UNIVERSIDADE FEDERAL DE OURO PRETO
DECOM

Trabalho Prático 2 - Ordenação Externa

Arthur Henrique Santos Celestino 21.1.4019
Caio Silas de Araujo Amaro 21.1.4111
Henrique Dantas Pighini 21.1.4025

Ouro Preto - MG
2022

1. Introdução:

Existem várias maneiras de ordenar um arquivo externo, todas elas com suas características e propriedades que tornam possível ler um arquivo e retorná-lo composto. Os métodos que nos foi apresentado são: Intercalação balanceada de vários caminhos, Intercalação balanceada de vários caminhos com criação de bloco por substituição e o QuickSort Externo.

Após uma breve revisão dos mesmos reunimos nosso grupo para realizar uma discussão dos benefícios e malefícios de cada e como nos foi introduzido no trabalho de implementação estamos realizando nosso registro dos resultados obtidos.

Assim, para verificar a eficiência dos métodos que aprendemos em sala, implementamos as funções na linguagem C e vamos apresentar este relatório detalhado.

A criação desses arquivos contou com a utilização das seguintes TADs e valores:

```c
//Definindo valor máximo de ORDENAÇÃO
#define MAXIMO 471705

//Definindo valor de FITAS
#define f 20
```

```c
typedef struct {
    long int numInscr;
    double nota;
    char estado[3];
    char cidade[51];
    char curso[31];

} Reg;
```

```c
typedef struct {
    int num_transE;
    int num_transL;
    int num_comp;
    double temp_exec;
    int valido;

} Comp;
```

```c
typedef struct{
```

```
    FILE* fitas[2*f];
    int tamFita[2*f];
} Fitas;
```

```
//Struct que simplifica as operações com fitas
typedef struct{
    Reg reg;
    int flag;
    int fita;
} ItemArea;
```

```
//Struct que simplifica as operações na área
typedef struct{
    ItemArea iArea[20];
    int tam;
    double ultimo;
} Area;
```

Além disso, como proposto pelo professor e doutor Guilherme Tavares de Assis, o programa foi implementado de maneira que sua execução ocorre através da seguinte linha de comando.

ordena <método> <quantidade> <situação> [-P]

- <método> representa o método de pesquisa externa a ser executado, podendo ser um número inteiro de 1 a 3, de acordo com a ordem dos métodos mencionados;
- <quantidade> representa a quantidade de registros do arquivo considerado;
- <situação> representa a situação de ordem do arquivo, podendo ser 1 (arquivo ordenado ascendentemente), 2 (arquivo ordenado descendentemente) ou 3 (arquivo desordenado aleatoriamente);
- [-P] representa um argumento opcional que deve ser colocado quando se deseja que as chaves de pesquisa dos registros do arquivo considerado sejam apresentadas na tela.

Com o objetivo de analisar os algoritmos, consideramos os seguintes quesitos:

- Número de transferências (leitura) de registros da memória externa para a memória interna.

- Número de transferências (escrita) de registros da memória interna para a memória externa.
- Número de comparações entre valores do campo de ordenação dos registros.
- Tempo de execução (tempo do término de execução menos o tempo do início de execução).

Para compilar nosso código, usamos as seguintes diretivas de compilação (usando o terminal do linux, pois alguns comandos e execução do código apresentam leves erros no Windows):

$ gcc main.c quick.c ord_ext.c gerador_arquivo.c interNorm.c -o programa -Wall

$ ./programa <metodo> <quantidade> <situação>

Para geração dos arquivos, usamos um programa gerador criador por nós mesmos, que se encontra presente na pasta do trabalho

2. Análise Dos Quesitos:

2.1 Intercalação Balanceada de Vários Caminhos Normal

Nesse método de ordenação externa, temos gerados blocos ordenados, que são colocados em F fitas. Diferente da geração por substituição (que será explicada mais abaixo no trabalho), nesse método ordenamos os blocos de tamanho fixo, utilizando um método de ordenação. Após os blocos gerados, os itens precisam ser intercalados (comparando os menores elementos de cada bloco, e pegando o menor deles, que é substituído pelo seguinte da fita que ele veio), até sobrar apenas um bloco grande, que é meu arquivo ordenado.

