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

Trabalho Prático 3 - AEDS 1

Algoritmos de ordenação de rochas minerais

HEITOR PORTO JARDIM DE OLIVEIRA - 5895
JÚLIO CESAR DE SOUZA OLIVEIRA - 5903
OTÁVIO FRANCISCO SABINO TAVARES - 5912

Florestal - MG
2024

Sumário

1. Introdução

O presente projeto visa a avaliação e comparação de desempenho de algoritmos de ordenação a partir da medida de tempo de execução, número de movimentações feitas no vetor e número de comparações entre os elementos. Para tal, são usadas listas de diferentes tamanhos contendo rochas minerais a serem ordenadas de acordo com o peso em ordem crescente. Assim, deve-se ordenar as rochas presentes na lista informada usando um algoritmo simples de ordenação (neste caso, o Insertion Sort), e um sofisticado (neste caso, o Quick Sort), ambos implementados com base em Ziviani, 1999[1], registrando os números de movimentações e de comparações feitas nas ordenações, bem como o tempo de execução de cada uma, de maneira a comparar o desempenho de tais algoritmos.

A abordagem escolhida no desenvolvimento se baseia na inicial importação dos TADs utilizados no Trabalho Prático 1 (mineral, rocha mineral, lista de minerais e compartimento, alterando apenas a estrutura do último, que passa de lista encadeada para vetor) e demais elementos em comum (leitura de arquivos e medida de tempo de execução, por exemplo). Em seguida, iniciou-se o desenvolvimento do arquivo 'main.c', e a adaptação dos algoritmos de ordenação escolhidos.

2. Organização

A organização do projeto é representada na Figura 2 e se dá da seguinte forma: na pasta **src/**, se encontra a implementação do projeto, separada em módulos: uma pasta para cada TAD (com seus respectivos arquivos .h e .c), e o arquivo main.c. Na pasta **input/** estão os arquivos de entrada para teste.

Além disso, está presente no repositório o arquivo **Makefile** (Figura 1) construído com base no material complementar disponibilizado pelos monitores, de forma a compilar os múltiplos arquivos de diferentes pastas:

```
1   compile: src/main.c src/Compartimento/Compartimento.c src/Mineral/Mineral.c src/ListaMinerais/ListaMinerais.c
    src/RochaMineral/RochaMineral.c
2           gcc src/main.c src/Compartimento/Compartimento.c src/Mineral/Mineral.c src/ListaMinerais/ListaMinerais.c
    src/RochaMineral/RochaMineral.c -Wall -Wextra -g -o main
```

Figura 1 - Arquivo Makefile.

[1] ZIVIANI, Nivio. Projeto de algoritmos com implementações em Pascal e C. 4ª Edição, São Paulo. Editora Pioneira, 1999.

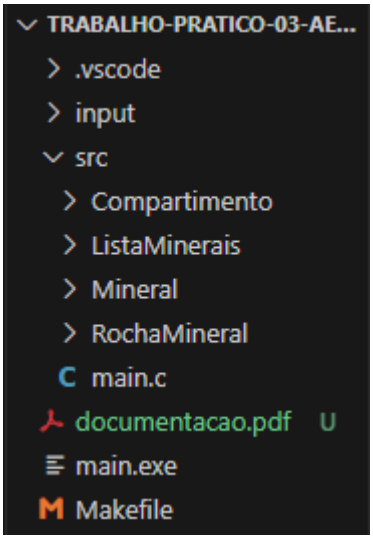

Figura 2 - Repositório do projeto.

3. Desenvolvimento

Observemos, a partir de módulos, os pontos mais importantes do projeto.

3.1 TAD Mineral

No TAD correspondente ao mineral, é preciso citar a função ‘atribui_mineral’ (Figura 3). Nela, um mineral é inicializado de acordo apenas com o nome, uma vez que a partir dele sabe-se as outras características (cor, dureza e reatividade). É com isso em mente que se compara a string correspondente ao nome do mineral que vai ser inicializado (recebido como parâmetro) com os nomes de mineral abrangidos pelo projeto, identificando o mineral em questão e atribuindo as características referentes ao mineral com esse nome. Com o fim das comparações e com o mineral devidamente atribuído, inicializa-se o mineral a partir da função ‘inicializa_mineral’, que, como representado na Figura 4, apenas atribui os valores de nome, cor, dureza e reatividade recebidos como parâmetro para o mineral também recebido como parâmetro.

```c
void atribui_mineral(Mineral * mineral, char nome_mineral[]){//coloca todas as informaçoes em um mineral
                                                             //de acordo com o nome recebido
    char cor[20];
    float dureza;
    float reatividade;

    if (strcmp(nome_mineral, "Ferrolita")==0){
        strcpy(cor,"cinza");
        dureza =0.5;
        reatividade =0.7;
    } else if (strcmp(nome_mineral , "Solarium")==0){
        strcpy(cor,"amarelo");
        dureza = 0.9;
        reatividade =0.2;
    }else if (strcmp(nome_mineral , "Aquavitae")==0){
        strcpy(cor,"azul");
        dureza =0.5;
        reatividade =0.8;

    } else if (strcmp(nome_mineral , "Terranita")==0){
        strcpy(cor,"marrom");
        dureza =0.7;
        reatividade =0.6;
    } else if (strcmp(nome_mineral ,"Calaris")==0){
        strcpy(cor,"vermelho");
        dureza =0.6;
        reatividade =0.5;
    }

    inicializa_mineral(mineral,nome_mineral, cor, dureza,reatividade);
}
```

Figura 3 - Função ’atribuiMineral’.

```c
//inicializa um mineral pegando todas as informaçoes
void inicializa_mineral(Mineral * mineral, char nome[], char cor[], float dureza, float reatividade){
    set_nome(mineral, nome);
    set_cor(mineral, cor);
    set_dureza(mineral, dureza);
    set_reatividade(mineral, reatividade);
}
```

Figura 4 - Função ’inicializa_mineral’.

3.2 TAD Lista de minerais

No TAD ‘ListaMinerais’, deve-se destacar a função ‘insereMineralLista’ (Figura 5), pois, deste TAD, é a mais utilizada ao longo do projeto. A função recebe uma lista de minerais e um item a ser inserido. Inicialmente, verifica-se se a lista está cheia, ou seja, se o tamanho máximo (3) é igual à posição para a qual o apontador ‘Ultimo’ se refere. Se tal condição for verdadeira, o item não é inserido na lista e a função retorna 0, indicando que não foi possível realizar a inserção. Caso contrário, o item é adicionado na posição para a qual o apontador ‘Ultimo’ se refere, ou seja, o item é adicionado na próxima posição disponível. Com isso, o apontador ‘Ultimo’ é incrementado, pois aquela que era a próxima posição disponível está ocupada, e a posição seguinte será a próxima disponível. Após isso, o valor retornado é 1, declarando que a inserção foi concluída e a função termina.

```c
//função que insere um novo mineral na lista
int insereMineralLista(ListaMinerais* pLista, TItem x){
    //certificando que a lista não está cheia - tamanho 3 ou maior
    if (pLista->Ultimo == MaxTam){
        //se a lista estiver cheia, não é possível adicionar o mineral
        return 0;
    }
    //adicionando o mineral x na última posição disponível
    pLista->minerais[pLista->Ultimo] = x;

    //incrementando o Apontador Ultimo, já que temos mais um elemento na lista
    pLista->Ultimo++;

    //inserção concluída
    return 1;
}
```

Figura 5 - Função 'insereMineralLista'.

3.3 TAD Rocha Mineral

Neste TAD, a função 'escolheCategoria' merece destaque. No primeiro processo da função (Figura 6.1), recebe-se uma rocha mineral como parâmetro e é analisada a lista de minerais que ela contém, a partir de variáveis booleanas (valor 0 ou 1) que indicam se os minerais, individualmente, existem na lista. Isso é feito a partir do nome dos minerais da lista em um laço de repetição que passa por todos os minerais da lista. As variáveis são declaradas como 0, mas se o nome do mineral 'm' atual corresponder à Ferrolita, por exemplo, a variável 'tFerrolita' passará a ser 1, indicando que a lista de minerais da rocha em questão possui Ferrolita. Após verificar todos os minerais da lista, sabe-se quais minerais compõem a lista.

```c
//função que define a categoria da rocha a partir da lista de minerais
void escolheCategoria(RochaMineral *rocha){

    //definindo variáveis booleanas para verificar se existe cada mineral na lista
    int tFerrolita = 0, tSolarium = 0, tAquavitae = 0, tTerranita = 0, tCalaris = 0;
    //imprimeListaMinerais(rocha->lista);
    //passando por todas as posições da lista de minerais
    for(int i = rocha->lista->Primeiro; i < rocha->lista->Ultimo; i++){
        TChave m = rocha->lista->minerais[i].Chave;
        //printf("%s\n", m.nome);
        //verificando, a partir do nome do mineral, qual mineral está na posição i
        if(strcmp(m.nome, "Ferrolita") == 0){
            tFerrolita = 1;
        }else if(strcmp(m.nome, "Solarium") == 0){
            tSolarium = 1;
        }else if(strcmp(m.nome, "Aquavitae") == 0){
            tAquavitae = 1;
        }else if(strcmp(m.nome, "Terranita") == 0){
            tTerranita = 1;
        }else if (strcmp(m.nome, "Calaris") == 0){
            tCalaris = 1;
        }
    }
```

Figura 6.1 - Primeiro processo da função 'escolheCategoria'.

No segundo processo (Figura 6.2), atribui-se a categoria à rocha do parâmetro a partir das definições de categoria com base na lista de minerais: se a lista possui Aquavitae, Calaris e Ferrolita, então a categoria da rocha é Aquacalis, por exemplo (primeiro 'if'). Essa é a lógica seguida nos 'if' para definir a categoria correta às rochas. A função 'escolheCategoria' é imprescindível para inicializar corretamente as rochas minerais, definindo a categoria a partir apenas da lista de minerais.

```c
void escolheCategoria(RochaMineral *rocha){
    //definindo a categoria da rocha de acordo com os minerais que estão presentes na lista
    if(tAquavitae && tCalaris && tFerrolita){
        setCategoria(rocha, "AQUACALIS");
    }else if(tTerranita && tFerrolita){
        setCategoria(rocha, "TERRALIS");

    }else if(tSolarium && tFerrolita){
        setCategoria(rocha, "SOLARISFER");

    }else if(tCalaris && tAquavitae){
        setCategoria(rocha, "CALQUER");

    }else if(tAquavitae && tTerranita){
        setCategoria(rocha, "AQUATERRA");

    }else if(tTerranita && tSolarium){
        setCategoria(rocha, "TERRASOL");

    }else if(tTerranita && tCalaris){
        setCategoria(rocha, "TERROLIS");

    }else if (tFerrolita && tAquavitae){
        setCategoria(rocha,"AQUAFERRO");

    }else if (tSolarium){
        setCategoria(rocha, "SOLARIS");

    }else if (tFerrolita){
        setCategoria(rocha, "FERROM");
    }
```

Figura 6.2 - Segundo processo da função ’escolheCategoria’.

3.4 TAD Compartimento e algoritmos de ordenação

Observemos, inicialmente, a ordenação por inserção (Figura 7). Basicamente, a função funciona da seguinte forma: o segundo elemento da lista é comparado com o primeiro, de maneira que, se o segundo for menor, é colocado na posição anterior. O processo é repetido para cada elemento seguinte, de forma a inseri-lo na posição correta dos elementos já ordenados, após compará-lo com eles. Assim, é feita a inserção de cada elemento de uma lista não ordenada em sua posição correta em uma parte ordenada da lista; é como organizar as cartas de uma mão de baralho, dividindo as cartas entre as cartas ordenadas e as cartas não ordenadas, sendo escolhida uma carta do grupo não ordenado a ser colocada no lugar certo no grupo ordenado.

```c
void insertion_sort(Compartimento* compartimento, int* comparacoes, int* movimentacoes){
    int n = compartimento->ultimo;

    for(int i = 1; i < n; i++){
        RochaMineral chave = compartimento->rochas[i];

        int j = i - 1;

        *comparacoes +=1;
        while ((j >= 0) && (compartimento->rochas[j].peso > chave.peso)){
            compartimento->rochas[j+1] = compartimento->rochas[j];
            *movimentacoes +=1;
            j--;
        }
        compartimento->rochas[j+1] = chave;
        *movimentacoes += 1;

    }
}
```

