## **Trabalho Prático 1**

Estudante: Sarah Menks Sperber

Matrícula: 2023001824

### **Introdução:**

Este trabalho consiste em melhorar a infraestrutura de dados de um arquivo com um grande número de informações. Para resolver esse problema, serão implementados três algoritmos diferentes: Quick Sort (com média de 3), Insertion Sort e Selection Sort, que farão o trabalho de ordenar o arquivo. Assim, no decorrer deste relatório, vamos fazer uma análise profunda desses algoritmos, discutindo pontos como metodologias usadas, uso de memória e complexidade assintótica.

### **Métodos:**

O código usado é, em tese, bem simples. Foram usados dois TADs, Pessoa.hpp e Ordenacao.hpp, que guardam as declarações de todas as funções do código.

Os arquivos Pessoa.cpp e Ordenacao.cpp apenas implementam as funções, que serão discutidas posteriormente.

Por fim, a main.cpp é aquela que executa o código, já que é ela que abre o arquivo e chama as funções.

**1. Pessoa.hpp:**

O TAD Pessoa.hpp é responsável por criar a struct Pessoa, que armazena o nome, cpf, endereço e as outras informações de uma das linhas do arquivo. Esse TAD também contém uma função que imprime essas informações da Pessoa.

**2. Ordenacao.hpp:**

Cria a struct Ordenação, que declara um vetor de Pessoas, um vetor de índices (usado para ordenação indireta), os algoritmos de ordenação e funções auxiliares, uma que retorna o tipo de chave a ser ordenada (nome, cpf ou endereço) e outra que imprime o arquivo.

As funções principais realizam as declarações do construtor (que cria e aloca o vetor de “Pessoa”) e as declarações dos algoritmos de ordenação (Quick Sort, Insertion Sort e Selection Sort).

As funções auxiliares consistem, basicamente, em imprimir a saída da ordenação e criar o vetor de Pessoas. As impressões incluem tanto a primeira parte do arquivo, com as informações de formatação (chamadas de cabeçalho) quanto as próprias linhas do arquivo. As outras funções escolhem o atributo a ser ordenado e retornam a string a ser comparada (para auxiliar na ordenação indireta), além de criar o vetor de pessoas.

**3. Main.cpp:**

Recebe o arquivo a ser ordenado por parâmetro, declara um objeto do tipo “Ordenacao” e chama as funções Insertion Sort, Quick Sort e Selection Sort. Além disso, para um código mais seguro, a main.cpp possui boa parte das estratégias de robustez (como as checagens de parâmetro).

### **Análise de Complexidade:**

Como dito anteriormente, foram implementados três algoritmos de ordenação, Quick sort, Insertion sort e Selection sort. Os casos de teste se deram por arquivos (não ordenados) de diversos tamanhos, em um intervalo de 10 a 5.000 linhas.

Vale destacar também o uso do vetor de índices mencionado anteriormente, que não é necessário para a execução dos algoritmos, mas que também depende do tamanho da entrada. Assim, esse vetor de índices adiciona O(n) na complexidade espacial do código.

**1. Quick Sort:**

O pior caso do quicksort é quando a divisão do vetor é mal balanceada, o que torna sua complexidade assintótica quadrática. Por causa disso, foi implementada a versão do quicksort que escolhe o pivô com uma média de 3 valores da lista, o que otimiza o algoritmo. Além disso, esse algoritmo não depende de um vetor auxiliar, porém, ele utiliza o vetor de índices declarado dentro do objeto “Ordenacao”, o que piora sua complexidade espacial.

Complexidade assintótica no pior caso: O(n²). Complexidade espacial: O(n) (com o vetor de índices).

**2. Insertion Sort:**

O insertion sort é muito eficiente para valores pequenos, já que assim ele realizará poucas comparações. Seu melhor caso é quando o vetor já está ordenado, apresentando uma complexidade linear. Entretanto, quando o vetor está inversamente ordenado, o algoritmo apresenta um tempo quadrático (que é seu pior caso).

Complexidade assintótica: O(n²). Complexidade espacial: O(n) (com o vetor de índices).

**3. Selection Sort:**

A complexidade do Selection sort não muda de acordo com a entrada do arquivo, ou seja, ele sempre será quadrático (o que faz dele um algoritmo mais lento). Sua implementação é simples, se baseando em encontrar o menor elemento do vetor e colocá-lo na primeira posição. Por não depender do tamanho da entrada, o Selection obteve o pior desempenho, ponto que discutiremos mais abaixo.

Complexidade assintótica: O(n²). Complexidade espacial: O(n) (com o vetor de índices).

**4. Outras funções:**

Como foi dito anteriormente, as outras funções auxiliam os algoritmos de ordenação. Suas implementações são mais simples e mais diretas, e por isso suas análises de complexidade são mais triviais.

*CriaPessoas: O(n)*

* Percorre todo o arquivo

*ImprimeCabecalho: O(k)*

* (onde k é o tamanho do cabeçalho)

*ImprimeArquivo: O(n)*

* Percorre todo o vetor de Pessoas

*ResetIndices: O(n)*

* Percorre todo o vetor de índices

*EscolheAtributo: O(1)*

*RetornaChave: O(1)*

### 

### **Estratégias de Robustez:**

Para garantir um código seguro e sem falhas, algumas estratégias de segurança foram implementadas, tais como:

**1. Função parse\_args que checa os parâmetros:**

Para evitar problemas com entradas inválidas, uma função de checagem dos parâmetros foi adicionada no arquivo main.cpp. Nela, nós checamos a quantidade de parâmetros passados para o programa, e caso o usuário não tenha colocado a quantidade de parâmetros adequada, o código não funcionará.

**2. Verificação de arquivo válido:**

Assim como a função parse\_args checa os parâmetros, a função main possui um teste de checagem do arquivo. Se o usuário não passar a entrada desejada por parâmetro (um arquivo válido), o código é encerrado e uma mensagem de texto é exibida.

**3. Verificação de atributo escolhido:**

Além das funções mencionadas acima, o código confere o atributo a ser ordenado. Caso a pessoa não escolha um dos três atributos a serem ordenados (nome, CPF e endereço), o programa exibe uma mensagem de erro e é encerrado.

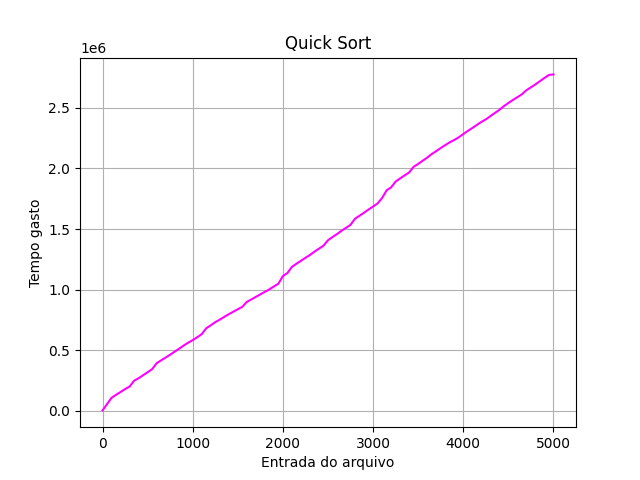
### **Análise Experimental:**

**1. Tempo:**

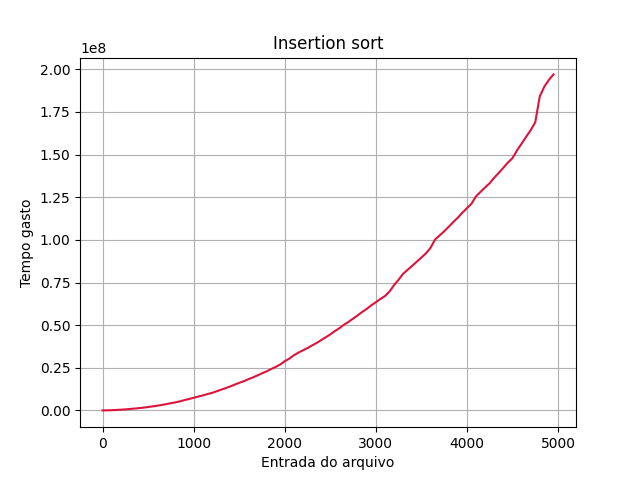
Para a análise temporal, usei a biblioteca chromo para medir o tempo de execução de cada algoritmo e, assim, montar os gráficos.

Como esperado, o Quicksort obteve o melhor desempenho nos casos de teste, com seu tempo total chegando à casa dos 3\*106 nanossegundos. Já o Insertion teve um tempo de execução pior que o Quick Sort, chegando aos 2\*108 nanossegundos. Por fim, o Selection de longe obteve o pior desempenho, \alcançando a marca dos 5\*108 nanossegundos.

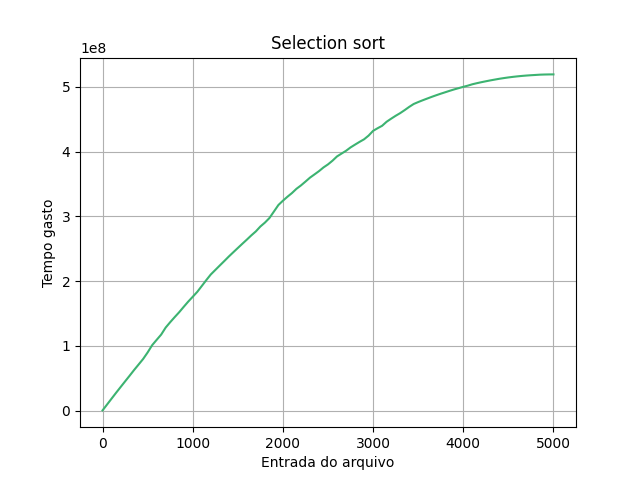
**Gráfico com tempo de execução Quick Sort:**

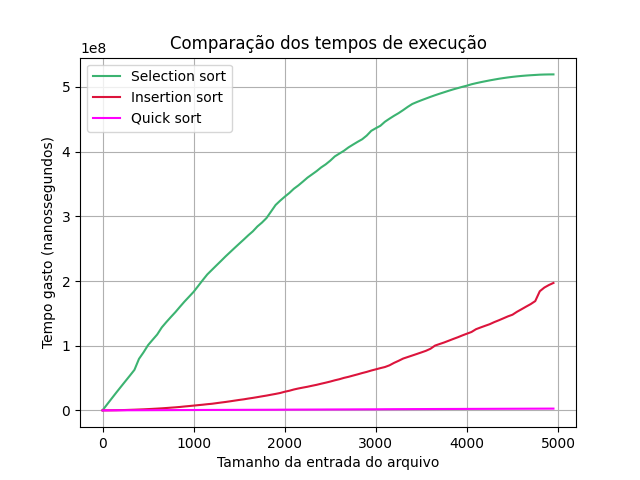


**Gráfico com tempo de execução Insertion Sort:**



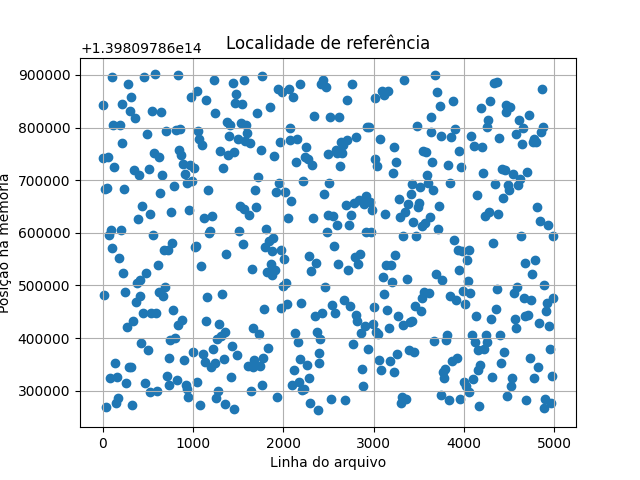
**Gráfico com tempo de execução Selection Sort:**





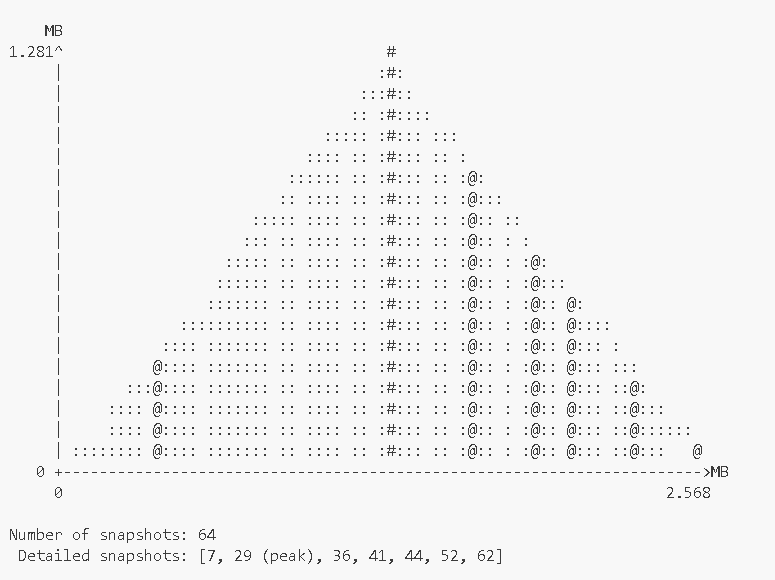
**2. Localidade de referência:**

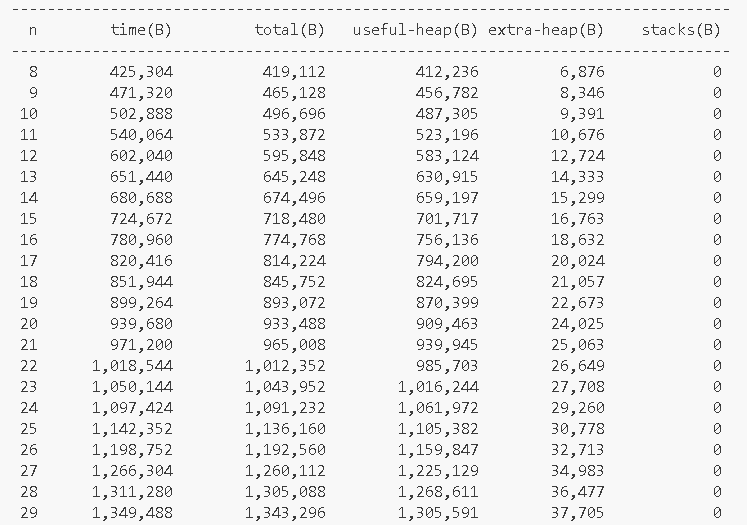
No gráfico abaixo, vemos a dispersão do uso de memória para a impressão do arquivo, que não é tão eficiente, já que ele acessa posições aleatórias na memória.



**3. Distância de Pilha**

Através do valgrind, podemos olhar a forma como o código usa o heap. Os algoritmos foram testados dentro do mesmo caso, começando pelo Insertion, seguido pelo quicksort e terminando com o selection sort. Ao analisar um arquivo menor, sem levar em consideração o uso de stack, o código apresentou um comportamento bem equilibrado, com seu ponto de maior uso localizado no meio da execução.

****



***Legenda tabela:***

*n: número da linha;*

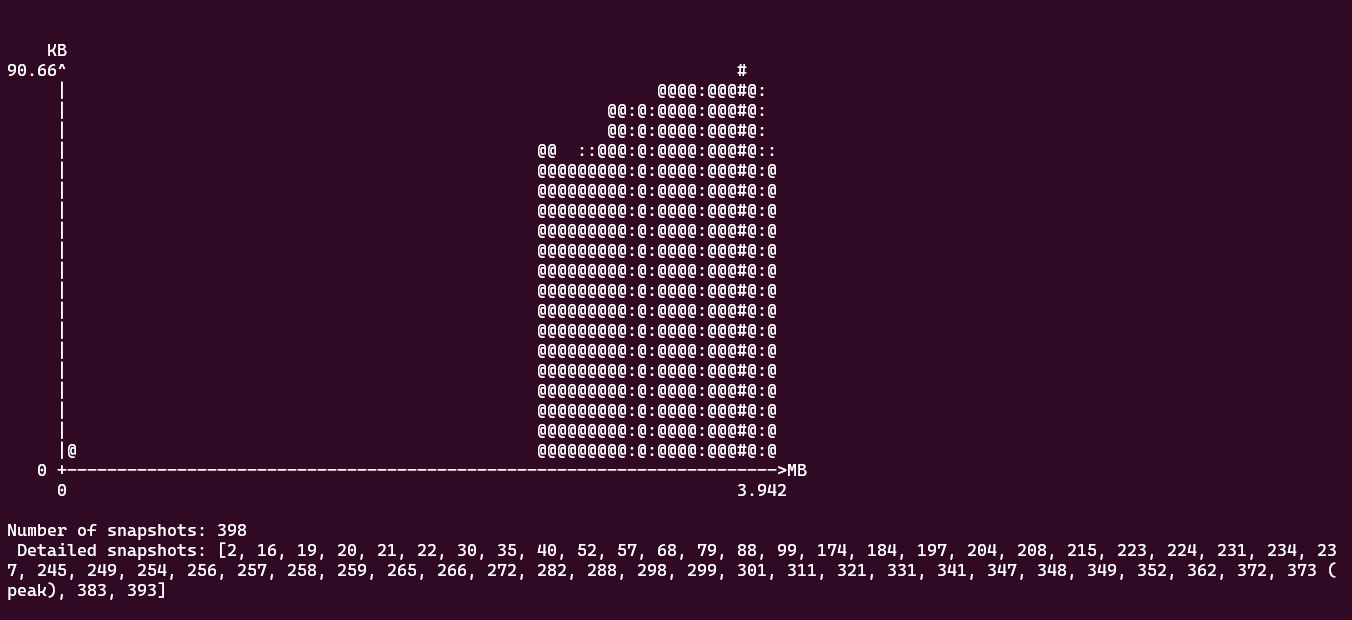
*time(B): Tempo, medido em bytes;*

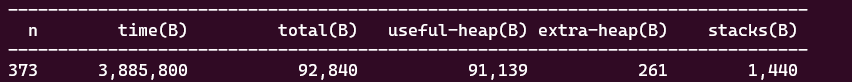
*Total(B): Quantidade de memória usada naquele ponto;*

*Useful-heap(B): Bytes alocados no heap, que são necessários para o programa;*

*Extra-heap(B): Memória extra alocada, como um processo auxiliar;*

Já para arquivos maiores, analisando também o uso de stack, o código obteve uma grande mudança no comportamento. Agora, seu maior uso de heap foi na parte final da execução, que é onde se encontra o Selection sort, algoritmo mais custoso deste trabalho.





***Legenda tabela:***

*n: número da linha;*

*time(B): Tempo, medido em bytes;*

*Total(B): Quantidade de memória usada naquele ponto;*

*Useful-heap(B): Bytes alocados no heap, que são necessários para o programa;*

*Extra-heap(B): Memória extra alocada, como um processo auxiliar;*

*Stacks(B): Uso de stack*

**Conclusão:**

Após analisar detalhadamente o código feito, concluímos que os diferentes algoritmos de ordenação possuem desempenhos diferentes, mesmo com complexidade assintótica semelhantes no pior caso.

Neste trabalho, aprendi a medir e comparar os diferentes tempos de execução dos algoritmos, aprendi também a avaliar a localidade de referência e a distância de pilha do código, além de conhecer os algoritmos de ordenação mais profundamente.

**Bibliografia:**

MATPLOTLIB. *Matplotlib 3.0.0 documentation*. Disponível em:<https://matplotlib.org/stable/>. Acesso em: 2 dez. 2024.

C++ REFERENCE. *chrono - C++ Reference*. Disponível em:<https://cplusplus.com/reference/chrono/>. Acesso em: 2 dez. 2024.

VALGRIND. *Valgrind User Manual*. Disponível em:<https://valgrind.org/docs/manual/ms-manual.html>. Acesso em: 2 dez. 2024.