# Link para acesso dos exercícios no GITHUB:

https://github.com/vinihgomes/Aulas-EDDA1/tree/master/Semana3

# Questão 1) Para cada exemplo apresentado em aula, elabore o respectivo programa-teste de Merge sort, Heap sort e Quick sort.

# // Criei só 1 programa que chama ambas as funções.

#include <stdlib.h>

#include <stdio.h>

#include <locale.h>

#include <conio.h>

// DECLARAÇÃO DE FUNÇÕES GLOBAIS :D

float abastece\_notas(float \* notas);

float merge\_sort(float \* notas, int inicio, int fim);

float heap\_sort(float \* notas);

void quick\_sort(float \* notas, int ini, int fim);

void troca (float \*A, float \*B);

// Varíaveis globais (só pra facilitar a passagem de parâmetros)

int TAMANHO\_VETOR, inicio = 0, fim;

// Código MAIN - Pergunta quantas notas serão recebidas, chama função para preencher com valores aleatórios e pergunta qual método de ordenação deseja realizar.

int main(){

int opc, i;

float \*notas;

setlocale (LC\_ALL, "Portuguese");

printf("Informe quantas notas deseja alocar: \n");

scanf("%d", &TAMANHO\_VETOR);

notas = (float \*) malloc (TAMANHO\_VETOR \* sizeof(float));

abastece\_notas(notas);

for (i = 0; i < TAMANHO\_VETOR ; i++)

{

printf("\nPosição alocada %d: %.2f", i+1, notas[i]);

}

fim = TAMANHO\_VETOR;

printf("\nPronto, iremos para a próxima etapa! Aperte qualquer tecla para continuar:");

getch();

system("cls");

printf("Informe qual método de ordenação deseja utilizar: \n");

printf("1 - Merge sort\n");

printf("2 - Heap sort\n");

printf("3 - Quick sort\n");

scanf("%d", &opc);

switch(opc){

case 1:

merge\_sort(notas, inicio, fim);

for (i = 1; i <= TAMANHO\_VETOR ; i++)

{

printf("\nOrdenado por Merge Sort: Posição %i: \t %.2f\n", i, notas[i]);

}

break;

case 2:

heap\_sort(notas);

for (i = 0; i < TAMANHO\_VETOR ; i++)

{

printf("\nOrdenado por Heap Sort: Posição %i: \t %.2f\n", i+1, notas[i]);

}

break;

case 3:

quick\_sort(notas, inicio, fim);

for (i = 1; i <= TAMANHO\_VETOR ; i++)

{

printf("\nOrdenado por Quick Sort: Posição %i: \t %.2f\n", i, notas[i]);

}

break;

default: printf("\nOpção inválida");

}

}

///////////////////////////////////////////////////////////////////////////////////////////////////////// OK OK OK OK OK OK OK OK OK

// Ordenação através de Merge Sort - Função recursiva, chama a ela mesma e também a função auxiliar MERGE

float merge\_sort (float \* notas, int inicio, int fim)

{

float merge(float \* notas, int inicio, int meio, int fim);

int meio;

if ( inicio < fim )

{

meio = (inicio + fim) / 2; /\*calcula o meio \*/

merge\_sort (notas, inicio, meio); /\*ordena o subvetor esquerdo \*/

merge\_sort (notas, meio+1, fim ); /\*ordena o subvetor direito \*/

merge (notas, inicio, meio, fim); /\*funde os subvetores esquerdo e direito\*/

}

}

///////////////////////////////////////////////////////////////////////////////////////////////////////// OK OK OK OK OK OK OK OK OK

// Função merge, auxiliar do merge sort

float merge (float \* vetor, int inicio, int meio, int fim)

{

int tamEsq, tamDir, i, idxEsq, idxDir;

float \*vetor\_Esq;

float \*vetor\_Dir;

tamEsq = meio - inicio + 1; /\* tamanho do subvetor esquerdo \*/

tamDir = fim - meio; /\* tamanho do subvetor direito \*/

/\* Tenta alocar tamEsq posições para abastecer o subvetor esquerdo \*/

vetor\_Esq = (float \*) malloc (tamEsq \* sizeof (float));

if (vetor\_Esq == NULL)

{ printf ("malloc devolveu NULL!\n");

getch();

exit (EXIT\_FAILURE);

}

/\* Tenta alocar tamDir posições para abastecer o subvetor direito \*/

vetor\_Dir = (float \*) malloc (tamDir \* sizeof (float));

if (vetor\_Dir == NULL)

{ printf ("malloc devolveu NULL!\n");

getch();

exit (EXIT\_FAILURE);

}

for (i=0; i<tamEsq; i++) /\* inicializar vetor\_Esq \*/

vetor\_Esq[i] = vetor[inicio+i]; /\* elementos do subvetor esquerdo \*/

for (i=0; i<tamDir; i++) /\* inicializar vetor\_Dir \*/

vetor\_Dir[i] = vetor[meio+1+i]; /\* elementos do subvetor direito \*/

idxEsq = 0; /\* índice do subvetor auxiliar esquerdo \*/

idxDir = 0; /\* índice do subvetor auxiliar direito \*/

for ( i=inicio; i<=fim; i++ ) /\* intercala os vetores \*/

if (idxEsq < tamEsq)

if (idxDir < tamDir)

if (vetor\_Esq[idxEsq] < vetor\_Dir[idxDir])

vetor[i] = vetor\_Esq[idxEsq++];

else

vetor[i] = vetor\_Dir[idxDir++];

else

vetor[i] = vetor\_Esq[idxEsq++];

else

vetor[i] = vetor\_Dir[idxDir++];