Começamos nosso processo com essa função abaixo:

```c
void intercalacaoBase(FILE* ArqExterno, int qnt, Comp* resultado){
    clock_t tempo = clock(); //inicio do clock usado para contrar o tempo

    Fitas fitas;
    ItemArea aux;

    criaFitas(&fitas, 0); //criando previamente todas as fitas que serao usados
    blocoAleatorio(ArqExterno, &fitas, qnt, resultado); //Gera os blocos em todas as fitas

    double tempoCria = ((double)(clock() - tempo))/CLOCKS_PER_SEC;
    resultado[0].temp_exec = tempoCria; //colocando o tempo de execução da criacao

    tempo = clock();
```

Primeiramente são criadas todas as fitas através da função criaFitas, do 0 ao 39 totalizando 40 fitas no caso. Nessa função utilizamos um segundo parâmetro pois ele é usado para criar quantias especificas de fitas em certos momentos do código:

```c
void criaFitas(Fitas* fitas, int t){
    if(t == 0){
        for(int i = 0; i < 2*f; i++){
            char arq[10];
            snprintf(arq, 10, "f_%d.bin", i);
            fitas->fitas[i] = fopen(arq, "w+b");
            fitas->tamFita[i] = 0;
        }
    }

    else if (t == 1){
        t = f;
        for(int i = t-f; i < t; i++){
            char arq[10];
            snprintf(arq, 10, "f_%d.bin", i);
            fitas->fitas[i] = fopen(arq, "w+b");
            fitas->tamFita[i] = 0;
        }
    }

    else{
        t = 2*f;
        for(int i = t-f; i < t; i++){
            char arq[10];
            snprintf(arq, 10, "f_%d.bin", i);
            fitas->fitas[i] = fopen(arq, "w+b");
            fitas->tamFita[i] = 0;
        }
    }
}
```

Logo após, é chamada a função que cria meus blocos. Dentro dessa função, vou lendo um por um os itens e inserindo na área de memória interna. A função de inserção ja insere ordenado o item, então ela ja tem um método de ordenação implícito nela. Essa função será

explicada mais afrente do trabalho. Quando a memória enche, um for retira eles e vai colocando na fita correspondente. Ao fim, é colocado um registro com nota -1, que é usado como controle para sinalizar o fim dos nossos blocos.

```
while(fread(&aux.reg, sizeof(Reg), 1, ArqExterno)){ //Lendo um por um os registros do arquivo original
    resultado[0].num_transL++;
    resultado[0].num_comp += 3;
    insereArea(&area, aux); //Inserindo cada item na area de memoria interna
    if(area.tam == f){

        for(int i = 0; i < f; i++){
            resultado[0].num_comp += 5;
            aux = retiraArea(&area); //Depois que a area esta cheia, indo retirando um por um e colocando na fita
            fwrite(&(aux.reg), sizeof(Reg), 1, fitas->fitas[fitaEscrita]);
            resultado[0].num_transE++;
        }

        aux.reg.nota = -1; //Colocando um -1 que serve como separador entre nossos blocos
        fwrite(&(aux.reg), sizeof(Reg), 1, fitas->fitas[fitaEscrita]);
        resultado[0].num_transE++;
        fitas->tamFita[fitaEscrita]++;
        fitaEscrita++;
        fitaEscrita %= f;
    }
}
```

Depois que temos os blocos gerados, chegamos no while que irá intercalar os itens em si

```
while(1){
    intercalacaoES(&fitas, qnt, resultado); //chama a funcao de intercalacao das fitas 0 a 19, para as fitas 20 a 39
    if(eFim(fitas, 1)){ //Se nao tem mais nada em nenhuma fita, a nao ser a 0 ou 20, libero as fitas antigas
        liberaFitas(&fitas, 1);
        criaFitas(&fitas, 1);
        resultado[1].valido = 0;
        break;
    }
    liberaFitas(&fitas, 1);
    criaFitas(&fitas, 1);
    intercalacaoSE(&fitas, qnt, resultado); //Processo semelhante ao citado acima, porem de 20 a 39 para 0 a 19
    if(eFim(fitas, 0)){
        liberaFitas(&fitas, -1);
        criaFitas(&fitas, -1);
        resultado[1].valido = 0;
        break;
    }
    liberaFitas(&fitas, -1);
    criaFitas(&fitas, -1);
}
```

Nesse while são chamadas nossas funções que realizam as intercalações em si. Separamos em duas funções distintas: uma que intercala da 0 a 19 para as 20 a 39, e outra que faz o contrário. Entre cada passo apagamos o conteúdo das fitas usadas anteriormente, para prevenir lixo nas fitas e simular um caso ideal, que seria retirar os itens em vez de copia-los apenas.

```
while(temBloco(fitas, blocosLidos, f, 2 * f)){
    area.ultimo = INT_MIN;
    k++;
    for(int i = f; i < 2 * f; i++){
        resultado[1].num_transL++;
        fread(&aux.reg, sizeof(Reg), 1, fitas->fitas[i]);
        if(feof(fitas->fitas[i])){
            break;
        }
        aux.flag = 0;
        aux.fita = i;
        // printf("Matricula %ld\n", aux.reg.numInscr);
        resultado[1].num_comp += 3;
        insereArea(&area, aux);
    }

    for(int i = 0; i < f; i++){
        acabou[i] = 0;
    }
```

Na função de intercalação em si, como mostrado acima, enquanto existem blocos nas fitas sendo intercalados, vai lendo-se um item de cada fita e colocando na memória interna ordenando, de modo que serão retirados e escritos na fita de saída ja em ordem. Mais detalhes dessas funções e as auxiliares usadas serão dadas mais abaixo, pois usamos as mesmas funções de intercalação em ambos os métodos.

