Projeto e Análise de Algoritmos
Algoritmos de Ordenação: Quick-Sort e Insertion-Sort

Guilherme Rodrigues Cler Gabril

Junho de 2025

1. Introdução:

A ordenação de dados é uma das bases mais importantes da Engenharia da Computação. Ela está presente em inúmeros sistemas e aplicações, sendo essencial para tornar operações como busca, inserção e remoção de dados mais rápidas e eficientes. Existem vários algoritmos que realizam essa tarefa, cada um com suas particularidades e desempenho em diferentes cenários. Neste trabalho, o foco é em dois deles: o Quick-Sort e o Insertion-Sort.

O Quick-Sort (ou ordenação rápida) é um algoritmo eficiente que segue a estratégia de dividir e conquistar. A ideia principal é escolher um elemento como pivô e, a cada etapa, reorganizar os demais elementos em torno dele: os menores à esquerda e os maiores à direita. Esse processo se repete recursivamente em cada parte até que todo o vetor esteja ordenado.

Já o Insertion-Sort (ou ordenação por inserção) tem uma abordagem mais simples e intuitiva. Ele percorre o vetor inserindo cada elemento em sua posição correta, comparando com os anteriores. Se encontrar um valor maior, realiza as trocas necessárias até que o item esteja no lugar certo.

Ao longo deste trabalho, é analisado o funcionamento, a complexidade e os contextos em que cada um desses algoritmos se destaca.

2. Metodologia

2.1 O que faz o Algoritmo?

2.1.1 Quick-Sort

O algoritmo Quick-Sort será aplicado a vetores gerados com três tipos distintos de ordenação inicial:

Ordem crescente: valores inteiros de 1 até n, onde n representa o tamanho do vetor.

Ordem decrescente: valores de n até 1, em ordem inversa.

Ordem aleatória: vetor com n elementos contendo valores inteiros gerados aleatoriamente no intervalo de 0 a 99.999.

Para a análise, foram utilizados vetores de quatro tamanhos diferentes: 50, 500, 5.000 e 50.000 elementos. Para cada tamanho, são gerados três vetores, um para cada tipo de ordenação inicial, totalizando doze execuções do algoritmo.

O Quick-Sort utilizado escolhe um pivô aleatório em cada etapa e organiza o vetor com base nesse pivô, posicionando os valores menores à esquerda e os maiores à direita. Em seguida, o processo é repetido recursivamente para as subpartes do vetor, seguindo a estratégia de dividir e conquistar, até que todo o vetor esteja ordenado.

O código também registra o tempo de execução da ordenação para cada combinação de tamanho e tipo de ordenação inicial, permitindo uma análise comparativa de desempenho.

2.1.2 Insertion-Sort

O algoritmo Insertion Sort também foi aplicado aos vetores gerados com três tipos distintos de ordenação inicial:

Ordem crescente: valores inteiros de 1 até n, onde n representa o tamanho do vetor.

Ordem decrescente: valores de n até 1, em ordem inversa.

Ordem aleatória: vetor com n elementos contendo valores inteiros gerados aleatoriamente no intervalo de 0 a 99.999.

Assim como no Quick-Sort, foram utilizados vetores de quatro tamanhos diferentes: 50, 500, 5.000 e 50.000 elementos. Para cada tamanho, são

gerados três vetores, um para cada tipo de ordenação inicial, totalizando doze execuções do algoritmo.

O Insertion Sort funciona de forma a percorrer o vetor da esquerda para a direita, comparando cada elemento com os anteriores e inserindo-o na posição correta. Se o valor atual for menor que os anteriores, os elementos maiores são deslocados para a direita, e a inserção ocorre na posição adequada. Esse processo se repete até que todo o vetor esteja ordenado.

O código também registra o tempo de execução da ordenação para cada combinação de tamanho e tipo de ordenação inicial, permitindo uma análise comparativa de desempenho.

2.2 Qual a linguagem utilizada?

A linguagem de programação utilizada foi o Python, devido à sua simplicidade e a disponibilidade de bibliotecas que facilitam o desenvolvimento e a análise de algoritmos. No projeto, foram utilizadas as seguintes bibliotecas:
random: responsável pela geração de valores aleatórios utilizados na criação dos vetores de entrada para os algoritmos de ordenação.
time: utilizada para medir o tempo de execução de cada algoritmo, possibilitando a análise de desempenho.
matplotlib.pyplot: empregada na construção de gráficos que representam visualmente os resultados obtidos nos testes.
Essa combinação de ferramentas tornou o processo de implementação, medição e visualização dos resultados mais prático e eficiente.

2.3 Qual a configuração do desktop?

Configurações do Hardware:
Placa de Vídeo: NVIDIA GeForce RTX 4060 ti
Memória RAM: 32GB
Processador: AMD Ryzen 7 5700X 8-Core Processor
Armazenamento disponível: 380 GB

Sistema Operacional: Windows 11

3. Resultados

A seguir, será realizada a análise dos resultados, com foco no comportamento dos algoritmos Quick-Sort e Insertion-Sort na ordenação de vetores com diferentes características iniciais.
O tópico 3.1 apresenta a avaliação do tempo de ordenação para vetores que estavam inicialmente em ordem crescente.
O tópico 3.2 analisa o tempo necessário para ordenar vetores em ordem decrescente.
O tópico 3.3 aborda a ordenação de vetores com valores dispostos de forma aleatória.
Em cada uma dessas etapas, será observada também a influência do tamanho do vetor no desempenho de cada algoritmo, permitindo uma comparação mais completa sobre a eficiência de ambos os métodos em diferentes cenários.

3.1 Ordenação de Vetores Inicialmente em Ordem Crescente

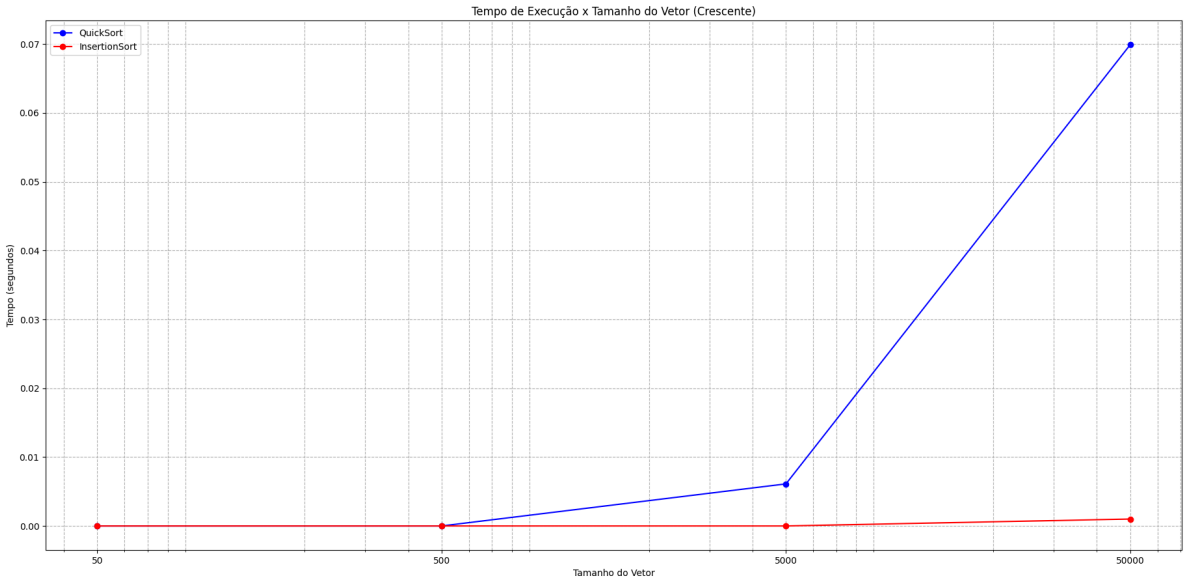

Figura 1: Gráfico dos métodos de ordenação Quick-Sort e Insertion-Sort para Dados inicialmente em ordem Crescente.

A partir da análise da figura 1, é possível observar que, para entradas pequenas (como vetores de tamanho 50 e 500), ambos os algoritmos apresentam tempos de execução semelhantes. No entanto, à medida que o tamanho da entrada aumenta significativamente, o tempo de execução do Quick-Sort cresce de forma expressiva em comparação ao Insertion-Sort, mesmo quando o vetor já está previamente ordenado.
Por exemplo, para um vetor de 50.000 elementos em ordem crescente, os tempos registrados foram: Quick-Sort: 0,06992 segundos e Insertion-Sort: 0,00100 segundos
Ou seja, mesmo que o Quick-Sort apresente complexidade O(n log n) no caso médio e o Insertion Sort tenha complexidade O(n²) no pior caso, neste cenário específico o Insertion Sort foi mais eficiente. Isso se deve ao fato de que a entrada já estava ordenada, o que representa o melhor caso para o Insertion Sort, no qual ele realiza apenas comparações sequenciais, sem necessidade de realizar trocas de posição. Já o Quick-Sort, mesmo diante de um vetor ordenado, ainda executa todas as chamadas recursivas, o que compromete seu desempenho nesse tipo de entrada.

3.2 Ordenação de Vetores Inicialmente em Ordem Decrescente

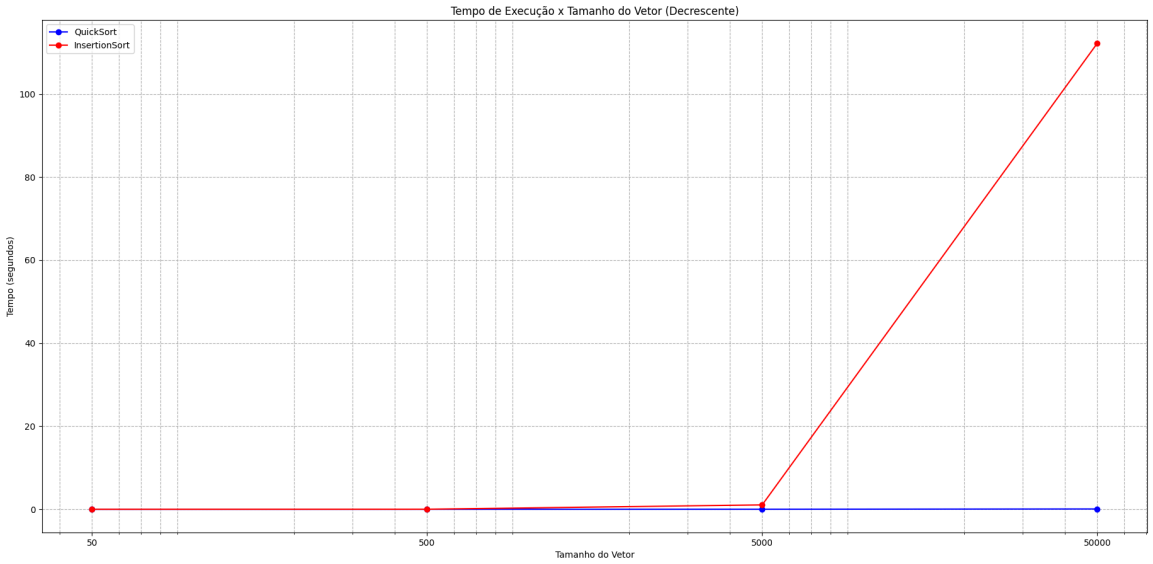

Figura 2: Gráfico dos métodos de ordenação Quick-Sort e Insertion-Sort para Dados inicialmente em ordem Decrescente.

A partir da análise da figura 2, é possível observar que, para entradas menores como vetores com 50 até mesmo 5.000 elementos, ambos os algoritmos apresentam tempos de execução relativamente próximos. No entanto, à medida que o tamanho da entrada aumenta significativamente, o tempo de execução do Insertion Sort cresce de forma absurda, destacando sua limitação de desempenho em certos cenários.Esse comportamento ocorre porque, no caso de vetores ordenados em ordem decrescente, o Insertion Sort enfrenta seu pior caso de desempenho. Cada novo elemento analisado precisa ser comparado e trocado sucessivamente com todos os elementos anteriores, o que gera um alto número de operações. Por exemplo, para um vetor com 50.000 elementos em ordem decrescente, os tempos registrados foram:
Quick-Sort: 0,07205 segundos e Insertion Sort: 112,21360 segundos
Esses resultados evidenciam a eficiência do Quick-Sort em entradas desordenadas ou contrárias à ordem final desejada, graças à sua estratégia de divisão e conquista. Já o Insertion Sort se mostra ineficiente nesse caso, devido à grande quantidade de comparações e movimentações necessárias.

3.3 Ordenação de Vetores Inicialmente em Ordem Aleatória

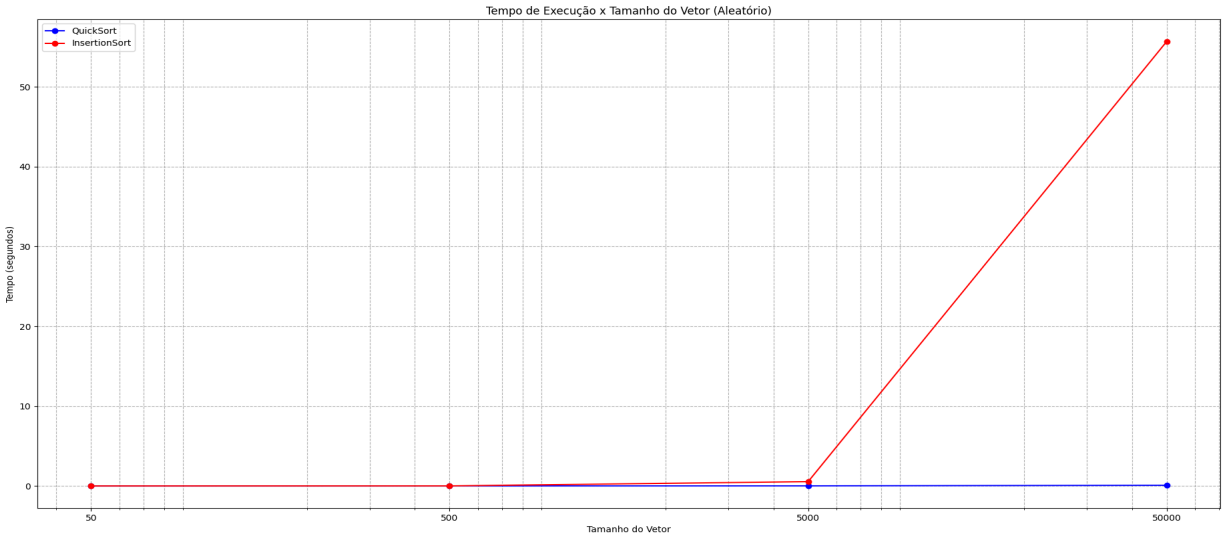

Figura 3: Gráfico dos métodos de ordenação Quick-Sort e Insertion-Sort para Dados inicialmente em ordem Aleatória.

A partir da análise da figura 3, é possível observar que, para vetores de tamanho reduzido (50, 500 e até mesmo 5.000 elementos), os tempos de execução do Quick-Sort e do Insertion Sort permanecem relativamente próximos. No entanto, à medida que o tamanho da entrada aumenta significativamente, o tempo de execução do Insertion Sort cresce de forma acentuada, evidenciando suas limitações de desempenho diante de entradas maiores.
Um exemplo claro disso pode ser visto no vetor com 50.000 elementos com valores distribuídos aleatoriamente, no qual os tempos registrados foram: Quick-Sort: 0,08134 segundos e Insertion Sort: 55,69198 segundos
Esse resultado confirma a eficiência do Quick-Sort em entradas não ordenadas, graças à sua estratégia de divisão e conquista e à sua complexidade média de O(n log n). Já o Insertion Sort, com complexidade $O(n^2)$, se mostra significativamente mais lento em situações que envolvem grande volume de dados e ordenações mais complexas.

4. Conclusão

Com base nas análises gráficas (figura 1, figura 2 e figura 3) realizadas, é possível concluir que o desempenho dos algoritmos Quick-Sort e Insertion-Sort varia significativamente de acordo com a configuração inicial dos vetores e o tamanho da entrada.
Para vetores já ordenados em ordem crescente, o Insertion-Sort se mostra mais eficiente, desde que o Quick-Sort não possua uma verificação prévia de ordenação. Isso ocorre porque o Quick-Sort, mesmo diante de um vetor já organizado, ainda executa todas as etapas do processo de divisão e conquista, o que gera um custo computacional desnecessário. Além disso, o desempenho do Quick-Sort é sensivelmente influenciado pela escolha do pivô, se o pivô for mal posicionado (como no início ou fim do vetor em casos extremos), o algoritmo pode se aproximar do pior caso, com complexidade $O(n^2)$.

Em contrapartida, para vetores em ordem decrescente, o Insertion-Sort enfrenta seu pior cenário, exigindo um número elevado de comparações e trocas, o que compromete seu desempenho. Já o Quick-Sort, com a

escolha aleatória de pivô utilizada neste trabalho, consegue manter um desempenho estável e superior, graças à sua complexidade média de O(n log n).

No caso dos vetores com valores distribuídos aleatoriamente, o Quick-Sort novamente se destaca, apresentando tempos de execução significativamente menores do que o Insertion-Sort, confirmando sua eficiência.

Por fim, é importante destacar que essas diferenças de desempenho se tornam mais evidentes em entradas maiores, como vetores de 50.000 elementos. Para vetores menores, a diferença entre os tempos de execução dos dois algoritmos é pouco significativa, não havendo grande discrepância de desempenho.

Bibliografia:

https://www.ic.unicamp.br/~raquel.cabral/pdf/Aula25.pdf - Acessado: 06/06/2025 - utilizado como base para o Algoritmo Quick-Sort
https://pt.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort - - Acessado: 06/06/2025 - utilizado como base para o Algoritmo Insertion-Sort