Universidade Federal de Ouro Preto

Departamento de Computação

Augusto Ferreira Guilarducci (20.1.4012)

Caio Monteiro (20.1.4110)

Paulo Correa (20.1.4036)

Pedro Lucas Damasceno (20.1.4003)

Trabalho Prático II – Estrutura de Dados II

Implementação e análise experimental de métodos de ordenação em memória secundária

Ouro Preto, Minas Gerais

2021

SUMÁRIO

1. Introdução

Este relatório se refere ao segundo trabalho prático da disciplina Estrutura de Dados 2, realizado durante o período letivo 21.1, cujo objetivo consiste na implementação e estudo da complexidade dos seguintes métodos de ordenação externa apresentados em sala de aula:

1. Intercalação balanceada de vários caminhos (*2f* fitas) utilizando, na etapa de geração dos blocos ordenados, o método de ordenação interna *selection sort* apresentado na disciplina "Estrutura de Dados I".
2. Intercalação balanceada de vários caminhos (*2f* fitas) utilizando, na etapa de geração dos blocos ordenados, a técnica de seleção por substituição apresentada na disciplina "Estrutura de Dados II".
3. Quicksort externo.

Dividimos tanto a implementação quanto as etapas de análise, sendo cada membro responsável por um método. A princípio, desenvolvemos a modularização e o fluxo de execução de modo que cada membro pudesse desenvolver sua parte separadamente e no seu tempo. Em seguida, implementamos a conversão do arquivo (*PROVAO.TXT*) para binário e vice-versa, dada a maior facilidade de implementação dos métodos de ordenação externa, e funções de uso geral (verificação dos parâmetros de execução, impressão de registros, métodos de ordenação interna) e outras diretivas do enunciado do trabalho.

Após diversas tentativas de subdividir a *string* contendo os dados não numéricos, optamos por não o fazer, haja vista que nenhuma etapa da ordenação externa (que considera exclusivamente um tipo *float*) e o enunciado do trabalho não requerem essa divisão. Além disso, vários erros subjetivos, que variavam de acordo com o sistema operacional dos membros, contribuíram para essa decisão. Após diversos testes, asseguramos que essa implementação não comprometeu sequer um caractere e produziu apenas resultados exatos. A implementação da estrutura de dados *Registro* em questão é:

```
typedef struct registro
{
    long int inscricao;
    float nota;
    char estado_cidade_curso[87];
} Registro;
```

Durante a análise experimental dos métodos, utilizamos a função *clock* da biblioteca *time.h* para calcular o tempo de execução e variáveis *int* globais para somar as comparações e leituras realizadas. As etapas de conversão dos arquivos foram descartadas e consideramos apenas a execução dos métodos de ordenação. Os *outputs* utilizados para a construção dos gráficos estão localizados no diretório 'analise experimental' do repositório https://github.com/augustofgui/grupo-21.1, que atualmente se encontra privado até a conclusão da disciplina para impossibilitar plágio.

2. *Método de Ordenação Interna*

O método da Ordenação Interna da Intercalação Balanceada de Vários Caminhos se utiliza de um conjunto de blocos ordenados para que seja possível ordenar arquivos externos que não caberiam na memória principal.

Para isso, são definidas estruturas auxiliares para a ordenação externa. Essas estruturas são chamadas de Fitas. Essas estruturas também são memórias externas, mas nelas serão inseridas partes do arquivo original. Essas partes são chamadas de Blocos, que por sua vez são um conjunto de Registos escritos no arquivo original mas em uma quantidade gerível pela memória principal.

Se utilizando dessas estruturas, é possível ordenar um arquivo externo o dividindo em partes que serão organizadas pelo método de ordenação interno escolhido. Depois, esses blocos são escritos nas fitas auxiliares, e organizados de modo a criar um bloco maior, mas ainda sim em ordem. Esse processo de criação de blocos maiores, que serão inscritos em outras fitas, se repete até que forme um único bloco, sendo este equivalente em tamanho ao arquivo original, porém ordenado seguindo os critérios desejados.

