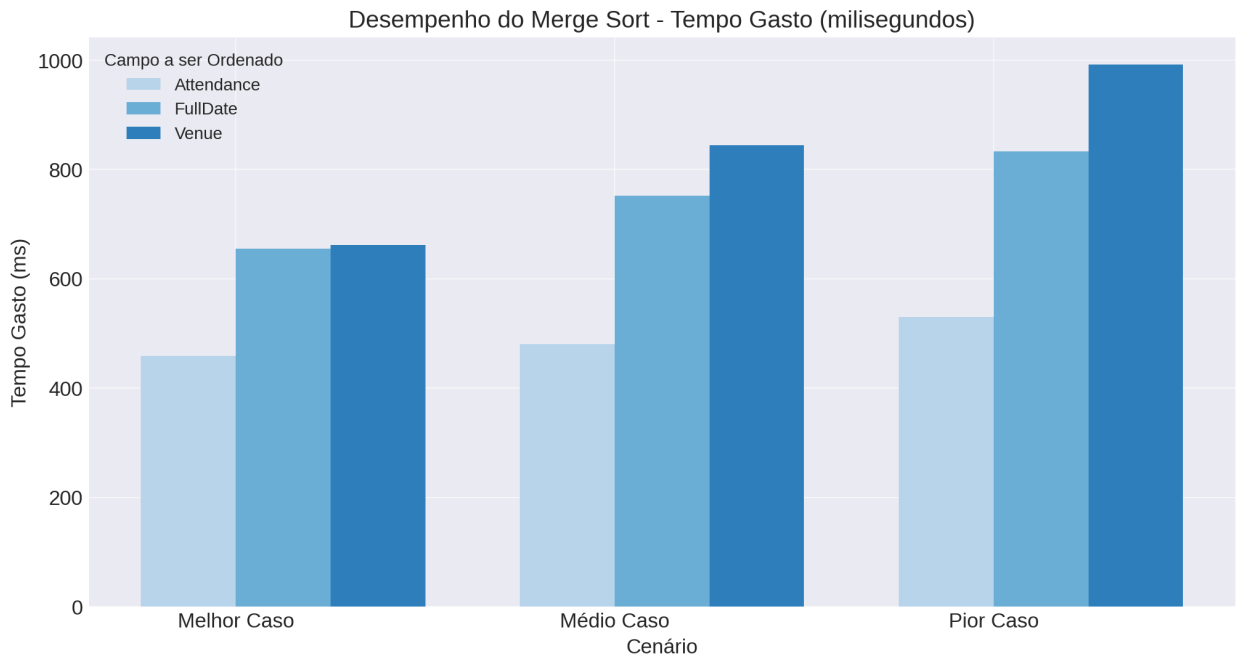
● Requer espaço adicional para armazenar as sub-listas temporárias durante a fusão.

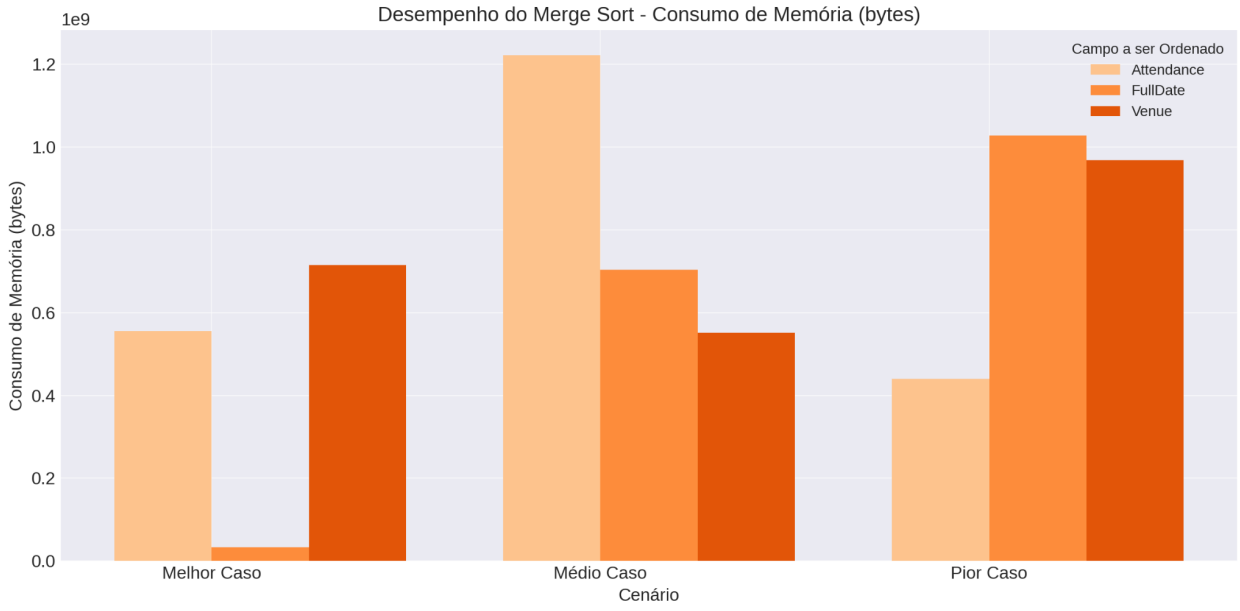
O Merge Sort é eficiente para valores constantes e ordenados, mas pode ser afetado pela presença de dados desordenados, especialmente em determinadas posições da amostra.

Aqui estão os dados obtidos na tabela:

| Método de Ordenação Heap Sort | | | | |
|---|---|---|---|---|
| Campo a ser Ordenado | Cenário | Tamanho da Amostra | Tempo Gasto (milisegundos) | Consumo de Memória (bytes) |
| Venue | Melhor Caso | 25725 | 932 | 714681072 |
| Venue | Médio Caso | 25725 | 1115 | 551955224 |
| Venue | Pior Caso | 25725 | 969 | 965891992 |
| Attendance | Melhor Caso | 25725 | 481 | 554873576 |
| Attendance | Médio Caso | 25725 | 570 | 1222286808 |
| Attendance | Pior Caso | 25725 | 519 | 440668584 |
| FullDate | Melhor Caso | 25725 | 784 | 327757832 |
| FullDate | Médio Caso | 25725 | 779 | 703224816 |
| FullDate | Pior Caso | 25725 | 742 | 1028018088 |

E sua representação gráfica:




Estratégias abordadas:

1. Uso de ArrayLists: Assim como no Heap Sort, ArrayLists são utilizados para armazenar dados de arquivos CSV. Eles são escolhidos pela facilidade de uso, redimensionamento dinâmico e eficiência em operações de acesso e inserção.

