**Trabalho 1**

**Nomes dos participantes:**

* Davi Gabriel Domingues (Número USP: 15447497)
* Francisco Eduardo Fontenele (Número USP: 15452569)

Tem – se as seguintes respostas para cada item desejado pela atividade em questão:

1. Considerar-se-á este exemplo de aplicação como resposta ilustrativa para o comando explicitado:

Texto

Descrição gerada automaticamente

1. O teste da ordenação correta dos valores presentes no vetor {45,56,12,43,95,19,8,67} é o seguinte:

Texto

Descrição gerada automaticamente

1. Os gráficos correspondentes de cada uma das situações são estes (Considerando n como o valor de entrada do vetor os valores, foi-se utilizado esta faixa de valores: 150.000, 200.000, 250.000, 300.000, 400.000):

(1) Vetores de entrada em ordem crescente

(2) Vetores de entrada em ordem decrescente

(3) Vetores de entrada com valores aleatórios

1. A partir dos gráficos obtidos, verifica-se que, quanto às respectivas análises assintóticas, os resultados dos algoritmos solicitados pela atividade se mostram, de fato, condizentes com as devidas análises assintóticas associadas a cada um dos programas. (A verificar)
2. Baseado nos resultados obtidos, conclui-se que o(s) algoritmos(s) de ordenação mais apropriado(s) é(são) para vetores de entrada em ordem crescente, para vetores de entrada em ordem decrescente e para vetores de entrada com valores aleatórios. (Colocar os devidos algoritmos 🡪 A preencher)

**Código usado para o trabalho:**

#include <ctime>

#include <iostream>

#include <cstdlib>

using namespace std;

// --------------------------------------------------------------------------------------

// prototipagem das funções não relacionadas diretamente à main():

void ordenacaoInsercaoDireta(int vetor1[], int tamanho);

void ordenacaoInsercaoBinaria(int vetor1[], int tamanho);

void ordenacaoSelecao(int vetor1[], int tamanho);

void ordenacaoBubbleSort(int vetor1[], int tamanho);

void ordenacaoHeapsort(int vetor1[], int tamanho);

void ordenacaoFusao(int vetor1[], int tamanho);

void ordenacaoQuickSort(int vetor1[], int tamanho);

// --------------------------------------------------------------------------------------

// função auxiliar de cada método relacionado diretamente à main():

void impressaoVetor(int vetor1[], int tamanho);

// --------------------------------------------------------------------------------------

// prototipagem das funções auxiliares de alguns dos processos de ordenação supracitados:

// Heapsort:

void ordenacaoHeapsortHeapify(int vetor1[], int L, int R);

// Fusão direta/Mergesort:

void mergePass(int A[], int tamanho, int particao, int C[]);

void processoFusao(int A[], int esquerda, int meio, int direita, int C[]);

// --------------------------------------------------------------------------------------

// QuickSort:

void aplicacaoOrdenacaoQuickSort(int A[], int esquerda, int direita);

// --------------------------------------------------------------------------------------

// variável global de ordenação (auxiliar universal):

int x;

/\*variáveis globais de contagem de total de comparações e de movimentações para cada algoritmo de ordenação:

Observação: essas variáveis estão relacionadas apenas às chaves do vetor1\*/

int totalComparacoes1, totalMovimentacoes1;

int totalComparacoes2, totalMovimentacoes2;

int totalComparacoes3, totalMovimentacoes3;

int totalComparacoes4, totalMovimentacoes4;

int totalComparacoes5, totalMovimentacoes5;

int totalComparacoes6, totalMovimentacoes6;

int totalComparacoes7, totalMovimentacoes7;

void preencherVetor(int vetor1[], int vetor2[], int tamanho) {

for (int i = 1; i <= tamanho; i++) {

cout << "Valor " << i << ": ";

cin >> vetor1[i];

vetor2[i] = vetor1[i];

}

}

void impressaoVetor(int vetor1[], int tamanho) {

cout << "[ ";

for (int i = 1; i <= tamanho; i++)

cout << vetor1[i] << " ";

cout << "]" << endl;

}

void desordenarVetor(int vetor1[], int vetor2[], int tamanho) {

for (int i = 1; i <= tamanho; i++)

vetor1[i] = vetor2[i];

}

void insercaoDireta(int vetor1[], int tamanho) {

clock\_t t1, t2;

t1 = clock();

ordenacaoInsercaoDireta(vetor1, tamanho);

t2 = clock();

cout << endl<< "Tempo final de execucao da ordenacao por insercao direta: "<< difftime(t2, t1) / CLOCKS\_PER\_SEC << endl;

cout<<"Total de comparacoes realizadas entre as chaves do vetor declarado: "<<totalComparacoes1<<endl;

cout<<"Total de movimentacoes realizadas entre as chaves do vetor declarado: "<<totalMovimentacoes1<<endl<<endl;

//impressaoVetor(vetor1, tamanho);

}

void insercaoBinaria(int vetor1[], int tamanho) {

clock\_t t3, t4;

t3 = clock();

ordenacaoInsercaoBinaria(vetor1, tamanho);

t4 = clock();

cout << endl<< "Tempo final de execucao da ordenacao por insercao binaria: "<< difftime(t4, t3) / CLOCKS\_PER\_SEC << endl;

cout<<"Total de comparacoes realizadas entre as chaves do vetor declarado: "<<totalComparacoes2<<endl;

cout<<"Total de movimentacoes realizadas entre as chaves do vetor declarado: "<<totalMovimentacoes2<<endl<<endl;

//impressaoVetor(vetor1, tamanho);

}

void selecao(int vetor1[], int tamanho) {

clock\_t t5, t6;

t5 = clock();

ordenacaoSelecao(vetor1, tamanho);

t6 = clock();

cout << endl<< "Tempo final de execucao da ordenacao por selecao: "<< difftime(t6, t5) / CLOCKS\_PER\_SEC << endl;

cout<<"Total de comparacoes realizadas entre as chaves do vetor declarado: "<<totalComparacoes3<<endl;

cout<<"Total de movimentacoes realizadas entre as chaves do vetor declarado: "<<totalMovimentacoes3<<endl<<endl;

//impressaoVetor(vetor1, tamanho);

}

void bubbleSort(int vetor1[], int tamanho) {

clock\_t t7, t8;

t7 = clock();

ordenacaoBubbleSort(vetor1, tamanho);

t8 = clock();

cout << endl<< "Tempo final de execucao da ordenacao por Bubblesort: "<< difftime(t8, t7) / CLOCKS\_PER\_SEC << endl;

cout<<"Total de comparacoes realizadas entre as chaves do vetor declarado: "<<totalComparacoes4<<endl;

cout<<"Total de movimentacoes realizadas entre as chaves do vetor declarado: "<<totalMovimentacoes4<<endl<<endl;

//impressaoVetor(vetor1, tamanho);

}

void heapsort(int vetor1[], int tamanho){

clock\_t t9, t10;

t9 = clock();

ordenacaoHeapsort(vetor1, tamanho);

t10 = clock();

cout<<endl<<"Tempo final de execucao da ordenacao por Heapsort: "<<difftime(t10,t9)/CLOCKS\_PER\_SEC<<endl;

cout<<"Total de comparacoes realizadas entre as chaves do vetor declarado: "<<totalComparacoes5<<endl;

cout<<"Total de movimentacoes realizadas entre as chaves do vetor declarado: "<<totalMovimentacoes5<<endl<<endl;

//impressaoVetor(vetor1, tamanho);

}

void fusao(int vetor1[], int tamanho){

clock\_t t11, t12;

t11 = clock();

ordenacaoFusao(vetor1, tamanho);

t12 = clock();

cout<<endl<<"Tempo final de execucao da ordenacao por Fusao: "<<difftime(t12,t11)/CLOCKS\_PER\_SEC<<endl;

cout<<"Total de comparacoes realizadas entre as chaves do vetor declarado: "<<totalComparacoes6<<endl;

cout<<"Total de movimentacoes realizadas entre as chaves do vetor declarado: "<<totalMovimentacoes6<<endl<<endl;

//impressaoVetor(vetor1, tamanho);

}

void quickSort(int vetor1[], int tamanho){

clock\_t t13, t14;

t13 = clock();

aplicacaoOrdenacaoQuickSort(vetor1, 1, tamanho);

t14 = clock();

cout<<endl<<"Tempo final de execucao da ordenacao por QuickSort:"<<difftime(t14,t13)/CLOCKS\_PER\_SEC<<endl;

cout<<"Total de comparacoes realizadas entre as chaves do vetor declarado: "<<totalComparacoes7<<endl;

cout<<"Total de movimentacoes realizadas entre as chaves do vetor declarado: "<<totalMovimentacoes7<<endl<<endl;

//impressaoVetor(vetor1, tamanho);

}

void ordenacaoInsercaoDireta(int vetor1[], int tamanho) { // ordenação por inserção direta

int j; // auxiliar para a busca linear com sentinela.

for (int i = 2; i <= tamanho; i++) {

x = vetor1[i]; // sentinela da busca.

vetor1[0] = x;

totalMovimentacoes2 += 2;

totalComparacoes1++; //já se releva o caso em que x < vetor1[j - 1] é falso.

for (j = i; j >= 0 && x < vetor1[j - 1]; j--){ // busca linear com sentinela.

vetor1[j] = vetor1[j - 1];

totalComparacoes1++;

totalMovimentacoes1++;

}

vetor1[j] = x; // troca de valores do sentinela.

totalMovimentacoes1++;

}

}

void ordenacaoInsercaoBinaria(int vetor1[], int tamanho) { // ordenação por inserção binária.

int j; // representa os índices do vetor final ordenado.

int R; // representa a parte direita do vetor.

int L; // representa a parte direita do vetor.

int m; // representa a parte direita do vetor.

for (int i = 2; i <= tamanho; i++) {

x = vetor1[i]; //atribuição de um "sentinela".

totalMovimentacoes2++;

for (L = 1, R = i; L < R; ) { // busca binária rápida

m = (L + R) / 2;

totalComparacoes2++;

if (vetor1[m] > x){

R = m;

totalComparacoes2++;

}

else{

L = m + 1;

totalComparacoes2++;

}

}

for (j = i; j > R; j--){

vetor1[j] = vetor1[j - 1];

totalMovimentacoes2++;

}

vetor1[R] = x; // troca de valores do "sentinela".

totalMovimentacoes2++;

}

}

void ordenacaoSelecao(int vetor1[], int tamanho) { // ordenação por seleção.

int aux; // indica o índice do menor número encontrado em cada iteração.

// busca do índice do menor valor de cada iteração.

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

aux = i;

for (int j = i + 1; j <= tamanho; j++) {

totalComparacoes3++;

if (vetor1[j] < vetor1[aux]){

aux = j;

totalComparacoes3++;

}

}

// troca efetiva do valor selecionado com o menor valor encontrado.

x = vetor1[i];

vetor1[i] = vetor1[aux];

vetor1[aux] = x;

totalMovimentacoes3 += 3;

}

}

void ordenacaoBubbleSort(int vetor1[], int tamanho) { // ordenação por Bubblesort.

// princípio da "bolha": menor valor, menor densidade, primeiros endereços.

for (int i = 2; i <= tamanho; i++) {

for (int j = tamanho; j >= i; j--) {

totalComparacoes4++;

if (vetor1[j - 1] > vetor1[j]) { // troca efetiva dos valores de menores "densidades" no vetor.

x = vetor1[j - 1];

vetor1[j - 1] = vetor1[j];

vetor1[j] = x;

totalComparacoes4++;

totalMovimentacoes4 += 3;

}

}

}

}

void ordenacaoHeapsort(int vetor1[], int tamanho){ //ordenação por Heapsort.

int aux; // valor auxiliar para a ordenação.

int L; // representa a parte esquerda do vetor.

int R; // representa a parte direita do vetor.

for (L = tamanho/2; L >= 1; L--) //formação do heap, estrutura de árvore binária.

ordenacaoHeapsortHeapify(vetor1, L, tamanho);

for (R = tamanho; R >= 2; R--){ //aplicação da ordenação, a partir do princípio de formação do heap.

//troca de posição do maior valor, para se "atualizar" a nova estrutura heap, desconsiderando o valor reposicionado.

aux = vetor1[1];

vetor1[1] = vetor1[R];

vetor1[R] = aux;

totalMovimentacoes5 += 3;

ordenacaoHeapsortHeapify(vetor1, 1, R - 1); // chamada do algoritmo de ordenação do heap, pelo processo de amontoação (tombamento à esquerda)

}

}

void ordenacaoHeapsortHeapify(int vetor1[], int L, int R){

int aux1 = L; // representa o endereço do nó pai de cada chamada dessa função.

int aux2 = 2\*L; // representa o endereço da folha esquerda de cada chamada dessa função, em relação ao nó pai associado.

x = vetor1[L]; // representa o valor do nó pai de cada iteração subsequente durante a execução da função.

totalMovimentacoes5++;

if ((aux2 < R) && (vetor1[aux2] < vetor1[aux2 + 1])){ //aplicação do princípio de heap máximo.

aux2++;

totalComparacoes5 ++;

}

else{

if (!(vetor1[aux2] < vetor1[aux2 + 1]))

totalComparacoes5++; // já se releva o caso em que uma das comparações é falsa (análise preguiçosa)

}

totalComparacoes5++; // já se releva o caso em que uma das comparações é falsa (análise preguiçosa)

while ((aux2 <= R) && (x < vetor1[aux2])){

vetor1[aux1] = vetor1[aux2];

totalMovimentacoes5++;

aux1 = aux2;

aux2 \*= 2;

totalComparacoes5 ++;

if ((aux2 < R) && (vetor1[aux2] < vetor1[aux2 + 1])){ //aplicação do princípio de heap máximo (processo repetido para cada nó filho associado).

aux2++;

totalComparacoes5 ++;

}

else{

if (!(vetor1[aux2] < vetor1[aux2 + 1]))

totalComparacoes5++; // já se releva o caso em que uma das comparações é falsa (análise preguiçosa)

}

}

vetor1[aux1] = x; // maior valor agora está no endereço do nó pai, obedecendo o princípio do heap máximo.

totalMovimentacoes5++;

}

void ordenacaoFusao(int vetor1[], int tamanho){ // ordenação por Fusão direta.

int particao = 1; // variável auxiliar que delimita o escopo de combinações do vetor a ser ordenado.

int \*A = new int[tamanho + 1];

int \*C = (int\*) calloc(tamanho + 1, sizeof(int)); // vetor auxiliar da fusão inicializado com zero em todos os endereços.

if (C == nullptr) // tratamento de erro, caso ocorra algum problema na execução do calloc.

exit(1);

for (int i = 1; i <= tamanho; i++)

A[i] = vetor1[i];

while (particao < tamanho){ // loop de chamada iterativa do processo de fusão.

mergePass(A, tamanho, particao, C);

particao \*= 2;

mergePass(C, tamanho, particao, A);

particao \*= 2;

}

/\* cópia do vetor ordenado de volta para vetor1 --> A é o vetor ordenado final, pois na última execução de mergePass no while,

o vetor C é argumento para o uso do vetor A.\*/

for (int i = 1; i <= tamanho; i++)

vetor1[i] = A[i];

delete[] A;

delete[] C;

}

void processoFusao(int A[], int esquerda, int meio, int direita, int C[]){ /\*fusão

de vetores "partidos":

A vai da esquerda até meio - 1, entretanto C vai de meio até direita.

\*/

int a = esquerda; // índice inicial do vetor particionado A.

int b = meio + 1; // índice inicial do vetor particionado C.

int k = esquerda - 1; //índice de auxílio para preenchimento do vetor resultante da fusão.

while (a <= meio && b <= direita){ //comparador intra-vetores.

k++;

if (A[a] < A[b]){

C[k] = A[a];

totalMovimentacoes6++;

a++;

}

else{

C[k] = A[b];

totalMovimentacoes6++;

b++;

}

}

while (a <= meio){ /\*preenchimento do restante do vetor final,

caso A.length > C.length.\*/

k++;

C[k] = A[a];

totalMovimentacoes6++;

a++;

}

while (b <= direita){ /\*preenchimento do restante do vetor final,

caso A.length < C.length.\*/

k++;

C[k] = A[b];

totalMovimentacoes6++;

b++;

}

}

void mergePass(int A[], int tamanho, int particao, int C[]){ /\* garante a passage de referência para o processo de fusão, ou seja,

delimita o intervalo a ser fundido e ordenado, considerando os devidos valores de partição a cada chamada do método na ordenação\*/

int aux = 1; // índice para o endereço auxiliar a ser modificado durante a ordenação.

while (aux < tamanho - 2\*particao + 1){ // garantia de que o comprimento considerado é múltiplo de potência de 2.

processoFusao(A, aux, aux + particao - 1, aux + 2\*particao - 1, C);

aux += 2\*particao;

}

if (aux + particao - 1 < tamanho) /\* eventual processo de fusão, caso o valor do meio após o while ainda ser menor do que o tamanho do vetor A

(ou C, dependendo do caso).\*/

processoFusao(A, aux, aux + particao - 1, tamanho, C);

else{

for (int j = aux; j <= tamanho; j++){ /\* passagem efetiva dos valores ordenados de cada chamada de mergePass, considerando o valor de partição

de cada chamada da função mergePass em ordenaçãoFusao\*/

C[j] = A[j];

totalMovimentacoes6++;

}

}

}

void aplicacaoOrdenacaoQuickSort(int vetor1[], int esquerda, int direita){ //ordenação por QuickSort.

int i = esquerda; // delimitação do endereço à esquerda de cada chamada da função.

int j = direita; // delimitação do endereço à direita de cada chamada da função.

int aux; // variável auxiliar para a troca dos valores dos devidos endereços de troca.

x = vetor1[(esquerda + direita)/2]; // variável auxiliar para comparação das chaves do vetor

do{

totalComparacoes7++; // considera-se previamente o caso em que será falso o valor booleano associado.

while (vetor1[i] < x){ // varredura da esquerda para a diretia

totalComparacoes7++;

i++;

}

totalComparacoes7++; // considera-se previamente o caso em que será falso o valor booleano associado.

while (vetor1[j] > x){ // varredura da direita para a esquerda

totalComparacoes7++;

j--;

}

if (i <= j){ // condicional para a troca dos valores encontrados nos endereços indicados após a execução das iterações "while".

aux = vetor1[i];

vetor1[i] = vetor1[j];

vetor1[j] = aux;

totalMovimentacoes7 += 3;

i++;

j--;

}

} while (i <= j); // estrutura a qual evita o caso de loop infinito.

if (i < direita) // acomodação do vetor para a esquerda.

aplicacaoOrdenacaoQuickSort(vetor1, i, direita);

if (esquerda < j) // acomodação do vetor para a direita.

aplicacaoOrdenacaoQuickSort(vetor1, esquerda, j);

}

int main() {

int n;

cout<<"Informe o tamanho do vetor: ";

cin>>n;

cout<<endl;

/\*(Trecho usado para item "b)":

cout << "Informe o tamanho do vetor: "; //Por convenção de teste, a entrada será:

//{45, 56, 12, 43, 95, 19, 8, 67}, ou seja, n = 8.

cin>>n;

cout<<endl;

\*/

/\*(Trecho usado para item "c)" (1):

//Aplicação de valores aleatórios:

int \*vetor1 = new int[n + 1];

srand(time(NULL)); // "Source random"

vetor1[1] = rand()%100;

for (int i = 2; i <= n; i++)

vetor1[i] = vetor1[i - 1] + rand()%3;

\*/

/\*

(Trecho usado para item "c)" (2):

//Aplicação de valores aleatórios:

int \*vetor1 = new int[n + 1];

srand(time(NULL)); // "Source random"

vetor1[1] = rand()%10000;

for (int i = 2; i <= n; i++)

vetor1[i] = vetor1[i - 1] - rand()%3;

\*/

/\*(Trecho usado para item "c)" (3):

//Aplicação de valores aleatórios:

int \*vetor1 = new int[n + 1];

unsigned int seed = time(0); // Uso da biblioteca ctime, para "burlar" a pseudoaleatoriedade dos números da classe Random tradicional.

srand(seed); // "Source random"

int \*vetor1 = new int[n + 1];

for (int i = 1; i <= n; i++)

vetor1[i] = rand()%100; //Aplicação da Random, adaptada para o uso da biblioteca ctime.

\*/

int \*vetor1 = new int[n + 1];

int \*vetor2 = new int[n + 1];

preencherVetor(vetor1, vetor2, n);

insercaoDireta(vetor1, n);

desordenarVetor(vetor1, vetor2, n);

insercaoBinaria(vetor1, n);

desordenarVetor(vetor1, vetor2, n);

selecao(vetor1, n);

desordenarVetor(vetor1, vetor2, n);

bubbleSort(vetor1, n);

desordenarVetor(vetor1, vetor2, n);

heapsort(vetor1, n);

desordenarVetor(vetor1, vetor2, n);

fusao(vetor1, n);

desordenarVetor(vetor1, vetor2, n);

quickSort(vetor1, n);

delete[] vetor1;

delete[] vetor2;

return 0;

}