Universidade Federal do Rio Grande do Norte

Centro de Ensino Superior do Seridó – CERES

Departamento de Computação e Tecnologia – DCT

Bacharelado em Sistemas de Informação – BSI

Estrutura de Dados - Relatório 1

Flávio Glaydson Guimarães Lopes

Orientador: Prof. Dr. João Paulo de Souza Medeiros

Caicó-RN, 25 de Maio de 2023

Resumo

Este relatório tem como objetivo principal apresentar uma análise relacionada a tempos de execução de alguns algoritmos de ordenação, usando como base os seguintes algoritmos: Selection-Sort, Insertion-Sort, Merge-Sort, Quick-Sort e o Distribution-Sort. No relatório foram desenvolvidas três atividade: A primeira delas foi a geração de gráficos dos tempos de execução de cada algoritmo utilizando a ferramenta Gnuplot; A segunda atividade foi a realização da análise de forma analítica dos cinco algoritmos; por fim, foram realizadas comparações de desempenho de tempo entre os algoritmos citados.

Abstract

The main objective of this report is to present an analysis related to the running times of some sorting algorithms, based on the following algorithms: Selection-Sort, Insertion-Sort, Merge-Sort, Quick-Sort and Distribution-Sort. Three activities were developed in the report: The first was the generation of graphs of the execution times of each algorithm using the Gnuplot tool; The second activity was to carry out the analytical analysis of the five algorithms; finally, time performance comparisons were made between the aforementioned algorithms.

Sumário

1. Introdução

O problema de ordenação é considerado um dos mais básicos e mais estudados em computação, ele consiste em, dada uma lista de elementos, ordená-los de acordo com a ordem estabelecida, por exemplo, dado um vetor $A = (a1, a2, \ldots an)$ com n números, obter uma permutação desses números $(a1, a2, \ldots an)$ de modo que $a1 \leq a2 \leq \ldots \leq an$. Desse modo, todos os elementos à esquerda de um certo elemento são menores ou iguais a ele e todos à direita são maiores ou iguais a ele.

Diante disso, existem vários algoritmos de ordenação que são muito úteis na computação, cada um com suas peculiaridades e métodos diferentes de resolver um mesmo problema. Neste relatório serão abordados cinco algoritmos: Selection-Sort, Insertion-Sort, Merge-Sort, Quick-Sort e Distribution-Sort.

2. Selection-Sort

O algoritmo Selection-Sort têm como base a ordenação por seleção, que é baseado no princípio de selecionar o menor (ou maior) elemento no vetor não ordenado e colocá-lo na posição correta, isso é feito comparando os elementos e realizando trocas de posições, se necessário, para garantir que o elemento mínimo (ou máximo) seja colocado na posição correta. Esse processo é repetido para os n-1 elementos restantes, até que os últimos dois elementos estejam em suas posições corretas. Ao final, toda a lista estará ordenada.

2.1. Gráficos dos tempos de execução

2.1.1. Caso base

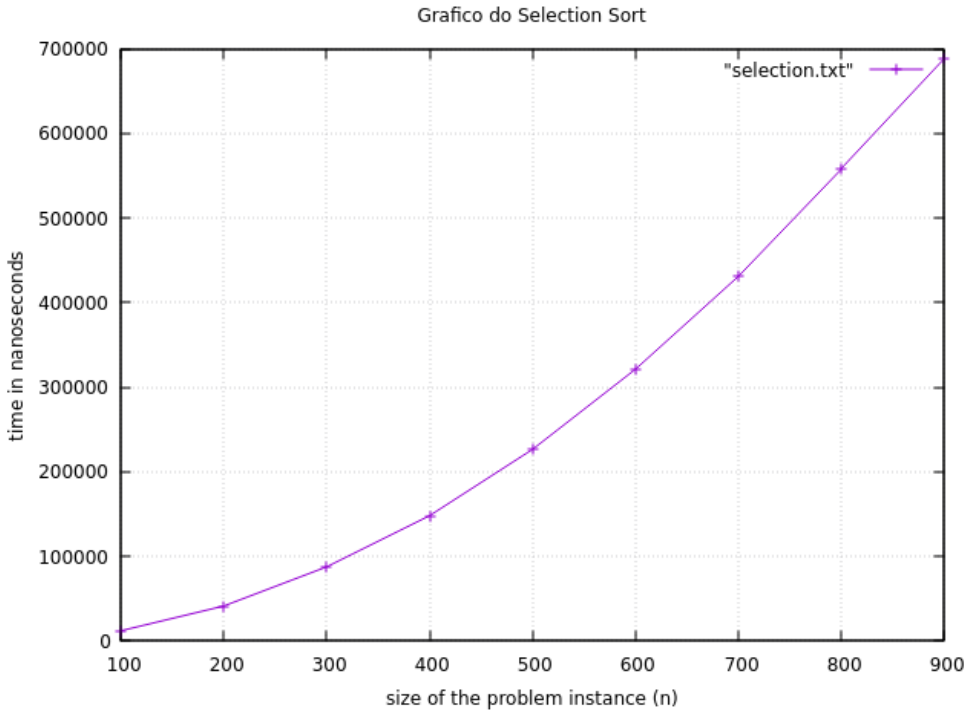


Figura 2.1: Selection-Sort: Caso base

O Selection Sort possui uma complexidade de tempo de $O(n^2)$, possuindo uma complexidade de tempo quadrática, onde "n" é o número de elementos na lista. Isso o torna menos eficiente do que algoritmos de ordenação mais avançados, entretanto, para listas pequenas, ele pode ser uma opção adequada e fácil de implementar.

