Algoritmos e Estruturas de Dados ||

Marcella Netto de Oliveira

Marcelo Ari de Oliveira

Universidade Federal de São João Del Rei – MG

23 de maio de 2019

Documentação – Trabalho prático

Algoritmos e Estruturas de Dados (2)

Introdução

“Algoritmo de ordenação”, segundo o seu significado em Ciência da computação, é um algoritmo que efetua a ordenação completa ou parcial de determinados elementos em uma sequência, organizando-os de forma crescente ou decrescente, em ordem numérica ou lexicográfica.

Dessa forma, se torna algo de extrema utilidade em projetos de programação, visto que tem o objetivo de facilitar as buscas e pesquisas de ocorrências de determinado elemento em um conjunto ordenado, e assim, acessar dados de forma eficiente.

O objetivo do trabalho proposto é a familiarização com os códigos de cada um dos algoritmos de ordenação avaliando o tempo de execução em algumas situações, as comparações e as movimentações.

Com isso, tendo em vista as regras apresentadas pelo trabalho, a realização da implementação e a análise do código se torna uma forma intuitiva de aprendizado.

Implementação:

Iniciamos o trabalho realizando um estudo dos principais algoritmos de ordenação interna e como eles são classificados.

É importante ressaltar, que os algoritmos classificados por uma ordenação interna são aqueles que todos os elementos a serem ordenados cabem dentro da memória principal e qualquer registro pode ser imediatamente acessado.

Além disso, os métodos de ordenação interna podem ser classificados como:

Simples, pois são adequados para pequenos arquivos, requerem $O(n2)$ comparações, e produzem programas pequenos.

Eficientes, pois são adequados para arquivos maiores, requerem $O(n \log n)$ comparações, essas que são mais complexas em detalhes.

TADs implementados:

Segundo as regras do trabalho, foi necessário testar os tempos de execução dos algoritmos de ordenação em diferentes situações, considerando vetores de tamanhos 20, 500, 5.000, 10.000 e 200.000 elementos, com duas possibilidades de registros.

Dessa forma, para implementar o que foi solicitado, criamos duas estruturas (structs) visto que essas nos permitem trabalhar com diversos tipos de informações de maneira rápida e organizada, e são muito utilizadas em programas que necessitam e fazem o uso de vários tipos de variáveis e características, além de possuírem auto referência (uso de ponteiros).

A primeira Struct foi criada com o objetivo de simular os registros pequenos e contém uma chave inteira. Já a segunda Struct foi criada para simular os registros grandes e contém, além da chave inteira, 50 campos com 50 strings de 50 caracteres.

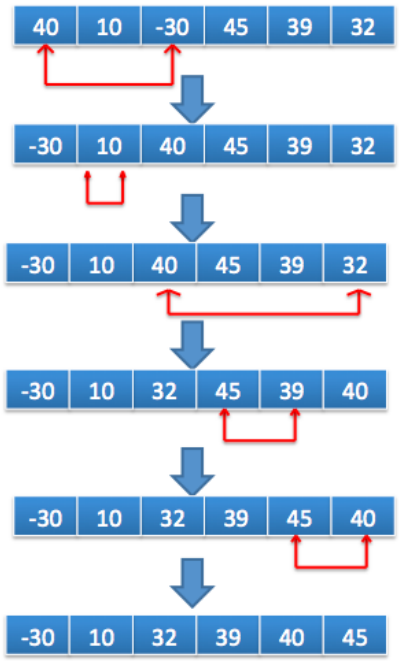
(isso torna a movimentação dos arquivos um pouco mais lenta).

Funções e Procedimentos

O código possui as seguintes funções:

void seleção(int *item , int count){ Função que utiliza o algoritmo de ordenação por seleção, que consiste em selecionar o menor item e colocar na primeira posição, selecionar o segundo menor item e colocar na segunda posição, seguindo esses passos até que reste apenas um elemento.

Segue, anexo exemplo para apresentar o funcionamento de um SelectionSort:



É importante destacar que não é um algoritmo estável, isso significa que ele realiza a troca da ordem relativa dos itens com chave de mesmo tamanho.

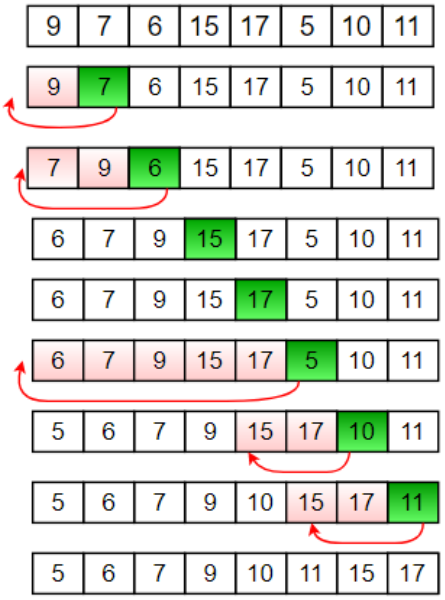
A **complexidade** da função é $O(n2)$.

C(n) = O(n²)

void insercao(int *item, int count){ Função que utiliza o algoritmo de ordenação por inserção, que consiste em percorrer um vetor de elementos da esquerda para a direita e à medida que avança vai ordenando os elementos à esquerda.

Em cada passo, a partir do segundo elemento seleciona o próximo item da sequência e o coloca no local apropriado de acordo com o critério de ordenação.

Segue, anexo exemplo, para apresentar o funcionamento de um InsertionSort:



É importante destacar que é um algoritmo estável, isso significa que ele não realiza a troca da ordem relativa dos itens com chave de mesmo tamanho.

A **complexidade** da função é $O(n2)$.

C(n) = O(n²)

void shell (int *item , int count){ Função que utiliza o algoritmo de ordenação ShellSort que consiste em, basicamente um tipo de inserção, porém, separando em grupos menores e os ordenando, para no fim, ordenar todo o vetor.

É importante ressaltar, que a ordenação Shell utiliza a quebra sucessiva da sequência a ser ordenada e implementa a ordenação por inserção na sequência obtida.

Segue, anexo exemplo, para apresentar o funcionamento do ShellSort:

É importante destacar que não é um algoritmo estável, isso significa que ele realiza a troca da ordem relativa dos itens com chave de mesmo tamanho.

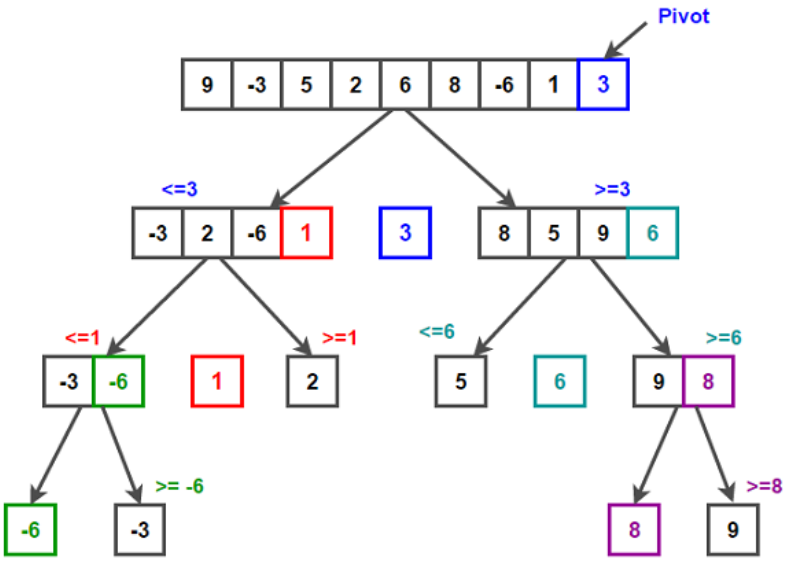
A complexidade do algoritmo é desconhecida, pois ninguém ainda foi capaz de encontrar uma *fórmula fechada* para sua função de complexidade.

void quicksort(int *item , int count){ Função que utiliza o método de ordenação QuickSort que consiste em dividir o problema de ordenar um conjunto com n itens em dois problemas menores. Dessa forma, os problemas menores são ordenados de forma independente e os resultados são combinados para produzir a solução final.

void qs(int *item , int left , int right){

Divide a lista de entrada em duas sub-listas a partir de um pivô, em seguida realiza o mesmo procedimento nas duas listas menores até sobrar apenas uma lista unitária.