Tendo este processo em mente, podemos analisar como este método foi implementado pelo nosso grupo.

2.1. Implementação

Para a implementação, foi utilizada a estrutura anteriormente citada de Registro. Com ela foi possível recuperar as informações do arquivo original de modo adequado. Como o intuito do trabalho era simular uma memória interna que suportaria apenas 20 (TAM_BLOCO) Registros em sua memória interna, assim foi estabelecido por um vetor de Registros.

```
Registro mem_interna[TAM_BLOCO];
```

Dessa maneira, foi possível criar os blocos, os salvando em um outra estrutura nomeada Fita. Na fita eram guardadas um ponteiro para o arquivo externo correspondente desta fita, um contador com a quantidade total de registros inseridos nesta fita, um booleano indicando se essa fita está ou não preenchida, ou seja, com algum registro escrito e uma variável para guardar a quantidade de registros escritos no último bloco, para que a quantidade de registros para garantir que as fitas sejam lidas da maneira correta, sem nenhuma leitura de lixo de memória.

```
typedef struct {
    FILE * arquivo;          // Ponteiro para arquivo.
    int quant_itens;         // Quantidades total de
itens.
    int tam_ultimo_bloco;    // O tamanho do ultimo bloco
inserido.
    bool preenchida;         // Booleano que diz se a
fita tem algo escrito nela.
} Fita;
```

Com isso foi criado também um estrutura chamada de Fitas, atenção ao S, de forma que nela fosse contida os vetores de fitas de entrada e saída. Foi optado por guardar estes vetores dentro da struct Fitas, para simplificar a passagem de parâmetros nas funções.

Com esta organização foi possível iniciar o processo de criação dos blocos ordenados para que a seja possível a ordenação.

2.2. Criação dos Blocos Ordenados

Para realizar a criação dos blocos ordenados foi criada uma função que recebe o mesmo nome (cria_blocos_ordenados). Nela os Registros são lidos, em blocos de 20 em 20 Registros, diminuindo a quantidade de leituras no arquivo original. A cada bloco de leituras, esses Registros são lidos e ordenados dentro do vetor de memória interna, usando o algoritmo de ordenação mergesort. Após estarem em um bloco ordenado, os Registros são inseridos nas fitas de entrada, que serão acessadas em sequência, da fita de número 1 até a 20, voltando para 1 depois.

Dessa forma, é garantido que cada fita recebe um bloco ordenado, até que todos os Registros dos arquivos sejam lidos. Foi calculado também a quantidade de blocos completos que o arquivo pode oferecer, para que, em caso de um arquivo que ofereça uma quantidade de registros não divisível por 20, os registros restantes sejam lidos, ordenados e escritos dentro da próxima fita da sequência.

Durante esse processo, as outras variáveis da Fita são alteradas de modo a descreverem a situação atual da Fita, em termos da quantidade de registros inseridos, se ela está ou não preenchida e qual o tamanho do último bloco, que pode ser incompleto.

2.3. Intercalação Balanceada de Vários Caminhos

Com os blocos devidamente ordenados e inseridos nas fitas, passamos para o processo de intercalar esse blocos, formando blocos maiores mas ainda sim ordenados, e escrevê-los nas fitas de saída.

Para isso, a função *intercalacao_ordenacao_interna* tenta recuperar o primeiro registro de cada fita preenchida, formando um vetor na memória principal. Então é buscado neste vetor qual o menor elemento, e este por sua vez é escrito na fita de saída. Estas fitas de saída também serão circuladas em ordem. Depois de um registro ser escrito, outro registro, da mesma fita de origem deste registro escrito, será lido e inserido na mesma posição do antigo registro escrito.