2.2 Intercalação Balanceada com criação de blocos por substituição

A intercalação balanceada com criação de blocos por substituição segue a mesma maneira de ordenar o arquivo do método semelhante mostrado acima, o que faz ela se diferenciar é a maneira que os blocos são criados.

Como visto na Intercalação Balanceada do 1° método deste trabalho prático, para ordenar um arquivo externo foi utilizado (2 * F) fitas para realizar uma intercalação destas fitas com blocos ordenados inseridos nelas representando partes do arquivo que se está ordenando, porém, é possível reduzir a quantidade de intercalações necessárias para ordenar o texto desejado se os blocos forem construídos de maneira mais óptima.

Assim, faremos uma construção de bloco de modo que o tamanho mínimo deste seja F e não exista um tamanho máximo, fazendo com que ele seja melhor do que a Intercalação com a construção aleatória que possui um tamanho máximo de bloco de F.

```
void blocoSelecao(FILE* ArqExterno, Fitas* fitas, int qnt, Comp*
resultado){
    int fitaEscrita = 0;
    Area area;
    ItemArea aux;
    area.tam = 0;
    area.ultimo = INT_MIN;

    for(int i = 0; i < 20; i++){

        resultado[0].num_transL++;
        fread(&aux.reg, sizeof(Reg), 1, ArqExterno);
        aux.flag = 0;
        if(feof(ArqExterno)){
            break;
        }

        resultado[0].num_comp += 3;
        insereArea(&area, aux);
    }
```

Começo preenchendo uma área com os primeiros 20 elementos de cada fita e garantindo que não tenha chegado no fim de nenhuma fita e se chegar saímos do laço de repetição já que chegamos ao fim dos blocos.

```
    resultado[0].num_comp += 5;
    aux = retiraArea(&area);

    resultado[0].num_transE++;
    fwrite(&(aux.reg), sizeof(Reg), 1, fitas->fitas[fitaEscrita]);
```

Após preencher o vetor e inserir os elementos na área vamos retirar o primeiro que será o menor da lista graças às funções “insereArea” e “retiraArea” que foram implementadas:

```
void insereArea(Area* area, ItemArea item){
    if(item.reg.nota < area->ultimo){
        item.flag = 1;
    }
    area->iArea[area->tam] = item;
    area->tam++;
```

```
    Heap_RemakeEnd(area->iArea, 0, area->tam - 1);
}
```

A função “insereArea”, como o nome já diz, insere um elemento no fim da área e chama um “Heap_RemakeEnd” para colocar o menor elemento do Heap na raiz. Entretanto ela também vai marcar os elementos inseridos que forem menores que os retirados.

```
ItemArea retiraArea(Area* area){
    ItemArea aux;

    if(area->tam == 0){
        aux.reg.nota = -1;
        aux.fita = -1;
        printf("Nao existe elemento na Area\n");
        return aux;
    }

    aux = area->iArea[0];

    Troca(area->iArea, 0, area->tam-1);
    area->tam--;
    area->ultimo = aux.reg.nota;

    Heap_Remake(area->iArea, 0, area->tam - 1);
    return aux;
}
```

Já a “retiraArea” vai retirar o elemento encontrado na raiz e também vai chamar um “Heap_Remake” para manter o Heap com sua propriedade regularizada.

Continuando, após retirar a raiz da área e inseri-la na primeira fita, vamos começar um laço de repetição que irá continuar lendo os elementos do arquivo até que a quantidade que você deseja seja alcançada, e cada elemento lido será incluído na área.

```
      for(int i = 20; i < qnt; i++){
              resultado[0].num_transL++;
              fread(&aux.reg, sizeof(Reg), 1, ArqExterno);
              aux.flag = 0;

              resultado[0].num_comp += 3;
              insereArea(&area, aux);
```

Porém, para manter a ideia de um bloco criado por seleção por substituição é necessário garantir que os valores lidos que forem menores que os últimos retirados sejam colocados em um bloco diferente do atual, assim, criamos uma função que garante que nem todos os elementos da área foram marcados.

```
int todasCheias(Area area){
    for(int i = 0; i < area.tam; i++){
        if(area.iArea[i].flag == 0){
            return 0;
        }
    }
    return 1;
}
```

Mas, se for necessário trocar o bloco ele vai zerar as marcações nos elementos, inserir um coringa com nota -01.0 e vai alterar a fita que será inserida.

```
if(todasCheias(area)){
                for(int j = 0; j < 20; j++){
                    area.iArea[j].flag = 0;
                }
                //Vamos mudar de fita então para delimitar o tamanho
insiro um elemento com -1
                aux.reg.nota = -1;
                fitas->tamFita[fitaEscrita]++;
                resultado[0].num_transE++;
                fwrite(&(aux.reg),      sizeof(Reg),      1,      fitas-
>fitas[fitaEscrita]);

                fitaEscrita++;
                fitaEscrita %= f;
            }
```

Se não for necessário trocar a fita vamos continuar inserindo na fita até que o valor inserido pelo usuário seja atingido:

```
resultado[0].num_comp += 5;
            aux = retiraArea(&area);
            resultado[0].num_transE++;
            fwrite(&(aux.reg),       sizeof(Reg),       1,       fitas-
>fitas[fitaEscrita]);
```

Agora nós temos que lidar com o problema de existir blocos que são menores que F, já que estes vão sobrar quando for feita a inserção de F elementos no começo.

Desta forma, fizemos um while que vai lendo os objetos da área até que o tamanho dela seja 0 e se for necessário também vai trocar de bloco (lembrando que se for necessário o bloco será trocado quando os elementos estiverem todos marcados).

Após essa verificação colocamos um coringa com nota -01.0 para sabermos que foi finalizado com sucesso o último bloco

```
      int trocouFita = 0;
          Reg aux2;

          while(area.tam > 0){
              resultado[0].num_comp += 5;
              aux = retiraArea(&area);
              if (aux.flag == 1 && !trocouFita) {

                  aux2 = aux.reg;
                  aux2.nota = -1;
                  fitas->tamFita[fitaEscrita]++;
                  resultado[0].num_transE++;
                  fwrite(&(aux2),       sizeof(Reg),       1,       fitas-
>fitas[fitaEscrita]);

                  fitaEscrita++;
                  trocouFita = 1;
              }
              resultado[0].num_transE++;
              fwrite(&(aux.reg),       sizeof(Reg),       1,       fitas-
>fitas[fitaEscrita]);
          }
          aux.reg.nota = -1;
          fitas->tamFita[fitaEscrita]++;
          resultado[0].num_transE++;
          fwrite(&(aux.reg), sizeof(Reg), 1, fitas->fitas[fitaEscrita]);
```

Com estes processos foi construído F fitas de entrada para serem intercaladas, agora vamos prosseguir com a Intercalação Balanceada com criação de blocos por substituição.

```
    void intercalacaoSelecao(FILE* ArqExterno, int qnt, Comp*
resultado){
        clock_t tempo = clock();

        Fitas fitas;
        ItemArea aux;

        criaFitas(&fitas, 0);
        blocoSelecao(ArqExterno, &fitas, qnt, resultado);
```

Criamos nossas fitas e preenchemos elas.

```
    while(1){
            intercalacaoES(&fitas, qnt, resultado);
            if(eFim(fitas, 1)){
                liberaFitas(&fitas, 1);
                criaFitas(&fitas, 1);
                resultado->valido = 0;
                break;
            }
            liberaFitas(&fitas, 1);
            criaFitas(&fitas, 1);

            intercalacaoSE(&fitas, qnt, resultado);
            if(eFim(fitas, 0)){
                liberaFitas(&fitas, -1);
                criaFitas(&fitas, -1);
                resultado->valido = 0;
                break;
            }
            liberaFitas(&fitas, -1);
            criaFitas(&fitas, -1);
        }
```

No código acima nós construímos um laço de repetição infinito que vai ser interrompido quando a condição do “eFim” for falsa, é justamente essa condição que vai realizar nossa intercalação:

```
    int eFim(Fitas fitas, int t){
        if(t == 1){
            t = f;
        }
```

```
        for(int i = t+1; i < f + t; i++){
            if(fitas.tamFita[i] !=0){
                return 0;
            }
        }

        return 1;
    }
```

É bem simples, mas dependendo da sua entrada na função ela vai garantir que apenas a fita 0 ou a fita 20 tem blocos e que o número desses blocos é 1. Já que com isso podemos validar que a intercalação acabou.

```
    intercalacaoES(&fitas, qnt, resultado);
            if(eFim(fitas, 1)){
                liberaFitas(&fitas, 1);
                criaFitas(&fitas, 1);
                resultado->valido = 0;
                break;
            }
            liberaFitas(&fitas, 1);
            criaFitas(&fitas, 1);
```

A função “intercalacaoES” vai pegar as fitas de entrada e vão intercalar os blocos das mesmas nas fitas de saída e após realizar a operação irá liberar as fitas de entrada e criar elas novamente para simular uma função que esvazia as mesmas.

```
            intercalacaoSE(&fitas, qnt, resultado);
            if(eFim(fitas, 0)){
                liberaFitas(&fitas, -1);
                criaFitas(&fitas, -1);
                resultado->valido = 0;
                break;
            }
            liberaFitas(&fitas, -1);
            criaFitas(&fitas, -1);
```

E para ter uma repetição de intercalação criamos a mesma função só que ela tira das fitas de saída e intercala nas de entrada.

Com a intenção de melhorar o entendimento do código vou passar passo a passo no código de intercalação.

```
void intercalacaoES(Fitas* fitas, int qnt, Comp* resultado){

    int fitaEscrita = f;
    int acabou[f]; // acabou bloco
    int blocosLidos[f];

    Area area;
    area.tam = 0;
    area.ultimo = INT_MIN;

    ItemArea aux;
    // printf("aqui \n");
    for(int i = 0; i < 2*f; i++){
        fseek(fitas->fitas[i], 0, SEEK_SET);
    }
```

Criamos as variáveis que vão nos auxiliar e rejeitamos os ponteiros das fitas para o começo.

```
while(temBloco(fitas, blocosLidos, 0, f)){
        area.ultimo = INT_MIN;
        k++;
        for(int i = 0; i < 20; i++){
            resultado[1].num_transL++;
            fread(&aux.reg, sizeof(Reg), 1, fitas->fitas[i]);
            if(feof(fitas->fitas[i])){
                break;
            }
            aux.flag = 0;
            aux.fita = i;

            resultado[1].num_comp += 3;
            insereArea(&area, aux);
        }
```

Enquanto existem blocos a serem intercalados você primeiro lê os 20 primeiros elementos de cada fita da mesma maneira que é feita na criação de blocos.

```
      while(area.tam != 0){
                  resultado[1].num_comp += 5;
                  aux = retiraArea(&area);

                  resultado[1].num_transE++;
                  fwrite(&(aux.reg),      sizeof(Reg),      1,      fitas-
>fitas[fitaEscrita]);

                  if(!acabou[aux.fita] && blocosLidos[aux.fita] < fitas-
>tamFita[aux.fita]){
                      resultado[1].num_transL++;
                      fread(&aux.reg,      sizeof(Reg),      1,      fitas-
>fitas[aux.fita]);

                      if(aux.reg.nota < 0){
                          acabou[aux.fita] = 1;
                          blocosLidos[aux.fita]++;
                      }
                      else{
                          resultado[1].num_comp += 3;
                          insereArea(&area, aux);
                      }
                  }
```

E de maneira similar a criação de blocos, enquanto a área tiver um tamanho diferente de 0 e vamos retirando os elementos da área e escrevo eles na fita de saída atual inserindo o próximo registro na área logo após.

```
      if(todasCheias(area)){
                      // printf("Flag: ");
                      for(int j = 0; j < 20; j++){
                          // printf("%d ", area.iArea[j].flag);
                          area.iArea[j].flag = 0;
                      }
                      // printf("\n");
                      aux.reg.nota = -1;
                      fitas->tamFita[fitaEscrita]++;
                      resultado[1].num_transE++;
                      fwrite(&(aux.reg),    sizeof(Reg),    1,    fitas-
>fitas[fitaEscrita]);
```

```
                    //Transformando a fita escrita
                    // printf("fitaEscrita: %d\n", fitaEscrita);
                    fitaEscrita++;
                    fitaEscrita %= f;
                    fitaEscrita += f;
                }
```

Dentro desse laço da área fizemos um if para garantir que não seria necessário trocar de bloco e se for necessário trocamos de maneira análoga com a criação de blocos.

Resumindo então, para realizar a Intercalação Balanceada com criação de blocos por substituição devemos criar os blocos de maneira Óptima e realizar a intercalação da maneira que já nos foi apresentada.

2.3 QuickSort Externo

Assim como o quicksort interno, o quicksort externo também utiliza o paradigma de divisão e conquista, o algoritmo ordena de forma in situ um arquivo A = {A1,.....,An} de n registros. colocando os valores menores do que ele de um lado e os maiores de outro, os dividindo em dois novos subvetores e repetindo o processo até que se chegue num vetor com apenas um elemento.

```
if (i - Esq < Dir - j){
    //ordena o subarquivo menor
    quickSorteExterno(ArqLi,ArqEI,ArqLEs,Esq,i,resultado);
    quickSorteExterno(ArqLi,ArqEI,ArqLEs,j,Dir,resultado);
}
else {
    quickSorteExterno(ArqLi,ArqEI,ArqLEs,j,Dir,resultado);
    quickSorteExterno(ArqLi,ArqEI,ArqLEs,Esq,i,resultado);
}
```

Para a partição do arquivo, é utilizado uma área de memória interna para o armazenamento do pivô, onde o tamanho da área é igual a j - i - 1, sendo necessário >= 3.

Nas chamadas recursivas, deve ser ordenada primeiramente os subarquivos de menor tamanho, os subarquivos vazios ou com um único registro são ignorados e no caso do arquivo de entrada for no máximo de (j - 1 - i) registro, ele é ordenado em uma única passada.

Diferente do que ocorre no quicksort interno, caminha-se pelo arquivo utilizando apontadores. Li é um apontador que estará no início do arquivo, servirá para o controle da leitura no limite inferior (ler um valor e passar a apontar para o próximo). Ls é um apontador

que estará no fim do arquivo, que partirá do último valor e irá retroceder no arquivo. Quando Li ultrapassa o valor de Ls, a partição é concluída.

```
void Particao (FILE **ArqLi,FILE **ArqEI,FILE **ArqLEs,TipoArea Area,int Esq, int Dir,int *i,int *j,Comp *resul
    int Ls = Dir,Es = Dir, Li = Esq, Ei = Esq, NRArea = 0;
    double Linf = INT_MIN, Lsup = INT_MAX;
    bool OndeLer = true;
    Reg UltLido, R;

    fseek(*ArqLi, (Li - 1) * sizeof(Reg),SEEK_SET);
    fseek(*ArqEI, (Ei - 1) * sizeof(Reg),SEEK_SET);
    *i = Esq - 1;
    *j = Dir + 1;

    while(Ls >= Li){
        // printf("Ls = %d >= %d = Li\n", Ls, Li);
        if (Area.quant < TAMAREA - 1){
            if (OndeLer){
                LeSup(ArqLEs,&UltLido,&Ls,&OndeLer,resultado);
            }
            else {
                Leinf(ArqLi,&UltLido,&Li,&OndeLer,resultado);
            }
            // printf ("notas %lf\n",UltLido.nota);
            InserirArea(&Area,&UltLido,&NRArea,resultado);
            continue;
        }
```

No decorrer da execução do quicksort externo, os pivôs vão sendo constituídos e armazenados na memória interna. Para saber quais elementos podem estar dentro dos pivôs, utiliza-se as variáveis Linf e Lsup, que são, respectivamente, o limite inferior e o limite superior do pivô.

```
resultado[1].num_comp++;
if (UltLido.nota > Lsup){
    *j = Es;
    EscreveMax(ArqLEs,UltLido,&Es,resultado);
    continue;
}
resultado[1].num_comp++;
if (UltLido.nota < Linf){
    *i = Ei;
    EscreveMin(ArqEI,UltLido,&Ei,resultado);
    continue;
}
// printf ("notas %lf\n",UltLido.nota);
InserirArea(&Area,&UltLido,&NRArea,resultado);
```

Linf terá inicialmente um valor de infinito negativo, mas no decorrer da execução armazenará valores menores do que o valor do pivô. Já Lsup terá inicialmente um valor infinito positivo, mas no decorrer da execução armazenará valores maiores do que o valor do pivô.

Para a ordenação da área antes que ela seja inserida no arquivo utilizamos o método selection sort, que é chamado toda vez que um item é inserido na área.

```
void InserirArea(TipoArea *Area, Reg *UltLido, int *NRArea,Comp *resultado){
    //insere o ultimo lido de forma ordenada na area
    IsereItem(UltLido,Area);
    *NRArea = ObeterNumCelOcupadas(Area);
    selectionsort(Area,resultado);
}
```

```
void selectionsort(TipoArea *Area,Comp *resultado){
    int menor = 0;
    for(int i = 0; i < Area->quant;i++){
        menor = i;
        for (int j = i + 1; j < Area->quant;j++){
            resultado[1].num_comp++;
            if(Area->vector[menor].nota > Area->vector[j].nota){
                menor = j;
            }
        }
        if (menor != i){
            Reg aux;
            aux = Area->vector[menor];
            Area->vector[menor] = Area->vector[i];
            Area->vector[i] = aux;
        }
    }
}
```

Sua complexidade geral é O(n log n).

O melhor caso, que ocorre quando o arquivo de entrada já se encontra ordenado, terá complexidade linear (O (n/b)).

O pior caso, que ocorre quando as partições geradas possuem tamanhos inadequados: uma com maior tamanho possível e outra vazia, possui complexidade quadrática (O2 /Tamanho Área).

3. Análise Dos Resultados:

Foram realizados alguns testes para determinar o desempenho de cada método, de forma a conseguir demonstrar maior eficiência de um método em relação a outro:

Intercalação balanceada de vários caminhos

Intercalação balanceada com método de ordenação Ascendente

Qtd. Registros		100	1000	10000	100000	471705
Criação Bloco	Escritas	105	1050	10500	105000	495291
	Leituras	100	1000	10000	100000	471705
	Comparação	800	8000	80000	800000	3773640
	Tempo (s)	0,0115	0,025	0,0995	0,7487	1,09049
Ordenação	Escritas	101	2004	30028	300264	1888063
	Leituras	106	2055	30529	305265	1911651
	Comparação	800	16000	240000	2400000	15094560
	Tempo (s)	0,009	0,035	0,2726	1,0476	3,0865

Nesse método, vemos que na criação, como escrita o número de elementos + o número de blocos, ja que cada bloco gera um -1 a mais que usamos como controle. Ja as leituras são exatamente o número de elementos do arquivo original. Ja no processo de ordenação, temos N x número de elementos + número de blocos gerados, sendo N o número de vezes que foram feitas intercalações completas. O número de comparações, tanto na criação quanto na ordenação, se dá devido ao jeito que escolhemos trabalhar com nossa área, que é mexendo com um heapsort.

Intercalação balanceada com método de ordenação Descendente

Qtd. Registros		100	1000	10000	100000	471705
Criação Bloco	Escritas	105	1050	10500	105000	495291
	Leituras	100	1000	10000	100000	471705
	Comparação	800	8000	80000	800000	3773640
	Tempo (s)	0,03194	0,0430	0,1169	0,7724	1,75025
Ordenação	Escritas	101	2004	30028	300264	1888063
	Leituras	106	2055	30529	305265	1911651
	Comparação	800	16000	2400000	2400000	15094560
	Tempo (s)	0,0265	0,0685	0,2692	0,7558	3,001378

No caso do arquivo descendente, os resultados são os mesmos que o caso anterior, pois nesse método, sempre são criados os blocos do mesmo tamanho, independendo se os próximos itens são maiores ou menores, ou seja, o processo é fundamentalmente o mesmo.

Intercalação balanceada com método de ordenação Aleatório

Qtd. Registros		100	1000	10000	100000	471705
Criação Bloco	Escritas	105	1050	10500	105000	495291
	Leituras	100	1000	10000	100000	471705
	Comparação	800	8000	80000	800000	3773640
	Tempo (s)	0,0364	0,0278	0,1178	0,77785	0,6455
Ordenação	Escritas	101	2004	30028	300264	1888063
	Leituras	106	2055	30529	305265	1911651
	Comparação	800	16000	240000	2400000	15094560
	Tempo (s)	0,028	0,0419	0,3136	1,31611	5,237385

Assim como nos dois casos anteriores, os resultados são basicamente os mesmos, diferindo apenas um pouco no quesito tempo, que é uma variável que oscila bastante, devido ao clock do computador.

Intercalação com seleção por substituição

Intercalação balanceada com seleção por substituição Ascendente

Qtd. Registros		100	1.000	10.000	100.000	471.705
Criação Bloco	Escritas	101	1001	10001	100001	471706
	Leituras	100	1000	10000	100000	471705
	Comparação	800	8000	80000	800000	3773640
	Tempo (s)	0	0,015625	0,03125	0,1250	0,453125
Ordenação	Escritas	101	1001	10001	100001	471706
	Leituras	102	1002	10002	100002	471707
	Comparação	800	8000	80000	800000	3773640
	Tempo (s)	0	0,015625	0,03125	0,078125	0,343750

A quantidade de escritas na criação de bloco em um arquivo ascendente será o número de (elementos + número de blocos), este + número de blocos vem do coringa que nós escolhemos para separar os blocos, e já que o número de blocos em um arquivo crescente será 1, só teremos que somar 1. A quantidade de leituras será referente a quantidade de elementos. A quantidade de comparações foi composta por “insereArea” e “retiraArea” sendo o valor de cada 3 e 5 respectivamente, então, seria (3*n) + (5*n).

A quantidade de escritas na ordenação em um arquivo ascendente será o número de (elementos + número de blocos), esta soma vem do coringa que nós escolhemos para separar os blocos. A quantidade de leituras será referente a quantidade de (elementos * número de blocos/F + 1) . A quantidade de comparações foi composta por “insereArea” e “retiraArea”

sendo o valor de cada 3 e 5 respectivamente, então, seria (3*n) + (5*n). Já o tempo será maior na criação de blocos já que a intercalação será feita apenas uma vez.

Intercalação balanceada com seleção por substituição Descendente

Qtd. Registros		100	1.000	10.000	100.000	471.705
Criação Bloco	Escritas	104	1025	10082	100232	472296
	Leituras	100	1000	10000	100000	471705
	Comparação	800	8000	80000	800000	3773640
	Tempo (s)	0	0,015625	0,03125	0,1250	0,453125
Ordenação	Escritas	101	2003	20006	200013	1415148
	Leituras	105	2029	20089	200246	1415741
	Comparação	800	16000	160000	1600000	11320920
	Tempo (s)	0	0,015625	0,03125	0,18750	1,5000

A quantidade de escritas na criação de bloco em um arquivo descendente será o número de (elementos + número de blocos), este número vem do coringa que nós escolhemos para separar os blocos. A quantidade de leituras será referente a quantidade de elementos. A quantidade de comparações foi composta por “insereArea” e “retiraArea” sendo o valor de cada 3 e 5 respectivamente, então, seria (3*n) + (5*n).