Segue, anexo exemplo, para apresentar o funcionamento do QuickSort:



É importante destacar que não é um algoritmo estável, isso significa que ele realiza a troca da ordem relativa dos itens com chave de mesmo tamanho.

A **complexidade** da função é $O(n \log n)$.

void constroiHeap(int *item, int count, int indice_raiz){ Função que tem como objetivo a construção da “Heap”.

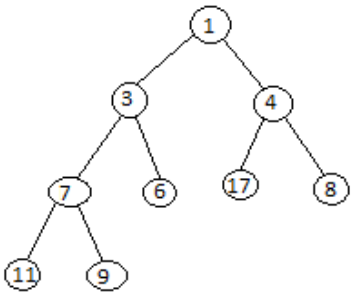
void HeapSort(int *item, int count){ Função que utiliza o método de ordenação HeapSort que enxerga o vetor como uma árvores binária, um vetor em que o valor de todo pai é maior ou igual ao valor de cada um de seus dois filhos.

É importante ressaltar que o algoritmo tem duas fases: a primeira transforma o vetor em heap e a segunda rearranja o heap em ordem.

É importante destacar que não é um algoritmo estável, isso significa que ele realiza a troca da ordem relativa dos itens com chave de mesmo tamanho.

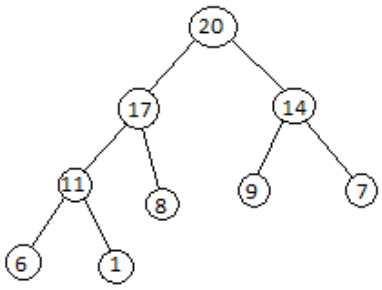
A **complexidade** da função é $O(n \log n)$.

Segue, anexo exemplo, para apresentar a diferença entre “Min-Heap” e “Max-Heap”:



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

Tradução:

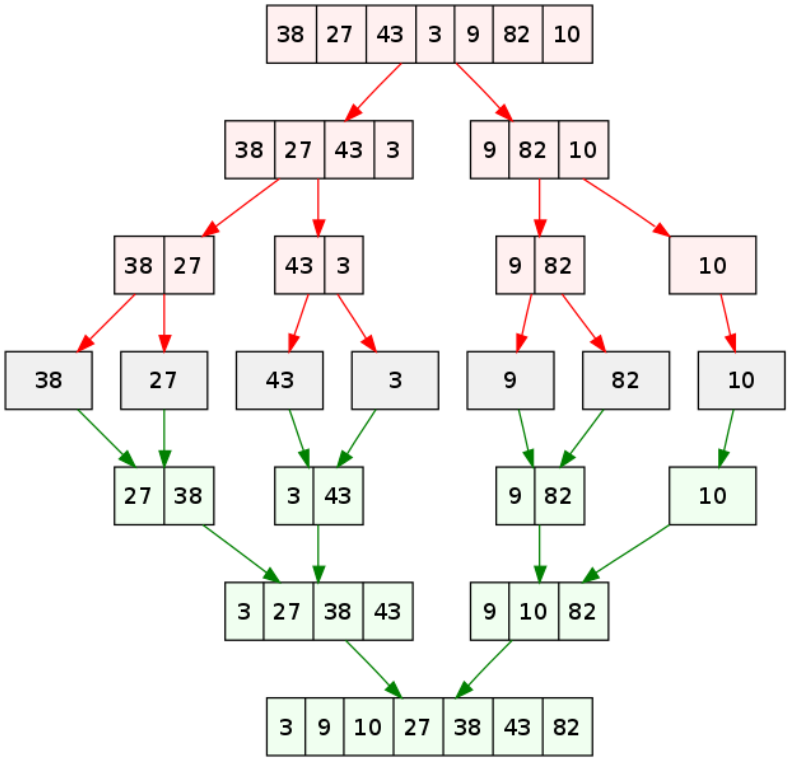
Min-Heap: “No min-heap, o primeiro elemento é o menor. Então quando nós precisamos classificar uma lista em ordem crescente, criamos um Min-heap a partir desta lista e pegamos o primeiro elemento, desde que ele seja o menor. Assim, nós repetimos o processo com os elementos restantes.”

Max-Heap:”No max-heap”, o primeiro elemento é o maior, consequentemente será usado quando é necessário classificar uma lista em ordem decrescente.”

void mergeSort(int *vetor, int posicaoInicio, int posicaoFim){ Função que utiliza o método de ordenação MergeSort que consiste em dividir o problema em pedaços menores e depois juntar (merge) os resultados obtidos. Ovetor é divido em duas partes iguais, que serão, cada uma, divididas em duaspartes, e assim até sobrar um ou dois elementos cuja ordenação é trivial.

Para juntar as partes ordenadas os dois elementos de cada parte são separados e o menor deles é selecionado e retirado de sua parte. Em seguida, os menores entre os restantes são comparados e assim prossegue até juntar todas as partes.

Segue, anexo exemplo, para apresentar o funcionando do MergeSort:



É importante destacar que não é um algoritmo estável, isso significa que ele realiza a troca da ordem relativa dos itens com chave de mesmo tamanho.

A **complexidade** da função é $O(n \log n)$.

O código também possui as seguintes funções incluídas nos arquivos de REGISTRO GRANDE e REGISTRO PEQUENO:

void selecaogran(struct Registrogrande *item, int count);

void insercaogran(struct Registrogrande *item, int count);

void quicksortgran(struct Registrogrande *item , int count);

void qs(struct Registrogrande *item , int left , int right);

void constroiHeap(struct Registrogrande *item, int count, int indice_raiz);

void HeapSortgran(struct Registrogrande *item, int count);

void mergeSortgran(struct Registrogrande *vetor, int posicaoInicio, int posicaoFim);

void selecaogran(struct Registrogrande *item, int count).

void selecaopeq(struct RegistroPequeno *item, int count);

void insercaopeq(struct RegistroPequeno *item, int count);

void quicksortpeq(struct RegistroPequeno *item , int count);

void qs(struct RegistroPequeno *item , int left , int right);

void mergeSort(struct RegistroPequeno *vetor, int posicaoInicio, int posicaoFim);

void selecaopeq(struct RegistroPequeno *item, int count).

Ao analisar essas funções, é notório que o que as difere das funções comentadas inicialmente são os parâmetros de entrada, que possuem um ponteiro para o registro grande e outro para o registro pequeno, e também as variáveis auxiliares que possuem o mesmo tipo do ponteiro para a movimentação dos dados.

