**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECONOLOGIA DE MINAS GERAIS**

**CAMPUS SÃO JOÃO EVANGELISTA**

**Arthur Caldeira**

**TRABALHO PRÁTICO I**

**São João Evangelista**

**2021**

**Arthur Caldeira**

**TRABALHO PRÁTICO I**

Trabalho apresentado por exigência da disciplina de Algoritmos e Estruturas de Dados II, ministrada pelo professor Eduardo Augusto Costa Trindade, para o 3º período do curso Bacharelado em Sistemas de Informação, do Instituto Federal de Minas Gerais.

Orientador :Eduardo Augusto Costa Trindade

**São João Evangelista**

**2021**

**SUMÁRIO**

**1 INTRODUÇÃO ..................................................................................................**1

**2 IMPLEMENTAÇÃO ..........................................................................................**2

**3 ANÁLISE DE COMPLEXIDADE ......................................................................**4

**4 RESULTADOS .................................................................................................**5

**5 CONCLUSÃO ..................................................................................................**9

**6 APÊNDICE .....................................................................................................**11

**7 REFÊNCIAS BIBLIOGRÁFICAS ..................................................................**26

**1 INTRODUÇÃO**

O presente trabalho proposto pela disciplina de Algoritmos e Estruturas de Dados II, desenvolvido na linguagem de programação C/C++, tem como objetivo principal fazer um estudo comparativo entre os métodos de ordenação Bubble Sort, Insertion Sort, Selection Sort, Shell Sort, Quick Sort e Merge Sort, medindo o tempo gasto e a quantidade de comparações realizadas para chegar a ordenação completa da seqüência.

Para obter o tempo de execução dos algoritmos de ordenação, cada um será executado 10 vezes e depois será calculada uma média entre o tempo de cada compilação do método escolhido.

Para isso vão ser utilizados diferentes tamanhos de entrada, com seqüências de mil, dez mil e cem mil números, que foram divididas em lista aleatórias, quase ordenadas, inversamente ordenadas e listas já ordenadas. Com isso serão 12 instâncias ao todo, sendo elas:

* Lista Aleatória com mil números;
* Lista Quase Ordenada com mil números;
* Lista Inversamente Ordenada com mil números;
* Lista Ordenada com mil números;
* Lista Aleatória com dez mil números;
* Lista Quase Ordenada com dez mil números;
* Lista Inversamente Ordenada com dez mil números;
* Lista Ordenada com dez mil números;
* Lista Aleatória com cem mil números;
* Lista Quase Ordenada cem com mil números;
* Lista Inversamente Ordenada com cem mil números;
* Lista Ordenada com cem mil números;

**2 IMPLEMENTAÇÃO**

Para a implementação do código referente ao trabalho, por decisão de projeto, foram criados quatro tipos de arquivos diferentes, já que em apenas um arquivo ficaria muito grande para poder manuseá-lo.

O primeiro chamado de “Trabalho Prático.cpp” onde o código deve ser compilado, pois é o arquivo principal. Esse arquivo contém três funções, uma delas sendo a principal, outra sendo o menu secundário, sendo aqui onde o usuário escolhe a instância, e uma terceira chamada de limpa tela, que tem por objetivo único e exclusivo apagar dados que ficam na tela de exibição.

O segundo arquivo chamado de “Metodos de Ordenacao.h”, nessa pasta estão todos os métodos de ordenação que foram utilizados na implementação do algoritmo. As funções contidas nesse arquivo são : Bubble Sort, Selection Sort, Insertion Sort, Shell Sort (essas funções estão nos apêndices 1,2,3 e 4 respectivamente), para essas funções é passado um vetor de inteiros, uma variável que contém o tamanho do vetor e um ponteiro para fazer a contagem das comparações que são realizadas, também tem o Quick Sort(apêndice 5), para essa função é passado um vetor de inteiros, uma variável para guarda a primeira posição do vetor e uma para a ultima, além de um ponteiro para realizar a contagem do número de comparações, e por último o Merge Sort e Intercala (apêndices 6 e 7) que juntas fazem a ordenação do vetor, as variáveis passadas para elas são um vetor, o inicio e o final do mesmo e uma variante para realizar a contagem de comparações, para o Intercala é passado todas essas referências e mais uma que é o meio do vetor.

