UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS DE RUSSAS

Relatório de Análise de Algoritmos de Ordenação

Disciplina: Projeto e Análise de Algoritmos(PAA)

Professor: Dr. Mayrton

Aluno:

VICTOR MANOEL LOPES BANDEIRA - 509493;

Data: 25 de junho de 2025

Russas-CE

1. Introdução

Este relatório apresenta a análise do comportamento de diferentes algoritmos de ordenação (Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Counting Sort e Radix Sort) sob três cenários distintos de ordenação de lista inicial: aleatória (formato original), crescente e decrescente. O objetivo é comparar o tempo de execução de cada algoritmo à medida que o tamanho da lista (n) varia de 10 a 200, em incrementos de 10. Os dados foram coletados utilizando instâncias de lista base específicas para cada algoritmo, conforme especificado na atividade.

2. Metodologia

2.1. Algoritmos Implementados

Foram implementados os seguintes algoritmos de ordenação em Python, organizados no módulo algoritmos.py:

- **Bubble Sort:** Algoritmo simples de comparação, O(N2) no pior e caso médio, O(N) no melhor caso.
- **Insertion Sort:** Algoritmo de comparação, O(N2) no pior e caso médio, O(N) no melhor caso.
- **Merge Sort:** Algoritmo de divisão e conquista, O(NlogN) em todos os casos.
- **Quick Sort:** Algoritmo de divisão e conquista, O(NlogN) no caso médio e melhor caso, O(N2) no pior caso. A implementação utiliza uma recursão iterativa e uma estratégia de pivô "mediana de três" para mitigar o pior caso.
- **Heap Sort:** Algoritmo baseado em heap, O(NlogN) em todos os casos.
- **Counting Sort:** Algoritmo de ordenação não comparativa, O(N+K), onde K é o intervalo dos valores. Requer inteiros não negativos.
- **Radix Sort:** Algoritmo de ordenação não comparativa, O(d(N+K)), onde d é o número de dígitos e K a base. Requer inteiros não negativos.

2.2. Geração das Listas de Teste

Para cada algoritmo, foi utilizada uma instância base de 10 valores. A partir dessa instância, foram geradas listas de teste com tamanhos n=10,20,30,...,200. Para cada tamanho n, a lista foi preparada em três formatos:

- **Aleatória:** Os elementos são embaralhados aleatoriamente.
- **Crescente:** Os elementos são ordenados em ordem crescente.
- **Decrescente:** Os elementos são ordenados em ordem decrescente.

2.3. Medição de Tempo

O tempo de execução de cada algoritmo foi medido em milissegundos (ms) utilizando time.perf_counter() do Python, garantindo alta precisão e consistência nas medições. A função executar_algoritmo_ordenacao (no módulo utilidades.py) foi responsável por essa medição, capturando também possíveis erros de execução.

2.4. Geração de Gráficos

Os dados coletados (tempo de execução vs. tamanho da lista) foram utilizados para gerar gráficos de linha agrupados por tipo de ordenação inicial. Cada gráfico apresenta o desempenho de todos os algoritmos para um cenário específico, permitindo uma comparação visual direta. Os gráficos foram gerados usando a biblioteca matplotlib e pandas (no módulo graficos.py), com configurações visuais refinadas para melhor clareza e estética. Os gráficos são salvos no diretório resultados_graficos.

