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

Trabalho 2 - AEDS 1

Problema do Compartimento

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 de desempenho de algoritmos a partir da medida de tempo de execução de um algoritmo de complexidade exponencial. Para tal, é proposto o desenvolvimento e o registro do tempo de execução de uma operação nomeada de “Problema do Compartimento”, que tem como objetivo a distribuição de rochas minerais (cada uma com um peso e um valor) em compartimentos de sondas espaciais de acordo com o valor agregado das combinações entre as rochas, sem que o peso agregado ultrapasse a capacidade dos compartimentos. Assim, deve-se gerar todas as combinações possíveis entre as N rochas da entrada e atribuir as três combinações de maior valor (e cujo peso não ultrapasse a capacidade 40 dos compartimentos) ao compartimento das 3 sondas disponíveis, de forma a armazenar a melhor combinação entre as N rochas no compartimento da sonda 1, gerar novas combinações entre as rochas restantes (aquelas que não foram armazenadas na sonda 1) e atribuir a melhor combinação ao compartimento da sonda 2 e, por fim, gerar novas combinações entre as rochas que não foram atribuídas às sondas 1 e 2, obter a melhor combinação entre elas e adicioná-la no compartimento da sonda 3. Tal implementação deve ser realizada com a utilização de força bruta, gerando todas as combinações de todos os tamanhos possíveis, o que faz de tal implementação um algoritmo de complexidade exponencial.

A abordagem escolhida no desenvolvimento se baseia na inicial importação dos TADs utilizados no Trabalho Prático 1 (rocha mineral, sonda espacial, lista de sondas espaciais e compartimento), alterando apenas os atributos da rocha mineral para ID, peso e valor. Em seguida, iniciou-se o desenvolvimento do arquivo ‘main.c’, a pesquisa e adaptação do algoritmo responsável por gerar as combinações e a medida de tempo de execução.

A estrutura da lista de sondas e de minerais foi feita com a utilização da lista encadeada.

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 baseado no material complementar disponibilizado pelos monitores, de forma a compilar os múltiplos arquivos de diferentes pastas:

```
compile: src/main.c src/Compartimento/Compartimento.c src/RochaMineral/RochaMineral.c src/Sonda_espacial/Sonda_espacial.c src/Lista_sondas_espaciais/Lista_sonda_espacial.c
	gcc src/main.c src/Compartimento/Compartimento.c src/RochaMineral/RochaMineral.c src/Sonda_espacial/Sonda_espacial.c src/Lista_sondas_espaciais/Lista_sonda_espacial.c -Wall -Wextra -g -o main
```

Figura 1 - Arquivo Makefile.

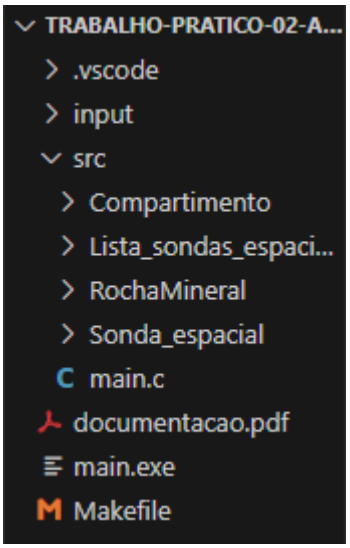

Figura 2 - Repositório do projeto.

3. Desenvolvimento

Observemos, a partir de módulos, os pontos mais importantes do projeto, com enfoque nas especificidades do Problema do Compartimento (seu funcionamento e as funções que a envolvem).

3.1 Medida de tempo

Para medir e retornar o tempo de execução, foi desenvolvido um algoritmo com base em FELIPE, Luiz, 2018[1]. 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

[1] 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.

‘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:

```
//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 3.1 - 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 divisão desse valor pela constante ‘CLOCKS_PER_SECOND’ - que também é obtida na biblioteca time e corresponde ao número de clocks feitos por segundo (Figura 3.2). Com isso, tem-se acesso ao tempo, em segundos, gasto com a execução:

```
//calcula o tempo total
double tempoTotal = (double)(fim - inicio)/CLOCKS_PER_SEC;

printf("Tempo gasto: %f segundos\n", tempoTotal);
```

Figura 3.2 - Conversão do número de clocks resultantes em segundos.

Como o principal processo deste projeto é a distribuição 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 de forma semelhante ao exemplo (Figura 3.1) - assim que se inicia a execução do main -, e o tempo final é capturado após o fim da saída da distribuição.

3.2 ‘main.c’

Abordemos alguns aspectos do arquivo ‘main.c’ que são executados antes de chegarmos à parte de combinação.
Inicialmente, são preparados alguns elementos necessários para fazer a distribuição, como a obtenção de um arquivo de entrada válido (Figura 4):

```c
//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 4 – Obtenção do arquivo de entrada.

Em seguida, inicializamos as sondas de acordo com a constante ‘N_sonda’, uma constante que corresponde a 3, número fixo de sondas (Figura 5):

```c
//cria e inicializa lista sonda espacial
Lista_sonda_espacial lista_de_sondas_file;
inicializa_lista_sonda_espacial(&lista_de_sondas_file);

//inicializa as sondas de acordo com o máximo de sondas e as coloca dentro da lista
for(int i = 0; i<N_sonda; i++){
    char id[20];
    sprintf(id, "%d", i+1);
    Sonda_espacial sonda_i;
    inicializa_Sonda_Espacial(&sonda_i,id, max_peso_compartimento);
    insere_item_lista_sonda_espacial(&lista_de_sondas_file,&sonda_i);
}
```

Figura 5 – Criação e inicialização das 3 sondas.

Depois, lemos o número de rochas do arquivo de entrada, inicializamos cada sonda com um ID (que vai de 0 a N_rochas – 1), adicionamos as N_rochas em um vetor, do qual serão extraídas as sondas que formarão as combinações (Figura 6):

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

int Id_rocha=0;
//faz um vetor de rochas para redistribuir depois
RochaMineral* lista_rochas = (RochaMineral*) malloc(N_rochas * sizeof(RochaMineral));
//pega as rochas e coloca no vetor
for (int i = 0; i < N_rochas; i++){
    int peso = 0, valor = 0;
    RochaMineral rocha;
    fscanf(file,"%d", &peso);
    fgetc(file);
    fscanf(file, "%d", &valor);
    fgetc(file);
    inicializaRochaMineral(&rocha,Id_rocha,peso,valor);
    Id_rocha++;
    lista_rochas[i] = rocha;
}
```

Figura 6 – Leitura do número de rochas e do peso e valor de cada uma.

Com isso, atinge-se o ponto em que será realizada a distribuição e é necessário explicar separadamente os pormenores que a envolvem.

3.3 Combinação

Foi tomado como base a função recursiva que escreve todas as combinações de tamanho r entre números inteiros presentes em uma lista de tamanho n, desenvolvida por SILVEIRA, Marcelo, 2021[2]:

```
void combinationUtil(int arr[], int data[], int start, int end, int index, int r)
{
  int i, j;

  if (index == r)
  {
    for (j=0; j<r; j++)
      printf("%d ", data[j]);
    printf("\n");
    return;
  }

  for (i=start; i<=end && end-i+1 >= r-index; i++)
  {
    data[index] = arr[i];
    combinationUtil(arr, data, i+1, end, index+1, r);
  }
}
```

Figura 7 - Função de combinação usada como base.

A função ‘combinacao’ representada na Figura 7 tem como parâmetros: uma lista da qual os elementos serão combinados, uma lista temporária que comportará cada combinação, o ponto de início da lista, ponto de fim da lista, o índice a partir do qual as os números serão ordenados e o tamanho das combinações desejadas. A partir disso, torna-se necessário: adaptar a função para lista de rochas e, ao invés de escrever no terminal as combinações, verificar qual é aquela cujo valor agregado é maior para cada tamanho de combinação. Então, a função acima foi adaptada de maneira a, resumidamente, gerar todas as combinações possíveis entre as rochas e, para cada tamanho de combinação (de 1 a N_rochas), eleger a melhor e adicioná-la a uma lista contendo a melhor combinação de cada tamanho. Tal processo se dá da seguinte forma:

Inicialmente, como representado na Figura 8, adiciona-se alguns novos parâmetros além dos citados acima: um ponteiro para sonda espacial, que será importante na função ‘problema_do_compartimento’, o ponteiro ‘peso_maior’, que corresponde ao peso total da melhor combinação, o ponteiro ‘qtd_rochas_maior’, que corresponde à

[2] SILVEIRA, Marcelo. “Curso de C. Análise Combinatória”. Disponível em: <https://marmsx.msxall.com/cursos/c/c16.html>. Último acesso em: 11 de dez de 2024.

quantidade de rochas na melhor combinação, e a ‘lista_melhor_comb’, que armazena a melhor combinação de cada tamanho. Além disso, são criadas variáveis de controle de valores e pesos, e uma variável estática que controla o número de combinações geradas:

```
void combinacao(RochaMineral lista_rochas[], RochaMineral lista_temp[], Sonda_espacial * sonda,
                int inicio, int fim, int indice, int r, int * maior_valor, int * peso_maior, int * qnt_rochas_maior,
                RochaMineral lista_melhor_comb[]){
    int maior_valor_atual = 0, peso_atual = 0, qnt_rochas_atual = 0, peso_max = 40, N_rochas = fim+1;
    static int combinacoesFeitas = 0;
```

Figura 8 - Parâmetros e variáveis de controle da função ’combinacao’.

A recursão da função termina quando o ‘indice’ atual da combinação é igual ao tamanho das combinações desejadas (Figura 8.1):

```
if(indice == r){
```

Figura 8.1 - Condição de fim da combinação atual.

Ao obter uma nova combinação, incrementa-se a variável ‘combinacoesFeitas’, que será importante para verificar se ainda resta alguma combinação desse tamanho r a ser feita (Figura 8.1.1):

```
combinacoesFeitas++;
```

Figura 8.1.1 - Incremento da variável de controle ‘combinacoesFeitas’.

A obtenção da nova combinação significa, ao mesmo tempo, que se deve verificar se seu valor é maior que o maior valor agregado obtido até então e se ela não ultrapassa o limite 40 de peso:

```
for (int j = 0; j < r; j++){// atualiza os valores da combinação atual
    maior_valor_atual += lista_temp[j].valor;
    qnt_rochas_atual++;
    peso_atual += lista_temp[j].peso;
}
//se o peso não for muito e o valor for maior, essa vira a melhor combinação
if((maior_valor_atual > *maior_valor) && !(peso_atual > peso_max)){

    *maior_valor = maior_valor_atual;
    *qnt_rochas_maior = qnt_rochas_atual;
    *peso_maior = peso_atual;
    for (int k = 0; k < r; k++){
      lista_melhor_comb[k] = lista_temp[k];
    }

}
```

Figura 8.1.2 - Verificando se a combinação atual é a melhor.

Note que é importante atualizar os atributos da melhor combinação encontrada até aqui (Figura 8.1.2) caso a combinação atual seja a melhor. Além disso, a combinação atual deve ser copiada para 'lista_melhor_comb', a lista que armazena a melhor combinação do tamanho r.

Após isso, é feita uma verificação (Figura 8.1.3): se o número de combinações feitas até então é igual ao número de combinações totais possíveis para esse tamanho r, significa que todas as combinações foram testadas e, logo, temos a melhor combinação definitiva do tamanho r:

```
int combinacoesTotais = qtdCombinacoes(N_rochas, r);
if(combinacoesFeitas == combinacoesTotais){//verifica se chegou a ultima combinação
    combinacoesFeitas = 0;//reseta a quantidade de combinaçoes para este tamanho
}
return;
```

Figura 8.1.3 - Condição de término da verificação de melhor combinação por tamanho.

Nesse momento, é importante citar alguns pontos:

- A variável 'combinacoesFeitas' é declarada como 0 no início da função, mas é uma variável estática e, sendo incrementada sempre que se tem uma nova combinação, só voltará a ser 0 quando todas as combinações de tamanho r forem geradas. Então, enquanto não forem geradas e verificadas todas as combinações de tamanho r, 'combinacoesFeitas' corresponde à quantidade de

combinações de tamanho r feitas e serve para ter controle de quando ter certeza que já foi obtida a melhor combinação de tamanho r.

- A variável ‘combinacoesTotais’ corresponde ao cálculo de quantas combinações de tamanho r são possíveis de serem feitas a partir do número de rochas na lista. Tal cálculo é feito com o uso de fórmulas de análise combinatória (Figura 8.1.4) também obtidas de SILVEIRA, Marcelo, 2021[3]:

```
int arranjo(int n, int p){//faz um arranjo
    if (p>1)
        return n*arranjo(n-1,p-1);
    else
        return n;
}

int fatorial(int n){//faz um numero fatorial
    if (n > 0){
        return n*fatorial(n-1);
    }else{
        return 1;
    }
}

int qtdCombinacoes(int n, int p){//verifica quantas combinaçoes serao feitas para r tamanho
    return arranjo(n,p)/fatorial(p); /*o mesmo que 'return fatorial(n)/(fatorial(p)*fatorial(n-p));'
                                       fórmula de combinação*/

}
```

Figura 8.1.4 - Funções recursivas de análise combinatória.

Caso não seja executado o ‘if’ que representa a obtenção de uma nova combinação (Figura 8.1), conclui-se que a combinação ainda não foi gerada, pois o ‘indice’ da combinação ainda não é igual ao tamanho de combinação desejado. Logo, o próximo índice deve ser gerado pela função (Figura 8.2):

```
for (int i = inicio; i <= fim && fim-i+1 >= r - indice; i++){
    lista_temp[indice] = lista_rochas[i];
    combinacao(lista_rochas, lista_temp, sonda, i+1, fim, indice + 1,
               r, maior_valor, peso_maior, qnt_rochas_maior, lista_melhor_comb);
}
```

Figura 8.2 - Geração do próximo índice da combinação.

A função gera as combinações recursivamente através do laço ‘for’, onde ’i’ percorre os índices possíveis para adicionar uma nova rocha à combinação, armazenada em ‘listaTemp’. A recursão ocorre para o próximo índice (‘indice’ + 1),

[3] SILVEIRA, Marcelo. “Curso de C. Análise Combinatória”. Disponível em: <https://marmsx.msxall.com/cursos/c/c16.html>. Último acesso em: 11 de dez de 2024.

limitando-se ao intervalo definido pelos parâmetros ‘inicio’ e ‘fim’ e garantindo que a quantidade de rochas restantes seja suficiente para completar a combinação (o critério ‘fim - i + 1 >= r – indice' impede que a recursão tente escolher mais rochas do que as disponíveis).

3.4 Problema do Compartimento

A ideia dessa função é fazer uma matriz em que cada linha corresponde à melhor combinação de cada tamanho (a primeira linha corresponde à melhor combinação de tamanho 1, a segunda linha corresponde à melhor combinação de tamanho 2, e assim por diante). Assim, obtém-se a melhor combinação e ela é adicionada à próxima sonda disponível.

Os parâmetros usados na função (Figura 9.1) são: a lista com as rochas a serem combinadas, um ponteiro para uma sonda espacial, um índice correspondente à posição da sonda na lista de 3 sondas, e o número de rochas na lista:

```
void problema_do_compartimento(RochaMineral lista_rochas[],Sonda_espacial * sonda,int sonda_atual, int *N_rochas){
```

Figura 9.1 - Parâmetros da função ’problema_do_compartimento’.

A função começa criando uma matriz que armazenará as melhores combinações de rochas para cada número de rochas (de 1 até N_rochas) - Figura 9.2.

```
RochaMineral matriz[*N_rochas][*N_rochas]; //matriz contendo as melhores combinações
```

Figura 9.2 - Declaração da matriz com as melhores combinações.

Em seguida, usando a função de combinação para cada tamanho possível (de 1 até o número de rochas da entrada), usa-se um ‘for’ (Figura 9.3.1) de forma a obter a melhor combinação de cada tamanho.

```
for(int r = 1; r <= *N_rochas; r++){//para cada ordem, faz uma combinação

    RochaMineral lista_temp[r];//uma combinação
    RochaMineral lista_melhor_combinacao[r];//melhor combinação

    for (int j = 0; j < r; j++)//deixa todos os valores como 0 para evitar lixo de memoria
        lista_melhor_combinacao[j].valor = 0;
```

Figura 9.3.1 - Inicio do for e preparação para obter as melhores combinações

Para cada valor de ‘r’, a função prepara duas listas: ‘lista_temp’, que armazena a combinação atual) e 'lista_melhor_comb’, para armazenar a melhor combinação (parâmetro usado na função de combinação como representado na Figura 8.1.2). A

função de combinação é chamada para encontrar a melhor combinação de ‘r’ rochas, que é armazenada na matriz (Figura 9.3.2).

```
    int maior_valor = 0, peso_maior = 0, qnt_rochas_maior = 0;

    combinacao(lista_rochas, lista_temp, sonda, 0, *N_rochas-1, 0,
                    r, &maior_valor, &peso_maior, &qnt_rochas_maior, lista_melhor_combinacao);

    for (int count = 0; count < r; count++){//coloca as rochas da melhor combinação na matriz
        matriz[r-1][count] = lista_melhor_combinacao[count];
    }
}
```

Figura 9.3.2 - Obtendo a melhor combinação de cada tamanho e salvando na matriz.

Em seguida, é feita a procura pela melhor combinação entre as melhores de cada tamanho (Figura 9.4).

```
int melhor_comb_valores = 0;
int melhor_atual = 0;
RochaMineral melhor_comb[*N_rochas];
// printf("%d\n", *N_rochas);
int aux=0;
for(int i = 0; i <*N_rochas; i++){
  for(int j = 0; j<i+1; j++){//vai somando o valor das rochas na linha para ver qual é a melhor
    melhor_atual += matriz[i][j].valor;
  }
  if (melhor_atual >= melhor_comb_valores){//se for a melhor, salva
    aux = i+1;
    melhor_comb_valores = melhor_atual;
    for(int k = 0;  k<aux; k++){
      melhor_comb[k] = matriz[i][k];
    }
  }
  melhor_atual = 0;
}
```

Figura 9.4 - Obtendo a melhor combinação de qualquer tamanho do intervalo.

Nesse ponto, o código procura a melhor combinação global entre todas as combinações possíveis armazenadas na matriz. Ele percorre as linhas da matriz (‘i’ de 0 até `*N_rochas`), somando o valor das rochas na linha atual e comparando com o melhor valor encontrado até o momento. Assim, se a soma do valor das rochas na linha for maior ou igual ao valor máximo até então, essa combinação é considerada a nova melhor, e é armazenada em ‘lista_melhor_comb’.

Após isso, é feita a distribuição das combinações entre as sondas (Figura 9.5) tendo o cuidado de, antes de distribuir as rochas na sonda 1, passar tais rochas da lista para o fim da lista e diminuir o tamanho da lista que será utilizada, de maneira a desconsiderar as rochas já usadas para gerar novas combinações para a sonda 2 (o mesmo vale para a sonda 3).

```
for(int i= 0; i<aux; i++){//coloca na sonda a melhor combinação
//    printf("%d ", melhor_comb[i].valor);
  inserir_rocha(&sonda->Compartimento,&melhor_comb[i]);
  for(int j=0;j<*N_rochas; j++){
    if(melhor_comb[i].id == lista_rochas[j].id){
      lista_rochas[j] = lista_rochas[*N_rochas-1];
      *N_rochas = *N_rochas-1;//diminui o tamanho do vetor original de rochas
    }
  }
}
```

Figura 9.5 - Distribuindo as rochas

Por fim, é mostrada no terminal a distribuição feita (Figura 9.6).

```
printf("\n");//imprime a sonda
printf("Sonda %d: Peso %d, Valor %d , Solucao [",sonda_atual, sonda->Compartimento.peso_atual, sonda->Compartimento.valor);
for(int i = 0; i<aux;i++){
    printf("%d", melhor_comb[i].id);
    if(!(i == aux-1)){
        printf(",");
    }
}
printf("]\n");
```

Figura 9.6 - Retorno de cada sonda com a combinação de rochas.

Assim, finalmente, realiza-se a distribuição para cada sonda na lista (Figura 9.7).

```
//faz o problema do compartimento para cada sonda na lista de sondas
Celula * celula_sonda = lista_de_sondas_file.pPrimeiro->pProx;
int sonda_atual = 1;
while(celula_sonda != NULL){
    problema_do_compartimento(lista_rochas, &celula_sonda->item_sonda, sonda_atual, &N_rochas);
    sonda_atual++;
    celula_sonda = celula_sonda->pProx;
}
```

Figura 9.7 - Redistribuindo as rochas entre as sondas.

Após isso, é calculado e mostrado o tempo de execução, da maneira mostrada no item 3.1 e a execução chega ao fim.

4. Compilação e Execução

4.1 Execução no Windows

É usado no terminal o comando **make** do MinGW[4] para compilar usando o GCC[5].

```
PS C:\Users\Cliente\Documents\GitHub\Trabalho-Pratico-02-AEDS-1> mingw32-make
```

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

Figura 10 - Comando de compilação no Windows

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

```
PS C:\Users\Cliente\Documents\GitHub\Trabalho-Pratico-02-AEDS-1> .\main -f .\input\teste3.txt
```

Figura 11 - 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/RochaMineral/RochaMineral.c src/Sonda_espacial/Sonda_espacial.c src/Lista_sondas_espaciais/Lista_sonda_espacial.c -Wall -Wextra -g -o main,** como no exemplo:

```
tavinescada@DESKTOP-FVHV23S:/mnt/c/Users/Cliente/Documents/GitHub/Trabalho-Pratico-02-AEDS-1$ gcc src/main.c src/Compartimento/Compar
timento.c src/RochaMineral/RochaMineral.c src/Sonda_espacial/Sonda_espacial.c src/Lista_sondas_espaciais/Lista_sonda_espacial.c -Wall
 -Wextra -g -o main
```

Figura 12 - Comando de compilação no Linux.

Para executar, digita-se o mesmo comando de execução do Windows, mas invertendo as barras:

```
./main -f ./input/teste3.txt
```

Figura 13 - Comando de execução no Linux.

5. Resultados

Analisemos alguns testes para diferentes tamanhos de entrada, sendo usado um PC com as seguintes especificações:

Processador	Intel(R) Core(TM) i3-1005G1 CPU @ 1.20GHz 1.19 GHz
RAM instalada	8,00 GB (utilizável: 7,79 GB)

Figura 14 - Especificações do PC.

Executando o programa para 20 rochas, o tempo gasto é menor que 1 segundo, como na Figura 15.

```
Sonda 1: Peso 37, Valor 175 , Solucao [5,8,10,14,15,19]

Sonda 2: Peso 38, Valor 57 , Solucao [1,6,18]

Sonda 3: Peso 40, Valor 39 , Solucao [0,13]
Tempo gasto: 0.133000 segundos
```

Figura 15 – Teste para 20 rochas.

Ao executar para 25 rochas, porém, já existe um aumento significativo, de 4 segundos (Figura 16).

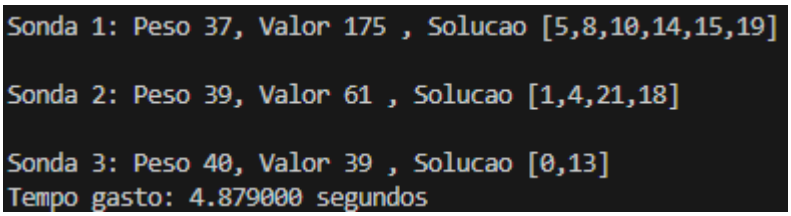

Figura 16 – Teste para 25 rochas.

Nesse ponto, o aumento fica cada vez mais significativo, de maneira que, para a casa das 30 rochas, por exemplo, já se observa uma duração na casa das centenas de segundos (Figura 17).

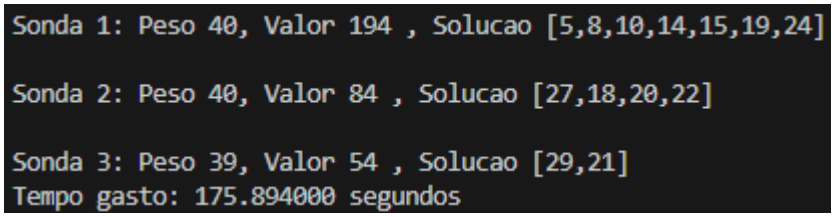

Figura 17 – Teste para 30 rochas.

Dessa maneira, fica evidente que, a partir de 40 rochas, por exemplo, o tempo de execução não é mais razoável. Portanto, para 50, 100 ou mais rochas, não é razoável executar o programa, e a duração da execução poderia ultrapassar as centenas de séculos.

6. Conclusão

No presente projeto, é possível observar na prática o comportamento de algoritmos de complexidade exponencial e como, a partir de certos tamanhos de entrada, fica impossível chegar ao final da execução, pois o tempo gasto por ela pode chegar a valores totalmente inviáveis (valores na casa de séculos, por exemplo).
Além disso, o desenvolvimento do Problema do Compartimento contribui para adquirir maturidade em relação à análise de complexidade de algoritmos e ao desenvolvimento de soluções inteligentes e viáveis.

7. Referências

SILVEIRA, Marcelo. “Curso de C. Análise Combinatória”. Disponível em: <https://marmsx.msxall.com/cursos/c/c16.html>. Último acesso em: 11 de dezembro de 2024.

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: 25 de novembro de 2024.