IPRJ
Universidade do Estado
do Rio de Janeiro

Trabalho Projeto e Análise de Algoritmos

Quick-Sort e Insertion-Sort

Aluno: Luís Felipe dos Santos Braido

Matrícula: 202110050111

Professora: Camila Martins Saporetti

1.Introdução

Nesse trabalho o objetivo proposto foi implementar os algoritmos de ordenação de vetores de acordo com o número de matrícula. Para descobrir o número mencionado era necessário pegar os seis números intermediários (XXXX123456XX) da matrícula, e dividir por 5, o resto dessa divisão determinaria o conjunto de algoritmos que seria implementado e analisado. O resultado obtido foi o número um, portanto os algoritmos a serem analisados seriam o Quick-Sort e Insertion-Sort.

Regras: Não utilizar funções prontas de bibliotecas para os métodos de ordenação! Pode usar para criação do vetor, adicionar em lista, ...

O Quick Sort e o Insertion Sort são dois algoritmos populares de ordenação que pertencem à categoria de algoritmos de comparação. Ambos os algoritmos visam organizar uma lista de elementos em ordem crescente ou decrescente. No entanto, eles diferem em suas abordagens e eficiência em diferentes situações.

1.1 Resumo

O Objetivo do trabalho é estudar e investigar a organização de conjuntos de números inteiros em ordem crescente, decrescente e aleatória usando os algoritmos Quick-Sort e Insertion-Sort. Serão analisados quatro tamanhos diferentes de conjuntos: 50, 500, 5000 e 50000 elementos. Através destes valores serão gerados gráficos, permitindo uma melhor visualização entre o tamanho do conjunto e o tempo de processamento necessário. Essa visualização facilitará a entender quais dos dois algoritmos é mais eficiente em diferentes situações.

2. Metodologia

2.1 O que o algoritmo faz:

- Algoritmo Quick-Sort

O Quick-Sort é um algoritmo de ordenação amplamente utilizado que segue a abordagem de "dividir para conquistar", e é muito conhecido por sua eficiência e desempenho em grande parte dos casos

O Algoritmo funciona da seguinte forma:

Primeiro, é escolhido um elemento como pivô a partir do conjunto a ser ordenado. O objetivo é rearranjar os elementos do conjunto de forma que todos os elementos menores que o pivô estejam à sua esquerda, e todos os elementos maiores estejam à sua direita. Esse processo é conhecido como particionamento.

Uma vez que o pivô esteja em sua posição correta, o algoritmo é aplicado recursivamente nos subconjuntos à esquerda e à direita do pivô. Isso é feito repetidamente até que todos estejam ordenados.

Uma das principais vantagens do Quick-Sort é que ele possui uma complexidade média de execução de O(n log n), o que o torna muito eficiente para conjuntos de dados grandes. No entanto, em casos desfavoráveis, o desempenho pode degradar para $O(n^2)$, onde n é o número de elementos a serem ordenados. Para mitigar esse problema, diferentes abordagens podem ser adotadas, como a escolha inteligente do pivô e a utilização de outros algoritmos de ordenação em casos específicos.

No código produzido, a função pivo() é responsável por encontrar o pivô e particionar a lista de entrada em duas partes. Ela recebe três parâmetros: vet (o vetor de entrada), baixo (o índice inferior) e alto (o índice superior).

A função utiliza o último elemento (vet[alto]) como pivô e percorre a lista do índice baixo até alto-1. Se um elemento for menor ou igual ao pivô, ele é trocado com o elemento imediatamente após o índice i. No final, o pivô é colocado na sua posição correta e é retornado o índice dessa posição.

A função quick() é implementada de forma iterativa utilizando um loop while. Ela recebe os mesmos parâmetros vet, baixo e alto e continua dividindo e ordenando as partições de forma recursiva. A escolha da próxima partição a ser ordenada é baseada no tamanho relativo das partições em relação ao pivô. Se a partição à esquerda for menor, ela é ordenada recursivamente e o índice inferior é atualizado. Caso contrário, a partição à direita é ordenada recursivamente e o índice superior é atualizado.