Figura 7 – Implementação do InsertionSort.

A função tem como parâmetros: o compartimento que possui a lista a ser ordenada, e os números de comparações e movimentações a serem contabilizados. É feito um laço de repetição que se inicia com ‘i = 1’, ou seja, índice 1 (segunda posição) do vetor e vai até o último índice. Dentro do ‘for’, tem-se um elemento chave, a rocha na posição ‘i’ a ser inserida na posição correta da parte ordenada da lista. Além disso, temos ‘j’, que corresponde à posição anterior ao ‘i’. Dessa forma, se inicia um ‘while’ que será executado enquanto ‘j’ (que é decrementado a cada execução) for maior ou igual a 0, e o peso da rocha no índice ‘j’ for maior que o peso da rocha chave, ou seja, da posição ‘i’. O código compara o peso da rocha chave com o peso do elemento anterior a ela e, enquanto a rocha à esquerda for maior que a chave, o elemento à esquerda é movido para a direita. Cada vez que um elemento é movido para a direita, o contador de movimentações é incrementado. Após o ‘while’, a rocha chave é inserida na posição correta da parte ordenada da lista.

O QuickSort, por sua vez, é dividido em duas partes: a de partição (Figura 8.1) e a de ordenação (Figura 8.2). A primeira consiste na inicial escolha de um pivô e em uma organização prévia do vetor, de maneira que os elementos menores que o pivô estejam à sua esquerda, e os maiores, à sua direita.

```c
void particao_quick(Compartimento*compartimento, int esq, int dir, int*i, int*j, int * comparacoes, int * movimentacoes){
    RochaMineral pivo, aux;
    *i = esq; *j = dir;
    pivo = compartimento->rochas[(*i + *j)/2];

    do{
        while(pivo.peso > compartimento->rochas[*i].peso){
            *comparacoes +=1;
            (*i)++;
        }
        *comparacoes+=1;
        while(pivo.peso < compartimento->rochas[*j].peso){
            *comparacoes +=1;
            (*j)--;
        }
        *comparacoes+=1;
        if(*i<=*j){
            aux = compartimento->rochas[*i];
            compartimento->rochas[*i] = compartimento->rochas[*j];
            compartimento->rochas[*j] = aux;
            *movimentacoes+=1;
            (*i)++; (*j)--;
        }
        *comparacoes +=2;
    }while(*i <=*j);
}
```

Figura 8.1 - Processo de partição do QuickSort.

Nessa etapa, as variáveis ‘i’ e ‘j’ são inicializadas com os índices ‘esq’ (início) e ‘dir’ (final) da parte do vetor que está sendo particionada.

O pivô é inicializado com o elemento central da parte do vetor que estamos analisando, ou seja, o elemento na posição (esq+dir) / 2. O laço ‘do while’ é usado para mover ‘i’ e ‘j’ até que eles se cruzem, sendo comparados com o pivô e, se necessário, trocados: 'i' é movido para a direita enquanto o peso do elemento na em tal posição for menor que o peso do pivô. A cada comparação, o contador de comparações é incrementado; 'j' é movido para a esquerda enquanto o peso do elemento em tal posição for maior que o peso do pivô. Novamente, o contador de comparações é incrementado a cada comparação. Finalmente, se ‘i’ e ‘j’ ainda não cruzaram, ainda há elementos que precisam ser trocados, e os elementos nas respectivas posições são trocados entre si. Após a troca, o contador de movimentações é incrementado, e 'i' é incrementado e ‘j’ é decrementado.

Já na etapa de ordenação, a função ‘ordenaQuick’ aplica a lógica do QuickSort recursivamente, recebendo o compartimento com a lista de rochas, os índices da parte do vetor a ser ordenada (‘esq’ e ‘dir’) e os contadores de comparações e movimentações. A primeira coisa que a função faz é chamar a função de partição, retornando os índices de partição ‘i’ e ‘j’. Esses índices marcam a posição onde a lista foi dividida, ou seja, onde os elementos menores que o pivô estão à esquerda e os maiores estão à direita. A função então verifica se a parte à esquerda do pivô contém elementos (ou seja, se ‘esq < j’). Se contiver, chama-se a ‘ordena_quick’ recursivamente para ordenar essa parte da lista. A função verifica se a parte à direita do pivô (índices entre ‘i’ e ‘dir’) contém elementos. Se contiver, chama-se o ‘ordena_quick’ recursivamente para ordenar essa parte da lista. O processo continua até que todas as sublistas sejam ordenadas. A recursão vai dividindo as listas em pedaços menores até que cada sublista tenha um único elemento (ou nenhum), o que significa que elas estão ordenadas.

```c
void ordena_quick(Compartimento* compartimento, int esq, int dir, int * comparacoes, int * movimentacoes){

    int i,j;
    particao_quick(compartimento, esq, dir, &i,&j, comparacoes, movimentacoes);
    if(esq<j)ordena_quick(compartimento,esq,j, comparacoes, movimentacoes);

    if(i<dir)ordena_quick(compartimento,i,dir, comparacoes, movimentacoes);

}

void quick_sort(Compartimento* compartimento, int* comparacoes, int* movimentacoes){
    ordena_quick(compartimento,0, compartimento->tamanho-1,comparacoes,movimentacoes);

}
```

Figura 8.2 - Processo de ordenação do QuickSort.

3.5 Medida de tempo

Para medir e retornar o tempo de execução, foi desenvolvido um algoritmo com base em FELIPE, Luiz, 2018[2]. No código citado, são executadas funções da biblioteca ‘time.h’, que disponibiliza os artifícios necessários para obter o número de clocks realizados pelo processador durante a execução, com o uso de variáveis do tipo ‘clock_t’. Dessa forma, armazena-se na variável ‘inicio’ o número de clocks realizados até o ponto de início (valor obtido com a função ‘clock()’) de um trecho de código, e armazena-se na variável ‘fim’ o número de clocks realizados até o ponto em que tal trecho termina (Figura 9).

```c
//registra o tempo de início
clock_t inicio = clock();

// Código de exemplo
int cont = 0;
for(int i = 0; i < 100000; i++){
    for(int j = 0; j < 120000; j++){
        cont++;
    }
}

//registra o tempo de término
clock_t fim = clock();
```

Figura 9 - Exemplo de registro de clocks iniciais e finais.

Tendo o número de clocks do início de do fim, faz-se a subtração do fim pelo início, e é obtido o número de clocks realizados na execução do trecho de código. Por fim, é necessário converter tal número de clocks para uma variável do tipo double que corresponde à quantidade de segundos que a execução levou, e isso se dá com a

[2] FELIPE, Luiz. “Como medir tempo de execução de um algoritmo de ordenação em C”. Stack Overflow, 2018. Disponível em: <https://pt.stackoverflow.com/questions/314041/calcular-tempo-de-execu%C3%A7%C3%A3o-de-um-algoritmo-de-ordena%C3%A7%C3%A3o-em-c>. Último acesso em: 10 de dezembro de 2024.

divisão desse valor pela constante ‘CLOCKS_PER_SEC’ - que também é obtida na biblioteca time e corresponde ao número de clocks feitos por segundo (Figura 10). Com isso, tem-se acesso ao tempo, em segundos, gasto com a execução.

```
double tempoTotal = (double)(fim-inicio)/CLOCKS_PER_SEC;
```

Figura 10 - Conversão do número de clocks resultantes em segundos.

