Trabalho Prático 2
Número de Ocorrências das Palavras

Lucas Rocha Laredo

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

lucaslaredo@ufmg.br

1. Introdução:

A proposta deste trabalho prático foi a implementação de um programa capaz de receber entradas textuais, com uma nova sequência lexicográfica e uma série de palavras a serem ordenadas de acordo com essa nova ordem, e retornar uma saída com as palavras ordenadas e a sua quantidade de aparições. A fim de solucionar o problema proposto neste trabalho, o projeto é capaz de lidar com arquivos, lendo-os, gerando-os e manipulando-os.

2. Implementação:

O programa foi desenvolvido na linguagem C++, utilizando o compilador G++, da GNU Compiler Collection.

2.1.1. Estrutura de Dados:

A implementação deste programa teve como base o uso de uma Estrutura de Dados simples: o Array, tendo como base o princípio da Programação Orientada a Objetos, possibilitado pela Linguagem C + +. O critério de escolha dessa Estrutura de Dados (ED) foi principalmente a praticidade em relação a implementação e o menor custo computacional dessa estrutura. Notadamente, Listas Simples, ou ainda Listas mais complexas, como uma Lista Encadeada, poderia, facilmente, substituir os arrays - isso ficará mais claro logo adiante, quando explicarmos qual o uso desses arrays - entretanto, o custo computacional aumentaria,

o que não é interessante, à medida que, em tese, esse programa possui uma proposta de ser "leve".

Na implementação deste programa, existem múltiplos Arrays, com diferentes funções. O primeiro uso dessa estrutura é a fim de armazenar o novo "alfabeto", que é fornecido no arquivo de entrada. Nesse caso, esse vetor é estático, e armazena os 26 caracteres do alfabeto.

```
char alfabeto[26];
```

Além disso, são armazenados em um outro vetor, todas as palavras do texto, diretamente lidas do arquivo de entrada. É fundamental mencionar que este vetor armazena Tipos Abstratos de Dados "*word"*, que será abordado mais adiante, na seção **2.1.2.**. Este é um momento cauteloso, uma vez que são utilizados ***TRÊS*** diferentes arrays, ambos armazenando TADs "*word"*. Primeiramente, um vetor armazena ***todas*** as palavras do texto, em seguida, através de um "filtro", contamos o número de aparição de cada palavra e selecionamos apenas uma das suas ocorrências, inserindo-a em um outro vetor. Por fim, este array final é ordenado, utilizando o algoritmo QuickSort.

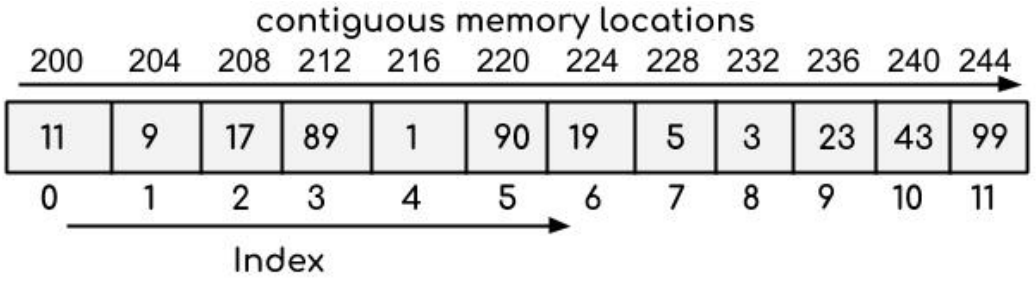


Figura 1. Diagrama Array

É importante enfatizar a escolha desta estrutura tão simples. O diagrama acima demonstra a simplicidade de um Array, mas além disso, demonstra uma propriedade ***muito importante*** dessa estrutura: a alocação contínua da memória. Essa propriedade é fundamental para diminuir o custo computacional do programa, como será visto adiante. É fundamental ter as instâncias de palavras próximas dentro da memória, pois, desta forma, podemos acessá-las com mais facilidade dentro da memória e dentro do programa, por meio dos índices do array.

2.1.1. Tipos Abstratos de Dados:

A implementação deste programa contou com apenas um Tipo Abstrato de Dados, as palavras. Esse TAD recebeu o nome *word*, e corresponde às palavras que são lidas dentro do

texto. Entretanto, esse TAD possui mais de um atributo, ou seja, extrapola apenas o "nome" - ***chave*** - da palavra, mas contém também a sua quantidade, dentro do texto, e uma "senha". O *nome* é o atributo utilizado para contar as quantidades dentro do texto, uma vez que funciona como a chave da palavra, ou seja, torna aquela instância única. Intuitivamente, o atributo *quantidade* é utilizado para facilitar a contagem, necessária para a saída do programa. Como cada palavra é "única" a sua quantidade sempre própria, e é armazenada na própria instância. Por fim, desta vez, não tão intuitivamente, a senha corresponde à chave da palavra "*criptografada"* no novo alfabeto. Pode parecer confuso, mas essa foi a estratégia utilizada para ordenar de acordo com o novo alfabeto. Ou seja, substituímos a posição de cada letra da palavra, pela a letra correspondente à anterior no novo alfabeto. Assim, conseguimos ordenar com propriedade. Para armazenar dentro do próprio objeto, foi criado o atributo *senha*. Além disso, dentro desta Classe temos também os métodos de definição da senha e os métodos de ordenação das palavras, que correspondem ao QuickSort.

```
class word {
    public:
        string chave;
        int quantidade;
        string senha;
        word() {
            this->quantidade = 1;
        };
        word(string chave) {
            this->chave = chave;
            this->quantidade = 1;
        }
        //metodos
        void defineSenha(char ordem[]);
        //metodos de ordenacao
        void Particao(int Esq, int Dir,int *i, int *j, word *A, int parametroMediana);
        void Ordena(int Esq, int Dir, word *A, int maxParticao, int parametroMediana);
        void QuickSort(word *A, int n, int maxParticao, int parametroMediana);

};
```

Figura 2. TAD *word*

2.2. Métodos e Procedimentos utilizados e Funcionamento:

A implementação deste programa não é tão robusta, ou seja, não possui uma grande quantidade de TADs e Estruturas de Dados. Isso é refletido na quantidade de métodos e procedimentos existentes. É fundamental mencionar, que algumas linhas de comando possuem papel fundamental para a ocorrência do programa, e, ainda que não sejam métodos, serão abordadas nessa seção.

