**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**

* **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.

* ***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.

1. 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.

1. 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**);**

**}**

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

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**--;**

**}**

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**--;**

**}**

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.

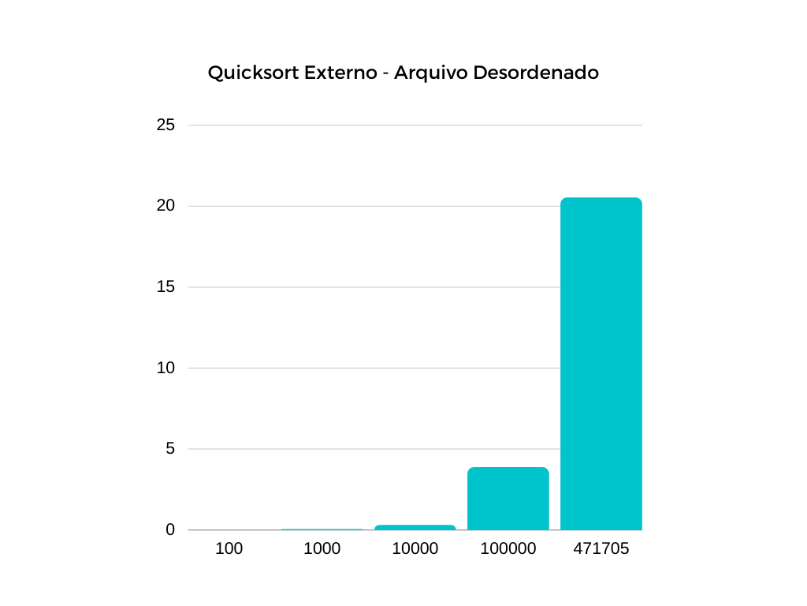
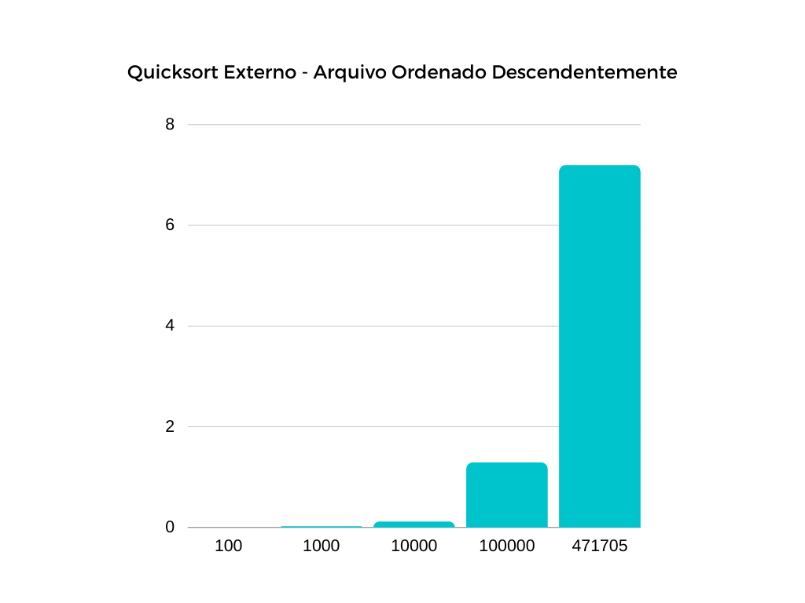
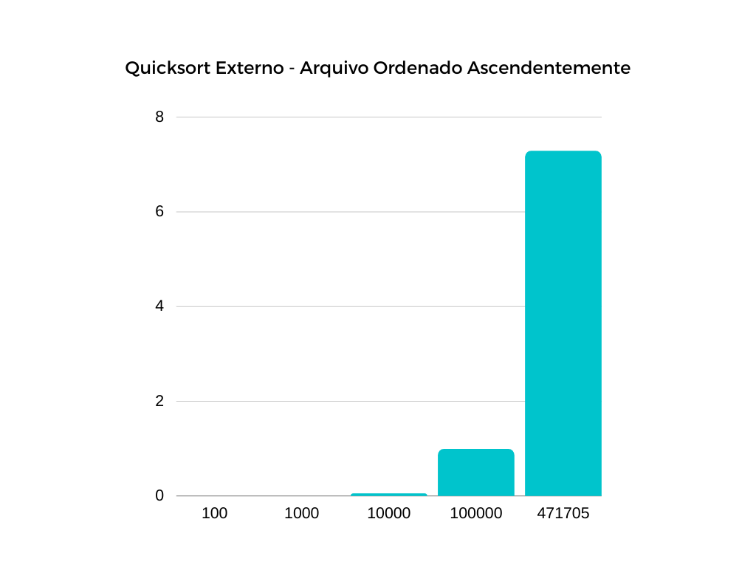
1. Análise experimental

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. Já os arquivos ordenados descendentemente necessitaram de aproximadamente 2x mais operações e tempo durante a execução em relação aos ordenados ascendentemente.

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’.



* **Conclusão**