Trabalho Prático 1

Thaíssa Malaquias Couto - 2022094896

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte - MG - Brasil

tmc2022@ufmg.br

1. Introdução

O Problema proposto foi o Ordenação de registros contidos em um arquivo. A Ordenação deve usar a movimentação de dados indireta, segundo a qual os dados não são copiados e sim rearranjados por meio do uso de ponteiros. Além disso, deve-se obter uma ordenação por múltiplas chaves.

Para cumprir a proposta apresentada acima, o programa utiliza uma matriz para representar esses dados, já que eles não podem ser ordenados diretamente no arquivo. Ou seja, o arquivo original se mantém intacto. Essa matriz possui como colunas os atributos dos dados que permite a ordenação por múltiplas chaves e como linhas números naturais (começando com 0) em ordem crescente, fazendo com que o total de linhas corresponda ao número total de registros no arquivo.

Ademais, devem ser apresentados três métodos de ordenação diferentes. Um deles deve ser o QuickSort. Os outros dois métodos escolhidos foram: Selection Sort, por ser ideal para grandes quantidades de registros, e MergeSort, por ser eficiente.

No fim, deve ser possível imprimir os registros ordenados, através de uma chave qualquer escolhida.

2. Método

2.1 Configurações da Máquina

O programa foi desenvolvido e testado em um ambiente Linux Mint 21.3 (Ubuntu), utilizando a linguagem C, e compilando com o GCC. O processador da máquina é um Intel core i5 9400f, com memória RAM de 16GB.

2.2 Estruturas de Dados

A estrutura de dados utilizada no programa é a **Matriz**. O uso dessa estrutura é justificado pelo requisito de ordenação indireta por múltiplas chaves, pois cada coluna representa uma chave e as linhas possuem as referências (índices) para cada dado, que é tratado como um elemento da matriz. Ou seja, o número de linhas é o número de dados no arquivo. Através de permutações das linhas (pelos métodos), obtém-se no fim, a matriz com os elementos ordenados de acordo com a chave escolhida.

A Matriz é implementada através do uso de ponteiros para ponteiros. Ela é alocada dinamicamente no momento em que o arquivo é carregado, para se obter a quantidade de memória necessária e evitar realocações. Depois de alocada, ela é preenchida com as referências (índices) aos dados.

Nos casos além da matriz de índices N x M (sendo N o número de registros e M o número de atributos), cada matriz N x 1 armazena os dados do arquivo de acordo um atributo especifico.

2.3 Classes

OrdInd é a estrutura utilizada por todo o programa. Ela armazena o número de registros, o número de atributos (corresponde a quantidade de itens no cabeçalho de uma planilha por exemplo), 4 matrizes n x 1 que armazenam os dados Nomes, CPFs, Endereços e Outros respectivamente, e a matriz NxM Indices.

Load é a classe responsável pelo carregamento e armazenamento dos dados na estrutura OrdInd.

2.4 Funções

CarregaArquivo: Essa função armazena os dados do arquivo na estrutura OrdInd por atributo, preenchendo as quatro matrizes n x 1.

Cria Indice: Essa função aloca e inicializa o vetor de índices de um atributo específico, com números começando em 0 (zero) e indo até o número total de registros, preenchendo uma coluna por vez, a matriz Indices.

Comparacao_Elementos: A partir de índices de referência, compara dois elementos na matriz correspondente ao atributo dado. Essa função é a base de todos os algoritmos de ordenação implementados aqui (pois são baseados em comparações de chaves).

OrdenaIndice_QuickSort: Essa é a função geral do método de ordenação do QuickSort. Implementa o algoritmo clássico, mas com uma pequena melhoria no cálculo do pivô, buscando evitar o pior caso. Ela depende das funções auxiliares Particao_QS e QuickSort_rec.

> **Particao_QS:** A partir do vetor de entrada, calcula o pivô utilizando mediana de três. Os elementos da esquerda e da direita são comparados com o pivô, afim de se obter os elementos maiores que ele do lado direito, e os menores ao lado esquerdo. Quando essa lógica é burlada, uma troca entre elementos que estão em posições inapropriadas é feita. O retorno da função é o ponto de particionamento do vetor.
>
> **QuickSort_rec:** Através de Particao_QS, a função obtém acesso ao meio do vetor, partindo-o em dois. Então, isso é feito recursivamente para criar novas partições, até que mais nenhuma partição possa ser feita.