O terceiro arquivo chamado de “Manipulação de Arquivos.h” que é responsável de ter todos os métodos utilizados tanto para a leitura dos arquivos com as instâncias quanto para o gravação do tempo decorrido de cada função de ordenação e suas comparações realizadas. A função FileRead1000 (apêndice 8) realiza a leitura dos arquivos que tem mil números, nela é feita uma comparação usando if, if else e else para saber qual foi a opção escolhida no menu secundário e de acordo com isso saber qual o nome do arquivo que deve ser lido. Após a identificação do arquivo, o mesmo é aberto para leitura do seu conteúdo, que ocorre através de um laço de repetição while, onde é verificado se o arquivo ainda não chegou ao seu fim, dentro desse laço é realizada a transição dos valores do arquivo para um vetor através da função fscanf. As funções FileRead10000 e FileRead100000 (apêndices 9 e 10, na devida ordem) realizam o mesmo processo da anterior, porém para arquivos de dez mil e cem mil números. Para todas essas funções são passados os mesmos parâmetros, sendo eles, um ponteiro para ponteiro do tipo FILE, um ponteiro para um vetor de inteiros e a opção escolhida no menu secundário. A função WriteFile é responsável por realizar a escrita do tempo decorrido do método de ordenação e a quantidade de comparações realizadas. Para essa função são passados como parâmetros um ponteiro para ponteiro do tipo FILE, uma variável do tipo long double para o tempo decorrido, uma variável long long int para o número de comparações, foi necessário utilizar uma variante maior que o inteiro comum devido ao grande número de comparações que eram realizadas por algumas funções de ordenação fazendo com que o tamanho normal do int fosse ultrapassado, também foram passadas as opções escolhidas tanto no menu inicial quanto no menu secundário. A opção do menu secundário serve para saber qual é tipo de arquivo que terá de ser criado, se é um arquivo para uma lista aleatória que foi ordenada, ou uma lista quase ordenada, ou uma lista inversamente ordenada ou ainda uma lista já ordenada além de definir qual é o seu tamanho. Agora a opção do menu inicial serve para saber qual o método que foi escolhido para a ordenação do arquivo, assim podendo nomeá-lo. Com isso é possível identificar o tipo de lista usado e com qual recurso.

O quarto arquivo chamado de “Resultado Final.h” realiza a manipulação de leitura e gravação em arquivos, além de ser nele onde é medido o tempo de execução dos métodos de ordenação através da função clock\_t. Nesse arquivo tem a atribuição MethodName (apêndice 11) que serve única e exclusivamente para mostrar o nome do artifício que foi usado para ordenação do vetor, esse método recebe apenas a opção do menu inicial como parâmetro, há também a função ImprimeVetor (apêndice 12) que tem por finalidade mostrar os números após o método de ordenação ser aplicado na lista. A função KeepFile1000 (apêndice 13) tem o objetivo de realizar a contagem de clocks que ocorrem até a ordenação completa de um vetor, para isso é necessário ter duas variáveis, uma para pegar o clock inicial, de quando o processo de ordenação começa, e a outra para guardar o ultimo clock do processo, depois disso é feito o calculo dos clocks gastos para alinhar completamente o vetor, ocorrendo através da subtração do clock final com o inicial e logo em seguida é feita uma conversão para segundos, utilizando-se do CLOCKS\_PER\_SEC, após isso é gravado no arquivo a quantidade comparações que foram necessárias para perfilar o vetor e o tempo gasto para isso, porém, com listas de tamanho mil. Com as funções KeepFile10000 e KeepFile100000 (apêndices 14 e 15) acontece o mesmo, contudo, para listas com tamanhos de dez mil e cem mil registros. Para todas essas funções os parâmetros passados são os mesmos, sendo eles, um ponteiro de ponteiro do tipo FILE, a opção escolhida no menu inicial, o tamanho do vetor e a opção do menu secundário que serve para ser passada como parâmetro para as funções de leitura de escrita em arquivos. A opção do menu inicial é utilizada para definir qual método de ordenação deve ser utilizado para aplicar no vetor.

**3 ANÁLISE DE COMPLEXIDADE**

O Bubble Sort é um algoritmo de ordenação simples, que percorre o vetor várias vezes sempre colocando o número de menor valor no inicio do mesmo. No melhor caso desse método o número de comparações é N – 1 comparação, onde N é o tamanho da entrada, e seu pior caso é N^2. Sendo assim a sua complexidade é de ordem quadrática, ou seja, O(n^2), dessa maneira sendo um recurso que demanda muito tempo quando a entrada de dados é muito grande, já que o tempo de execução cresce quadraticamente.

O Insertion Sort é um método que utiliza a inserção para fazer a organização de um vetor, ele percorre o vetor da esquerda para a direita ordenando os elementos a esquerda conforme vai avançando. A sua complexidade é O(n) no melhor caso, crescendo de forma linear, e O(n^2) no seu caso médio e pior caso, com isso seu crescimento passa a ser quadrático. Além de ser considerado um método de ordenação estável, uma vez que algum número já foi ordenado ele não será trocado de posição mais.

