Universidade Federal de São Paulo – UNIFESP

Campus: Parque Tecnológico – São José dos Campos

Instituto de Ciência e Tecnologia – ICT

Bacharelado em Ciência e Tecnologia – BCT

Algoritmos e Estruturas de Dados II

Ordenação Externa

Prof. Reginaldo Massanobu Kuroshu

João Pedro da Silva Zampoli **RA: 168880**

Luiza de Souza Ferreira **RA: 170453**

Viviane Flor Park **RA: 169259**

São José dos Campos, 2024

SUMÁRIO

RESUMO

Neste relatório da UC “Algoritmos e Estruturas de Dados II”, o objeto de estudos é um código de ordenação externa em Python agindo sobre 3 entradas de tamanhos diferentes, tendo como objetivo comparar o desempenho do algoritmo aplicado sobre elas.

1 INTRODUÇÃO

Em um mundo onde o volume de dados gerados e processados está em constante crescimento, a eficiência na manipulação e ordenação desses dados tornou-se crucial. Quando o volume de dados excede a capacidade da memória principal (RAM), técnicas de ordenação especializadas são necessárias para garantir que a manipulação e análise dos dados sejam realizadas de maneira eficaz. Neste contexto, a ordenação externa surge como uma solução essencial.

A ordenação externa refere-se a métodos e algoritmos projetados para ordenar grandes conjuntos de dados que não cabem inteiramente na memória principal. Esses algoritmos são especialmente relevantes em sistemas de gerenciamento de bancos de dados e em aplicações que lidam com grandes volumes de dados, como processamento de dados em grandes servidores e sistemas de arquivos distribuídos.

O External Merge Sort é um dos algoritmos mais eficientes para ordenação externa, desenvolvido para superar as limitações das abordagens tradicionais de ordenação em memória. O algoritmo divide o conjunto de dados em blocos menores que podem ser processados na memória, ordena esses blocos individualmente e, em seguida, realiza uma fusão dos blocos ordenados para obter o conjunto de dados final ordenado. Esse processo é altamente eficiente, principalmente devido à sua capacidade de minimizar o número de operações de entrada e saída, que são geralmente o obstáculo em sistemas de ordenação externa.

Este relatório explora a aplicação dos princípios fundamentais da ordenação, em especial, a do External Merge Sort. Abordaremos a estrutura e o processo do algoritmo, e faremos uma análise do tempo de execução da leitura e escrita de registros para entradas de tamanhos diferentes. Ao final, a análise proporcionará uma compreensão clara das aplicações e benefícios desta técnica na ordenação de grandes volumes de dados.

2 OBJETIVOS

2.1 Objetivo geral

Avaliar o desempenho de um algoritmo de ordenação externa para 3 entradas distintas.

2.2 Objetivos específicos

- Calcular a quantidade de vezes que cada registro é lido e escrito no disco para cada arquivo de entrada;
- Comparar o tempo de execução do código usado para cada entrada;
- Descrever detalhes dos experimentos realizados e discutir sobre resultados observados.

3 MÉTODOS

Primeiramente, foi escolhida a linguagem de programação de tipagem dinâmica Python para o código de ordenação externa. A escolha foi direcionada pela impressão que os integrantes do grupo têm de que o tratamento de arquivos nessa linguagem é mais simples.

O código segue a lógica de um External Merge Sort. Para isso, foram seguidos os passos 1 a 5, descritos abaixo:

1) Lemos 5000 registros do arquivo de entrada e ordenamos esse bloco utilizando HeapSort;
2) Escrevemos registros ordenados em um arquivo intermediário;
3) Repetimos 1) e 2) até que todos os registros do arquivo de entrada tenham sido lidos e gravados;
4) Intercalamos os arquivos intermediários dois a dois com HeapSort, guardando os registros ordenados em um novo arquivo intermediário;
5) Repetimos 4) enquanto o passo gerar mais de um arquivo intermediário.