A quantidade de escritas na ordenação em um arquivo ascendente será referente ao número de vezes que a intercalação é realizada podendo aumentar consideravelmente a quantidade de escritas e leituras. A quantidade de comparações foi composta por “insereArea” e “retiraArea” sendo o valor de cada 3 e 5 respectivamente, então, seria intercalações * ((3*n) + (5*n)). Já o tempo será maior na intercalação já que a criação será feita apenas uma vez enquanto a intercalação será feita várias vezes.

Intercalação balanceada com seleção por substituição Aleatória

Qtd. Registros		100	1.000	10.000	100.000	471.705
Criação Bloco	Escritas	103	1025	10239	102436	483033
	Leituras	100	1000	10000	100000	471705
	Comparação	800	8000	80000	800000	3773640
	Tempo (s)	0	0,015625	0,03125	0,1250	0,484375
Ordenação	Escritas	101	2003	20013	300130	1887419
	Leituras	104	2029	20253	302568	1898750
	Comparação	800	16000	160000	2400000	15094560
	Tempo (s)	0	0,015625	0,046875	0,31250	2,234375

A quantidade de escritas na criação de bloco em um arquivo aleatório será o número de (elementos + número de blocos), este número vem do coringa que nós escolhemos para separar os blocos. A quantidade de leituras será referente a quantidade de elementos. A quantidade de comparações foi composta por “insereArea” e “retiraArea” sendo o valor de cada 3 e 5 respectivamente, então, seria número de intercalações * ((3*n) + (5*n)).

A quantidade de escritas na ordenação em um arquivo aleatório será referente ao número de vezes que a intercalação é realizada podendo aumentar consideravelmente a quantidade de escritas e leituras. A quantidade de comparações foi composta por “insereArea” e “retiraArea” sendo o valor de cada 3 e 5 respectivamente, então, seria (3*n) + (5*n). Já o tempo será maior na intercalação já que a criação será feita apenas uma vez enquanto a intercalação será feita várias vezes.

QuickSort Externo

Como o Quicksort externo não utiliza vetores de criação suas tabelas tem apenas os resultados de pesquisa.

Quicksort Externo Crescente

	Arq.Crescente				
Qtd. Registros	100	1000	10000	100000	471705
Leituras	100	1000	10000	100000	471705
Escritas	100	1000	10000	100000	471705
Comparações	16692	189492	1917492	19197492	90564852
Tempo	0	0.015	0.06	0.60	3.04

Quicksort Externo Decrescente

	Arq.Decrescente				
Qtd. Registros	100	1000	10000	100000	471705
Leituras	133	1852	23196	258998	1461098
Escritas	133	1852	23196	258998	1461098
Comparações	20330	345579	4394597	49551974	280241537
Tempo	0	0.015	0.14	1.67	12.98

Quicksort Externo Aleatório

	Arq.Aleatorio				
Qtd. Registros	100	1000	10000	100000	471705
Leituras	197	4637	70122	876750	4403215
Escritas	197	4637	70122	876750	4403215
Comparações	17162	243931	3139739	41054259	214055815
Tempo	0	0.015	0.34	5.73	36.28

Ao se observar as tabelas vemos que a melhor forma de ordenação é quando o arquivo é crescente, em vista que o tempo de execução do quicksort é muito dependente da quantidade de escritas e leituras que foram feitas. E quanto maior o número de registros procurados maior será o tempo de execução do algoritmo. Para o decrescente e para o aleatório a leituras e escritas no arquivo são exponencialmente maiores que o Crescente. Chamasse a atenção para a relação entre a quantidade de comparações nos arquivos nas situações Decrescente e Aleatório: apesar do arquivo desordenado aleatoriamente possuir um número inferior de comparações, este possui um gasto de tempo superior, indicando que o custo de tempo é mais influenciado pelas leituras e escritas nas memórias.

4. Conclusão:

Conclusão Geral

Neste trabalho, foi criado uma aplicação para executar diferentes métodos de ordenação externa, em diferentes situações de arquivos (crescentes, decrescentes e aleatório)

Analisando os resultados encontrados, vimos que os métodos de intercalação, apresentam menor número de comparações em geral. Ja em relação a leitura e escrita, percebemos que o melhor é a intercalação com seleção por substituição. Contando com as situações iniciais dos arquivos concluímos, vemos que em geral a seleção por substituição é o melhor método, pois no seu pior caso (arquivo descendente), gera o mesmo número de blocos da intercalação normal, enquanto no ascendente, gera um único bloco gigantesco, poupando muitas etapas. Porém, temos um ponto a favor do quicksort, que esta no fato que ele nao precisa gerar arquivos adicionais, ao contrário das intercalações que precisa das fitas.

Tivemos como principal dificuldade as intercalações, ja que foi um código que tivemos que fazer do 0, principalmente para fazer a intercalação trocar entre entrada e saída até poder parar.