2. Criação de Três Cenários de Ordenação: O código prepara três cenários diferentes (melhor, médio e pior caso) para testar o desempenho do algoritmo Merge Sort. Essa abordagem permite avaliar a eficiência do algoritmo em diferentes condições de dados.
3. Medição de Desempenho: A classe mede e imprime o tempo de execução e o consumo de memória do processo de ordenação, oferecendo uma análise quantitativa do desempenho do algoritmo.
4. Implementação do Merge Sort: O algoritmo é implementado de forma recursiva, dividindo o conjunto de dados em subconjuntos cada vez menores, ordenando-os e, em seguida, mesclando-os de volta de forma ordenada.
5. Função de Mesclagem (Merge): Essencial no Merge Sort, a função merge combina dois subconjuntos ordenados em um único conjunto ordenado. A comparação dos elementos é feita com base no venueIndex, indicando uma ordenação personalizada baseada em uma coluna específica dos dados.
6. Ignorar Cabeçalho do CSV Durante a Leitura: O código ignora a primeira linha do arquivo CSV (cabeçalho) para garantir que apenas dados relevantes sejam processados.
7. Normalização e Comparação de Strings: A função compareStrings limpa e compara strings de forma consistente, removendo caracteres especiais e ignorando diferenças entre maiúsculas e minúsculas.
8. Cópia de Arquivos para Cenários Diferentes: O código copia o arquivo de entrada para criar cenários de teste diferentes, o que é uma maneira prática de preparar os dados para a análise de desempenho.

### 3.3 Quick Sort

O Quick Sort é um algoritmo de ordenação baseado na técnica de divisão e conquista. Ele escolhe um elemento pivô na lista e rearranja os elementos de forma que todos os elementos menores que o pivô estejam à esquerda e todos os elementos maiores estejam à direita.

Em seguida, o Quick Sort é aplicado recursivamente às sub-listas à esquerda e à direita do pivô até que toda a lista esteja ordenada.

Principais Características:

● Escolha um elemento pivô da lista.

● Divide a lista em duas sub-listas, uma contendo elementos menores que o pivô e outra com elementos maiores.

● Aplica o Quick Sort recursivamente a ambas as sub-listas.

● Tem uma complexidade de tempo média e no pior caso de O(n log n), tornando-o eficiente para grandes conjuntos de dados.

● É in-place, o que significa que ordena a lista sem a necessidade de espaço de memória adicional.

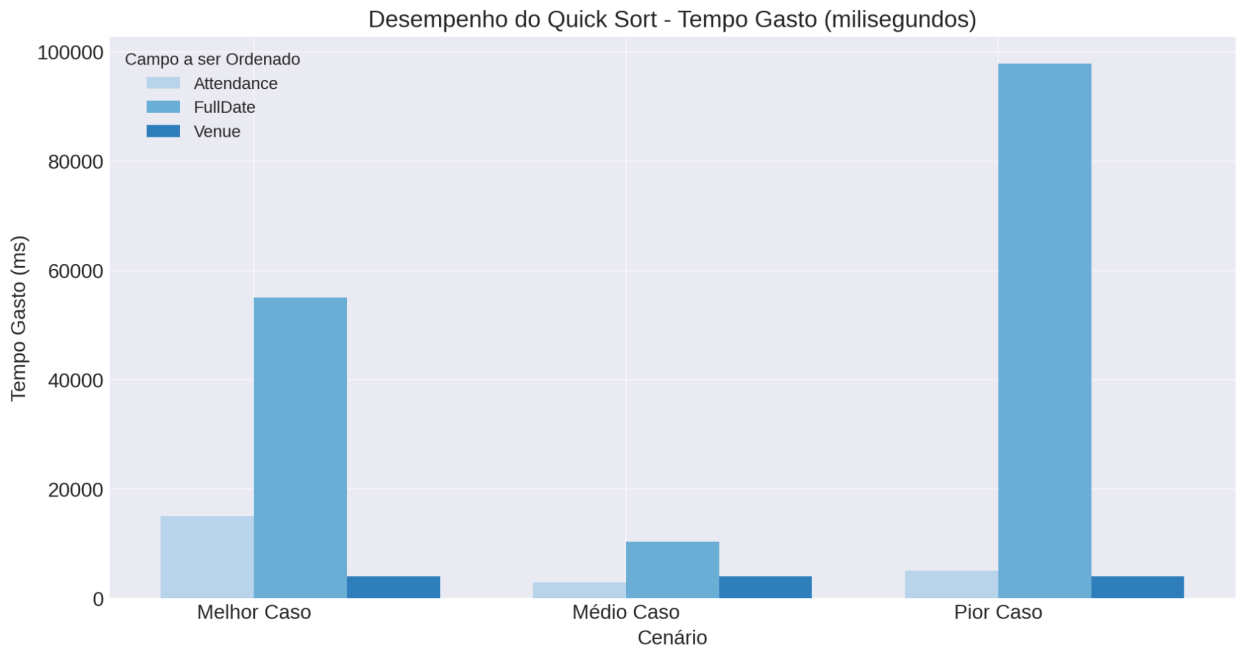
● Pode ser sensível à escolha do pivô em casos extremos, mas estratégias de escolha inteligente do pivô (como o pivô de mediana de três) ajudam a mitigar esse problema.

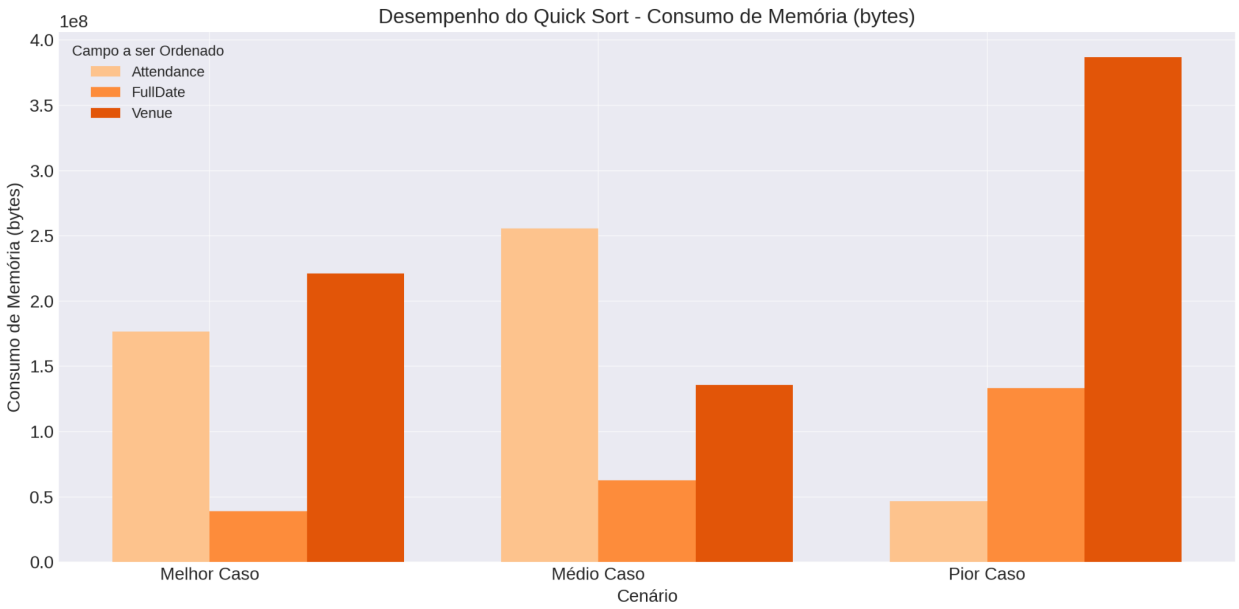
No Quick Sort, no código que usamos ele encontrou muitos problemas em ordenar um array de argumentos de valores muitos próximos um dos outros, como foi possível ver no gráfico. Na tabela logo abaixo, você poderá ver precisamente o tempo gasto na execução.

