Universidade do Estado do Rio de Janeiro

Instituto Politécnico do Rio de Janeiro

João Vinicius Vitral **- 202010358111**

Trabalho de Projeto e Análise de Algoritmos

Nova Friburgo
2025

Resumo

Este trabalho tem como objetivo analisar experimentalmente o desempenho dos algoritmos **Insertion-Sort** e **Quick-Sort**. Para isso, foram realizados testes com vetores de tamanhos variados (50, 500, 5000 e 50.000 elementos), organizados em três ordens distintas: **crescente**, **decrescente** e **aleatória**. O tempo de execução de cada algoritmo foi mensurado, permitindo uma análise prática que complementa as previsões teóricas de complexidade. Os resultados obtidos são apresentados por meio de gráficos e discutidos de forma detalhada. A análise evidencia o impacto da natureza da entrada sobre a eficiência dos algoritmos de ordenação.

1. Introdução

A análise de algoritmos é um pilar essencial da ciência da computação. Embora a análise assintótica forneça estimativas teóricas de desempenho, é a avaliação empírica que revela o comportamento real dos algoritmos frente às particularidades do ambiente e dos dados de entrada.
Neste trabalho, comparamos dois algoritmos de ordenação com características distintas:

- **Insertion-Sort**: algoritmo simples, baseado em comparações diretas e inserções.
- **Quick-Sort**: algoritmo eficiente, estruturado com base na estratégia de divisão e conquista.

Ambos os algoritmos foram implementados manualmente e avaliados sob diferentes condições para verificar sua performance prática.

2. Metodologia

2.1 O que os algoritmos fazem

- **Insertion-Sort**: percorre o vetor da esquerda para a direita, inserindo cada elemento na posição correta dentro da porção já ordenada. Seu desempenho é bom para listas pequenas ou quase ordenadas, com complexidade de O(n)O(n)O(n) no melhor caso e O(n2)O(n^2)O(n2) no pior.

- **Quick-Sort**: seleciona um pivô (neste caso, o elemento central do vetor), particiona o vetor em elementos menores e maiores que o pivô, e aplica o algoritmo recursivamente em cada partição. Seu desempenho médio é O(nlogn)O(n \log n)O(nlogn), com pior caso O(n2)O(n^2)O(n2), embora este último seja mitigado por uma boa escolha de pivô.

2.2 Linguagem de programação utilizada

- **Linguagem:** Python 3

- **Bibliotecas:**

    - `time` para medir o tempo de execução

    - `matplotlib` para geração de gráficos

    - `pandas` para manipulação dos dados

    - `tqdm` para visualização de progresso

Todos os algoritmos foram implementados manualmente, sem uso de funções prontas como `sort()` ou `sorted()`.

2.3 Configuração do computador

Os experimentos foram executados em um computador com as seguintes especificações:

- **Sistema Operacional:** Linux Ubuntu 22.04 LTS

- **Processador:** Intel Core i5

- **Memória RAM:** 8 GB

- **Ambiente de execução:** Terminal com ambiente virtual Python

Essa configuração garante uma base estável para medição dos tempos de execução, mas ressalta-se que resultados podem variar em ambientes distintos.

3. Resultados

Os algoritmos foram testados com vetores em ordem **aleatória**, **crescente** e **decrescente**, nos tamanhos 50, 500, 5000 e 50000. A seguir, apresentam-se os gráficos com os respectivos comentários.

Vetor Aleatório

Análise:
Neste cenário, o **Quick-Sort** apresenta desempenho amplamente superior. Seu tempo cresce de maneira suave mesmo com 50 mil elementos, confirmando a eficiência da abordagem por divisão e conquista.

Já o **Insertion-Sort** sofre fortemente com o aumento da entrada: o tempo cresce de forma quadrática, tornando-se inviável em contextos com grandes volumes de dados.

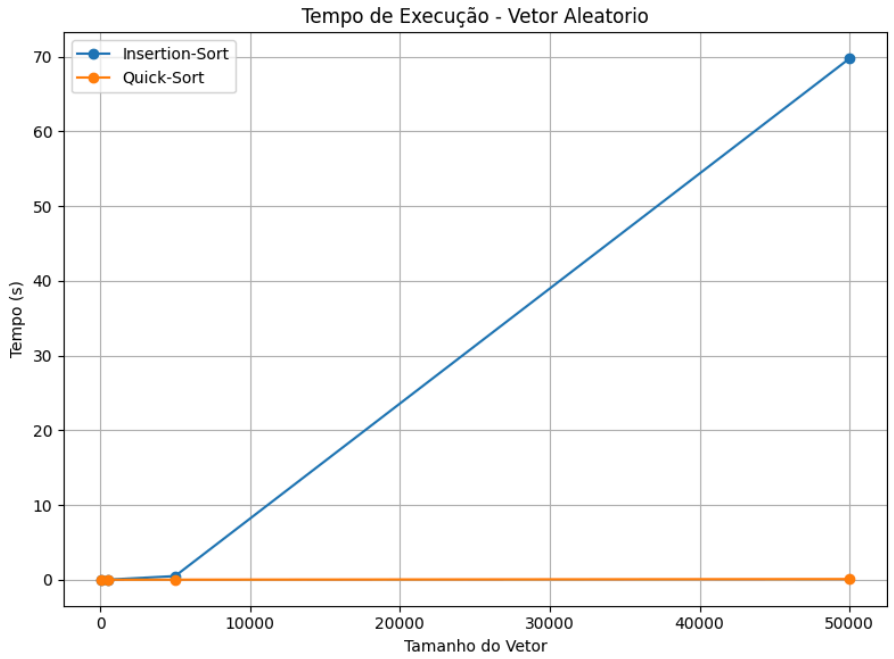

Figura 1: Tempo de execução – Vetor Aleatório

Vetor Crescente

Análise:
Com vetores já ordenados, o **Insertion-Sort** se beneficia da estrutura da entrada. Como não há necessidade de realocações, o algoritmo se aproxima do melhor caso, com comportamento linear.

Já o **Quick-Sort**, mesmo com pivô central, realiza partições e chamadas recursivas desnecessárias, o que o torna menos eficiente neste cenário específico.

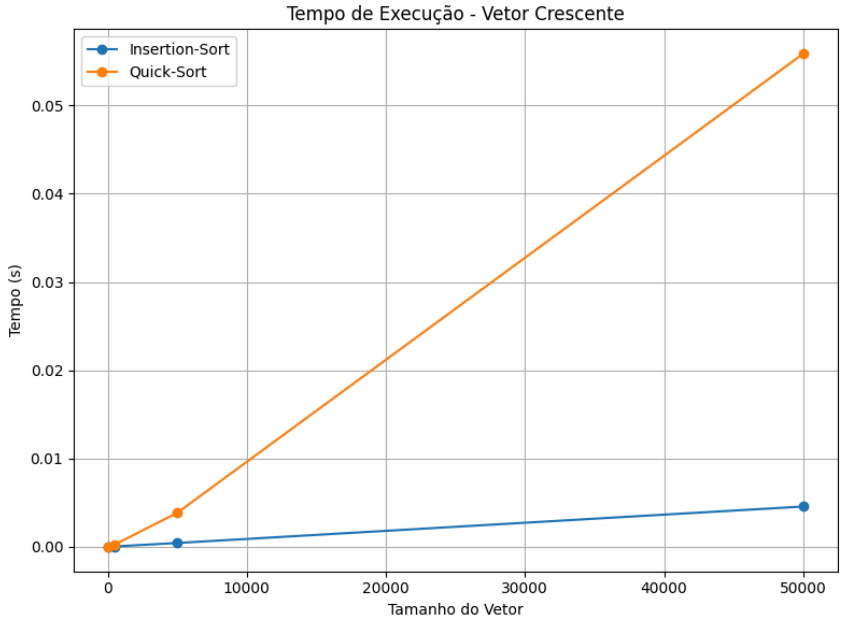

Figura 2: Tempo de execução – Vetor Crescente

Vetor Decrescente

Análise:
Este é o pior caso para o **Insertion-Sort**. Cada elemento novo precisa ser deslocado até o início da lista, o que gera um número elevado de comparações e trocas.

O **Quick-Sort**, por outro lado, mantém sua eficiência, demonstrando estabilidade mesmo diante de entradas completamente ordenadas de forma inversa.

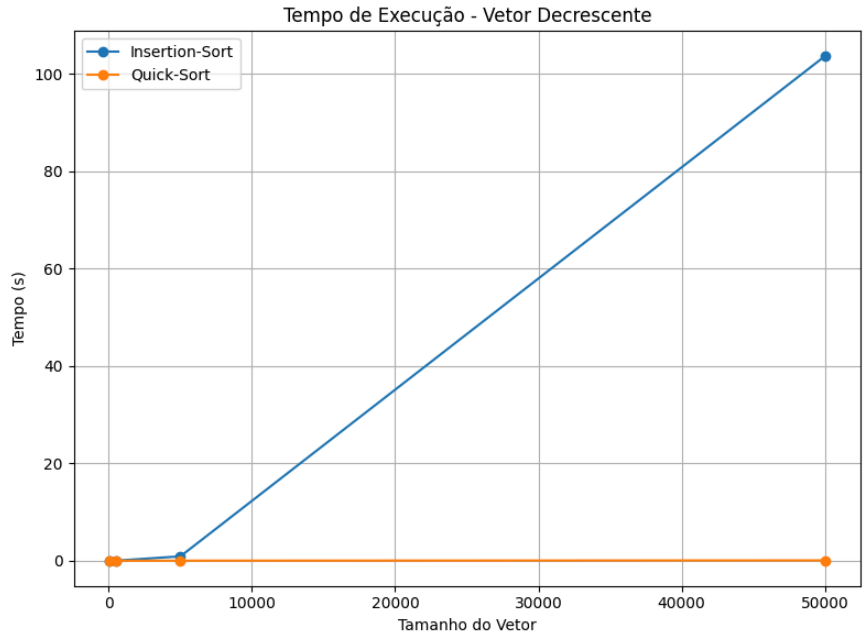

Figura 3: Tempo de execução – Vetor Decrescente

4. Conclusão

Com base nos experimentos realizados, é possível concluir que:

- O **Quick-Sort** é, de maneira geral, o algoritmo mais eficiente e indicado para vetores de tamanho médio ou grande, sobretudo quando a ordem dos elementos é imprevisível.

- O **Insertion-Sort** é útil apenas em contextos muito específicos, como listas pequenas ou já ordenadas, onde sua simplicidade se traduz em alta velocidade.

- A análise experimental reforça os conceitos teóricos: **Quick-Sort** oferece desempenho médio O(nlogn)O(n \log n)O(nlogn), enquanto o **Insertion-Sort** é penalizado com crescimento quadrático.

- A escolha do algoritmo ideal depende diretamente do tipo de entrada e do contexto de aplicação.

5. Referências

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to Algorithms*. MIT Press, 2009.

- Python Software Foundation. Documentação oficial do Python

- Slides da disciplina de Projeto e Análise de Algoritmos – UERJ