Universidade Federal do Rio Grande do Norte
Centro de Ensino Superior do Seridó
Departmento de Computação e Tecnologia
Curso de Bacharelado em Sistemas de Informação

Atividade Avaliativa

Halyson Santos de Araújo

Caicó-RN
Maio de 2023

Halyson Santos de Araújo

Avaliação

Trabalho apresentado a disciplina estrutura de dados do Departamento de Informática da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção da nota da avaliação I.

Orientador

João Paulo de Souza Medeiros

BSI – Bacharelado em Sistemas de Informação
DCT – Departamento de Computação e Tecnologia
CERES – Centro de Ensino Superior do Seridó
UFRN – Universidade Federal do Rio Grande do Norte

Caicó-RN

Maio de 2023

Avaliação 01

Autor: Halyson Santos de Araújo
Supervisor: João Paulo de Souza Medeiros

Resumo

Trata-se de uma atividade avaliativa da disciplina de estrutura de dados, focada na análise de algoritmos de ordenação. O trabalho consiste na primeira tarefa avaliativa que aborda os conceitos relacionados à análise de algoritmos, onde será realizada uma avaliação da complexidade dos algoritmos e o estudo dos diferentes métodos de ordenação, incluindo selection-sort(), insertion-sort(), merge-sort(), quick-sort() e distribution-sort(). Durante a atividade, será realizado um exame comparativo das características, desempenho e eficiência de cada algoritmo, explorando aspectos como os passos envolvidos, complexidade temporal (tempo de execução), complexidade espacial (uso de memória), vantagens e desvantagens de cada abordagem, bem como a identificação de cenários adequados para a aplicação de cada algoritmo.

Palavras-chave: estrutura de dados, análise de algoritmos, algoritmos de ordenação, eficiência dos algoritmos, complexidade,

Assessment 01

Author: Halyson Santos de Araújo
Supervisor: João Paulo de Souza Medeiros

Abstract

This is an evaluative activity of the data structure discipline, focused on the analysis of sorting algorithms. The work consists of the first evaluative task that approaches the concepts related to the analysis of algorithms, where an evaluation of the complexity of the algorithms will be carried out and the study of the different sorting methods, including selection-sort(), insertion-sort(), merge- sort(), quick-sort() and distribution-sort(). During the activity, a comparative examination of the characteristics, performance and efficiency of each algorithm will be carried out, exploring aspects such as the steps involved, temporal complexity (execution time), spatial complexity (memory use), advantages and disadvantages of each approach, as well as as the identification of suitable scenarios for the application of each algorithm.

Keywords: data structure, algorithm analysis, sorting algorithms, algorithm efficiency, complexity,

Sumário

1 Introdução

Este trabalho de pesquisa tem como foco a avaliação e análise de algoritmos de ordenação, abrangendo os conceitos relacionados à análise de algoritmos dentro do contexto da disciplina de Estrutura de Dados. O objetivo deste estudo é obter insights sobre a complexidade, desempenho e eficiência de diversos métodos de ordenação. A primeira tarefa avaliativa do curso explora a análise de diferentes algoritmos, incluindo selection-sort(), insertion-sort(), merge-sort(), quick-sort() e distribution-sort(). Por meio de uma análise comparativa, este estudo explora as características, os passos envolvidos, a complexidade temporal (tempo de execução), a complexidade espacial (uso de memória), bem como as vantagens e desvantagens associadas a cada algoritmo. Além disso, será dada ênfase à identificação de cenários adequados para a aplicação de cada algoritmo de ordenação. Ao investigar esses aspectos de forma abrangente, esta pesquisa tem como objetivo contribuir para uma compreensão mais profunda do funcionamento interno e implicações práticas dos algoritmos de ordenação.

1.1 Organização do trabalho

Usando o python como codigo base, implementei 3 algoritmos de ordenação. O codigo tem como base uma lista com os tamanhos dos arrays que serão testados e a quantidade de teste que cada array será submetido, na sequência entra em um loop onde a lista é igual ao array gerado,é coletado o tempo inicial, é chamada a função do algoritmo e após isso o tempo final é coletado, acontece o calculo do tempo final menos o inicial e mutiplicamos por mil para ficar em milisegundos, esse loop se repete n vezes até que termine de ordenar o array. após isso é feita uma media de tempo para cada tamanho de array passado, em seguida é gravado no arquivo que utilizamos para gerar o grafico.

