UNIVERSIDADE FEDERAL DO CEARÁ

Universidade Federal do Ceará — Campus Quixadá
Disciplina: QXD0010 – Estruturas de Dados
Professor: Prof. Atílio G. Luiz

Relatório Técnico — Comparação Empírica de Algoritmos de Ordenação

Análise experimental do desempenho de Bubble, Insertion, Selection, Merge, Quick e Heap Sort

Aluno: Gleydson Rodrigues Lins — Matrícula 553596
Data: Novembro de 2025

SUMÁRIO

1. Descrição dos Algoritmos

🧮 Bubble Sort

Ideia básica: segue o sistema de ordenação por comparação, ele percorre repetidamente uma lista de vetores , comparando pares de elementos proximos e trocando a posição se eles nao estiverem na ordem correta . Assim, os maiores elementos sobem como “bolhas” para o final da lista a cada passagem.

Complexidade:

- **Melhor caso:** O(n) (quando a lista já está ordenada)
- **Pior caso e caso médio:** $O(n^2)$

Fácil de implementar: É um dos algoritmos mais simples de compreender e programar.

- **Pouco eficiente** para grandes conjuntos de dados.

🧮 Insertion Sort

Ideia básica: Segue o sistema de segue o sistema de ordenação por comparação, percorrendo uma lista da esquerda para a direita, elemento por elemento e inserindo ele na posição correta em uma parte já ordenada da lista, movendo os elementos maiores para direita.

Complexidade:

- **Melhor caso**: O(n) (quando a lista já está ordenada).
- **Pior caso e caso médio**: $O(n^2)$.

Estável: Sim.

Fácil de implementar: Muito simples de compreender e programar, ideal para listas pequenas ou quase ordenadas.

- **Pouco eficiente para grandes conjuntos de dados.**

🧮Selection Sort

Ideia básica: Segue o principio de ordenação por comparação. Onde percorre a lista procurando o menor elemento e a cada iteração, e o coloca na posição correta, e fazendo um swap (trocando este elemento com o da posição atual).

Complexidade:

- **Melhor caso**: $O(n^2)$ (lista já ordenada).
- **Pior caso e caso médio**: $O(n^2)$ (sempre realiza o mesmo número de comparações).

Estável: Não

Fácil de implementar: É simples de entender e programar.

- **Pouco eficiente para grandes conjuntos de dados.**

🧮 Merge Sort

Ideia básica: Segue o principio de ordenação por comparação e divisão e conquista. Ele divide recursivamente a lista ao meio até que cada sublista tenha um único elemento, e as combina de forma ordenada.

Complexidade:

- **Melhor caso**: O(n log n) (mesmo quando a lista já está ordenada).
- **Pior caso e caso médio**: O(n log n) (sempre realiza a mesma quantidade de comparações).

Estável: Sim.

Fácil de implementar: A implementação é mais complexa devido ao uso recursivo.

Muito eficiente para grandes conjuntos de dados — o Merge Sort tem uma performance estável e ótima para listas grandes, especialmente quando a estabilidade é necessária.

🧮 Quick Sort

Ideia básica: Segue o principio de ordenação por comparação e divisão e conquista. Ele escolhe um "pivô" e particiona a lista em torno dele, colocando os elementos menores que o pivô à esquerda e os maiores à direita. Em seguida, aplica recursivamente o mesmo processo nas sublistas à esquerda e à direita do pivô.

Complexidade:

- **Melhor caso**: O(n log n) (quando o pivô divide a lista de forma equilibrada).
- **Pior caso**: $O(n^2)$

Estável: Não.

Fácil de implementar: A implementação é simples, mas pode ser difícil de otimizar corretamente, especialmente ao escolher o pivô.

Muito eficiente para grandes conjuntos de dados — geralmente é mais rápido que o Merge Sort, especialmente em listas de tamanho médio, devido à sua natureza in-place. No entanto, no pior caso, pode ser tão lento quanto o Insertion e Selection Sort. Ideal para listas grandes.

🧮 Heap Sort

Ideia básica: Segue o principio de ordenação por comparação. Ele utiliza uma estrutura de dados chamada heap (geralmente um heap máximo ou mínimo) para ordenar a lista. Ele transforma a lista em um heap, onde o maior (ou menor) elemento está na raiz. Depois, retira o elemento da raiz e reorganiza o heap, repetindo o processo até que todos os elementos estejam ordenados.

