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

Estudo dos Tempos de Execução de Algoritmos de Ordenação em Estruturas de Dados

Charles Eduardo Araújo de Faria

Orientador: Prof. Dr. João Paulo de Souza Medeiros

Relatório Técnico apresentado ao Curso de Bacharelado em Sistemas de Informação como parte dos requisitos para obtenção de nota na disciplina de Estrutura de Dados.

Caicó, RN, 22 de junho de 2023

Resumo

Este trabalho realiza uma análise dos algoritmos de ordenação mais comumente utilizados: selection-sort, insertion-sort, merge-sort, quick-sort e distribution-sort. O estudo inclui uma explicação sobre o funcionamento de cada algoritmo, bem como uma análise dos tempos de execução em diferentes cenários, como melhor caso, pior caso e caso médio. Além disso, são apresentados resultados de análises comparativas, utilizando tanto descrições textuais quanto representações gráficas, com o objetivo de fornecer uma visão das diferenças de desempenho entre esses algoritmos de ordenação.

Palavras-chave: Algoritmos de ordenação; Tempo de execução; Desempenho.

Abstract

This work conducts an analysis of the most commonly used sorting algorithms: selection sort, insertion sort, merge sort, quick sort, and distribution sort. The study includes an explanation of the functioning of each algorithm, as well as an analysis of their execution times in different scenarios, such as best case, worst case, and average case. Additionally, comparative analysis results are presented, using both textual descriptions and graphical representations, to provide an overview of the performance differences among these sorting algorithms.

Keywords: Sorting algorithms; Runtime; Performance.

Sumário

Lista de Algoritmos

Lista de Figuras

Lista de Tabelas

1. Introdução

Os algoritmos de ordenação realizam tarefas fundamentais na ciência da computação, sendo cruciais para a resolução de diversos problemas computacionais. Muitos algoritmos foram desenvolvidos ao longo dos anos com o objetivo de melhorar o desempenho desses métodos de ordenação. Nesse contexto, os principais algoritmos são: selection sort, insertion sort, merge sort, quick sort e distribution sort.

Ambos apresentam um objetivo em comum: ordenar, geralmente de maneira crescente, valores contidos em vetores de diferentes tamanhos. Contudo, cada um desses algoritmos apresenta suas devidas particularidades nos quesitos de tempo de execução, usabilidade e estratégia de implementação.

Este trabalho tem como principal objetivo realizar uma análise desses algoritmos de ordenação, levando em consideração, principalmente, seus respectivos tempos de execução, assim como comparar os diferentes cenários de cada uma das tarefas de ordenação.

1.1 Metodologia

Para o desenvolvimento deste trabalho, foi necessário o estudo dos algoritmos de ordenação, a fim de desenvolver códigos equivalentes na linguagem C.

Na execução de cada um dos códigos dos algoritmos de ordenação, foi utilizado um script, criado pelo professor João Paulo, que realiza a gravação dos tempos de execução de cada algoritmo para determinados tamanhos de vetores.

Tendo os dados de tempo de execução de cada algoritmo, torna-se possível utilizar a aplicação Gnuplot para montarmos gráficos que demonstram de maneira visual como os algoritmos se desempenham em relação ao tempo para diferentes tamanhos de vetores. Conhecendo os algoritmos e obtendo os gráficos referentes aos seus tempos de execução, torna-se viável realizar uma análise dos procedimentos de ordenação de forma mais clara.

Os códigos utilizados para gerar os resultados deste trabalho foram primeiramente testados na plataforma online Replit e executados para obtenção de dados no Windows Subsystem For Linux.

1.2 Organização do trabalho

O trabalho está organizado em um capítulo dividido em subcapítulos. Cada um dos subcapítulos será nomeado de acordo com o algoritmo de ordenação analisado. Os subcapítulos contam com seções que abordam o algoritmo, o código em C, a representação assintótica, a análise analítica e a amostragem do gráfico do tempo de execução em função do tamanho dos vetores.

2. Desenvolvimento

2.1 Selection-Sort

O selection-sort é o algoritmo de ordenação mais simples e intuitivo que será tratado neste trabalho. Este algoritmo percorre o vetor a ser ordenado, compara a posição atual com a posição seguinte e organiza os valores menores a esquerda do vetor e os maiores a direita, através de troca de posições.

2.1.1 Algoritmo do Selection-Sort

Algoritmo 2.1 (Selection-Sort)**.** Ordenação por seleção

```
algorithm selection-sort(v, n)
1   for i from 1 to (n − 1) do
2       m ← i
3       for j from (i + 1) to n do
4           if v[m] > v[j] then
5               m ← j
6       swap(v[m], v[i])
```
□

2.1.2 Código em C

Segue na figura 2.1 o código em C equivalente do algoritmo selection-sort

2.1.3 Análise de casos do Selection-Sort

Na implementação do selection-sort, ocorre uma verificação de todas as posições do vetor, o que torna seu tempo de execução parecido para diversos casos de vetores com mesmo tamanho, mas com diferentes organizaçázões de valores armazenados.

Devido a isso, o selection-sort não tem um melhor ou pior caso, já que, independentemente de como estão organizados os valores dentro do vetor, o algoritmo verificará posição por posição várias vezes. A única diferenciação é que a troca de valores pode ocorrer mais ou menos vezes, porém esta operação apresenta tempo constante, desconsiderável para aferir a complexidade do algoritmo.

As várias verificações tornam o algoritmo ineficiente para vetores de tamanhos maiores

2.1.4 Calculando o tempo de execução

A tabela 2.1 mostra as combinações de ‘i’ e ‘j’ para iterações específicas do laço externo (linha 1) e do laço interno (linha 3) do algoritmo 2.1. Cada linha da tabela representa uma

```c
void swap(int *a, int *b)
{
    int m;
    m = *a;
    *a = *b;
    *b = m;
}

void selection_sort(int *v, unsigned int n)
{
    unsigned int i, j, min;

    for (i = 0; i < (n - 1); i++)
    {
        min = i;
        for (j = (i + 1); j < n; j++)
        {
            if (v[j] < v[min])
                min = j;
        }
        swap(&v[i], &v[min]);
    }
}
```

Figura 2.1: Exemplo de apresentação de código do selection-sort em C

i	j	Linha 1	Linha 3
1	2	n	n - 1
2	3	n - 1	n - 2
3	4	n - 2	n - 3
4	5	n - 3	n - 4

Tabela 2.1: Tempo de execução em etapas do Selection-Sort

iteração do laço externo, onde o valor de 'i' é fixo, e as colunas representam as combinações de 'j' correspondentes para essa iteração específica.

A relação entre 'i' e 'j' pode ser explicada da seguinte forma:

- Na primeira iteração do laço externo, o valor de 'i' é 1. A partir disso, 'j' varia de 2 até n. Portanto, a primeira linha da tabela mostra a combinação de 'i = 1' com 'j' variando de 2 a n.

- Na segunda iteração do laço externo, o valor de 'i' é 2. Nesse caso, 'j' varia de 3 até n. Portanto, a segunda linha da tabela mostra a combinação de 'i = 2' com 'j' variando de 3 a n.

- Nas iterações subsequentes do laço externo, o valor de 'i' aumenta em 1 a cada iteração. E, conforme 'i' aumenta, o valor inicial de 'j' aumenta em 1 e o valor final de 'j' diminui em 1. Isso ocorre porque, em cada iteração do laço externo, já foram realizadas comparações e trocas anteriores, e a parte final do vetor já está parcialmente ordenada.

