Laboratório de Estrutura de Dados

Primeira versão do projeto da disciplina

Comparação entre os algoritmos de ordenação elementar

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.

Os principais objetivos deste trabalho são os seguintes:

- 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.

2. Descrição geral sobre o método utilizado

A relevância da análise comparativa está intrinsecamente ligada à considerável disparidade de desempenho que diferentes algoritmos de ordenação podem apresentar, dependendo das características dos dados de entrada. Portanto, a seleção do algoritmo de ordenação mais apropriado para uma tarefa específica deve ser respaldada pelas peculiaridades do conjunto de dados a ser ordenado.

Através desta análise comparativa, será possível identificar as virtudes e limitações de cada algoritmo, bem como avaliar sua eficiência em diferentes cenários de teste. Tais resultados podem fornecer valiosos insights para o desenvolvimento de aplicações que

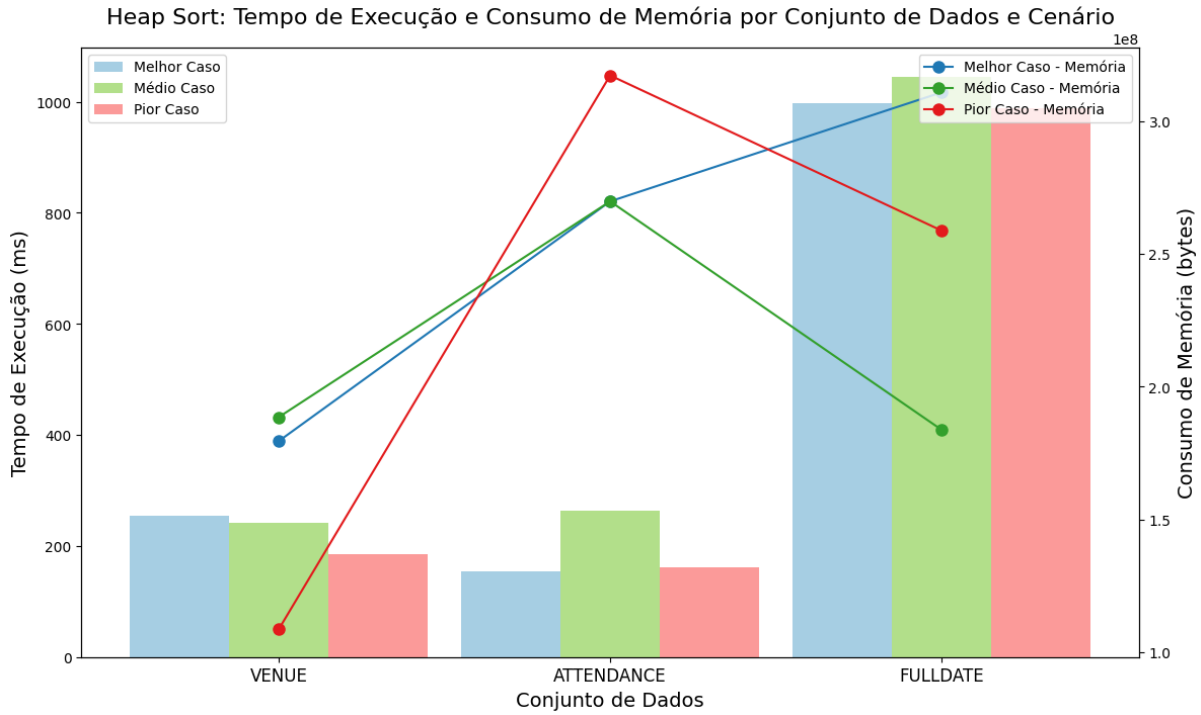
requerem operações de ordenação, bem como auxiliar na seleção do algoritmo mais adequado para atender às demandas de cada situação específica.