A primeira etapa do programa é a leitura dos dados de entrada do arquivo disponibilizado. Entretanto, esse "procedimento" requer o nome dos arquivos de entrada. Desse modo, um passo prévio é tomado, com o objetivo de ler os parâmetros passados pela

linha de comando, e armazenar nas variáveis correspondentes ao *nome do arquivo de entrada* (`string lognome`) , o *nome do arquivo de saída* (`string lognome`), e os parâmetros *m* e *s,* usados para calcular o pivô, e o tamanho máximo da partição, respectivamente. Após ler todos os dados e armazená-los nas variáveis correspondentes, seguimos para o procedimento da leitura do arquivo. Esse procedimento é bastante intuitivo, e **fundamental** para o funcionamento do programa. A partir dos "comandos" dados na entrada, os dois principais *arrays* do programa são preenchidos: *alfabeto* e *vetPalavras*. Na verdade, antes de armazenar as palavras lidas do *input,* elas são transferidas (com repetição) para dentro de um vetor, chamado Palavras. Isso é fundamental para contar o número de palavras, e lidar com as intempéries presentes nas palavras. Ou seja, é possível que as palavras possuam caracteres não desejados, que devem ser tratados, logo, o armazenamento dentro de um vetor inicial, que representa todas as palavras, facilita esse tratamento. A função *fix_word* é a responsável pela correção. Ela recebe todas as palavras, e remove, os caracteres indesejados, conforme foi solicitado no enunciado do projeto. Agora sim, após o tratamento das palavras dentro do vetor Palavras*,* cada palavra é transferida para o vetor *vetPalavras*, estático, e o seu tamanho é o mesmo do número de palavras contadas do *array* anterior. Essa contagem é feita a partir de um procedimento chamado *countWords*. No caso, esse vetor armazena as ***instâncias*** de palavras, ou seja, ele armazena a chave, a senha e a quantidade de cada uma. Contudo, ele armazena-as com repetição - há mais de uma palavra com a mesma chave - o que é indesejado. Logo, filtramos os dados do vetor, e transferimo-nos para um outro *array*, sem repetição, mas armazenando a quantidade correta de cada instância de palavra dentro do próprio objeto. O *array* que armazena essa "lista" final de palavras é chamada de *finalPalavras*. Agora, restam dois passos: 'Criptografar' as palavras e ordená-las com a nova ordem.

```
int add = 1;
int numeroPalavras = countWords(palavrasTexto);
int tamanhoFinal = 0;
word vetPalavras[numeroPalavras];
stringstream palavra(palavrasTexto);
for(int i = 0; i < numeroPalavras; i++)
{
  palavra >> aux;
  for(int j = 0; j < numeroPalavras; j++) {
    if(vetPalavras[j].chave == aux) {
      vetPalavras[j].quantidade++;
      add = 0;
    }
  }
  if(add) {
    vetPalavras[tamanhoFinal].chave = aux;
    tamanhoFinal++;
  }
  add = 1;
}

word finalPalavras[tamanhoFinal];
for(int i = 0; i < tamanhoFinal; i++) {
  finalPalavras[i] = vetPalavras[i];
}
```

Figura 3. Armazenando os dados no *array*

As chaves foram transcritas para senhas, que, por sua vez, foram ordenadas, com o objetivo de fazer a ordenação na nova ordem, indicada no *input*. Isso foi feito através do método define senha, dentro do TAD *word*. Esse método pega cada caracter da string *chave* e o troca por seu correspondente no "novo alfabeto". Desse modo, a ordenação pode ser feita de acordo com essa nova ordem disponibilizada.

Por fim, chegamos, finalmente, à ordenação. Como exigido no enunciado, o método utilizado foi o QuickSort. Esse é um método recursivo de ordenação, com um bom custo computacional, como será visto adiante. Há a divisão em três etapas: O próprio QuickSort, Ordenação e Partição. Todos esses métodos pertencem à Classe *word*, ou seja, estão dentro dos objetos. No caso, passamos o *array finalPalavra* para a chamada da função e ele é ordenado. É fundamental mencionar que essa ordenação utilizando quickSort é interrompida, e substituída por um outro método, que, arbitrariamente, foi escolhido o método selectionsort. Isso ocorre pois, a partir de um tamanho mínimo *s*, definido pelo usuário, para a partição, utilizamos esse método alternativo, e isso oferece um melhor custo computacional para o programa. Um subvetor é também ordenado utilizando outro método, o *bolha*. Nesse caso, o objetivo é a escolha do *pivô* dentro de cada partição. Esse subvetor tem tamanho *m*, também definido pelo usuário.

Após a ordenação, resta apenas a escrita da saída do programa em um novo arquivo, cujo nome também é escolhido pelo usuário.

3. Análise de Complexidade:

Começaremos a Análise de Complexidade pela função principal: a *main.* Antes de mais nada, é fundamental enfatizar que a função principal ficou recheada de linhas de comando fundamentais para o funcionamento do programa, logo, é importante passar por todas. Um bom começo é o processo de leitura e escrita dos arquivos de entrada e saída, respectivamente. Contudo, antes disso, menciona-se o processo de verificação das entradas da linha de comando. Nesse caso, temos um *laço* com as *n* entradas presentes na linha de comando. Caso apenas o nome do programa seja executado, esse *n* seria igual a 1. Logo, esse processo possui complexidade $O(n)$, no pior caso, e $O(1)$ no melhor caso. A mesma complexidade é compartilhada no processo de escrita da saída no arquivo final. Nesse caso, percorremos os *n* elementos do *array finalPalavras* e os imprimimos, junto a sua quantidade. Caso, apenas uma palavra esteja nesse *array,* teríamos complexidade $O(1)$, caso contrário, $O(n)$. Diferentemente das citadas anteriormente, a função de leitura dos dados de entrada,

possui complexidade $O(n^3)$ no pior caso. Isso ocorre, pois primeiro armazenamos *cada palavra* da entrada em um variável, e, caso estejamos tratando das palavras após *#TEXTO*, entramos em um novo laço, até o final do arquivo - ou a palavra *#ORDEM*, em que "consertamos" a palavra, e a inserimos na string de palavras citada anteriormente. Desse modo, até então, temos complexidade $O(n^2)$. Contudo, dentro da função *fix_word*, temos um novo laço, agora passando por ***cada caractere*** da palavra lida. Ou seja, temos uma complexidade $O(n^3)$ dentro dessa função.