2.2. Análise análitica

2.2.1. Caso base

Selection Sort : Caso base

$$T(n) = C_2(n) + C_3(n-1) + C_4 \sum_{i=2}^{n} i + C_5\, 6 \sum_{i=1}^{n-1} i + C_7(n-1)$$

$$T(n) = C_2(n) + C_3(n-1) + C_4 \frac{n}{2}(n+1) - 1 + C_5\, 6 \frac{n}{2}(n+1) \ldots$$

$$\ldots - n + C_7(n-1)$$

$$T(n) = (C_2 + C_3 + C_4)n - C_3 - C_7 + C_4\left(\frac{n^2+n-2}{2}\right) \ldots$$

$$\ldots C_5\, 6\left(\frac{n^2-n}{2}\right)$$

3. Insertion-Sort

O algoritmo Insertion-Sort tem como base a ordenação por inserção, ou seja, percorre uma lista de elementos e, a cada iteração, insere o elemento atual em sua posição correta dentro da porção já ordenada da lista. Nesse sentido, em cada iteração do algoritmo, selecionamos o próximo elemento não ordenado e o inserimos na posição correta dentro da porção já ordenada à esquerda. Para encontrar a posição correta, comparamos o elemento atual com os elementos da porção ordenada,

3.1. Gráficos dos tempos de execução

3.1.1. Melhor caso

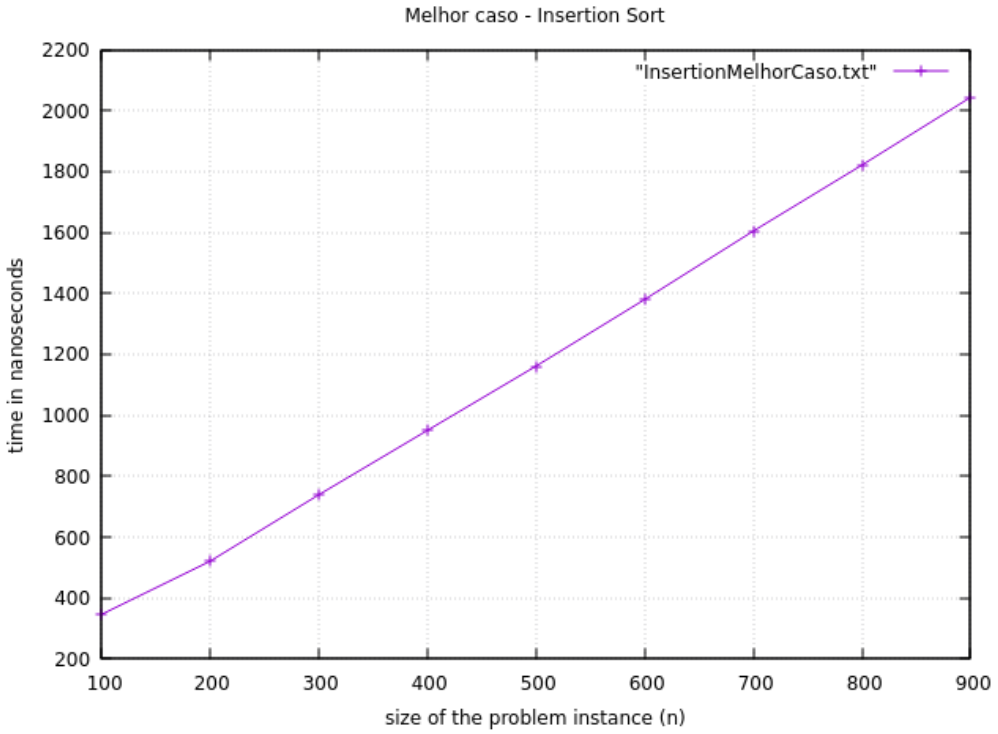


Figura 3.1: Insertion-Sort: Melhor caso

O melhor caso do Insertion-Sort é quando ele recebe um vetor já ordenado. Isso ocorre, pois, todos os elementos do vetor já estão em suas devidas posições, não sendo necessário mover nenhum elemento da sua posição inicial. Como a inserção é realizada n vezes, o custo total é linear, ou $O(n)$. É possível observar isso, analisando o gráfico (**Figura 3.1**).

3.1.2. Caso médio

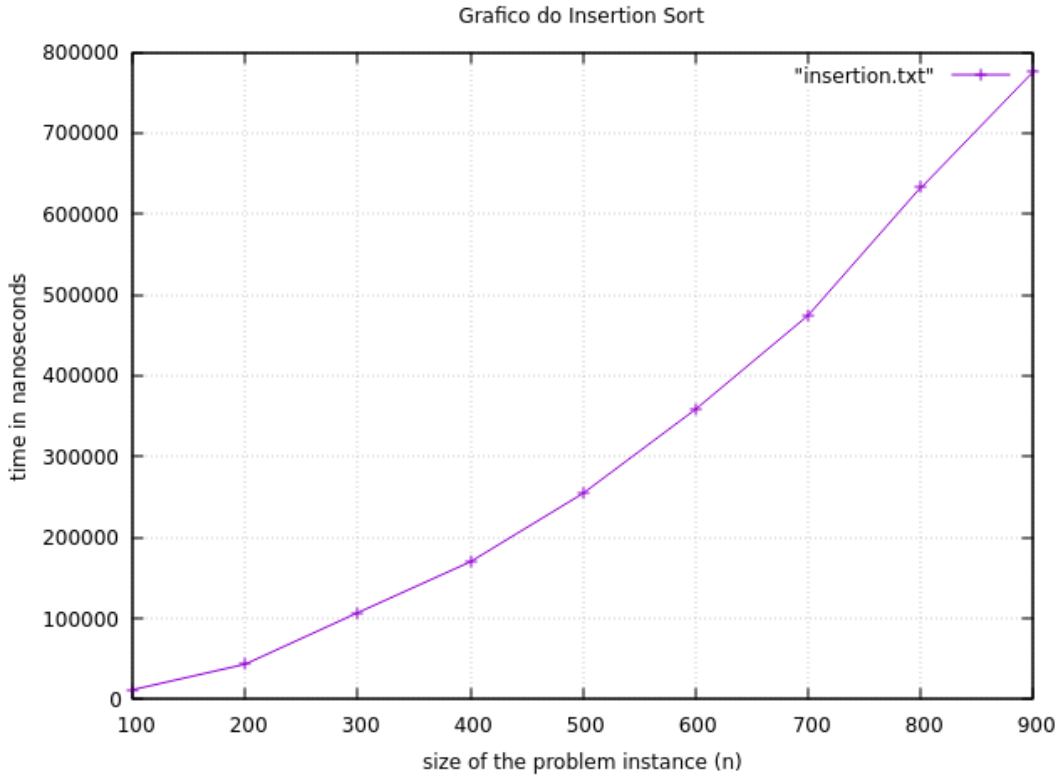


Figura 3.2: Insertion-Sort: Caso médio

O caso médio do insertion acontece quando é fornecido um vetor desordenado, "embaralhado". Nesse caso, o algoritmo terá um tempo de execução médio entre todas as entradas. Ao observar o gráfico (Figura 3.2) é possível notar que o custo total será $O(n^2)$, ou seja, será quadrática, pois o tempo de execução aumenta quadraticamente, principalmente nas últimas execuções, à medida em que a quantidade de elementos no vetor vai aumentando.

3.2.3. Pior caso

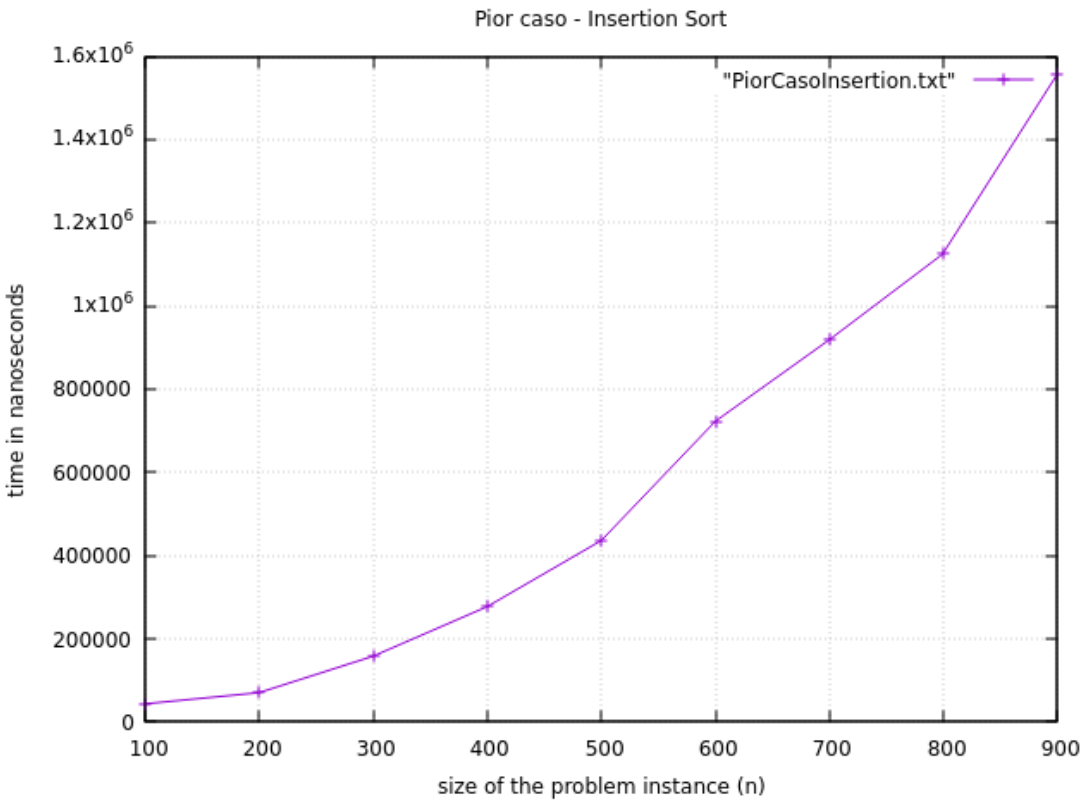


Figura 3.3: Insertion-Sort: Pior caso

O pior caso do Insertion-Sort acontece quando ele recebe um vetor ordenado em ordem inversa, isso ocorre, pois toda nova tentativa de inserção ordenada percorre todo o vetor à esquerda, trocando os elementos até encaixar o atual na primeira posição. Nesse sentido, o laço interno realizará a quantidade máxima de interações, logo o custo total, assim como no caso médio será quadrático, ou $O(n^2)$. Ao observar o gráfico (Figura 3.3) é possível notar isso acontecendo.

3.2.4. Comparação

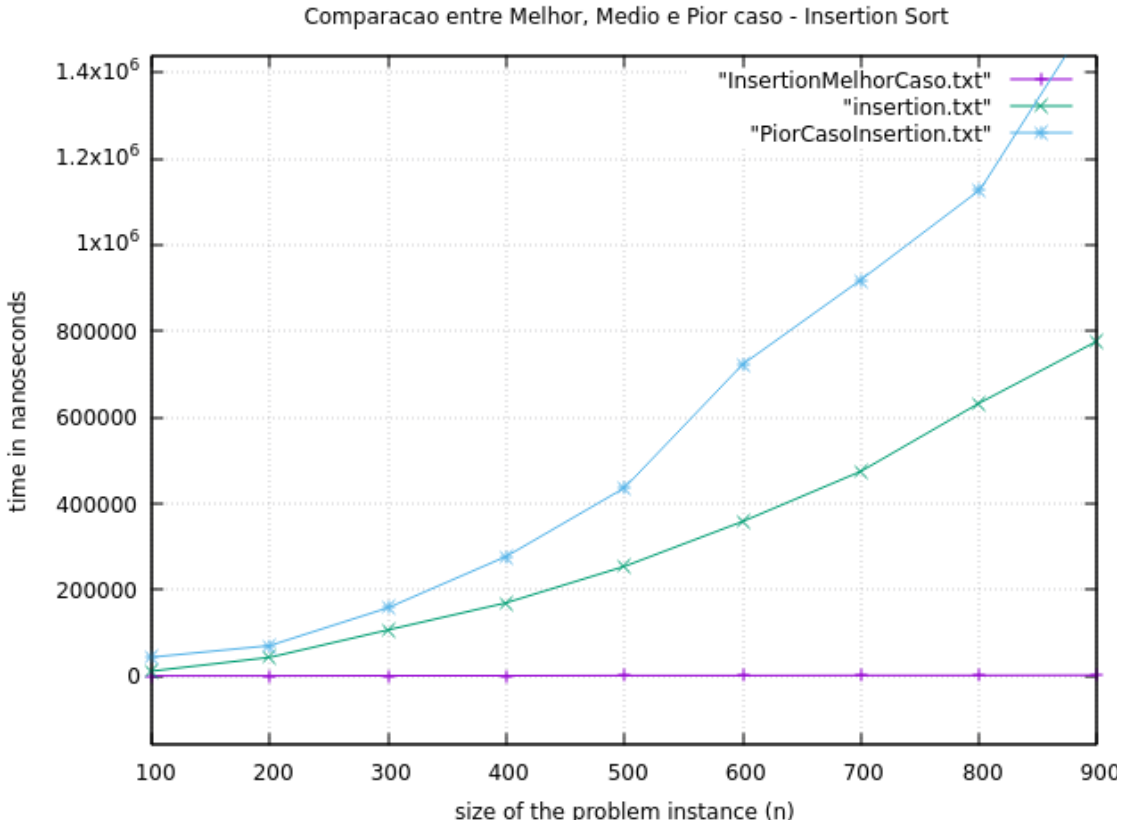


Figura 3.4: Comparação entre os casos do Insertion-Sort

Neste gráfico (Figura 3.4) é demonstrado uma comparação entre todas as complexidades do Insertion-Sort. Ao analisá-lo, é possível notar a diferença de tempos de execução de cada um dos casos, onde o pior caso possua tempo de execução tão lento que o melhor caso na comparação chega a parecer uma constante.

3.2. Análise análitica

3.2.1. Melhor caso

Insertion - Sort : Melhor caso

$$T_b(n) = C_1 + C_2 + C_3 + n(C_4) + (n-1)\cdot(C_5 + C_6 + C_7 + C_{10})$$

$$Tb(n) = \underbrace{n(C_4 + C_5 + C_6 + C_7 + C_{10})}_{a} + \underbrace{(C_1 + C_2 + C_3 - C_5 - C_6 - C_7 - C_{10})}_{b}$$

$$Tb(n) = an + b$$

3.2.2. Pior caso

Insertion - Sort : Pior caso

$$T_w(n) = C_1 + C_2 + C_3 + n(C_4) + (n-1)\cdot(C_5 + C_6 + C_{10} + Q_7(n) + Q_8(n) + Q_9(n))$$

$$Q_7(n) = C_7(2 + 3 + 4 + \dots + n)$$

$$Q_{89}(n) = (C_8 + C_9)\cdot(1 + 2 + 3 + \dots + n - 1)$$

} Somatório

$$C_7 \sum_{i=2}^{n} i = \sum_{i=1}^{n} i - 1$$

$$= \frac{n}{2}(n+1) - 1$$

$$(C_8 + C_9) \sum_{i=1}^{n-1} i = \sum_{i=1}^{n} i - n \rightarrow \frac{n}{2}(n+1) - n$$

$$\frac{n^2 - n}{2} \rightarrow \frac{n^2 + n - 2n}{2}$$

$$\frac{n}{2}(n-1)$$

$$T_w(n) = C_1 + C_2 + C_3 + (n-1)\cdot(C_5 + C_6 + C_{10}) + C_7\left[\frac{n}{2}(n+1) - 1\right] + \dots$$

$$\dots + (C_8 + C_9)\cdot\left[\frac{n}{2}(n-1)\right]$$

$$T_w(n) = n^2\left[\frac{C_7}{2} + \frac{C_8}{2} + \frac{C_9}{2}\right] + n\left(C_5 + C_6 + C_{10} + \frac{C_7}{2} - \frac{C_8}{2} - \frac{C_9}{2}\right) + \dots$$

$$\dots + \underbrace{(C_1 + C_2 + C_3 - C_5 - C_6 - C_7 + C_{10})}_{c}$$

$$T_w(n) = an^2 + bn + c$$

4. Merge-Sort

O Merge-Sort é um algoritmo bastante eficiente, pois tem como base a ordenação por divisão e conquista, mais precisamente a intercalação. É um método que age em três etapas, que seria dividir o problema em pedaços menores (subproblemas), conquista cada pedaço resolvendo-os recursivamente e depois combinar (merge) as soluções dos subproblemas em uma solução para o problema original.

Uma particularidade importante do Merge-Sort é que sua eficiência é a mesma ($n\ logn$) para o pior, melhor e caso médio, ou seja, independente da disposição dos dados em um vetor, a ordenação será eficiente.

4.1. Gráficos dos tempos de execução

4.1.1. Caso base

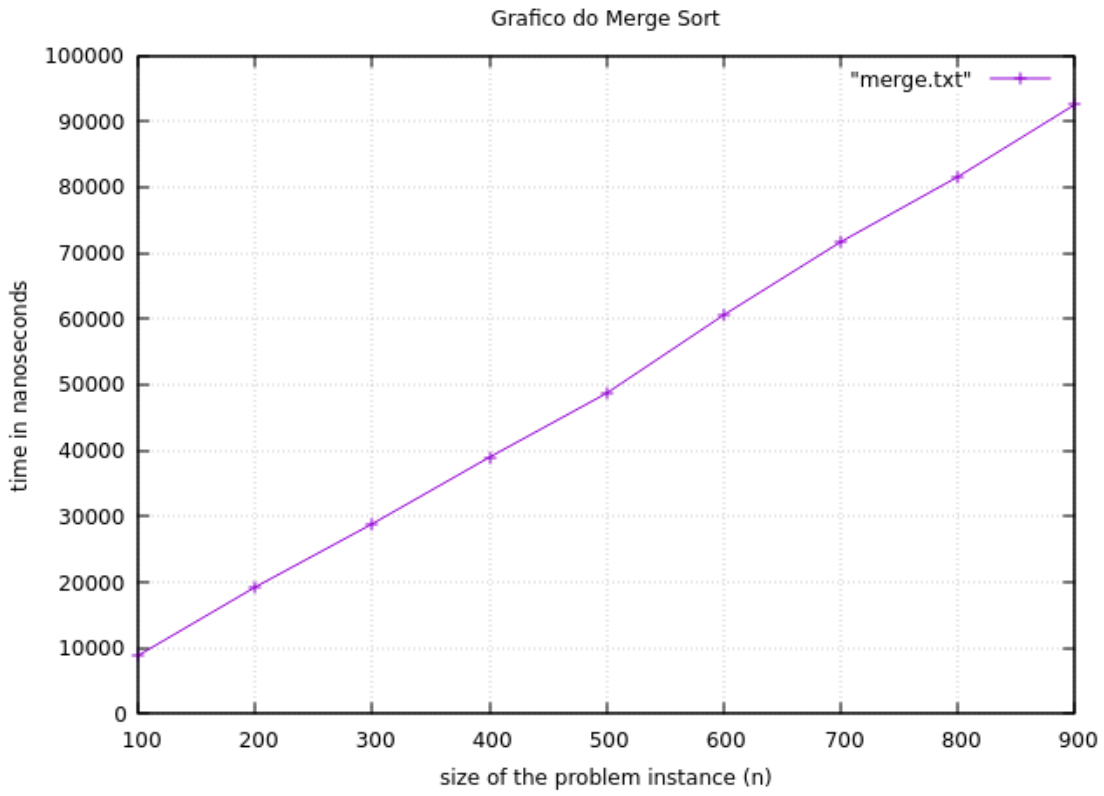


Figura 4.1: Merge-Sort: Caso base

Ao observar o gráfico (Figura 4.1) é possível notar que o caso base será linear, ou $O(n)$, pois o tempo de execução aumenta linearmente à medida que a quantidade de elementos no vetor vai aumentando. Portanto, o merge-sort é um algoritmo de ordenação eficiente, com uma complexidade de tempo $O(n\ log\ n)$ e uma complexidade de espaço $O(n)$.

4.2. Análise análitica

4.2.1. Caso base

Merge Sort : Caso base

$$T(n) = (C_1 + C_2).-(1) + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + C_5.-(n)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2}\right) + \frac{n}{2}$$

$$T(n) = 2\left[2T\left(\frac{n}{9}\right) + \frac{n}{2}\right] + n$$

$$T(n) = 4T\left(\frac{n}{9}\right) + 2n$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$$

$$T(n) = 4\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$$

$$T(n) = 8T\left(\frac{n}{8}\right) + 3n$$

$$T(n) = 2^x T\left(\frac{n}{2^x}\right) + x \cdot n$$

$$T(n) = 2^{\log_2 n} T(1) + \log_2 n \cdot n$$

$$T(n) = n \log_2 n + n_x$$

Caso base : $T(1)$, log :

$\frac{n}{2^x} = 1$

$2^x = n$

$\log_{2^2} x = \log_2 n$

$x \log_{2^2} 2^1 = \log_2 n$

$x = \log_2 n$

5. Quick-Sort

O Quick-Sort, é um algoritmo recursivo e eficiente de ordenação por divisão e conquista, porém, ele não é estável. O funcionamento do Quick-sort baseia-se em uma rotina fundamentada cujo nome é particionamento. Particionar significa escolher um número qualquer presente no vetor, chamado de pivô, e colocá-lo em uma posição em que todos elementos à esquerda são menores ou iguais e todos elementos a direita são maiores, isso cria uma espécie de "fronteira" em torno do pivô.

5.1. Gráficos dos tempos de execução

5.1.1. Caso médio

Observação: O algoritmo Quicksort possui melhor, pior e caso médio, entretanto não conseguir desenvolver o código e gráfico do melhor e pior caso, diante disso, será mostrado nessa seção de gráficos - 5.1, apenas o gráfico do caso médio. Ficando o melhor e pior caso para a seção de análise analítica - 5.2.

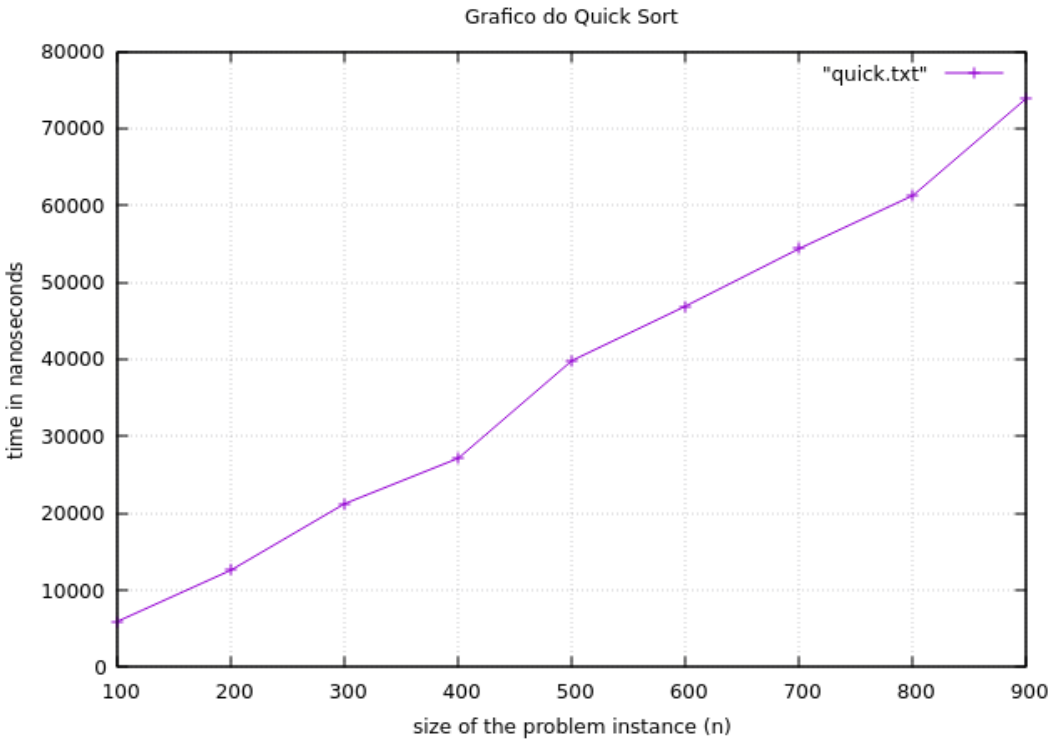


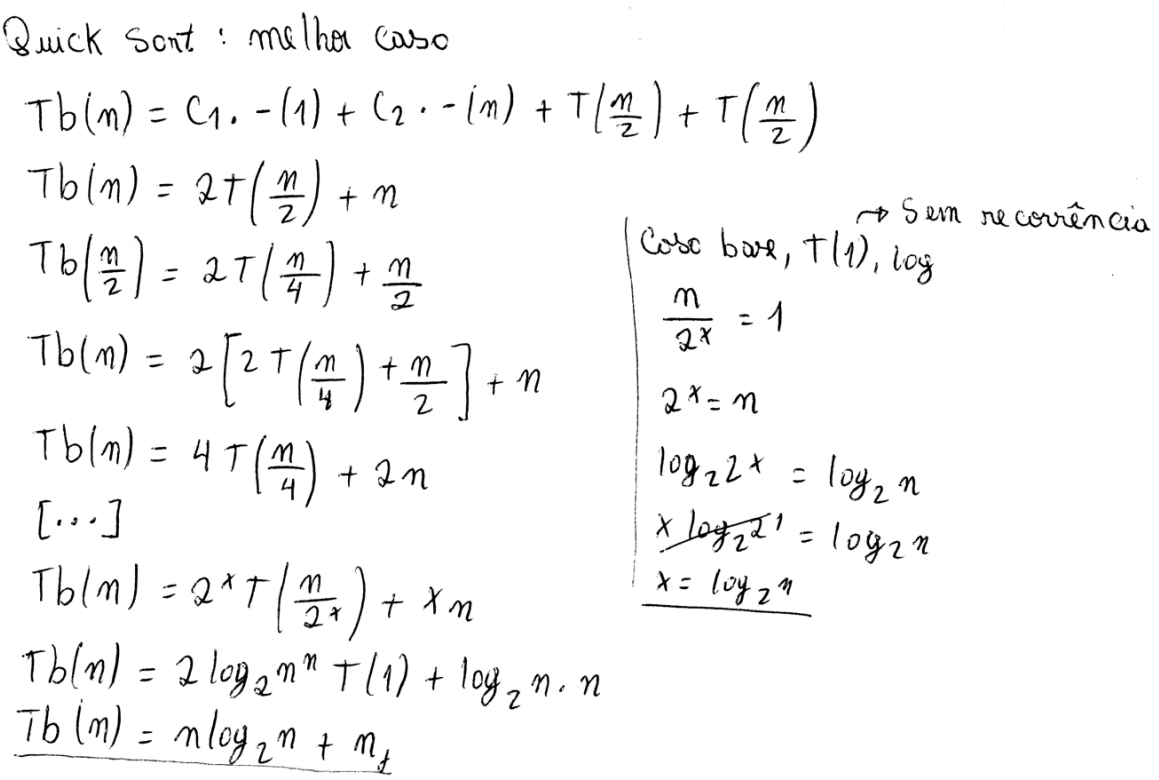
Figura 5.1: Caso médio - Quick-Sort

O caso médio do Quick-Sort acontece quando ele recebe um vetor “embaralhado”. Dessa forma, o algoritmo possuíra um tempo de execução médio entre todas as entradas. Ao observar o gráfico (Figura 5.1), é possível notar que o custo do caso médio é $O(n\ logn)$, pois, na medida em que a quantidade de elementos no vetor vai aumentando, o tempo de execução também aumenta.

É importante ressaltar que o caso médio do Quick-Sort é altamente provável de ocorrer. Mesmo que haja alternância entre particionamentos bons e ruins, o algoritmo ainda terá uma complexidade de tempo de O(n log n). Isso significa que, na prática, o Quick-Sort geralmente terá um desempenho eficiente, mesmo com alguns particionamento menos favoráveis. Isso se deve à baixa probabilidade de escolher pivôs extremamente desfavoráveis em um vetor embaralhado. Em resumo, o Quick-Sort é eficiente na maioria dos casos e pode lidar com variações no particionamento sem comprometer muito seu desempenho geral.

5.2. Análise analítica

5.2.1. Melhor caso



O melhor caso do Quick-Sort depende do seu particionamento, que seria quando o pivô sempre estivesse no meio do vetor. Dessa forma, obteremos algo parecido com a busca binária, ou melhor, uma árvore binária, na recursão em que a direita tem metade do tamanho do array a direita também tem a metade do tamanho. Como é possível observar no cálculo, o tempo de execução é $O(n\,logn)$.

5.2.2. Pior caso

Quick-Sort : Pior Caso

$$T_w(n) = c_1 + c_2 \cdot n + T(1) + T(n-1)$$
$$T_w(n) = c_1 + c_2 \cdot n + T(n-1)$$
$$T_w(n-1) = c_1 + c_2 \cdot (n-1) + T(n-2)$$
$$T_w(n) = c_1 + c_2 \cdot n + [c_1 + c_2(n-1) + T(n-2)]$$
$$T_w(n) = 2c_1 + c_2 \cdot (n + (n-1)) + T(n-2)$$
$$[\ldots]$$
$$T_w(n) = x c_1 + c_2 \cdot (n \ldots + (n-x-1)) + T(n-x)$$
$$T_w(n) = (n-1) \cdot c_1 + c_2(n + (n-1)) + \ldots + (n-(n-1)-1) + T(1)$$
$$T_w(n) = (n-1) \cdot c_1 + c_2(n + (n-1) + \ldots + 0)$$
$$c_2 \cdot (n + (n-1) + \ldots + 2 + 1 + 0), \quad c_2 \cdot 0 = 0$$
$$n+1 \; ; \; (n-1) + 2 = n + 1 \rightarrow \frac{n}{2}(n+1)$$
$$T_w(n) = (n-1) \cdot c_1 + c_2\left(\frac{n}{2}(n+1)\right)$$
$$T_w(n) = n^2\left(\frac{c_2}{2}\right) + n\left(c_1 + \frac{c_2}{2}\right) + \underbrace{(-c_1)}_{c}$$
$$T_w(n) = an^2 + bn + c$$

O pior caso do Quick-Sort acontece quando ele recebe um vetor já ordenado, seja em ordem crescente ou decrescente, pois o pivô sempre vai dividir o vetor em duas porções de tamanho 0 e n-1. Ao observar o cálculo acima, é possível notar que o custo de execução é $O(n^2)$, ou quadrático.

6. Distribution-Sort

O algoritmo Distribution Sort tem como base a ordenação sem comparação, ou seja, ele não compara os elementos entre si para determinar sua ordem. Em vez disso, ele explora o conhecimento prévio sobre o intervalo de valores possíveis no vetor de entrada. Ele conta quantas vezes cada valor ocorre e, em seguida, distribui os elementos em posições corretas no vetor ordenado. Esse processo é divido em três etapas: contagem, acumulação e ordenação.

6.1. Gráficos dos tempos de execução

6.1.1. Caso base

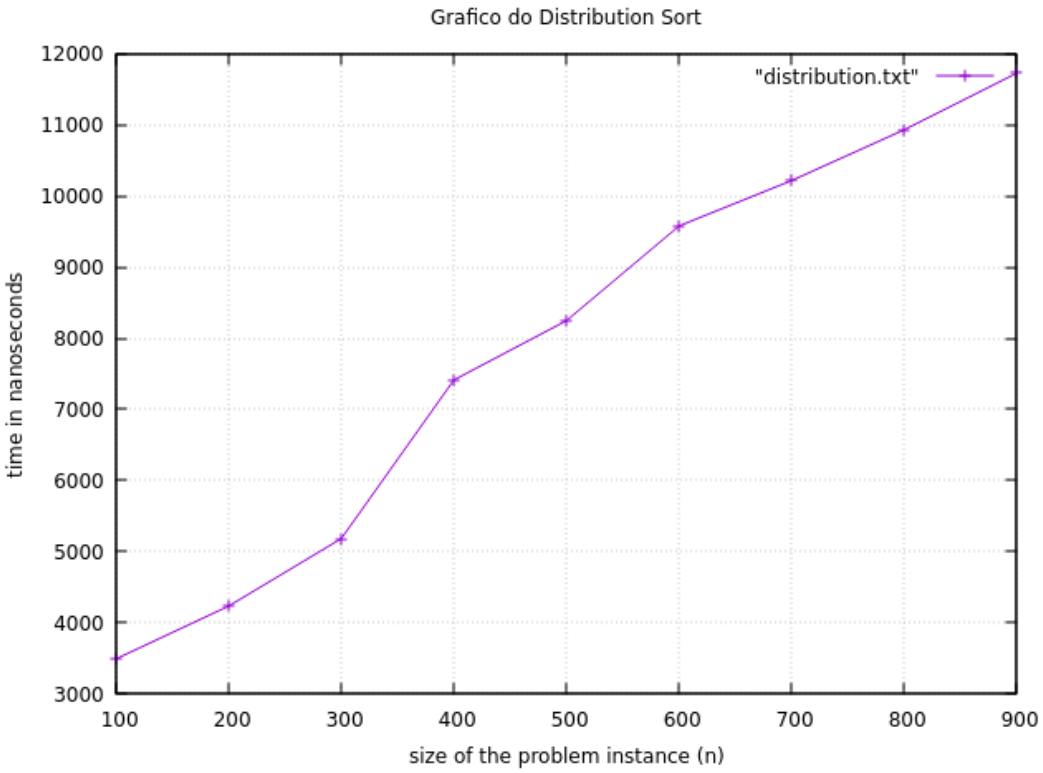


Figura 6.1: Distribution-Sort: Caso base

Ao observar o gráfico (Figura 6.1) é possível notar que o caso base será linear, ou $O(n)$ Nesse sentido, o tempo de execução do algoritmo cresce de forma proporcional ao tamanho do vetor, sendo considerado um dos algoritmos de ordenação mais eficientes em termos de tempo

de execução, especialmente quando o intervalo de valores possíveis no vetor de entrada é relativamente pequeno.

7. Comparação entre os 5 algoritmos de ordenação

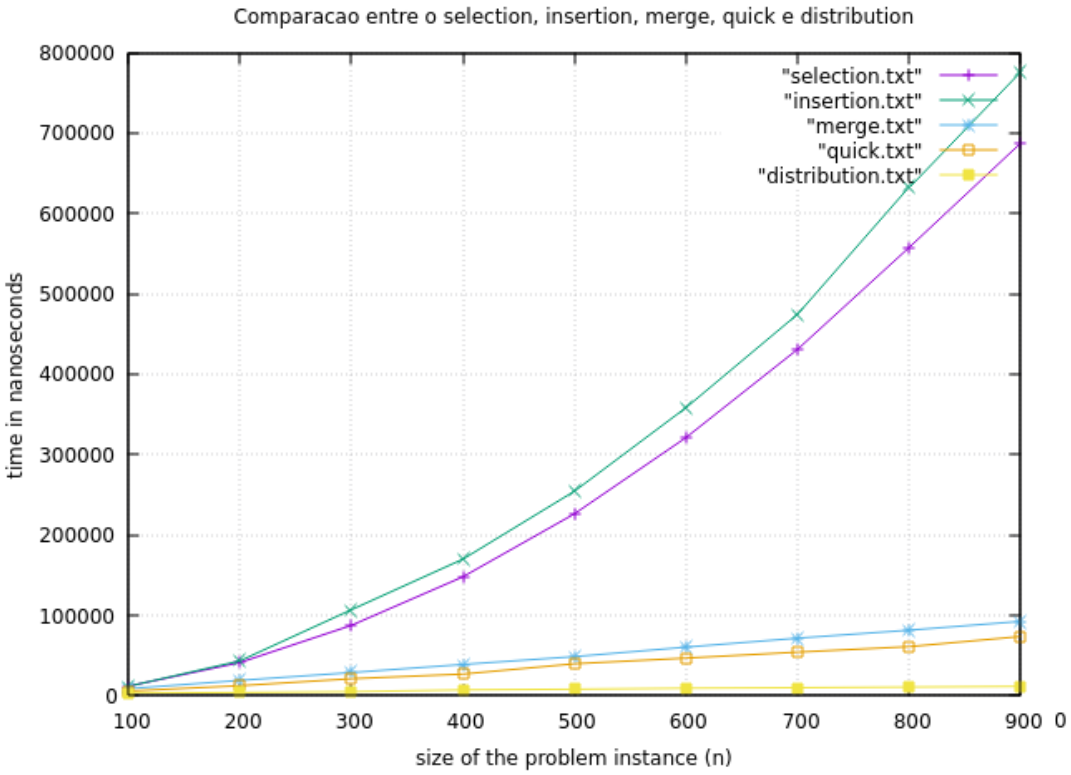


Figura 7.1: Comparação entre os 5 algoritmos de ordenação