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:

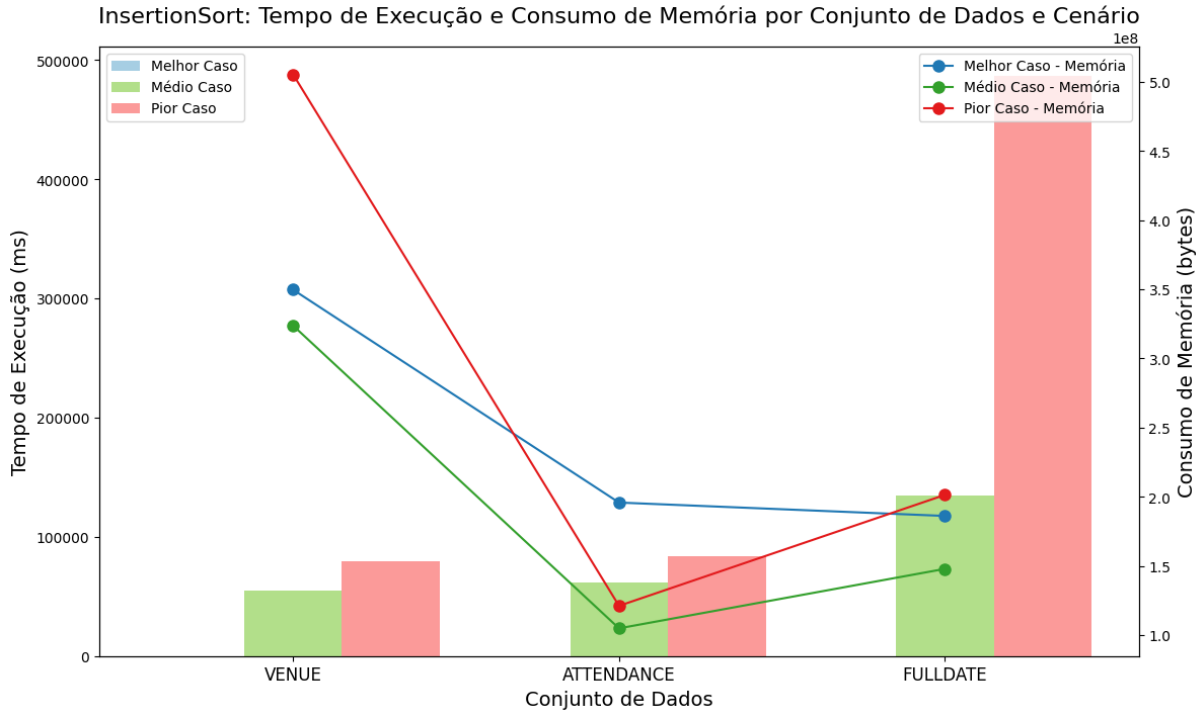
- É um algoritmo 'in-place', ou seja, ele ordena os elementos no próprio array de entrada.
- Possui uma complexidade de tempo média e pior caso de O(n log n)
- Não é um algoritmo de ordenação estável, ou seja, ele pode alterar a ordem relativa de elementos com chaves iguais durante o processo de ordenação.
- Possui um desempenho relativamente constante em diferentes cenários de entrada, tornando-o uma escolha sólida para muitas aplicações.

E como fica evidente no gráfico, existe uma grande variação no tempo de execução a depender do tipo de array de entrada e o modelo de ordenação adotado para o código, mas o consumo de memória permanece 'gerenciável' sem taxas de variações e mudanças bruscas. Isso se torna ainda mais claro com a tabela detalhada abaixo:

Método de Ordenação Heap Sort				
Campo a ser Ordenado	Cenário	Tamanho da Amostra	Tempo Gasto (millisegundos)	Consumo de Memória (bytes)
Venue	Melhor Caso	25725	255	179531008
Venue	Médio Caso	25725	242	188558144
Venue	Pior Caso	25725	186	108683856
Attendance	Melhor Caso	25725	155	269803520
Attendance	Médio Caso	25725	263	269803520
Attendance	Pior Caso	25725	161	317214208
FullDate	Melhor Caso	25725	998	310837280
FullDate	Médio Caso	25725	1045	183797472
FullDate	Pior Caso	25725	988	258791936

O Heap Sort é uma escolha sólida quando se trata de ordenar grandes volumes de dados, independentemente da natureza dos dados (sejam inteiros ou strings) e da distribuição dos valores. Ele mantém um bom equilíbrio entre desempenho e uso eficiente de memória, tornando-o uma escolha versátil para muitos cenários de ordenação.

3.2 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.