Como o principal processo deste projeto é a ordenação das rochas, a execução dos outros processos (leitura de arquivo de entrada, por exemplo) não interfere no tempo final de forma significativa. Portanto, é capturado o tempo de início e fim como representado na Figura 11, medindo o tempo gasto apenas com as ordenações.

```
//inicio da medicao de tempo do algoritimo 1
inicio = clock();
insertion_sort(&compartimento_1,&comparacoes1, &movimentacoes1);
fim = clock();
tempo1 = (double)(fim-inicio)/CLOCKS_PER_SEC;

//inicio da medicao de tempo do algoritimo 2
inicio = clock();
quick_sort(&compartimento_2,&comparacoes2,&movimentacoes2);
fim = clock();
tempo2 = (double)(fim-inicio)/CLOCKS_PER_SEC;
```

Figura 11 - Medida de tempo das ordenações.

3.6 ‘main.c’

Inicialmente, são preparados alguns elementos necessários para fazer a distribuição, como a obtenção de um arquivo de entrada válido (Figura 12).

```
//executa o codigo com a leitura de arquivo
if(leitura_arq(argc, argv) != 0){
    FILE *file = leitura_arq(argc,argv);
    if( file == NULL){//verifica se o arquivo é nulo
        printf("arquivo inválido!\n");
        return 0;
    }
```

Figura 12 – Obtenção do arquivo de entrada.

Depois, declara-se algumas variáveis que serão usadas nas funções de ordenação (Figura 13): número de comparações, número de movimentações e tempo de execução para cada algoritmo. Além disso, são criados dois compartimentos – um para cada algoritmo de ordenação, e é lido o número de rochas da entrada.

```
int comparacoes1 = 0,comparacoes2 = 0, movimentacoes1 = 0, movimentacoes2 = 0;
clock_t inicio = 0, fim = 0;
double tempo1 = 0,tempo2 = 0;

//le o numero de rochas do arquivo
int N_rochas = 0;
fscanf(file,"%d",&N_rochas); fgetc(file);

//cria vazio os compartimentos a ser preenchido com as rochas e, posteriormente, ordenados
Compartimento compartimento_1;
Compartimento compartimento_2;
faz_compartimento_vazio(&compartimento_1,N_rochas);
faz_compartimento_vazio(&compartimento_2,N_rochas);
```

Figura 13 - Elementos necessários para a ordenação.

Em seguida, são lidas e inicializadas as rochas da lista, e elas são adicionadas ao ‘compartimento_2’ (Figura 14). Posteriormente, o compartimento_1 é inicializado como uma cópia do outro (Figura 15), pois, como dito anteriormente, a mesma lista será ordenada usando os dois algoritmos, e deve-se, então, usar compartimentos diferentes para operar a ordenação da lista. Assim, os compartimentos recebem a lista de rochas da entrada.

```
//executa a leitura das N_rochas
for(int i=0;i<N_rochas;i++){
    ListaMinerais lista_minerais_file;
    fListaMineraisVazia(&lista_minerais_file);
    case_R_file(file, &lista_minerais_file, &compartimento_2);
}
```

Figura 14 - Leitura das rochas da entrada.

```
//Clonando o compartimento para que os dados possam ser ordenados duas vezes
compartimento_1 = compartimento_2;

//inicio da medicao de tempo do algoritimo 1
inicio = clock();
insertion_sort(&compartimento_1,&comparacoes1, &movimentacoes1);
fim = clock();
tempo1 = (double)(fim-inicio)/CLOCKS_PER_SEC;

//inicio da medicao de tempo do algoritimo 2
inicio = clock();
quick_sort(&compartimento_2,&comparacoes2,&movimentacoes2);
fim = clock();
tempo2 = (double)(fim-inicio)/CLOCKS_PER_SEC;
```

Figura 15 - Execução das ordenações

Com isso, resta a formatação da saída, a ser observada posteriormente.

4. Compilação e execução

4.1 Execução no Windows

É usado no terminal o comando **make** do MinGW[3] para compilar usando o GCC[4].

Figura 16 - Comando de compilação no Windows

Para realizar a execução, deve-se digitar **./main -f** seguido do endereço do arquivo em questão:

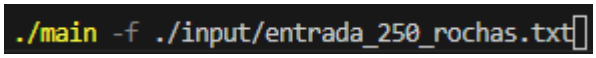

Figura 17 - Comando de execução no Windows.

4.2 Execução no Linux

Antes de compilar, deve-se instalar o GCC com o comando **sudo apt install gcc** no terminal.

Para compilar o projeto, digita-se o comando: **gcc src/main.c src/Compartimento/Compartimento.c src/Mineral/Mineral.c src/ListaMinerais/ListaMinerais.c src/RochaMineral/RochaMineral.c -Wall -Wextra -g -o main,** como no exemplo:

Figura 18 - Comando de compilação no Linux.

Para executar, digita-se o mesmo comando de execução do Windows (Figura 17).

5. Resultados

Dados arquivos de entrada formatados de forma que a primeira linha corresponde à quantidade de rochas na lista e as demais representam os dados de cada rocha, analisemos alguns testes para diferentes tamanhos de entrada, considerando o modelo de saída como: a lista ordenada pelo InsertionSort (Figura 19.1) e as informações sobre a execução de tal algoritmo, seguido pela lista ordenada pelo QuickSort (Figura 19.2) e as informações sobre a execução dele.

[3] MingGW. Disponível em: <https://sourceforge.net/projects/mingw/>. Último acesso em: 12 de dezembro de 2024.
[4] GCC. Disponível em: <https://gcc.gnu.org/>. Último acesso em: 12 de dezembro de 2024.

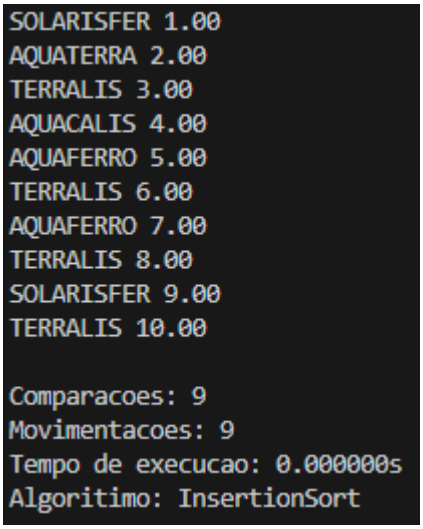

Figura 19.1 - Resultado do InsertionSort para 10 rochas.

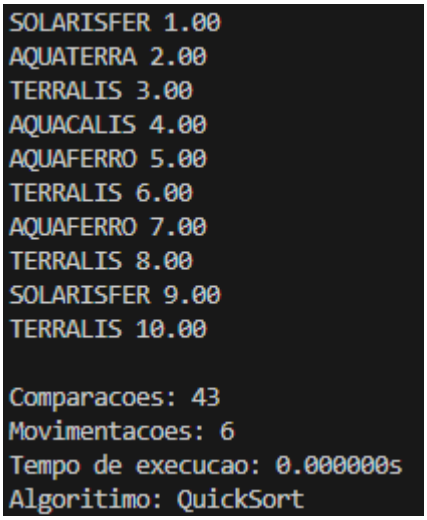

Figura 19.2 - Resultado do QuickSort para 10 rochas.

Isto posto, ao comparar os resultados para diferentes tamanhos de entrada, percebe-se que é observada diferença no tempo de execução dos algoritmos apenas a partir de 1000 rochas de entrada (Figura 20), quando se nota uma diferença, mesmo que quase nula.

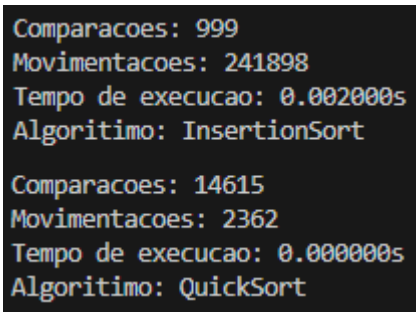

Figura 20 - Comparação dos algoritmos para 1000 rochas.