No fim da função, quando não há mais particionamentos possíveis por QuickSort_rec, obtemos os elementos já ordenados por consequência dos processos em Particao_QS.

OrdenaIndice_Selecao: A versão do algoritmo clássico utilizado é o que ordena do fim para o início trocando o último elemento do vetor já ordenado, pelo maior elemento no subvetor ainda não ordenado.

OrdenaIndice_MergeSort: Essa função implementa o algoritmo clássico de MergeSort, com uma pequena melhoria de parar o algoritmo ao verificar que o vetor já está ordenado. Ela utiliza duas funções auxiliares para funcionar corretamente:

> **Merge:** Essa é a função que junta os subvetores que foram criados durante a partição. Ela os combina de forma a ordená-los.
>
> **MergeSort_rec:** Durante suas chamadas recursivas, parte os subvetores ao meio, até que chegue em vetores com um elemento cada um. Neste momento, Merge é chamado.

3. Instruções de compilação e execução.

Em um ambiente Linux, utilizar o comando make all.

4. Análise de Complexidade

4.1 Tempo

CarregaArquivo: A complexidade da função é proporcional ao número de registros (n). Apresenta um for que tem um custo constante O(1), pois o número de atributos, que equivale ao número de iterações é muito menor que n. também há um while que executa n iterações. A complexidade assintótica é **Theta(n).**

Cria Indice: Utiliza um laço for proporcional a n. A complexidade assintótica é **Theta(n)**.

Comparacao_Elementos: A função não utiliza nenhum laço pois apenas compara dois elementos. Seu custo é **O(1)**.

OrdenaIndice_QuickSort: Para analisar essa função, primeiro deve-se analisar Particao_QS e QuickSort_rec.

> **Particao_QS:** O laço principal percorre todo o vetor realizando comparações e trocas. Portanto, sua complexidade é **Theta(n)**.
>
> **QuickSort_rec:** A cada chamada da função, Particao_QS é chamada com um custo O(n). Em cada chamada recursiva, a função divide o problema em dois subproblemas menores. O custo disso é O (log n). Logo, O custo total é **O(n log n)**.

Vale ressaltar que devido ao uso da mediana de três para calcular o pivô, caímos sempre no caso médio, pois estamos evitando o pior caso, que seria $O(n^2)$.

OrdenaIndice_Selecao: O loop mais externo percorre o vetor de trás para frente, n-1 vezes. Para encontrar o maior elemento no loop interno, são necessárias n iterações. No fim, temos que o algoritmo é **O(n²).**

OrdenaIndice_MergeSort: O custo total é **O(n log n).**

Merge: Realiza a cópia dos elementos a um custo O(n).

MergeSort_rec: Divide o problema em subproblemas a um custo O(log n).

4.2 Espaço

OrdenaIndice_Selecao: O(1), pois é in-place.