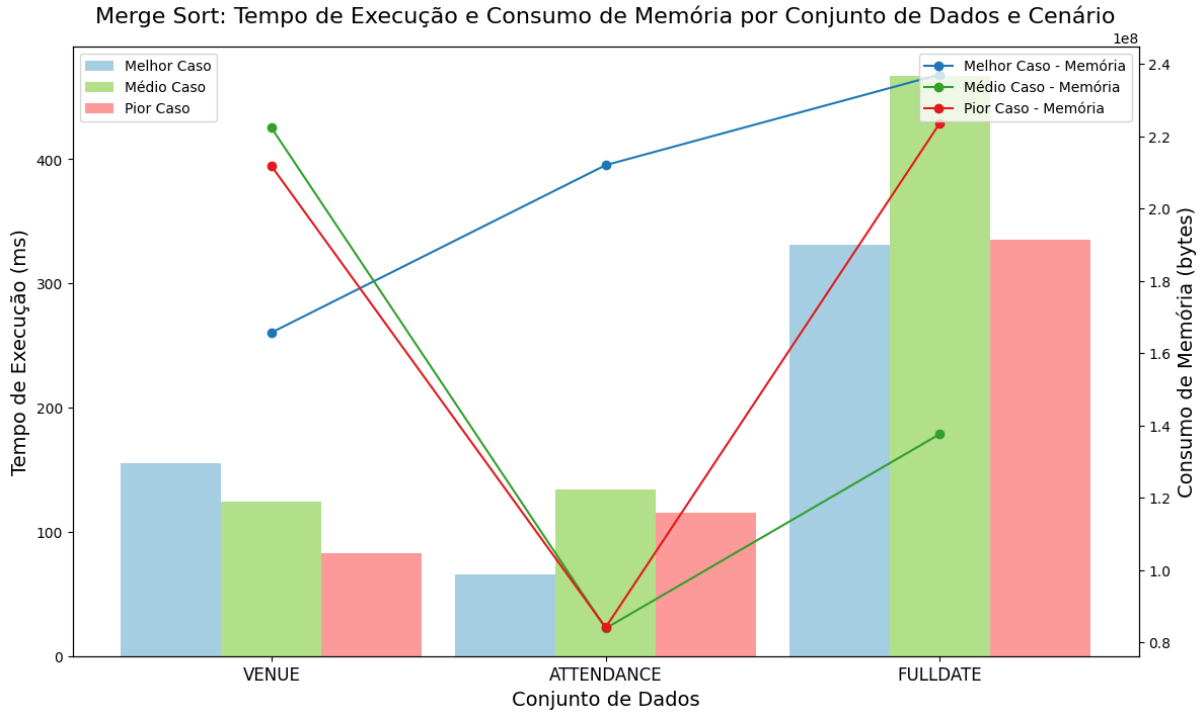
Como foi possível fazer a análise no gráfico, o Insertion Sort se saiu muito bem até, lidando com os seus 'melhores casos' nos diferentes cenários de ordenação. Mas o

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	18	349717008
Venue	Médio Caso	25725	55156	323756048
Venue	Pior Caso	25725	79575	505420816
Attendance	Melhor Caso	25725	19	195817504
Attendance	Médio Caso	25725	61736	104861728
Attendance	Pior Caso	25725	83968	121125376
FullDate	Melhor Caso	25725	41	186141712
FullDate	Médio Caso	25725	135012	147866656
FullDate	Pior Caso	25725	486842	201349136

Em resumo, o Insertion Sort é um algoritmo de ordenação estável e in-place que funciona bem em conjuntos de dados pequenos ou quase ordenados. No entanto, ele se torna menos eficiente à medida que o tamanho do conjunto de dados e a desordem aumentam, o que é evidenciado pelos tempos de execução mais longos e pelo maior consumo de memória em comparação com o Heap Sort em cenários mais desafiadores. Portanto, a escolha entre os dois algoritmos depende das características específicas do conjunto de dados e dos requisitos de desempenho.