Aqui estão os dados obtidos na tabela:

| Método de Ordenação QuickSort | | | | |
|---|---|---|---|---|
| Campo a ser Ordenado | Cenário | Tamanho da Amostra | Tempo Gasto (milisegundos) | Consumo de Memória (bytes) |
| Venue | Melhor Caso | 25725 | 4020 | 221061272 |
| Venue | Médio Caso | 25725 | 4035 | 135496672 |
| Venue | Pior Caso | 25725 | 3978 | 386919104 |
| Attendance | Melhor Caso | 25725 | 15030 | 176763544 |
| Attendance | Médio Caso | 25725 | 2860 | 255816224 |
| Attendance | Pior Caso | 25725 | 4971 | 46668440 |
| FullDate | Melhor Caso | 25725 | 55051 | 39098440 |
| FullDate | Médio Caso | 25725 | 10345 | 62491648 |
| FullDate | Pior Caso | 25725 | 97848 | 133402608 |

E sua representação gráfica:






Estratégias Abordadas:

1. Uso de Array Bi-dimensional: Diferente do uso de ArrayList, o código utiliza um array bi-dimensional (String[][]) para armazenar os dados do arquivo CSV. Essa escolha pode ser devido à simplicidade e ao acesso direto aos elementos do array, o que é eficiente para a manipulação de dados durante o Quick Sort.
2. Criação de Três Cenários de Ordenação: Assim como nos outros algoritmos, são criados cenários de melhor, médio e pior caso para testar o desempenho do Quick Sort sob diferentes condições de dados.
3. Medição de Desempenho: O código mede e imprime o tempo de execução e o consumo de memória durante o processo de ordenação, proporcionando uma análise quantitativa do desempenho do algoritmo.
4. Implementação Iterativa do Quick Sort: Ao contrário da abordagem recursiva tradicional, este código implementa o Quick Sort de forma iterativa, usando uma pilha (Stack) para gerenciar os intervalos de índices a serem ordenados. Isso pode ajudar a evitar o problema de estouro de pilha em casos de grandes conjuntos de dados.

5. Escolha do Pivô: O algoritmo utiliza um pivô aleatório para a divisão dos dados, o que pode ajudar a evitar o desempenho pior caso em conjuntos de dados já ordenados ou quase ordenados.
6. Ignorar Cabeçalho do CSV Durante a Leitura: O código ignora a primeira linha do arquivo CSV (cabeçalho) para garantir que apenas dados reais sejam processados.
7. Normalização e Comparação de Strings: A função compareStrings limpa e compara strings de forma consistente, removendo caracteres especiais e normalizando as maiúsculas e minúsculas.
8. Inversão de Dados para o Pior Caso: O código inverte os dados após a ordenação para criar o cenário de pior caso, explorando o desempenho do Quick Sort com dados em ordem decrescente.
9. Copiar Arquivo para Cenário Médio: O arquivo de entrada é copiado para criar o cenário médio, uma abordagem prática para testar o algoritmo com dados não manipulados.

## 3.4 Quick Sort (Mediana 3)

O algoritmo Quick Sort, com a escolha da mediana de três como estratégia de pivoteamento, é um algoritmo de ordenação eficiente com características notáveis:

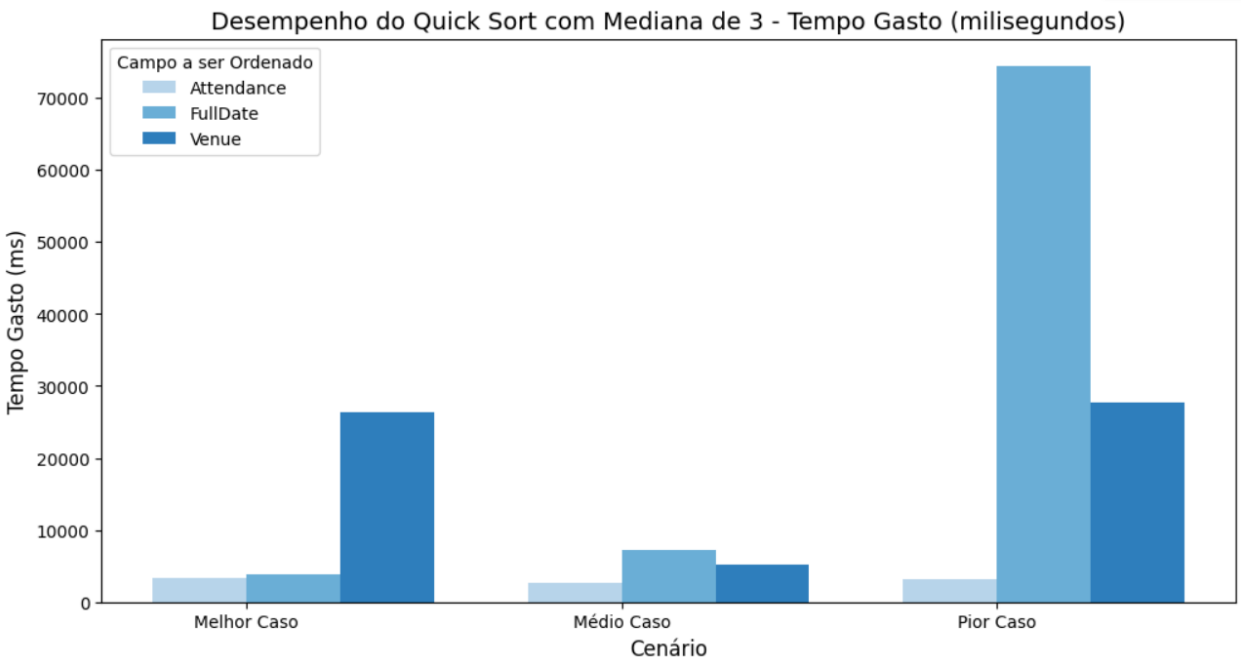
● Complexidade de Tempo: Em média, o Quick Sort tem uma complexidade de tempo de O(n log n), o que o torna muito rápido para grandes conjuntos de dados. No entanto, seu desempenho no pior caso pode degradar para O(n^2) em cenários específicos, embora essa situação seja rara quando a estratégia de mediana de três é utilizada.