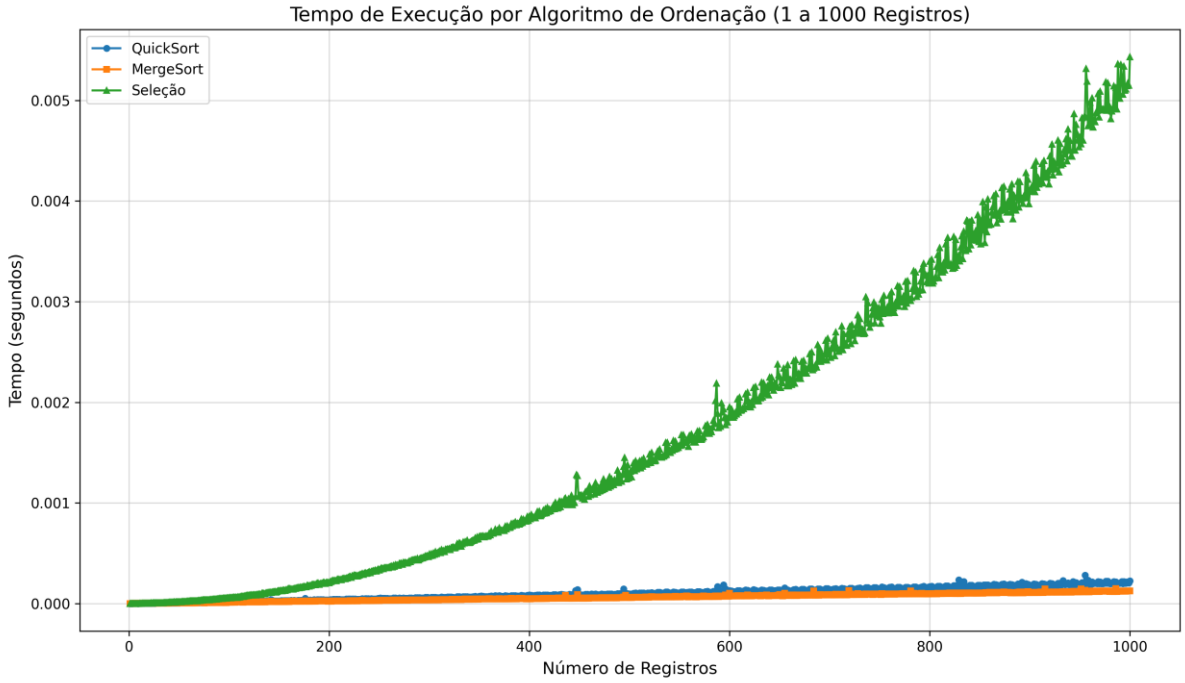
OrdenaIndice_QuickSort: O(1), in-place.

OrdenaIndice_MergeSort: O(n), a versão implementada não é in-place.

5. Estratégias de Robustez

Uma classe **excecoes** foi criada para tratar casos excepcionais. Tais como: falha na alocação de memória, atributos inválidos e falha em leituras de arquivos. No caso de ocorrência destes casos, a abordagem aplicada pela classe é a abortiva.

6. Análise Experimental



Em relação ao **Tempo de Execução**, o algoritmo de Seleção obteve o custo esperado de O(n²). QuickSort e MergeSort obtiveram custos quase idênticos, sendo o QuickSort um pouco mais custoso. Este resultado se deve ao fato de que o QuickSort geralmente atinge o caso médio, sendo este cenário ainda mais persistente quando o conceito de "mediana de três" é utilizado para a escolha do "pivô".

No que diz respeito a Localidade de Referência, os algoritmos obtiveram os seguintes resultados:

