**UNIVERSIDADE FEDERAL DO CEARÁ - CAMPUS CRATEUS**

**CURSO: CIÊNCIA DA COMPUTAÇÃO**

**DISCIPLINA: ESTRUTURA DE DADOS (2022.2)**

**PROF. ARNALDO BARRETO VILA NOVA**



##### **ALUNO:** LUCAS EDUARDO MOTA

**AVALIAÇÃO 03**

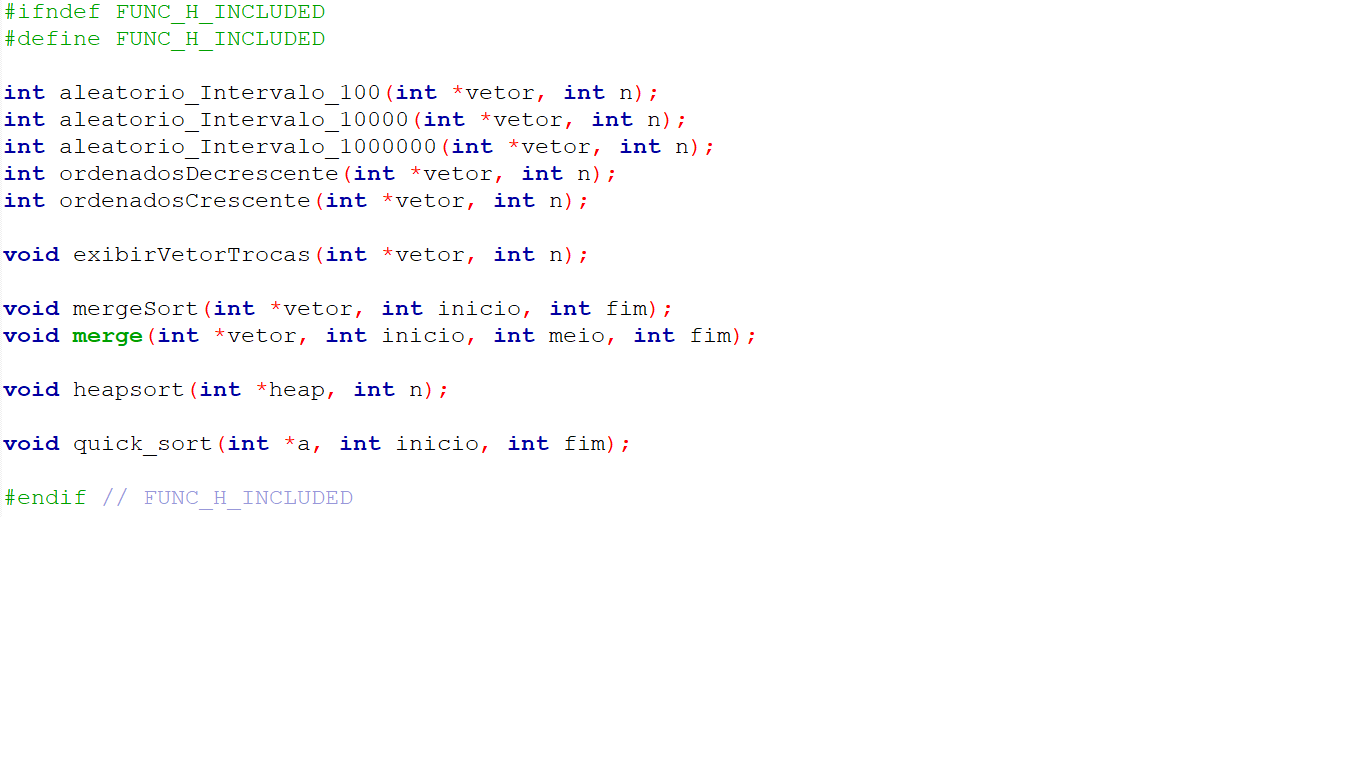
Comparação entre métodos de ordenação

#### Crateús – CE, 2022

**Código completo:**

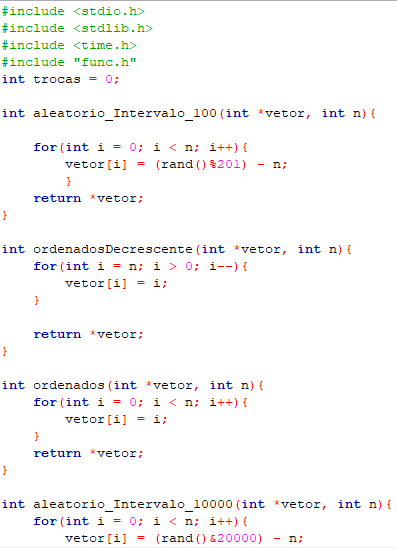
**FUNC.h**

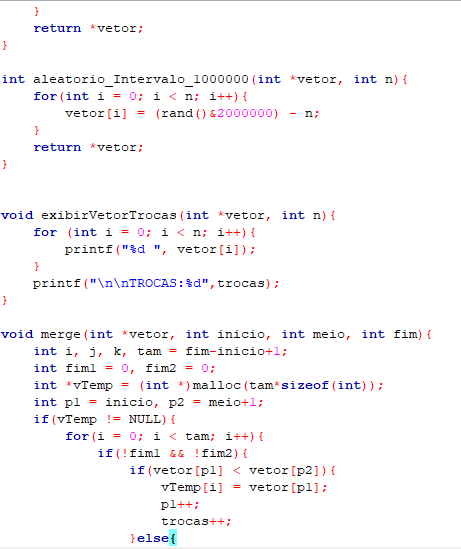
**FOTO:**

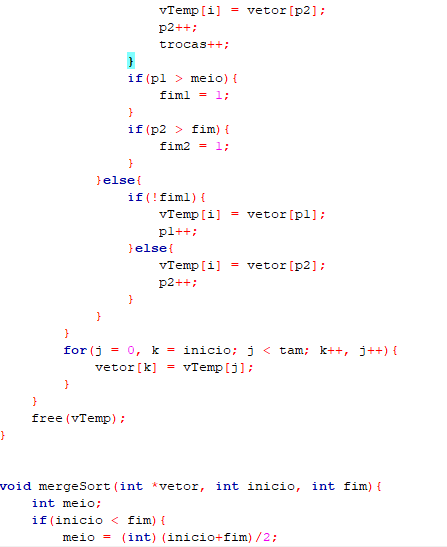
****

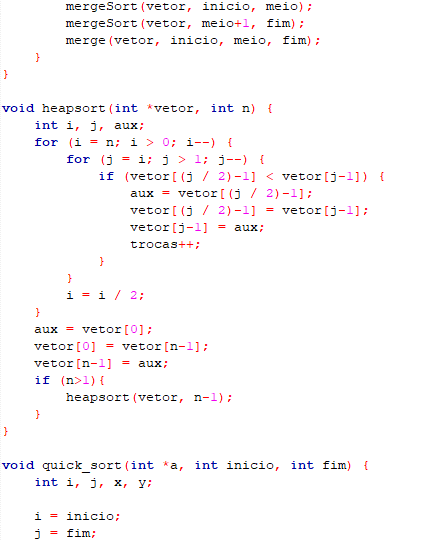
**FUNC.c**

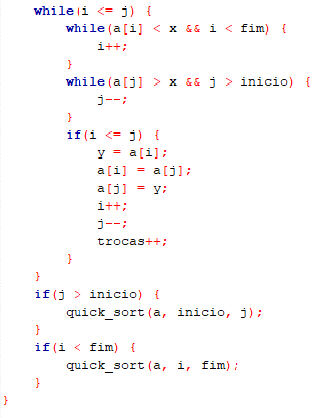
**FOTOS DO CÓDIGO:**

****

****

****

****

****

**Código Escrito:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "func.h"

int trocas = 0;

int aleatorio\_Intervalo\_100(int \*vetor, int n){

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

vetor[i] = (rand()%201) - n;

}

return \*vetor;

}

int ordenadosDecrescente(int \*vetor, int n){

for(int i = n; i > 0; i--){

vetor[i] = i;

}

return \*vetor;

}

int ordenados(int \*vetor, int n){

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

vetor[i] = i;

}

return \*vetor;

}

int aleatorio\_Intervalo\_10000(int \*vetor, int n){

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

vetor[i] = (rand()&20000) - n;

}

return \*vetor;

}

int aleatorio\_Intervalo\_1000000(int \*vetor, int n){

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

vetor[i] = (rand()&2000000) - n;

}

return \*vetor;

}

void exibirVetorTrocas(int \*vetor, int n){