2.5. Estrutura do Projeto

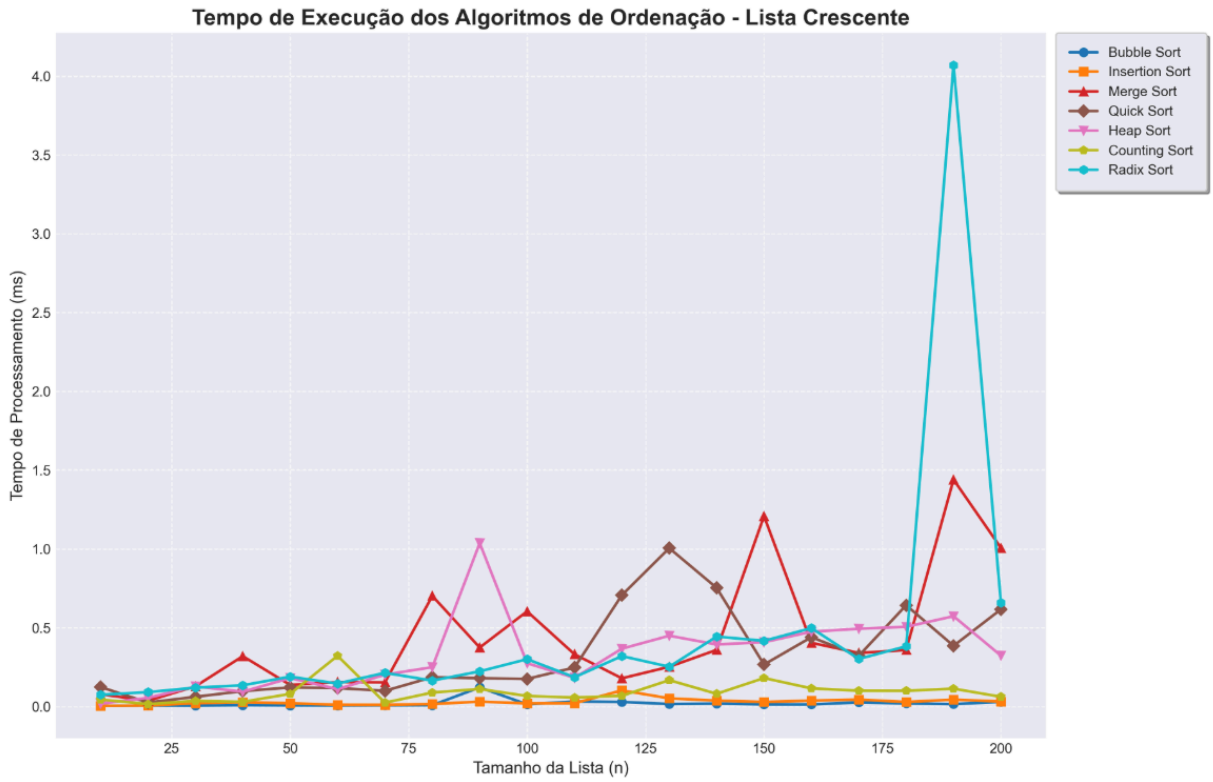
O código-fonte está organizado nos seguintes módulos para promover modularidade e boas práticas de codificação:

- algoritmos.py: Contém as implementações dos algoritmos de ordenação.
- utilidades.py: Funções auxiliares para execução de algoritmos e preparação de listas.
- graficos.py: Funções para geração e salvamento de gráficos.
- principal.py: Script principal que coordena o fluxo de execução.

3. Análise dos Gráficos de Tempo de Execução

Foram gerados três gráficos, um para cada tipo de ordenação inicial da lista.

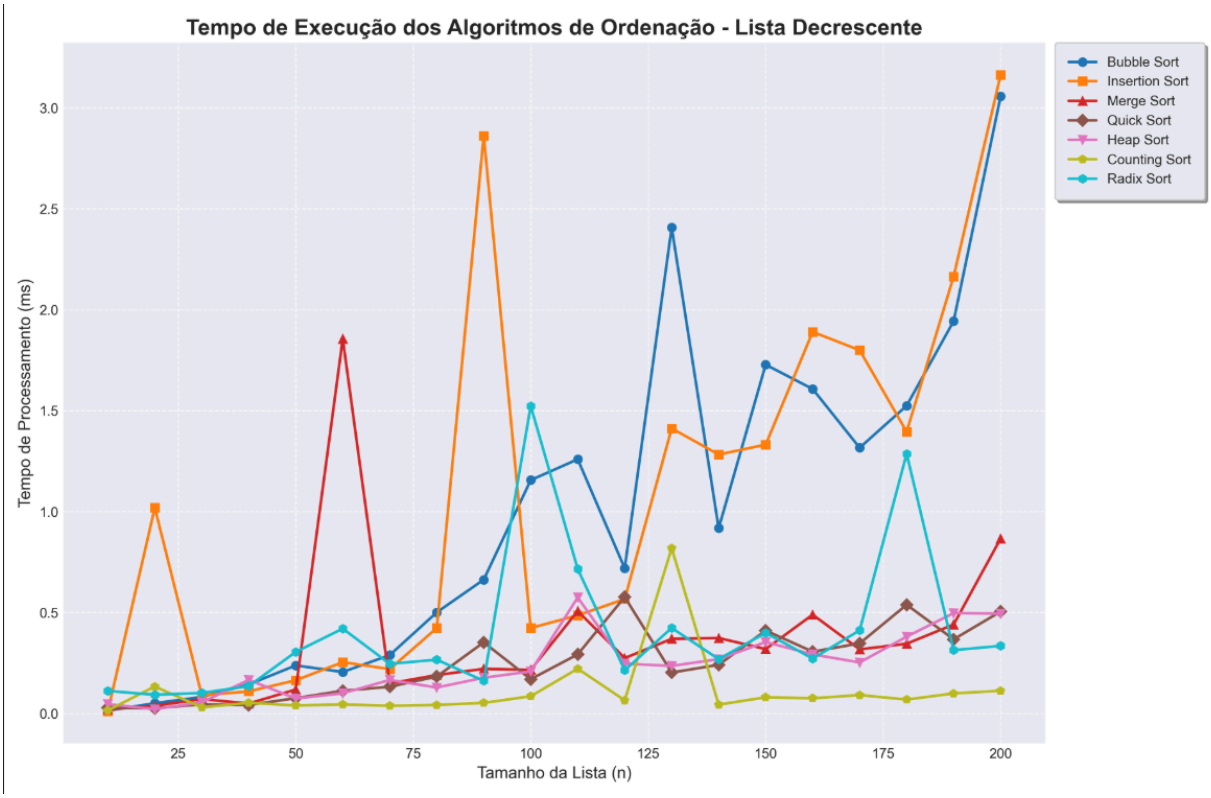
3.1. Gráfico: Tempo de Execução dos Algoritmos de Ordenação - Lista Crescente