As Figuras 3.1 e 3.2 mostram o código implementado.

```python
import os

Leitura = 0
Escrita = 0
QtdNumeros = 5000
ParteDeExecucao = 1
TamanhoTotal = len(open('input2.txt').readlines())

def ContarLeitura():
    global Leitura
    Leitura += 1

def ContarEscrita():
    global Escrita
    Escrita += 1

def Heapify(Vet, Tamanho, Indice, min_heap=False):
    Extremo = Indice
    Esquerda = 2 * Indice + 1
    Direita = 2 * Indice + 2

    if Esquerda < Tamanho and ((not min_heap and Vet[Indice] < Vet[Esquerda]) or (min_heap and Vet[Indice] > Vet[Esquerda])):
        Extremo = Esquerda

    if Direita < Tamanho and ((not min_heap and Vet[Extremo] < Vet[Direita]) or (min_heap and Vet[Extremo] > Vet[Direita])):
        Extremo = Direita

    if Extremo != Indice:
        Vet[Indice], Vet[Extremo] = Vet[Extremo], Vet[Indice]
        Heapify(Vet, Tamanho, Extremo, min_heap)

def HeapSort(Vet):
    Tamanho = len(Vet)

    for i in range(Tamanho // 2 - 1, -1, -1):
        Heapify(Vet, Tamanho, i, min_heap=False)

    for i in range(Tamanho - 1, 0, -1):
        Vet[i], Vet[0] = Vet[0], Vet[i]
        Heapify(Vet, i, 0, min_heap=False)

def CriarArquivosIntermediarios():
    Pedacos = []
    with open('input3.txt', 'r') as f:
        Pedaco = []
        for Linha in f:
            ContarLeitura()
            Pedaco.append(int(Linha))
            if len(Pedaco) == QtdNumeros:
                HeapSort(Pedaco)
                Pedacos.append(Pedaco[:])
                Pedaco = []
        if Pedaco:
            HeapSort(Pedaco)
            Pedacos.append(Pedaco[:])
    return Pedacos

def EscreverArquivosIntermediarios(Pedacos):
    if not os.path.exists("Pedacos"):
        os.makedirs("Pedacos")
    for i in range(len(Pedacos)):
        with open(f'Pedacos/Parte0{ParteDeExecucao}Pedaco_0{i+1}.txt', 'w') as f:
            for Num in Pedacos[i]:
                ContarEscrita()
                f.write(f'{Num}\n')

```

Figura 3.1 - Primeira parte do código implementado para a ordenação externa.

```python
def CriarPartes():
    global ParteDeExecucao
    global QtdNumeros
    global TamanhoTotal

    Pedacos = CriarArquivosIntermediarios()
    EscreverArquivosIntermediarios(Pedacos)
    while len(Pedacos) > 1:
        ParteDeExecucao += 1
        QtdNumeros *= 2
        Pedacos = CriarArquivosIntermediarios()
        EscreverArquivosIntermediarios(Pedacos)
    return Pedacos

def CriarOutput():
    f = open('output3.txt', 'w')
    g = open(f'Pedacos/Parte0{ParteDeExecucao}Pedaco_01.txt', 'r')

    for Linha in g:
        f.write(Linha)
    f.close()
    g.close()

Pedacos = CriarPartes()
CriarOutput()

print("Tamanho Total: ", TamanhoTotal)

print(f'Leituras: {Leitura}')
print(f'Escritas: {Escrita}')

LeituraRegistro = Leitura / TamanhoTotal
EscritaRegistro = Escrita / TamanhoTotal

print(f'Quantidade de Leituras dos Registros: {ParteDeExecucao}')
print(f'Quantidade de Escritas dos Registros: {ParteDeExecucao}')

def ApagarPasta():
    try:
        if not os.path.exists("Pedacos"):
            raise ValueError("Pasta não existe")

        for item in os.listdir("Pedacos"):
            CaminhoItem = os.path.join("Pedacos", item)

            if os.path.isfile(CaminhoItem):
                os.remove(CaminhoItem)
                # print(f"Arquivo {item} apagado")
            else:
                raise ValueError("Item não é um arquivo")

    except Exception as e:
        print(f"Erro ao tentar remover: {e}")

    try:
        os.rmdir("Pedacos")
        #print("Pasta Pedacos removida")

    except Exception as e:
        print(f"Erro ao tentar remover: {e}")

ApagarPasta()
```

Figura 3.2 - Segunda parte do código implementado para a ordenação externa.

