TP 2 - AEDS II

Nome: Diego Resende Braz **Matrícula: 212050084**

1 - Introdução

Ordenar é um processo de rearranjar um conjunto de objetos em ordem ascendente ou descendente. Ela visa a facilitar a recuperação posterior de itens do conjunto ordenado. Na programação, a ordenação trabalha sobre os registros de um arquivo. Cada registro possui uma chave utilizada para controlar a ordenação. Qualquer tipo de chave sobre a qual exista uma regra de ordenação pode ser utilizado. Um método de ordenação é estável se a ordem relativa dos itens com chaves iguais não se altera durante a ordenação, porém nem todos os métodos de ordenação mais eficientes são estáveis.

Apesar dos métodos de ordenação serem baseados em comparações de chaves, existem dois métodos de ordenação:

- Ordenação interna: o arquivo ordenado cabe inteiramente na memória principal. Qualquer registro pode ser imediatamente acessado.
- Ordenação externa: o arquivo ordenado não cabe inteiramente na memória principal. Os arquivos são acessados sequencialmente ou em grandes blocos.

Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.

Os métodos eficientes são aqueles melhor adequados para arquivos maiores, eles usam menos comparações, elas são mais complexas nos detalhes.

1.1 - Ordenação por Seleção

Um dos algoritmos mais simples de ordenação.

A ordenação é realizada selecionando o menor elemento não ordenado. Repete-se esse processo até que todos os demais elementos do vetor estejam ordenados.

- Vantagens: Tem custo linear na entrada para o número de movimentos de registro, com isso, torna-se um algoritmo ideal para ser utilizado para arquivos de registros grandes, e interessante para arquivos pequenos também.
- Desvantagens: O algoritmo não é estável, e o vetor estar ordenado não ajuda em nada, pois o custo continuará linear.

1.2 - Ordenação por Inserção

É o método a ser utilizado quando o arquivo está quase ordenado.

É um método bom quando se deseja adicionar alguns itens a um arquivo ordenado, devido ao seu custo linear.

A ordenação é realizada a partir do segundo elemento do vetor. A comparação acontece com os elementos anteriores e a partir disso é posto na posição correta, deslocando todos os elementos maiores para a direita.

1.3 - ShellSort

A ordenação por shellsort divide o vetor em grupos e aplica a ordenação por inserção em cada um dos grupos com um intervalo entre eles. O intervalo é reduzido até 1, quando a inserção é realizada no vetor por completo.

- Vantagens: É uma ótima opção para arquivos de tamanho moderado, e sua implementação é simples e requer códigos pequenos.

- Desvantagens: Método não estável, e o tempo de execução depende da ordem inicial do vetor.

1.4 - QuickSort

A ordenação por quicksort se dá a partir da escolha de um pivô e rearranja o vetor de modo que os elementos menores que o pivô estejam à sua esquerda e os elementos maiores estejam à sua direita. Esse processo é realizado recursivamente nas subpartes do vetor até que o vetor seja ordenado por inteiro.

- Vantagens: É eficiente para ordenar arquivos de dados, necessita de uma pilha como memória auxiliar, requer n log n comparações para ordenar n itens.
- Desvantagens: Um pequeno erro pode levar a efeitos inesperados para algumas entradas de dados. O método não é estável.

1.5 - HeapSort

Possui os mesmos princípios da ordenação por seleção. O custo para encontrar o menor ou maior item entre n itens pode ser reduzido utilizando uma fila de prioridades - TAD.

Não é recomendado para arquivos que tenham poucos registros.

- Vantagens: O comportamento do Heapsort é sempre O(n log n), qualquer seja a entrada.
- Desvantagens: Não é estável. O anel interno do algoritmo é complexo se comparado com o QuickSort.

1.6 - MergeSort

A ordenação por MergeSort acontece dividindo o vetor em duas metades, ordenando cada metade de maneira recursiva, e em seguida, mescla as duas metades ordenadas.

