PPGCA - Algoritmos e Programação

PROJETO 1

Algoritmos de Ordenação

Análise de performance de algoritmos de ordenação em diferentes cenários

Este projeto tem como objetivo **comparar o desempenho** de diferentes algoritmos de ordenação em diferentes cenários. Os algoritmos escolhidos para a comparação foram:

- Bubble Sort
- Bubble Sort Otimizado
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

Os cenários, considerando vetores de tamanhos **1.000**, **10.000** e **100.000**, foram:

- Vetor aleatório (Caso médio)
- Vetor ordenado em ordem crescente (Melhor caso)
- Vetor ordenado em ordem decrescente (Pior caso)

Metodologia

Metodologia utilizada para realizar os testes e coletar os resultados

Equipamento Utilizado

O programa foi executado em um computador com as seguintes configurações:

- Macbook Pro 2021 com Apple **M1 Pro**
  - CPU de **8 núcleos** (6 de desempenho e 2 de eficiência)
  - GPU de 14 núcleos.
- **16 GB** de memória RAM
- MacOS - Sonoma 14.6.1
- Python 3.12.5

Metodologia

Massa de Dados

Para a comparação dos algoritmos:

- Foi desenvolvido um programa em Python para gerar vetores de tamanhos **1.000, 10.000 e 100.000**
- Os vetores foram gerados de **forma crescente**, **decrescente** (com valores sequenciais) e **aleatórios**.
- Para cada vetor gerado, **é feita a ordenação** utilizando os algoritmos mencionados anteriormente e medido o tempo de execução para concluir a tarefa.

```
arr_middle = random.sample(range(0, 100_000), n)
arr_best = list(range(0, n))
arr_worst = list(range(n, 0, -1))
```

Algoritmos

Para garantir a precisão do teste algumas medidas foram tomadas:

- Os algoritmos utilizados para a ordenação dos vetores foram extraídos do repositório TheAlgorithms e da página Programiz.
- O tempo de execução foi medido utilizando a biblioteca `time` do Python em milissegundos.
- Os algoritmo foram executados **5 vezes para cada vetor.** O tempo de execução considerado foi a média dos 5 testes.
- O vetor foi **copiado antes de ser ordenado** para garantir que o vetor original não fosse alterado. No código foram inseridos verificadores para garantir que os vetores não foram modificados
- O tempo de execução foi calculado usando o tempo no início e no final da execução do algoritmo. Outras operações, **não foram consideradas no cálculo do tempo** de execução.

https://github.com/z-fab/ppgca/tree/master/programacao-algoritmos/projeto-ordenacao

Resultados

Resultados obtidos através dos testes realizados

Resultados

Descrição dos resultados

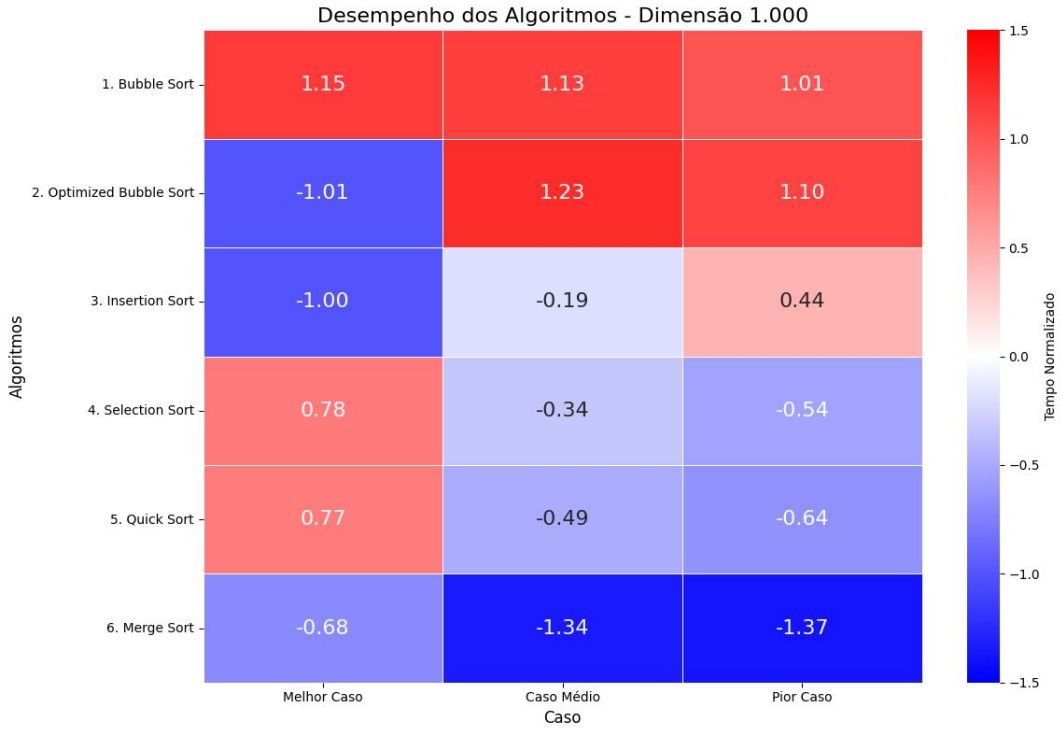
- Em média, no melhor caso, os algoritmos levaram **34 segundos** enquanto no pior caso demoraram, em média, **199 segundos**
- **75% das iterações** (independente do algoritmo ou do tamanho do array) levaram no máximo **3,8 segundos** para ordenar
- Houve um algoritmo que, no pior caso, demorou ~56 minutos para ordenar.