Programa Principal:

O trabalho conta com 3 funções principais que estão em arquivos separados.

Arquivo vetor.c:

int tam, m;

int *vet, n, opcao, i, aux, cont;

srand(time(NULL));

double ti, tf, tempo;

ti = tf = tempo = 0;

struct timeval tempo_inicio, tempo_fim;

Onde são declaradas as variáveis e contadores necessários para o funcionamento completo do programa. Inclusive um ponteiro para um vetor de inteiros e as variáveis utilizadas na contagem de tempo de execução dos algoritmos inicializadas com zero.

Atribuímos valores as variáveis de acordo com a opção do usuário, como, por exemplo, qual algoritmo de ordenação deve ser utilizado, qual o tamanho do vetor a ser ordenado e qual a ordem inicial do vetor.

É importante ressaltar que o vetor tem seu espaço alocado de forma dinâmica de acordo com o tamanho de vetor desejado pelo usuário.

Logo após os procedimentos é testada a ordenação inicial solicitada pelo usuário e ordena o vetor de forma aleatória, crescente ou decrescente.

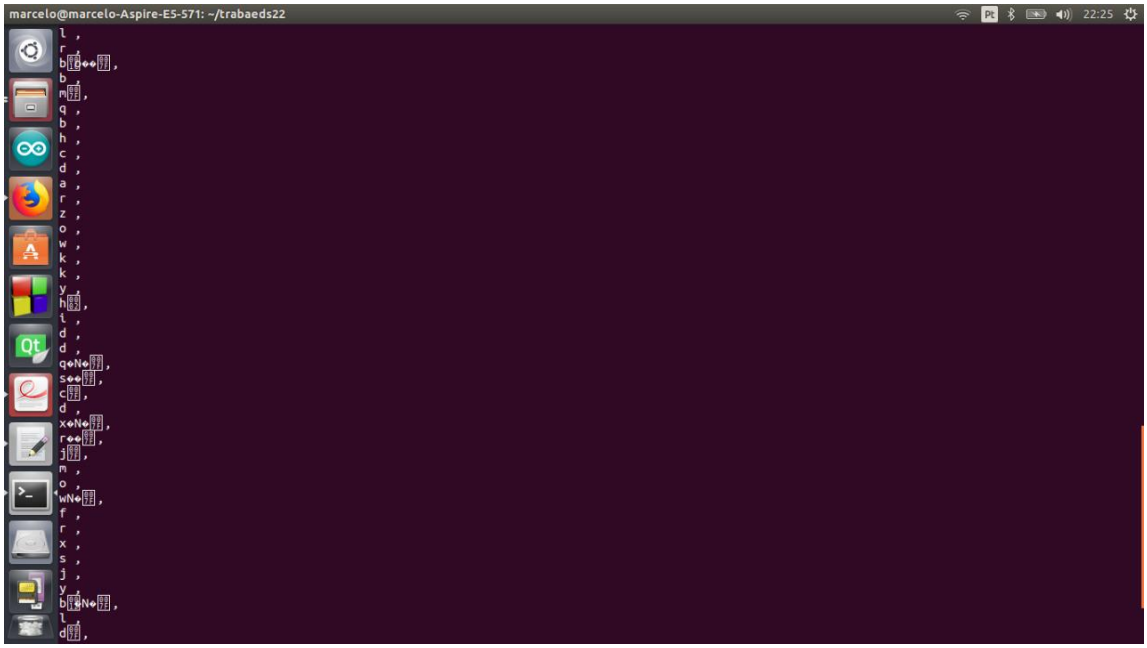
Há também uma estrutura (switch case) em que ocorre o teste de qual algoritmo será utilizado.

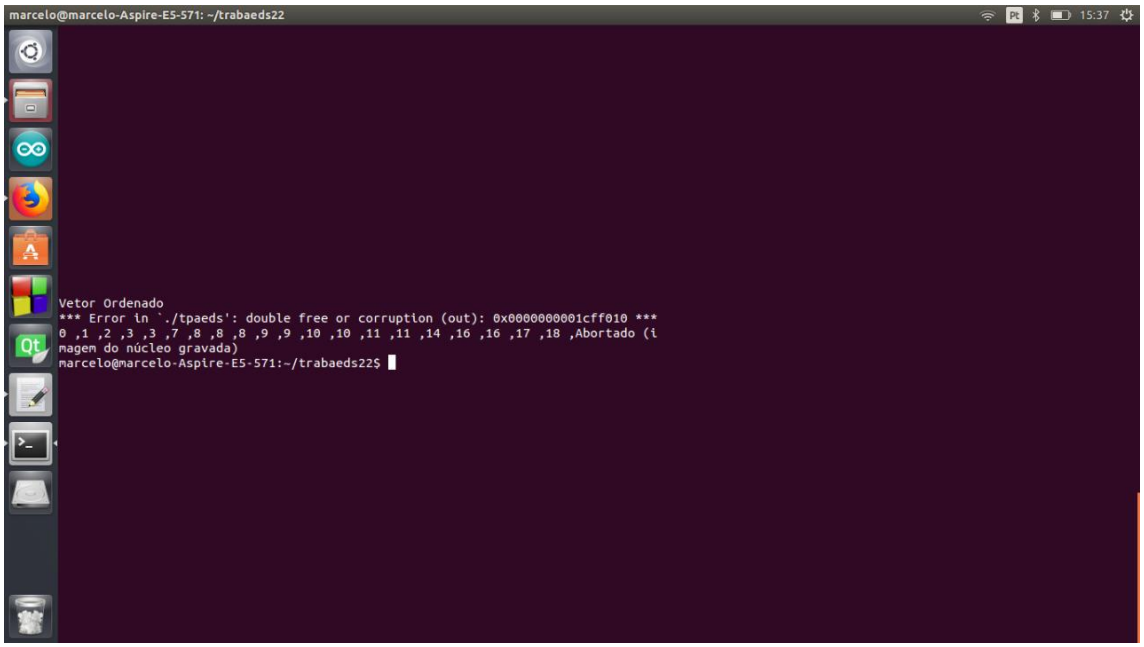
Para a análise de tempo de execução do algoritmo foi utilizada a função “gettimeofday” contida na biblioteca **<sys/time.h>.**