Complexidade:

- **Melhor caso**: O(n log n)
- **Pior caso e caso médio**: O(n log n) (sempre realiza a mesma quantidade de comparações).

Estável: Não.

Fácil de implementar: A implementação é mais complexa do que o Insertion ou Selection Sort, devido à necessidade de manter a estrutura de heap durante o processo.

Muito eficiente para grandes conjuntos de dados — o Heap Sort tem uma performance de tempo garantida de **O(n log n)**, o que o torna bastante eficiente para grandes listas. No entanto, devido à falta de estabilidade e ao custo de manutenção do heap, pode ser menos eficiente em termos de constante de tempo do que outros algoritmos como o Quick Sort.

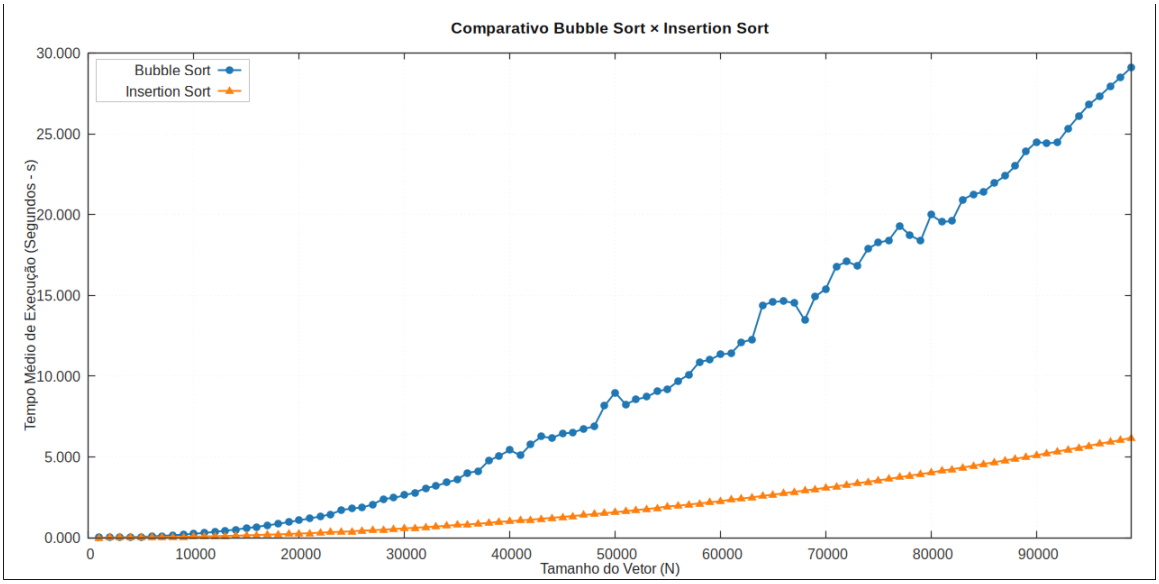
Algoritmo	Complexidade de Pior Caso	Estável	Tipo	Estrutura	Ideia Principal
Bubble Sort	$O(n^2)$	Sim	Iterativo	In-place	Troca repetida de vizinhos fora de ordem
Insertion Sort	$O(n^2)$	Sim	Iterativo	In-place	Insere elementos em uma sublista ordenada
Selection Sort	$O(n^2)$	Não	Iterativo	In-place	Seleciona o menor elemento e o coloca na posição correta
Merge Sort	O(n log n)	Sim	Recursivo	Não In-place	Divide e mescla sublistas ordenadas

Quick Sort	O(n log n)	Não	Recursivo	In-place	Particiona o vetor em torno de um pivô
Heap Sort	O(n log n)	Não	Iterativo	In-place	Constrói um heap e extrai o maior elemento sucessivamente

Os algoritmos Bubble, Insertion e Selection possuem comportamento quadrático, enquanto Merge, Quick e Heap têm complexidade O(n log n), o que garante desempenho superior em grandes listas.