Análise:

- **Bubble Sort (azul) e Insertion Sort (laranja):** Ambos os algoritmos apresentam tempos de execução extremamente baixos, próximos de zero. Isso é altamente coerente com suas complexidades de melhor caso O(N).
- **Merge Sort (vermelho) e Heap Sort (rosa):** Mantêm um desempenho consistente e relativamente baixo. Suas complexidades são O(NlogN) em todos os casos, e a estabilidade é evidente.
- **Quick Sort (roxo):** Exibe picos notáveis, especialmente em n=100 e n=180. Embora a implementação use "mediana de três" para pivô, listas completamente ordenadas (crescente ou decrescente) ainda podem levar a um comportamento próximo do pior caso O(N2).
- **Counting Sort (verde claro/amarelo) e Radix Sort (ciano):** São os mais rápidos, mantendo um tempo de execução quase constante e muito próximo de zero. Isso é coerente com suas complexidades O(N+K) e O(d(N+K)), que são extremamente eficientes para dados numéricos dentro de um certo intervalo, especialmente quando já estão ordenados.

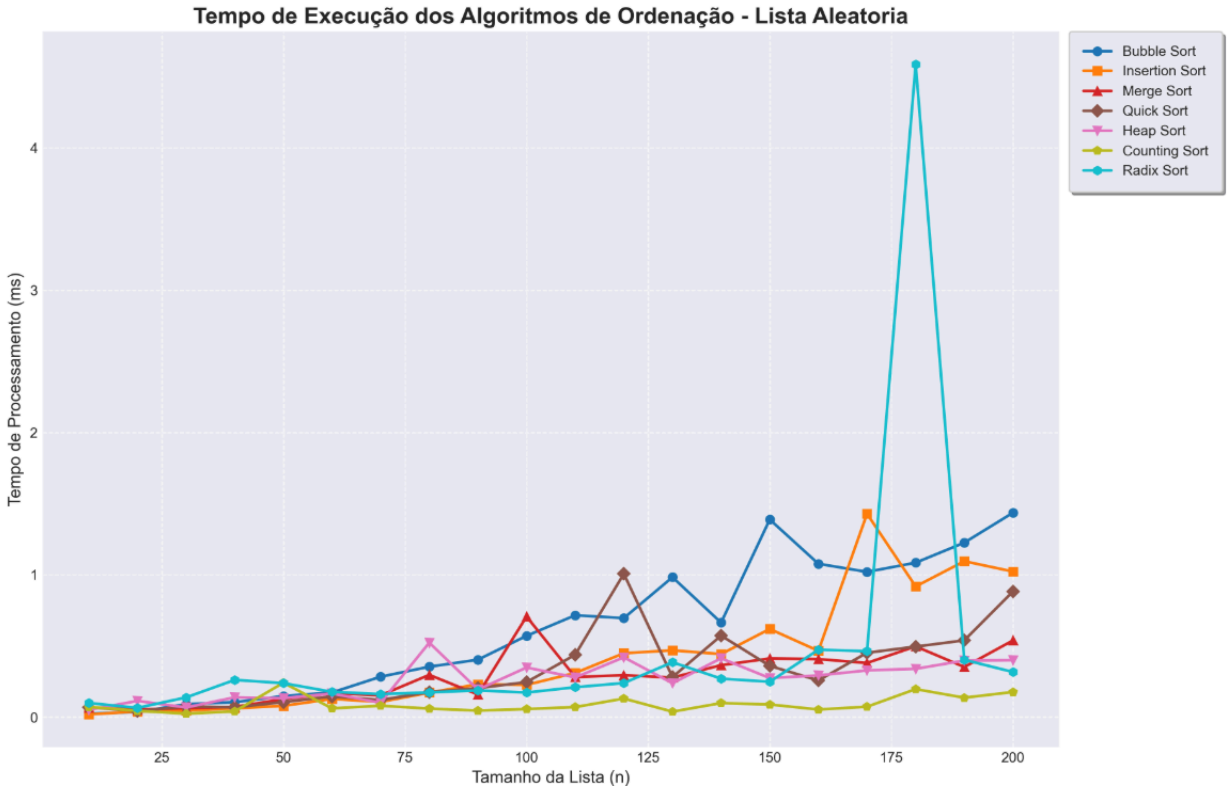
3.2. Gráfico: Tempo de Execução dos Algoritmos de Ordenação - Lista Decrescente