O código foi então executado para três entradas distintas, fornecidas junto ao enunciado do exercício. Para cada uma delas, o código foi executado dez vezes em uma terminal macOS, tendo os tempos de execução anotados para análise posterior. O código retorna, também, a quantidade de vezes que os registros foram lidos e escritos no disco.

4 RESULTADOS E DISCUSSÕES

4.1 Tempos de Execução

A Tabela 4.1.1 mostra os tempos de execução real, de usuário e de sistema registrados.

Tabela 4.1.1 - Tempos de execução do código para cada entrada e para cada execução, em segundos.

	input 1			input 2			input 3		
	real	usuário	sistema	real	usuário	sistema	real	usuário	sistema
1	0.391	0.304	0.028	9.189	7.784	0.338	150.445	129.856	4.765
2	0.355	0.296	0.023	9.027	7.711	0.302	150.849	130.296	4.690
3	0.351	0.292	0.024	9.665	7.867	0.315	150.614	130.122	4.771
4	0.350	0.291	0.024	8.914	7.690	0.302	152.292	130.300	4.789
5	0.361	0.294	0.025	8.934	7.708	0.304	149.901	129.641	4.667
6	0.358	0.298	0.024	9.111	7.771	0.306	151.324	129.827	4.671
7	0.358	0.294	0.025	8.865	7.699	0.294	151.033	130.613	4.815
8	0.980	0.328	0.029	8.986	7.762	0.301	153.103	130.732	4.828
9	0.394	0.304	0.028	8.936	7.698	0.297	157.701	134.210	4.829
10	0.481	0.307	0.030	8.747	7.617	0.287	150.927	129.407	4.652

A partir dos dados de tempo de execução real foram construídos os gráficos das Figuras 4.1.1, 4.1.2 e 4.1.3 por meio do RStudio.

Figura 4.1.1 - Gráfico de linha para o tempo real de execução do input 1.

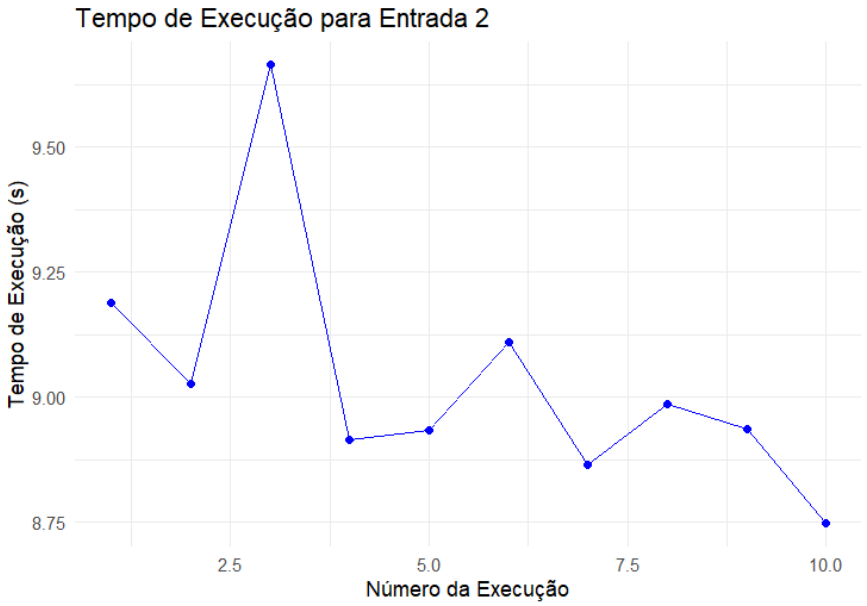

Figura 4.1.2 - Gráfico de linha para o tempo real de execução do input 2.