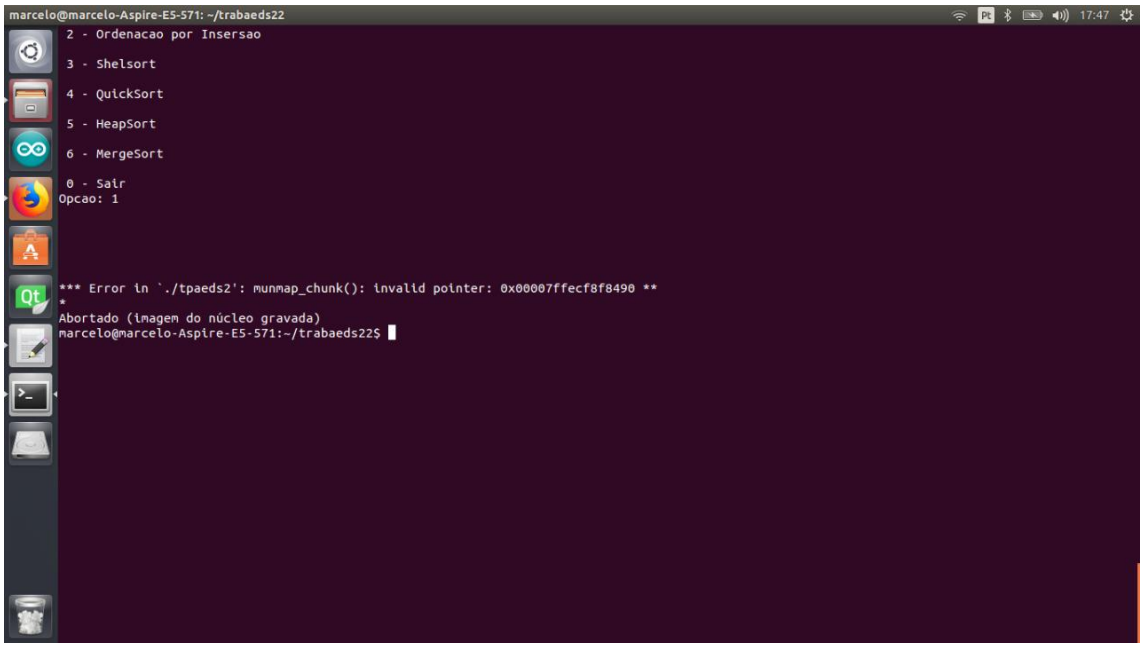
Nos Arquivos **RegistroPequeno.c** e **RegistroGrande.c** não há muita diferença do que é atribuído na função “main” do arquivo vetor.c. A diferença pode ser encontrada no tipo do vetor que será ordenado, neste caso, para as duas funções há as bibliotecas **"registrogran.h"** e **"registropeq.h"** que contém as duas estruturas (registro grande e registro pequeno). Para cada uma há um ponteiro focado em cada um desses dois tipos respectivamente. O restante do código continua o mesmo.

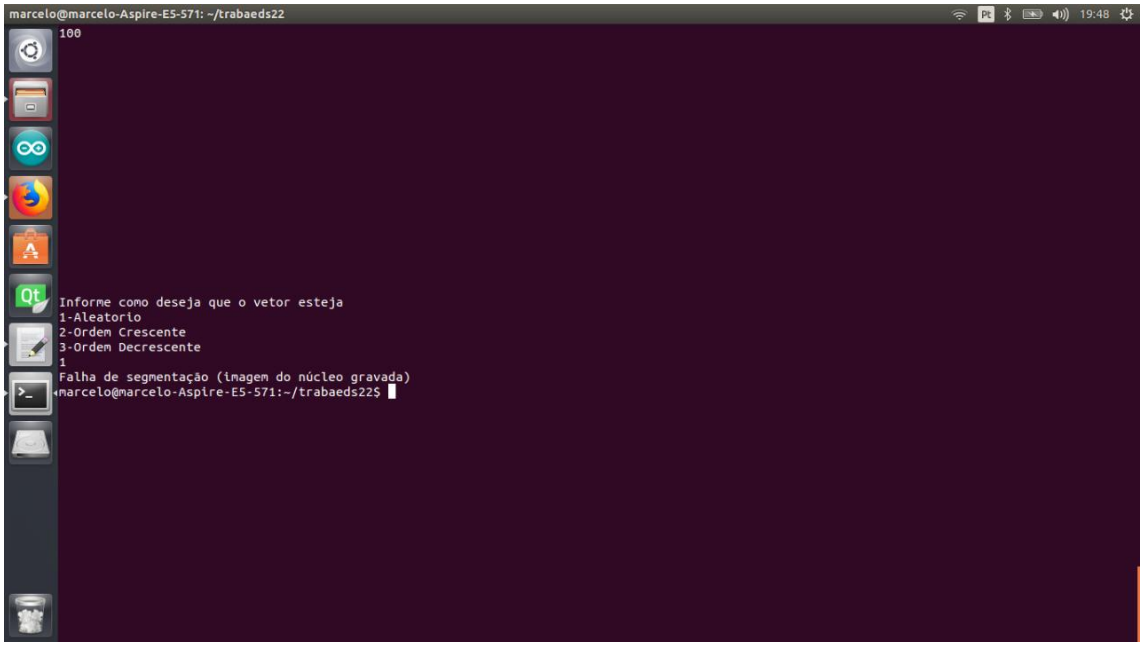
Testes

Para a criação do código foram utilizados dois aplicativos:
Falcon (versão para Windows)
Atom (versão para Linux)
Os testes foram realizados em dois computadores
– AMD Quad-Core A12-9720P RADEON R7 12 COMPUTE CORES 4C+8G 2.70GHz
– Intel(R)Core(TM)i3-4005 CPU @ 1.70GHz k1.70GHz
As figuras abaixo mostram saídas com erros na execução que foram resolvidos durante a criação do programa.

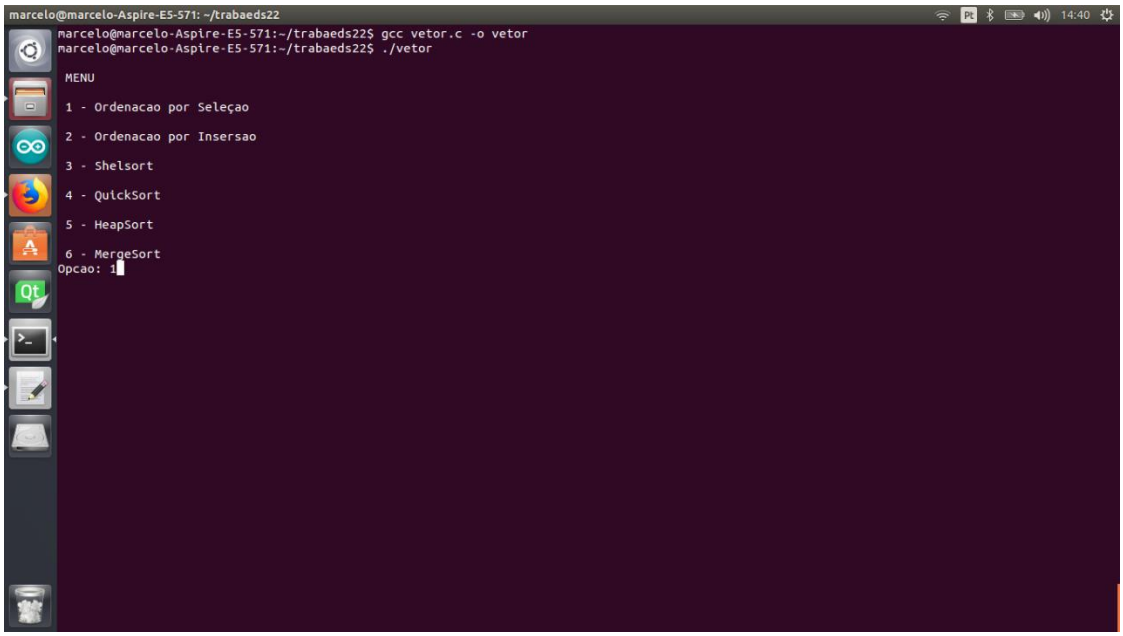


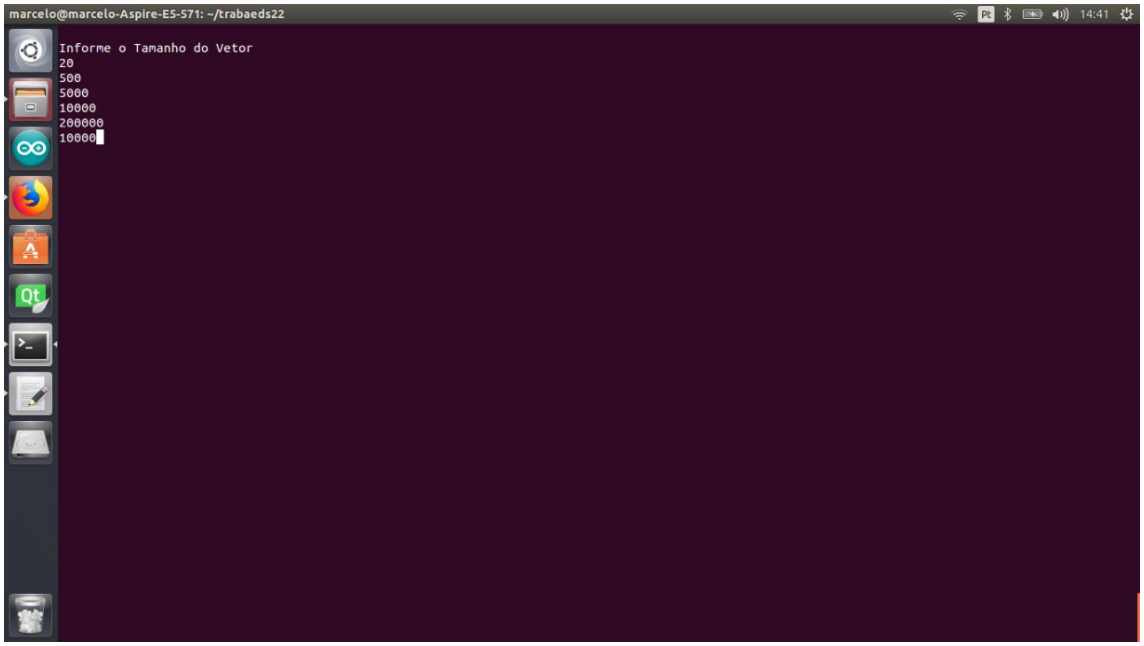


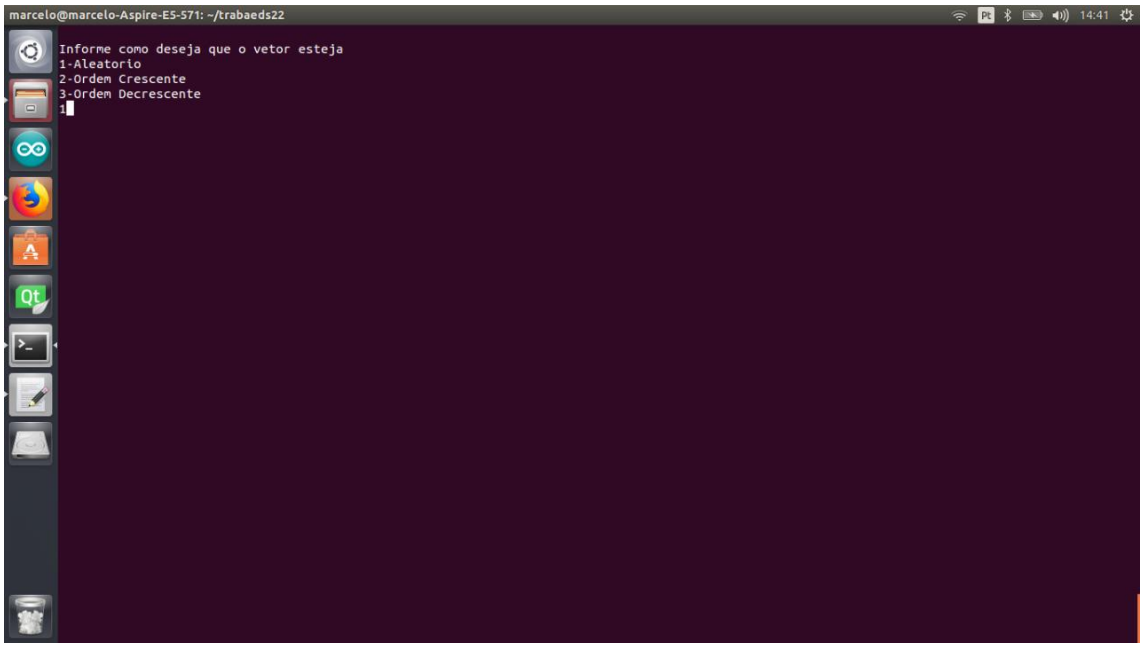


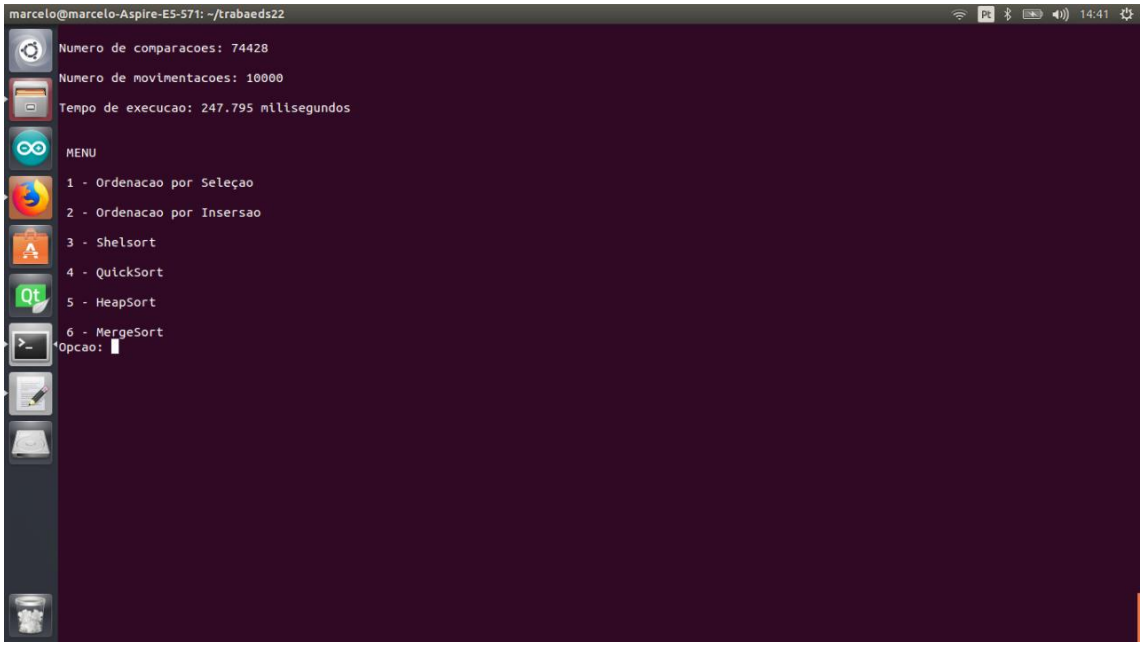


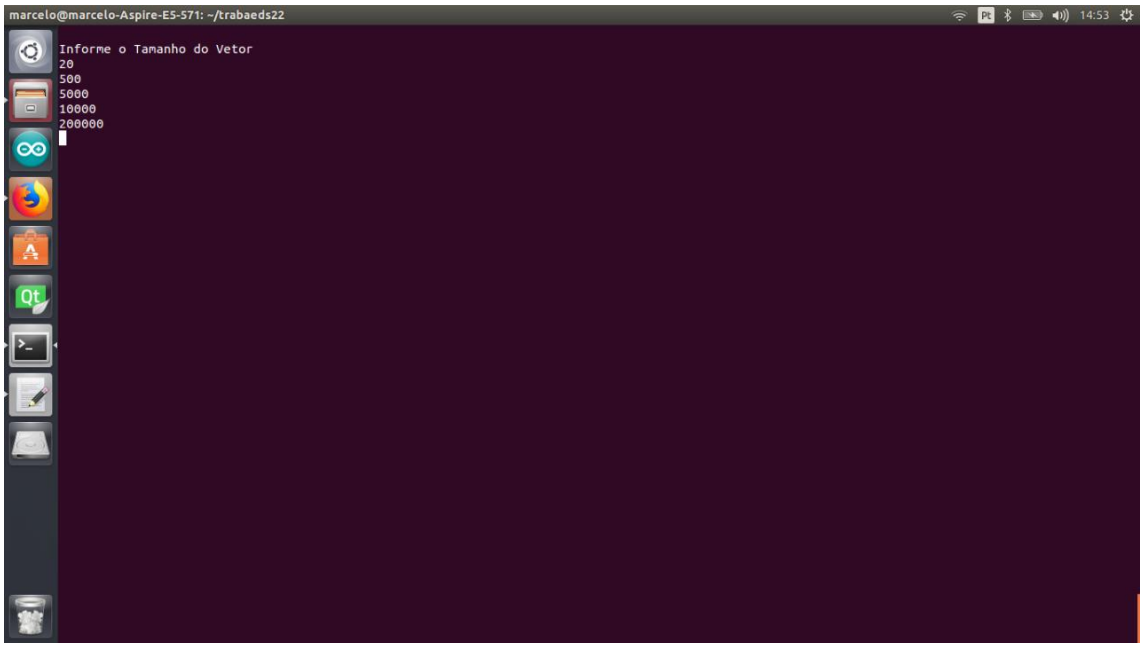
Já, estas figuras abaixo apresentam a saída de uma execução típica com o programa e código em seu devido funcionamento:

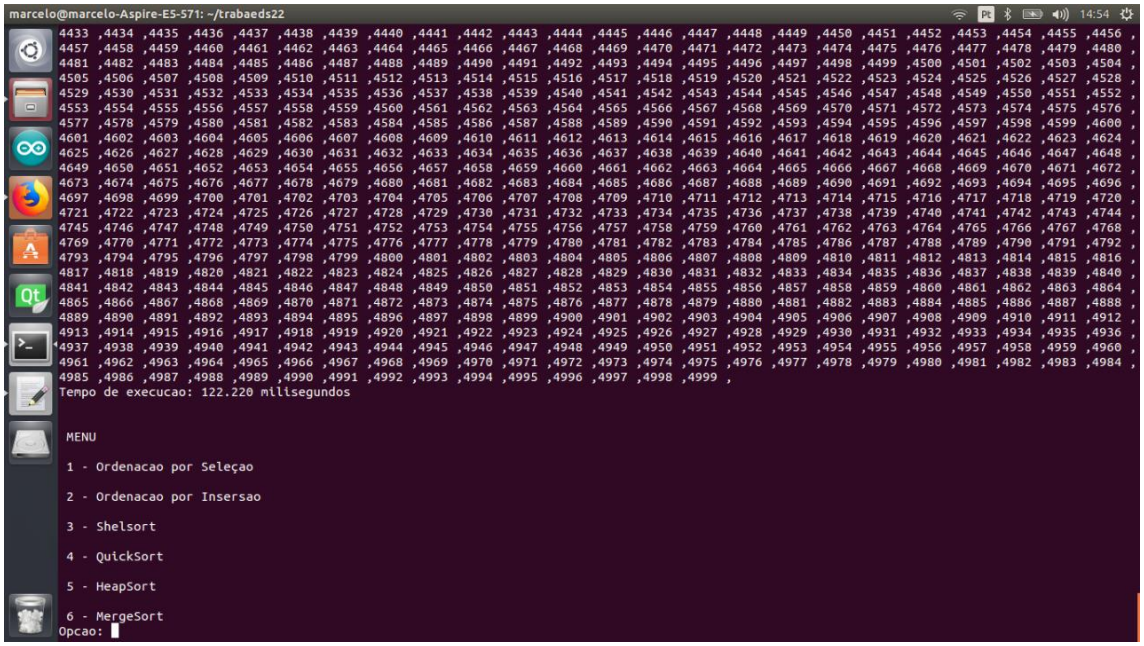

marcelo@marcelo-Aspire-E5-571: ~/trabaeds22
14:40
marcelo@marcelo-Aspire-E5-571:~/trabaeds22$ gcc vetor.c -o vetor
marcelo@marcelo-Aspire-E5-571:~/trabaeds22$ ./vetor
MENU
1 - Ordenacao por Seleçao
2 - Ordenacao por Insersao
3 - Shelsort
4 - QuickSort
5 - HeapSort
6 - MergeSort
Opcao: 1


marcelo@marcelo-Aspire-E5-571: ~/trabaeds22
14:41
Informe o Tamanho do Vetor
20
500
5000
10000
200000
10000


marcelo@marcelo-Aspire-E5-571: ~/trabaeds22
14:41
Informe como deseja que o vetor esteja
1-Aleatorio
2-Ordem Crescente
3-Ordem Decrescente
1


marcelo@marcelo-Aspire-E5-571: ~/trabaeds22
14:41
Numero de comparacoes: 74428
Numero de movimentacoes: 10000
Tempo de execucao: 247.795 milisegundos
MENU
1 - Ordenacao por Seleçao
2 - Ordenacao por Insersao
3 - Shelsort
4 - QuickSort
5 - HeapSort
6 - MergeSort
Opcao:





Análise do Tempo de execução

<u>Média do tempo de execução de cada caso:</u>

seleção

<u>aleatório</u>
20 ->0.004 milisegundos
500->1.023 milisegundos
5000->63.947 milisegundos
10000->247.232 milisegundos
200000->101409.598 milisegundos

Crescente
20->0.003 milisegundos
500->0.965 milisegundos
5000->63.035 milisegundos
10000->248.742 milisegundos
200000->98523.424 milisegundos

Decrescente
20->0.005 milisegundos
500->1.335 milisegundos
5000->63.911 milisegundos
10000->246.964 milisegundos
200000->98431.366 milisegundos

Inserção
aleatório
20->0.002 milisegundos
500->0.172 milisegundos
5000->7.165 milisegundos
10000->28.479 milisegundos
200000->55084.944 milisegundos

Crescente
20->0.001 milisegundos
500->0.019 milisegundos
5000->0.242 milisegundos
10000->0.377 milisegundos
200000->2.017 milisegundos

Decrescente
20->0.003 milisegundos
500->0.312 milisegundos
5000->14.020 milisegundos
10000->56.130 milisegundos
200000->110459.611 milisegundos

ShellSort

Aleatório
20->0.002 milisegundos
500->0.061 milisegundos
5000->0.665 milisegundos
10000->1.307 milisegundos
200000->123.304 milisegundos

Crescente
20->0.003 milisegundos
500->0.026 milisegundos
5000->0.468 milisegundos
10000->1.133 milisegundos
200000->21.384 milisegundos

Decrescente
20->0.002 milisegundos
500->0.054 milisegundos

5000->0.765 milisegundos
10000->1.215 milisegundos
200000->38773 milisegundos

QuickSort

Aleatório
20->0.001 milisegundos
500->0.116 milisegundos
5000->2.555 milisegundos
10000->7.744 milisegundos
200000-> 9023.923 milisegundos

Crescente
20->0.001 milisegundos
500->0.219 milisegundos
5000->11.302 milisegundos
10000->43.196 milisegundos
200000->48155.390

Decrescente
20->0.002 milisegundos
500->0.193 milisegundos
5000->5.948 milisegundos
10000->21.655 milisegundos
200000->42105.390 milisegundos

HeapSort

Aleatório
20->0.002 milisegundos
500->0.172 milisegundos
5000->1.866 milisegundos
10000->2.752 milisegundos
200000->87.535 milisegundos

Crescente
20->0.003 milisegundos
500->0.185 milisegundos
5000->2.212 milisegundos
10000->4.235 milisegundos
200000->77.077 milisegundos

Decrescente
20->0.002 milisegundos
500->0.092 milisegundos
5000->1.409 milisegundos
10000->3.226 milisegundos
200000->81.716 milisegundos

MergeSort

Aleatório
20->0.005 milisegundos
500->0.181 milisegundos

5000->2.394 milisegundos
10000->2.798 milisegundos
200000->95.204

Crescente
20->0.003 milisegundos
500->0.085 milisegundos
5000->1.099 milisegundos
10000->2.308 milisegundos
200000->59.317 milisegundos

Decrescente
20->0.004 milisegundos
500->0.176 milisegundos
5000->1.109 milisegundos
10000->2.229 milisegundos
200000->57.725 milisegundos

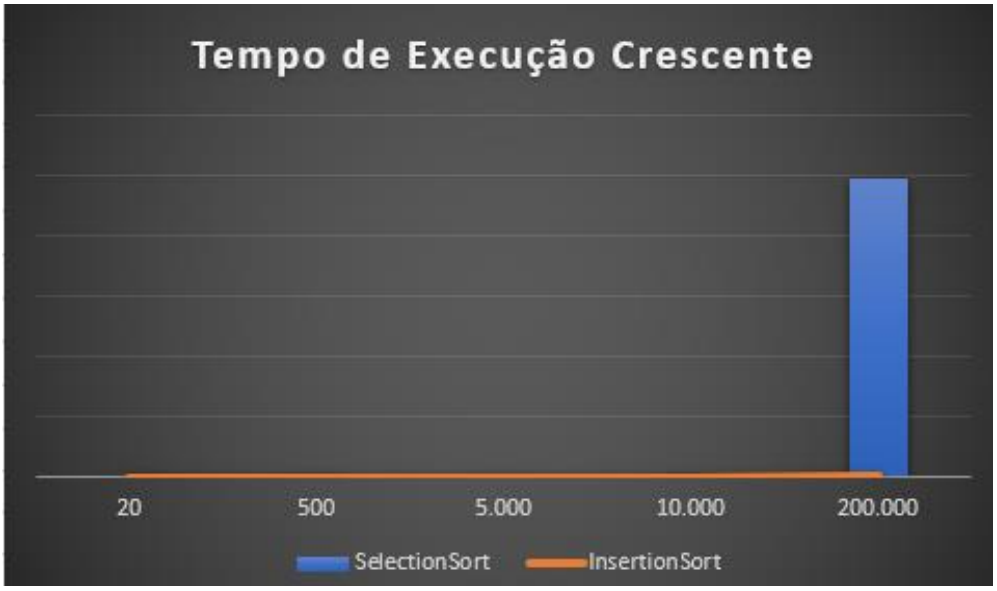
Comparação entre os métodos de ordenação simples:

Tempo de execução aleatório:

Método	20	500	5.000	10.000	200.000
SelectionSort	0.004	1.023	63.974	247.232	101.409.598
InsertionSort	0.002	0.172	7.165	28.479	55.084.944

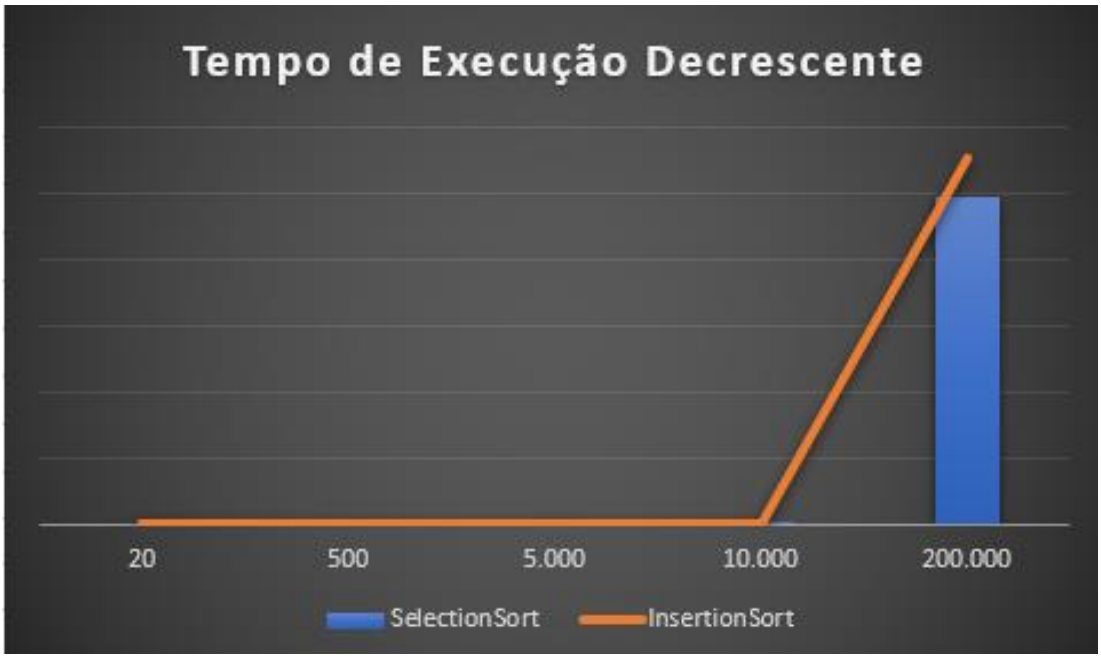
Tempo de Execução Crescente:

Método	20	500	5.000	10.000	200.000
SelectionSort	0.003	0.965	63.035	248.742	98.523.366
InsertionSort	0.001	0.019	0.242	0.377	2.017



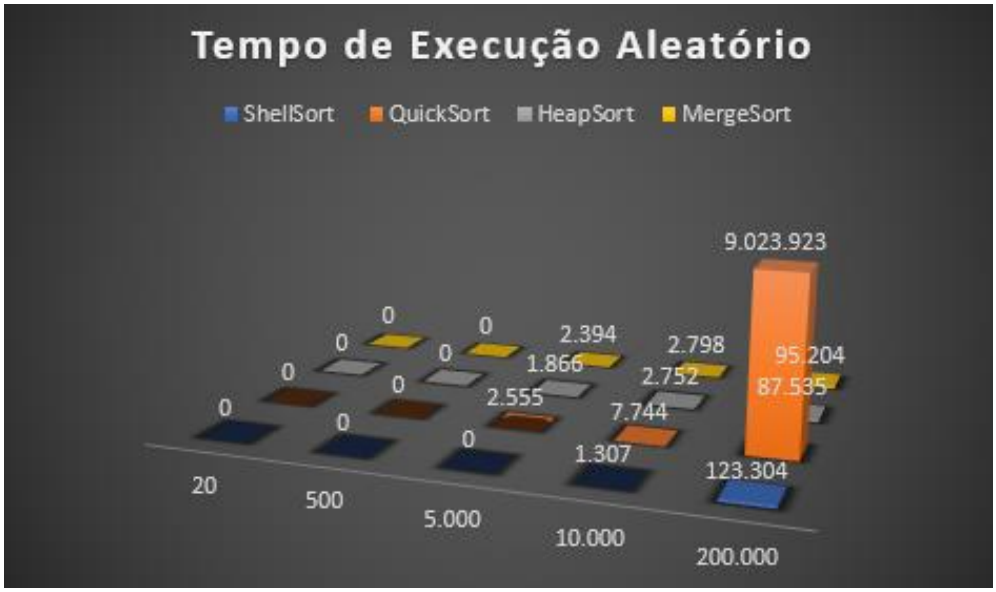
Tempo de execução Decrescente

Método	20	500	5.000	10.000	200.000
SelectionSort	0.005	1.335	63.911	246.964	98.431.366
InsertionSort	0.003	0.312	14.020	56.130	110.459.611

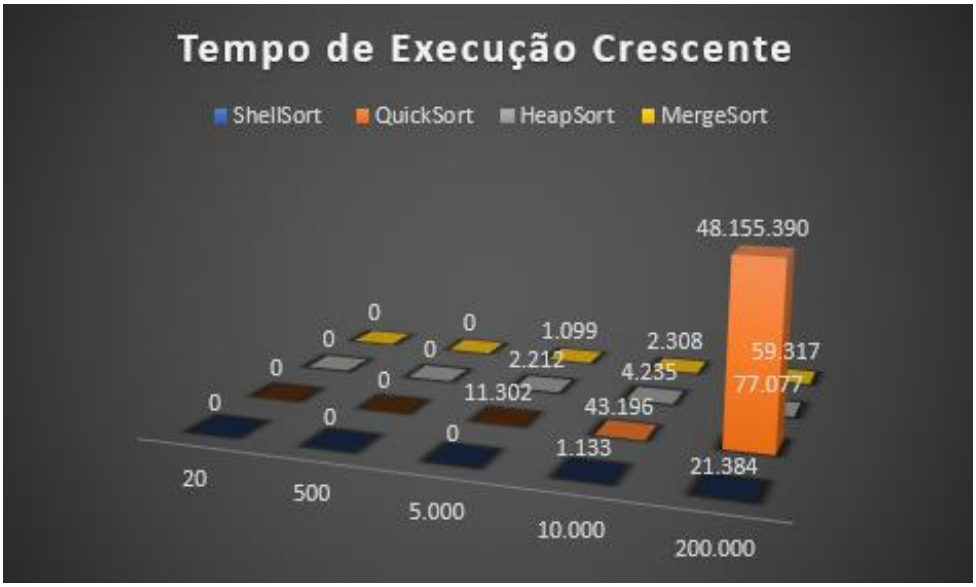


Comparação entre os métodos de ordenação Eficientes:

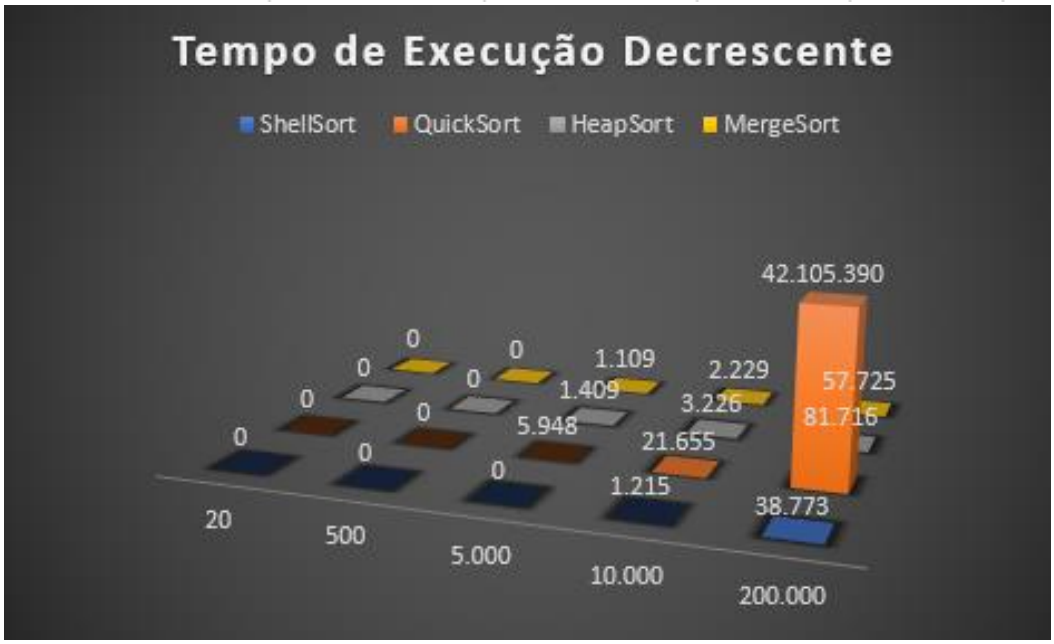
Métodos	20	500	5.000	10.000	200.000
ShellSort	0.002	0.061	0.665	1.307	123.304
QuickSort	0.001	0.116	2.555	7.744	9.023.923
HeapSort	0.002	0.172	1.866	2.752	87.535
MergeSort	0.005	0.181	2.394	2.798	95.204



Métodos	20	500	5.000	10.000	200.000
ShellSort	0.003	0.026	0.468	1.133	21.384
QuickSort	0.001	0.219	11.302	43.196	48.155.390
HeapSort	0.003	0.185	2.212	4.235	77.077
MergeSort	0.003	0.085	1.099	2.308	59.317



Métodos	20	500	5.000	10.000	200.000
ShellSort	0.002	0.054	0.765	1.215	38.773
QuickSort	0.002	0.193	5.948	21.655	42.105.390
HeapSort	0.002	0.092	1.409	3.226	81.716
MergeSort	0.004	0.176	1.109	2.229	57.725



Conclusão

Dessa forma, tendo em vista todas as recomendações, estudos em sala e externos, discussões sobre o tema e implementações realizadas, foi possível finalizar o trabalho, apesar das dificuldades encontradas que contribuíram para o atraso da entrega do trabalho.

Para resolver os problemas encontrados houve estudos paralelos a partir de leituras (obra declarada em referências) e também o comparecimento da dupla na monitoria disponibilizada pelo orientador e pela universidade.

Dessa forma, concluímos que a realização do trabalho foi de suma importância para o aprendizado sobre algoritmos de ordenação, visto que trabalhamos com diversas implementações até chegarmos no código correto e entregamos o trabalho com maior facilidade em enxergar soluções para os problemas propostos pelo assunto.

Referências

Obra *C completo e total - terceira edição*
Autor Herbert Schildt – Osborne

www.cprogressivo.net

Embasamento em sala de aula – Professor Rafael Sachetto UFSJ