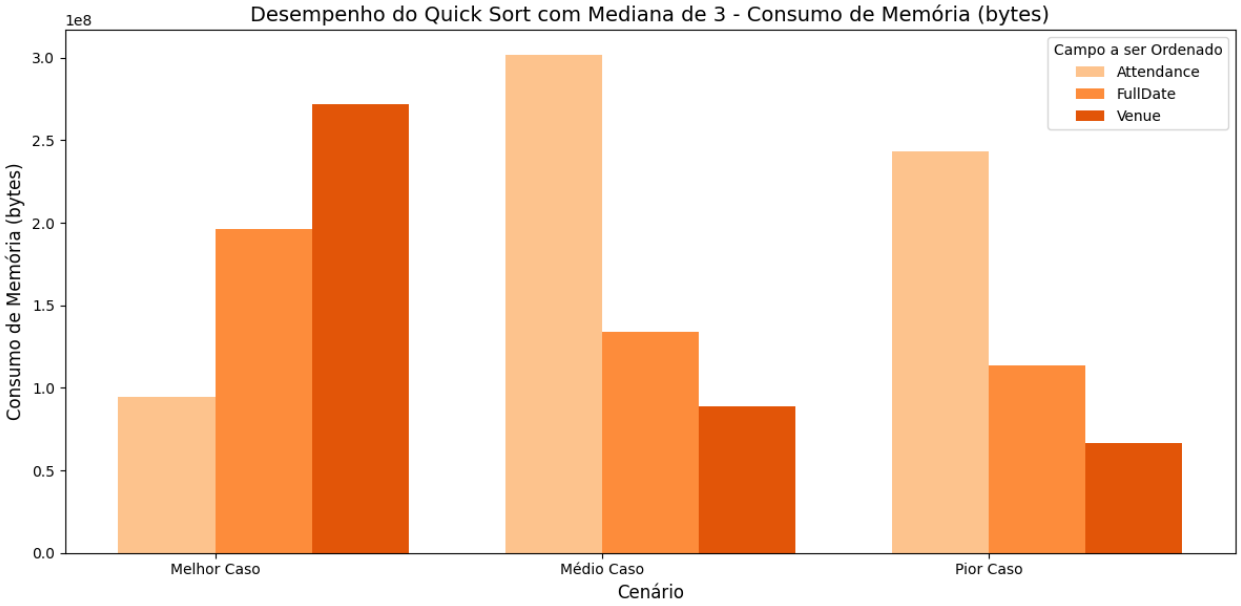
● Divisão e Conquista: O Quick Sort segue o paradigma de "divisão e conquista", onde divide o conjunto de dados em partições menores, ordena essas partições e, em seguida, as junta para obter a lista ordenada. Isso permite que o Quick Sort tenha um desempenho rápido na maioria dos casos.

● Estratégia de Pivoteamento com Mediana de Três: A escolha da mediana de três como pivô ajuda a evitar o pior cenário de desempenho em muitos casos. Ela envolve a escolha do pivô como a mediana entre o primeiro, o último e o elemento do meio da lista. Isso ajuda a reduzir a chance de escolher um pivô ruim e, assim, melhora o desempenho geral do algoritmo.

Foi possível observar que, com o uso da estratégia de pivoteamento com mediana de 3, que os testes levaram um tempo de execução maior nos melhores casos em relação aos outros nos diferentes cenários. É perceptível ter uma visão mais detalhada com a sua representação em tabela e gráficos abaixo:

| Método de Ordenação QuickSortMediana3 | | | | |
|---|---|---|---|---|
| Campo a ser Ordenado | Cenário | Tamanho da Amostra | Tempo Gasto (milisegundos) | Consumo de Memória (bytes) |
| Venue | Melhor Caso | 25725 | 26342 | 272163720 |
| Venue | Médio Caso | 25725 | 5158 | 88894240 |
| Venue | Pior Caso | 25725 | 27659 | 66679448 |
| Attendance | Melhor Caso | 25725 | 3384 | 94739144 |
| Attendance | Médio Caso | 25725 | 2687 | 301701056 |
| Attendance | Pior Caso | 25725 | 3233 | 243491096 |
| FullDate | Melhor Caso | 25725 | 3889 | 196171856 |
| FullDate | Médio Caso | 25725 | 7166 | 134087760 |
| FullDate | Pior Caso | 25725 | 74480 | 113566752 |




Estratégias Abordadas:

1. ArrayList para Armazenamento de Dados: O código utiliza ArrayList<String[]> para armazenar dados de arquivos CSV. Esta escolha combina a eficiência do acesso direto de arrays com a flexibilidade de redimensionamento dinâmico dos ArrayLists.

2. Mediana de 3 para Escolha do Pivô: Ao invés de escolher um pivô aleatório ou fixo, o algoritmo utiliza a técnica da mediana de 3 (escolhendo o pivô entre o primeiro, o meio e o último elemento do subconjunto). Isso visa melhorar o desempenho do QuickSort, especialmente em conjuntos de dados que podem causar casos de pior desempenho com escolhas de pivôs tradicionais.
3. Cenários de Ordenação (Melhor, Médio e Pior Caso): O código cria três cenários de ordenação para testar o algoritmo sob diferentes condições de dados, semelhante às outras implementações de algoritmos de ordenação.
4. Medição do Tempo de Execução e Consumo de Memória: Assim como nas outras implementações, este código mede o tempo de execução e o uso de memória, permitindo uma avaliação quantitativa do desempenho do algoritmo.
5. Tratamento de Cabeçalho e Formatação do CSV: O código ignora a primeira linha (cabeçalho) durante a leitura do arquivo CSV e lida com potenciais problemas de formatação (como aspas e espaços).
6. Comparação de Strings Normalizadas: As strings são normalizadas (removendo caracteres especiais e convertendo para minúsculas) antes da comparação, garantindo uma ordenação consistente e justa.
7. Reversão de Dados para o Pior Caso: Após a ordenação, os dados são invertidos para criar o cenário de pior caso, permitindo testar o algoritmo com dados em ordem decrescente.
8. Uso de Collections.swap para Troca de Elementos: Este método é usado para trocar elementos na lista, simplificando o código e potencialmente aproveitando otimizações internas.

## 3.5 Counting Sort

O Counting Sort é um algoritmo de ordenação eficiente, mas é adequado apenas para classificar números inteiros não negativos dentro de um intervalo específico. Ele funciona contando o número de ocorrências de cada elemento na lista original e, em seguida, usando essa contagem para reconstruir a lista ordenada.

Principais características:

● É eficiente quando a faixa dos números na lista é relativamente pequena.

● Requer conhecimento prévio da faixa dos números a serem ordenados.

● Conta o número de ocorrências de cada elemento e cria um array de contagem.