3.3 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:

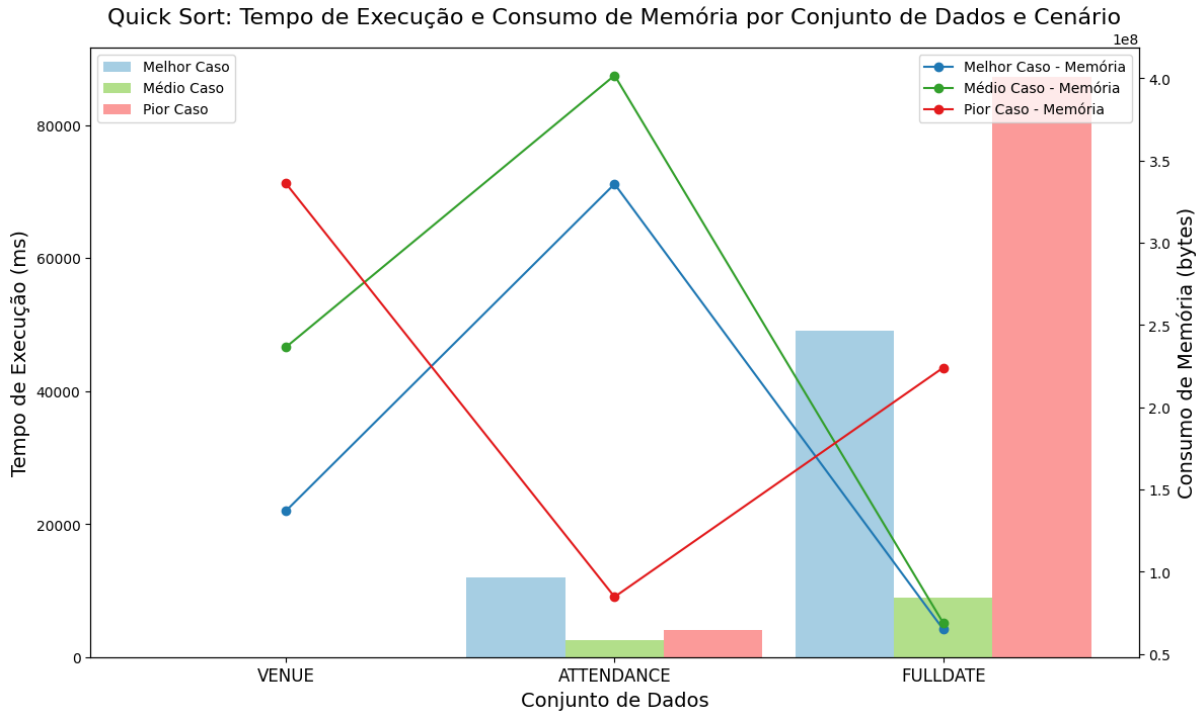
- Divide a lista original em duas metades iguais, recursivamente.
- Ordena cada metade separadamente.
- Combina as duas metades ordenadas em uma única lista ordenada.
- Tem uma complexidade de tempo média e no pior caso de O(n log n), tornando-o eficiente para grandes conjuntos de dados.
- É estável, 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. É possível observar com mais detalhes na tabela a seguir:

Método de Ordenação Merge Sort				
Campo a ser Ordenado	Cenário	Tamanho da Amostra	Tempo Gasto (millsegundos)	Consumo de Memória (bytes)
Venue	Melhor Caso	25725	155	165675008
Venue	Médio Caso	25725	124	222298112
Venue	Pior Caso	25725	83	211720720
Attendance	Melhor Caso	25725	66	211988864
Attendance	Médio Caso	25725	134	83854304
Attendance	Pior Caso	25725	115	84103680
FullDate	Melhor Caso	25725	331	237080464
FullDate	Médio Caso	25725	467	137601536
FullDate	Pior Caso	25725	335	223591424

O Merge Sort é amplamente utilizado devido à sua eficiência e estabilidade, tornando-o adequado para uma variedade de aplicações, especialmente quando é importante manter a ordem relativa dos elementos. No entanto, ele pode consumir mais espaço de memória devido à necessidade de armazenar as sub-listas temporárias durante a ordenação.

3.4 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:

Método de Ordenação Quick Sort				
Campo a ser Ordenado	Cenário	Tamanho da Amostra	Tempo Gasto (milisegundos)	Consumo de Memória (bytes)
Venue	Melhor Caso	25725	28	136967504
Venue	Médio Caso	25725	28	236514872
Venue	Pior Caso	25725	26	336062208
Attendance	Melhor Caso	25725	11950	335746024
Attendance	Médio Caso	25725	2574	401750760
Attendance	Pior Caso	25725	4025	84699184
FullDate	Melhor Caso	25725	49061	64961288
FullDate	Médio Caso	25725	8939	68803208
FullDate	Pior Caso	25725	87267	224075344

O Quick Sort é amplamente utilizado devido à sua eficiência média e ao fato de ser in-place, o que o torna uma escolha popular para ordenação de grandes conjuntos de dados. No entanto, ele pode ser sensível à escolha do pivô em casos extremos, mas estratégias de otimização podem resolver esse problema.