2 Algoritmos

2.1 Insertion-sort

O insertion-sort é um algoritmo eficiente de ordenação que percorre uma lista de elementos e os insere em sua posição correta em uma sublista ordenada. Ele é especialmente adequado para lidar com listas de tamanho pequeno ou quase ordenadas. O algoritmo compara os elementos com a sublista ordenada e os move para a posição correta, resultando em uma lista totalmente ordenada. Ele tem a vantagem de ter uma implementação simples e requer menos comparações e movimentos de elementos em comparação com outros algoritmos de ordenação.

Complexidade temporal no melhor caso:

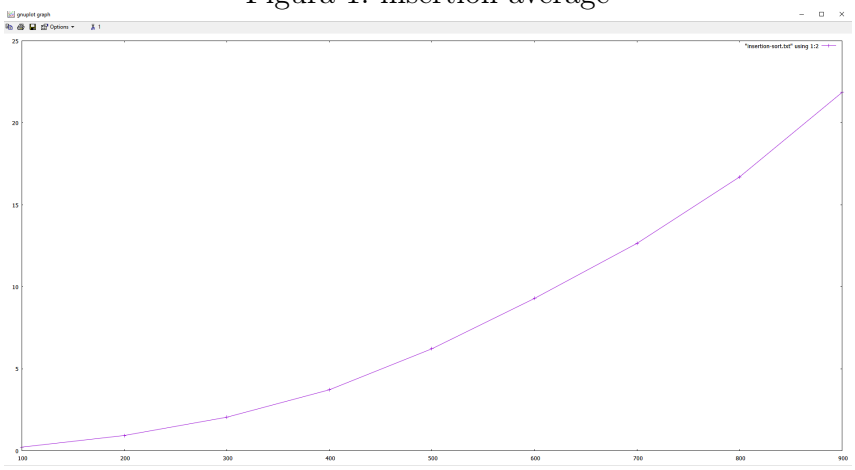
$$C(n) = O(n) \tag{2.1}$$

Complexidade temporal no caso base e pior caso:

$$C(n) = O(n^2) \tag{2.2}$$

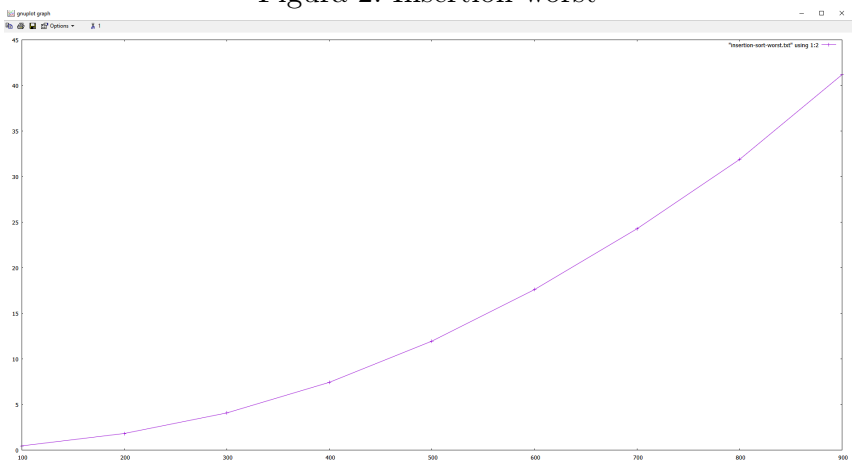
A complexidade temporal do insertion-sort é O(n$\hat{2}$) no pior caso e O(n) no melhor caso, onde n é o número de elementos na lista. A complexidade espacial é O(n), pois o algoritmo opera in-place, ou seja, não requer espaço adicional além da lista de entrada.