Portanto, para calcular o tempo de execução, basta somarmos as iterações da linha 1 e linha 3.

$$\text{linha}_1 = \sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1 = \frac{n}{2} \cdot (n+1) - 1$$

$$\text{linha}_3 = \sum_{i=1}^{n-1} i = \left(\sum_{i=1}^{n} i\right) - n = \frac{n}{2} \cdot (n+1) - n$$

Ao fim, realizando as operações de multiplicação, iremos encontrar um termo quadrático n^2.

2.1.5 Análise Assintótica

Todos os casos	O(n²)

Tabela 2.2: Análise Assintótica do Selection-Sort

2.1.6 Gráfico

Como podemos perceber, o gráfico na figura 2.2 tem forma de curva, característica de funções quadráticas.

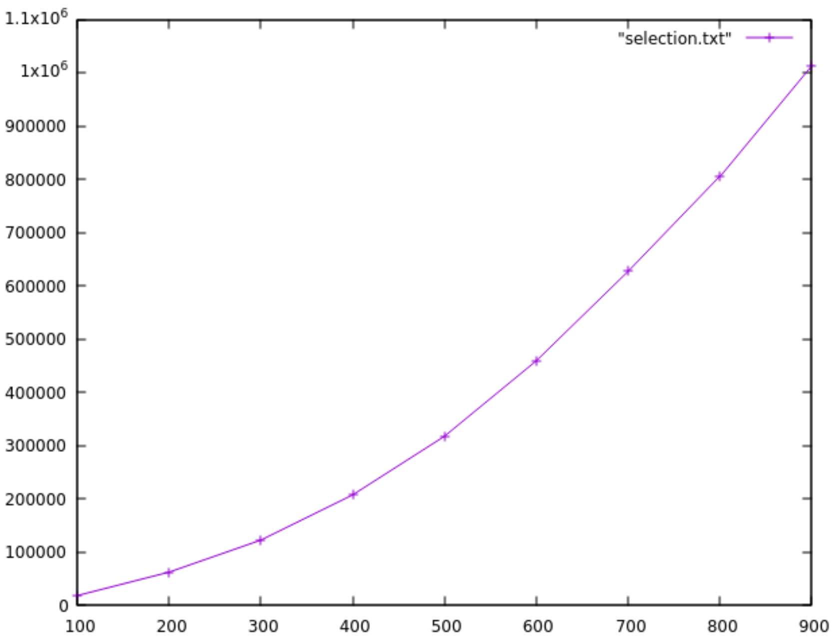

Figura 2.2: Gráfico de desempenho do selection-sort

2.2 Insertion-Sort

O insertion-sort é um algoritmo um pouco mais complexo em relação ao selection-sort, porém simples comparado aos próximos que serão tratados mais a frente. A ordenação por inserçao utiliza a estratégia de percorrer o vetor e inserir os elementos nas posições corretas até que o vetor esteja completamente ordenado.

Inicialmente, a primeira posição do vetor é considerada "ordenada". O algoritmo percorre o vetor, começando do segundo elemento, compara o elemento atual com os elementos anteriores, desloca os elementos maiores para a direita até encontrar a posição correta para inserir o elemento atual. O elemento atual é então inserido na posição correta.

2.2.1 Algoritmo do Insertion-Sort

Algoritmo 2.2 (Insertion-Sort)**.** Ordenação por inserção

algorithm insertion-sort(v, n)
for e from 2 to n do
    $i \leftarrow e$
    while $i > 1$ and $v[i-1] < v[i]$ do
        swap$(v[i-1], v[i])$
        $i \leftarrow i - 1$ □

2.2.2 Código em C

Segue na Figura 2.3 o código em C equivalente do algoritmo insertion-sort. Nota-se que há uma diferença entre este código e o algoritmo 2.2, pois no algoritmo a verificação está começando da penúltima posição, enquanto no código começa da segunda posição.

```
void insertion_sort(int *v, unsigned int n)
{
    int i, j, k;

    for (i = 1; i < n; i++)
    {
        k = v[i];
        j = i - 1;

        while (j >= 0 && v[j] > k)
        {
            v[j + 1] = v[j];
            j--;
        }

        v[j + 1] = k;
    }
}
```

Figura 2.3: Exemplo de apresentação de código do selection-sort em C

i	**Linha 3**	**Linha 4**
2	2	1
3	3	2
4	4	3
$\vdots$	$\vdots$	$\vdots$
n	n	(n - 1)

Tabela 2.3: Tempo de execução em etapas do Insertion-Sort

2.2.3 Análise de casos do Insection-Sort

Melhor caso: O melhor caso ocorre quando o vetor já está completamente ordenado. Nesse caso, o algoritmo percorre o vetor uma vez para verificar se os elementos já estão ordenados, assim não necessitando realizar nenhuma troca. O tempo de execução do melhor caso do insertion sort é de ordem $O(n)$, onde 'n' é o número de elementos no vetor.

Pior caso: O pior caso ocorre quando o vetor está ordenado em ordem decrescente. Nesse caso, o algoritmo precisará realizar o maior número de comparações e trocas para ordenar o vetor. O tempo de execução do pior caso do insertion sort é de ordem $O(n^2)$, onde 'n' é o número de elementos no vetor.

Caso médio: O caso médio ocorre quando o vetor de entrada está em uma ordem aleatória, sem uma estrutura específica. O tempo de execução médio do insertion sort também é de ordem $O(n^2)$, mas, em média, é mais rápido do que o pior caso. No entanto, o caso médio do insertion sort ainda é menos eficiente do que outros algoritmos de ordenação mais avançados, como o quick-sort ou merge-sort.

2.2.4 Calculando o tempo de execução

Utilizando como base o algoritmo 2.2 e levando em consideração a tabela 2.3, podemos calcular os tempos de execução

Melhor caso

Como já dito antes, o melhor caso do insertion-sort ocorre quando o vetor já está ordenado.

$$\mathrm{T}_b(\mathrm{n}) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1)$$

$$\mathrm{T}_b(\mathrm{n}) = c_1 \cdot n + (c_2 + c_3) \cdot (n-1)$$

No melhor caso, temos que o tempo de execução é $O(n)$.

Pior caso

Novamente, tomando como base o algoritmo 2.2 e a tabela 2.3, vamos, de maneira analítica, comprovar o pior caso.

$$\mathrm{T}_w(\mathrm{n}) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot \sum_{i=2}^{n} i + (c_4 + c_5) \cdot \sum_{i=1}^{n-1} i$$

$$\mathrm{T}_w(\mathrm{n}) = c_1 \cdot n + c_2 \cdot (n-1) + c3 \cdot \left[\frac{n}{2} \cdot (n+1) - 1\right] + (c_4 + c_5) \cdot \left[\frac{n}{2} \cdot (n+1) - n\right]$$

Realizando as multiplicações, nos deparamos com um termo quadrático. Logo, comprovamos que o tempo de execução do pior caso do insertion-sort é $O(n^2)$.

2.2.5 Análise Assintótica

Melhor caso	O(n)
Pior caso	$O(n^2)$
Caso médio	$O(n^2)$

Tabela 2.4: Análise Assintótica do Insertion-Sort

2.2.6 Gráfico

Melhor caso

O gráfico na figura 2.4 comprova que, no melhor caso, o tempo de execução do insertion-sort é linear.