- Vantagens: Complexidade é constante, e possui uma implementação simples.
- Desvantagens: Gasta muita memória

2 - Problema Proposto

A ideia do trabalho é desenvolver um programa que consiga ordenar, utilizando códigos dos algoritmos de ordenação vistos em sala de aula (Seleção, Inserção, Shellsort, Quicksort, Heapsort e Mergesort), utilizando também de algoritmos que calcule o tempo de execução em diversas situações (vetores de tamanho de 20 elementos, 500 elementos, 5000 elementos, 10000 elementos e 200000 elementos), considerando duas possibilidades de tamanho de registros (pequeno, com apenas a chave, e um registro grande, de no mínimo 50 campos com strings de 50 caracteres além da chave). A ordenação deve ser dada através de algumas possibilidades de ordenação inicial de cada vetor de entrada (ordem crescente, ordem decrescente e aleatória). No programa deve ser analisada o tempo de execução, a quantidade de comparações e o número de movimentações que foram realizadas para ordenar o vetor. Foi realizada a execução 10 vezes para cada algoritmo de ordenação, sendo apresentado no final o tempo médio gasto por cada algoritmo.

3 - Modelagem

Para resolver esse problema foi utilizado a linguagem C no Visual Studio Code, pela arquitetura Windows.

4 - Funções

Nessa sessão, será explicada a funcionalidade de algumas funções utilizadas no decorrer do algoritmo.

4.1 - typedef struct{
int chave;
} Item_leve;

Essa função tem como objetivo registrar a chave que será responsável por ordenar os vetores de registro pequeno.

4.2 - typedef struct {
int chave;
char peso_morto [50][50];
} Item_pesado;

Essa função tem como objetivo registrar a chave que será responsável por ordenar os vetores de registro grande, e a string peso_morto será utilizada para fazer peso no vetor, utilizando 2500 bytes de memória a mais, por registro. Com isso, levará mais tempo para fazer as trocas na hora da ordenação.

4.3 - typedef Item_leve Item;
typedef Item_pesado Item;

Essas funções são utilizadas para alternar entre os registros. Dessa forma, sempre que no código for visto “Item” ele vai assimilar como “Item_leve” ou como “Item_pesado”.

4.4 - void trocar (Item *a, Item *b){
Item temp = *a;
*** a = *b;**
***b = temp;**
}

É uma função auxiliar utilizada em algoritmos de ordenação. Cria-se a variável ‘temp’, para salvar o valor original de ‘a’ antes da troca, e é atribuído a ela o valor apontado por ‘a’. O valor de ‘b’ é copiado para a variável apontada por ‘*a’. O valor original de ‘*a’ é copiado para a variável apontada por ‘b’. Dessa forma, os valores das variáveis ‘*a’ e ‘*b’ são trocados.

4.5 - void selecao (Item vetor[], int tamanho)

A função ‘selecao’ implementa o algoritmo de ordenação por seleção para ordenar um vetor do tipo Item. Ela recebe como parâmetros o vetor de itens ‘vetor[]’ e o tamanho do vetor ‘tamanho’.

4.6 - void insercao (Item vetor[], int tamanho)

A função ‘insercao’ implementa o algoritmo de ordenação por inserção para ordenar um vetor do tipo Item. Ela recebe como parâmetros o vetor de itens ‘vetor[]’ e o tamanho do vetor ‘tamanho’.

4.7 - void shellsort(Item vetor[], int tamanho)

A função ‘shellsort’ implementa o algoritmo de ordenação por shellsort para ordenar um vetor do tipo Item. Ela recebe como parâmetros o vetor de itens ‘vetor[]’ e o tamanho do vetor ‘tamanho’.

4.8 - int particionar(Item vetor[], int esq, int dir)

A função ‘particionar’ é essencial para o algoritmo de ordenação quicksort. Ela é responsável por particionar o vetor em duas partes, de forma que todos os elementos menores que um pivô fiquem à esquerda e todos os maiores ou iguais fiquem à direita.

A função recebe como parâmetros o vetor de itens 'vetor', o índice da posição esquerda ‘esq’ e o índice da posição direita ‘dir’. O objetivo é escolher um pivô e rearranjar os elementos do vetor de modo que todos os elementos menores que o pivô fique à esquerda dele e todos os elementos maiores ou iguais fiquem à direita.

4.9 - void quicksort (Item vetor[], int esq, int dir)

A função ‘quicksort’ implementa o algoritmo de ordenação quicksort, que utiliza o conceito de divisão e conquista para ordenar um vetor de itens do tipo ‘Item’. Ela recebe como parâmetros o vetor de itens ‘vetor[]’, o índice da posição esquerda ‘esq’ e o da posição direita ‘dir’. A função quicksort utiliza o particionamento do vetor para dividir o problema em subproblemas menores, aplicando o algoritmo de forma recursiva. A estratégia, dividir para conquistar, permite uma ordenação eficiente do vetor.

4.10 - void heapify (Item vetor[], int tamanho, int x)

A função ‘heapify’ é essencial para o algoritmo de ordenação heapsort para construir um heap máximo de um nó específico em um vetor de itens ‘vetor[]’. Ela recebe como parâmetros o vetor de itens ‘vetor[]’, o tamanho do heap ‘tamanho’ e o índice do nó ‘x’.

4.11 - void heapsort (Item vetor[], int tamanho)

A função ‘heapsort’ implementa o algoritmo de ordenação heapsort para ordenar um vetor de itens do tipo ‘Item’. Ela recebe como parâmetros o vetor de itens ‘vetor[]’ e o tamanho do vetor ‘tamanho’. A função heapsort utiliza a função heapify para construir um heap e ajustar os elementos durante o processo de extração do máximo (ou mínimo) do heap.

4.12 - void merge (Item vetor[], int inicio, int meio, int fim)

A função ‘merge’ é essencial para o algoritmo de ordenação mergesort para realizar a etapa de mesclagem de duas partes do vetor em ordem crescente. Ela recebe como parâmetros o vetor de itens ‘vetor[]’, o índice de início da primeira parte ‘inicio’, o índice do meio que divide as partes ‘meio’ e o índice de fim da segunda parte ‘fim’. Essa função realiza a etapa de combinação dos elementos ordenados e é chamada recursivamente pelo algoritmo mergesort para ordenar o vetor.

4.13 - void mergesort(Item vetor[], int inicio, int fim)

A função ‘mergesort’ implementa o algoritmo de ordenação mergesort para ordenar um vetor de itens do tipo ‘Item’. Ela recebe como parâmetros o vetor de itens ‘vetor[]’, o índice de início da região a ser ordenada ‘inicio’ e o índice de fim da região ‘fim’.

4.14 - void exibirVetor (Item vetor[], int tamanho)

A função ‘exibirVetor’ é responsável por exibir na tela os elementos de um vetor do tipo ‘Item’. Ela recebe como parâmetros o vetor de itens ‘vetor[]’ e o tamanho do vetor ‘tamanho’.

4.15 - void embaralharVetor (Item vetor[], int tamanho)

A função ‘embaralharVetor’ é responsável por embaralhar aleatoriamente os elementos de um vetor de itens do tipo ‘Item’. Ela recebe como parâmetros o vetor de itens ‘vetor[]’ e o tamanho do vetor ‘tamanho’. O algoritmo percorre o vetor do último elemento até o primeiro e, em cada iteração, seleciona aleatoriamente um elemento não embaralhado e o troca com o elemento da posição atual.

5 - Complexidade

5.1 - void trocar (Item *a, Item *b){

```
    Item temp = *a;
    * a = *b;
    *b = temp;
}
```

A complexidade dessa função é considerada de tempo constante, ou seja, O(1). Isso ocorre porque a troca de valores entre duas variáveis é uma operação simples que não depende do tamanho do vetor ou de outras variáveis. Portanto, a função ‘trocar’ é uma operação eficiente e rápida.

5.2 - void selecao (Item vetor[], int tamanho)

A complexidade dessa função é considerada de tempo quadrática, ou seja, $O(n^2)$, onde n é o tamanho do vetor. Isso ocorre porque são realizadas duas iterações aninhadas sobre o vetor, o que leva a um crescimento quadrático do tempo de execução à medida que o tamanho do vetor aumenta. Portanto, a função ‘selecao’ é menos eficiente em relação a outros algoritmos de ordenação.

5.3 - void insercao (Item vetor[], int tamanho)

A complexidade dessa função é considerada de tempo quadrática, ou seja, $O(n^2)$, onde n é o tamanho do vetor. No pior caso, quando o vetor está ordenado em ordem decrescente, o algoritmo realiza o número máximo de comparações e deslocamentos, levando à complexidade quadrática. Entretanto, a ordenação por inserção é eficiente para pequenos conjuntos de dados ou quando o vetor já está parcialmente ordenado. Há também a vantagem de ser um algoritmo de ordenação estável, preservando a ordem relativa de elementos com chaves iguais.

5.4 - void shellsort(Item vetor[], int tamanho)

A complexidade dessa função depende da sequência de espaçamentos utilizada. No pior caso, a complexidade é de O(n 2), mas em média, a complexidade pode ser melhorada para O(n log n).

5.5 - int particionar(Item vetor[], int esq, int dir)

A complexidade dessa função depende do tamanho do intervalo que está sendo particionado. No pior caso, em que o pivô escolhido é sempre o menor ou o maior elemento do intervalo, a função pode ter uma complexidade de tempo quadrática $O(n^2)$, onde n é o número de elementos no intervalo. Em média o particionamento do quicksort tem uma complexidade O(n), o que significa que é linear em relação ao número de elementos no intervalo. Isso ocorre porque, em média, o particionamento divide o intervalo em duas partes aproximadamente iguais.

5.6 - void quicksort (Item vetor[], int esq, int dir)

A complexidade dessa função varia dependendo do caso. No pior caso, em que o pivô escolhido é sempre o menor ou o maior elemento do intervalo, a complexidade é de $O(n^2)$, onde n é o número de elementos no vetor. No entanto, em média, a complexidade é O(n log n), tornando o quicksort um dos algoritmos de ordenação mais eficientes em média.

5.7 - void heapify (Item vetor[], int tamanho, int x)

A complexidade dessa função é O(log n), onde n é o tamanho do heap. Isso ocorre porque o ajuste de um elemento dentro do heap requer a troca de posições em uma árvore binária completa, cuja altura é logarítmica em relação ao número de elementos.

5.8 - void heapsort (Item vetor[], int tamanho)

A complexidade dessa função é O(n log n), onde n é o número de elementos no vetor. É um algoritmo eficiente para ordenação.

5.9 - void merge (Item vetor[], int inicio, int meio, int fim)

A complexidade dessa função é O(n), onde n é o número de elementos no intervalo a ser mesclado. Isso ocorre porque a função copia os elementos dos subvetores temporários para o vetor original em uma única passagem.

5.10 - void mergesort(Item vetor[], int inicio, int fim)

A complexidade dessa função é O(n log n), onde n é o número de elementos no vetor. Isso ocorre porque o vetor é dividido em log n camadas, e em cada camada, é feita uma mesclagem de todos os elementos, resultando em uma complexidade de tempo eficiente.

5.11 - void exibirVetor (Item vetor[], int tamanho)

A complexidade dessa função é de tamanho linear, ou seja, O(n), onde n é o tamanho do vetor. Isso ocorre porque a função precisa percorrer todos os elementos do vetor e exibi-los, o que requer uma quantidade de operações proporcionais ao tamanho do vetor.

5.12 - void embaralharVetor (Item vetor[], int tamanho)

A complexidade dessa função é de tamanho linear, ou seja, O(n), onde n é o tamanho do vetor. Isso ocorre porque a função tem que percorrer todos os elementos presentes no vetor e realizar a troca de posição entre os pares de elementos.

6 - Resultados

Os testes foram realizados no meu notebook pessoal:

- 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz
- 8gb de memória ram
- 256gb de armazenamento - SSD

Os resultados obtidos estão detalhados adiante com tabelas e descrição:

Os resultados inconclusivos ou que o programa não respondeu corretamente estão representados por '-'.

Os resultados do vetor de 200000 foi inconclusivo, visto que na roda dos testes o programa não respondeu, crashando assim que pedia para executar tal parte.

	VETOR DE 20	media de 10	
	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕES	1045	2945	4845
MOVIMENTAÇÕES	313	883	1453

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕES	5804	7039	4845
MOVIMENTAÇÕES	1814	3049	1453

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕES	1045	10237	4845
MOVIMENTAÇÕES	313	6247	1453

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕES	12505	14405	15661
MOVIMENTAÇÕES	8828	10748	11751

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕES	16991	18613	20161
MOVIMENTAÇÕES	13366	15499	17521

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕES	21198	21758	22411
MOVIMENTAÇÕES	18480	18480	18480

	VETOR DE 500	media de 10	
	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕES	686125	1933625	3181125
MOVIMENTAÇÕES	8233	23203	38173

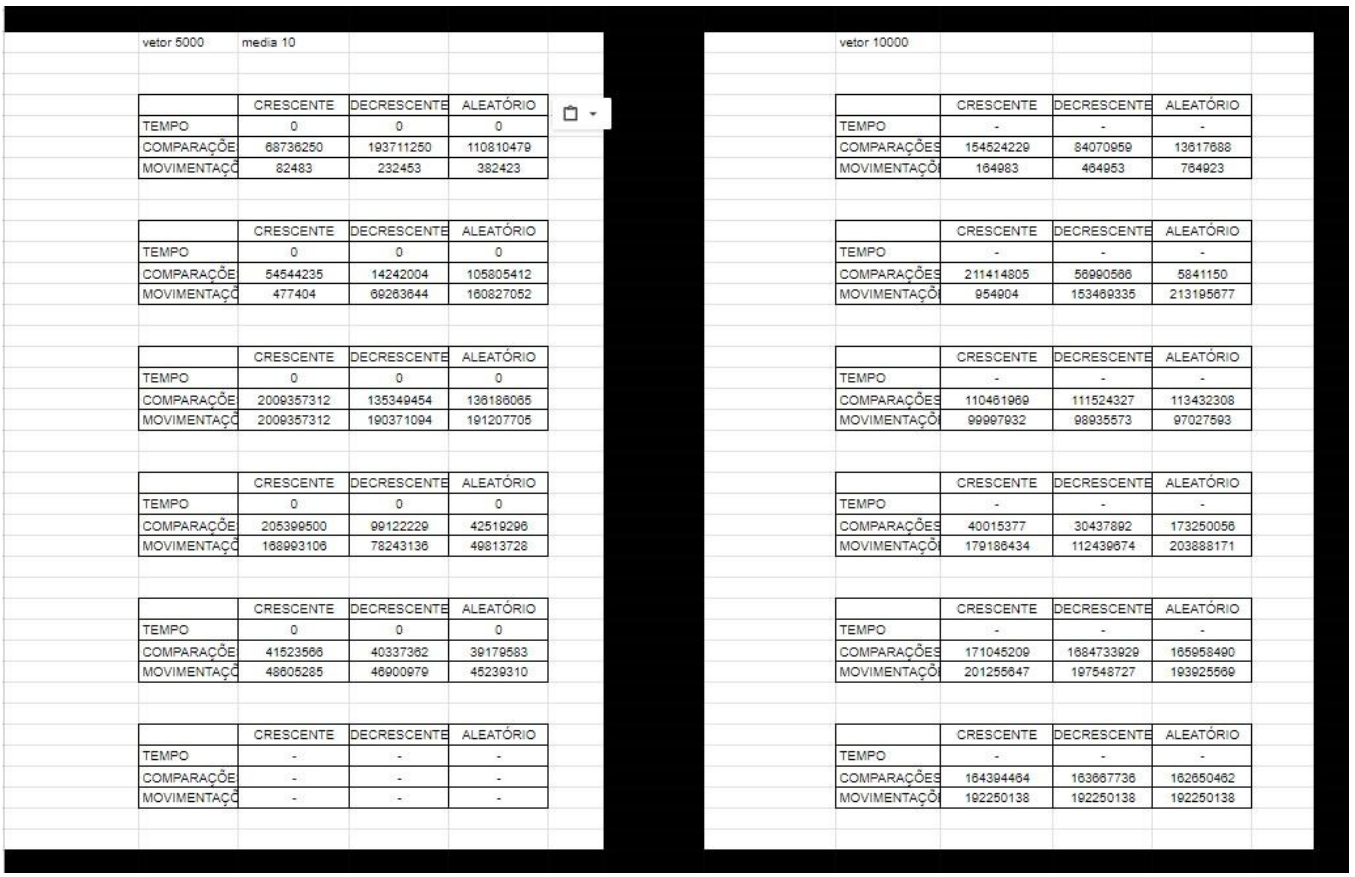
	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕES	3745244	4436359	5353206
MOVIMENTAÇÕES	47654	738769	1655616

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕES	5655690	-	5742120
MOVIMENTAÇÕES	1958100	-	2044530

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕES	6454845	7702345	8287903
MOVIMENTAÇÕES	2765488	3684208	3989198

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕES	8358345	8442987	8525267
MOVIMENTAÇÕES	4075887	4195350	4311270

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕES	8577508	8602492	8636132
MOVIMENTAÇÕES	4365500	4365500	4365500



vetor 5000 media 10

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕE	68736250	193711250	110810479
MOVIMENTAÇÕ	82483	232453	382423

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕE	54544235	14242004	105805412
MOVIMENTAÇÕ	477404	69263644	160827052

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕE	2009357312	135349454	136186065
MOVIMENTAÇÕ	2009357312	190371094	191207705

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕE	205399500	99122229	42519296
MOVIMENTAÇÕ	168993106	78243136	49813728

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	0	0	0
COMPARAÇÕE	41523566	40337362	39179583
MOVIMENTAÇÕ	48605285	46900979	45239310

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	-	-	-
COMPARAÇÕE	-	-	-
MOVIMENTAÇÕ	-	-	-

vetor 10000

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	-	-	-
COMPARAÇÕES	154524229	84070959	13617688
MOVIMENTAÇÕ	164983	464953	764923

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	-	-	-
COMPARAÇÕES	211414805	56990586	5841150
MOVIMENTAÇÕ	954904	153469335	213195677

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	-	-	-
COMPARAÇÕES	110461969	111524327	113432308
MOVIMENTAÇÕ	99997932	98935573	97027593

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	-	-	-
COMPARAÇÕES	40015377	30437892	173250056
MOVIMENTAÇÕ	179186434	112439674	203888171

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	-	-	-
COMPARAÇÕES	171045209	1684733929	165958490
MOVIMENTAÇÕ	201255647	197548727	193925569

	CRESCENTE	DECRESCENTE	ALEATÓRIO
TEMPO	-	-	-
COMPARAÇÕES	164394464	163667736	162650462
MOVIMENTAÇÕ	192250138	192250138	192250138

7 - Conclusão

O trabalho ajudou a praticar o conteúdo ensinado na disciplina, sobre ordenação interna, bem como aplicá-lo em um cenário real, como a ordenação de dados de pequeno e grande registros, que são muito utilizados em diversas áreas atualmente.

Tive dificuldades na implementação da função de inserir dados de forma automatizada em ordem decrescente e aleatória. Houve um pouco de dúvida também na implementação de um contador para contabilizar a quantidade de movimentações e trocas entre os elementos dentro dos vetores. Tive dificuldade também na hora de implementar os vetores de tamanho 200000, onde era necessário o uso de alocação de memória dinâmica. Ademais, consegui implementar todas as outras funções, consultando os slides disponibilizados pelo Portal Didático, assim como implementando os conhecimentos adquiridos durante as aulas.

Referências Bibliográficas:

Slides disponibilizados na plataforma Campus Virtual