Então esse processo é repetido até que se atinja total de registros lidos ou até que o novo bloco atinja seu tamanho. Este tamanho depende do tamanho do bloco inicial, que como é sempre 20 no nosso programa, ele assume o tamanho de *20 x quantidade de fitas preenchidas*. Desse modo, é possível assegurar que cada bloco novo seja de tamanho correspondente à quantidade de blocos disponíveis nas fitas de entrada.

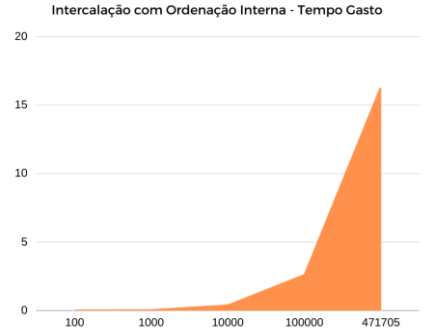
Depois de um bloco novo ser preenchido, é passada para próxima fita de saída, que por sua vez repetirá este processo até que, novamente, atinja o total de registros lidos ou o tamanho máximo do novo bloco.

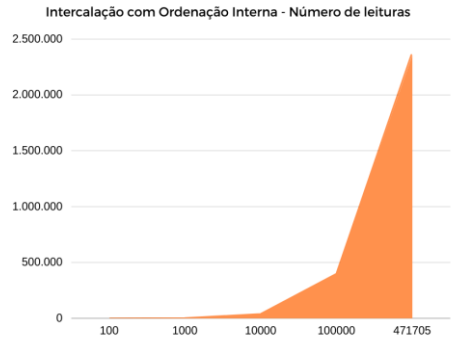
Assim ocorre a intercalação da primeira formação de blocos, mas caso os novos blocos não correspondam ao tamanho total do arquivo, ou seja, caso a intercalação não ordene todos os Registros do arquivo original pode repetir esse processo até que isso ocorra. Este caso acontece quando uma intercalação resulta em apenas uma fita de saída preenchida, correspondendo a todo o arquivo lido originalmente porém ordenado. Caso este não seja o caso, o processo de intercalação é repetido mas com os ponteiros do vetores das fitas invertidos. Portanto os vetores de fitas de entrada se tornam os de saída, e vice-versa.

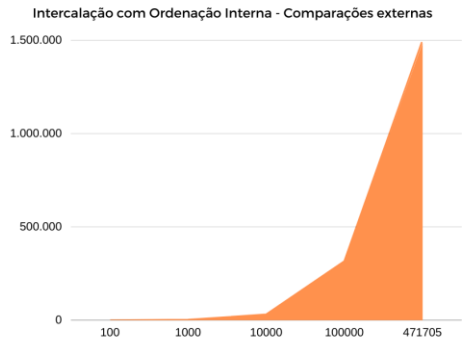
Após este processo é assegurado que apenas uma fita irá conter todo o arquivo original ordenado e esta fita será então processada e convertida de volta em um arquivo de texto, concluindo o processo de intercalação com ordenação interna.

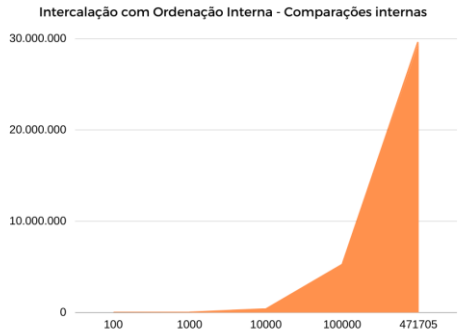
2.4. Análise

Este processo de ordenação externa oferece uma solução simples porém custosa, devido às várias leituras e escritas necessárias nas fitas. Ainda sim, ela se mostra capaz de ordenar arquivos complexos e aleatórios, não se diferenciando muito em seu pior e melhor caso, que são resolvidos usando os mesmos processos. A seguir os gráficos de análise de tempo gasto, número de comparações internas e externas, número total de leituras e escritas em arquivos externos:









Com estes dados é fácil perceber que o tamanho de registro influencia bastante no custo do processo, aumento principalmente o número de escritas e leituras em arquivos externos, uma vez que as fitas não suportam mais arquivos cada vez maiores em poucos blocos, sendo necessária inúmeras intercalações de blocos antes de atingir o arquivo completo em fitas externas.