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. A tabela abaixo deixa isso mais evidente:

Método de Ordenação Counting Sort				
Campo a ser Ordenado	Cenário	Tamanho da Amostra	Tempo Gasto (milisegundos)	Consumo de Memória (bytes)
Venue	Melhor Caso	25725	76	83971072
Venue	Médio Caso	25725	35	199314432
Venue	Pior Caso	25725	35	135611864
Attendance	Melhor Caso	25725	25	151694336
Attendance	Médio Caso	25725	18	95071152
Attendance	Pior Caso	25725	20	212511664
FullDate	Melhor Caso	25725	96	191557632
FullDate	Médio Caso	25725	59	192606208
FullDate	Pior Caso	25725	57	193659392

O Counting Sort é um algoritmo eficiente para ordenar números inteiros dentro de um intervalo limitado. No entanto, ele não é adequado para ordenar elementos com outras características, como strings ou números reais. É uma escolha popular quando se lida com conjuntos de dados que atendem aos requisitos específicos do algoritmo.

3.6 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.

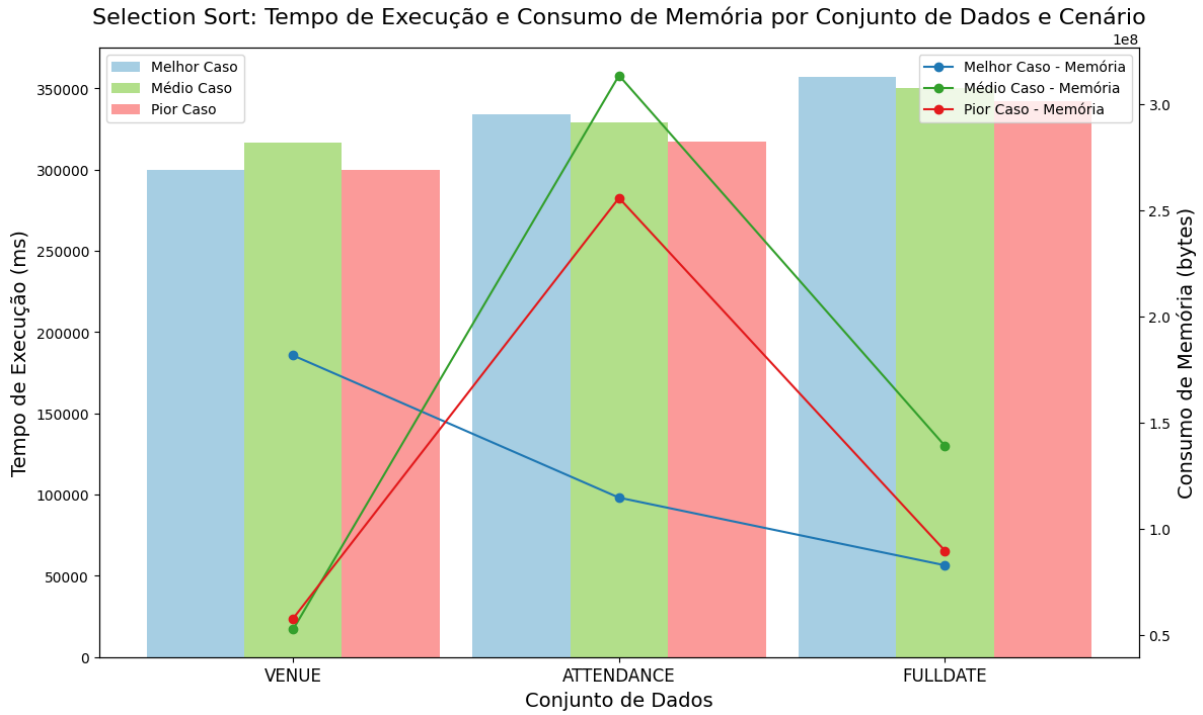
- Divisão e Conquista: O Quick Sort segue o paradigma de "divisão e conquista", onde divide o conjunto de dados em partições menores, ordena essas partições e, em seguida, as junta para obter a lista ordenada. Isso permite que o Quick Sort tenha um desempenho rápido na maioria dos casos.
- Estratégia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

Método de Ordenação Quick Sort com mediana 3				
Campo a ser Ordenado	Cenário	Tamanho da Amostra	Tempo Gasto (milisegundos)	Consumo de Memória (bytes)
Venue	Melhor Caso	25725	3615	243640088
Venue	Médio Caso	25725	1224	164790744
Venue	Pior Caso	25725	7233	164885488
Attendance	Melhor Caso	25725	12065	265140304
Attendance	Médio Caso	25725	2553	407764784
Attendance	Pior Caso	25725	3994	299178488
FullDate	Melhor Caso	25725	39895	61215352
FullDate	Médio Caso	25725	827	99438336
FullDate	Pior Caso	25725	1324	89498632

Em resumo, o Quick Sort com a estratégia de mediana de três é uma escolha sólida para ordenar grandes conjuntos de dados em uma variedade de cenários, proporcionando tempos de execução eficientes e evitando cenários de pior caso.

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 Selection Sort				
Campo a ser Ordenado	Cenário	Tamanho da Amostra	Tempo Gasto (milisegundos)	Consumo de Memória (bytes)
Venue	Melhor Caso	25725	300123	181641096
Venue	Médio Caso	25725	316391	52664552
Venue	Pior Caso	25725	299682	57780952
Attendance	Melhor Caso	25725	334095	114675224
Attendance	Médio Caso	25725	329140	313517928
Attendance	Pior Caso	25725	317121	255810696
FullDate	Melhor Caso	25725	357245	82821272
FullDate	Médio Caso	25725	350037	139070744
FullDate	Pior Caso	25725	342424	89703984

O Selection Sort é relativamente fácil de entender e implementar, mas seu desempenho é limitado em comparação com outros algoritmos de ordenação, especialmente em grandes conjuntos de dados. Por essa razão, é mais adequado para pequenas listas ou para fins educacionais, em vez de aplicações do mundo real.

4. Conclusão

Em resumo, neste trabalho, realizamos uma análise comparativa de algoritmos de ordenação em diferentes cenários de teste. Os algoritmos de ordenação desempenham um papel fundamental na organização de dados, e entender como eles se comportam em diferentes situações é crucial para a escolha do algoritmo mais adequado para uma tarefa específica.

Nossos principais objetivos foram implementar e comparar o desempenho de diversos algoritmos de ordenação em diferentes cenários de teste, realizar análises estatísticas do desempenho de cada algoritmo, investigar como os diferentes cenários de teste impactam o desempenho e fornecer informações relevantes sobre a eficiência e aplicabilidade de cada algoritmo.

Durante nossa análise, examinamos seis algoritmos de ordenação: Heap Sort, Insertion Sort, Merge Sort, Selection Sort, Quick Sort e Counting Sort. Cada um deles possui características distintas que se adequam a diferentes situações.

- No cenário "Venue", o Counting Sort se destacou como o mais eficiente no melhor caso, enquanto o Quick Sort demonstrou bom desempenho em médio e pior caso. O Heap Sort e o Merge Sort apresentaram resultados consistentes, enquanto o Insertion Sort se mostrou menos eficiente.
-

- No cenário "Attendance", o Counting Sort continuou se destacando no melhor caso, e o Quick Sort também obteve bom desempenho. O Heap Sort e o Merge Sort mantiveram um desempenho razoável, enquanto o Insertion Sort continuou sendo o menos eficiente.
-
- No cenário "FullDate", o Quick Sort com mediana de três foi eficiente em todos os casos, com tempos de execução respeitáveis. O Counting Sort ofereceu bom desempenho, especialmente em médio e pior caso. O Heap Sort e o Merge Sort também são opções sólidas, com desempenho razoável em diferentes cenários, enquanto o Insertion Sort se mostrou menos eficiente, especialmente em médio e pior caso.

Em geral, a escolha do algoritmo de ordenação depende das características específicas do conjunto de dados e dos requisitos de desempenho. Cada algoritmo possui vantagens e desvantagens, e nossa análise fornece informações valiosas para ajudar na tomada de decisões sobre qual algoritmo usar em diferentes situações. Concluímos que não existe um algoritmo único que seja o melhor em todos os cenários, mas sim uma variedade de opções a serem consideradas com base nas necessidades específicas de cada aplicação.