statistic (str)	Melhor Caso (f64)	Caso Médio (f64)	Pior Caso (f64)
count	90,00	90,00	90,00
null_count	0,00	0,00	0,00
mean	34.178,05	120.400,12	199.403,47
std	135.712,91	315.340,94	547.329,57
min	0,04	2,23	2,05
25%	2,06	25,82	24,31
50%	13,58	1.044,86	1.113,77
75%	1.118,94	3.006,05	3.863,31
max	1.111.469,95	1.399.610,91	3.397.482,11

Resultados

Desempenho por Dimensão

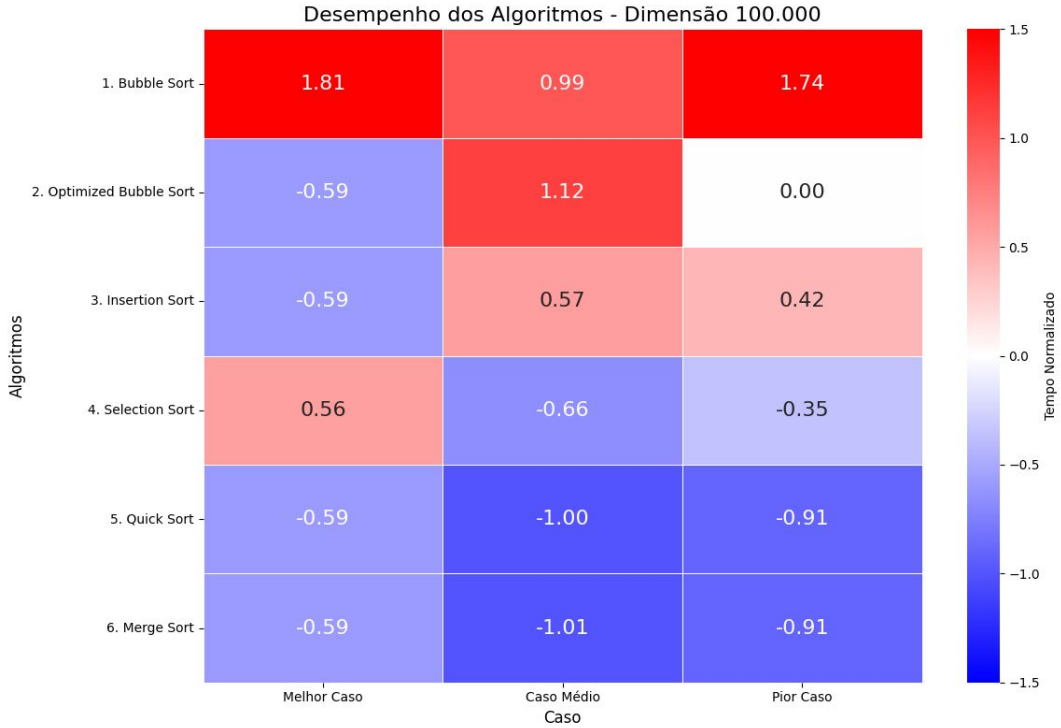
- **Bubble Sort** obteve um resultado semelhante nos três casos, o que é esperado já que a complexidade é sempre **O(n²)**
- **Optimized Bubble Sort** teve um desempenho excelente no Melhor Caso e péssimo nos outros. Isso é resultado da complexidade **O(n)** no melhor caso e **O(n²)** no restante
- O **Insertion Sort** teve um comportamento semelhante ao Bolha otimizado



Resultados

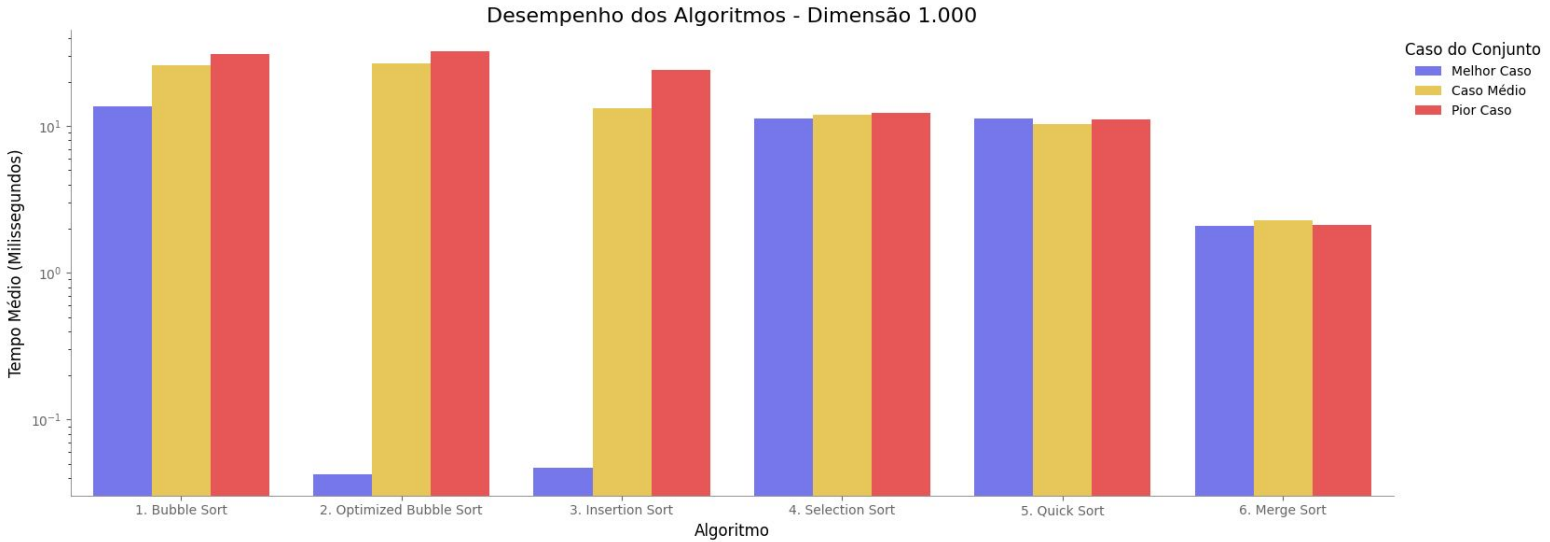
Desempenho por Dimensão

- Conforme a massa de dados aumentou o **Quick Sort** e o **Merge Sort** se demonstraram bem superiores aos demais.
- O uso de um pivô aleatório no Quick Sort permitiu manter o desempenho do algoritmo como **O(n log n)** nos três cenários.
- No geral Merge Sort e Quick Sort tiveram resultados próximos, porém o mais rápido em todos os casos foi o **Merge**