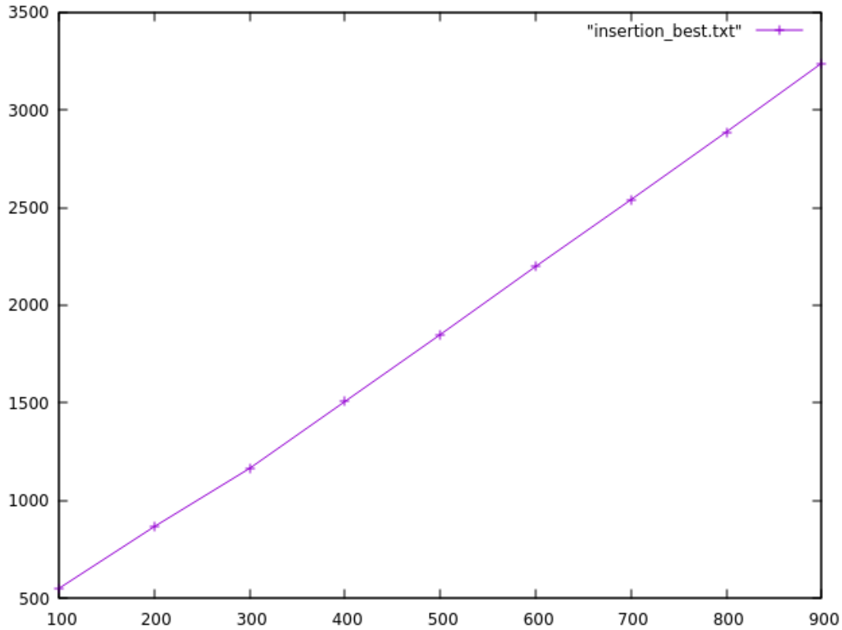

Figura 2.4: Gráfico de desempenho do insertion-sort no melhor caso

Caso médio

Percebemos pela Figura 2.5 que o gráfico tem o formato de curva, característico de funções de grau 2, comprovando assim, que o tempo de execução do insertion-sort no caso médio aproxima-se muito de $O(n^2)$.

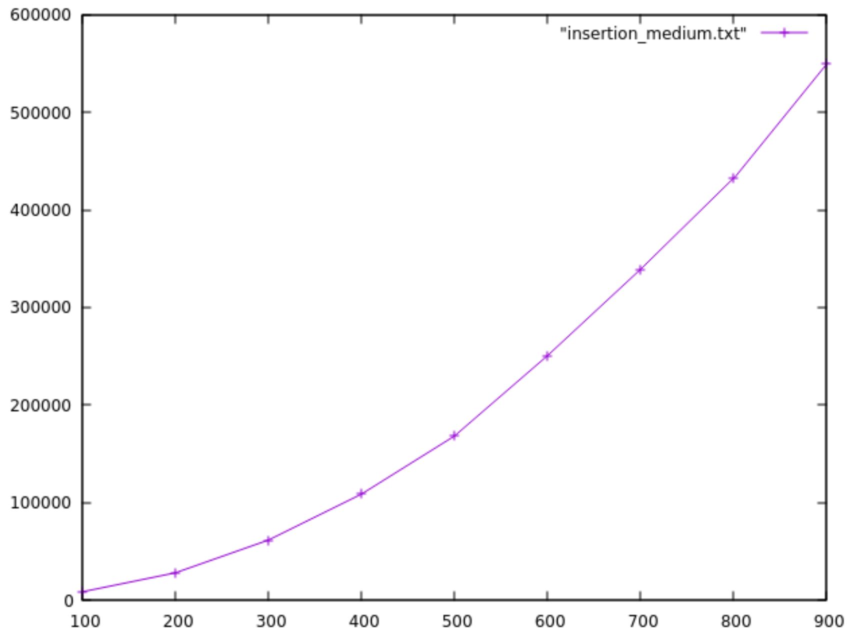

Figura 2.5: Gráfico de desempenho do insertion-sort no caso médio

Pior caso

No pior caso, o tempo de execução é quadrático, assim como foi mostrado anteriormente. Logo, percebemos que o caso médio do insertion-sort se aproxima do pior caso. Vejamos o gráfico na Figura 2.6.

2.2.7 Todos os casos

O gráfico 2.7 mostra uma comparação do desempenho de todos os casos do insertion-sort.

2.3 Merge-Sort

O merge-sort é um algoritmo de ordenação eficiente e baseado na técnica de "dividir para conquistar". Ele funciona dividindo o vetor original em sub-vetores menores, ordenando esses sub-vetores e, em seguida, mesclando-os para obter o vetor ordenado final. Estas são as etapas que representam o funcionamento do merge-sort:

- Divisão: O vetor não ordenado é dividido em duas metades aproximadamente iguais. Esse processo é repetido recursivamente até que cada subvetor tenha apenas um elemento, o que é considerado um vetor ordenado.

- Ordenação: Em seguida, começa o processo de mesclagem e ordenação dos subvetores. Os subvetores são combinados em pares e seus elementos são comparados em ordem. Os elementos são rearranjados para que fiquem ordenados.

- Mesclagem: Após a ordenação dos pares de subvetores, o processo de mesclagem continua para combinar os subvetores maiores. Os elementos dos subvetores são

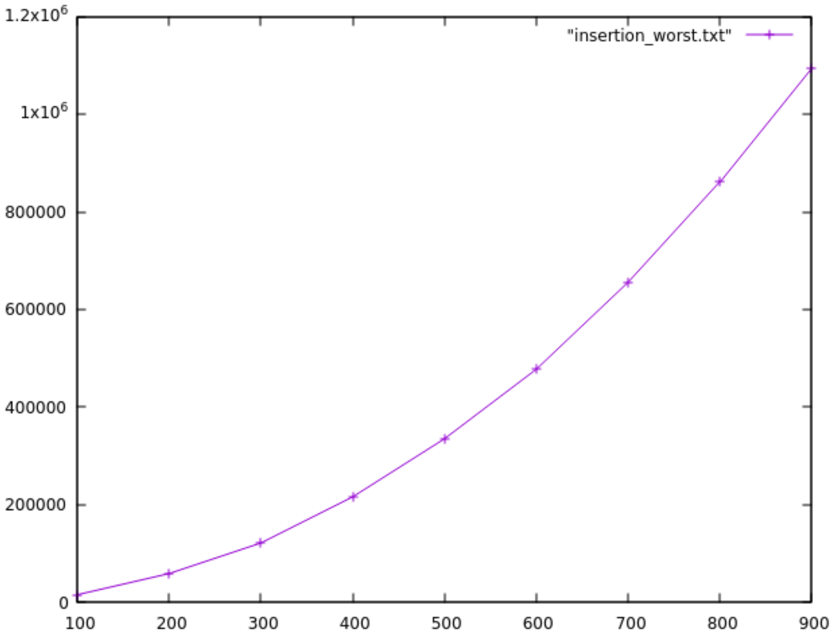

Figura 2.6: Gráfico de desempenho do insertion-sort no pior caso

comparados novamente em ordem e rearranjados em um único novo vetor ordenado. Esse processo é repetido até que todos os subvetores sejam mesclados em um único vetor ordenado no final.

2.3.1 Algoritmo do Merge-Sort

Algoritmo 2.3 (Merge-Sort)**.** Ordenação por mesclagem

```
algorithm merge-sort(v, n)
1    if s < e then
2        m ← ⌊(s + e)/2⌋
3        merge-sort(v, s, m)
4        merge-sort(v, m + 1, e)
5        merge(v, s, m, e)
algorithm merge(v, s, m, e)
1    p ← s
2    q ← m + 1
3    for i from 1 to (e − s + 1) do
4        if (q > e) or ((p ≤ m) and (v[p] < v[q])) then
5            w[i] ← v[p]
6            p ← p + 1
         else
7            w[i] ← v[q]
8            q ← q + 1
9    for i from 1 to (e − s + 1) do
10       v[s + i − 1] ← w[i]
```
□

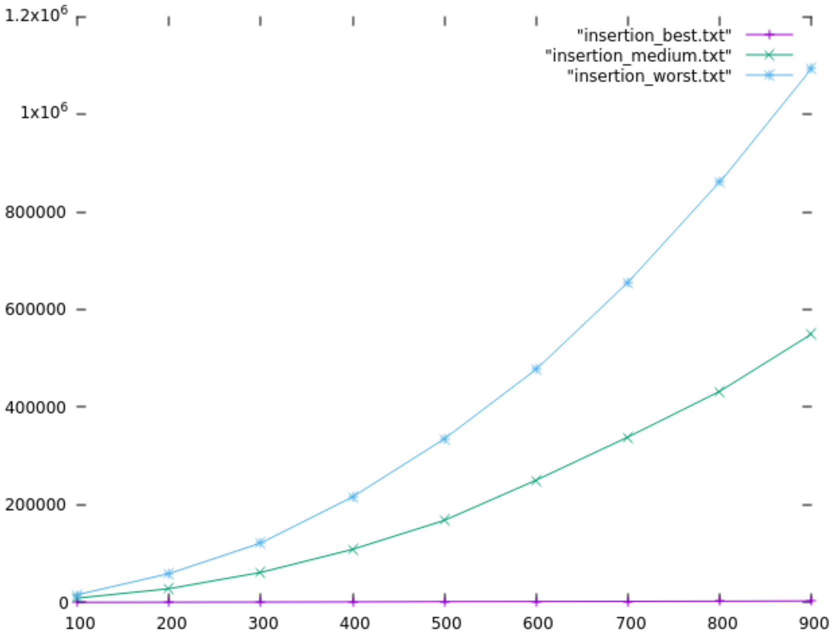

Figura 2.7: Comparação de todos os casos do insertion-sort

2.3.2 Código em C

Na Figura 2.8 temos um código equivalente na linguagem C.

2.3.3 Análise de casos do Merge-Sort

O merge-sort apresenta uma complexidade de tempo médio, melhor e pior caso de $O(n \cdot \log n)$. No entanto, há uma distinção entre o melhor, médio e pior caso em termos de uso de espaço.

Melhor caso: O melhor caso ocorre quando o vetor já está totalmente ordenado. Nesse caso, o merge-sort ainda divide o vetor em subvetores, mas as operações de mesclagem são desnecessárias, pois os subvetores já estão ordenados. Portanto, o melhor caso do merge-sort tem uma complexidade de tempo de $O(n \cdot \log n)$, assim como o caso médio e pior.

Caso médio: O caso médio do merge-sort ocorre quando o vetor está desordenado ou parcialmente ordenado. O algoritmo divide o vetor em subvetores e realiza as operações de mesclagem necessárias para obter o vetor completamente ordenado. O caso médio também tem uma complexidade de tempo de $O(n \cdot \log n)$.

Pior caso: O pior caso do merge-sort ocorre quando o vetor está em ordem inversa, ou seja, em ordem decrescente. Nesse caso, o algoritmo precisa realizar o máximo de operações de mesclagem para obter o vetor ordenado final. Embora ainda tenha uma complexidade de tempo de $O(n \cdot \log n)$, o pior caso do merge-sort pode ser um pouco mais lento em comparação com o caso médio, pois requer um número maior de comparações e rearranjos de elementos.

Em relação ao espaço, o merge-sort requer espaço adicional na memória para armazenar os subvetores durante o processo de mesclagem. Esse espaço adicional é proporcional ao tamanho do vetor original. Portanto, o merge-sort tem uma complexidade de espaço de

```
void merge(int *v, int n, int m)
{
    int i, j, k;
    int *aux = (int *)malloc(n * sizeof(int));

    for (i = 0, j = m, k = 0; k < n; k++)
    {
        if (j == n)
            aux[k] = v[i++];
        else if (i == m)
            aux[k] = v[j++];
        else if (v[j] < v[i])
            aux[k] = v[j++];
        else
            aux[k] = v[i++];
    }

    for (i = 0; i < n; i++)
        v[i] = aux[i];

    free(aux);
}

void merge_sort(int *v, int n)
{
    if (n < 2)
        return;

    int m = n / 2;

    merge_sort(v, m);
    merge_sort(v + m, n - m);
    merge(v, n, m);
}
```

Figura 2.8: Exemplo de apresentação de código do merge-sort em C

$O(n)$ em todos os casos (melhor, médio e pior), considerando o espaço adicional para os subvetores temporários.

2.3.4 Calculando o tempo de execução

Primeiramente, vamos calcular o tempo de execução da função merge. Em seguida, é possível calcular o tempo de execução do merge-sort.

Calculo do tempo de execução da função merge

$$\mathrm{T}_m(\mathrm{n}) = (c_3 + c_4 + c_{57} + c_9 + c_{10}) \cdot n + (c_1 + c_2 + c_3 + c_9)$$

Podemos reescrever como:

$$\mathrm{T}_m(\mathrm{n}) = a \cdot n + b$$

Calculo do tempo de execução da função merge-sort

Caso base (vetor com apenas uma posição):

$$\mathrm{T}(1) = c_1$$

Calculando recorrência T(n):

$$\mathrm{T(n)} = (c_1 + c_2 + c_3 + c_4 + c_5) + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + T_m(n)$$

$$\mathrm{T(n)} = d + 2 \cdot T\left(\frac{n}{2}\right) + T_m(n)$$

$$\mathrm{T}\left(\frac{n}{2}\right) = d + 2 \cdot T\left(\frac{n}{4}\right) + T_m\left(\frac{n}{2}\right)$$

$$\mathrm{T(n)} = d + 2 \cdot \left[d + 2 \cdot T\left(\frac{n}{4}\right) + T_m\left(\frac{n}{2}\right)\right] + T_m(n)$$

$$\mathrm{T(n)} = 3 \cdot d + 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot T_m\left(\frac{n}{2}\right) + T_m(n)$$

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

$$\mathrm{T(n)} = 3 \cdot d + 4 \cdot \left[d + 2 \cdot T\left(\frac{n}{8}\right) + T_m\left(\frac{n}{4}\right)\right] + 2 \cdot T_m\left(\frac{n}{2}\right) + T_m(n)$$

$$\mathrm{T(n)} = 7 \cdot d + 8 \cdot T\left(\frac{n}{8}\right) + 4 \cdot T_m\left(\frac{n}{4}\right) + 2 \cdot T_m\left(\frac{n}{2}\right) + T_m(n)$$

Podemos reescrever esta parte como um somatório:

$$T_m\left(\frac{n}{4}\right) + 2 \cdot T_m\left(\frac{n}{2}\right) + T_m(n)$$

Podemos perceber o padrão:

$$\mathrm{T(n)} = (x-1) \cdot d + x \cdot T\,(n-x)$$

$$\mathrm{T(n)} = (2^z - 1) + 2^z \cdot T\left(\frac{n}{2^z}\right) + \sum_{i=0}^{z-1} 2^i \cdot T_m\left(\frac{n}{2^i}\right)$$

Sabemos que:

$$\frac{n}{2^z} = 1 \quad \text{(Caso base)}$$

$$\mathrm{n} = 2^z$$

$$\log_2 n = \log_2 2^z$$

$$\log_2 n = z$$

Então podemos fazer algumas substituições:

$$\sum_{i=0}^{z-1} 2^i \cdot T_m\left(\frac{n}{2^i}\right) = \sum_{i=0}^{(\log_2 x)-1} 2^i \cdot T_m\left(\frac{n}{2^i}\right)$$

$$\mathrm{T(n)} = (2^{\log_2 n} - 1) \cdot d + 2^{\log_2 n} \cdot T\left(\frac{n}{2^{\log_2 n}}\right) + \sum_{i=0}^{(\log_2 n)-1} 2^i \cdot T_m\left(\frac{n}{2^i}\right)$$

$$\mathrm{T(n)} = (n-1) \cdot d + n \cdot T(1) + \sum_{i=0}^{(\log_2 n)-1} 2^i \cdot T_m\left(\frac{n}{2^i}\right)$$

Substituindo T_m:

$$\sum_{i=0}^{(\log_2 n)-1} 2^i \cdot T_m\left(\frac{n}{2^i}\right) = \sum_{i=0}^{(\log_2 n)-1} 2^i \cdot \left(a \cdot \frac{n}{2^i} + b\right) = \sum_{i=0}^{(\log_2 n)-1} 2^i \cdot \left(a \cdot n + b \cdot 2^i\right)$$

$$= \sum_{i=0}^{(\log_2 n)-1} a \cdot n + \sum_{i=0}^{(\log_2 n)-1} b \cdot 2^i = a \cdot n \cdot (\log_2 n) + b \cdot \sum_{i=0}^{(\log_2 n)-1} 2^i$$

$$= a \cdot n \cdot (\log_2 n) + b \cdot \left(2^{(\log_2 n)-1}\right)$$

Com isso, temos que:

$$\mathrm{T(n)} = (n-1) \cdot d + c_1 \cdot n + a \cdot n \cdot (\log_2 n) + b \cdot (n-1)$$

Logo, comprovamos que o tempo de execução do merge-sort é $O(n \cdot \log n)$.

2.3.5 Análise Assintótica

Todos os casos	$O(n \cdot \log n)$

Tabela 2.5: Análise Assintótica do merge-sort

2.3.6 Gráfico

Pelo gráfico na figura 2.9, podemos visualizar o desemepenho do algoritmo. Apesar de parecer um pouco linear, percebe-se que não se trata de uma linha reta. Isso se deve ao fato de que o tempo de execução não é linear, apresentando uma multiplicação de n com um termo logaritmico $log_2 n$.

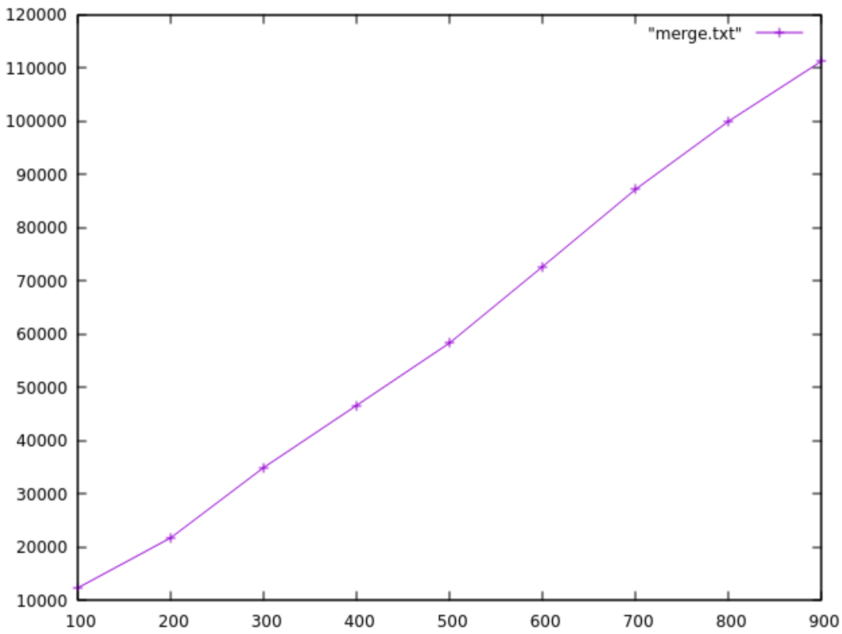

Figura 2.9: Gráfico de desempenho do merge-sort

2.4 Quick-Sort

2.4.1 Algoritmo do Quick-Sort

O quick-sort é um algoritmo de ordenação eficiente que, assim como o merge-sort, utiliza a estratégia de dividir para conquistar. Ele opera selecionando um elemento como pivô e particionando o vetor em dois subvetores, de forma que os elementos menores que o pivô fiquem à esquerda e os elementos maiores fiquem à direita. Esse processo é realizado recursivamente nos subvetores resultantes até que todo o vetor esteja ordenado.

Algoritmo 2.4 (Quick-Sort)**.** Ordenação rápida

algorithm quick-sort(v, s, e)
1 if $s < e$ then
2 $\quad p \leftarrow$ partition(v, s, e)
3 $\quad$ quick-sort$(v, s, p-1)$
4 $\quad$ quick-sort$(v, p+1, e)$

algorithm partition(v, s, e)
1 $d \leftarrow s - 1$
2 for i from s to $(e-1)$ do
3 $\quad$ if $v[i] \leq v[e]$ then
4 $\qquad d \leftarrow d + 1$
5 $\qquad$ swap$(v[d], v[i])$
6 swap$(v[d+1], v[e])$
7 return $(d+1)$ □

2.4.2 Código em C

Na figura 2.10 temos o código equivalente em C do quick-sort

```c
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int v[], int n) {
    int pivot = v[n];
    int i = -1;
    int j = 0;

    while (j < n) {
        if (v[j] < pivot) {
            i++;
            swap(&v[i], &v[j]);
        }
        j++;
    }

    swap(&v[i + 1], &v[n]);

    return i + 1;
}

void quick_sort(int v[], int n) {
    if (n < 2)
        return;

    int pivot = partition(v, n - 1);

    quick_sort(v, pivot);
    quick_sort(v + pivot + 1, n - pivot - 1);
}

```

Figura 2.10: Exemplo de apresentação de código do quick-sort em C

2.4.3 Análise de casos do Quick-Sort

Melhor caso: O melhor caso do quick-sort ocorre quando o pivô escolhido divide o vetor em duas partes aproximadamente iguais a cada iteração. Isso resulta em um tempo de execução mais eficiente. No melhor caso, a complexidade de tempo do quick-sort é $O(n \cdot \log n)$. Essa eficiência é semelhante à do merge-sort.

Caso médio: O caso médio do quick-sort ocorre quando o vetor não está totalmente ordenado e o pivô divide o vetor de forma equilibrada em duas partes. Nesse caso, o quick-sort também tem uma complexidade de tempo média de $O(n \cdot \log n)$.

Pior caso: O pior caso do quick-sort ocorre quando o pivô escolhido não divide o vetor de forma equilibrada e resulta em uma partição desigual. Isso acontece, por exemplo, quando o vetor já está ordenado em ordem crescente ou decrescente e o pivô é sempre escolhido como o primeiro ou o último elemento. No pior caso, a complexidade de tempo do quick-sort é quadrática, $\mathrm{O(n^2)}$.

2.4.4 Calculando o tempo de execução

Levando em consideração o pior caso:

$$T_w(0) = T_w(1) = c_1 \quad \text{(Caso base)}$$

$$T_p(n) = \text{Tempo de execução da função partition} = a \cdot n + b$$

$$T_w(n) = c_1 + c_2 + c_3 + c_4 + c_5 + T_p(n) + T_w(0) + T_w(n-1)$$

$$T_w(n) = d + T_p(n) + T_w(0) + T_w(n-1)$$

$$T_w(n-1) = d + T_p(n-1) + T_w(0) + T_w(n-2)$$

$$T_w(n) = d + T_p(n-1) + T_w(0) + [d + T_p(n-1) + T_w(0) + T_w(n-2)]$$

$$T_w(n) = 2 \cdot (d + T_w(0)) + T_p(n) + T_p(n-1) + T_w(n-2)$$

$$T_w(n-2) = d + T_p(n-2) + T_w(0) + T_w(n-3)$$

$$T_w(n) = 3 \cdot (d + T_w(0)) + T_p(n) + T_p(n-1) + T_p(n-2) + T_w(n-3)$$

Podemos perceber o seguinte padrão:

$$T_w(n) = x \cdot (d + T_w(0)) + T_w(n-x) + \sum_{i=0}^{x-1} T_p(n-i)$$

$$T_w(n) = x \cdot (d + T_w(0)) + T_w(n-x) + \sum_{i=0}^{x-1} a \cdot (n-i) + b$$

$$T_w(n) = x \cdot (d + T_w(0)) + T_w(n-x) + b \cdot x + a \cdot \sum_{i=0}^{x-1} a \cdot (n-i) + b$$

$$T_w(n) = x \cdot (d + T_w(0)) + T_w(n-x) + b \cdot x + a \cdot (n+1) \cdot \frac{n}{2}$$

Realizando as multiplicações, iremos encontrar um termo quadrático, que comprova o pior caso do quick-sort: $O(n^2)$.

2.4.5 Análise Assintótica

Na tabela 2.6, está a análise assinstótica dos três casos do quick-sort.

Melhor caso	$O(n \cdot logn)$
Pior caso	$O(n^2)$
Caso médio	$O(n \cdot logn)$

Tabela 2.6: Análise Assintótica do Quick-Sort

2.4.6 Gráfico

Caso médio

Assim como no merge-sort, teremos um gráfico parecido com o linear que sofre influência da multiplicação por um termo logarítimico. Na figura 2.11, podemos visualizar este gráfico.

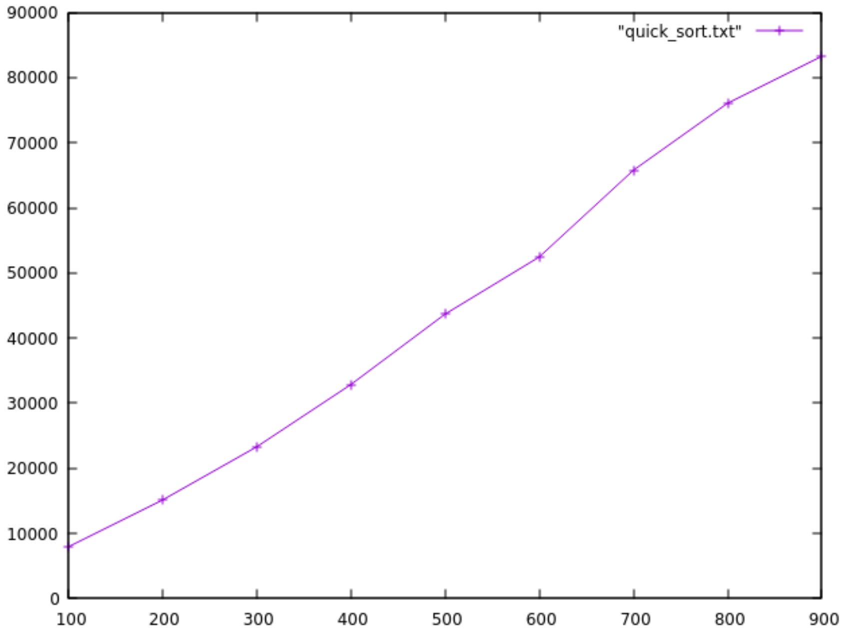

Figura 2.11: Gráfico de desempenho do quick-sort no caso médio

Pior caso

Podemos perceber a drástica diferença do tempo de execução do caso médio e pior caso apenas olhando para os valores atingidos no eixo y do gráfico. Além disso, o gráfico do pior caso comprova a ineficiência do quick-sort em uma situação no pior caso, tornando seu tempo de execução quadrático. Veja na figura 2.12 o gráfico do desempenho do quick-sort no pior caso.

Comparação entre o caso médio e pior caso

Pelo gráfico 2.13, pode-se visualizar a notável distância entre os tempos de execução do caso médio e pior caso do quick-sort.

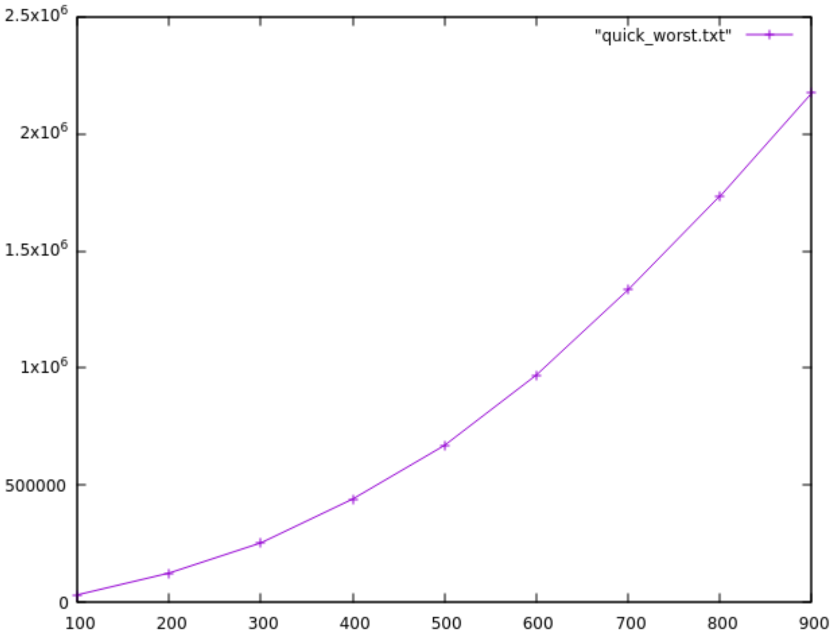

Figura 2.12: Gráfico de desempenho do quick-sort no pior caso

2.5 Distribution-Sort

2.5.1 Algoritmo do Distribution-Sort

O distribution-sort se refere a qualquer algoritmo de ordenação em que os dados são distribuídos a partir da entrada para outras estruturas intermidiárias, organizando os dados e os colocando na saída do processo.

Neste trabalho, foram feitos experimentos utilizando o distribution couting-sort.

O distribution counting-sort é um algoritmo de ordenação eficiente que é utilizado quando se conhece previamente o intervalo dos valores a serem ordenados e que não apresenta valores negativos. O algoritmo baseia-se na contagem das ocorrências de cada elemento e no uso dessas contagens para determinar a posição correta de cada elemento no vetor ordenado.

As etapas do distribution counting-sort são as seguintes:

- Contagem: O vetor de entrada é percorrido, contando o número de ocorrências de cada elemento. Para isso, criamos uma estrutura adicional no nosso código (geralmente outro vetor), onde cada índice representa um elemento e o valor armazenado no índice representa a contagem. Por isso, não podemos utilizar este método de ordenação para valores negativos, pois é impossível a existência de um índice negativo no vetor.

- Cálculo das posições: Com base no vetor de contagem, as posições corretas de cada elemento são aferidas. Podemos fazer isso assumindo que os índices do vetor de contagem representam as posições de cada elemento em um vetor ordenado.

- Construção da lista ordenada: Um vetor vazio do mesmo tamanho do vetor de entrada é criado. O vetor de entrada é percorrido novamente, adicionando cada

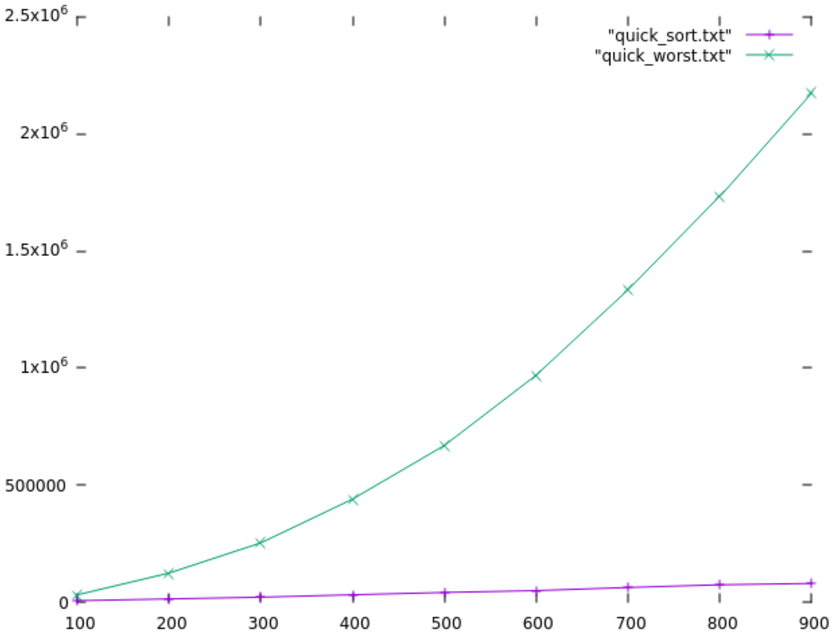

Figura 2.13: Gráfico comparativo do desempenho do quick-sort no caso médio e pior caso

elemento em sua posição correta com base no vetor de contagem e nas posições calculadas no passo anterior.

Algoritmo 2.5 (Distribution-Sort)**.** Ordenação por distribuição

algorithm distribution-sort(v, n)
- $s \leftarrow \min(v, n)$
- $b \leftarrow \max(v, n)$
- for i from 1 to $(b - s + 1)$ do
  - $c[i] \leftarrow 0$
- for i from 1 to n do
  - $c[v[i] - s + 1] \leftarrow c[v[i] - s + 1] + 1$
- for i from 2 to $(b - s + 1)$ do
  - $c[i] \leftarrow c[i] + c[i - 1]$
- for i from 1 to n do
  - $d \leftarrow v[i] - s + 1$
  - $w[c[d]] \leftarrow v[i]$
  - $c[d] \leftarrow c[d] - 1$
- for i from 1 to n do
  - $v[i] \leftarrow w[i]$ $\square$

2.5.2 Código em C

Na figura 2.14, vemos o código equivalente em C do counting distribution-sort.

2.5.3 Análise de casos do Distribution-Sort

O distribution counting-sort não tem um melhor ou pior caso específico. Isso ocorre porque o counting-sort possui uma complexidade de tempo linear, o que significa que seu

```c
void distribution_sort(int *v, int n) {
    int s = min(v, n);
    int b = max(v, n);

    int *c = (int *) malloc((b - s + 2) * sizeof(int));
    int *w = (int *) malloc(n * sizeof(int));

    int d, i;

    for (i = 0; i < b - s + 2; i++)
        c[i] = 0;

    for (i = 0; i < n; i++)
        c[v[i] - s + 1]++;

    for (i = 1; i < b - s + 2; i++)
        c[i] += c[i - 1];

    for (i = 0; i < n; i++) {
        d = v[i] - s + 1;
        w[c[d] - 1] = v[i];
        c[d]--;
    }

    for (i = 0; i < n; i++)
        v[i] = w[i];
}
```

Figura 2.14: Exemplo de apresentação de código do distribution counting-sort em C

tempo de execução depende principalmente do tamanho do vetor de entrada e do intervalo dos valores a serem ordenados.

A contagem das ocorrências tem complexidade de tempo $O(n)$, onde “n”é o número de elementos na lista de entrada. A construção do vetor ordenado também tem complexidade de tempo $O(n)$, pois envolve percorrer as contagens e colocar os elementos na posição correta.

2.5.4 Calculando o tempo de execução

Como vimos, o tempo de execução do distribution counting-sort depende não só do tamanho do vetor que, usualmente, chamamos de “n”, mas também do vetor auxiliar que utilizamos para fazer o processo de contagem onde cada posição do vetor representa um valor possível que os elementos de entrada podem ter. Vamos chamar este vetor auxiliar de “k”.

Lembrando também que as funções min e max apresentam tempo de execução linear.

$$T(n,k) = c_1 + T_{min}(n) + c_2 + T_{max}(n) + c_3 \cdot (k+1) + c_4 \cdot k + c_5 \cdot (n+1) + c_6 \cdot n + c_7 \cdot k +$$

$$c_8 \cdot (k-1) + c_9 \cdot (n+1) + c_{10} \cdot n + c_{11} \cdot n + c_{12} \cdot n + c_{13} \cdot (n+1) + c_{14} \cdot n$$

Podemos perceber então que se trata de um algoritmo de tempo de execução $O(n+k)$.

2.5.5 Análise Assintótica

Na tabela 2.7, temos a representação assintótica do tempo de execução do distribution counting-sort.

Todos os casos	$O(n+k)$

Tabela 2.7: Análise Assintótica do Distribution-Sort

2.5.6 Gráfico

Na figura 2.15, está o gráfico que representa o tempo de execução do distribution counting-sort. Como provado anteriormente, o gráfico se aproxima do de uma função linear.

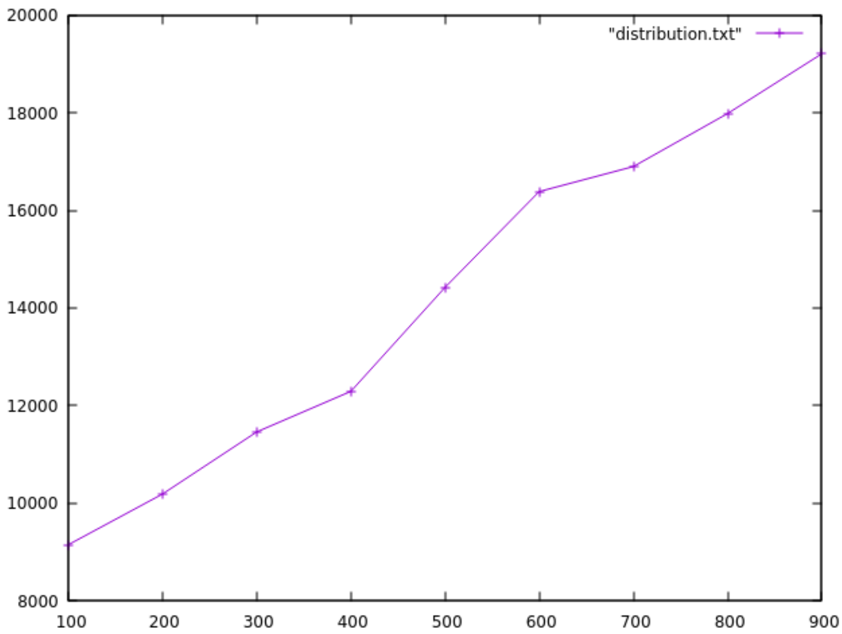

Figura 2.15: Gráfico de desempenho do distribution counting-sort

3. Conclusões

3.1 Resultados

Agora faremos uma análise dos resultados obtidos de cada algoritmo apresentado ao longo do trabalho. Para as comparações, será levado em consideração principalmente os casos médios de cada técnica de ordenação.

Pelos gráficos analisados, é nítido perceber que o selection-sort é o mais ineficiente entre todos apresentados. Sua implementação faz com que o algoritmo precise realizar muitas buscas e comparações, mesmo quando o vetor já contém elementos ordenados ou está completamente ordenado.

O insertion-sort, por sua vez, é superior ao selection sort, mas ao mesmo tempo não fica muito à frente. Se tivermos um vetor com alguns elementos já ordenados, o insertion-sort apresentará um tempo de execução linear, o que o torna significativamente melhor que o selection-sort. Contudo, seu caso médio se aproxima bastante do pior caso, que é um tempo de execução quadrático.

Procurando por um método de ordenação com o tempo de execução mais uniforme, podemos usar o merge sort. Sua complexidade é estimada em $O(n \cdot \log n)$ em todas as situações. O merge-sort é muito mais eficiente do que os outros dois mencionados anteriormente. No entanto, o merge sort possui uma peculiaridade em relação ao tempo de execução tradicionalmente analisado nos algoritmos. Ele cria um vetor auxiliar, responsável por receber os valores ordenados provenientes do processo de ordenação e mesclagem.

Visando evitar o consumo extra de memória proporcionado pelo merge-sort, a melhor alternativa apresentada neste trabalho é o quick-sort. Essa implementação é a mais comumente aplicada na ciência da computação. Sua eficiência no melhor caso e caso médio é bastante satisfatória, pois se aproximam muito de $O(n \cdot \log n)$ sem a necessidade de criar outra estrutura de dados durante seu funcionamento. Além disso, o pior caso do quick sort é uma situação pouco comum, o que nos convence ainda mais de suas vantagens.

O distribution counting-sort, também conhecido como counting-sort, possui um tempo de execução linear, o que é notável e uma vantagem desse algoritmo. No entanto, ele requer o conhecimento prévio do intervalo dos valores do vetor a ser ordenado, o que pode exigir um processamento adicional para determinar o maior e o menor valor. Além disso, o Counting-Sort não lida bem com intervalos muito grandes de valores e não permite que o vetor contenha valores negativos.

Se o vetor a ser ordenado não contiver números com grandes intervalos entre si e não tiver valores negativos, então a estratégia do counting-sort pode ser uma escolha eficiente. No entanto, se houver intervalos grandes de valores ou a presença de números negativos, pode ser necessário considerar outras opções de algoritmos de ordenação.

3.2 Ressalvas

Para a realização deste trabalho, os códigos em C foram testados no Replit e executados para a geração dos gráficos no Windows Subsystem For Linux. Como não foi utilizado um sistema Linux nativamente, os tempos obtidos podem parecer destoantes do ideal, caso fossem rodados num sistema Linux nativo. Mesmo assim, os resultados foram surpreendentementes positivos.

O distribution counting sort foi um dos que mais trouxe dificuldades. O código foi executado com valores completamente aleatórios. No entanto, a função $rand()$ do C gera valores muito grandes, o que estava resultando em *killed* ao executar o algoritmo. A solução paliativa foi limitar os valores dentro do vetor de 0 até 900. Vale lembrar que esse problema também pode ter sido causado pela limitação do algoritmo, que não lida bem com grandes intervalos de valores presentes no vetor.

3.3 Comparando os tempos de execução

Por fim, vamos visualizar por meio da tabela 3.1 e da figura 3.1 comparações entre os tempos de execução obtidos neste trabalho que justificam os resultados apresentados para cada algoritmo.

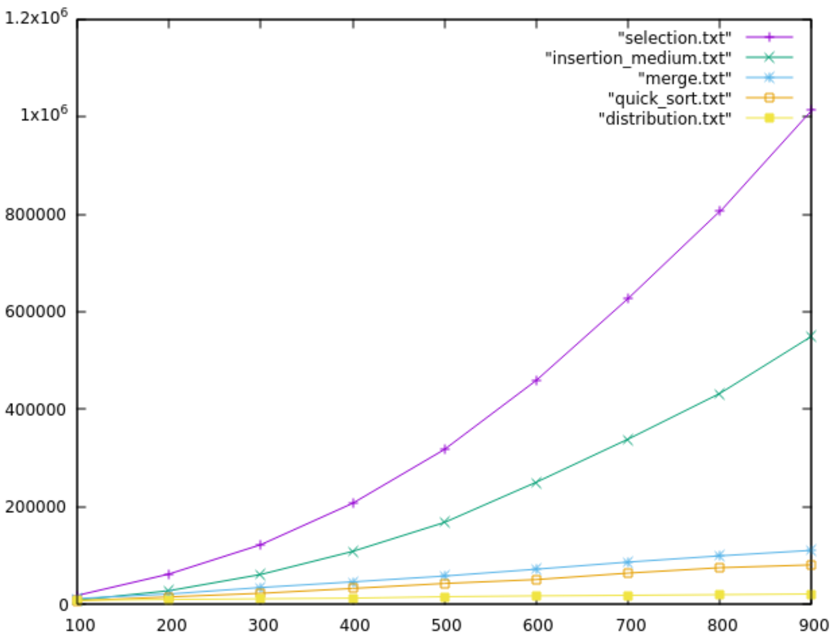

Figura 3.1: Gráfico comparativo do desempenho dos algoritmos testados

3.4 Considerações Finais

Neste trabalho, pudemos explorar os algoritmos de ordenação mais conhecidos. Conseguimos explicar seus devidos funcionamentos, calcular os tempos de execução de cada um em diversos cenários, além de registrar, de maneira gráfica, os tempos de execução dos algoritmos tratados neste documento.

n	Selection-Sort	Insertion-Sort	Merge-Sort	Quick-Sort	Distribution-Sort
100	18644	9034	12368	7994	8768
200	62676	28364	21788	15678	10472
300	122904	61894	34960	23214	11964
400	207572	108942	46644	33220	13358
500	317730	168434	58366	43350	16350
600	459430	250792	72700	51376	17974
700	627308	339076	87206	64596	18926
800	805874	432582	99970	75632	20292
900	1013932	549884	111250	81510	21838

Tabela 3.1: Tempos de execução em nanosegundos de cada algoritmo para diferentes tamanho de vetores