Figura 1 - Distância de pilha

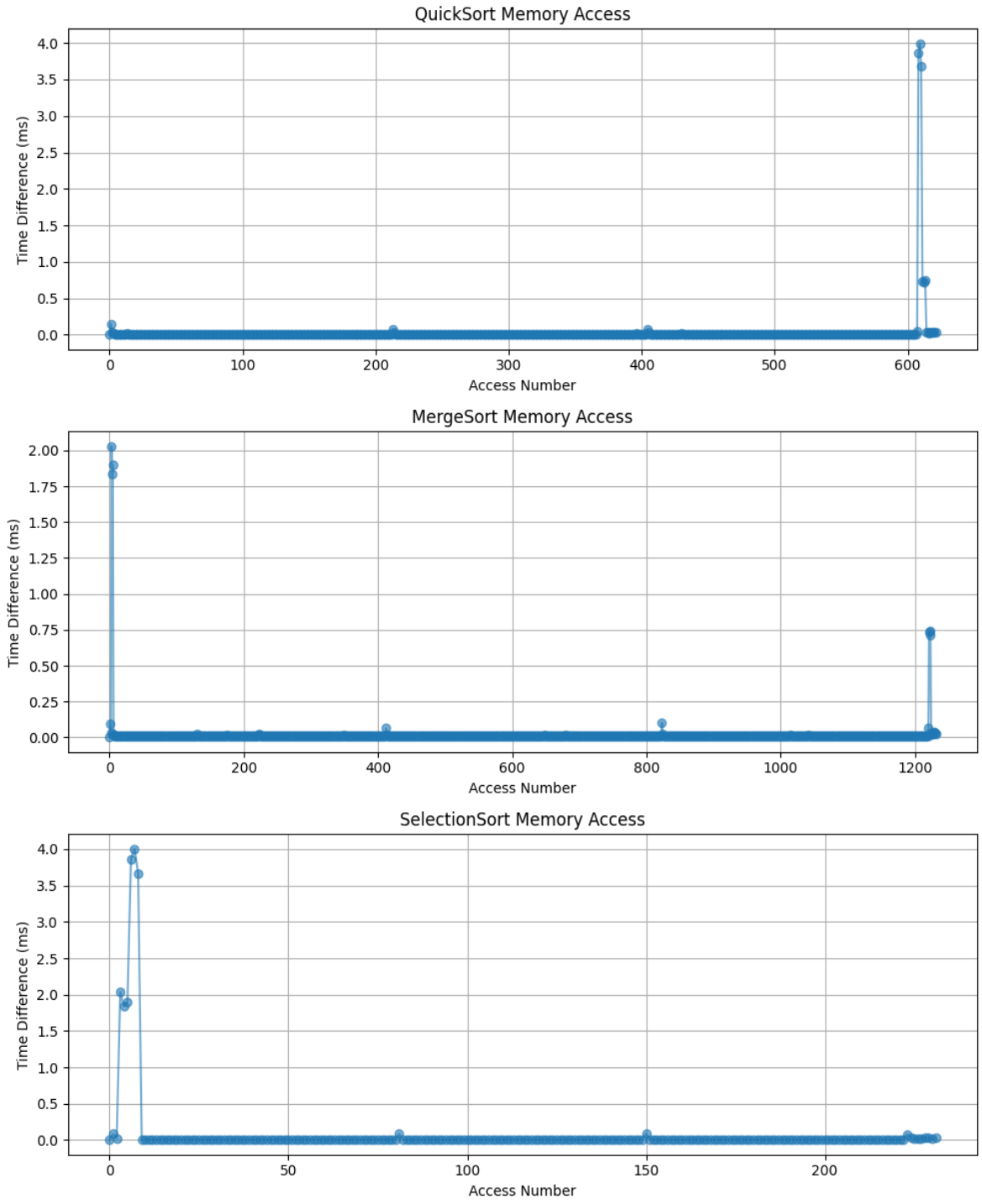


Figura 2 - Localidade de Referência do Algoritmo Seleção

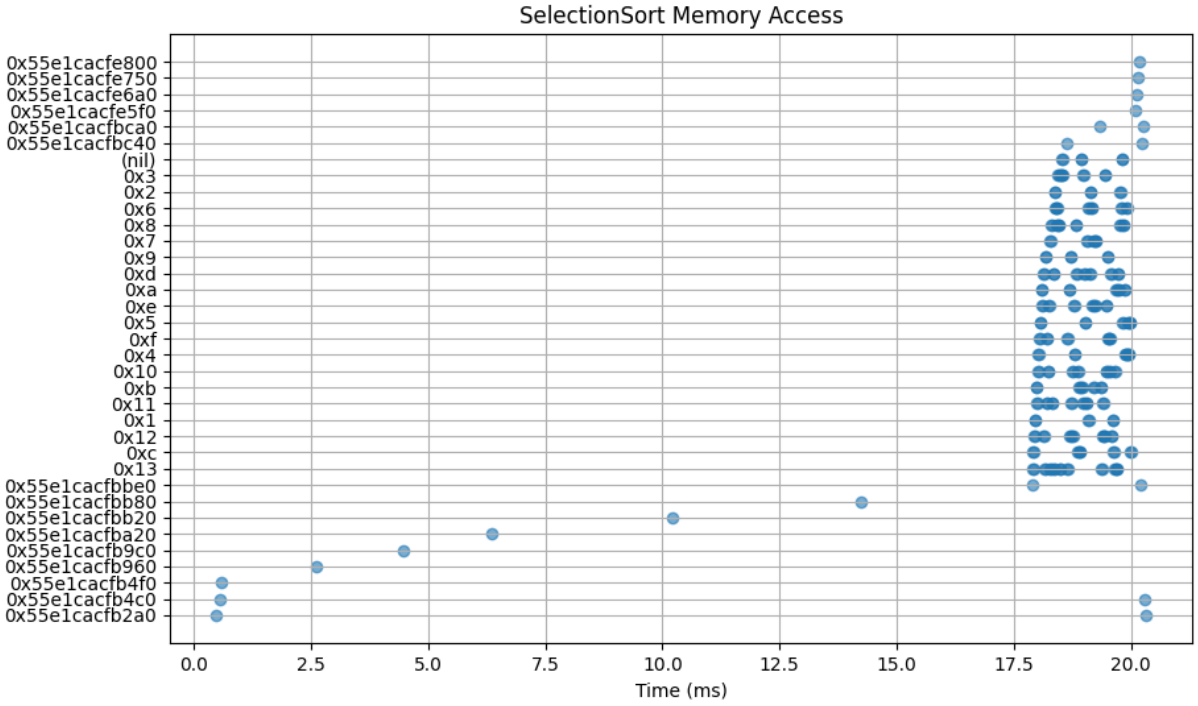