O Selection Sort consiste em dividir a estrutura a ser ordenada em duas, uma contendo os valores já ordenados e a outra com valores a serem ordenados. Desse modo ele compara um elemento com todos os outros visando encontrar o menor, assim pode-se entender que não existe um melhor, pior ou caso médio. A complexidade desse algoritmo é O(n^2), já que ele realiza todas as comparações possíveis mesmo o vetor estando ordenado, fazendo com que seu tempo cresça quadraticamente.

O Shell Sort é um refinamento do Insertion Sort, já que também utiliza inserções para realizar a ordenação do vetor, porém ele é mais eficiente já que considera o array a ser ordenado como vários segmentos, com isso aplica-se o método de inserção em cada um deles. A complexidade desse algoritmo é de O(n^2), uma vez que N cresce as operações crescem quadraticamente. Com tudo esse método de ordenação é o melhor dos que tem essa complexidade.

O Quick Sort é um artifício que utiliza o método da divisão e conquista para conseguir ordenar completamente um vetor. A divisão desse método consiste em partir o vetor ao meio escolhendo um pivô, daí em diante começa a conquista, colocando os valores menores que o pivô a sua direita e os maiores a esquerda, repetindo isso até o vetor ficar totalmente ordenado. O melhor caso desse método só ocorre quando a partição é balanceada, ou seja, cada parte com n/2 elementos, o pior caso acontece quando um dos lados do vetor fica com mais elementos que o outro. A complexidade desse algoritmo é O(n log n), pois o tempo aumenta de forma log linear e não exponencialmente como nos métodos citados anteriormente.

O Merge Sort também utiliza a divisão e conquista para ordenar o vetor, para isso ele faz a divisão até chegar a um ponto indivisível, daí parte para a combinação dos resultados, reagrupando as divisões de forma ordenada. A complexidade desse algoritmo é O(n log n), pois como no Quick Sort, o tempo aumenta de forma log linear, porém há uma diferença, o merge necessita de um espaço adicional proporcional a N, por isso do ponto de vista da memória ele é O(n), pois necessita uma alocação de espaço extra.

**4 RESULTADOS**

Para obter os resultados necessários para construir as tabelas e gráficos o seguinte processo foi realizado:

1º- cada método de ordenação foi executado dez vezes e o seu resultado foi salvo em um arquivo que contém o nome do processo realizado, a quantidade de comparações feitas e o tempo gasto para a execução;

2º- após gravar os resultados em arquivos diferentes, cada um com apenas uma instância para cada método de ordenação, os dados recolhidos foram colocados em uma planilha do Excel;

3º- para calcular o tempo de execução dos algoritmos desenvolveu-se um calculo para obter a média entre os tempos, assim conseguindo um valor estático mais confiável;

4º- logo após o alcance desse valor médio, ocorreu a implementação das tabelas, que contém o nome do método utilizado, o tipo da instância em que foi compilado e a quantidade de comparações juntamente com o tempo de execução;

5º- posteriormente efetuou-se a construção dos gráficos que demonstram os resultados das tabelas, para isso, novamente, foi utilizada a ferramenta Excel;

Tabela das instâncias de 1000 números

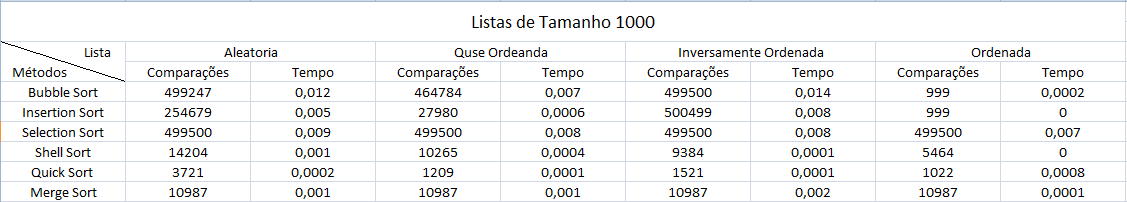
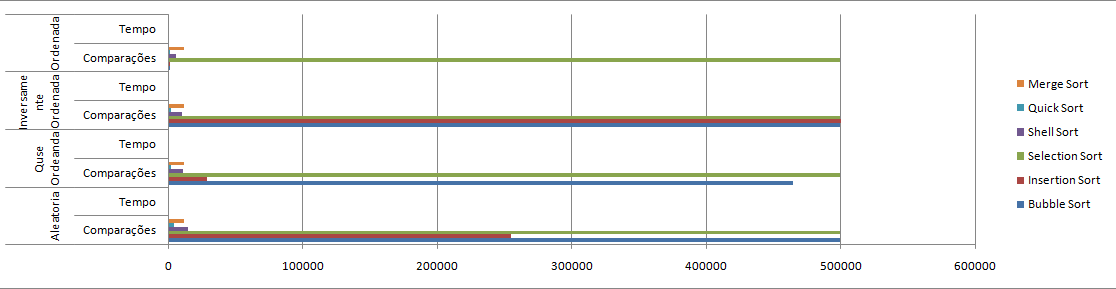