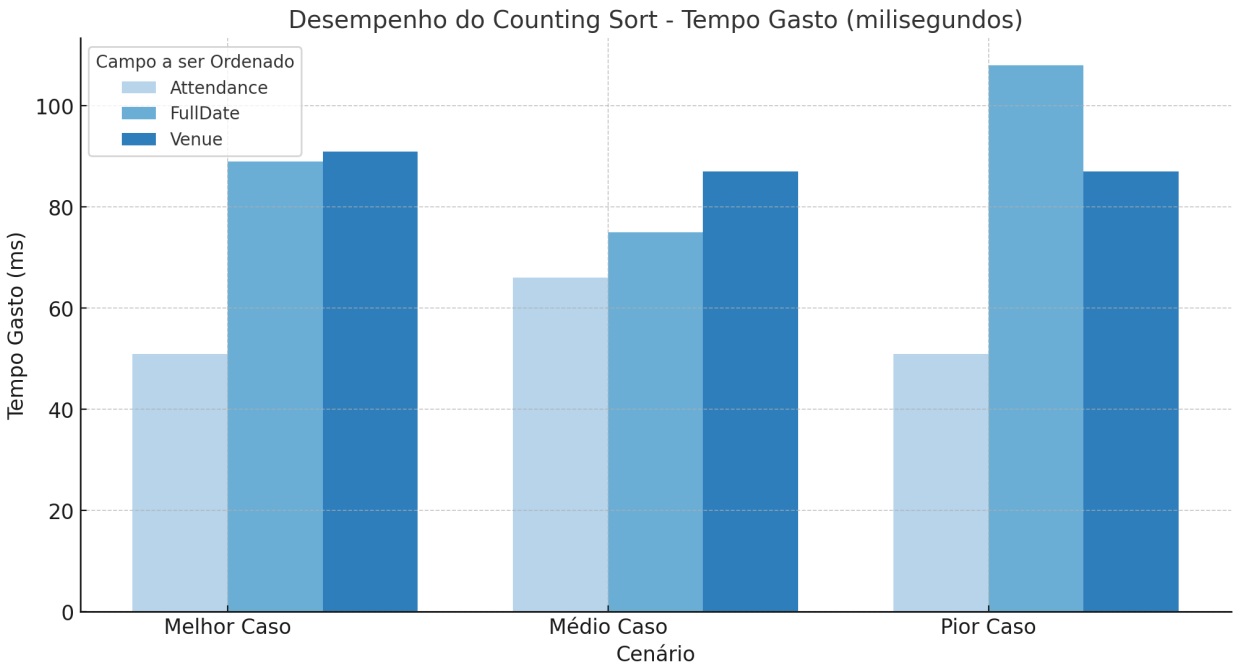
● Usa o array de contagem para reconstruir a lista ordenada.

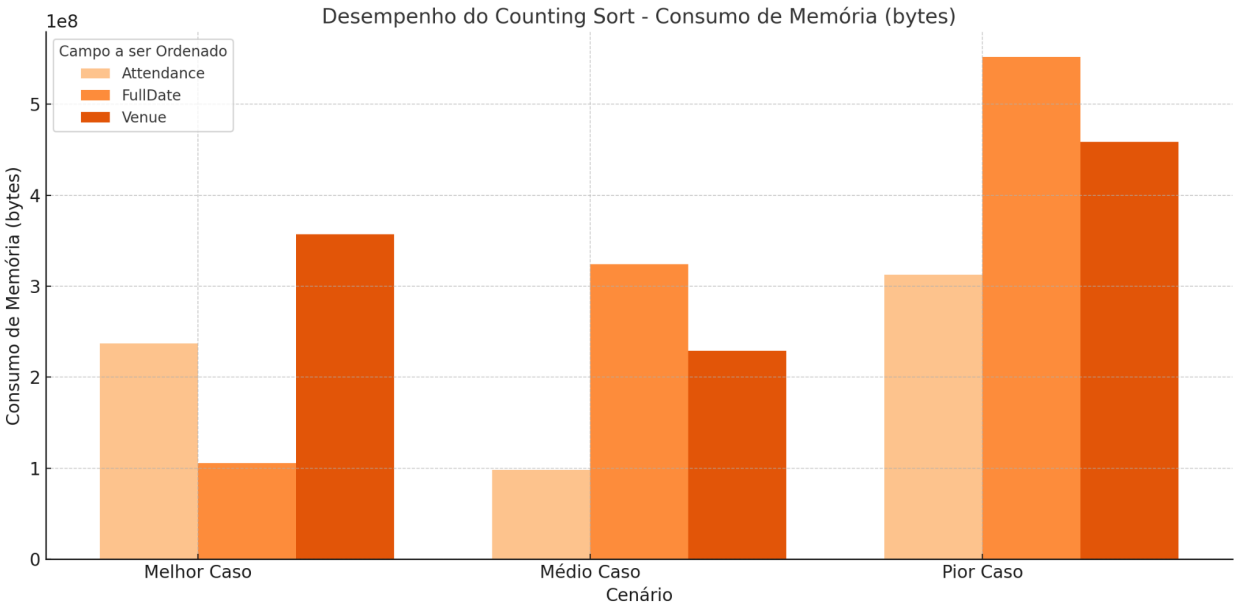
● Tem uma complexidade de tempo linear de O(n + k), onde 'n' é o número de elementos na lista e 'k' é a diferença entre o maior e o menor elemento.

● Não é in-place, pois requer espaço adicional para armazenar o array de contagem.

O Counting Sort é um algoritmo que consome bastante memória, uma vez que ele precisa alocar um vetor de contagem para prosseguir com suas ordenações. As representações abaixo evidenciam isso:

| Método de Ordenação Counting Sort | | | | |
|---|---|---|---|---|
| Campo a ser Ordenado | Cenário | Tamanho da Amostra | Tempo Gasto (milisegundos) | Consumo de Memória (bytes) |
| Venue | Melhor Caso | 25725 | 91 | 357156240 |
| Venue | Médio Caso | 25725 | 87 | 228824432 |
| Venue | Pior Caso | 25725 | 87 | 454828728 |
| Attendance | Melhor Caso | 25725 | 51 | 237243312 |
| Attendance | Médio Caso | 25725 | 66 | 98089272 |
| Attendance | Pior Caso | 25725 | 51 | 312938736 |
| FullDate | Melhor Caso | 25725 | 89 | 105794208 |
| FullDate | Médio Caso | 25725 | 75 | 324386184 |
| FullDate | Pior Caso | 25725 | 108 | 552085808 |




Estratégias Abordadas:

1. ArrayList para Armazenamento de Dados: O código utiliza ArrayList<String[]> para armazenar os dados dos arquivos CSV. Esta escolha é eficiente para a manipulação de dados e permite um redimensionamento dinâmico, sendo útil para a leitura de arquivos com um número desconhecido de linhas.

2. Criação de Três Cenários de Ordenação: O código prepara cenários de melhor, médio e pior caso para testar o desempenho do algoritmo Counting Sort em diferentes condições de dados.
3. Medição de Desempenho: Há uma medição do tempo de execução e do consumo de memória durante o processo de ordenação, proporcionando uma avaliação quantitativa do desempenho do algoritmo.
4. Normalização e Ordenação Única de Strings: Para lidar com a variedade de strings nos dados CSV, o algoritmo normaliza as strings (removendo caracteres não alfanuméricos e convertendo para minúsculas) e cria uma lista de strings únicas e ordenadas. Isso é essencial para o funcionamento eficiente do Counting Sort em dados textuais.
5. Implementação do Counting Sort para Strings: O Counting Sort é tradicionalmente usado para números inteiros, mas este código adapta o algoritmo para lidar com strings. Ele mapeia cada string normalizada e única para um índice em um array de contagem.
6. Contagem e Ordenação dos Dados: O algoritmo conta as ocorrências de cada string e usa essas contagens para posicionar cada elemento em sua posição correta em um novo ArrayList, efetivamente ordenando os dados.
7. Tratamento do Cabeçalho e Formatação do CSV: O código ignora a primeira linha (cabeçalho) durante a leitura do arquivo CSV e trata potenciais problemas de formatação.
8. Reversão de Dados para o Pior Caso: Após a ordenação, os dados são invertidos para criar o cenário de pior caso, permitindo testar o desempenho do algoritmo com dados em ordem decrescente.
9. Uso de Collections.swap para Troca de Elementos: Embora não usado explicitamente no processo de ordenação, Collections.swap é utilizado para inverter os dados no cenário de pior caso.