Em seguida, o próximo método executado é o de contagem de palavras. Novamente, temos um método de complexidade $O(n)$, uma vez que, passamos por ***todos os caracteres da string*****,** buscando os espaços, para contar as palavras. O processo seguinte, é o de verificação e contagem das palavras repetidas dentro da string. Nesse caso, temos complexidade $O(n^2)$, ao passo que temos dois laços aninhados. Novamente, caso o número de palavras contado anteriormente seja 1, temos complexidade $O(1)$ no melhor caso. Em seguidas, temos complexidade $O(n)$ para transferir as palavras repetidas para um novo array, agora sem repetição.

A parte de definição das senhas também é levemente custosa para o programa. Isso significa que, à medida que o número de chaves cresce, assim como o tamanho das palavras, o programa se torna mais custoso. A senha deve ser definida para *cada uma das palavras*, logo, percorremos o vetor *finalPalavras* e chamamos o método *defineSenha* para cada uma delas. Dentro desse método temos um laço que, caso entre em uma das condições, aninha com outro laço, de 26 execuções. Ou seja, no **pior caso**, esse processo possui custo computacional $O(n^3)$, enquanto, no melhor caso, possui $O(n^2)$.

Por fim, temos o processo de ordenação. O método utilizado possui complexidade $nlogn$, entretanto, no pior caso, a complexidade é $O(n^2)$. Entretanto, pontuamos que as medidas citadas anteriormente, como um tamanho mínimo para as partições e a tomada dos pivôs a partir da mediana de um subvetor de *m* posições são tomadas a fim de diminuir a ocorrência desse pior caso, e aprimorar o algoritmo. Tanto o *bolha* quanto o *selectionsort* possuem complexidade $O(n^2)$, porém, para vetores pequenos, são mais eficientes do que o *quicksort*, computacionalmente falando.

Por fim, podemos sintetizar que todo o programa possui complexidade $O(n^3)$, entretanto, ainda assim, oferece um bom desempenho, à medida que as palavras inseridas tenham um tamanho razoável, assim como o número de palavras também seja ponderado.

$$4O(n) + 5O(n^2) + nlog(n) + 3O(n^3) = O(n^3)$$

3.2. Espaço:

A complexidade de espaço desse programa pode toda ser pensada em torno da sua estrutura de dados: O *array.* Essa Estrutura não é estática, mas sim dinâmica. Ou seja, em termos de memória há uma espécie de perda em relação à implementação dinâmica. Entretanto, enfatiza-se que todos os *arrays* utilizados foram alocados com o número exato de posições necessárias. O que isso significa? Isso implica que, para todos os vetores, foram realizadas contagens - já explicitadas anteriormente - que indicavam o número de posições necessárias. Por exemplo, para alocar as palavras dentro do array *vetPalavras* foram contadas quantas palavras estavam no arquivo de entrada, e, em seguida, o array foi alocado. Portanto, é perceptível, que, em termos de memória, há um bom custo, ou seja, há uma "boa" complexidade de espaço. Como foram utilizados os vetores estáticos, a complexidade de espaço será da ordem de O(n).

4. Estratégias de Robustez:

Todo programa mais robusto exige mecanismos para evitar bugs. Desse modo, para a depuração foi utilizado o gdb. As estratégias de robustez nesse programa consistem em, principalmente, "tratar" das palavras inseridas "inadequadamente". Ou seja, palavras que possuírem caracteres indesejados, devem ser corrigidas, removendo-os. Isso é feito através da função *fix_word*, apresentada anteriormente. Entretanto, outros tratamentos surgem ao longo do programa. Em primeiro lugar, exigir do usuário todos as entradas da linha de comando. Ou seja, o programa não roda, caso não sejam inseridas todas as *flags*, -s, -m ,-o, -i. Isso é necessário para o funcionamento do programa, pois são acompanhadas de informações fundamentais para o algoritmo. São recusadas as entradas para *m* negativo, ou nulo, assim como para *s.* Além disso, para os casos em que *m* é maior que o tamanho da partição, o programa opta pela ***mediana de 1***, ou seja, ao invés de fazer a mediana de *m* elementos dentro de um subvetor, apenas a mediana do vetor principal é escolhida. O mesmo é válido para um *m* menor que *s.* Nesse caso, um método alternativo - *selectionsort* - será utilizado para ordenar o vetor.

5. Testes:

Os testes realizados visam dois objetivos: Estimar a complexidade do programa para entradas de grande porte e o seu funcionamento, do ponto de vista de localidade de referência e depuração de desempenho. Em primeiro lugar, foram utilizadas as entradas fornecidas, para

testar o funcionamento do programa. Após finalizar a implementação, essas entradas foram modificadas da seguinte maneira:

O arquivo *7.tsi.i* foi alterado, ao passo que *n* repetições do texto foram testadas. Ou seja, um texto com 64 repetições do texto inicial, um com 128 e um com 256 foram testadas. No caso da localidade de referência, foi utilizada a entrada com 256 repetições do texto disponível no arquivo citado anteriormente. Esses testes auxiliaram na determinação do custo computacional do programa, assim como o seu tempo de execução. Quanto maior o desafio, mais necessário é um controle em relação à sua estabilidade e segurança de suas respostas. Por esse motivo, foram realizados múltiplos testes, possibilitando assegurar a qualidade do algoritmo.