Figura 3 - Localidade de Referência do Algoritmo QuickSort

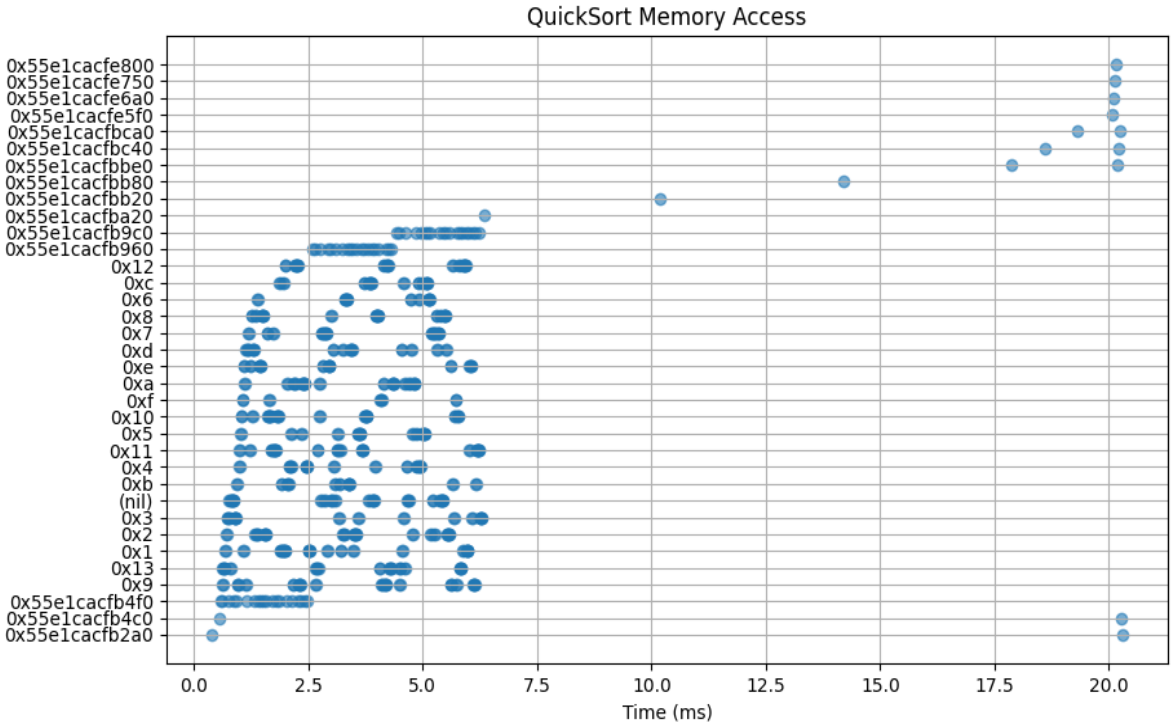
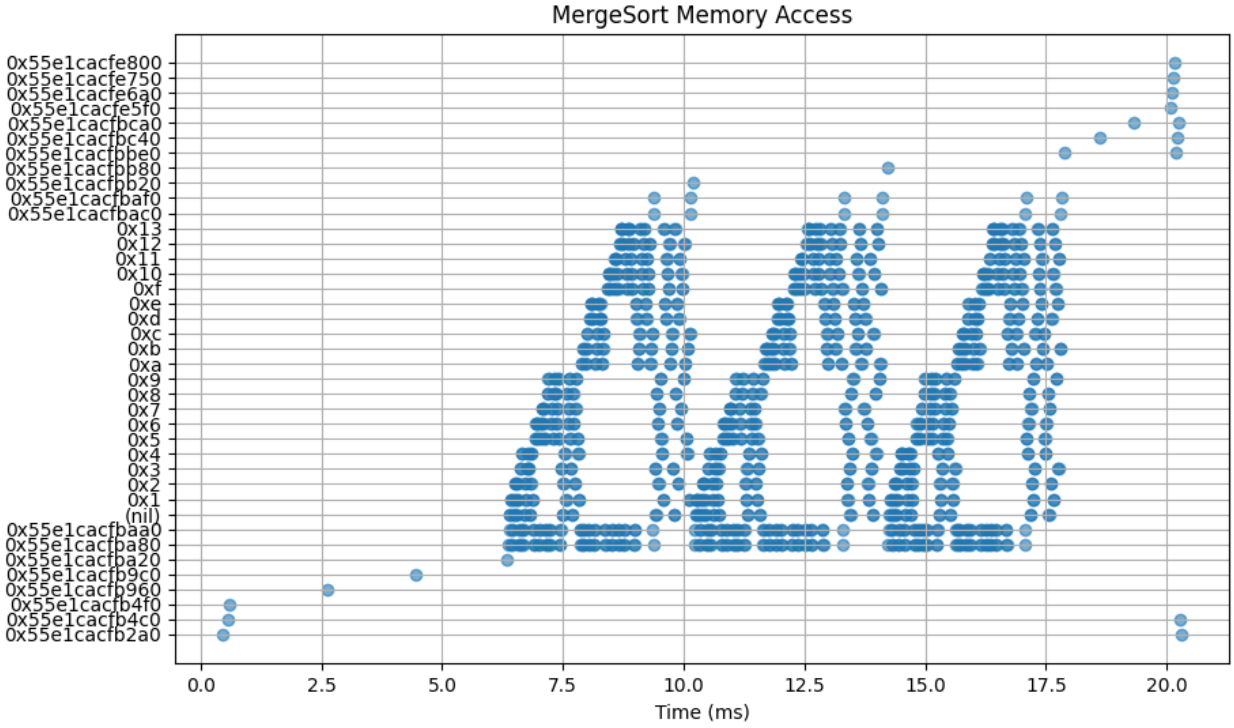