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

printf("%d ", vetor[i]);

}

printf("\n\nTROCAS:%d",trocas);

}

void merge(int \*vetor, int inicio, int meio, int fim){

int i, j, k, tam = fim-inicio+1;

int fim1 = 0, fim2 = 0;

int \*vTemp = (int \*)malloc(tam\*sizeof(int));

int p1 = inicio, p2 = meio+1;

if(vTemp != NULL){

for(i = 0; i < tam; i++){

if(!fim1 && !fim2){

if(vetor[p1] < vetor[p2]){

vTemp[i] = vetor[p1];

p1++;

trocas++;

}else{

vTemp[i] = vetor[p2];

p2++;

trocas++;

}

if(p1 > meio){

fim1 = 1;

}

if(p2 > fim){

fim2 = 1;

}

}else{

if(!fim1){

vTemp[i] = vetor[p1];

p1++;

}else{

vTemp[i] = vetor[p2];

p2++;

}

}

}

for(j = 0, k = inicio; j < tam; k++, j++){

vetor[k] = vTemp[j];

}

}

free(vTemp);

}

void mergeSort(int \*vetor, int inicio, int fim){

int meio;

if(inicio < fim){

meio = (int)(inicio+fim)/2;

mergeSort(vetor, inicio, meio);

mergeSort(vetor, meio+1, fim);

merge(vetor, inicio, meio, fim);

}

}

void heapsort(int \*vetor, int n) {

int i, j, aux;

for (i = n; i > 0; i--) {

for (j = i; j > 1; j--) {

if (vetor[(j / 2)-1] < vetor[j-1]) {

aux = vetor[(j / 2)-1];

vetor[(j / 2)-1] = vetor[j-1];

vetor[j-1] = aux;

trocas++;

}

}

i = i / 2;

}

aux = vetor[0];

vetor[0] = vetor[n-1];

vetor[n-1] = aux;

if (n>1){

heapsort(vetor, n-1);

}

}

void quick\_sort(int \*a, int inicio, int fim) {

int i, j, x, y;

i = inicio;

j = fim;

x = a[(inicio + fim) / 2];

while(i <= j) {

while(a[i] < x && i < fim) {

i++;

