UNIVERSIDADE FEDERAL DE MINAS GERAIS

DCC005 - ALGORITMOS E ESTRUTURAS DE DADOS III

HERNANE BRAGA PEREIRA

2014112627

TRABALHO PRÁTICO - 1

Belo Horizonte - MG
Maio/2018

INTRODUÇÃO

O objetivo deste trabalho é a resolução do problema de Mathias, que deseja ordenar as linhas de um documento de texto escrito por macacos, onde todas as linhas possuem um número fixo de caracteres e Mathias possui uma limitação de memória RAM em seu computador. Mathias entregou o programa quase completo, faltando apenas a função de ordenação que será implementada em um arquivo chamado *sort.c*.

SOLUÇÃO DO PROBLEMA

Para solucionar o problema apresentado é necessário utilizar um método de ordenação externa, que consiste em ordenar um arquivo de tamanho maior que a memória interna disponível. A memória RAM do computador é tratada como a memória interna, enquanto o HD é a memória externa.

O método consiste em ler o arquivo original de forma fracionada, ou seja, os dados são lidos e armazenados na memória externa, através de arquivos temporários. Os arquivos temporários criados são ordenados, e ao final, são unificados de tal forma que o resultado é o arquivo original, de forma ordenada.

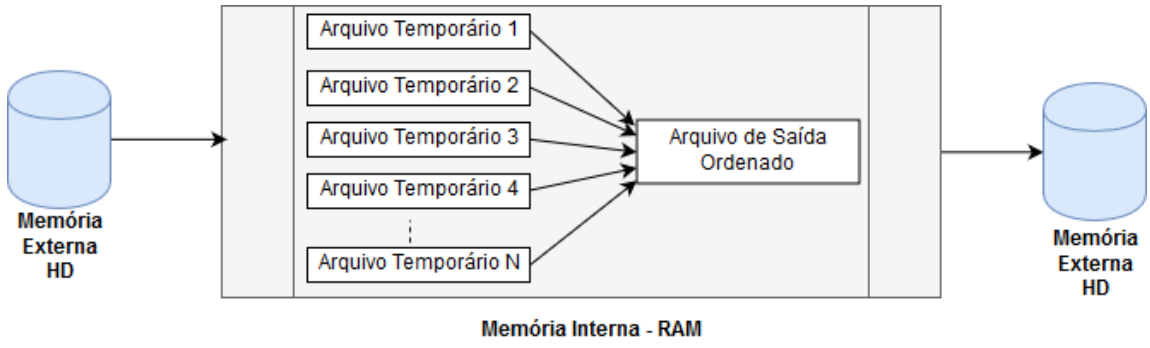

Figura 1. Exemplo de um algoritmo de ordenação externa

2.1 Modelagem do Problema

Para solucionar o problema proposto, foi utilizado o algoritmo Merge Sort externo, pois ele utiliza a técnica de intercalação balanceada por vários caminhos para ordenar os arquivos. Esta estratégia foi adotada pois não se sabe de antemão qual é a quantidade de linhas que devem ser ordenadas, apenas o tamanho da mesma. Devido a este fator, não é possível prever a quantidade de arquivos temporários que serão gerados pelo algoritmo, por isso a técnica de intercalação balanceada é a melhor escolha quando comparada com os métodos de intercalação polifásica e quicksort externo.

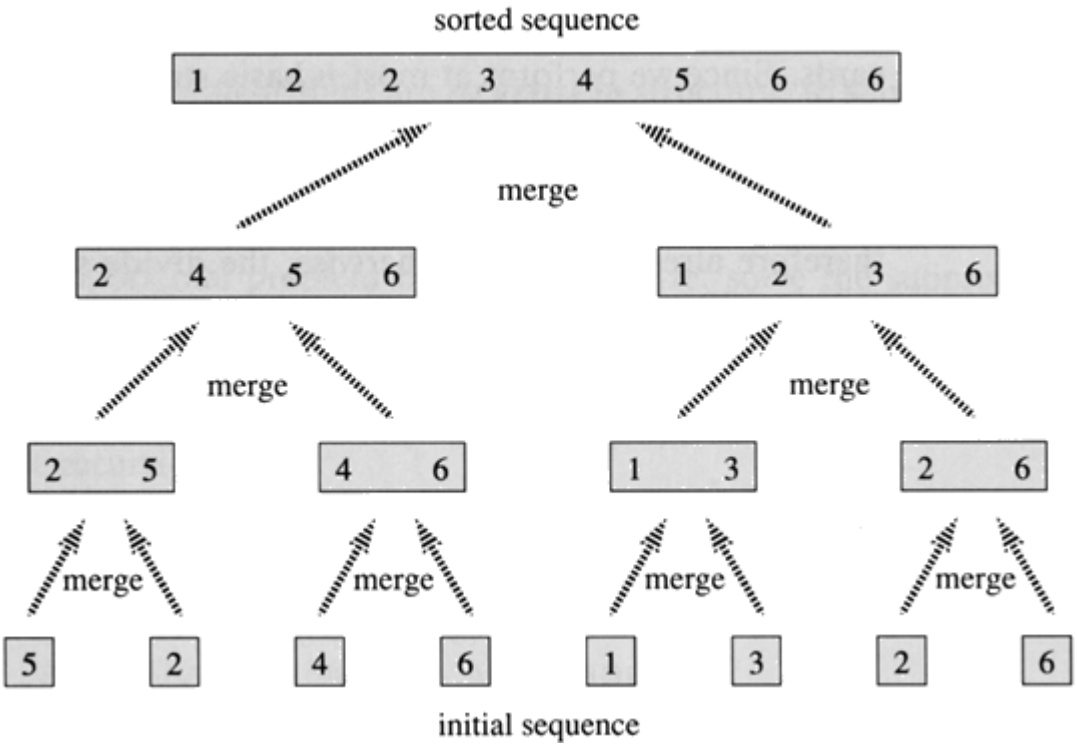

Figura 2. Exemplo de funcionamento do Merge Sort externo

A implementação do merge sort externo foi feita da seguinte forma:

1. Lê-se uma parte de tamanho B do arquivo de entrada, que possui tamanho N, e ordena-se os dados na memória interna utilizando o método quicksort.
2. Cria-se um arquivo temporário com os valores ordenados.
3. Repete-se os passos 1 e 2, até que o arquivo tenha sido completamente lido e sejam criados F arquivos temporários ordenados.

4. Lê-se os primeiros G (= B / (F+1)) valores de cada arquivo temporário e os compara. O menor valor dentre eles, é inserido em um buffer de tamanho G. Esta operação se repete até que o buffer esteja cheio.
5. Salva-se o buffer cheio no arquivo de saída, pois temos a garantia que ele está ordenado. Em seguida, esvazia-se o buffer. Os passos 4 e 5 se repetem até que todos os arquivos temporários tenham sido completamente percorridos. Ao final, o arquivo de saída estará completo e ordenado.

Para implementar o algoritmo descrito acima, foi criada uma estrutura *arquivo* e as seguintes as funções:

- *salvaArquivo* – Gerencia a escrita nos arquivos, sabendo quando é necessário pular linhas, ou imprimir o número total de caracteres no arquivo de saída.
- *criaArquivosOrdenados* - Retorna o numero de arquivos temporários ordenados que foram criados, a partir do arquivo de entrada.
- *preencheBuffer* - Preenche o buffer da estrutura arquivo, baseado no arquivo temporário ordenado já criado na função *criarArquivosOrdenados*
- *procuraMenor* – Função que procura o menor valor dentro de todos os arquivos temporários abertos.
- *merge* – Une os arquivos temporários em um único arquivo de saída.
- *external_sort* – Chama a função *criarArquivosOrdenados, merge* e deleta todos os arquivos temporários criados ao longo do processo de ordenação.
- *a_menor_que_b* – Compara 2 cadeias de caracteres e retorna qual deve vir primeiro na ordenação. É usada nos testes de comparação.