No caso de 10000 rochas, a diferença ainda é pequena, mas se torna mais notável (Figura 21).

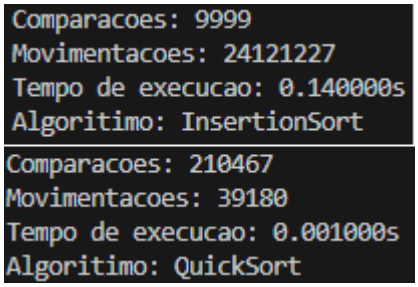

Figura 21 - Comparação dos algoritmos para 10000 rochas.

Por fim, na entrada de 40000 rochas, a diferença de desempenho dos algoritmos (Figura 22) se torna, finalmente, evidente, com uma lacuna superior a dois segundos.

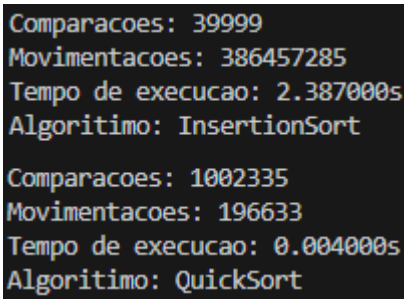

Figura 22 - Comparação dos algoritmos para 40000 rochas.

6. Conclusão

Feitas as comparações entre os algoritmos, torna-se possível a análise dos pontos positivos e negativos de cada um, bem como uma conclusão geral a partir disso.

6.1 InsertionSort

O algoritmo de inserção é um algoritmo simples e eficiente para listas pequenas ou parcialmente ordenadas, tendo fácil implementação e baixo custo de execução em tais casos. O algoritmo realiza poucas movimentações e comparações em listas já ordenadas, resultando em uma complexidade de tempo de O(n) no melhor caso. No entanto, ele se torna ineficiente para listas grandes, já que sua complexidade no pior caso é O(n^2), o que pode resultar em um desempenho inferior em comparação com outros algoritmos de ordenação mais eficientes, como o QuickSort.

6.2 QuickSort

O QuickSort, por outro lado, é amplamente reconhecido por seu desempenho em listas grandes. Com uma complexidade média de O(n log n), o quicksort é extremamente eficiente para conjuntos maiores em que é feita boa implementação da escolha de pivôs. Porém, o algoritmo tem uma desvantagem significativa no pior caso, que ocorre quando o pivô escolhido é sempre o menor ou maior elemento criando partições “desequilibradas”, o que resulta em uma complexidade de O(n^2). Isso pode

ser evitado com a escolha de um bom pivô, como em versões do algoritmo que usam a mediana de três elementos da lista. Vale citar, ainda, que tal algoritmo é recursivo, o que pode resultar em alto uso de memória, especialmente quando a profundidade da recursão é grande.

6.3 Considerações finais

Com o presente projeto, a partir dos testes realizados, observa-se a diferença de performance entre um algoritmos simples de ordenação e um sofisticado, e é feita a análise prática de pontos positivos e negativos de cada algoritmo, bem como as situações adequadas para a preferência de um em relação ao outro – no caso, a depender do tamanho da entrada – ou até a opção por combiná-los, considerando o intervalo de possíveis tamanhos de entrada.

7. Referências

ZIVIANI, Nivio. Projeto de algoritmos com implementações em Pascal e C. 4ª Edição, São Paulo. Editora Pioneira, 1999.

FELIPE, Luiz. “Como medir tempo de execução de um algoritmo de ordenação em C”. Stack Overflow, 2018. Disponível em: <https://pt.stackoverflow.com/questions/314041/calcular-tempo-de-execu%C3%A7%C3%A3o-de-um-algoritmo-de-ordena%C3%A7%C3%A3o-em-c>. Último acesso em: 10 de dezembro de 2024.

MingGW. Disponível em: <https://sourceforge.net/projects/mingw/>. Último acesso em: 12 de dezembro de 2024.

Github. Disponível em: <https://github.com/> Último acesso em: 26 de janeiro de 2025.