}

///////////////////////////////////////////////////////////////////////////////////////////////////////// OK OK OK OK OK OK OK OK OK

// Método de ordenação através de Heap sort, cria um vetor onde remove o valor pai (maior) e após isso é chamada as funções auxiliares

float heap\_sort(float \* notas){

int esquerdo(int i) ;

int direito(int i);

int pai(int i);

void BuildHeap(float \*notas);

int i, atual;

BuildHeap(notas);

for(i=TAMANHO\_VETOR-1; i>0; i--) /\* percorre da última até a penúltima posição do vetor \*/

{ /\* (se todas as posições até a penúltima estiverem ordenadas, a última também estará) \*/

troca( &notas[0], &notas[i] ); /\* troca a posição corrente do heap com a raiz \*/

/\* i não pertence mais ao vetor \*/

/\* (o vetor vai diminuindo de tamanho, pois a última posição CERTAMENTE terá o maior valor) \*/

/\* ajusta o heap \*/

atual = 0;

while ( direito(atual) <= (i-1) /\* Enquanto tiver 2 filhos \*/

&& /\* E \*/

( notas[atual] < notas[esquerdo(atual)] || /\* o pai for menor que um dos dois filhos... \*/

notas[atual] < notas[direito(atual)] ) )

{ /\*...troca o pai com o maior dos filhos \*/

if ( notas[esquerdo(atual)] > notas[direito(atual)] )

{

troca( &notas[atual], &notas[esquerdo(atual)] ); /\*troca o atual com o filho esquerdo\*/

atual = esquerdo(atual);

}

else

{

troca( &notas[atual], &notas[direito(atual)] ); /\*troca o atual com o filho direito\*/

atual = direito(atual);

}

}

/\* Verificar se sobrou um filho (esquerdo) maior que o atual \*/

if ( esquerdo(atual) <= i-1 && notas[atual] < notas[esquerdo(atual)] )

{

troca( &notas[atual], &notas[esquerdo(atual)] ); /\*troca o atual com o filho esquerdo\*/

atual = esquerdo(atual);

}

}

}

///////////////////////////////////////////////////////////////////////////////////////////////////////// OK OK OK OK OK OK OK OK OK

// Função auxiliar de Heap Sort.

int esquerdo(int i)

{

return (2\*i+1);

}

// Função auxiliar de Heap Sort.

int direito(int i)

{

return (2\*i+2);

}

// Função auxiliar de Heap Sort.

int pai(int i)

{

return ( (i-1)/2 );

}

// Constrói um heap de máximo (desce desde a posição 1 até os elementos que não são folhas). Rearranja um vetor[1..TAMANHO\_VETOR] de modo a transformá-lo em heap.

void BuildHeap(float \*vetor)

{

int i, atual;

/\* percorre todo o vetor a partir do segundo elemento \*/

/\* pois o primeiro é a raiz do heap e antes dele nada há\*/

for(i=1; i<TAMANHO\_VETOR; i++)

{

atual = i;

/\* enquanto o atual for maior que o pai...\*/

while(atual > 0 && vetor[atual] > vetor[pai(atual)])

{

/\*...troca o atual com o pai... \*/

troca( &vetor[atual], &vetor[pai(atual)] );

/\*...e o atual passa a ser o pai \*/

atual = pai(atual);

}

}

}

///////////////////////////////////////////////////////////////////////////////////////////////////////// OK OK OK OK OK OK OK OK OK

//Função de ordenação Quick Sort

void quick\_sort (float \* notas, int inicio, int fim)

{

int particiona (float \* notas, int ini, int pivo);

int pivo;

if ( inicio < fim ) /\* Caso base \*/

{

pivo = particiona ( notas, inicio, fim ); /\* Particiona o vetor\*/

quick\_sort( notas, inicio, pivo-1 ); /\* Ordena do início do vetor até antes do pivô \*/

quick\_sort( notas, pivo+1, fim); /\* Ordena desde após o pivô até o fim do vetor \*/

}

}

//Função auxiliar quick sort

int particiona (float \* vetor, int inicio, int pivo)

{

int i;

int p\_maior = inicio; /\* a posição do maior é a do início do vetor \*/

for ( i=inicio; i<pivo; i++ ) /\* percorre o vetor da posição ini até a do pivô \*/

{

if ( vetor[i] < vetor[pivo] ) /\*se o elemento da posição atual for menor que o pivô...\*/

{

troca( &vetor[i], &vetor[p\_maior] ); /\*...troca o atual com o maior \*/

p\_maior++; /\* o maior avança uma posição \*/

}

}

/\* percorrido o vetor, o pivô troca de posição com a do maior elemento... \*/

troca( &vetor[p\_maior], &vetor[pivo] );

return p\_maior; /\* ... e o maior elemento passa a ser o novo pivô \*/

}

// Função auxiliar de Heap sort e merge sort

void troca (float \*A, float \*B)

{

float aux = \*A;

\*A = \*B;

\*B = aux;

}

// Função para gerar números aleatórios para o vetor notas. Os valores após o % podem ser alterados para obter números de acordo com seus critérios

float abastece\_notas (float \* notas)

{

int i;

for (i=0; i<TAMANHO\_VETOR; i++)

do

{

notas[i] = rand()%10\*1.13;

}

while (notas[i] > 10);

}