5.1. Tempo de Execução:

A análise do Tempo de Execução do algoritmo implementado teve como base a variação na entrada dos dados, ou seja, no texto presente no *input*. Como citado anteriormente, foram utilizadas variações da entrada *7.tsi.i* disponibilizada. Para o teste do tempo de execução, foram testadas variações com 64, 128 e 256 e 512 repetições do texto, todas com parâmetro s = 3 e m = 4, sendo que os testes foram realizados na mesma máquina, sob mesmas condições. O gráfico a seguir apresenta os resultados:

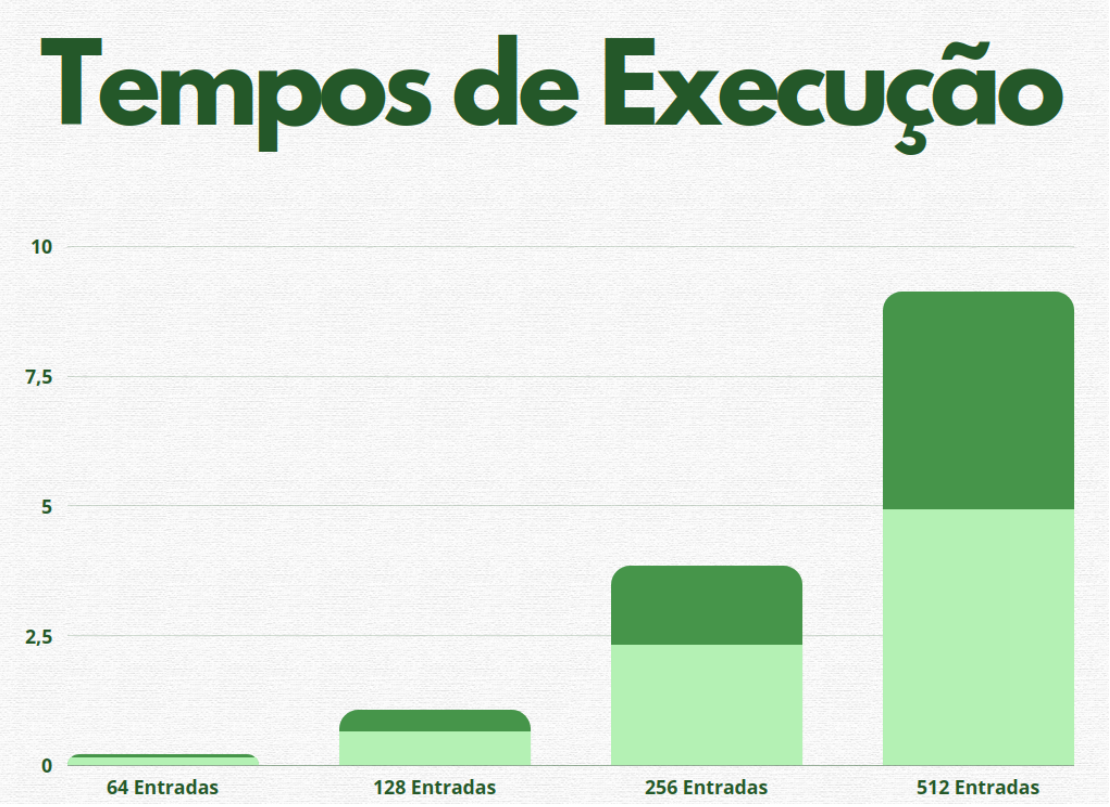


Figura 4. Variação dos tempos de execução

A partir deste gráfico, percebe-se que os tempos de execução são extremamente afetados pelo tamanho das entradas. Quanto maior o *input*, maior o tempo de execução. Isso já era esperado, ao passo que a complexidade do programa não ficou tão desejada, ou seja, de ordem $O(n^3)$.

6. Análise Experimental:

Agora que os resultados já foram apresentados, faremos a análise. Para facilitar a compreensão, iremos separá-la em subseções.

6.1. Desempenho Computacional:

O desempenho computacional é bastante afetado pela forma que o programa é implementado. Funções otimizadas, ou seja, com complexidades preferenciais, como *O(log n)* oferecem um melhor desempenho computacional ao programa. Contudo, como citado anteriormente, muitas funções dentro do programa possuem uma complexidade de ordem *O(n²) - e algumas O(n³),* o que significa que, computacionalmente, dependendo do número de operações a serem realizadas, o programa pode ser bastante custoso em termos de processamento.

A métrica utilizada levou em consideração as análises de complexidades das funções citadas nessa documentação e para análise, foram utilizados arquivos de textos que são variações das entradas disponibilizadas.

O algoritmo leva em consideração basicamente uma coisa: O arquivo de entrada. Logo, o volume de dados dentro desse arquivo é o que irá afetar o desempenho computacional do programa. Portanto, é fundamental enfatizar: As entradas foram geradas a partir de variações da entrada padrão, de modo que fosse possível medir como as entradas utilizadas afetaram as operações realizadas, ou seja, evitando que houvesse algum tipo de manipulação. Contudo, é fato que, as operações podem ser mais simples, ou mais complexas, dependendo das entradas. Por exemplo, uma entrada com 200 palavras terá um custo computacional muito menor do que uma entrada com 2000. Ainda que geradas aleatoriamente, as entradas do programa afetam seu desempenho.

A partir da análise da saída do *gprof,* percebe-se um fenômeno não tão óbvio: a porção da função principal, *main,* não é a mais custosa computacionalmente. Isso é

surpreendente, ao passo que dentro dela diversas funções de alto custo computacional são utilizadas.