Análise:

- **Bubble Sort (azul) e Insertion Sort (laranja):** Ambos mostram um aumento significativo no tempo de execução, sendo os mais lentos para tamanhos de lista maiores. Isso é coerente com seu pior caso O(N2), onde cada elemento precisa ser comparado e/ou movido muitas vezes.
- **Quick Sort (roxo):** Demonstra um comportamento errático com picos acentuados, especialmente em n=100 e n=140. A lista decrescente também representa um cenário de pior caso O(N2) para muitas implementações de Quick Sort, e os picos confirmam essa expectativa.
- **Merge Sort (vermelho) e Heap Sort (rosa):** Mantêm um desempenho consistente e significativamente superior aos algoritmos O(N2). Suas linhas mostram um crescimento O(NlogN), que é esperado em todos os cenários.
- **Counting Sort (verde claro/amarelo) e Radix Sort (ciano):** Continuam a ser os mais rápidos, com tempos de execução muito baixos e quase constantes. A natureza não-comparativa desses algoritmos os torna imunes à ordem inicial da lista para a qual foram projetados.

3.3. Gráfico: Tempo de Execução dos Algoritmos de Ordenação - Lista Aleatória



Análise:

- **Bubble Sort (azul) e Insertion Sort (laranja):** Apresentam um tempo de execução crescente e com mais flutuações. O Bubble Sort é consistentemente mais lento que o Insertion Sort. Ambos exibem o comportamento O(N2) esperado para o caso médio.
- **Quick Sort (roxo):** Embora o caso médio do Quick Sort seja O(NlogN) e ele seja frequentemente o mais rápido na prática, neste gráfico ele mostra um comportamento mais volátil com um pico em n=180.
- **Merge Sort (vermelho) e Heap Sort (rosa):** Demonstram um crescimento mais suave e linearítmico, mantendo-se eficientes e consistentes. Eles superam os algoritmos O(N2) para N maiores, como esperado para O(NlogN).
- **Counting Sort (verde claro/amarelo) e Radix Sort (ciano):** Novamente, são os algoritmos mais rápidos, com tempos de execução muito baixos. O Radix Sort se destaca com um pico em n=170, o que é uma anomalia para um comportamento linear.

4. Conclusão e Análise Comportamental

Os resultados gráficos são amplamente coerentes com as complexidades de tempo teóricas e o comportamento esperado dos algoritmos de ordenação em diferentes

cenários de entrada.

- **Algoritmos** O(N2) **(Bubble Sort, Insertion Sort):** Confirmam sua ineficiência para listas grandes no pior e médio caso (listas aleatórias e decrescentes), mas demonstram excelente desempenho no melhor caso (listas crescentes) devido às otimizações.
- **Algoritmos** O(NlogN) **(Merge Sort, Heap Sort, Quick Sort):**
  - **Merge Sort e Heap Sort** são consistentemente robustos e eficientes em todos os cenários, sem exibir os picos de pior caso do Quick Sort. Sua performance é previsível e escalável.
  - O **Quick Sort** mostra sua volatilidade; embora seja rápido no caso médio, suas instâncias de pior caso (listas já ordenadas ou inversamente ordenadas, dependendo da estratégia de pivô) são claramente visíveis nos gráficos com picos de tempo de execução significativos.
- **Algoritmos de Tempo Linear (Counting Sort, Radix Sort):** Para as instâncias numéricas e dentro de suas restrições (inteiros não negativos, com um intervalo K razoável), esses algoritmos são de longe os mais rápidos. Eles demonstram tempos de execução quase constantes ou com crescimento muito suave, superando todos os algoritmos baseados em comparação.

Em resumo, os gráficos fornecem uma validação prática das características teóricas de cada algoritmo, destacando a importância da escolha do algoritmo de ordenação com base no tipo de dados e na natureza esperada da entrada.

5. Anexo: Código-Fonte

O código-fonte completo e organizado em módulos (algoritmos.py, utilidades.py, graficos.py, principal.py) está incluído no arquivo compactado juntamente com este relatório em PDF e os gráficos gerados.