Resultados

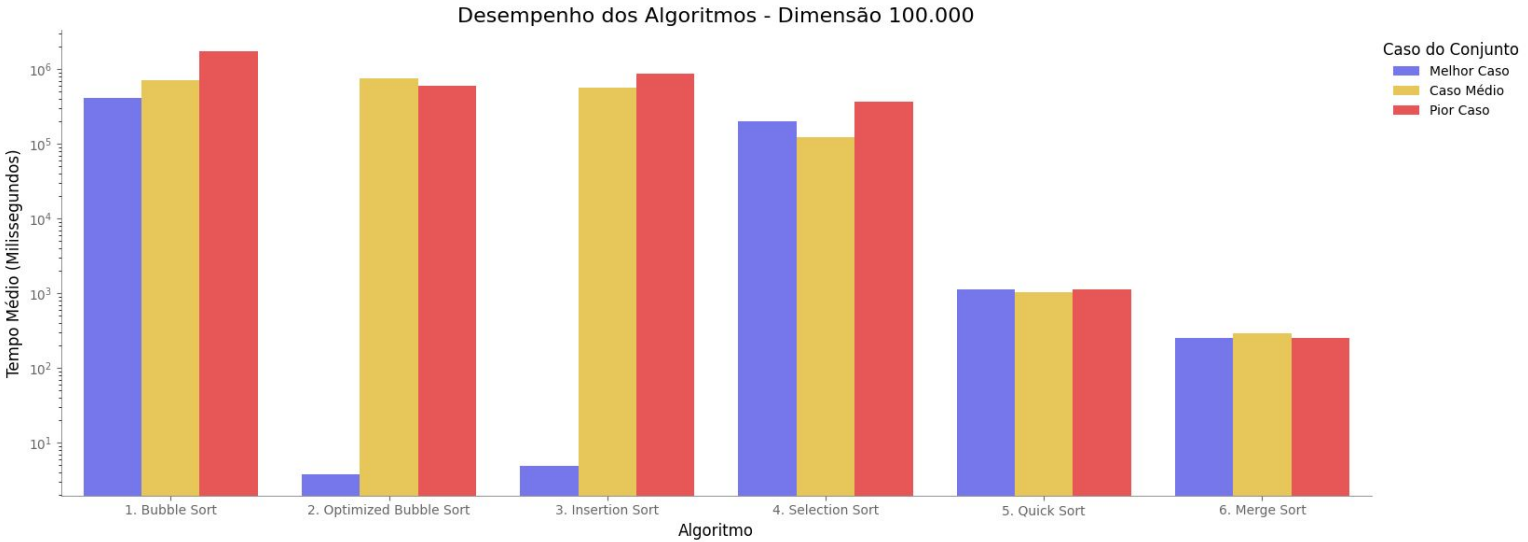
Desempenho por Algoritmos



- Ao analisar o gráfico de tempos (em escala logarítmica) percebemos o comportamento de **O(n²)** vs **O(n)** no melhor caso do **Bolha otimizada** e **Insertion Sort**
- A diferença entre os tempos do **Bolha** pode estar relacionado com o número de trocas necessárias (Melhor caso não há trocas e no pior caso sempre há trocas)

Resultados

Desempenho por Algoritmos

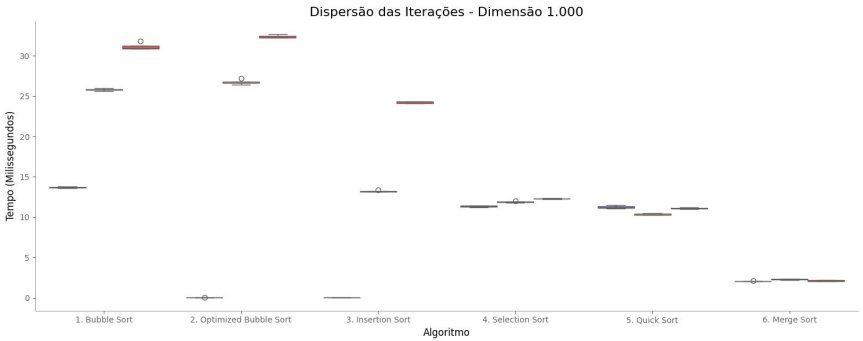


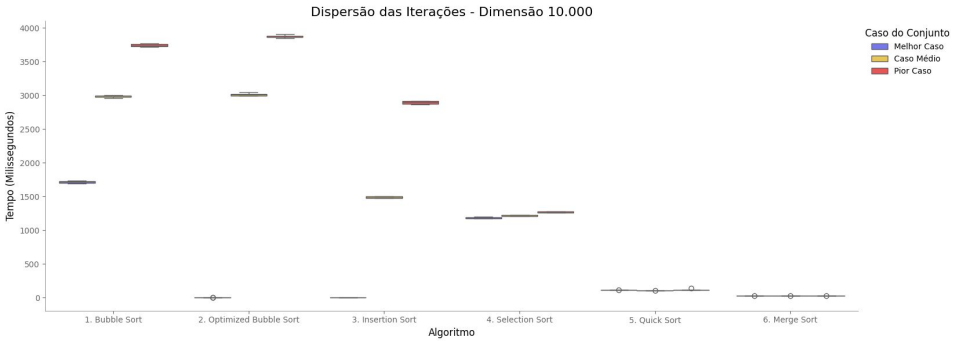
- Conforme aumenta-se a massa de dados percebemos a superioridade do Merge Sort. Insertion e Bolha Otimizada só se mostram viáveis em **casos de vetores ordenados** (o que deixa de fazer sentido o uso desses algoritmos)