Figura 4 - Localidade de Referência do Algoritmo MergeSort



A partir da Figura 2, podemos ver que o Selection Sort obteve uma boa localidade espacial, o que já era esperado pelo fato de seu algoritmo iterar sobre o vetor, acessando posições adjacentes a procura do maior elemento. Seguindo essa lógica, a localidade temporal é menos favorável, pois à medida em que avançamos na ordenação, posições deixam de ser acessadas durante todo o resto do algoritmo como comprovado no gráfico.

Na Figura 3, podemos observar que o QuickSort obteve uma localidade temporal e espacial boa, com destaque na temporal. Mas, não se compara ao Merge Sort que obteve uma ótima localidade, tanto espacial como temporal como se pode ver pela Figura 4. Isso provavelmente se deve ao fato de que apesar dos dois algoritmos apresentarem a abordagem de divisão e conquista, MergeSort ainda apresenta a função Merge para a junção dos subvetores, que faz com que as posições sejam novamente acessadas, assim como suas posições adjacentes.

Dadas as considerações acima, a Figura 1 nos mostra a distância de pilha obtida para cada um dos algoritmos. QuickSort e MergeSort obtiveram diferenças de tempo no acesso à memória muito pequenas. Em contraponto, Selection Sort obteve uma diferença considerável logo no início da execução, mostrando assim sua ineficiência em comparação aos outros dois algoritmos.

7. Conclusão

Este trabalho teve como objetivo abordar os impactos da escolha de implementação dos algoritmos de ordenação na execução de um programa, podendo-se expandir para os algoritmos em geral. Observa-se que a eficiência deve

ser considerada ao se programar qualquer coisa, pois no geral estamos buscando solucionar problemas e não causar outros. Dito isso, cada caso deve ser analisado e observado para melhor aproveitamento dos recursos.

8. Bibliografia

Meira, W. and Lacerda, A. (2024/2). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte