

TRABALHO I

Nome: Rafael de Almeida

Nº USP: 11872028

algorithm.c: Faz a leitura de cada arquivo e ordena todos os 5 meses para cada um dos 7 algoritmos, em seguida cria dois arquivos txt por algoritmo.

O primeiro txt: nº linhas x nº comparações.

O segundo txt: nº linhas x nº movimentações.

Os dois arquivos servem para fazer o gráfico no GNU PLOT para cada arquivo.

É necessário as seguintes pastas:

“meses”: com os dados dos meses txt dentro.

“algs”: contém as pastas “inserction”, “binary”, “selection”, “bubble”, “merge”, “heap” e “quick”.

(As pastas já estão lá no código em anexo)

ALGORITHM.C:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void inserction_sort(char** code, int n, int* C, int *M){
    for(int i = 2; i <= n; i++){

        char* tmp = code[i]; (*M)++;
        code[0] = tmp;      (*M)++;
        int j = i;
        (*C)++;
        while(strcmp(tmp,code[j-1]) < 0){
            code[j] = code[j-1]; (*M)++;
            j = j-1;
            (*C)++;
        }
        code[j] = tmp; (*M)++;
    }
}

void binary_inserction_sort(char** code, int n, int* C, int *M){
    for(int i = 2; i <= n; i++){

        char* tmp = code[i]; (*M)++;
        int L = 1;
        int R = i;

        while(L < R){
            int m = (L + R)/2;
            (*C)++;
        }
    }
}
```

```

        if(strcmp(code[m],tmp) < 0)
            L = m + 1;
        else
            R = m;
    }

    int j = i;
    while(j > R){
        code[j] = code[j-1];    (*M)++;
        j = j-1;
    }
    code[R] = tmp;              (*M)++;
}
}

void selection_sort(char** code, int n, int* C, int *M){
    int menor;

    for(int i = 1; i <= n-1; i++){
        menor = i;

        for(int j = i+1; j <= n; j++){
            (*C)++;
            if(strcmp(code[menor],code[j]) > 0){
                menor = j;
            }
        }

        char* tmp = code[i]; (*M)++;
        code[i] = code[menor];    (*M)++;
        code[menor] = tmp;        (*M)++;
    }
}

void bubble_sort(char** code, int n, int* C, int *M){

    for(int i = 2; i <= n; i++){
        for(int j = n; j >= i; j--){
            (*C)++;
            if(strcmp(code[j],code[j-1]) < 0){
                char* tmp = code[j]; (*M)++;
                code[j] = code[j-1];    (*M)++;
                code[j-1] = tmp;        (*M)++;
            }
        }
    }
}

void merge(char** code, int L, int h, int R, char** c, int* C, int *M){
    int i = L;
    int j = h + 1;
    int k = L - 1;

```

```

while(i <= h && j <= R){
    k++;
    (*C)++;
    if(strcmp(code[i],code[j]) < 0){
        c[k] = code[i];    (*M)++;
        i++;
    }
    else{
        c[k] = code[j];    (*M)++;
        j++;
    }
}
while(i <= h){
    k++;
    c[k] = code[i];    (*M)++;
    i++;
}
while(j <= R){
    k++;
    c[k] = code[j];    (*M)++;
    j++;
}
}

void mpass(char** code, int N, int p, char** c, int* C, int *M){
    int i = 1;
    while(i <= N-2*p+1){
        merge(code,i,i+p-1,i+2*p-1,c,C,M);
        i += 2*p;
    }

    if(i+p-1 < N){
        merge(code,i,i+p-1,N,c,C,M);
    }
    else{
        for(int j = i; j <= N; j++){
            c[j] = code[j];    (*M)++;
        }
    }
}

void merge_sort(char** code, int N, int* C, int *M){
    int p = 1;

    char** c;
    c = (char**) malloc((N+1)*sizeof(char*));

    while(p < N){
        mpass(code,N,p,c,C,M);
        p *= 2;
        mpass(c,N,p,code,C,M);
        p *= 2;
    }
}

```

```

    free(c);
}

void heapify(char** code, int L, int R, int* C, int *M){
    int i = L;
    int j = 2*L;
    char* tmp = code[L]; (*M)++;

    (*C)++;
    if((j < R) && strcmp(code[j],code[j+1]) < 0){
        j++;
    }

    (*C)++;
    while((j <= R) && (strcmp(tmp,code[j]) < 0)){
        code[i] = code[j]; (*M)++;
        i = j;
        j = 2*j;
        if((j < R) && (strcmp(code[j],code[j+1]) < 0)){
            j++;
        }
        (*C)++;
    }
    code[i] = tmp; (*M)++;
}

void heap_sort(char** code, int n, int* C, int *M){
    for(int L = n/2; L >= 1; L--){
        heapify(code,L,n,C,M);
    }
    for(int R = n; R >= 2; R--){
        char* w = code[1]; (*M)++;
        code[1] = code[R]; (*M)++;
        code[R] = w; (*M)++;
        heapify(code,1,R-1,C,M);
    }
}

void quick_sort(char** code, int L, int R, int* C, int *M){
    int i = L;
    int j = R;
    char* mid = code[(L+R)/2]; (*M)++;
    do{
        (*C)++;
        while(strcmp(code[i],mid) < 0){
            i++;
            (*C)++;
        }
        (*C)++;
        while(strcmp(mid,code[j]) < 0){
            j--;
            (*C)++;
        }
    } while(i < j);
    code[i] = mid; (*M)++;
    quick_sort(code,L,i-1,C,M);
    quick_sort(code,i+1,R,C,M);
}

```

```

    }
    if(i <= j){
        char* tmp = code[i];    (*M)++;
        code[i] = code[j];    (*M)++;
        code[j] = tmp;        (*M)++;
        i++;
        j--;
    }
}while(i <= j);
if(L < j)
    quick_sort(code,L,j,C,M);
if(i < R)
    quick_sort(code,i,R,C,M);
}

//Algoritmo que retorna o número de linhas do arquivo
int get_number_lines(FILE* file){
    char c;
    int n = 0;
    while(!feof(file)){
        c = getc(file);
        if(c == '\n'){
            n++;
        }
    }

    return n;
}

void intiataNumberLines(int* n){
    FILE* file;

    file = fopen("meses//mes_1.txt","r");
    n[0] = get_number_lines(file) - 1;
    fclose(file);

    file = fopen("meses//mes_2.txt","r");
    n[1] = get_number_lines(file) - 1;
    fclose(file);

    file = fopen("meses//mes_3.txt","r");
    n[2] = get_number_lines(file) - 1;
    fclose(file);

    file = fopen("meses//mes_4.txt","r");
    n[3] = get_number_lines(file) - 1;
    fclose(file);

    file = fopen("meses//mes_5.txt","r");
    n[4] = get_number_lines(file) - 1;
    fclose(file);
}

```

```

//Escolhe um método de ordenação com base na opcao, na ordem de escrita dos arquivos
void ordenarVetor(char** code, int* n, int i, int opcao, int* c, int* m){
    if(opcao == 1){
        inserction_sort(code,n[i-1],c,m);
    }
    if(opcao == 2){
        binary_inserction_sort(code,n[i-1],c,m);
    }
    if(opcao == 3){
        selection_sort(code,n[i-1],c,m);
    }
    if(opcao == 4){
        bubble_sort(code,n[i-1],c,m);
    }
    if(opcao == 5){
        merge_sort(code,n[i-1],c,m);
    }
    if(opcao == 6){
        heap_sort(code,n[i-1],c,m);
    }
    if(opcao == 7){
        quick_sort(code,1,n[i-1],c,m);
    }
}

char** cria_vetor(FILE* file, int n){
    char** code;
    code = (char**) malloc((n+1)*sizeof(char*));

    for(int i = 1; i <= n; i++){
        code[i] = (char*) malloc(11*sizeof(char));
        fscanf(file,"%s",code[i]);
    }

    return code;
}

void apaga_vetor(char*** code, int n){
    for(int i = 1; i <= n; i++){
        free((*code)[i]);
    }
    free(*code);
}

int main(){

    int n[5];

    initiateNumberLines(n); //carrega no vetor n o numero de linhas de cada arquivo

    //Os vetores de strings abaixo servem para facilitar o nomenclatura dos arquivos
    char arquivoComparacoes[7][30] =
    {"algs//inserction//cmp.txt",

```

```

"algs//binary//cmp.txt",
"algs//selection//cmp.txt",
"algs//bubble//cmp.txt",
"algs//merge//cmp.txt",
"algs//heap//cmp.txt",
"algs//quick//cmp.txt"};

char arquivoMovimentacoes[7][30] =
{"algs//inserction//mov.txt",
"algs//binary//mov.txt",
"algs//selection//mov.txt",
"algs//bubble//mov.txt",
"algs//merge//mov.txt",
"algs//heap//mov.txt",
"algs//quick//mov.txt"};

char mes[5][20] =
{"meses//mes_1.txt",
"meses//mes_2.txt",
"meses//mes_3.txt",
"meses//mes_4.txt",
"meses//mes_5.txt"};

for(int opcao = 1; opcao <= 7; opcao++){    //roda uma vez por cada algoritmo de
ordenação

    FILE* cmp = fopen(arquivoComparacoes[opcao-1],"w"); //arquivo para escrita das
comparações
    if(cmp == NULL) exit(1);

    FILE* mov = fopen(arquivoMovimentacoes[opcao-1],"w");    //arquivo para escrita
das movimentações
    if(mov == NULL) exit(1);

    for(int i = 1; i <= 5; i++){    //roda para cada mês

        FILE* file = fopen(mes[i-1],"r");    //arquivo para leitura do mês
        if(file == NULL) exit(1);

        char** code = cria_vetor(file,n[i-1]);

        int c = 0,m = 0;    //n° de comparações e movimentações

        ordenarVetor(code,n,i,opcao,&c,&m);

        fprintf(cmp,"%d\t%d\n", n[i-1],c); //Escreve no arquivo cmp.txt linhas x
comparações
        fprintf(mov,"%d\t%d\n", n[i-1],m); //Escreve no arquivo mov.txt linhas x
movimentações

        apaga_vetor(&code,n[i-1]);

        fclose(file);

```

```

    }

    fclose(cmp);
    fclose(mov);
}

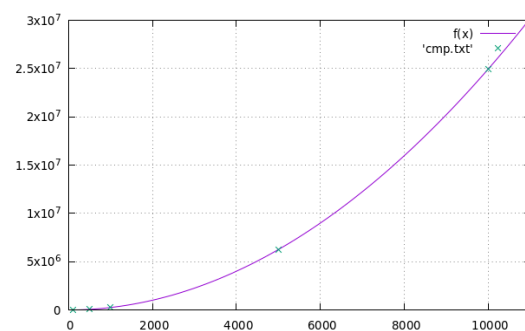
return 0;
}

```

INSERCTION SORT:

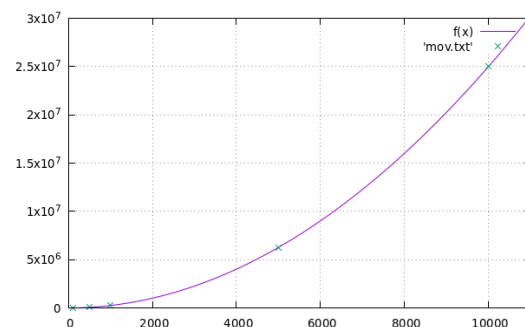
linhas x comparações

100	2616
500	64221
1000	257894
5000	6272994
10000	24991689



linhas x movimentações

100	2418
500	63223
1000	255896
5000	6262996
10000	24971691

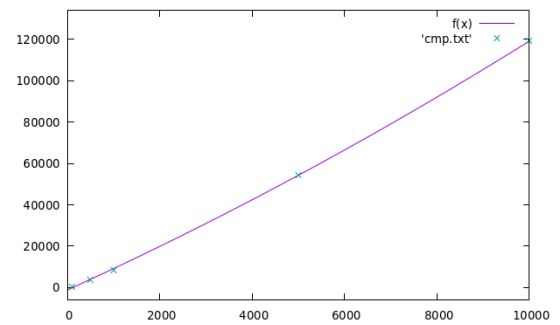


Os dois gráficos crescem em forma de função quadrática, logo condiz com a complexidade média do inserction sort ser n^2 .

BINARY INSERTION SORT:

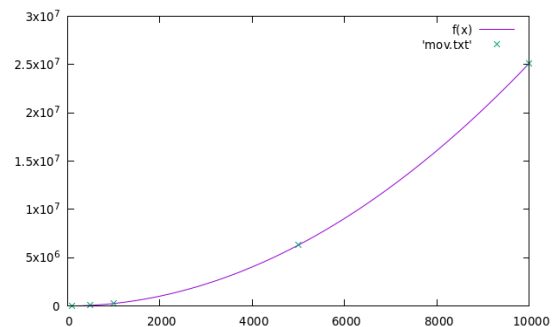
linhas x comparações

100	2534
500	64016
1000	258315
5000	6310265
10000	25145509



linhas x movimentações

100	533
500	3809
1000	8567
5000	54532
10000	119060

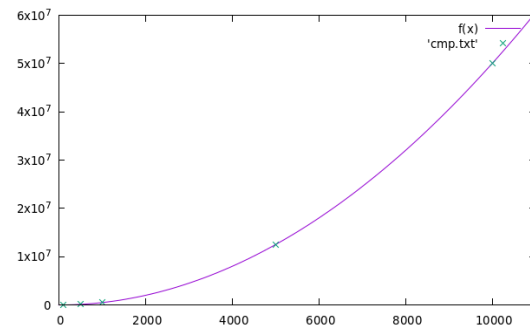


Pelo menos um gráfico cresce em forma de função quadrática, logo condiz com a complexidade média do binary insertion sort ser n^2 .

SELECTION SORT:

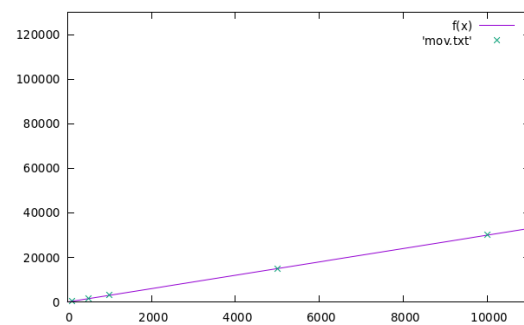
linhas x comparações

100	4950
500	124750
1000	499500
5000	12497500
10000	49995000



linhas x movimentações

100	297
500	1497
1000	2997
5000	14997
10000	29997

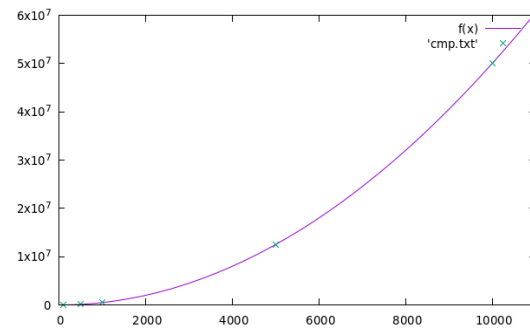


Pelo menos um gráfico cresce em forma de função quadrática, logo condiz com a complexidade média do selection sort ser n^2 .

BUBBLE SORT:

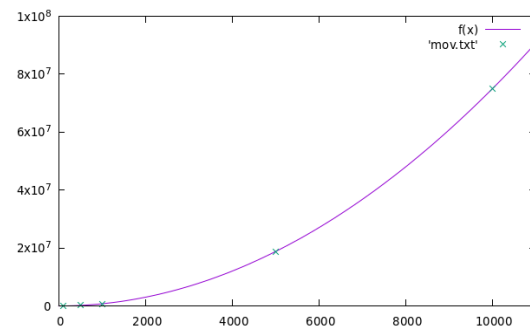
linhas x comparações

100	4950
500	124750
1000	499500
5000	12497500
10000	49995000



linhas x movimentações

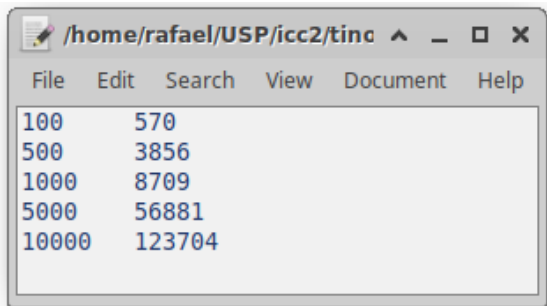
100	6957
500	188172
1000	764691
5000	18773991
10000	74885076



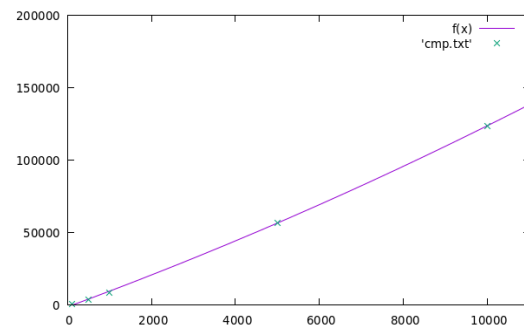
Os dois gráficos crescem em forma de função quadrática, logo condiz com a complexidade média do bubble sort ser n^2 .

MERGE SORT:

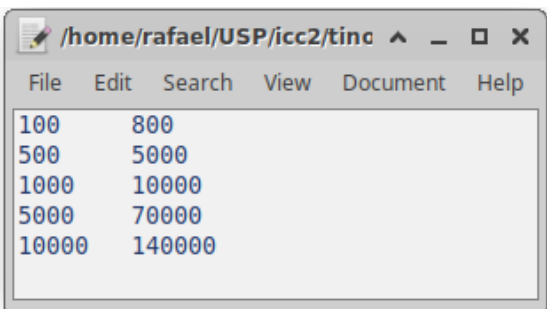
linhas x comparações



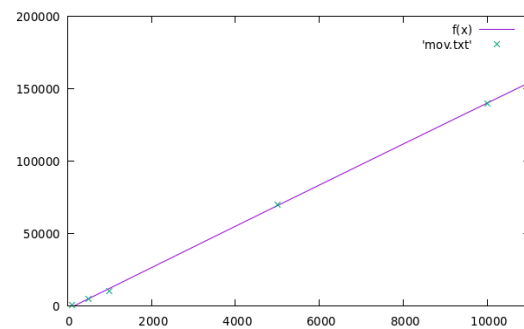
100	570
500	3856
1000	8709
5000	56881
10000	123704



linhas x movimentações



100	800
500	5000
1000	10000
5000	70000
10000	140000

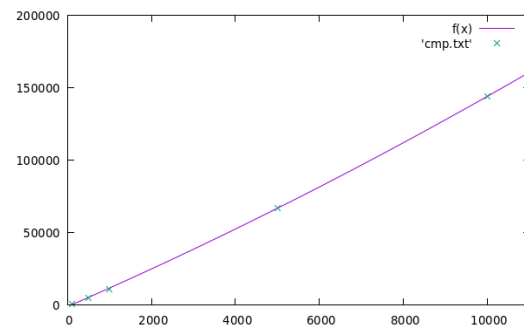


Os dois gráficos crescem muito pouco para os número de linhas, logo condiz com a complexidade média do merge sort ser $n \cdot \log n$.

HEAP SORT:

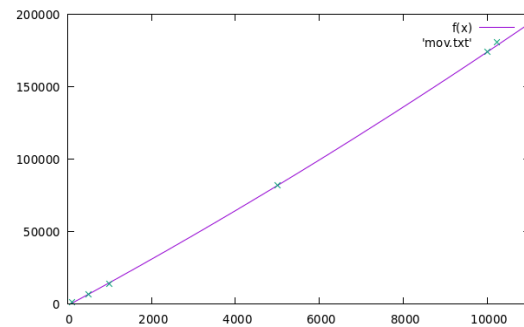
linhas x comparações

100	777
500	5013
1000	11055
5000	67027
10000	144024



linhas x movimentações

100	1074
500	6510
1000	14052
5000	82024
10000	174021

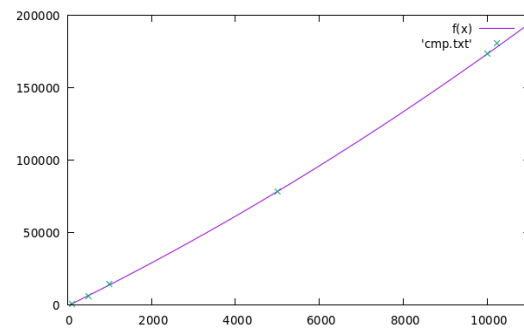


Os dois gráficos crescem muito pouco para os número de linhas, logo condiz com a complexidade média do heap sort ser $n \cdot \log n$.

QUICK SORT:

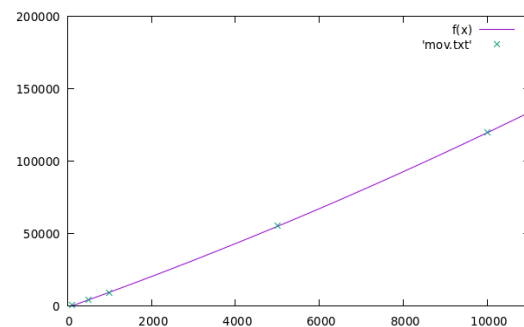
linhas x comparações

100	797
500	6002
1000	14423
5000	78261
10000	173425



linhas x movimentações

100	589
500	4051
1000	8908
5000	55193
10000	119651



Os dois gráficos crescem muito pouco para os número de linhas, logo condiz com a complexidade média do merge sort ser $n \log n$.

Melhores algoritmos: Observando o número de comparações e movimentações para os casos com maiores números de linhas é possível observar que o melhor algoritmo para se usar no problema em questão de linhas x movimentações é o **quick sort**, já o melhor algoritmo em questão de linhas x comparações é o **merge sort**.

Em geral os 3 últimos algoritmos de complexidade média $n \log n$ são bem melhores que os 4 primeiros de complexidade média n^2 .

app.c: Aplicativo que o professor pediu, cria os arquivos ordenados de cada mês, em seguida utiliza outro arquivo para consultar cada mês e alocar em um vetor de strings, usa a busca binária para buscar em cada mês o código.

Entrada: código digitado pelo usuário.

Saída: Se for possível encontrar o código, exibe o mês e posição.

APP.C:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void merge(char** code, int L, int h, int R, char** c){
    int i = L;
    int j = h + 1;
    int k = L - 1;

    while(i <= h && j <= R){
        k++;
        if(strcmp(code[i],code[j]) < 0){
            c[k] = code[i];
            i++;
        }
        else{
            c[k] = code[j];
            j++;
        }
    }
    while(i <= h){
        k++;
        c[k] = code[i];
        i++;
    }
    while(j <= R){
        k++;
        c[k] = code[j];
        j++;
    }
}

void mpass(char** code, int N, int p, char** c){
    int i = 1;
    while(i <= N-2*p+1){
        merge(code,i,i+p-1,i+2*p-1,c);
        i += 2*p;
    }

    if(i+p-1 < N){
        merge(code,i,i+p-1,N,c);
    }
    else{
        for(int j = i; j <= N; j++){

```

```

        c[j] = code[j];
    }
}

void merge_sort(char** code, int N){ //feito
    int p = 1;

    char** c;
    c = (char**) malloc((N+1)*sizeof(char*));

    while(p < N){
        mpass(code,N,p,c);
        p *= 2;
        mpass(c,N,p,code);
        p *= 2;
    }

    free(c);
}

int busca_binaria(char* chave,char** code,int left, int right){
    if(left > right) return -1;

    int mid = (left + right)/2;

    int cmp = strcmp(code[mid],chave);

    if(cmp == 0) return mid;
    else if(cmp < 0) return busca_binaria(chave,code,mid+1,right);
    else return busca_binaria(chave,code,left,mid-1);
}

int get_number_lines(FILE* file){
    char c;
    int n = 0;
    while(!feof(file)){
        c = getc(file);
        if(c == '\n'){
            n++;
        }
    }

    return n;
}

void initiateNumberLines(int* n){
    FILE* file;

    file = fopen("meses//mes_1.txt","r");
    n[0] = get_number_lines(file) - 1;
    fclose(file);
}

```



```

    file = fopen("meses//mes_2.txt", "r");
    n[1] = get_number_lines(file) - 1;
    fclose(file);

    file = fopen("meses//mes_3.txt", "r");
    n[2] = get_number_lines(file) - 1;
    fclose(file);

    file = fopen("meses//mes_4.txt", "r");
    n[3] = get_number_lines(file) - 1;
    fclose(file);

    file = fopen("meses//mes_5.txt", "r");
    n[4] = get_number_lines(file) - 1;
    fclose(file);
}

char** cria_vetor(FILE* file, int n){
    char** code;
    code = (char**) malloc((n+1)*sizeof(char*));

    for(int i = 1; i <= n; i++){
        code[i] = (char*) malloc(11*sizeof(char));
        fscanf(file, "%s", code[i]);
    }

    return code;
}

void apaga_vetor(char*** code, int n){
    for(int i = 1; i <= n; i++){
        free((*code)[i]);
    }
    free(*code);
}

int main(){

    int n[5];

    intiateNumberLines(n); //carrega no vetor n o numero de linhas de cada arquivo

    char mes[5][20] =
    {"meses//mes_1.txt",
    "meses//mes_2.txt",
    "meses//mes_3.txt",
    "meses//mes_4.txt",
    "meses//mes_5.txt"};

    char mesOrdenado[5][30] =
    {"mesesOrdenados//mes_1.txt",
    "mesesOrdenados//mes_2.txt",
    "mesesOrdenados//mes_3.txt",

```

```

"mesesOrdenados//mes_4.txt",
"mesesOrdenados//mes_5.txt"};

// Ordenação
for(int i = 1; i <= 5; i++){

    FILE* file = fopen(mes[i-1], "r");
    if(file == NULL) exit(1);

    FILE* ord = fopen(mesOrdenado[i-1], "w");
    if(ord == NULL) exit(1);

    char** code = cria_vetor(file, n[i-1]);

    merge_sort(code, n[i-1]);

    for(int j = 1; j <= n[i-1]; j++){
        fprintf(ord, "%s\n", code[j]);
    }

    apaga_vetor(&code, n[i-1]);

    fclose(file);
    fclose(ord);
}

char chave[11];
scanf("%s", chave);

// busca binaria do arquivo
for(int i = 1; i <= 5; i++){

    FILE* file = fopen(mesOrdenado[i-1], "r");
    if(file == NULL) exit(1);

    char** code = cria_vetor(file, n[i-1]);

    int bb = busca_binaria(chave, code, 1, n[i-1]);

    apaga_vetor(&code, n[i-1]);

    fclose(file);

    if(bb != -1){
        printf("O codigo foi achado no mes %d na posicao %d\n", i, bb);
        return 0;
    }
}

printf("Nao achou o codigo : (\n");

return 0;
}

```