Resultados

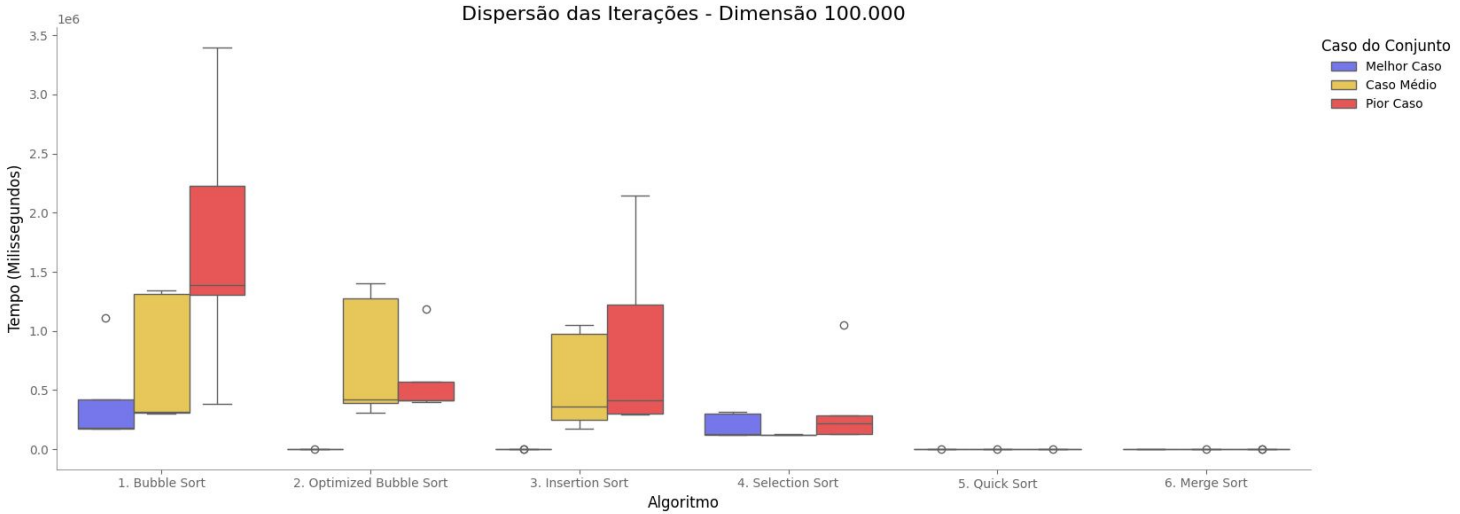
Iterações

- Nos conjuntos de 1.000 elementos e 10.000 elementos a variação de tempo das execuções foram pouco significativos, o que nos faz poder considerá-los como praticamente iguais





Iterações



- Já nas iterações no array de 100.000 percebemos uma variação muito maior dos valores. Essa variação, principalmente em algoritmos **O(n²)** pode se dar por características da linguagem usada: Python é interpretada e por consequência acaba tendo um gerenciamento de memória e cache diferente de linguagens compiladas.
- Algoritmos **O(n²)** levam mais tempo para serem concluídos e portanto estão mais expostos a esses detalhes

Conclusão

Conclusões obtidas através do trabalho

- A **análise assintótica** se prova uma ferramenta **poderosa para prever o comportamento dos algoritmos**, especialmente para grandes conjuntos de dados.

- **Merge Sort e Quick Sort** demonstraram consistentemente o **melhor desempenho** para grandes conjuntos de dados (100.000 elementos), confirmando sua superioridade teórica e prática sobre algoritmos $O(n^2)$

- Algoritmos como **Bubble Sort Otimizado e Insertion Sort** mostraram grande diferença de desempenho dependendo da ordenação inicial dos dados.

- A eficiência relativa dos algoritmos muda significativamente com o aumento do tamanho do conjunto de dados, com algoritmos como Selection Sort sendo competitivos para pequenos conjuntos (1.000 elementos) mas ineficientes para **grandes conjuntos**

- As **características da linguagem** utilizada na implementação do algoritmo interfere em seu desempenho teórico

{ VALEU }

Fabricio Zillig

https://github.com/z-fab