Figura 1: insertion-average



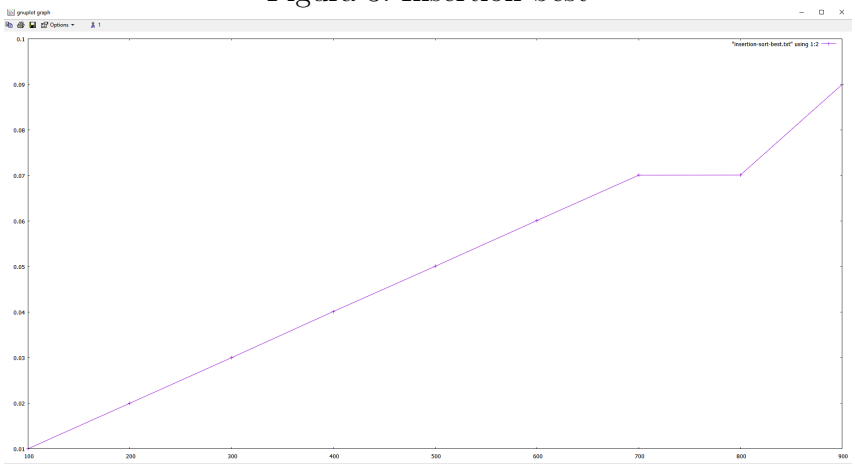
No caso médio do insertion-sort, estamos lidando com uma situação em que a lista de entrada não está totalmente ordenada, mas também não está em ordem reversa. Nesse cenário, o algoritmo ainda percorre a lista e insere cada elemento em sua posição correta dentro de uma sublista ordenada. No entanto, como a lista não está perfeitamente ordenada, são necessárias algumas comparações e movimentos de elementos.

Figura 2: Insertion-worst



No pior caso do insertion-sort, encontramos uma situação em que a lista de entrada está ordenada em ordem decrescente. Nesse cenário desafiador, o algoritmo terá que realizar o máximo de movimentações possíveis para inserir cada elemento em sua posição correta na sublista ordenada.

Figura 3: Insertion-best

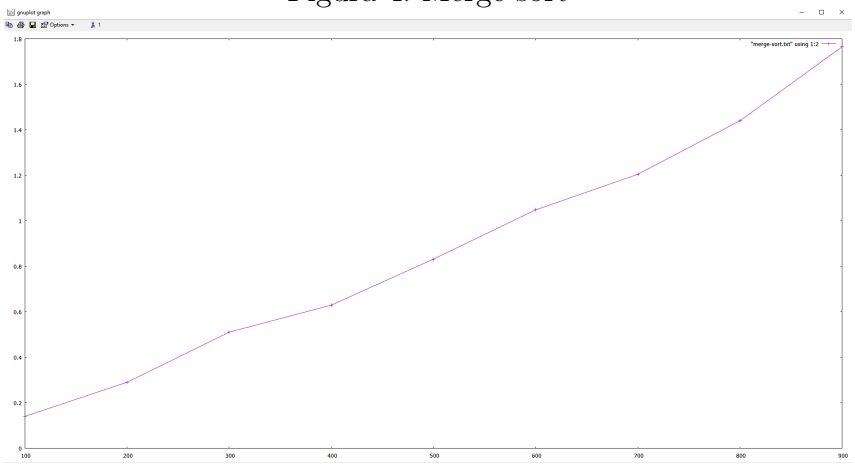


No melhor caso do insertion-sort, temos uma situação em que a lista de entrada já está ordenada em ordem crescente. Nesse cenário, o algoritmo realiza o percurso da lista, mas não precisa fazer nenhuma movimentação de elementos, pois cada elemento já está na posição correta dentro da sublista ordenada. Portanto, o insertion-sort simplesmente verifica cada elemento e passa para o próximo sem executar nenhuma troca.

2.2 Merge-sort

O merge-sort é um algoritmo de ordenação eficiente que utiliza o conceito de dividir para conquistar. Ele divide a lista em sublistas menores, ordena cada sublista separadamente e, em seguida, combina essas sublistas de forma ordenada para obter a lista final ordenada. O merge-sort é conhecido por sua eficiência, estabilidade e capacidade de lidar com grandes conjuntos de dados. Ele tem uma complexidade temporal de O(n log n) e preserva a ordem relativa dos elementos com chaves iguais. Apesar de ter um requisito adicional de espaço, o merge-sort é amplamente utilizado devido à sua eficiência e adaptabilidade a diferentes tipos de dados.