## 3.6 Insertion Sort

O Insertion Sort é um algoritmo de ordenação que constrói a lista ordenada um elemento de cada vez, comparando e movendo elementos para a posição correta conforme avança na lista.

Principais Características:

● Percorre a lista da esquerda para a direita, inserindo cada elemento na posição correta.

● Mantém uma parte da lista ordenada, à esquerda, e uma parte não ordenada, à direita.

● É eficiente para conjuntos de dados pequenos ou quase ordenados.

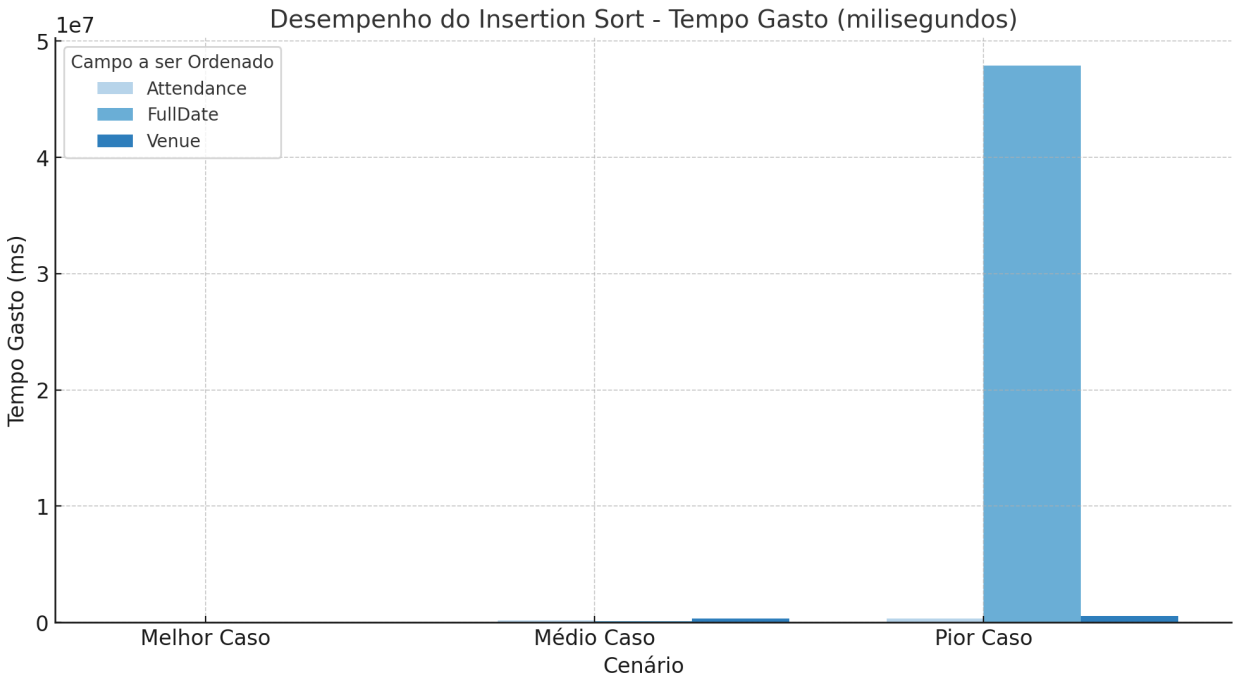
● Tem uma complexidade de tempo no pior caso de O(n^2), onde 'n' é o número de elementos na lista.

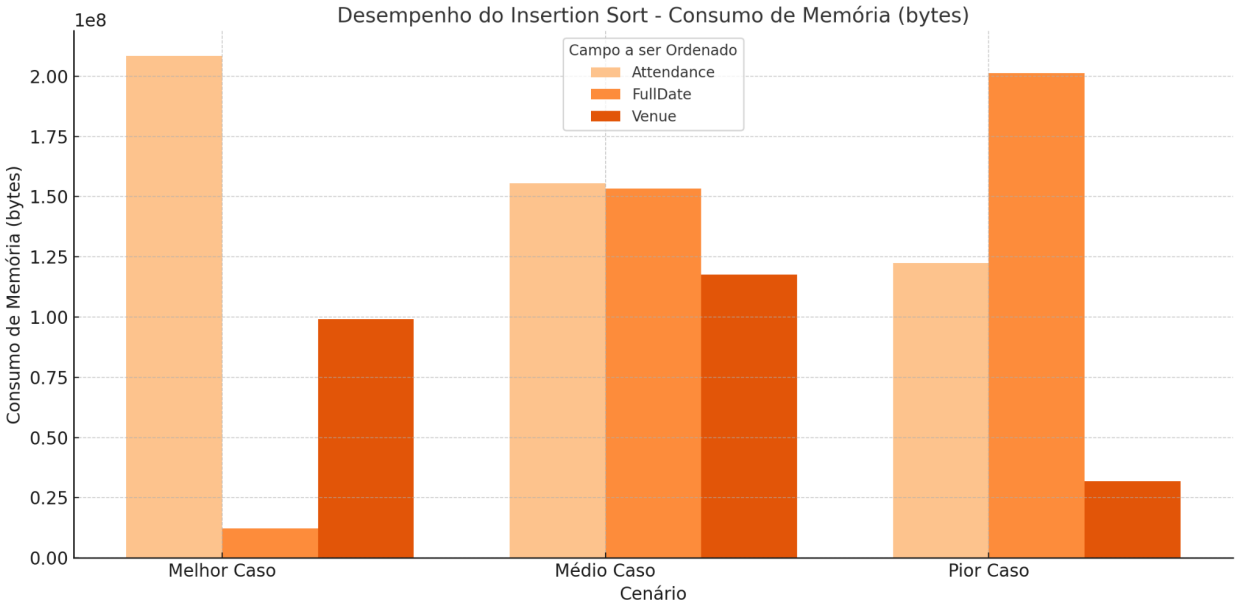
● É estável, o que significa que mantém a ordem relativa de elementos iguais.

● É um algoritmo in-place, o que significa que não requer espaço adicional para ordenar a lista.

Como foi possível fazer a análise no gráfico, o Insertion Sort se saiu muito bem até, lidando com os seus 'melhores casos' nos diferentes cenários de ordenação. Mas o mesmo não pode ser dito dos outros, quando ele encontrou uma grande massa de itens a serem ordenados, vale lembrar que o seu consumo de memória é alto nos estágios iniciais da ordenação (especialmente no pior caso), mas ao longo da execução do código ele vai se estabilizando. A seguir você verá com detalhes as representações gráficas:

| Método de Ordenação Insertion Sort | | | | |
|---|---|---|---|---|
| Campo a ser Ordenado | Cenário | Tamanho da Amostra | Tempo Gasto (milisegundos) | Consumo de Memória (bytes) |
| Venue | Melhor Caso | 25725 | 45 | 99231680 |
| Venue | Médio Caso | 25725 | 340164 | 117674352 |
| Venue | Pior Caso | 25725 | 562514 | 31877640 |
| Attendance | Melhor Caso | 25725 | 29 | 208440776 |
| Attendance | Médio Caso | 25725 | 189090 | 155619064 |
| Attendance | Pior Caso | 25725 | 331785 | 122528464 |
| FullDate | Melhor Caso | 25725 | 28 | 121251768 |
| FullDate | Médio Caso | 25725 | 131529 | 153399736 |
| FullDate | Pior Caso | 25725 | 47890032 | 201349136 |




Estratégias Abordadas:

1. Uso de ArrayList para Armazenamento de Dados: O código utiliza ArrayList<String[]> para armazenar os dados dos arquivos CSV. Isso é eficiente para manipular dados,

permitindo redimensionamento dinâmico, o que é útil ao lidar com arquivos de tamanho desconhecido.

2. Criação de Três Cenários de Ordenação: O código prepara três cenários diferentes (melhor, médio e pior caso) para avaliar o desempenho do algoritmo de ordenação Insertion Sort sob várias condições de entrada.
3. Medição de Desempenho: Durante o processo de ordenação, o código mede o tempo de execução e o consumo de memória, fornecendo uma avaliação quantitativa do desempenho do algoritmo. Isso ajuda a identificar como o algoritmo se comporta em diferentes situações.
4. Ordenação de Strings Únicas e Normalização: Para lidar com strings variadas nos dados CSV, o código normaliza as strings, removendo caracteres não alfanuméricos e convertendo tudo para minúsculas. Além disso, cria uma lista de strings únicas e ordenadas. Essa etapa é crucial para a eficácia do Insertion Sort em dados de texto.
5. Implementação do Insertion Sort para Strings: O Insertion Sort é tradicionalmente usado para ordenar números inteiros, mas este código adapta o algoritmo para trabalhar com strings. Ele mapeia cada string normalizada e única para um índice em um array de contagem.
6. Contagem e Ordenação dos Dados: O algoritmo conta as ocorrências de cada string e usa essas contagens para posicionar cada elemento em sua posição correta em um novo ArrayList, efetivamente ordenando os dados.
7. Tratamento do Cabeçalho e Formatação do CSV: O código ignora a primeira linha (cabeçalho) durante a leitura do arquivo CSV e trata possíveis problemas de formatação.
8. Reversão de Dados para o Pior Caso: Após a ordenação, os dados são invertidos para criar o cenário de pior caso, permitindo avaliar o desempenho do algoritmo com dados em ordem decrescente.
9. Uso de Collections.swap para Troca de Elementos: Embora não seja usado explicitamente no processo de ordenação, Collections.swap é utilizado para inverter os dados no cenário de pior caso, o que simplifica a implementação.

## 3.7 Selection Sort

O Selection Sort é um algoritmo de ordenação simples que divide a lista em duas partes: a parte ordenada à esquerda e a parte não ordenada à direita. Ele encontra o elemento mínimo na parte não ordenada e o move para a posição correta na parte ordenada. Esse processo é repetido até que toda a lista esteja ordenada.

Principais Características:

● Divide a lista em duas partes: ordenada e não ordenada.

● Encontra o elemento mínimo na parte não ordenada e o coloca na posição correta na parte ordenada.

● Repete esse processo para todos os elementos da parte não ordenada.

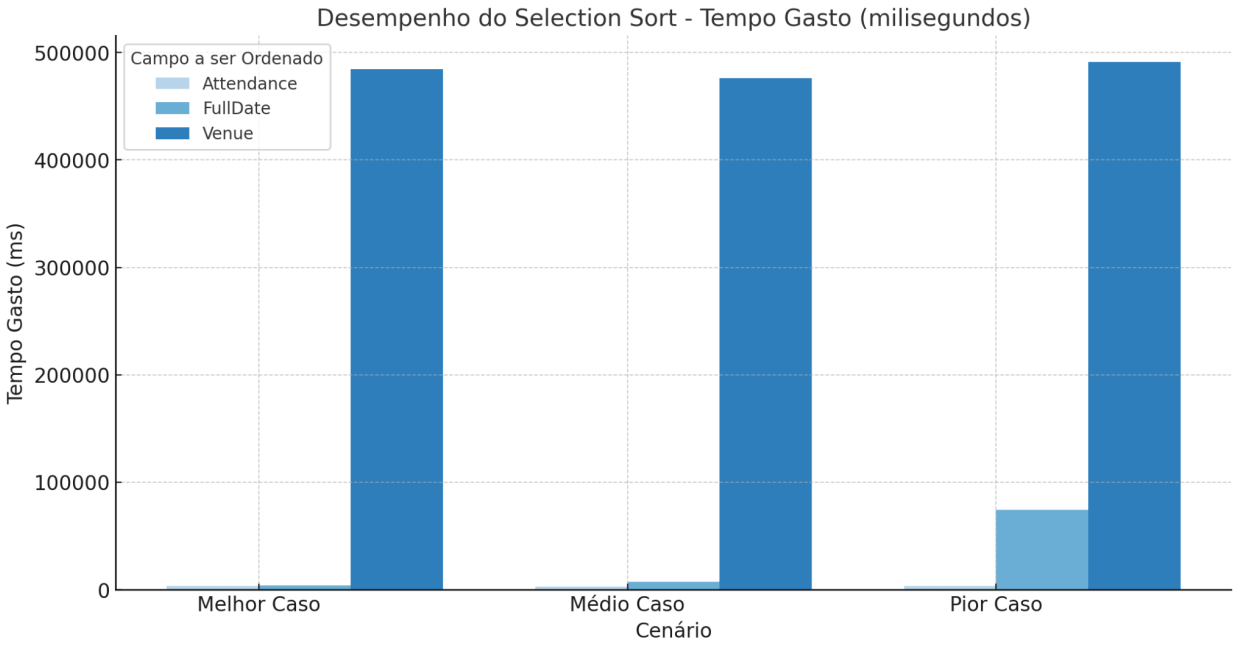
● Tem uma complexidade de tempo no pior caso de O(n^2), onde 'n' é o número de elementos na lista.

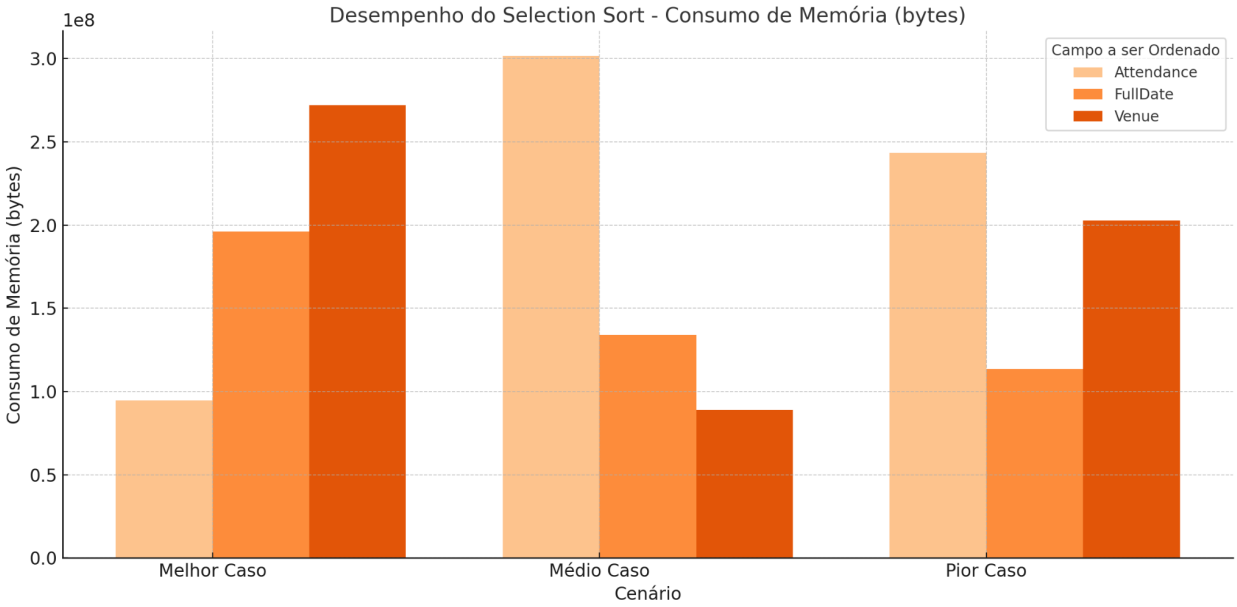
● Não é eficiente para grandes conjuntos de dados devido à sua baixa eficiência.

● É um algoritmo in-place, o que significa que não requer espaço adicional para ordenar a lista.

Sua representação gráfica é sem muitos segredos, ele segue um ‘padrão’ crescente de acordo com o tamanho do array que deve ser ordenado. Segue abaixo a tabela com mais detalhes:

| Método de Ordenação SelectionSort | | | | |
|---|---|---|---|---|
| Campo a ser Ordenado | Cenário | Tamanho da Amostra | Tempo Gasto (milisegundos) | Consumo de Memória (bytes) |
| Venue | Melhor Caso | 25725 | 484361 | 272163720 |
| Venue | Médio Caso | 25725 | 475968 | 88894240 |
| Venue | Pior Caso | 25725 | 491144 | 202768664 |
| Attendance | Melhor Caso | 25725 | 3384 | 94739144 |
| Attendance | Médio Caso | 25725 | 2687 | 301701056 |
| Attendance | Pior Caso | 25725 | 3233 | 243491096 |
| FullDate | Melhor Caso | 25725 | 3889 | 196171856 |
| FullDate | Médio Caso | 25725 | 7166 | 134087760 |
| FullDate | Pior Caso | 25725 | 74480 | 113566752 |




Estratégias Abordadas:

1. Uso de ArrayList para Armazenamento de Dados: O código utiliza ArrayList<String[]> para armazenar os dados dos arquivos CSV. Isso permite uma manipulação eficiente de dados com redimensionamento dinâmico, útil quando se lida com arquivos de tamanho desconhecido.

2. Criação de Três Cenários de Ordenação: O código prepara três cenários de ordenação diferentes (melhor, médio e pior caso) para avaliar o desempenho do algoritmo Selection Sort sob várias condições de entrada.
3. Medição de Desempenho: Durante o processo de ordenação, o código mede o tempo de execução e o consumo de memória, fornecendo uma avaliação quantitativa do desempenho do algoritmo. Isso ajuda a entender como o algoritmo se comporta em diferentes situações.
4. Ordenação com Base em Datas: O algoritmo de Selection Sort é usado para ordenar os dados com base em datas presentes na coluna "full_date" dos arquivos CSV.
5. Normalização e Comparação de Datas: As datas no formato "dd/MM/yyyy" são normalizadas e comparadas para determinar a ordem correta. O código lida com conversões e formatação para garantir que as datas sejam comparadas de maneira apropriada.
6. Tratamento do Cabeçalho e Formatação do CSV: O código ignora a primeira linha (cabeçalho) durante a leitura do arquivo CSV e trata potenciais problemas de formatação.
7. Criação de Casos de Ordenação: São criados casos de ordenação melhor (ordenando os dados), médio (cópia dos dados originais) e pior (ordenando e invertendo a ordem dos dados) para testar diferentes cenários de entrada.
8. Uso de Collections.swap para Troca de Elementos: Embora não seja usado explicitamente no processo de ordenação, Collections.swap é utilizado para inverter os dados no cenário de pior caso, simplificando a implementação.

# 4. Conclusão

Neste relatório, realizamos uma análise comparativa de diferentes algoritmos de ordenação, concentrando nossos esforços na remodelação do projeto da disciplina de Estrutura de Dados. Em vez de criar novos algoritmos, exploramos estratégias para melhorar a eficiência das ordenações ao incorporar estruturas de dados como Listas, ArrayLists, Pilhas e Collections.

Os principais objetivos deste trabalho incluíram a implementação e comparação do desempenho de algoritmos de ordenação em diversos cenários de teste, análises estatísticas do desempenho, investigação dos impactos dos diferentes cenários de teste e fornecimento de informações relevantes sobre a eficiência e aplicabilidade de cada algoritmo.

Através da análise comparativa dos algoritmos Heap Sort, Merge Sort, Quick Sort, Quick Sort com mediana de três, Counting Sort, Insertion Sort e Selection Sort, pudemos identificar suas principais características e desempenho em diferentes cenários. Cada algoritmo apresentou vantagens e desvantagens, sendo mais adequado para diferentes tipos de dados e situações.

Por exemplo, o Heap Sort demonstrou ser eficiente em ordenar grandes conjuntos de dados, enquanto o Merge Sort manteve um desempenho consistente em vários cenários. O Quick Sort, com ou sem a estratégia de mediana de três, mostrou-se eficaz para a maioria dos casos, exceto quando lidava com conjuntos de dados quase ordenados. O Counting Sort revelou-se eficiente para números inteiros em uma faixa específica, mas consome mais memória. O Insertion Sort apresentou bom desempenho em casos de conjuntos de dados pequenos ou quase ordenados. Por fim, o Selection Sort não foi eficiente para grandes conjuntos de dados devido à sua baixa eficiência.

Essa análise comparativa proporcionou uma compreensão mais abrangente do desempenho dos algoritmos de ordenação em diferentes cenários, permitindo uma

escolha mais informada e eficiente das estratégias de ordenação no contexto deste projeto remodelado.

Além disso, foram implementadas estratégias específicas para lidar com diferentes tipos de dados, como normalização e comparação de strings e ordenação de datas, tornando os algoritmos mais versáteis e adaptáveis a diferentes tipos de dados.

Em resumo, este trabalho não apenas explorou a eficiência dos algoritmos de ordenação, mas também destacou a importância da escolha de estruturas de dados apropriadas e estratégias específicas para lidar com diferentes tipos de dados, proporcionando uma base sólida para o desenvolvimento de aplicações que envolvam operações de ordenação.