3. Método de Seleção por Substituição

1. Utilização do *Heapsort* como método de ordenação interna

Um dos diferenciais desse método é o *Heapsort*. O *Heapsort* pega o vetor bloco de registros e os ordena em uma estrutura de árvore binária. A partir daí ele começa a ordenar os valores contidos no vetor. No nosso caso foi necessário usar o heap-mínimo, onde os nós-filhos possuem valores maiores que os nós-pais.

O *Heapsort* foi implementado com duas funções, a 'principal', que é chamada para começar a ordenação e a *peneira*, onde as comparações e as trocas entre um nó-pai e o nó filho são feitas.

O nosso *Heap* checa duas coisas durante a ordenação:

```
if((vet[j].nota < vet[j+1].nota || vet[j].f==1) &&
vet[j].f==0){
if((aux.nota < vet[j].nota || vet[j].f == 1) && aux.f == 0){
```

A primeira é o valor da nota do registro atual, que no primeiro caso é comparada com a próxima no vetor e no segundo caso é comparada com *aux* que possui um nó-pai, e a segunda coisa é a checagem da marca f, que checa se o item atual é o maior de todos (se *f* == 1) e, caso seja, para evitar a troca desse item.

2. Geração dos blocos ordenados por meio do *Heapsort*

Para o criação dos blocos, estabelecemos um registro, que representa a memória interna, para que nele fosse armazenado - através da leitura do arquivo - cada item contido no arquivo.

```
fread(mem_interna, sizeof(Registro), TAM_BLOCO, arquivo_binario);
    if (imprimir_dados)
        for (k = 0; k < TAM_BLOCO; k++)
            PrintFRead(&mem_interna[k]);
```

Para a marcação de cada item - quando necessário - acrescentamos ao registro um 'int f', sua dinâmica é basicamente: se 'f' igual a 0, o item não está marcado, se for igual a 1, ele está marcado. Sua marcação seguiu a mesma lógica, se o próximo item é menor que o que está saindo, então ele é marcado, assim, sendo tratado como maior que todos os itens do bloco corrente.

```
if (mem_interna[0].nota < anterior.nota) {
        num_reg_marcados++;
        mem_interna[0].f = 1;
}
```

Quando todos os itens presentes no bloco são marcados, o bloco corrente é encerrado, porém ainda é inserido um registro nulo, servindo como um marcador para o final deste bloco. Após isso um outro bloco é iniciado e isso continua até que todos os registros tenham sido lidos e consequentemente todos os blocos ordenados sejam criados.

```
while (num_total_reg_lidos < num_registros){
        num_reg_marcados = 0;
        num_atual_reg_escritos = 0;
        for (i = 0; i < TAM_BLOCO; i++)
            mem_interna[i].f = 0;
       while (num_reg_marcados < TAM_BLOCO && num_total_reg_lidos <
num_registros)
```

3. A fase de Intercalação dos blocos ordenados

Basicamente, substituímos o item do topo da fila de prioridades, escrevendo-o em uma fita de saída, pelo próximo item do mesmo bloco do item que está sendo substituído. E em seguida, a propriedade da fila foi reconstituída, o processo foi repetido até que não haja mais itens nos blocos ordenados.