Pseudo-código das principais estruturas e funções apresentadas:

```
// Estrutura para gerenciar buffers e arquivos temporários
struct arquivo{
    FILE *f;        // Arquivo que está sendo trabalhado no momento
    int pos, MAX;   // pos -> Posição atual dentro do arquivo. MAX ->
Tamanho máximo de buffer
    char **buffer;  // Linha de caracteres que será escrita no arquivo
};
```

Código 1: Estrutura de arquivo

```
// Funcao que retorna quantos arquivos ordenados foram criados
int criaArquivosOrdenados(const char *arquivo_entrada){
        Abrir_arquivo(arquivo_entrada);
        Ler_do_arquivo("Numedo de caracteres por linha");
 // Alocacao do vetor que lerá o arquivo de entrada, com tamanho baseado na restricao do
usuario
    char** vetor =  mathias_malloc( max_linhas * sizeof(char**));
    for ( i = 0; i < max_linhas ; i++){
         vetor[i] = mathias_malloc( (len+1) * sizeof(char));
     }
    /* Leitura do arquivo de entrada */
    Enquanto != EOF{
        vetor[total] -> Ler_do_arquivo("linha inteira");
        total++;
 // Quando o buffer estiver cheio, dados sao armazenados no arquivo temporario, ja ordenados
        if(total == max_linhas){
            num_arquivos++;
            Cria_Arquivo_Temporario("Temp - 01");
            Ordena(vetor);
            salvaArquivo(Arquivo_Temporario, vetor);
        }}
  // Caso algum dado tenha sobrado sem ser salvo em um arquivo, ele é salvo neste momento
    if(total > 0){
        num_arquivos++;
        Cria_Arquivo_Temporario("Temp - 01");
        Ordena(vetor);
        salvaArquivo(Arquivo_Temporario, vetor);}
    Fechar_arquivo(arquivo_entrada);
    // Desalocando o vetor
    for ( i = 0; i < max_linhas; i++){
        mathias_free(vetor[i]);}
```

Código 2: Função *criaArquivosOrdenados*

A variável *max_linhas*, é definida a partir da memória disponível informada pelo usuário e pelo número de caracteres por linha. Seu objetivo é de maximizar o número de linhas que poderão ser armazenadas no buffer.

```
// Esta função preenche o buffer da estrutura arquivo, baseado no arquivo temporário ordenado
já criado na função criarArquivosOrdenados
void preencheBuffer(struct arquivo* arq, int tam_buffer){
    int i;
    if(arq->f == NULL)
        return;
    arq->pos = 0;
    arq->MAX = 0;
    for(i=0; i < tam_buffer; i++){
        if(!feof(arq->f)){
            Escrever_no_Arquivo(arq, arq->buffer[arq->MAX][0]);
            arq->MAX++;
        }
        else{
            Fechar_arquivo(arq->f);
            arq->f = NULL;
            break;
        }}}
```

Código 3: Função *preencheBuffer*

```
// Esta função retorna (1) caso encontre o menor valor dentro de todos os arquivos temporários
abertos abertos
int procuraMenor(struct arquivo* arq, int num_arquivos, int tam_buffer, char** menor, int
qtdBuffer){
    int i, idx = -1;
 // Procura o menor valor dentro de todos os arquivos temporários abertos, começando pela
posicao[0] de todos eles
    for(i=0; i < num_arquivos; i++){
        if(arq[i].pos < arq[i].MAX){
            if(idx == -1)
                idx = i;
            else{
                 if( a_menor_que_b( arq[i].buffer[arq[i].pos], arq[idx].buffer[arq[idx].pos]) )
                idx = i;
            }}}
 // Caso tenha sido encontrado um menor valor, ele é salvo para ser gravado no arquivo final e
atualiza-se a posicao atual do arquivo
    if(idx != -1){
        strcpy(menor[qtdBuffer], arq[idx].buffer[arq[idx].pos]);
        arq[idx].pos++;
        // Preenche buffer novamente, caso ele fique vazio
        if(arq[idx].pos == arq[idx].MAX)
            preencheBuffer(&arq[idx], tam_buffer);
        return 1;}
    else
        return 0;}
```

Código 4: Função *procuraMenor*

```
// Funcao que junta os vários arquivos temporários em um único arquivo ordenado de saída
void merge(const char *arquivo_saida, int num_arquivos, unt tam_buffer){

    // Alocacao do Buffer que será escrito no arquivo final
    char** buffer = mathias_malloc( tam_buffer * sizeof(char**));
    for ( i = 0; i < tam_buffer ; i++){
        buffer[i] = mathias_malloc( (len+1) * sizeof(char));
    }

    // Essa estrutura gerencia todos os arquivos temporários que serão abertos
    Arquivo = Cria_Struct_Arquivo();

    // Enquanto existir um elemento de menor valor dentro dos arquivos, esta funcao os salva no
arquivo final de forma ordenada
    Enquanto (procuraMenor == 1){
        qtdBuffer++;
        if(qtdBuffer == tam_buffer){
            salvaArquivo(arquivo_saida, buffer);
            qtdBuffer = 0;
        }}
    // Caso tenha sobrado algum valor no buffer que não foi salvo no arquivo final, ele é salvo
neste momento.
    if(qtdBuffer != 0){
        salvaArquivo(arquivo_saida, buffer);
    }

    // Desalocando os buffers
    for (i = 0; i < tam_buffer; i++)
        mathias_free(buffer[i]);
    free(Arquivo);
}
```

Código 5: Função *merge*

3.1 Análise Teórica do Custo Assintótico de Tempo

Podemos realizar a análise de custo assintótico avaliando apenas a ordenação externa, pois o número de passadas no arquivo são o que importam na complexidade, fazendo com que o método de ordenação interna não implique na complexidade final do algoritmo. Sendo **n** o número de registros a serem ordenados,

m o tamanho da memória principal, **f** o número de caminhos intercalados, **b** o número de blocos ordenados e **P(n)** o número de passadas na fase de intercalação, que é definido por $P(n) = \log_f(b)$. O método de intercalação balanceada possui complexidade final de $P(n) = \log_f\left(O\left(\frac{n}{m}\right)\right)$

3.2 Análise Teórica do Custo Assintótico de Espaço

Para fazer a análise de custo de espaço, foca-se no tamanho de fitas, ou arquivos temporários, utilizados na ordenação. O método de intercalação balanceada possui complexidade de espeço de *2f,* onde f é o número de fitas.

4.1 Testes do código

A máquina utilizada para rodar os testes possui 6,0 GB de memória RAM, com processador Core™ i7-5500U CPU 2.40Hz. Foi utilizado o tempo que é impresso na parte “Resultados” para contabilizar o tempo gasto pelo algoritmo. Abaixo está uma tabela contendo 24 casos de testes, com suas respectivas informações de caracteres por linha e número de linhas por arquivo, além de dois gráficos que comparam a memória interna disponível com o tempo de execução do programa e o número de arquivos temporários gerados.

ARQUIVO TESTE	NUM CARACTERES	LINHAS ARQUIVO	ARQUIVO TESTE	NUM CARACTERES	LINHAS ARQUIVO
test_010.010_3.txt	10	10	test_040.010_3.txt	40	10
test_010.040_3.txt	10	40	test_040.040_3.txt	40	40
test_010.080_3.txt	10	80	test_040.080_3.txt	40	80
test_010.160_3.txt	10	160	test_040.160_3.txt	40	160
test_010.320_3.txt	10	320	test_040.320_3.txt	40	320

test_010.640_3.txt	10	640	test_040.640_3.txt	40	640
test_020.010_3.txt	20	10	test_080.010_3.txt	80	10
test_020.040_3.txt	20	40	test_080.040_3.txt	80	40
test_020.080_3.txt	20	80	test_080.080_3.txt	80	80
test_020.160_3.txt	20	160	test_080.160_3.txt	80	160
test_020.320_3.txt	20	320	test_080.320_3.txt	80	320
test_020.640_3.txt	20	640	test_080.640_3.txt	80	640

Tabela 1: Nome dos arquivos de testes contendo número de caracteres por linha e número máximo de linhas do arquivo

```
=--------------------------------------=
=------------ Informações -------------=
Entrada: testes/test_080.640_3.txt
Saida: resultados/test_080.640_3.txt.out
Memoria: 1
=--------------------------------------=
=------------ Resultados --------------=
Ordenado: SIM
Memoria: OK (0.9561 kb)
Tempo: 169.8480 milisegundos
=--------------------------------------=


=------------------------------------------------------------------------=
=------------------------- RESULTADOS -----------------------------------=
=------------------------------------------------------------------------=
SUCESSO!!
=------------------------------------------------------------------------=
hernane-linux@hernanelinux-X555LF:/media/hernane-linux/DATA/AEDS 3 - 2018_1/TPs/TP1/template$
```

Figura 3: Solução sendo aprovada em todos os casos de testes fornecidos

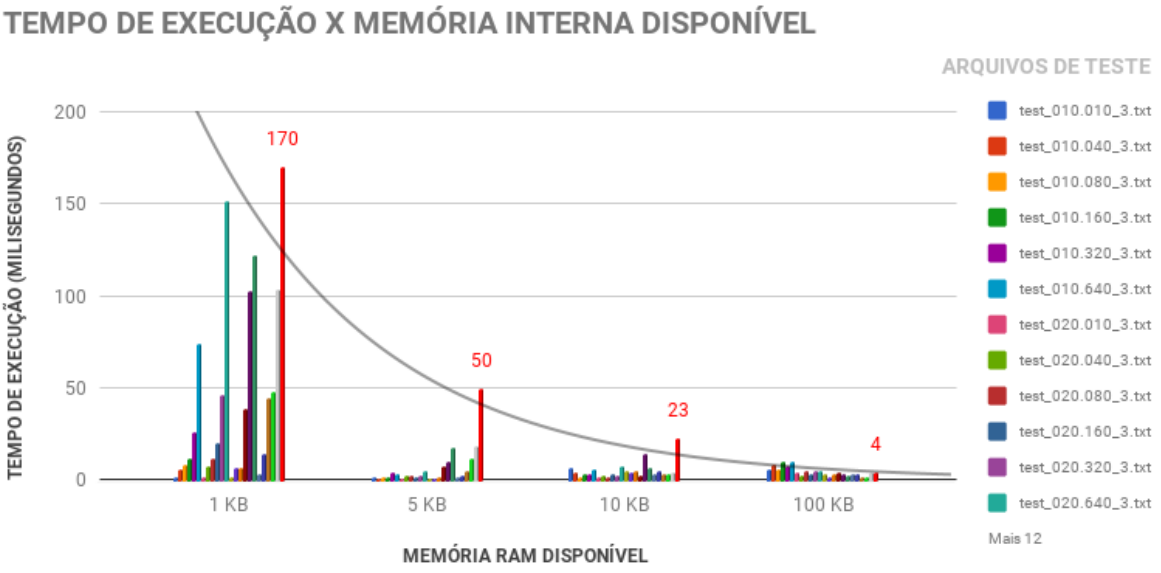

Gráfico 1: Tempo de execução x Memória interna disponível. Destaque em vermelho para um dos piores casos que possui 80 caracteres por linha e 640 linhas. A linha preta prova que o algoritmo possui comportamento exponencial, dado pela relação n/m.

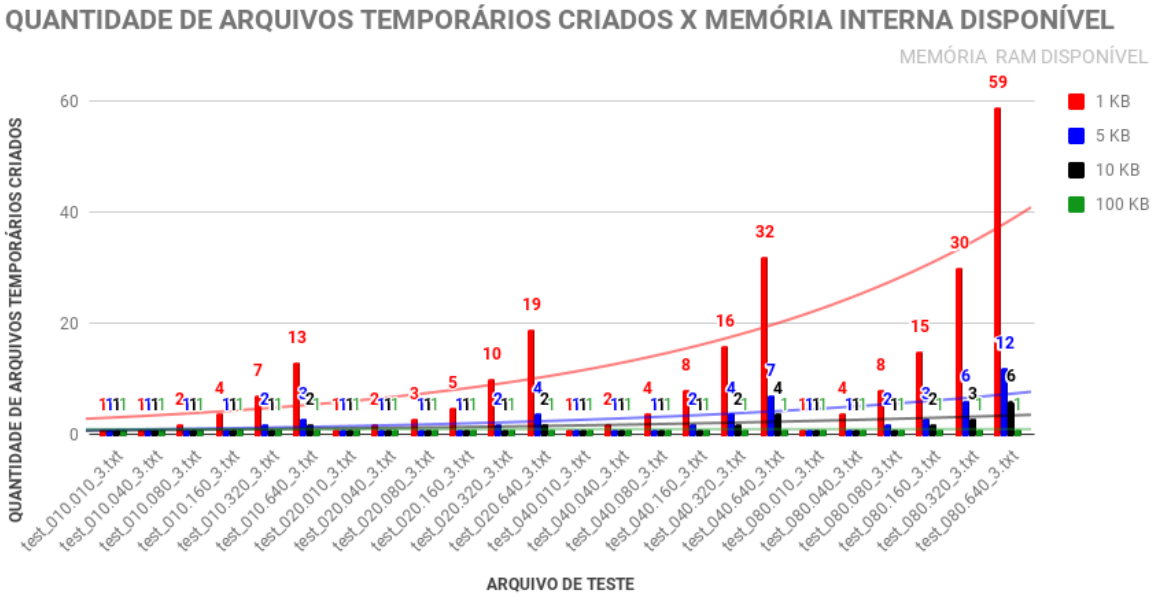

Gráfico 2: Quantidade de arquivos temporários gerados x Memória interna disponível

A partir do gráfico é possível notar que o programa tem tempo de execução que varia com o número de acessos à memória externa, o que comprova a análise teórica do item 3 desta documentação.

5.1 Conclusão

Neste trabalho, foi resolvido o problema de ordenar o arquivo de Mathias, que possui memória RAM limitada. O problema foi resolvido utilizando-se o algoritmo merge sort externo. A análise de complexidade teórica de tempo foi comprovada através de experimentos que utilizaram entradas grandes suficientes para analisar o comportamento assintótico.

REFERÊNCIAS

Ziviani, Nívio; Projeto de Algoritmos. 4ed. São Paulo: Pioneira. 1999

Programação descomplicada em C, playlist do YouTube:
https://www.youtube.com/watch?v=sVGbj1zgvWQ

Geeks for Geeks, External Sorting:
https://www.geeksforgeeks.org/external-sorting/

Artigo sobre ordenação externa:
https://en.wikipedia.org/wiki/External_sorting

External sorting using a heap structure:
http://web.eecs.utk.edu/~leparker/Courses/CS302-Fall06/Notes/external-sorting2.html

Programação de Computadores, YouTube:
https://www.youtube.com/watch?v=BbeEfYID_dM

Imagem 2 – Minimal Codes, Sorting Large Numbers of Elements (External Sort) in C++
https://minimalcodes.wordpress.com/2016/05/29/sorting-large-number-of-elements-external-sort-in-cpp/