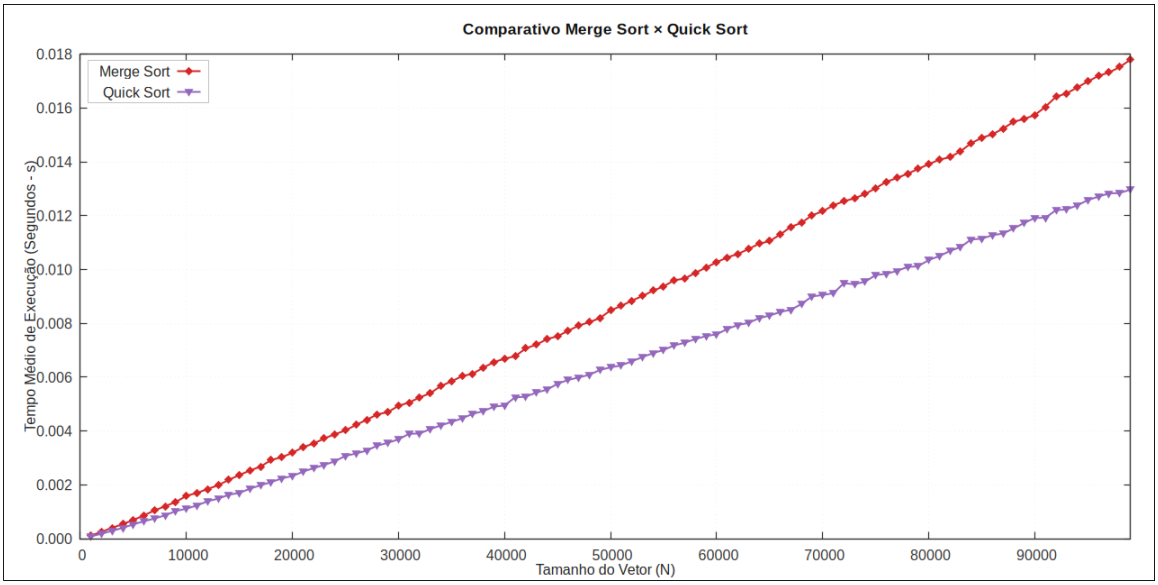
2. Gráficos e Interpretação dos Resultados

- Comparativo Bubble Sort × Insertion Sort



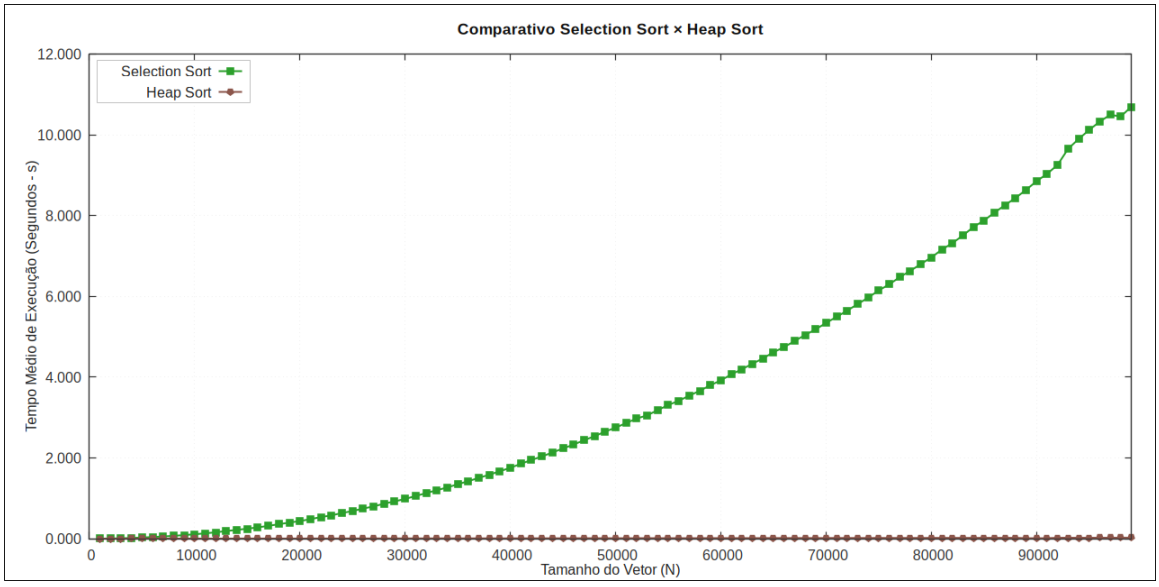
Ambos apresentam crescimento quadrático, mas o Insertion Sort tende a ter melhor desempenho em listas parcialmente ordenadas.

- Comparativo Merge Sort × Quick Sort.