A leitura do item no arquivo de origem e a escrita do item na fita de saída, respectivamente:

```
fread(&mem_interna[i], sizeof(Registro), 1, origem[i].arquivo);
fwrite(&mem_interna[pos_menor], sizeof(Registro), 1,
destino[j].arquivo);
```

A condição de parada verifica se ainda existe uma fita disponível e que todos os registros já tenham sido lidos, o loop segue até não possuir mais nenhuma. Uma fita é considerada disponível quando o bloco inserido nela atinge um registro nulo, constatando o fim deste bloco.

```
while (num_total_leituras < nro_registros) {
…
while (alguma_fita_disponivel(fitas_disponiveis,
NUM_FITAS_ENTRADA))
      …
}
```

A função que verifica se existe alguma fita disponível - caso encontre algum valor dentro do vetor, ‘true’ é retornado, caso contrário, retorna ‘false’.

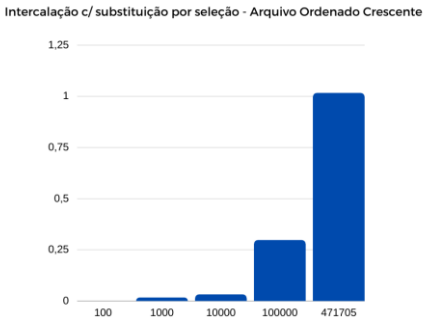
```
bool alguma_fita_disponivel(bool *vetor, int tam)
{
    int i;
    for (i = 0; i < tam; i++)
        if (vetor[i])
            return true;

    return false;
}
```

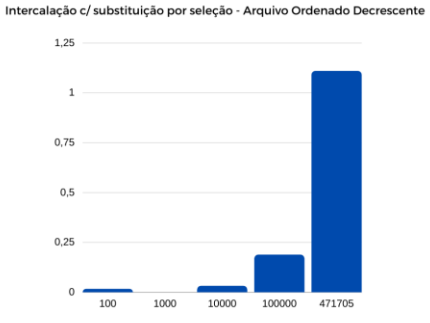
4. Análise experimental

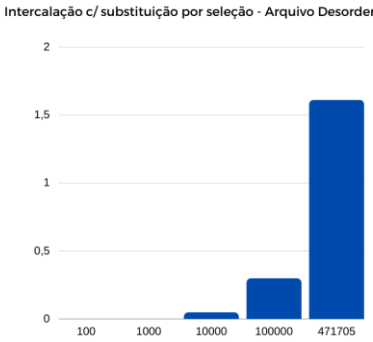
Durante o processo de análise, observamos que todas as grandezas medidas (comparações, leituras, escritas e tempo de execução) crescem proporcionalmente à quantidade de registros a serem ordenados para qualquer situação de ordenação do arquivo original, como esperado.

Durante os testes utilizando o arquivo ordenado crescente e o decrescente, foi possível notar que a diferença entre o tempo de execução dos dois é bem pequena. Porém o tempo de execução foi constantemente menor no crescente, já que não foi preciso ordenar os blocos e apenas uma intercalação foi necessária, enquanto o decrescente precisa de duas ou mais.

Considerando os testes com o arquivo desordenado, foi possível perceber que o seu tempo de execução é constantemente o dobro do arquivo ordenado crescente, o que torna uma diferença considerável. Isso acontece devido ao elevado número de comparações, que também dobra em relação ao arquivo ordenado crescente, por conta do arquivo estar fora de ordem.







Abaixo, gráficos contendo dados referentes ao tempo de execução e número de registros ordenados. Outros gráficos considerando as grandezas de comparações, leituras e escritas estão presentes no diretório ‘analise experimental’.

4. *Quicksort* Externo

1. Definição do *TipoArea*

Com a intenção de otimizar o método em geral, optamos por implementar uma estrutura de dados contendo o vetor de registros e um uma variável auxiliar para armazenar o número de índices ocupados. Dessa forma, para realizar a exclusão lógica quando e a ordenação do vetor, atribuímos um o valor de *INT_MAX* à nota do índice removido e ordenamos apenas a quantidade necessária de elementos do vetor. Em um vetor com 5 posições ocupadas, por exemplo, ao remover o primeiro registro, ordenamos apenas as primeiras 5 posições, economizando comparações entre as outras 15 do vetor.

```
typedef struct {
    Registro array[TAM_AREA];
    int nro_cels_ocupadas;
} TipoArea;
```

Para inicializar uma área vazia, atribuímos o mesmo valor de *INT_MAX* a todas as posições do vetor e zero à variável *nro_cels_ocupadas*. Ademais, a variável *NRArea*

presente no código de exemplo disponibilizado foi dispensada, haja vista que *nro_cels_ocupadas* cumpre exatamente a mesma função.

2. Inicialização dos apontadores de escrita e leitura e partição

Tal como no código de exemplo, utilizamos duas variáveis para a leitura e escrita inferiores e apenas uma para as superiores. Ambas são inicializadas sobre o arquivo binário convertido a partir do *.txt* com as permissões de abertura para leitura e escrita *r+b*.

As etapas de partição também seguem o código de exemplo disponibilizado, diferenciando apenas nos tipos de algumas variáveis como *Linf* e *Lsup* que, como as notas são do tipo float, não poderiam ser do tipo *int*; *OndeLer*, que teve o tipo alterado para booleano; e *NRArea*, que foi dispensada.

3. Funções *InsereItem, RetiraUltimo e RetiraPrimeiro*

Implementadas de acordo com a estrutura de dados *TipoArea*, realizam a inserção e remoção dos registros de forma ordenada de acordo com a quantidade de índices ocupados.

```
void InsereItem(Registro *UltLido, TipoArea *Area)
{
    Area->array[Area->nro_cels_ocupadas] = *UltLido;
    Area->nro_cels_ocupadas++;
    selection_sort_ascendente(Area->array, Area->nro_cels_ocupadas);
}
```

```
void RetiraUltimo(TipoArea *Area, Registro *R)
{
    *R = Area->array[Area->nro_cels_ocupadas - 1];
    Area->array[Area->nro_cels_ocupadas - 1].nota = INT_MAX;
    Area->nro_cels_ocupadas--;
}
```

```
void RetiraPrimeiro(TipoArea *Area, Registro *R)
{
    *R = Area->array[0];
    Area->array[0].nota = INT_MAX;
    selection_sort_ascendente(Area->array, Area->nro_cels_ocupadas);
    Area->nro_cels_ocupadas--;
}
```

4. Utilização do *insertion sort* como método de ordenação interna

Em um primeiro momento, implementamos o método de ordenação *merge sort*, dada sua complexidade logarítmica. Todavia, após testes com a totalidade do arquivo de 471.705 registros, observamos uma diferença de aproximadamente 8 segundos em favor do *selection sort* que, apesar da complexidade quadrática, se provou mais eficiente.

Acreditamos que esse fato esteja relacionado à pequena dimensão do vetor de registros, que possui apenas 20 posições. Como o método de ordenação interna é referenciado várias vezes durante a execução do código, o *merge sort*, por precisar alocar e desalocar memória repetidas vezes, teve sua eficiência um pouco comprometida.

Ainda na busca pelo método de ordenação interna ideal, implementamos o método *insertion sort* dada sua eficiência para vetores quase ordenados, haja vista que a inserção

e remoção efetua apenas a transferência de 1 elemento para dentro ou fora do vetor. Dentre os 3 testados, o *insertion sort* se provou mais eficiente.

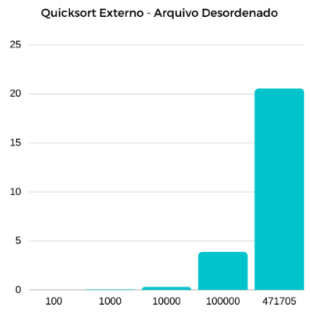
5. Análise experimental

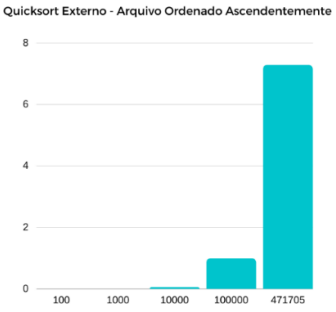
Durante o processo de análise, observamos que todas as grandezas medidas (comparações, leituras, escritas e tempo de execução) crescem proporcionalmente à quantidade de registros a serem ordenados para qualquer situação de ordenação do arquivo original, como esperado.