É fundamental compreender que, apesar da função principal ser extremamente custosa, a ordenação é sempre um problema computacionalmente custoso. E ***atenção***, fica claro, pela análise gprof, que, mesmo utilizando um método de ordenação ***extremamente eficiente***, a comparação entre os elementos ainda é custosa. É importante mencionar o fato de que foram utilizadas algumas estratégias que amenizassem os custos, e, de certa forma, esse processo foi eficiente. A entrada utilizada tinha mais de ***2000 linhas,*** com 256 repetições da entrada *7.tsi.i,* e ainda assim, o tempo foi de - aproximadamente - 6 segundos de execução. Em um certo nível, esse é um tempo de execução satisfatório para realizar a contagem e ordenação de um grande número de palavras, a partir de uma nova ordem.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 60.31      3.56     3.56                             __gnu_cxx::__enable_if<std::__is_char<char>::__value, bool>::__type
std::operator==<char>(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&, std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> > const&)
 39.64      5.91     2.34                             main
  0.17      5.92     0.01                             std::char_traits<char>::compare(char const*, char const*, unsigned long)
  0.00      5.92     0.00    29539     0.00     0.00  word::~word()
  0.00      5.92     0.00    29435     0.00     0.00  word::word()
  0.00      5.92     0.00      736     0.00     0.00  word::operator=(word const&)
  0.00      5.92     0.00      545     0.00     0.00  bool std::operator< <char, std::char_traits<char>, std::allocator<char> >(std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > const&, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      5.92     0.00      376     0.00     0.00  bool std::operator><char, std::char_traits<char>, std::allocator<char> >(std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > const&, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      5.92     0.00      156     0.00     0.00  std::remove_reference<word&>::type&& std::move<word&>(word&)
  0.00      5.92     0.00      104     0.00     0.00  word::operator=(word&&)
  0.00      5.92     0.00       93     0.00     0.00  word::defineSenha(char*)
  0.00      5.92     0.00       52     0.00     0.00  word::word(word&&)
  0.00      5.92     0.00       52     0.00     0.00  word::word(word const&)
  0.00      5.92     0.00       52     0.00     0.00  std::enable_if<std::__and_<std::__not_<std::__is_tuple_like<word> >, std::is_move_constructible<word>,
std::is_move_assignable<word> >::value, void>::type std::swap<word>(word&, word&)
  0.00      5.92     0.00       37     0.00     0.00  Bolha(word*, int)
  0.00      5.92     0.00       37     0.00     0.00  word::Particao(int, int, int*, int*, word*, int)
  0.00      5.92     0.00       26     0.00     0.00  selectionsort(word*, int, int, int)
  0.00      5.92     0.00        1     0.00     0.00  _GLOBAL__sub_I_ZN4word11defineSenhaEPc
  0.00      5.92     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      5.92     0.00        1     0.00     0.00  word::Ordena(int, int, word*, int, int)
  0.00      5.92     0.00        1     0.00     0.00  word::QuickSort(word*, int, int, int)
```

Figura 5. Custo computacional *gprof*

Outro ponto a se analisar da saída mostrada anteriormente é a quantidade de chamadas para o método construtor da Classe *word.* Isso era esperado, ao passo que tivemos ***inúmeras*** palavras ao longo da entrada. Para cada uma delas houve o processo de "transferência" de *arrays* explicada anteriormente. Logo, é evidente que haveria um grande número de chamadas desse método, entretanto, essas instanciações possuem mais impacto para a memória do computador, algo que será abordado logo em sequência.

```
void Bolha(word *v, int n) {
    int i, j;
    for(i = 0; i < n-1; i++)
        for(j = 1; j < n-i; j++)
            if (v[j].senha < v[j-1].senha)
                Troca(v[j-1], v[j]);
}

void selectionsort(word *array, int esq, int dir, int n) {
   int i, j, min;
   n += esq; j = dir;

   for (i = esq; i < n - 1; i++) {
      min = i;
      for (j = i + 1 ; j < n; j++) {
         if (array[j].senha < array[min].senha)
            min = j;
      }
      swap(array[i], array[min]);
   }
}
```

```
void word::Particao(int Esq, int Dir, int *i, int *j, word *A, int parametroMediana) {
    word x, w;
    *i = Esq; *j = Dir;
    word xMedian[parametroMediana]; //crio o subvetor para descobrir a mediana e pegar como pivo
    if(Dir - Esq >= parametroMediana) {
        //populo o novo vetor
        for(int t = 0; t < parametroMediana; t++)
            xMedian[t] = A[Esq + t];

        Bolha(xMedian, parametroMediana); //ordeno o subvetor

        x = xMedian[parametroMediana/2]; /* obtem o pivo x */
    }
    x = A[(*i + *j)/2]; /* obtem o pivo x */
    do
    {
        while (x.senha > A[*i].senha) (*i)++;
        while (x.senha < A[*j].senha) (*j)--;
        if (*i <= *j)
        {
            w = A[*i]; A[*i] = A[*j]; A[*j] = w;
            (*i)++; (*j)--;
        }
    } while (*i <= *j);
}
```

```
void word::Ordena(int Esq, int Dir, word *A, int maxParticao, int parametroMediana){
    int i;
    int j;
    int maxTam = Dir - Esq + 1;

    if(maxTam <= maxParticao)
        selectionsort(A, Esq, Dir, maxTam);
    else {
        Particao(Esq, Dir, &i, &j, A, parametroMediana);
        if (Esq < j) Ordena(Esq, j, A, maxParticao, parametroMediana);
        if (i < Dir) Ordena(i, Dir, A, maxParticao, parametroMediana);
    }
}

void word::QuickSort(word *A, int n, int maxParticao, int parametroMediana)
{
    Ordena(0, n-1, A, maxParticao, parametroMediana);
}
```

Figura 6. Função *quicksort* e seus auxiliares

Portanto, os resultados obtidos condizem com a análise de complexidade abordada nessa documentação. Como dito anteriormente, algumas funções não possuem uma complexidade pouco custosa computacionalmente, entretanto, os algoritmos utilizados buscam otimizar o desempenho, junto a simplicidade. A melhora desse programa, em termos de desempenho computacional poderia vir principalmente de dois fatores: O uso de Estruturas de Dados dinâmicas, oferecendo menos custo à memória e uma função main mais otimizada, com métodos mais bem distribuídos ao longo do programa, com custos computacionais mais eficientes, evitando **ao máximo** qualquer função próxima de $O(n^3)$. Em ambos os casos, o desempenho computacional poderia ser otimizado, mas não necessariamente haveriam ganhos em todos os âmbitos. Por exemplo, o código se tornaria consideravelmente mais complexo.

6.2. Padrão de acesso à Memória e Localidade de Referência:

Para a análise de acesso à Memória e localidade de referência foi utilizada a entrada *7.tsi.i*, disponibilizada no enunciado deste TP. Essa escolha busca encontrar um intervalo entre um programa de tamanho razoável, oferecendo desempenho suficiente para análise de localidade.

A análise de localidade de referência parte da divisão do programa em fases. Ou seja, para facilitar a interpretação dos gráficos e a análise do comportamento do programa, ele foi dividido em “partes” que refletem o seu funcionamento. A *fase 1* é o momento de leitura e armazenamento dos dados de entrada. Ou seja, nessa porção do programa tem-se, simultaneamente, a leitura e a escrita dos dados vindos da memória principal em uma grande

quantidade. Durante a *fase 2* temos a criação e preenchimento do *array* final das palavras, em que não há repetição. Nesse momento, há também um processo de leitura - do vetor anterior - e escrita, dentro do novo vetor. Além disso, nessa fase há o procedimento de criação das senhas, em que as palavras são lidas dentro do *array* e convertidas em novas, sobrescrevendo as antigas. Por fim, temos a *fase 3*, que é a final. Essa parte do programa é responsável pela ordenação dos elementos do vetor principal. Esse momento é predominantemente de leitura dos dados dentro da memória, entretanto, há também a escrita, uma vez que temos que ordená-los novamente, ao passo que eles estavam fora de posição, de acordo com a nova ordem sugerida.

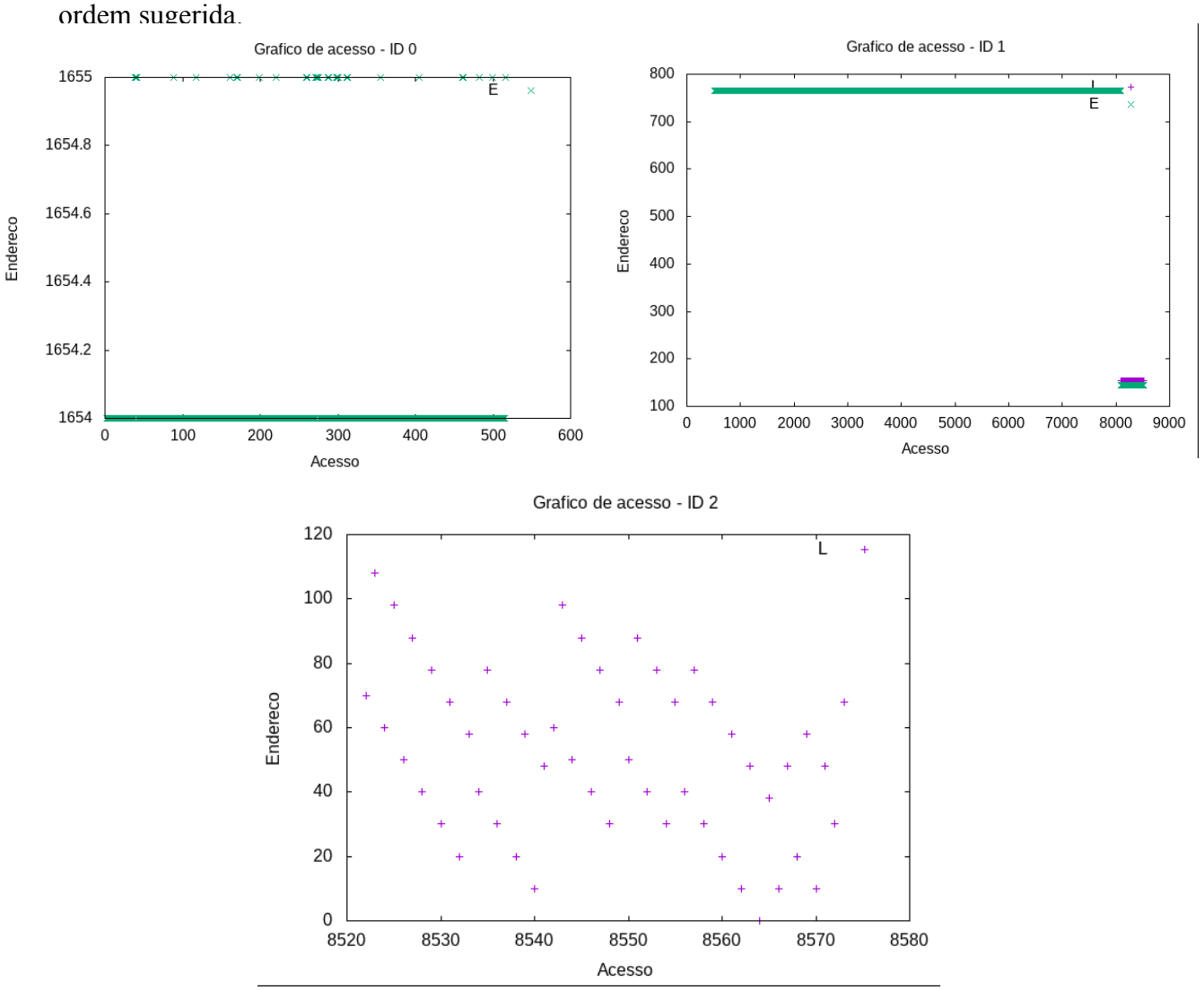


Figura 7. Gráficos de acesso à memória

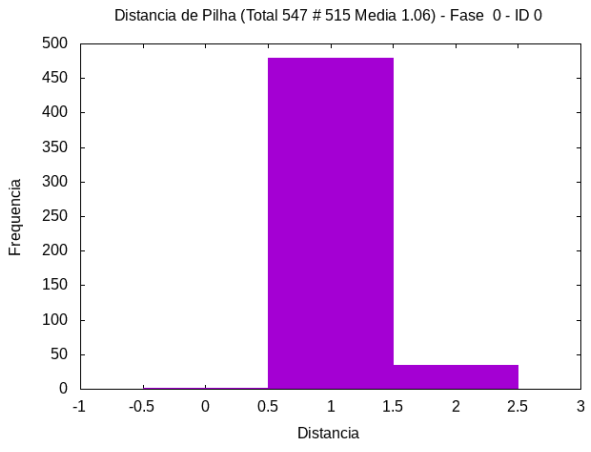
Vamos começar essa análise na ordem sequencial das fases, que também corresponde à ordem lógica e implementação do programa. Percebe-se no gráfico de **ID 0** como a localidade de referência da memória está bem aplicada. Isso é visível pois temos cerca de 500 acessos, todos muito próximos em termos de memória, ou seja, todos em um mesmo

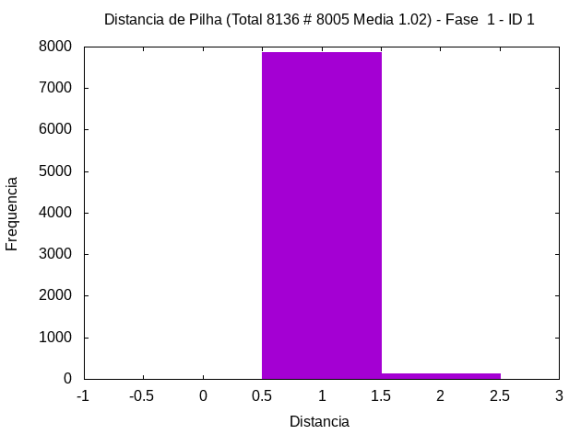
endereço. Do ponto de vista computacional isso é extremamente interessante, pois dessa forma, o acesso se torna *extremamente mais rápido*. É evidente que temos alguns processos de escrita que não são tão bem localizados, ao passo que estão mais dispersos. A partir de uma análise do código, pode-se dizer que, muito provavelmente, essa dispersão corresponde ao método *fix_word*, ou seja, aquele responsável por lidar com as intempéries das palavras. Ainda assim, em termos de localidade de referência, pode-se dizer que a *fase 0* está muito bem.

Dando sequência, percebe-se que o mesmo dito anteriormente vale para a fase seguinte. Contudo, dessa vez é importante pontuar que não existem acessos dispersos ao longo dessa fase. Tanto a escrita quanto a leitura dos dados foram extremamente eficientes em termos de localidade de referência. Também é interessante perceber que a parte de criação das chaves se encontra em uma porção de memória distinta, uma vez que se encontra em um método distinto, dentro da Classe *word.* Essa percepção é interessante, pois, apesar de ocupar uma posição de memória distinta, ainda assim, a localidade de referência é bem aproveitada.

Por fim, a *fase 2* é completamente diferente das anteriores. Como dito anteriormente, essa fase corresponde aos métodos de ordenação, ou seja, principalmente ao *quicksort*. Um brevíssimo resumo sobre esse método é importante, principalmente para compreender seu funcionamento, o que é fundamental para a análise de localidade de referência. O *quicksort* é um método ***recursivo***, logo, ele conta com chamadas para ele próprio até uma condição de parada estabelecida. Esse procedimento consiste em definir um pivô e dividir o vetor a ser ordenado em partições menores, com pivôs internos à ela, sempre buscando deixar os elementos menores à esquerda do pivô e os maiores à direita. Sendo assim, partimos para a análise do gráfico. Percebe-se que, em termos de localidade de referência, a *fase 2* não é interessante. Isso se deve ao fato dos acessos estarem ***extremamente dispersos***. A grande explicação para isso é o método ser recursivo. Ou seja, as diversas chamadas para a própria função são interpretadas pela memória de um modo que a localidade de referência não é tão bem aproveitada, logo, temos o resultado visto anteriormente.

Agora, para uma análise mais ampla, visando a quantificação dos acessos, faremos uma análise das distâncias dessa mesma operação:





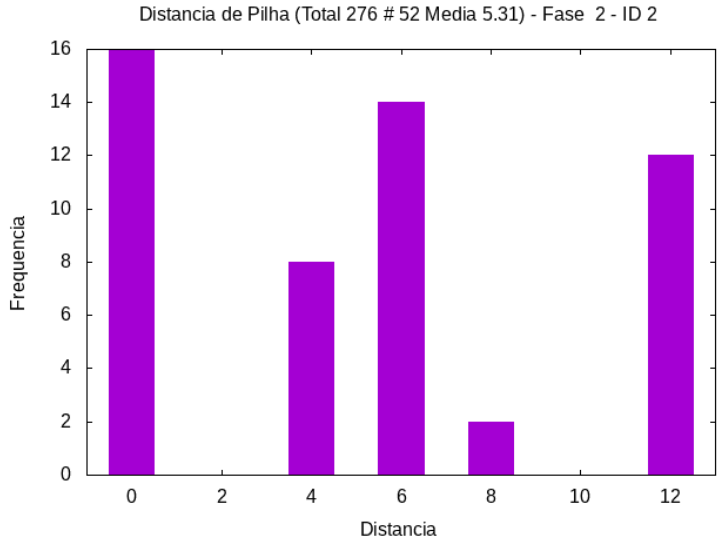
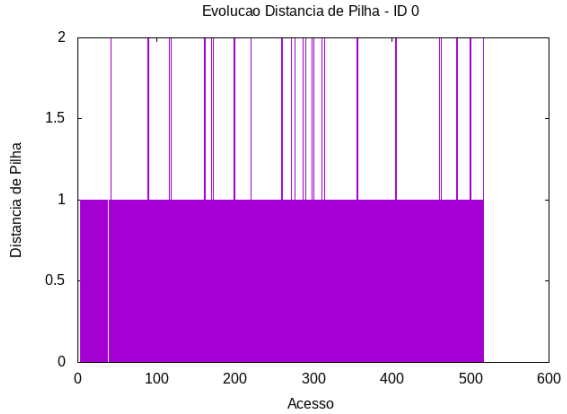
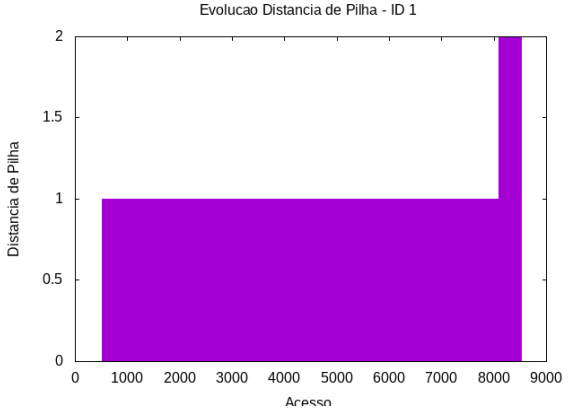


Figura 8. Gráficos de distância de pilha

Como era de se esperar, os gráficos de distância de pilha das fases 0 e 1 são extremamente similares. Isso é refletido pela boa localidade de referência, ou seja, ao passo que os acessos são próximos, temos uma distância de pilha menor, algo muito positivo para execução do programa. A partir de uma pequena variação na frequência de acessos um pouco mais espaçados, a fase 1 e a fase 0 são muito semelhantes, e refletem a boa execução do programa. De maneira oposta, o gráfico de distância de pilha da *fase 2* é bastante mais complexo e distinto das anteriores. Novamente, era esperado que o gráfico dessa fase fosse distinto, uma vez que o mapa de acesso é bastante mais robusto que os anteriores. Nesse caso, temos muitos acessos dispersos e com diferentes distâncias de pilhas, o que não é tão interessante do ponto de vista de uso de memória, como já foi explicado anteriormente. Quanto maior as distâncias de pilha, maior as porções de memória ocupadas. Quanto menos ocupação da memória do computador um programa possui, mais interessante é a sua implementação.

Por fim, apenas citamos os gráficos de evolução de distância de pilha que confirmam o que foi dito anteriormente:





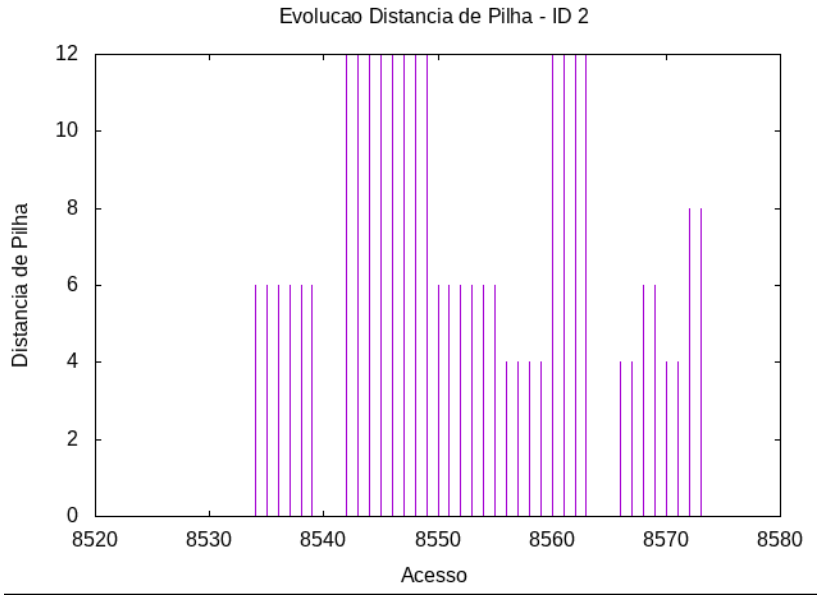


Figura 9. Gráficos de evolução de distância de pilha

7. Conclusão:

O programa implementado envolveu o desenvolvimento de um algoritmo capaz de receber entradas equivalentes a um texto e retornar uma saída, ordenando as palavras de acordo com um novo alfabeto fornecido e contar suas quantidades. Este trabalho definitivamente instigou o desenvolvimento de diversas técnicas que envolvem a programação. Esse desenvolvimento extrapolou apenas o desenvolvimento em si, mas também atingiu níveis como abstração, percepção da realidade, pesquisa, solução de problemas e o planejamento de software.