Gráfico das instâncias de 1000 números



Observando a lista aleatória pode ser visto que o método que realizou a ordenação em menos tempo e com menos comparações foi o Quick Sort, isso deve-se ao fato de ser um algoritmo que utiliza a divisão e conquista para realizar a organização do vetor além de ter suas constantes menores que as do Merge Sort, que também utiliza divisão e conquista, sendo assim o mais eficiente. Na lista quase ordenada o Quick Sort mostrou mais uma vez que consegue realizar a ordenação com menos movimentos e tempo, isso pode ser observado na lista inversamente ordenada também. Já na lista ordenada os métodos que realizaram menos comparações foram o Bubble Sort e Insertion Sort, com 999 cada um, isso se deve ao fato desses métodos realizarem N – 1 comparação quando o array já está ordenado, porém os métodos mais rápidos nessa ordenação foram o Insertion e Shell Sort, já que não foi necessário fazer nenhuma troca, o processo foi instantâneo.

Tabela das instâncias de 10000 números

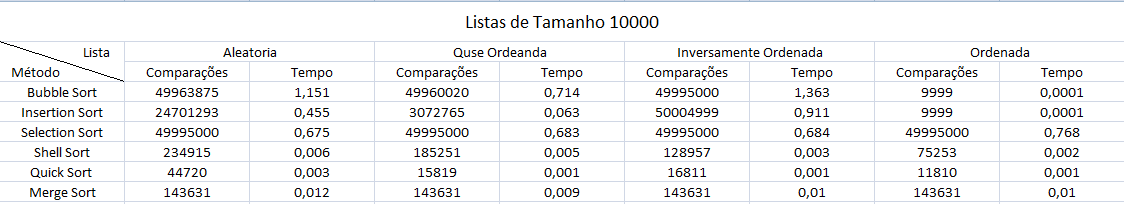
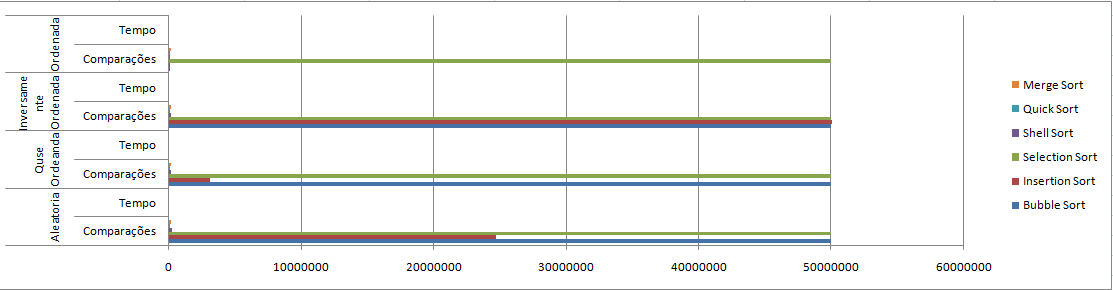


Gráfico das instâncias de 10000 números



Ao observar a tabela e o gráfico pode ser identificado, primeiramente na lista aleatória, que o melhor método é o Quick Sort, pois realizou menos comparações e em menos tempo que os demais, já o pior método que realiza essa ordenação é o Insertion Sort, já que ele realiza mais comparações, agora na questão do tempo o pior foi o Bubble Sort. Na lista quase ordenada e inversamente ordenada, mais uma vez, o Quick Sort mostrou-se mais eficiente devido a sua complexidade ser O(n log n), o pior algoritmo, no caso da lista quase ordenada, é o selection sort porque independentemente da posição dos números ele sempre faz o mesmo número de comparações, para a lista inversamente ordenada o pior foi o Insertion Sort, isso deve-se ao fato da lista estar em ordem contrária. Agora na lista ordenada os métodos que se provaram serem os melhores são o Bubble e o Insertion Sort, uma vez que nos seus melhores casos eles tendem a fazer uma comparação a menos que a quantidade de dados que se da na entrada e por fazerem menos comparações, nesse caso, também foram os métodos que gastaram menos tempo para verificar a ordenação.

Tabela das instâncias de 100000 Números

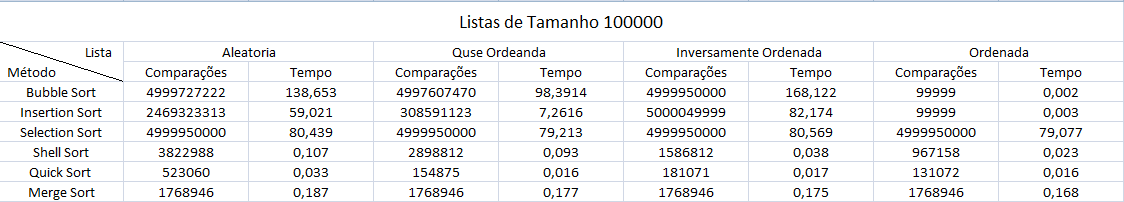
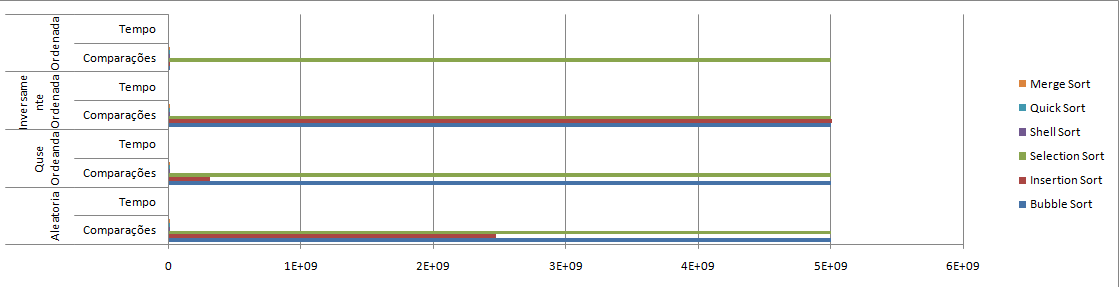


Gráfico das instâncias de 100000 números



No primeiro caso, lista aleatória, o método que se mostrou ser o melhor foi, novamente, o Quick Sort, já que gastou menos tempo e menos comparações com uma entrada de dados muito grande, o pior algoritmo nesse caso, em questão de número de comparações, foi o Selection Sort, já que ele sempre busca o menor valor para colocar na posição correta, agora olhando na questão do tempo, o pior algoritmo foi o Bubble Sort, isso deve-se ao fato dele comparar número por número para chegar a uma ordenação completa. Na lista quase ordenada o melhor algoritmo é o Quick Sort, tanto no tempo quanto na quantidade de comparações, e tendo como os piores o Bubble Sort e Selection Sort. Na lista inversamente ordenada o pior código para ordenação foi o Inserction Sort, uma vez que a lista está em ordem decrescente é necessário um alto número de comparações para poder identificar a posição correta de cada valor, tendo como melhor método o Quick Sort. Na lista já ordenada o pior método foi o Selection Sort, pois independentemente de como o array estiver ele sempre fará a mesma quantidade de comparações assim tendo o pior tempo de todos, o melhor nesse caso foi o Bubble e Insertion já que essa lista representa seus melhores casos.

**5 CONCLUSÃO**

O algoritmo de ordenação Bubble Sort embora seja de fácil implementação, não apresentou resultados satisfatórios para entradas grandes de dados, em que esses não estão completamente ordenados, uma vez que é um algoritmo muito lento fazendo com que demore muito tempo para organizar um vetor, assim utilizando muitos recursos computacionais que às vezes podem não estar a disposição.

O Insertion Sort se mostrou mais efetivo em listas menores, de preferência já ordenadas, pois é um método que demanda um certo números de comparações elevados quando não está em seu melhor caso e quando a entrada de dados é muito grande, exemplo, na lista aleatória com cem mil números sua quantidade de comparações foi de 2469323313 (dois bilhões quatrocentos e sessenta e nove milhões trezentos e vinte e três mil trezentos e treze), demandando um certo nível processamento maior para listas com mais números ainda.

O Selection Sort, assim como o Insertion Sort, não se mostrou muito eficiente quando a grandes entradas de elementos, demandando muitas comparações até mesmo para lista que estão completamente ordenadas.

O Shell Sort é um dos algoritmos que apresentou um ótimo desempenho tanto com entradas de dados pequenas quanto entradas grandes independentemente da forma que os elementos estejam dispostos na lista, esse método consegue organizá-los evitando elevados números de comparações como o insetion, selection e bubble, por conseqüência sendo mais efetivo na questão do tempo gasto para o arranja mento do array.

O Quick Sort foi o algoritmo que apresentou os melhores resultados, independente da quantidade de dados que foram processados e o modo como estavam ordenados, esse método realizou, na maioria das vezes, o melhor tempo para arranjamento em ordem crescente das listas, além de não realizar tantas comparações mesmo quando a instância já está ordenada.

O Merge Sort não é o melhor método de ordenação dos que foram propostos, porém os seus resultados são bastante satisfatórios, uma vez que não é gasto muito tempo para a ordenação e a quantidade de comparações não é tão exorbitante quanto à de outros métodos vistos anteriormente.

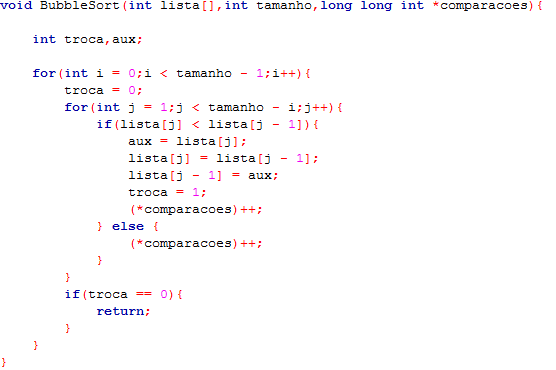
Após todas as comparações realizadas e resultados obtidos pode-se chegar a conclusão que o Quick Sort é o melhor método de ordenação dos que foram estudados, isso se pode ser visto nas listas com maiores quantidades de números, ele sempre consegue ordená-las com uma quantidade razoável de comparações e o mais importante com um tempo muito pequeno já que em todas as instâncias utilizadas o mesmo não levou nem um segundo para ordenação completa da mesma. Vale ressaltar também que o método Shell Sort também é muito bom, pois de acordo com os testes realizados e resultados obtidos, na maioria das vezes, ele se mostrou superior as Merge Sort em questão de tempo de ordenação.

As principais dificuldades encontradas para a implementação do algoritmo que fornece os dados utilizados para está pesquisa, foi a parte da manipulação dos arquivos, principalmente para realizar a leitura dos arquivos que continham as instancias. Para conseguir fazer isso foram necessários vários testes com diferentes formas de guardar os dados do arquivo em um vetor, uma vez que não me adaptei a utilizar alguns artifícios da linguagem de programação C++.

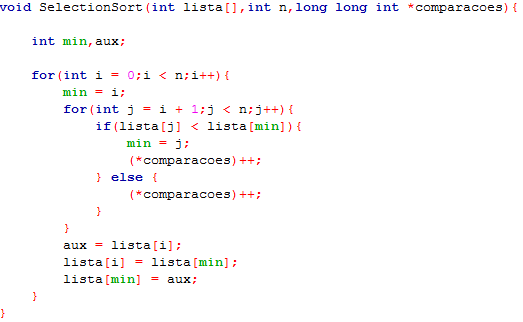
Contudo estou bem satisfeito com os resultados obtidos, que na maioria das vezes já eram esperados, todavia o Shell Sort me impressionou, já que não utiliza recursão, pensei que não seria tão eficiente quanto o Quick e Merge Sort.

**6 APÊNDICE**

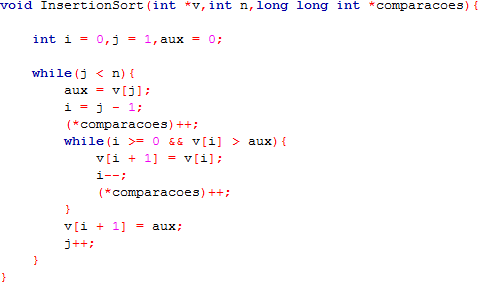
**APÊNDICE 1 –** Código do Bubble Sort



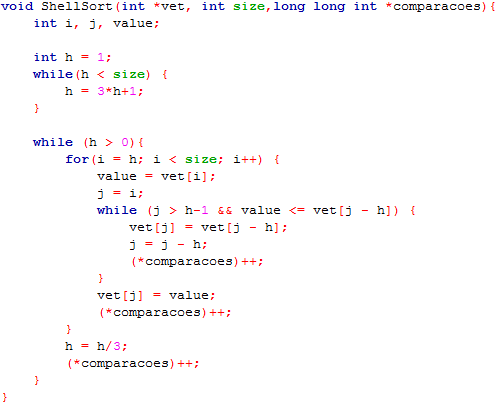
**APÊNDICE 2 –** Código do Selection Sort



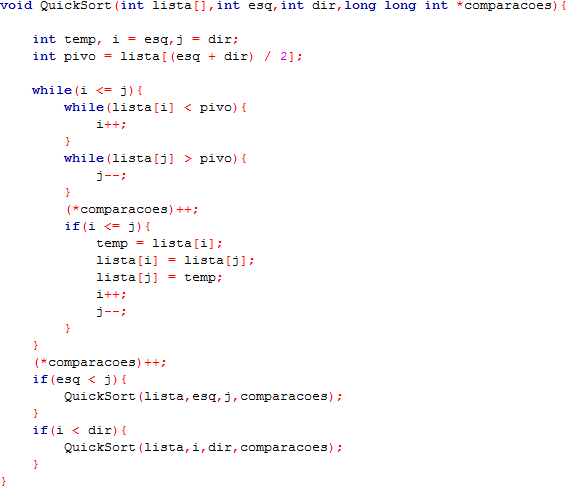
**APÊNDICE 3 –** Código do Insertion Sort



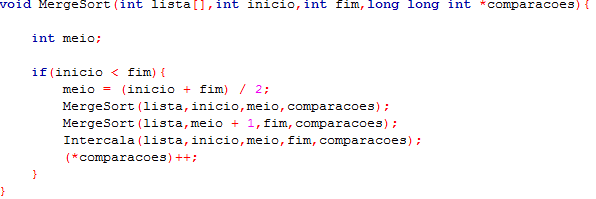
**APÊNDICE 4 –** Código do Shell Sort



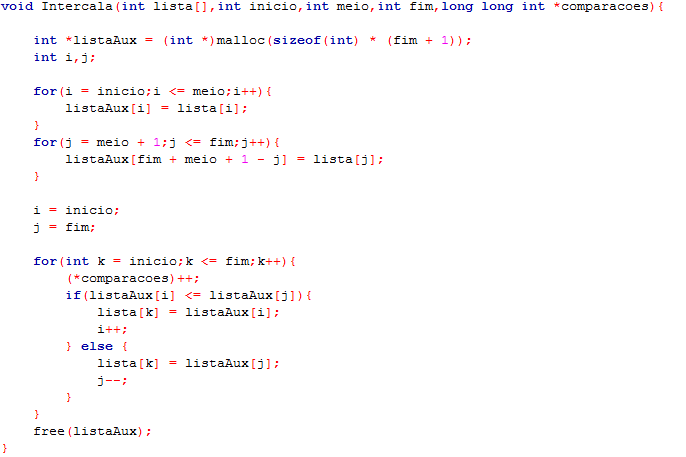
**APÊNDICE 5 –** Código do Quick Sort



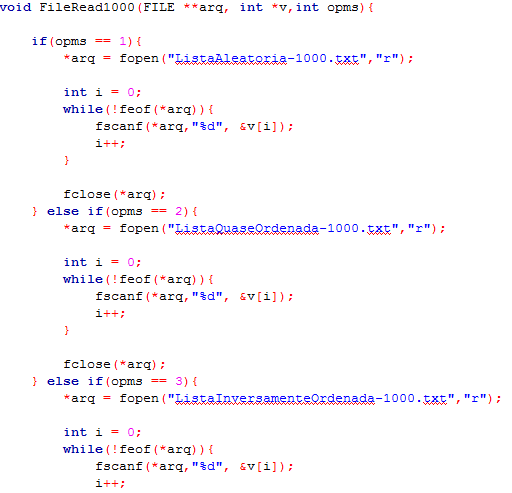
**APÊNDICE 6 –** Código do Merge Sort

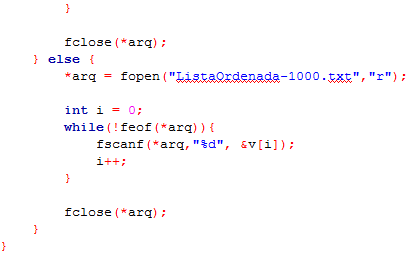


**APÊNDICE 7 –** Código do Intercala

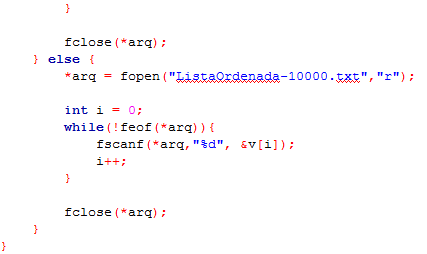
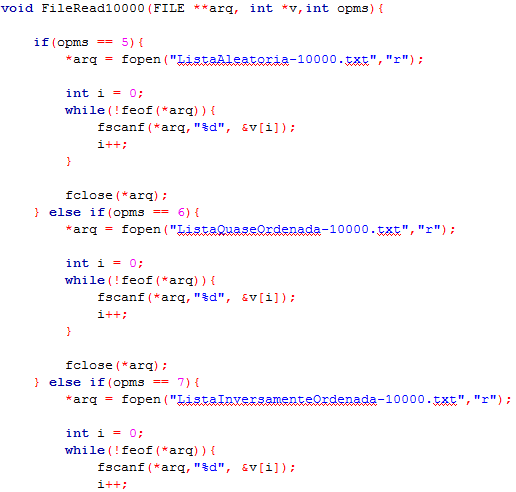