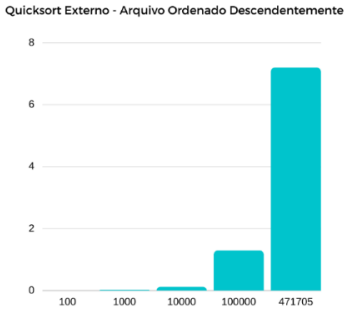
Em relação às situações de ordenação, observamos que o arquivo desordenado realiza significativamente mais comparações, leituras e escritas, além de demorar mais de 3 vezes o tempo de execução para arquivos ordenados ascendentemente. Isso se deve ao fato da maior ocorrência de leitura de registros que não estejam entre os limites de *Linf* e *Lsup*, o que coincide com a menor quantidade de comparações realizadas pelo método de ordenação interna, e faz com que esses registros sejam imediatamente gravados nos subarquivos para serem referenciados novamente em uma chamada recursiva mais adiante. Já entre os arquivos previamente ordenados, observamos uma pequena diferença em favor do arquivo ordenado ascendentemente, como esperado do melhor caso do algoritmo.

O pior caso (partições geradas de tamanhos maior e menor possível) não foi identificado, visto que a grande quantidade de registros tende essa probabilidade de ocorrência a zero.

Abaixo, gráficos contendo dados referentes ao tempo de execução e número de registros ordenados. Outros gráficos considerando as grandezas de comparações, leituras e escritas estão presentes no diretório ‘analise experimental’.







5. Conclusão

Após comparar os resultados obtidos nos diferentes ambientes à disposição, optamos por extrair as estatísticas a partir dos testes realizados no ambiente onde os tempos de execução foram menores. O ambiente em questão possui as seguintes especificações:

Processador: *Intel Core i5 8265U (1,6 GHz até 3,9 GHz) 6 MB Cache*
Memória: *8 GB de RAM DDR4 2400 MHz*
Dispositivo de memória secundária: *HD de 1 TB 5400 RPM*

Dentre todos os algoritmos implementados, elegemos o método de intercalação com substituição por seleção como o mais eficiente para ordenar o arquivo no contexto do trabalho prático, chegando a atingir a marca de 1.609375 segundos para ordenar todos os 471.705 registros do arquivo desordenado. O método de intercalação com ordenação interna se mostrou ligeiramente menos eficiente em comparação à substituição por seleção, demorando poucas frações de segundo a mais durante a execução.

O método de *quicksort* externo se mostrou o menos eficiente dentre os 3 implementados, demorando 20.515625 segundos para ordenar todo o arquivo desordenado. Todavia, vale ressaltar que, diferentemente dos métodos que utilizam a intercalação, o método de *quicksort* ordena *in situ*, o que pode ser uma característica decisiva a depender da circunstância na qual a ordenação externa é aplicada e que requer um processamento significativamente maior sobre o arquivo a ser ordenado.

6. Considerações Finais

Durante os testes realizados com o *quicksort* externo, identificamos uma anomalia específica de um ambiente de testes. No caso, ao realizar a ordenação de apenas 100 registros, algumas frações do arquivo permaneciam desordenadas ao final da execução. Para outras quantidades de registros, o resultado obtido era o esperado (ordenado) e, após muito esforço para identificar a causa, concluímos que ela é particularidade do ambiente em questão e não está localizada na implementação. Ademais, como mencionado, a anomalia é específica de apenas um ambiente e não foi observada noutros 4 nos quais o mesmo código foi testado.