Figura 4: Merge-sort

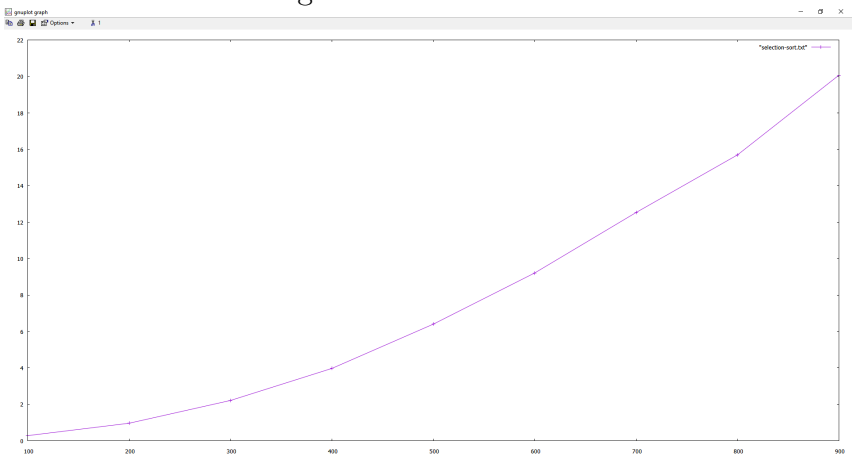


Durante a execução do merge-sort, o processo de mesclagem ocorre quando duas sublistas ordenadas são combinadas para formar uma sublista maior e ordenada. Esse processo de mesclagem é repetido até que todas as sublistas tenham sido combinadas e a lista final esteja completamente ordenada.

2.3 Selection-sort

O selection-sort é um algoritmo de ordenação simples em que o menor elemento é selecionado a cada iteração e colocado na posição correta. Ele divide a lista em duas partes: uma sublista ordenada e uma sublista não ordenada. O algoritmo percorre a sublista não ordenada, encontra o menor elemento e o troca com o primeiro elemento da sublista ordenada. Esse processo se repete até que toda a lista esteja ordenada.

Figura 5: Selection-sort

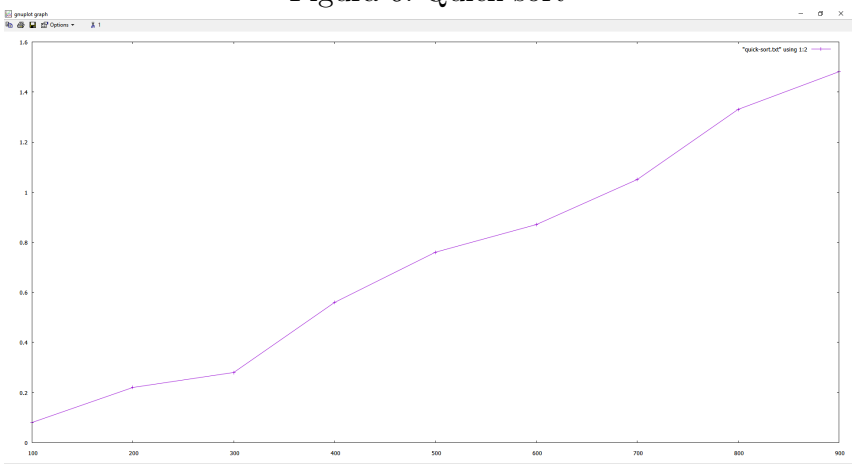


Durante a execução do selection-sort, o algoritmo percorre a lista várias vezes para encontrar o menor elemento e colocá-lo na posição correta. O selection-sort repete esse processo até que todos os elementos estejam colocados em suas posições corretas. A cada iteração, a sublista ordenada cresce, enquanto a sublista não ordenada diminui. Esse processo de encontrar o menor elemento e trocá-lo se repete até que a lista esteja completamente ordenada.

2.4 Quick-sort

O Quick Sort é um algoritmo de ordenação que divide a lista em subpartes menores, usando um elemento chamado pivô. Os elementos menores que o pivô são colocados à sua esquerda, e os maiores, à sua direita. Esse processo é repetido nas subpartes até que a lista esteja completamente ordenada. O Quick Sort é eficiente na maioria dos casos, mas pode ter desempenho ruim em situações específicas.