- # Algoritmo Insertion-sort

O Insertion sort, ou "ordenação por inserção", é um algoritmo de ordenação simples e intuitivo. Ele percorre uma lista de elementos, construindo um segmento ordenado no início da lista à medida que avança.

O processo de ordenação por inserção começa com o segundo elemento da lista. Esse elemento é comparado com os elementos anteriores no segmento ordenado. Se o elemento for menor, ele é movido para a esquerda, abrindo espaço para inserir o elemento em sua posição correta. Esse processo é repetido até que todos os elementos estejam na posição correta.

Essencialmente, o Insertion sort "insere" cada elemento na posição apropriada em relação aos elementos anteriores. À medida que a lista é percorrida, o segmento ordenado vai crescendo até que todos os elementos estejam ordenados.

O algoritmo possui uma complexidade média de tempo de execução de $O(n^2)$, onde n é o número de elementos a serem ordenados. Isso significa que, em casos de conjuntos de dados grandes, o tempo de execução pode ser significativo. No entanto, para conjuntos de dados menores ou já parcialmente ordenados, o Insertion sort pode ser eficiente e apresentar um desempenho melhor do que outros algoritmos mais complexos. Além disso, ele é estável, ou seja, preserva a ordem relativa dos elementos com chaves iguais durante o processo de ordenação.

No código produzido, a função insertion() é implementada usando um loop for para percorrer a lista de entrada. A cada iteração, o elemento atual é comparado com os elementos anteriores. Se um elemento anterior for maior que o elemento atual, ele é deslocado para a direita. Esse processo de deslocamento continua até que o elemento anterior seja menor ou igual ao elemento atual. Em seguida, o elemento atual é inserido na posição correta.O algoritmo continua percorrendo a lista e inserindo os elementos restantes em suas posições corretas.

2.2 Linguagem de programação utilizada:

A Linguagem utilizada no trabalho foi o Python, pela facilidade de implementação e pelas bibliotecas disponíveis para uso. a linguagem foi usada em um Jupyter notebook, pois permite escrever, executar e depurar códigos interativamente. Com ele, é possível experimentar e testar os códigos em células individuais, visualizando os resultados à medida que eles avançam.

2.3 Configurações do computador onde os testes foram executados:

Processador: Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz 3.00 GHz

RAM instalada: 8,00 GB

Sistema operacional: Windows 10

Tipo de sistema: Sistema operacional de 64 bits, processador baseado em x64

Placa de vídeo: NVIDIA GeForce GTX 1050

3. Resultados

Com as configurações do meu computador ao executar o programa, o tempo de processamento para realizar todas as análises e criar gráficos de comparação foi de 8 minutos e 55 segundos aproximadamente.

Obs: O tempo de exucução pode mudar dependendo do hardware em que for feito, o que pode impactar em mudança nos resultados.

Após a execução do programa, os resultados foram os seguintes:

1. Comparação dos Algoritmos em Ordem Crescente

Ao analisar a situação de ordenação em ordem crescente, é possível perceber que o algoritmo mais eficiente entre o Quick sort e o Insertion sort é o Insertion sort.

O motivo é que o Insertion sort possui uma característica especial que o torna mais eficiente para conjuntos de dados que já estão parcialmente ordenados, como é o caso da ordem crescente. Quando os elementos de um vetor já estão próximos de sua posição correta, o Insertion sort requer menos comparações e movimentações de elementos, o que resulta em um tempo de execução menor.

Por outro lado, o Quick sort é um algoritmo mais geral e não se beneficia diretamente da pré-ordenação do vetor. Ele realiza divisões e comparações em todos os elementos, mesmo que já estejam em ordem crescente, o que aumenta o tempo de execução em comparação com o Insertion sort.

Deste modo, na figura 1 podemos observar que mesmo o Quick sort possuindo vantagens pela sua ordem de complexidade ser O(n log n), o insertion sort é mais eficiente devido à sua particularidade de eficiência quando os elementos já estão próximos da sua posição correta. Fazendo dele muito mais rápido nessa situação.

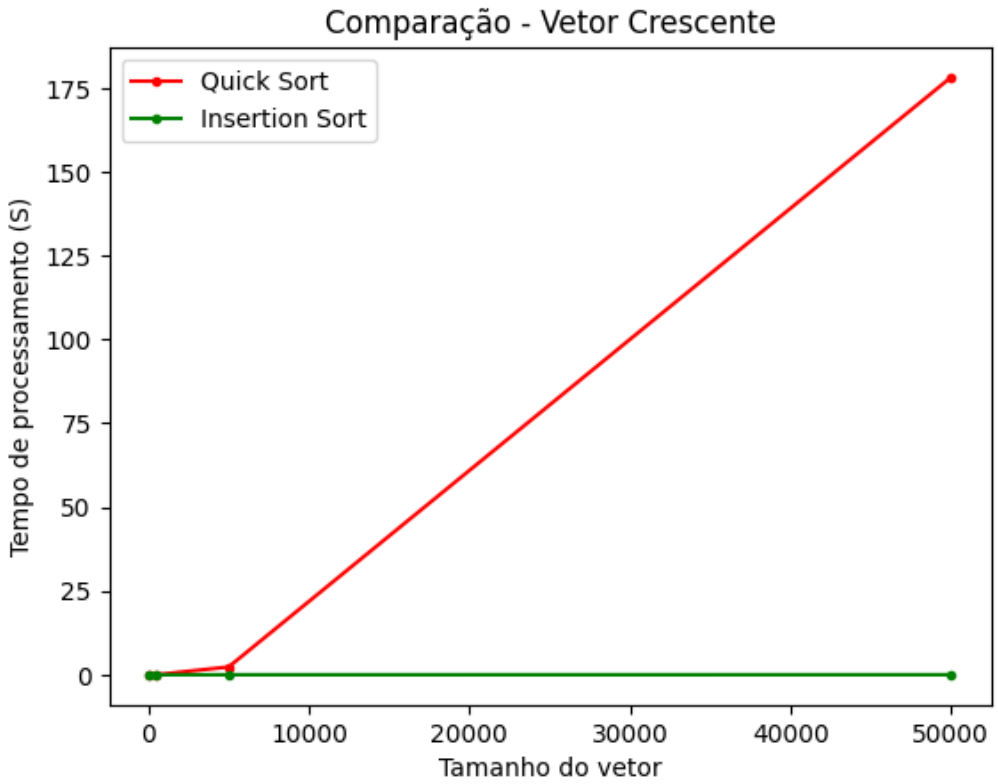

Figura 1 – Gráfico Comparativo dos algoritmos de ordenação Quick-Sort e Insertion-Sort em relação a ordem crescente.

2. Comparação dos Algoritmos em Ordem Decrescente

Ao analisar a situação de ordenação de ordem decrescente, é possível notar que neste caso o algoritmo mais eficiente entre o Quick sort e o Insertion sort é o Quick sort.

A razão para isso é que o Quick sort é um algoritmo de divisão e conquista que pode lidar eficientemente com conjuntos de dados desordenados, incluindo a ordem decrescente. Ele possui um bom desempenho em cenários onde há uma distribuição aleatória dos elementos.

Enquanto o Insertion sort, requer um número maior de comparações e movimentações de elementos para ordenar uma lista em ordem decrescente. À medida que o tamanho do vetor aumenta, o tempo de execução do Insertion sort tende a crescer de forma mais significativa, pois o número de comparações e movimentações necessárias aumenta exponencialmente.

Por outro lado, o Quick sort divide o vetor em subconjuntos menores com base em um pivô escolhido, rearranjando-os de forma eficiente. Isso permite que o algoritmo lide de forma mais eficaz com a ordem decrescente, reduzindo o tempo de execução em comparação com o Insertion sort.

Na figura 2, é possível notar que os dois crescem consideravelmente, mas o Quick sort leva certa vantagem neste caso pois leva menos tempo para realizar todas as comparações e trocas necessárias, devido ao seu método de divisão e conquista. Logo, é recomendado utilizar o Quick sort devido à sua eficiência em lidar com conjuntos de dados desordenados.

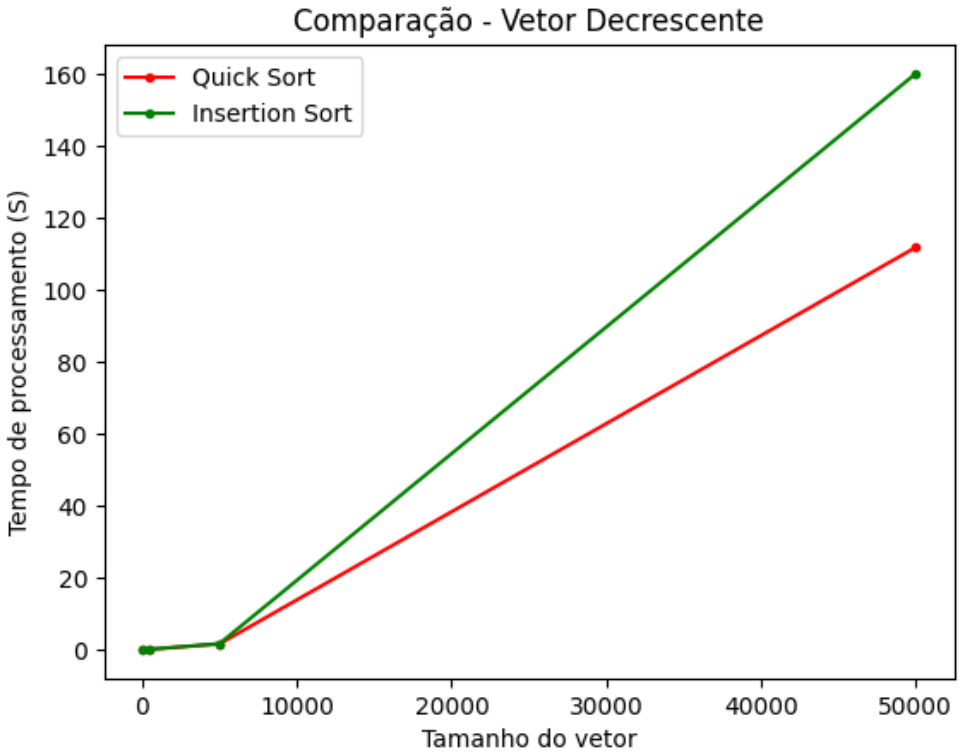

Figura 2 – Gráfico Comparativo dos algoritmos de ordenação Quick-Sort e Insertion-Sort em relação a ordem decrescente.

3. Comparação dos Algoritmos em Ordem Aleatória

Ao analisar a situação de ordem aleatória, é possível perceber que o algoritmo mais eficiente entre o Quick sort e o Insertion sort seria o Quick sort.

Pois, O Quick sort é um algoritmo de ordenação eficiente para conjuntos de dados aleatórios de tamanho moderado a grande. Utilizando a abordagem de “Dividir para conquistar”, permite que o Quick sort lide bem com uma variedade de padrões e distribuições de dados, incluindo ordem aleatória.

Por outro lado, o Insertion sort, embora seja eficiente para conjuntos pequenos ou parcialmente ordenados, possui uma complexidade quadrática ($O(n^2)$). Isso significa que à medida que o tamanho dos vetores aumenta, o tempo de execução do Insertion sort aumenta significativamente. Enquanto o Quick sort tem uma complexidade média de tempo de execução de O(n log n), o que torna sua execução mais rápida em relação ao Insertion sort para conjuntos de dados aleatórios de tamanho considerável.

Portanto, Na figura 3 é possível notar a grande diferença de tempo necessário para a execução dos algoritmos, na qual o Insertion sort cresce a medida que o tamanho do vetor cresce, e o Quick sort possui um crescimento muito menos significativo, ficando evidente a eficácia do Quick sort neste caso.

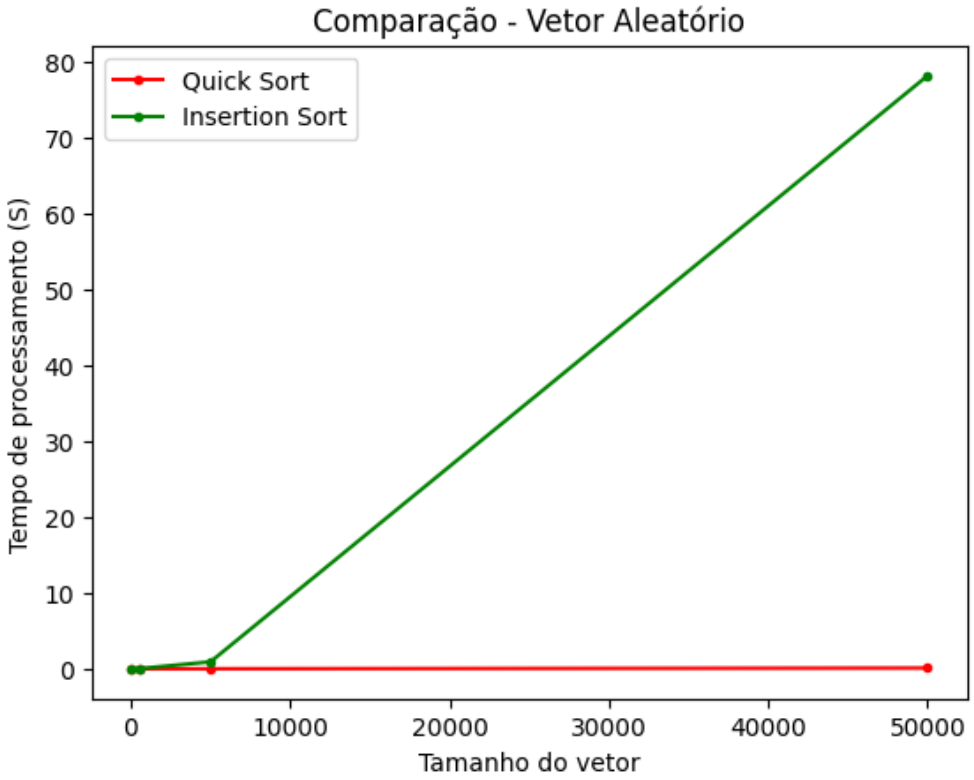

Figura 3 – Gráfico Comparativo dos algoritmos de ordenação Quick-Sort e Insertion-Sort em relação a ordem aleatória.

4.Conclusão

Desta maneira, Podemos concluir após as análises e comparações dos algoritmos Quick sort e Insertion sort, que o desempenho dos algoritmos irá depender dos vetores de entrada, a análise mostra que quando o vetor já está ordenado, o Insertion sort se sobressai em relação ao Quick sort devido à sua particularidade que o torna mais eficiente para conjuntos de dados que já estão parcialmente ordenados, como é o caso da ordem crescente. Porém, nos outros casos de ordem decrescente e aleatória, o Quick sort se destaca graças a sua estratégia de divisão e conquista, que reduz o número de trocas e comparações a serem realizadas.

Portanto, ao analisar e comparar de uma forma geral o Quick sort é mais eficiente para conjuntos de dados aleatórios, grandes ou desordenados graças a sua abordagem de dividir e conquistar. Enquanto o Insertion sort pode ser eficiente para conjuntos menores ou parcialmente ordenados.