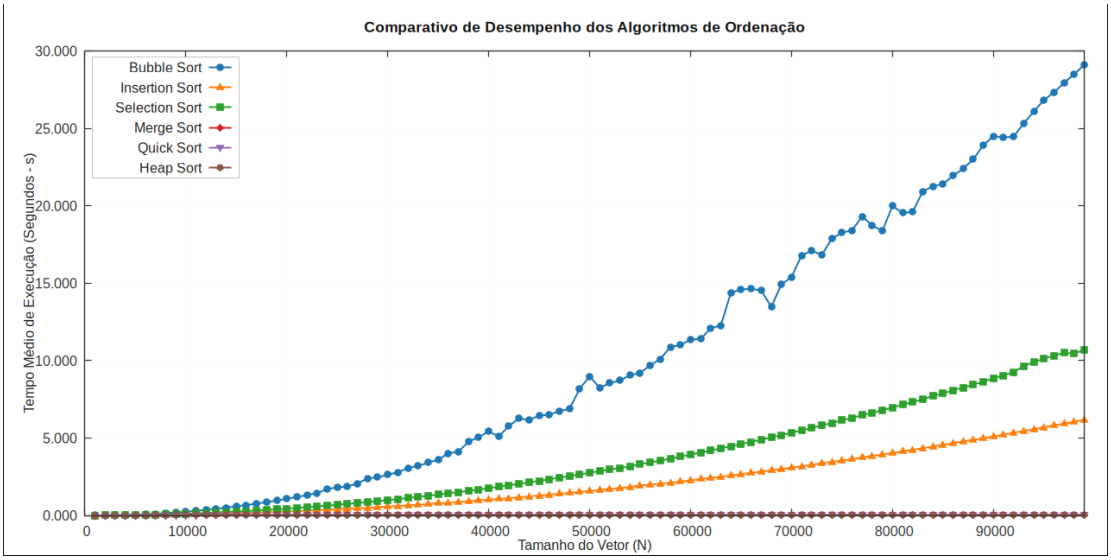
Quick Sort é geralmente mais rápido na prática, mas pode ter pior caso quadrático. Merge Sort mantém estabilidade e previsibilidade de desempenho.

- Comparativo Selection Sort × Heap Sort



Heap Sort supera o Selection Sort em qualquer cenário, mantendo O(n log n) de complexidade.

- Comparativo Geral (todos os algoritmos)



A análise geral evidencia duas categorias distintas: os algoritmos quadráticos escalam rapidamente em tempo, enquanto os logarítmicos mantêm desempenho eficiente.

Os resultados empíricos confirmam integralmente as previsões teóricas:

- Algoritmos **$O(n^2)$** são inviáveis para $N > 10^4$.
- Algoritmos **O(n log n)** escalam com excelente eficiência.
- O **Quick Sort** é o melhor algoritmo prático, e o **Heap Sort** o mais equilibrado entre desempenho e estabilidade.

3. Algoritmo Pesquisado e Implementado

- Algoritmo Pesquisado: Heap Sort

O algoritmo Heap Sort foi o escolhido para estudo mais aprofundado. Ele constrói um heap binário máximo e move o maior elemento para o final do vetor, reconstruindo o heap até que todos os elementos estejam ordenados.

5. Dificuldades Encontradas

- Implementação recursiva de Merge e Quick Sort.
- Geração e leitura de arquivos binários.
- Medição precisa de tempo em microssegundos.
- Automação das execuções e integração com Gnuplot.
- Comparação equilibrada entre algoritmos de complexidades diferentes.
- Encontrar melhor forma de fazer as chamadas de funcao e a medicao do tempo de execucao.
- Pesquisa e criancao dos graficos.
- Estudos dos graficos.
- Analize e adaptacao do codigo disponibilizada pelo professor.

6. Referências Bibliográficas

https://www.ime.usp.br/~pf/algoritmos/aulas/ordena.html

https://www.youtube.com/watch?v=lL0tox33q90

André Backes - Linguagem C - Completa e Descomplicada-GEN LTC_ 2º edição (2022)

https://www.youtube.com/watch?v=xw9DxhxP5S4

https://www.youtube.com/playlist?list=PLqJK4Oyr5WSgsSi26lXEm-SOnZkhkUO7k

SEDGEWICK, R.; WAYNE, K. Algorithms. Addison-Wesley, 2011.

https://www.freecodecamp.org/portuguese/news/algoritmos-de-ordenacao-explicados-com-exemplos-em-python-java-e-c/

CHAIMOWICZ, Luiz. *Projeto e Análise de Algoritmos*

LINTZMAYER, Carlos N. *Análise de Algoritmos e de Estruturas de Dados.*