Figura 6: Quick-sort

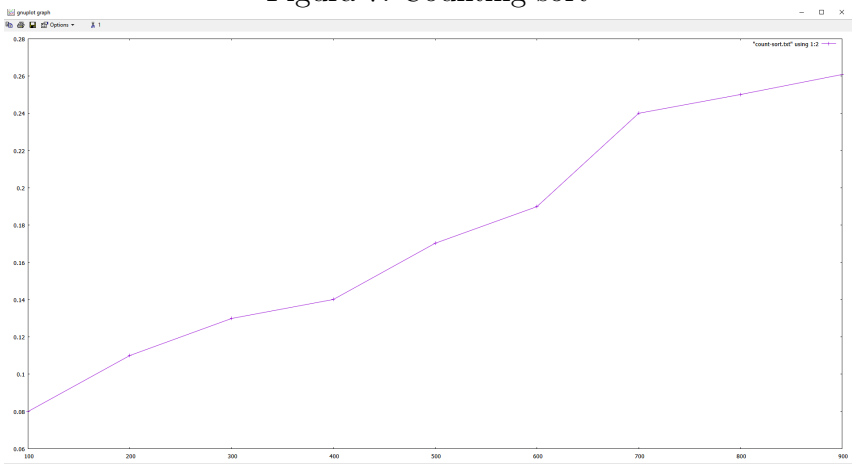


O pivô desempenha um papel crucial no desempenho do Quick Sort. Uma boa escolha de pivô pode levar a uma divisão equilibrada da lista, reduzindo o número de comparações e trocas. Por outro lado, uma escolha ruim de pivô pode levar a um desequilíbrio na divisão, resultando em um pior desempenho do algoritmo.

2.5 Counting-sort

O Counting Sort possui uma eficiência linear, com complexidade de tempo O(n+k), onde n é o número de elementos na lista original e k é o intervalo máximo dos elementos. Ele é especialmente eficiente quando o intervalo de elementos é pequeno em relação ao tamanho da lista. No entanto, o Counting Sort requer que os elementos sejam inteiros não negativos e que o intervalo máximo seja conhecido ou possa ser determinado de antemão. Além disso, ele consome uma quantidade significativa de memória adicional para o array de contagem, o que pode ser problemático em listas com valores muito grandes ou com intervalos muito amplos.

Figura 7: Counting-sort



O Counting Sort tem uma execução eficiente, com uma complexidade de tempo linear, tornando-o adequado para casos em que o intervalo de valores é pequeno em relação ao tamanho da lista. No entanto, ele requer que os elementos sejam inteiros não negativos e que o intervalo máximo seja conhecido ou possa ser determinado antecipadamente.

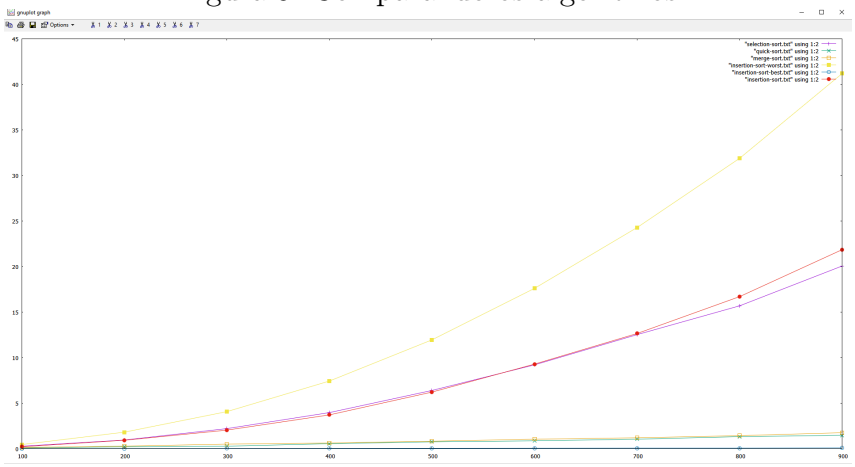
2.6 Comparando tempo de execução

Em resumo, o merge-sort tende a ter um desempenho melhor do que o selection-sort e o insertion-sort em termos de tempo de execução, especialmente para listas de tamanho considerável. No entanto, é importante considerar outros fatores, como a facilidade de implementação e o espaço de memória necessário, ao escolher um algoritmo de ordenação para um determinado problema.

Figura 8: Tabela comparativa

Algoritmo	**Tempo**		
	Melhor	Médio	Pior
Merge sort	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Figura 9: Comparando os algoritmos



Como podemos observar o merge-sort junto do quick-sort no caso medio são os algoritmos que melhor se comportam se ignorarmos o melhor caso do insertion-sort, observando a tabela conseguindo ver o motivo de tal resultado, o merge sort é constante independende do caso, já o insertion em seu caso médio apresenta pior desempenho. Lembrando que os tempos estão em milisegundos.

Referências

https://pt.wikipedia.org/wiki/Merge_sort

https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao