SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE VIÇOSA · UFV
CAMPUS FLORESTAL

Trabalho 3 - AEDS 1

Ordenação de Rochas Minerais

Beatriz Queiroz Cruz Silva[5890]
Eduarda Anaely Suar Borges [5911]
Pâmela Lúcia Lara Diniz [5898]

Florestal - MG
2025

Sumário

1. Introdução

O assunto abordado nesta documentação é o desenvolvimento de um programa em C para ordenar uma lista de rochas minerais com base no peso, utilizando algoritmos de ordenação eficientes e conceitos de Tipos Abstratos de Dados (TADs). Atendendo às especificações do trabalho prático, foram escolhidos dois algoritmos de ordenação: um algoritmo simples, o Insertion Sort, escolhido por sua facilidade de implementação e entendimento, e um algoritmo mais avançado, o QuickSort, reconhecido por sua eficiência em grande escala. Além de ordenar os dados, o sistema coletará métricas de desempenho, como o número de comparações, movimentações realizadas e o tempo total de execução. Essas informações serão utilizadas para analisar e comparar as diferenças de desempenho entre os algoritmos implementados. Por fim, este relatório final contém uma apresentação detalhada dos resultados obtidos, discutindo as vantagens, desvantagens e as situações ideais de aplicação para cada técnica de ordenação. Com isso, busca-se aplicar na prática os conceitos de algoritmos e estruturas de dados, avaliando a eficiência e a aplicabilidade de diferentes métodos de ordenação em um contexto real.

2. Organização

O projeto foi estruturado de forma modular para garantir clareza e facilidade de manutenção. A organização geral do repositório é apresentada na Figura 1, destacando as principais pastas e seus respectivos conteúdos.

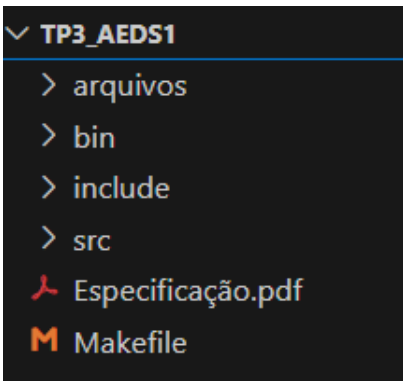

Figura 1 - Organização do repositório do projeto

A pasta **TP3_AEDS** contém toda a implementação do projeto, dividida em módulos que facilitam a separação das funcionalidades. Nela, também estão incluídos os arquivos que permitem a execução do programa.

Na pasta **include** e **src**, encontram-se respectivamente, os Tipos Abstratos de Dados (TADs) necessários para o trabalho e a implementação das funções utilizadas, incluindo os algoritmos de ordenação, estruturas que foram importantes para lidar com os dados de forma eficaz. O arquivo principal main.c também está localizado em src.

A pasta **arquivos** agrupa casos de entrada para a realização de testes.

Na pasta **bin** fica contido o executável gerado a partir da compilação do código.

3. Desenvolvimento

Nessa etapa iremos detalhar a implementação dos TADS e suas funções, com ênfase nos algoritmos de ordenação escolhidos pelo grupo, o Insertion sort e o Quicksort.

3.1 Mineral.c

O código implementa um TAD Mineral e funções que permitem a criação, configuração e acesso às características dos minerais. As funções são responsáveis por identificar o mineral com base no nome fornecido.

inicializa_mineral- faz a inicialização do mineral a partir de seu nome.

get_nome: retorna o valor armazenado no campo nome, localizado na estrutura mineral.

3.2 ListadeMinerais.c

Utilizando o TAD Mineral para gerenciar as informações dos minerais, o código implementa uma lista sequencial de minerais utilizando um array para gerenciar os elementos. A lista começa vazia e o código oferece funcionalidades para adicionar minerais à lista, verificar se a lista está cheia ou vazia e exibir todos os minerais armazenados.

inicializa_lista_minerais- Configura uma nova lista de minerais. Define os índices primeiro e prox_disponivel com o valor inicial, indicando que a lista está vazia.

verifica_l_vazia- Retorna 1 (verdadeiro) se a lista estiver vazia, isto é, quando os índices de primeiro e prox_disponivel forem iguais.

insere_mineral- Insere um mineral na lista quando há espaço. O mineral é adicionado na posição indicada por prox_disponivel, que é incrementado após a operação. Caso a lista esteja cheia, a função exibe uma mensagem informando a situação.

imprime_lista_minerais- Percorre a lista e exibe o nome de todos os minerais contidos nela utilizando a função get_nome, previamente definida no TAD Mineral.

3.3 Rocha.c

O código realiza manipulações de informações, implementando um sistema para gerenciar as rochas minerais, atribuindo a cada uma, uma categoria específica com base nos minerais que a compõem. A estrutura central, armazena informações como peso, coordenadas geográficas (latitude e longitude), a categoria e os minerais presentes nela; classifica a rocha em uma categoria específica e exibe suas informações.

Inicializa_Rocha- Inicializa uma rocha com seus atributos principais e associa uma lista de minerais e atribui os valores recebidos aos campos do TAD RochaMineral.

ImprimeRocha- Exibe os dados da rocha e imprime os minerais associados.

contem_mineral- Percorre a lista de minerais e compara o nome do mineral fornecido com os nomes armazenados para verificar se um mineral específico está presente na lista de minerais associada à rocha.

copia_Rocha- cria uma cópia de uma rocha, incluindo todas as suas propriedades e os minerais associados. Essa função foi feita para duplicar informações de uma rocha existente e garantir que a nova rocha tenha os mesmos dados, mas com gerenciamento independente.

setCategoria- Define a categoria da rocha com base nos minerais presentes na lista, usando combinações de minerais específicos para categorizar a rocha.

3.4 Compartimento.c

A estrutura do compartimento se baseia em gerenciar uma lista vetorial que armazena itens do tipo RochaMineral e conta com funções implementadas de acordo com as necessidades desse processo.

inicializa_compartimento- Responsável por configurar um compartimento de rochas vazio.
exibe_compartimento- Percorre a lista de rochas armazenadas e exibe as categorias e os pesos de cada rocha presente no compartimento.
insere_nova_rocha- Insere uma nova rocha no compartimento, alocando memória para uma nova célula e ajustando os ponteiros da lista para que a nova rocha se torne o último item.

Além disso, foram implementados no compartimento dois algoritmos responsáveis por fazer a ordenação das rochas armazenadas, o Insertionsort e o Quicksort.

3.5 Arquivo.c

Neste módulo estão contidas as implementações de funções necessárias à manipulação de arquivos de entrada.

abrearquivo: É a função responsável por fazer a abertura do arquivo informado, indicando erro em casos irregulares.

fechaArquivo: Realiza o fechamento do arquivo informado.

qtdoperacao: Faz a leitura da primeira linha do arquivo de entrada que deve ser um inteiro que representa a quantidade de elementos no vetor.

ler_rocha: Faz a leitura das linhas subsequentes, realizando a coleta de informações como latitude,longitude, peso e minerais através do uso de funcionalidades como *strtok* que realiza a diferenciação dos elementos pelo delimitador “espaçamento” e *buffer* para armazenamento das mesmas, que posteriormente são atribuídas às suas respectivas posições/variáveis.

3.6 Insertion sort

O Insertion Sort é um algoritmo de ordenação simples e intuitivo, frequentemente comparado à forma como organizamos cartas em uma mão durante um jogo. A ideia central é construir uma sequência ordenada, inserindo cada elemento em sua posição correta dentro dessa sequência em construção.

Como funciona o Insertion Sort?

1. **Divisão do Vetor:** O vetor a ser ordenado é dividido em duas partes:
    - Uma parte ordenada, inicialmente vazia.
    - Uma parte não ordenada, contendo todos os elementos.
2. **Inserção:**
    - O primeiro elemento da parte não ordenada é selecionado.
    - Esse elemento é comparado com os elementos da parte ordenada, movendo-os para a direita até encontrar a posição correta para inserção.
    - O elemento selecionado é inserido na posição encontrada.

Análise de Complexidade do Insertion Sort:

Pior Caso

- Ocorre quando: O vetor está ordenado em ordem decrescente.
- Comparações: Para cada elemento, são necessárias n-1 comparações no pior caso, pois ele precisará ser comparado com todos os elementos já ordenados.
- Movimentações: Para cada elemento, são necessárias n-1 movimentações no pior caso, pois todos os elementos à esquerda precisarão ser deslocados uma posição para a direita.
- Complexidade: $O(n^2)$, **tanto para comparações quanto para movimentações.**

Melhor Caso

- Ocorre quando: O vetor já está ordenado.
- Comparações: Para cada elemento, é necessária apenas 1 comparação, pois ele já está na posição correta.
- Movimentações: Não são necessárias movimentações.
- Complexidade: $O(n)$, **tanto para comparações quanto para movimentações.**

Caso Médio

- Ocorre quando: A ordem dos elementos é aleatória.

- **Complexidade: O(n²), tanto para comparações quanto para movimentações.**

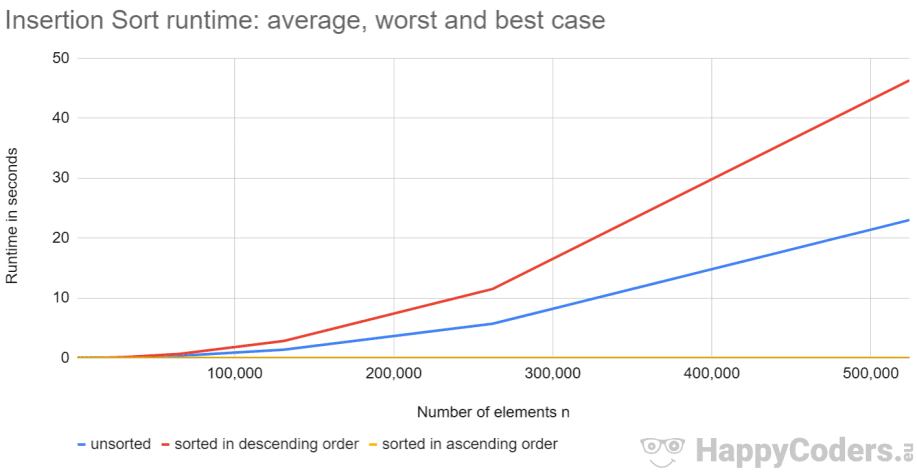

Figura 2 - Representação gráfica do comportamento do Insertionsort

O Insertion Sort é um algoritmo de ordenação simples e eficiente para listas pequenas ou quase ordenadas. No entanto, para listas grandes e desordenadas, algoritmos mais eficientes, como o QuickSort, são preferíveis.

Algoritmo

```c
void ordena_insertion(int n,lista_rochas *vetor,int *comp,int *trocas){
    int i,j;
    RochaMineral tmp;
    Inicializa_Rocha(&tmp,&vetor->arranjo_rochas[0].minerais,vetor->arranjo_rochas[0].peso,vetor->arranjo_rochas[0].latitude,vetor->arranjo_rochas[0].l
    for(i=1;i<n;i++){
        copia_Rocha(&tmp,&vetor->arranjo_rochas->minerais, vetor->arranjo_rochas[i].peso, vetor->arranjo_rochas[i].latitude, vetor->arranjo_rochas[i].l
        for(j = i-1;j >= 0;j--){
            (*comp)++; // Incrementa o contador de comparações
            if (tmp.peso < vetor->arranjo_rochas[j].peso){
            copia_Rocha(&vetor->arranjo_rochas[j + 1], &vetor->arranjo_rochas[j].minerais, vetor->arranjo_rochas[j].peso, vetor->arranjo_rochas[j].lati
            (*trocas)++; // Incrementa o contador de trocas
            }else{
                break;//encerra as comparacoes quando tmp>=a[j]
            }
        }
        // Insere o tmp na posição correta
        copia_Rocha(&vetor->arranjo_rochas[j + 1],&tmp.minerais, tmp.peso, tmp.latitude, tmp.longitude, tmp.categoria);
        (*trocas)++; // Incrementa o contador de trocas      You, há 1 segundo • Uncommitted changes
    }
```

Figura 3 - Implementação em linguagem C da função “ordena_insertion”

O código apresentado implementa o algoritmo de ordenação por inserção para ordenar um vetor de estruturas do tipo `RochaMineral`, ordenando-as pelo atributo `peso`. A função `copia_Rocha` é utilizada para copiar os dados de uma rocha para outra posição.A variável que contabiliza o número de comparações (comp) se encontra dentro do segundo laço de repetição pois é lá que tem o if que faz a comparação de peso entre duas rochas.A variável (trocas) armazena a quantidades de movimentações,ela está dentro do if onde se a condição for

verdadeira as rochas são “arrastadas” para a direita. E fora do segundo laço de repetição onde o copia_rocha tem a função de alocar o número armazenado em tmp para sua posição correta no vetor já ordenado.

3.7 Quicksort

O Quicksort é um dos algoritmos de ordenação mais eficientes e amplamente utilizados. Ele se baseia na estratégia de **dividir e conquistar.**Em média, o Quicksort tem complexidade temporal de O(n log n), o que o torna muito eficiente para grandes conjuntos de dados.

Como Funciona o Quicksort?

1. **Escolha do Pivô:**
    - Seleciona-se um elemento aleatório do vetor, chamado de pivô.
    - A escolha do pivô influencia significativamente o desempenho do algoritmo.
2. **Particionamento:**
    - Reorganiza-se o vetor de forma que todos os elementos menores que o pivô estejam à sua esquerda e todos os maiores, à sua direita.
    - O pivô ocupa sua posição final na sequência ordenada.
3. **Chamadas Recursivas:**
    - O Quicksort é aplicado recursivamente às sublistas à esquerda e à direita do pivô.
    - O processo se repete até que todas as sublistas tenham um único elemento (que já está ordenado).

Análise de Complexidade do Quicksort:

Melhor Caso

- **Ocorre quando:** O pivô divide o array em duas partes de tamanho aproximadamente igual a cada iteração.
- **Comparações:** A cada nível da recursão, são realizadas n comparações, onde n é o tamanho da sublista. A altura da árvore de recursão é $\log_2 n$. Portanto, o número total de comparações é aproximadamente $n \log_2 n$.
- **Movimentações:** O número de movimentações também é proporcional ao número de comparações, resultando em $O(n \log_2 n)$.
- **Complexidade:** $O(n \log_2 n)$ tanto para comparações quanto para movimentações.

Pior Caso

- **Ocorre quando:** O pivô é sempre o menor ou o maior elemento do array.

- **Comparações:** A cada nível da recursão, são realizadas n-1 comparações, e a altura da árvore de recursão é n. Portanto, o número total de comparações é aproximadamente n^2.
- **Movimentações:** O número de movimentações também é proporcional ao número de comparações, resultando em $O(n^2)$.
- **Complexidade:** $O(n^2)$ tanto para comparações quanto para movimentações.

Caso Médio

- **Ocorre quando:** A escolha do pivô é aleatória ou boa o suficiente para dividir o array em partes de tamanho aproximadamente igual.
- **Complexidade:** $O(n \log_2 n)$ em média, tanto para comparações quanto para movimentações.

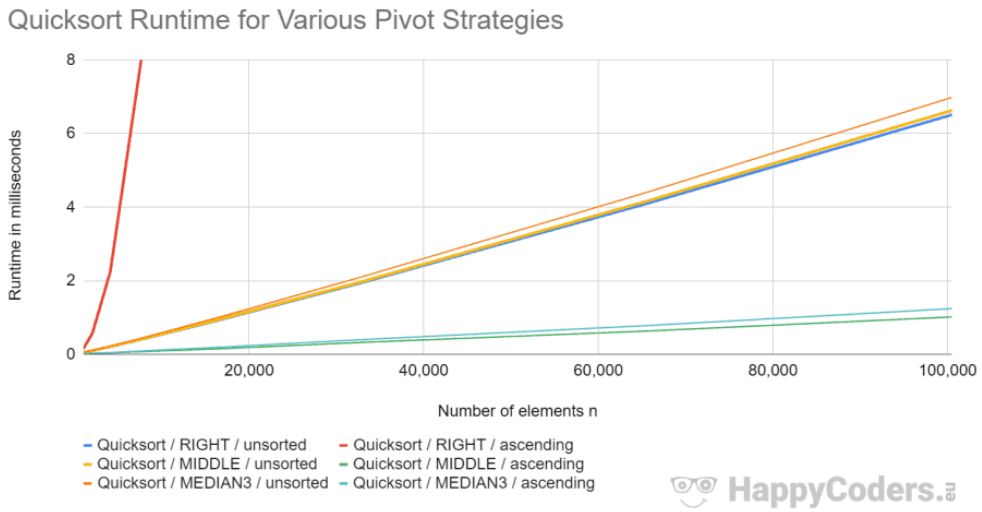

Figura 4 - Representação gráfica do comportamento do Quicksort

O Quicksort é um algoritmo eficiente em média, com complexidade $O(n \log_2 n)$. No entanto, seu desempenho pode se degradar para $O(n^2)$ no pior caso, quando a escolha do pivô é ruim. Para garantir um bom desempenho, é importante escolher uma estratégia de seleção de pivô adequada.

Algoritmo

A implementação do algoritmo quicksort se baseou nos conceitos já citados anteriormente. Em primeiro lugar, foi feita a implementação da função principal quicksort, ela recebe como parâmetros um vetor de rochas e a quantidade de elementos contidos nessa lista, em seguida são inicializadas as variáveis para contabilizar as comparações e movimentações realizadas e é feita a primeira

chamada da função ordena (figura x), que recebe como parâmetros o vetor, os índice do primeiro e último elementos, além das variáveis métricas.

```c
void ordena_quick(int esq, int dir, lista_rochas *vetor, int *comparacoes,int *trocas){
    int i,j;
    particao(esq,dir,&i,&j,vetor,comparacoes,trocas);
    if (esq<j) {ordena_quick(esq,j,vetor,comparacoes,trocas);}

    if (i<dir) {ordena_quick(i,dir,vetor,comparacoes,trocas);}
}

void quicksort (lista_rochas *vetor, int n){
    int comparacoes=0;
    int trocas=0;
    ordena_quick(0,n-1,vetor,&comparacoes,&trocas);

    /*for (int k=0;k<n;k++){
        printf("%s %.1f\n",vetor->arranjo_rochas[k].categoria,vetor->arranjo_rochas[k].peso);
    }*/
    printf("\nComparacoes: %d\n",comparacoes);
    printf("Movimentacoes: %d\n",trocas);
    printf("Algoritmo: Quicksort\n");

}
```

Figura 5 - Implementação da função “ordena_quick”, referente ao quicksort

Tal funcionalidade é responsável por fazer chamadas recursivas que buscam a função “particao” (figura 6), esta que é encarregada de realizar a criação de sub listas, determinando em cada uma delas um elemento para ser considerado o pivô. Depois disso faz a organização dos elementos de modo que os menores fiquem localizados à esquerda do pivô e os maiores à sua direita.

```c
void particao(int esq, int dir, int *i,int *j, lista_rochas *vetor, int *comparacoes, int *trocas){
    Inicializa_Rocha(&aux,&vetor->arranjo_rochas[0].minerais,vetor->arranjo_rochas[0].peso,vetor->arranjo_rochas[0].latitude,vetor->arranjo_rochas[0].longitude);
    *i = esq; *j = dir;
    pivo = vetor->arranjo_rochas[(*i + *j) / 2];
    do{
        while (pivo.peso > vetor->arranjo_rochas[*i].peso){
            (*i)++;
            (*comparacoes)++;
            }
        while (pivo.peso < vetor->arranjo_rochas[*j].peso){
            (*j)--;
            (*comparacoes)++;
            }
        if (*i <= *j){
            copia_Rocha(&aux,&vetor->arranjo_rochas[*i].minerais,vetor->arranjo_rochas[*i].peso,vetor->arranjo_rochas[*i].latitude,vetor->arranjo_rochas[*i].longi
            copia_Rocha(&vetor->arranjo_rochas[*i],&vetor->arranjo_rochas[*j].minerais,vetor->arranjo_rochas[*j].peso,vetor->arranjo_rochas[*j].latitude,vetor->ar
            copia_Rocha(&vetor->arranjo_rochas[*j],&aux.minerais,aux.peso,aux.latitude,aux.longitude,aux.categoria);

            (*trocas)++;

            (*i)++;
            (*j)--;
        }
    } while (*i<= *j);
}
```

Figura 6 - Implementação da função “particao”, referente ao quicksort

Ademais, enquanto o bloco de execução “do while” realiza esse procedimento de organização, ele também atualiza a contagem de comparações feitas pelo código sempre que é comparado o peso de dois elementos. Já a contabilização das movimentações, representada pela variável “trocas”, é acrescida em uma unidade quando um elemento do vetor é trocado de lugar com outro.

Por fim, após a execução da partição, o algoritmo chama recursivamente a função “ordena” para organizar os elementos à esquerda e à direita de cada novo pivô. Esse processo de particionamento continua sendo repetido até chegar em uma partição que tenha um único elemento ou nenhum, indicando que o vetor está ordenado.

3.8 Main.c

Por fim, foi implementado o módulo principal que realiza a conexão entre todas as funcionalidades estabelecidas anteriormente.

A priori, foi decidido que o sistema só receberia entradas via arquivos pois seriam tratados majoritariamente casos de teste muito grandes, que seriam inviáveis de serem informados através de um menu interativo. Além disso, também optamos por limitar a execução do programa a uma ordenação por vez, independente do algoritmo selecionado. Portanto, para realizar sucessivas ordenações é necessário realizar a execução do programa novamente.

Com isso estabelecido, foi feita a implementação do main. Primeiramente foram declaradas algumas variáveis necessárias ao funcionamento do programa, como compartimento, arquivo, quantidade de rochas,etc. Em seguida foi configurado o recebimento do arquivo através da chamada de funções implementadas em Arquivo.c para registro das rochas e sua inserção no compartimento. E finalmente, uma função switch case foi utilizada para fazer a chamada do algoritmo de ordenação escolhido pelo usuário através do terminal.

4. Compilação e execução

Certifique-se de ter um compilador C instalado, como o **gcc**, e o **Make** disponível em seu sistema. Caso esteja usando o Windows, pode ser necessário instalar o **MinGW** e garantir que o **mingw32-make** esteja acessível no terminal.No terminal, navegue até a pasta principal do projeto, onde está localizado o arquivo Makefile. Execute o comando “mingw32-make compile” para compilar o programa utilizando o arquivo Makefile fornecido. Esse comando compila todos os módulos necessários e gera o executável. Após a compilação, execute o programa com comando “./main.exe” .

5. Resultados

Uma vez implementados os algoritmos de ordenação e as funcionalidades referentes ao processamento das rochas, foi possível realizar a execução do programa e posteriormente analisar o seu comportamento em casos diversos. As

principais métricas utilizadas foram a quantidade de comparações e movimentações realizadas, além da medição do tempo de processamento.

5.1 Quicksort

No que se refere ao Quicksort, a tabela a seguir demonstra os resultados obtidos a partir de sua aplicação em um compartimento desordenado com 250, 1000, 10000 e 50000 rochas, para situações em que o pivô escolhido é sempre o elemento do meio da partição.

Pivô - meio	250 rochas (1)	1000 rochas (2)	10000 rochas (3)	50000 rochas (4)
Comparações	1102	4094	46102	206224
Movimentações	650	3452	50219	310127
Tempo de execução	0s	0s	0,1s	0,3s

Em relação aos testes um e dois, observa-se que o número de rochas aumentou em 4 vezes e, de maneira análoga, a quantidade de comparações realizadas acompanhou esse crescimento, com um pequeno aumento. Já para as movimentações, houve uma diferença um pouco maior, sendo necessário 5 vezes mais trocas.

Seguindo para o terceiro caso de teste, em que é atribuído um aumento de dez vezes à quantidade anterior de rochas, os resultados da execução mostram que as comparações foram multiplicadas em onze vezes, e para as movimentações o número foi aproximadamente quatorze vezes maior.

Além disso, um teste extra foi realizado com uma entrada de 50000 rochas com o objetivo de testar o desempenho do programa quanto ao processamento de uma maior quantidade de itens, e a proporção de aumento das métricas seguiu o padrão já observado nos testes anteriores e o computador utilizado não demonstrou dificuldades em executar rapidamente o programa.

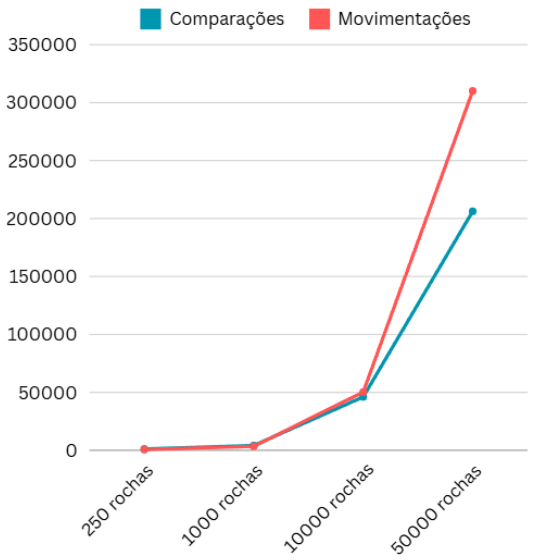

Figura 7 - Quantidade de operações realizadas pelo quicksort ref. aos testes realizados

Diante dos dados obtidos nessa análise, observa-se que embora as comparações e as movimentações cresçam de maneira semelhante, o número de trocas aumenta um pouco mais rápido à medida que a quantidade de rochas cresce, como representado pelo gráfico da figura 7, seguindo uma complexidade de aproximadamente n log n .

Ademais, apesar das diferenças apontadas entre a quantidade de comparações realizadas em cada teste, o tempo de execução do programa não apresentou variações consideráveis. Isso pode ser explicado pelo fato de que as entradas ainda são pequenas o suficiente para que o algoritmo consiga processá-las sem um impacto direto no desempenho ou que a ordenação inicial do vetor proporcionou um caso tratável de forma eficiente.

Por fim, comparações foram feitas de acordo com a escolha de diferentes pivôs. A tabela abaixo e a figura 8 representam o resultado para os mesmos testes, quando processados com o pivô escolhido sendo o mais à direita da partição.

Pivô - direita	250 rochas (1)	1000 rochas (2)	10000 rochas (3)	50000 rochas (4)
Comparações	1092	4489	39938	223566
Movimentações	632	3461	50657	308701

Tempo de execução	0s	0s	0,01s	0,03s

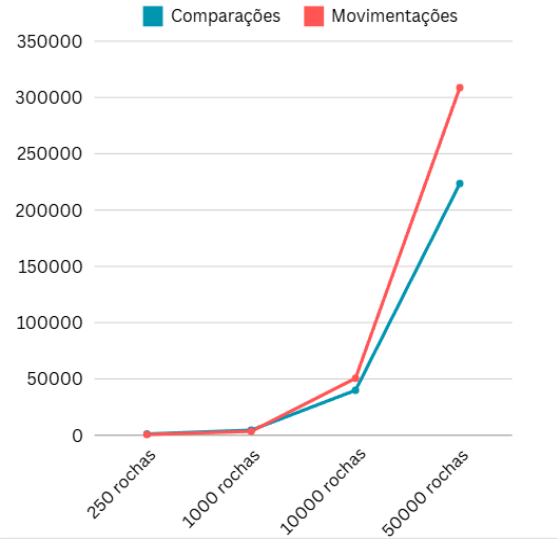

Figura 8 - *Quantidade de operações realizadas pelo quicksort com o pivô à extrema-direita*

Já a próxima tabela, junto da figura 9, retratam a situação quando o pivô escolhido foi o elemento mais à esquerda.

Pivô - esquerda	250 rochas (1)	1000 rochas (2)	10000 rochas (3)	50000 rochas (4)
Comparações	1098	3850	44163	226247
Movimentações	653	3492	50341	310891
Tempo de execução	0s	0s	0,01s	0,04s

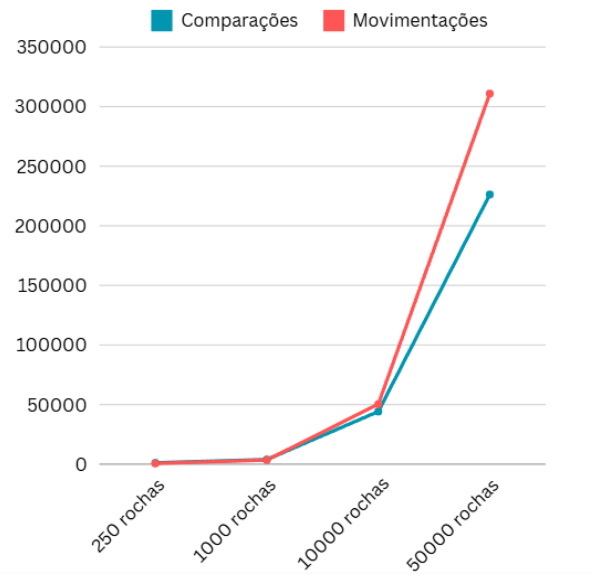

Figura 9 - *Quantidade de operações realizadas pelo quicksort com o pivô à extrema-esquerda*

A seguir pode-se observar através das figuras 10 e 11, a comparação direta do comportamento do quicksort para os diferentes pivôs a partir das mesmas entradas.

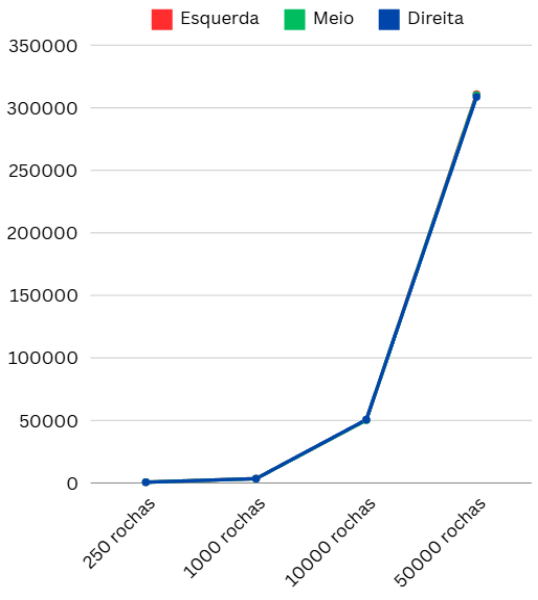

Figura 10 - *Movimentações (valores se sobrepõem)*

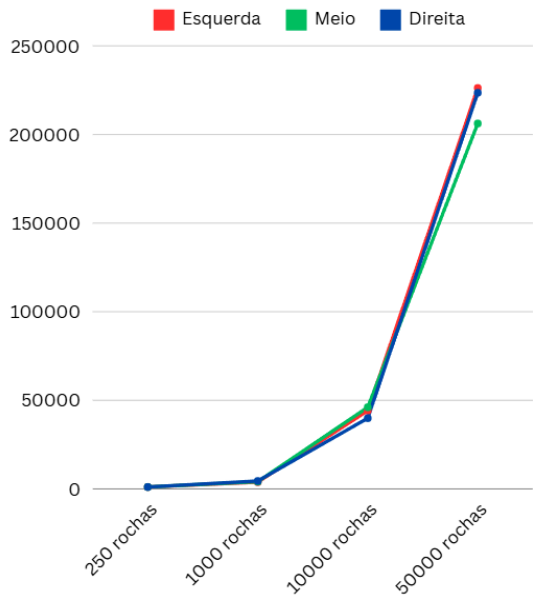

Figura 11 - *Comparações*

Perante os resultados obtidos, concluiu-se que a escolha do pivô influenciou a quantidade de operações realizadas pelo algoritmo e dentre as situações de

entrada analisadas, a opção do elemento à direita como pivô apresentou menos comparações para a entrada de 10000 rochas, já com a escolha do item do meio como pivô, foi obtido um menor número de comparações no caso de 50000 rochas, indicando se tratar de uma melhor opção para entradas maiores.

Destarte, pode-se concluir que o algoritmo Quicksort se comportou muito bem nos testes fornecidos, resolvendo o problema de ordenação de maneira eficiente e não alcançando sua situação de pior caso, caracterizando-se como uma boa opção de uso.

5.2 Insertionsort

Posteriormente, os mesmos testes realizados pelo quicksort foram reproduzidos em um algoritmo de ordenação simples, o insertionsort, e a partir dos resultados obtidos, um comportamento diferente pôde ser observado.

	250 rochas (1)	1000 rochas (2)	10000 rochas (3)	50000 rochas (1)
Movimentações	15616	241898	24121227	600326554
Comparações	15620	241896	24121224	600326559
Tempo execução	0s	0,1s	0,74s	18,4s

Diante das métricas observadas a partir da execução do algoritmo e apresentadas na tabela acima, nota-se que houve um aumento muito grande na quantidade de operações realizadas. Utilizando-se dos mesmos arquivos de teste executados anteriormente, o teste 2 contou com um acréscimo de 4 vezes na quantidade de elementos do vetor, e isso causou um crescimento de 15 vezes no número de operações realizadas, em relação ao teste 1. Já quanto aos testes 2 e 3, os elementos no vetor aumentaram em dez vezes, o que culminou na quantidade de comparações e movimentações sendo 100 vezes maiores. A respeito do tempo de execução, houve um aumento até então irrelevante.

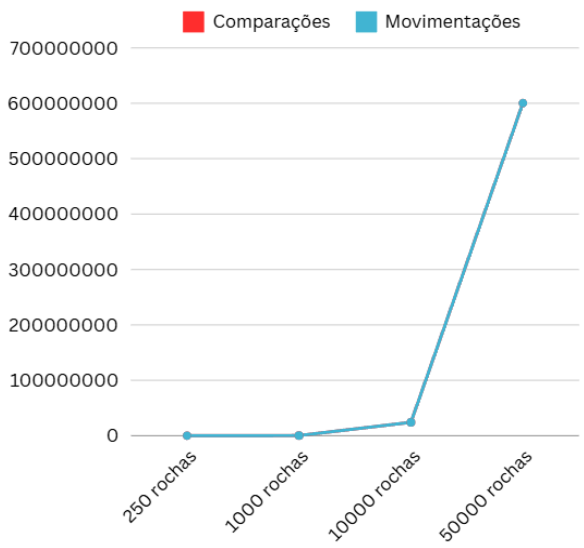

Figura 12 - Quantidade de operações realizadas pelo insertionsort ref. aos testes realizados

Além disso, a entrada de 50000 rochas também foi ordenada pelo insertionsort, e esse procedimento representou o primeiro indício de impacto no desempenho durante a execução, haja vista que foi o teste que mais demorou a ser processado por algum dos algoritmos implementados, levando um tempo de 18,4 segundos para entregar o resultado final.

Por fim, com a contabilização de todos os resultados, observou-se que houve um crescimento exponencial O(n^2) na quantidade de comparações e movimentações realizadas à medida que as entradas aumentavam, o que indica a ocorrência do pior caso dentre os possíveis para o insertionsort, com elevado custo computacional. Assim, conclui-se que apesar de entregar resultados satisfatórios para entradas de vetores pequenos, o insertion enfrenta dificuldades para ordenar entradas muito grandes quando o posicionamento inicial dos elementos não é favorável, devendo ser desconsiderado como primeira opção de método, uma vez que outro algoritmo (quicksort) se mostrou mais eficaz para a situação.

6. Conclusão

A implementação mostrou a prática dos conceitos de Tipos Abstratos de Dados (TADs) e técnicas de ordenação, permitindo uma análise detalhada do desempenho das abordagens selecionadas.

O **Insertion Sort** apresentou bons resultados em listas pequenas ou quase ordenadas, confirmando sua eficiência para esses casos simplificados. No entanto, seu desempenho foi inferior ao lidar com grandes volumes de dados devido à sua complexidade.

Por outro lado, o **QuickSort** destacou-se como a abordagem mais eficiente, especialmente em listas de grande volume e com elementos distribuídos de forma aleatória. A complexidade média e a eficiência nas movimentações e comparações determinaram o desempenho superior comparado com o Insertion sort. Porém, como seu algoritmo é dependente da escolha do pivô, seu resultado pode ser afetado negativamente.

Conclui-se que foram demonstrados os resultados esperados, confirmando todas as desvantagens e as vantagens de cada método em diferentes contextos. Portanto, a combinação de algoritmos simples, juntamente com o sofisticado, permite uma comparação eficaz e justa para o problema de ordenação, o que também destaca a importância de selecionar o algoritmo baseado nas características dos dados. Sendo assim, fica evidente o papel de algoritmos de ordenação para o tratamento de determinados projetos e destaca-se a importância da eficácia e adequação de acordo com os problemas.

7. Referências

[1] Github. Disponível em: < https://github.com/pamyd05/TP3_AEDS1>

[2] IDE Visual Studio Code. Disponível em: <https://code.visualstudio.com/>

[3] HappyCoders.eu. Disponível em:<https://www.happycoders.eu/>

[4] Projeto de Algoritmos | Nivio Ziviani» implementações. Ufmg.br. Disponível em: <https://www2.dcc.ufmg.br/livros/algoritmos/implementacoes-05.php>

[5] Notas de aula.