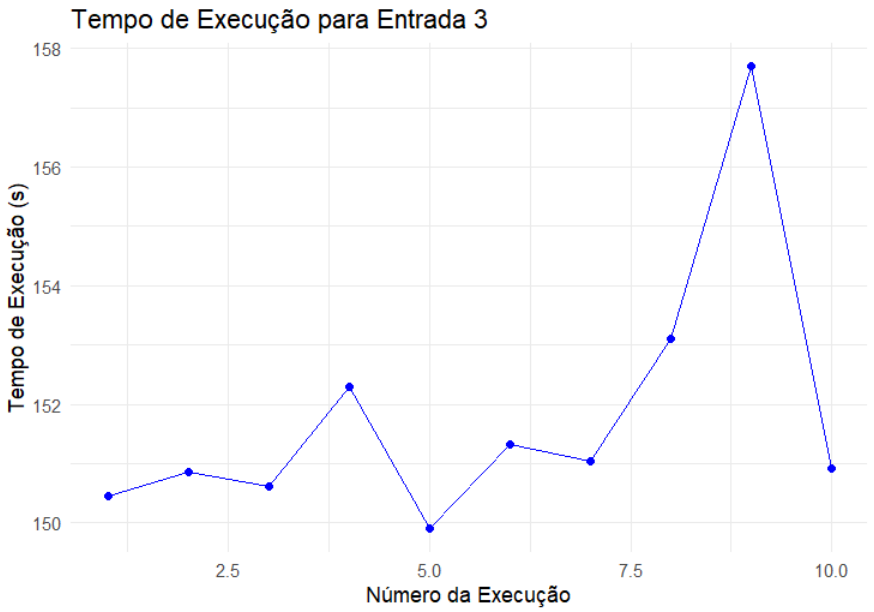

Figura 4.1.3 - Gráfico de linha para o tempo real de execução do input 3.

Após isso, foi feito o gráfico da Figura 4.1.4 comparando o tempo de execução real das 3 entradas por meio do RStudio.

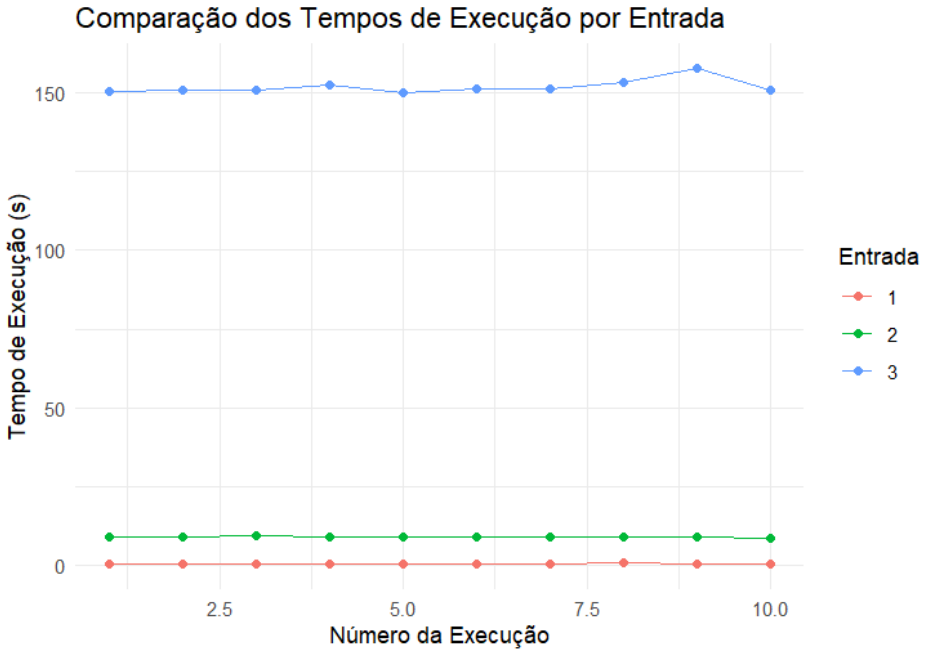

Figura 4.1.4 - Gráfico de linhas comparando tempo real de execução entre as 3 entradas.

4.2 Leitura e Escrita de Registros

As Figuras 4.2.1, 4.2.2 e 4.2.3 mostram as saídas do código para uma das execuções das entradas 1, 2 e 3, respectivamente. Essas saídas fornecem a

quantidade de vezes que os registros foram lidos e escritos no disco, referentes tanto a um único registro, quanto a todos os registros do arquivo de entrada.

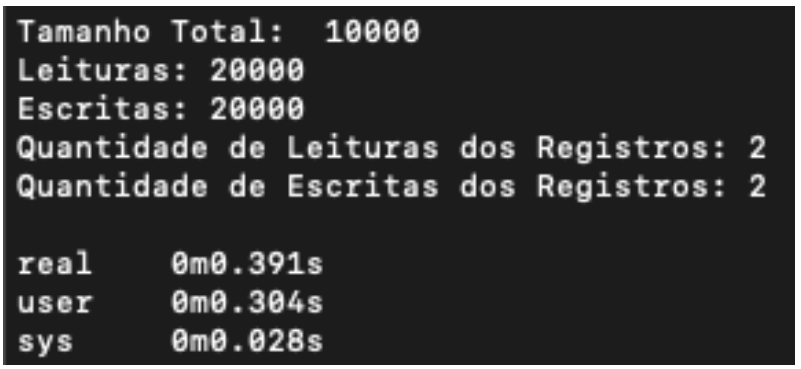

Figura 4.2.1 - Saída da 1ª execução do input 1.

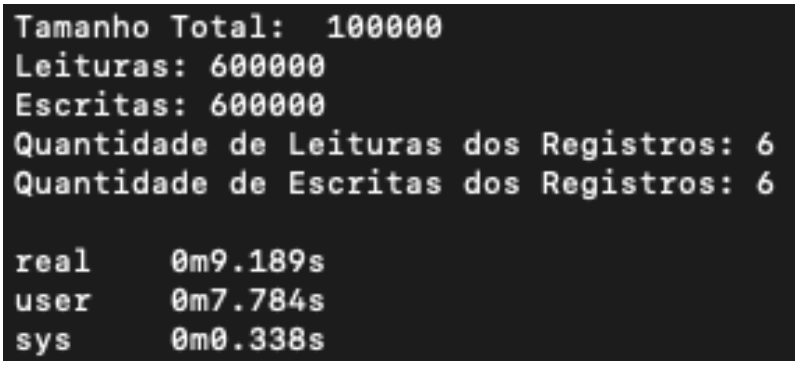

Figura 4.2.2 - Saída da 1ª execução do input 2.

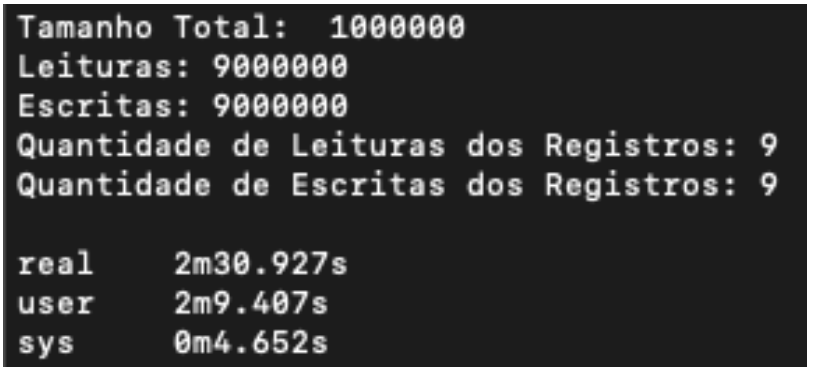

Figura 4.2.3 - Saída da 10ª execução do input 3.

5 CONCLUSÃO

De acordo com os resultados obtidos, os tempos de execução foram pequenos de acordo com cada entrada fornecida, além disso, foi observado que o número de acesso às unidades de memória externa foi baixo comparado com o tamanho de cada entrada fornecida, portanto foi mostrado a importância da ordenação externa para arquivos de tamanho maior que a memória interna disponível.