}

while(a[j] > x && j > inicio) {

j--;

}

if(i <= j) {

y = a[i];

a[i] = a[j];

a[j] = y;

i++;

j--;

trocas++;

}

}

if(j > inicio) {

quick\_sort(a, inicio, j);

}

if(i < fim) {

quick\_sort(a, i, fim);

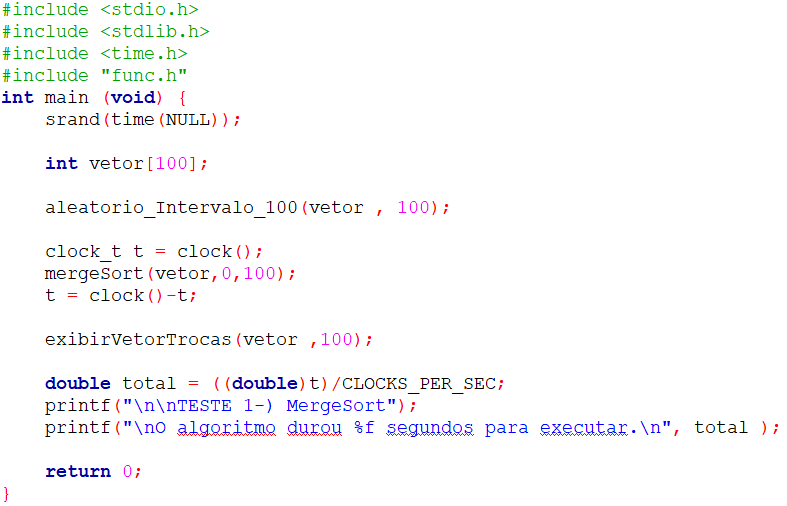
}

}

**TESTES:**

1. **Ordenação de um conjunto com 100 valores inteiros aleatórios no intervalo de -100 a 100:**

**Exemplo:**

****

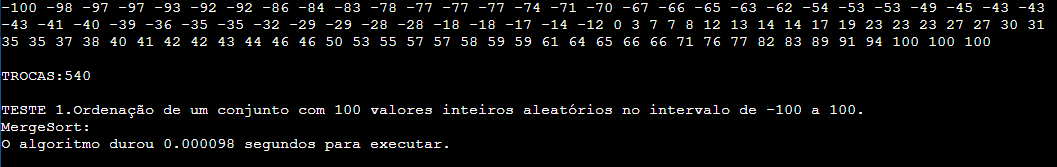
**Todos os testes a seguir foram feitos com essa mesma forma, mudando apenas o nome da função chamada, seus parâmetros e o tamanho do vetor.**

**1.Ordenação de um conjunto com 100 valores inteiros aleatórios no intervalo de -100 a 100:**

**MergeSort:**

Tempo: 0,000098 segundos

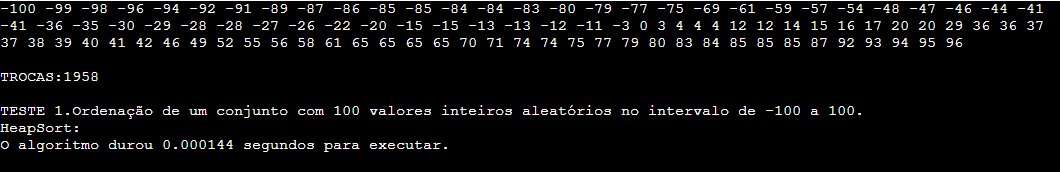
Trocas: 540



**HeapSort:**

Tempo: 0,000144 segundos

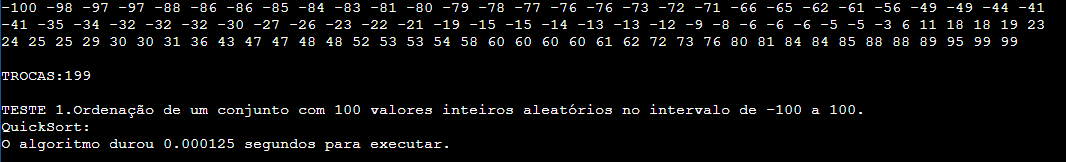
Trocas: 1958



**QuickSort:**

Tempo: 0,000125 segundos

Trocas: 199

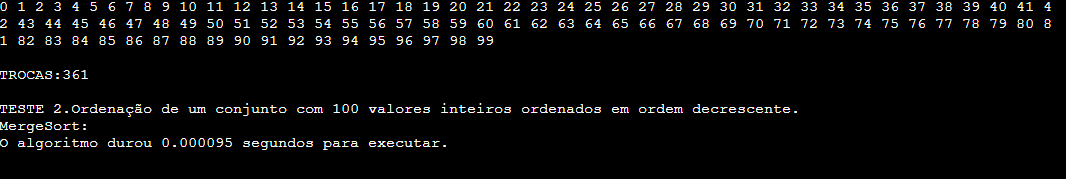


**2. Ordenação de um conjunto com 100 valores inteiros ordenados em ordem decrescente:**

**MergeSort:**

Tempo: 0,000095 segundos

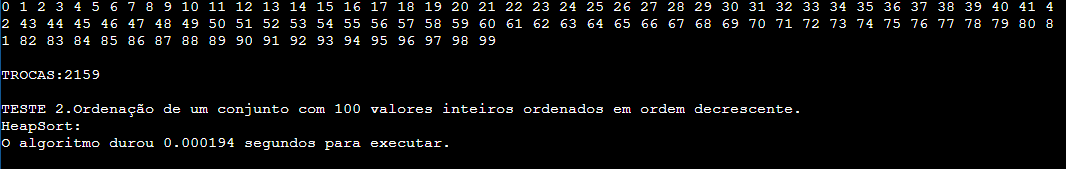
Trocas: 361



**HeapSort:**

Tempo: 0,000194 segundos

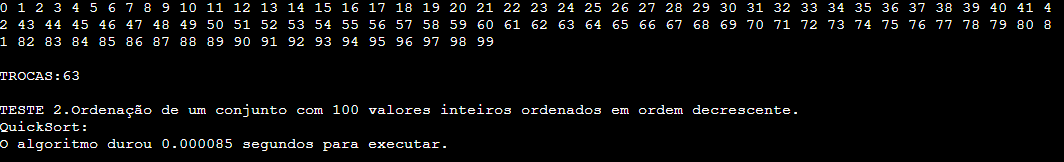
Trocas: 2159



**QuickSort:**

Tempo: 0,000085 segundos

Trocas: 63

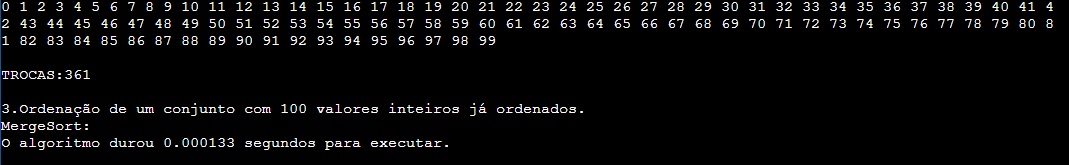


**3. Ordenação de um conjunto com 100 valores inteiros já ordenados:**

**MergeSort:**

Tempo: 0,000133 segundos

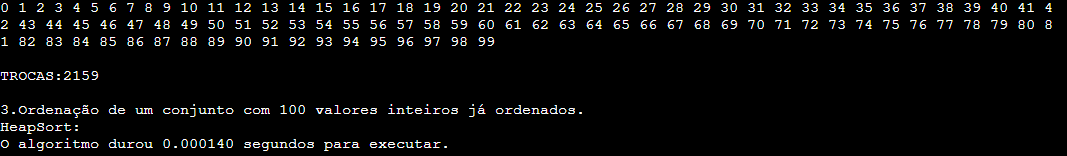
Trocas: 361



**HeapSort:**

Tempo: 0,000140 segundos

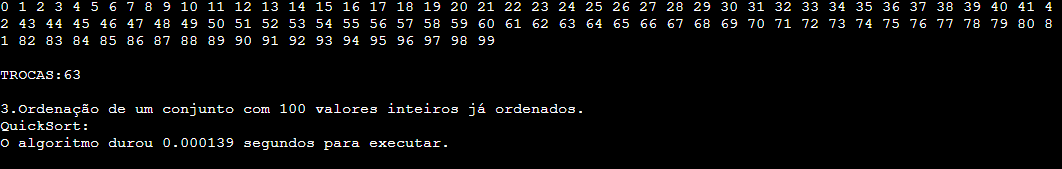
Trocas: 2159



**QuickSort:**

Tempo: 0,000139 segundos

Trocas: 63

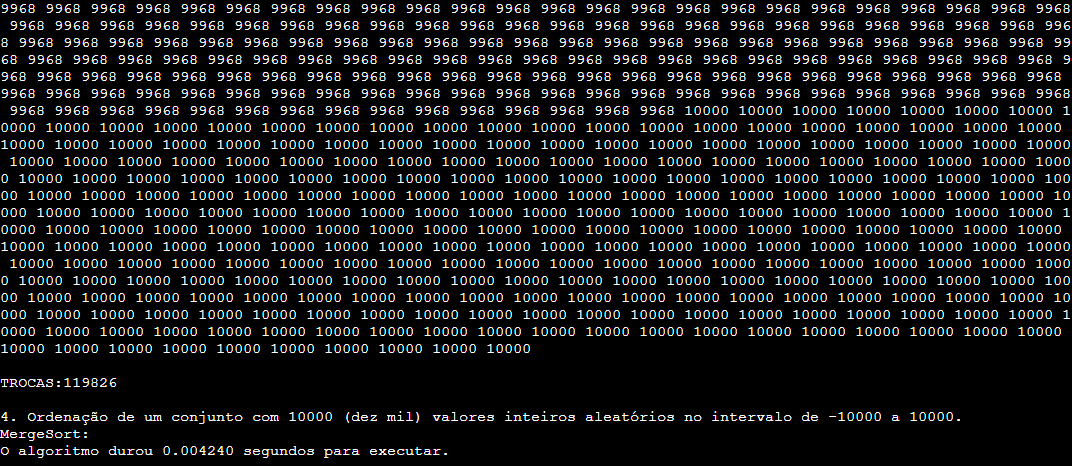


**4. Ordenação de um conjunto com 10000 (dez mil) valores inteiros aleatórios no intervalo de -10000 a 10000:**

**MergeSort:**

Tempo: 0,004240 segundos

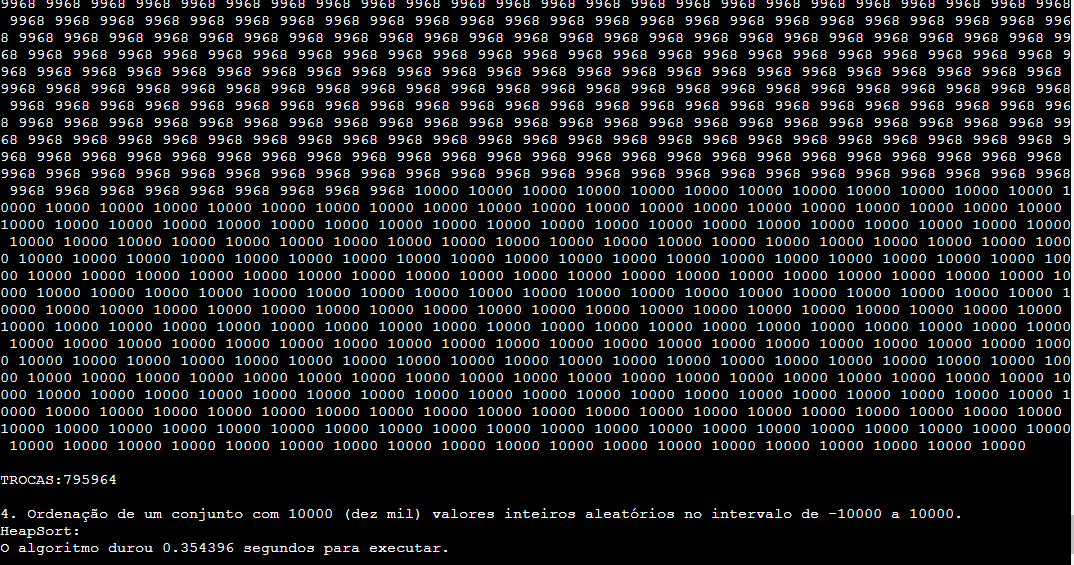
Trocas: 119826



**HeapSort:**

Tempo: 0,354396 segundos

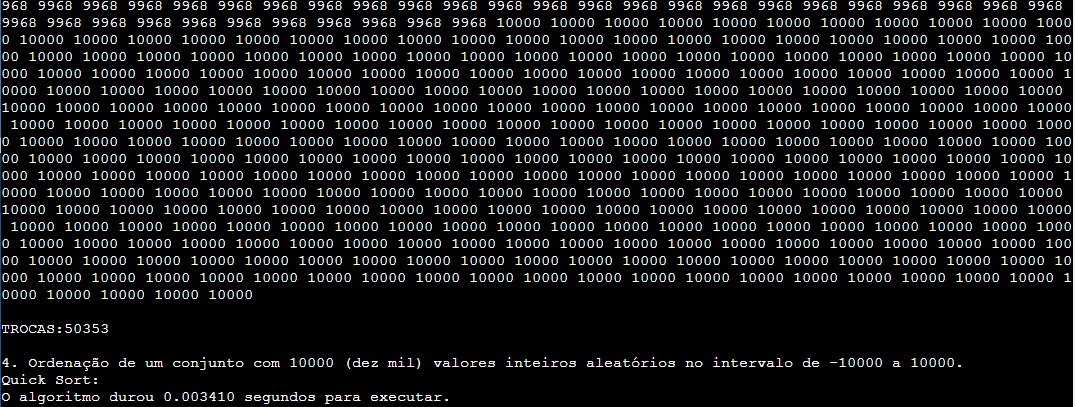
Trocas: 795964



**QuickSort:**

Tempo: 0,003410 segundos

Trocas: 50353

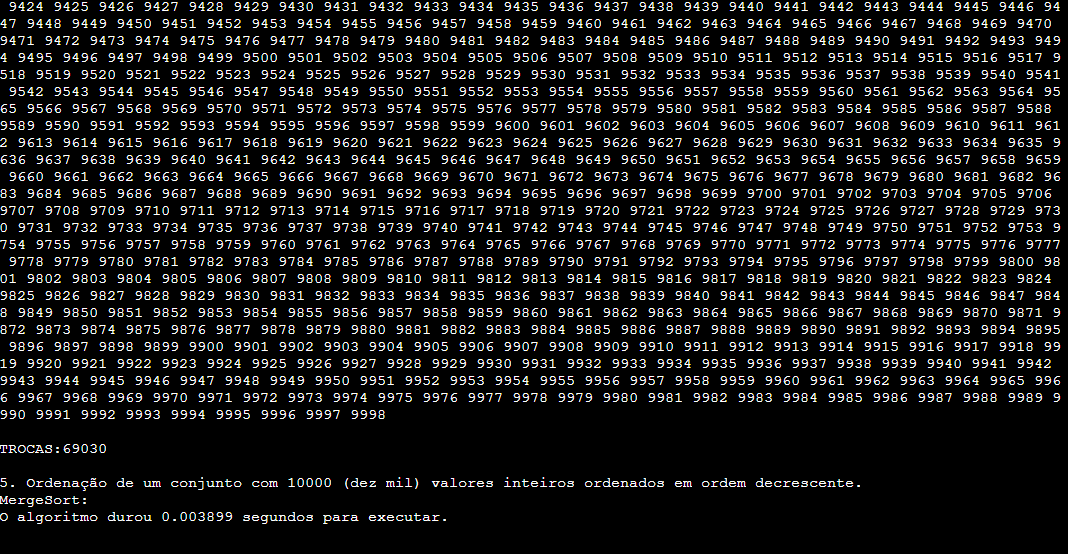


**5. Ordenação de um conjunto com 10000 (dez mil) valores inteiros ordenados em ordem decrescente:**

**MergeSort:**

Tempo: 0,003899 segundos

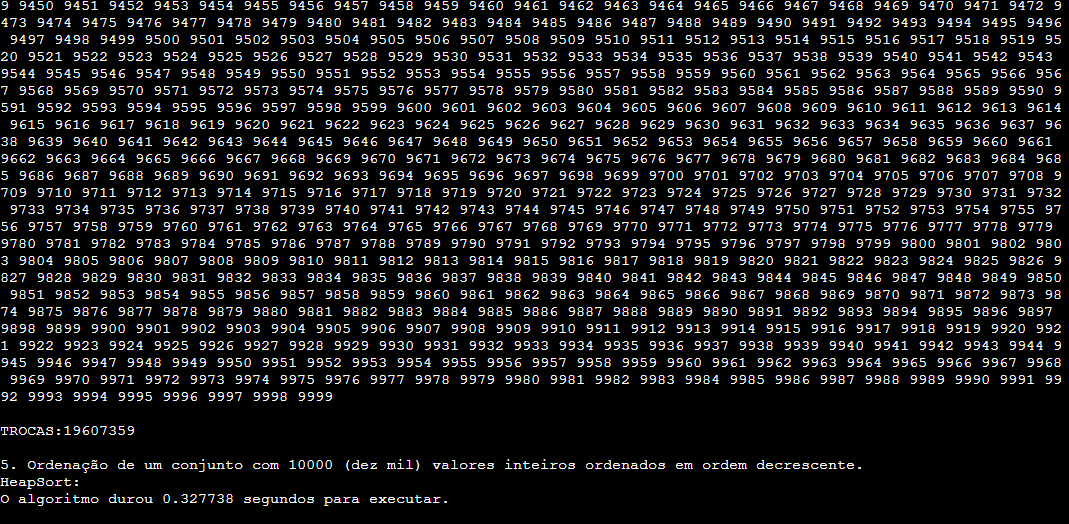
Trocas: 69030



**HeapSort:**

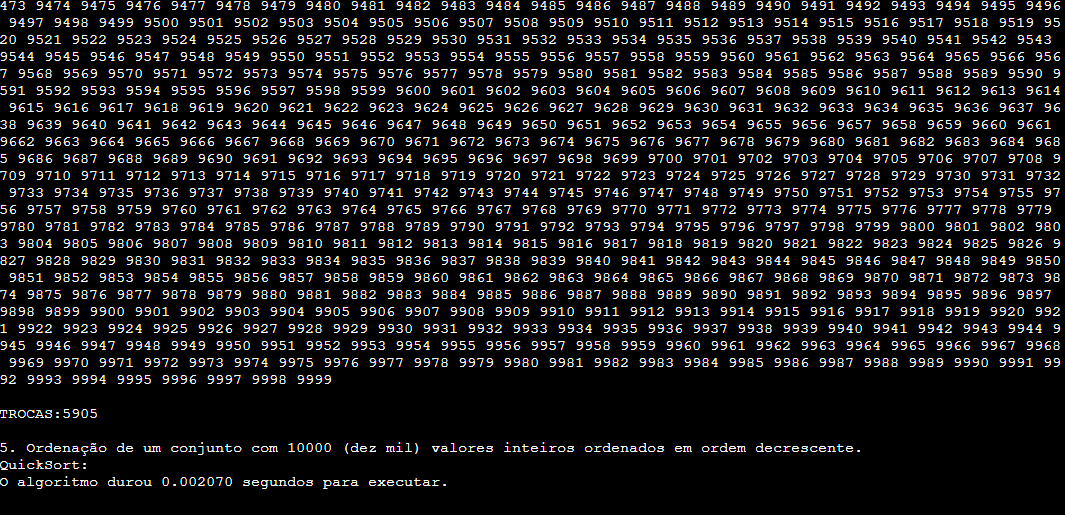
Tempo: 0,327738 segundos

Trocas: 19607359



**QuickSort:**

Tempo: 0,002070 segundos

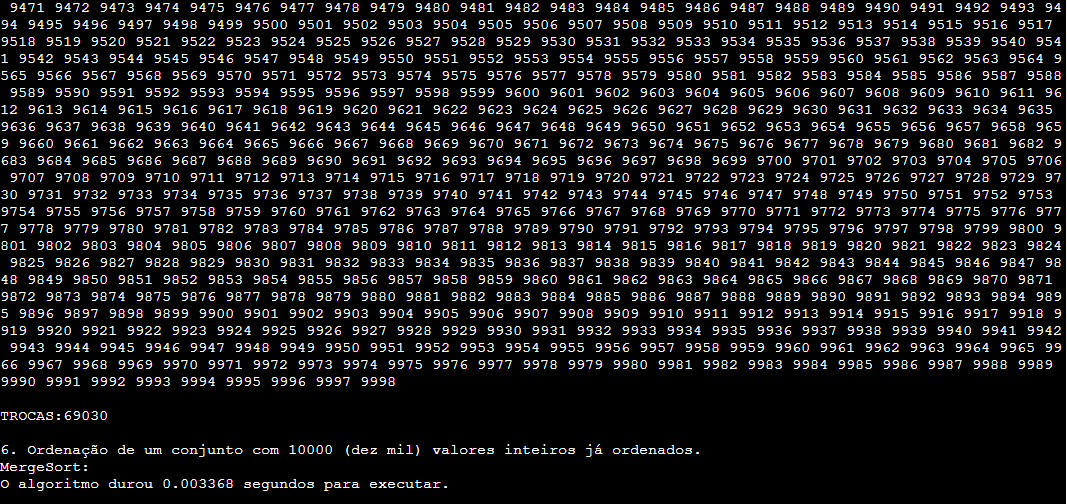
Trocas: 5905

**6. Ordenação de um conjunto com 10000 (dez mil) valores inteiros já ordenados:**

**MergeSort:**

Tempo: 0,003368 segundos

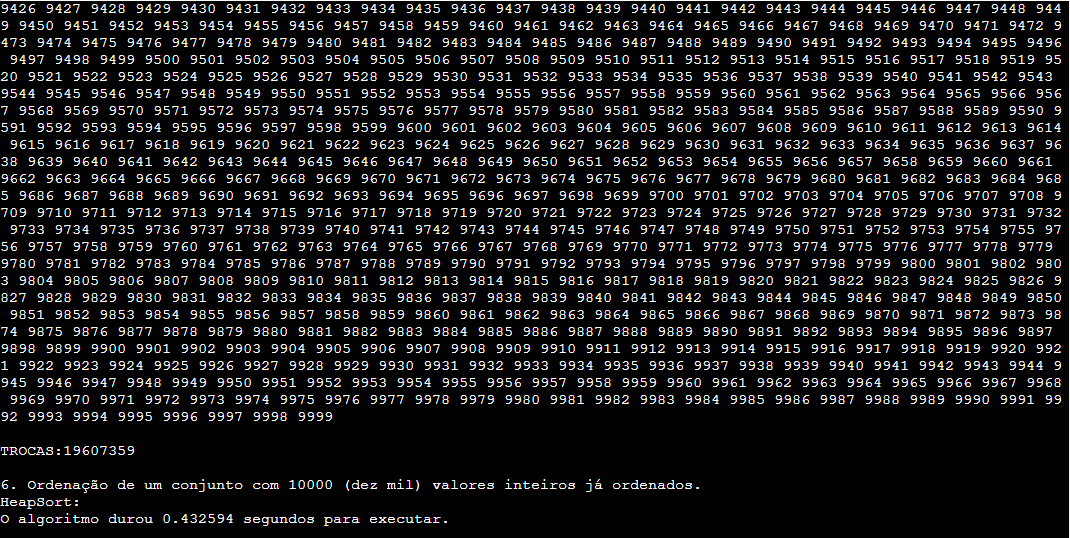
Trocas: 69030



**HeapSort:**

Tempo: 0,432594 segundos

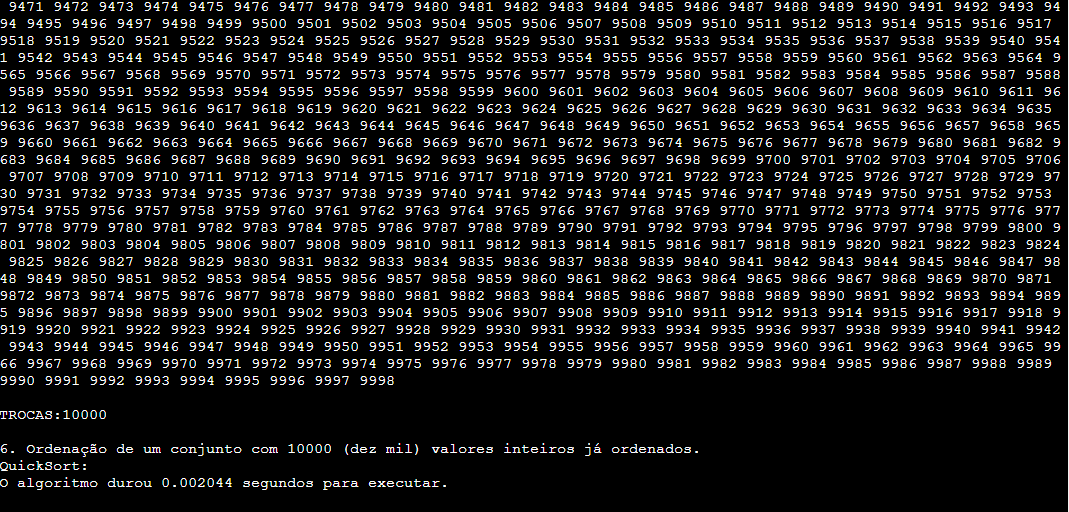
Trocas: 19607359



**QuickSort:**

Tempo: 0,002044 segundos

Trocas: 10000

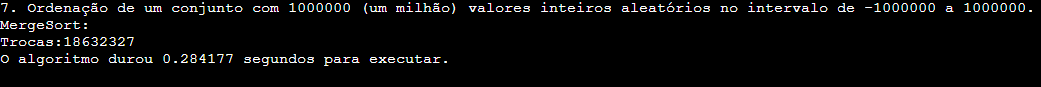


**7. Ordenação de um conjunto com 1000000 (um milhão) valores inteiros aleatórios no intervalo de -1000000 a 1000000:**

**MergeSort:**

Tempo: 0,284177 segundos

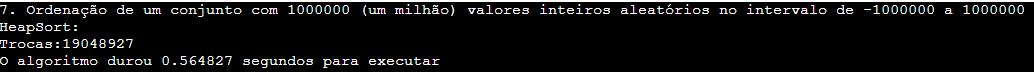
Trocas: 18632327



**HeapSort:**

Tempo: 0,564827 segundos

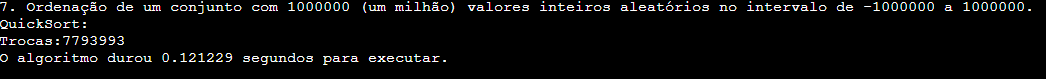
Trocas: 1904827



**QuickSort:**

Tempo: 0,121229 segundos

Trocas: 7793993

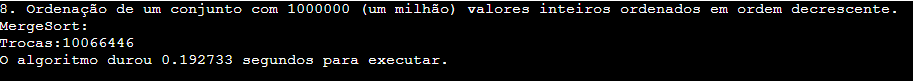


**8. Ordenação de um conjunto com 1000000 (um milhão) valores inteiros ordenados em ordem decrescente:**

**MergeSort:**

Tempo: 0,192733 segundos

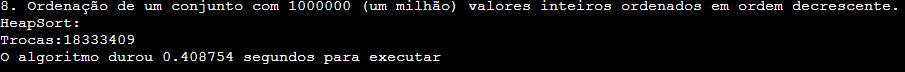
Trocas: 10066446



**HeapSort:**

Tempo: 0,408754 segundos

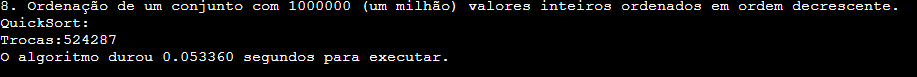
Trocas: 18333409



**QuickSort:**

Tempo: 0,053360 segundos

Trocas: 524287

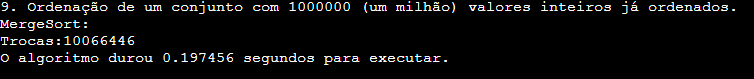


**9. Ordenação de um conjunto com 1000000 (um milhão) valores inteiros já ordenados:**

**MergeSort:**

Tempo: 0,197456 segundos

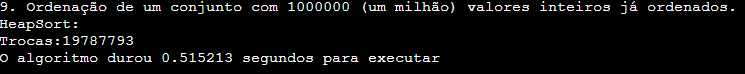
Trocas: 10066446



**HeapSort:**

Tempo: 0,515213 segundos

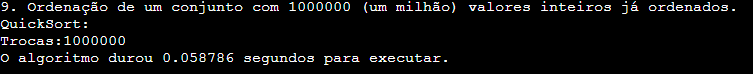
Trocas: 19787793



**QuickSort:**

Tempo: 0,058786 segundos

Trocas: 1000000



**Conclusão:**

Após analisar as informações de todos os testes, temos as seguintes informações refente as trocas realizadas:

**Total de Trocas Realizadas:**

**- Mergesort:** 28.957.921 (vinte e oito milhões novecentos e cinquenta e sete mil novecentos e vinte e um) de trocas.

**- Heapsort:** 80.042.987 (oitenta milhões quarenta e dois mil novecentos e oitenta e sete) de trocas.

- **Quicksort**: 9.384.863 (nove milhões trezentos e oitenta e quatro mil oitocentos e sessenta e três) de trocas.

**Média de trocas para cada algoritmo:**

**- Mergesort:** 3.217.547 (três milhões duzentos e dezessete mil quinhentos e quarenta e sete).

- **Heapsort:** 8.893.665 (oito milhões oitocentos e noventa e três mil seiscentos e sessenta e cinco).

- **Quicksort**: 1.042.761 (um milhão quarenta e dois mil setecentos e sessenta e um).

Com isso temos que o algoritmo que realizou o maior número de trocas foi o Heapsort, seguido pelo Mergesort e o Quicksort.

Seguindo agora para o tempo que cada algoritmo levou para executar suas ordenações:

**Total de Tempo Utilizado:**

**- Mergesort:** 0,686199 segundos totais que foram levados para compilar todos os testes.

**- Heapsort:** 2,604 segundos totais que foram levados para compilar todos os testes.

- **Quicksort**: 0,241248 totais que foram levados para compilar todos os testes.

**Média de tempo levado por cada algoritmo:**

**- Mergesort:** 0,07624433 segundos.

- **Heapsort:** 0,28933 segundos.

- **Quicksort**: 0,02680533 segundos.

Com isso temos que o algoritmo que realizou as ordenações em um maior tempo foi o Heapsort, seguido pelo Mergesort e pelo Quicksort.

Concluindo temos que o algoritmo de ordenação Heapsort foi o que além de levar mais tempo efetuou um maior número de trocas, durante todo o processo de testagem. Seguido pelo algoritmo Mergesort e pelo Quicksort.

Temos então que o algoritmo de ordenação Quicksort, dos três têm resultados mais chamativos, sendo assim tornando-o na ordenação o mais eficiente.