Sem dúvida, o criar um programa com um certo nível de complexidade como esse, partindo do zero, exige um certo planejamento, ao passo que seja possível uma modularização de código mais eficiente, permitindo atingir melhores resultados mais a frente, tanto em relação ao aproveitamento das Classes e Objetos, quanto aos resultados finais em si. Nesse sentido, a abstração também é desenvolvida, uma vez que é importante modularizar o código, à medida que, a implementação de alguns métodos e procedimentos não fiquem evidentes a um possível usuário final, mas apenas interesse o seu uso. Entretanto, em um sentido mais amplo, a abstração é fundamental para conseguir capturar as nuances de um problema e transformá-las em Classes, TADs, métodos e atributos dentro do código.

Além disso, ainda na implementação, esse trabalho proporcionou o desenvolvimento do uso de Estruturas de Dados de forma mais concreta, ou seja, exigindo a implementação dessas estruturas.

Por fim, houveram ganhos '*extras'* ao longo do desenvolvimento. Em um projeto amplo é esperado o surgimento de *bugs* e erros sucessivos. A fim de contornar esses problemas, é importante a pesquisa e a depuração. Logo, esse programa ainda proporcionou o desenvolvimento da capacidade de encontrar os problemas e solucioná-los, o que é fenomenal, dado que essa é uma competência fundamental dentro da Ciência da Computação.

Sendo assim, a implementação do TP2 foi fundamental para o meu crescimento, tanto profissional quanto pedagógico. Lidar com novas ferramentas ofereceram uma oportunidade para aumentar meu conhecimento, e esse aprendizado me instigou a correr atrás de novos instrumentos, o que aumentou o meu repertório. Portanto, a implementação do TP2 ofereceu grande margem de aprendizado, em múltiplas competências, como citado anteriormente.

8. Bibliografia:

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Java e C++:˜ Capítulo 3: Estruturas de Dados Básicas. Editora Cengage.

9. Instruções para compilação e execução:

A compilação do programa é "automática", via Makefile, implementado para esse programa. O comando "*make all"* deve ser realizado via terminal na raiz do diretório TP2. Esse comando irá compilar o programa, gerando o executável e todos os arquivos **.out* necessários para o funcionamento. Caso seja necessário "*reiniciar"* a compilação, o comando "*make clean"* apaga todos os **.o* e **.out*, permitindo uma recompilação.

Após compilar, o programa implementado terá o seu '*executável'* disponível na pasta bin. Esse executável terá nome *tp.exe* e **nessa pasta deve estar contido o arquivo de entrada** que com **NOME** *indicado pela flag -i.*

ATENÇÃO: *Caso o arquivo de entrada.txt não esteja presente na pasta 'bin', a execução do programa será interrompida, e uma mensagem de erro irá indicar o problema.*

A execução do programa é bastante simples, e a saída **deve ser determinada pelo usuário,** por meio da flag *-p,* e também ficará disponível na pasta bin.

Seguem as flags esperadas via linha de comando:

-s: Determina o tamanho máximo da partição. *Exemplo de uso:*

./executavel -s (...)

-o: Determina o nome do arquivo de saída do programa. *Exemplo de uso:*

./executável -o [OUTPUT.o] (...)

-m: Determina o número de elementos usados para calcular a mediana e descobrir o pivô. *Exemplo de uso:*

./executável -m (...)

-i: Determina o nome do arquivo de entrada. *Exemplo de uso:*

./executável -i [INPUT.i] (...)

ATENÇÃO: TODAS *as flags citadas anteriormente são* ***OBRIGATÓRIAS*** *para o funcionamento do programa. Caso alguma dessas flags não estejam presentes, o programa* ***não irá executar.***

ATENÇÃO: *A ordem em que cada flag aparece* ***NÃO É RELEVANTE PARA O FUNCIONAMENTO DO PROGRAMA.***