**APÊNDICE 8 –** Código da função FileRead1000

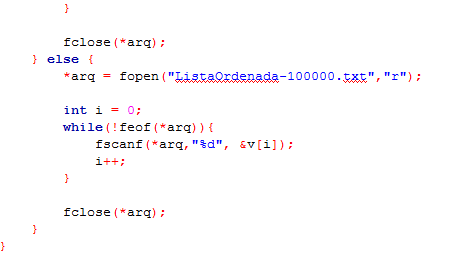
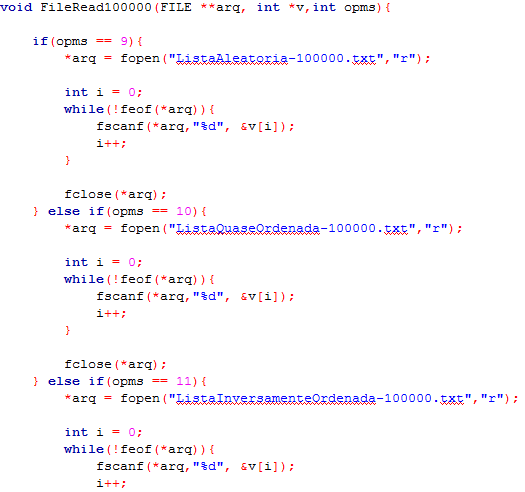




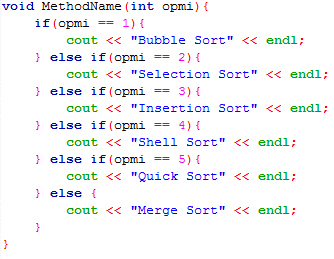
**APÊNDICE 9 –** Código da função FileRead10000



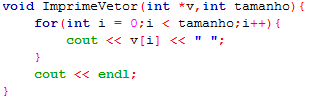
**APÊNDICE 10 –** Código da função FileRead100000



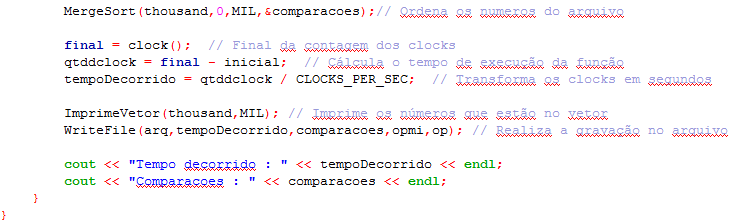
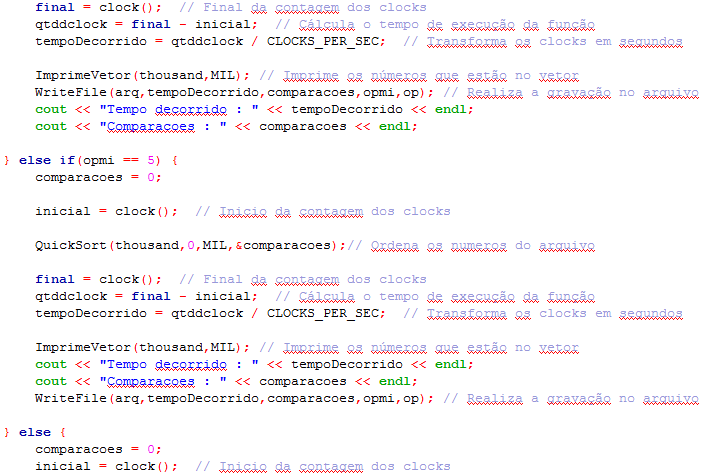
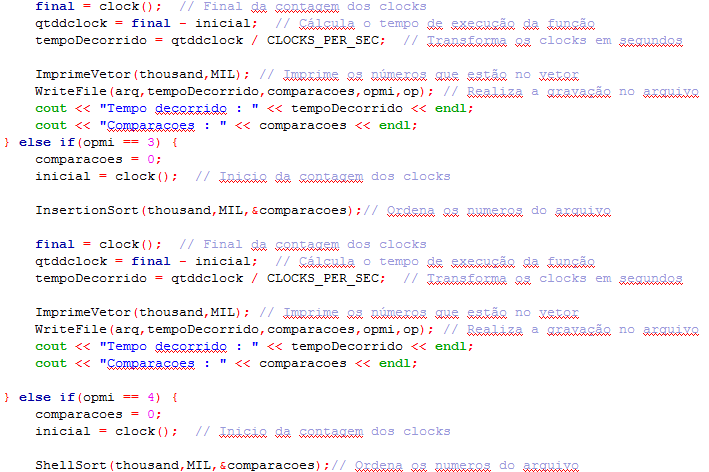
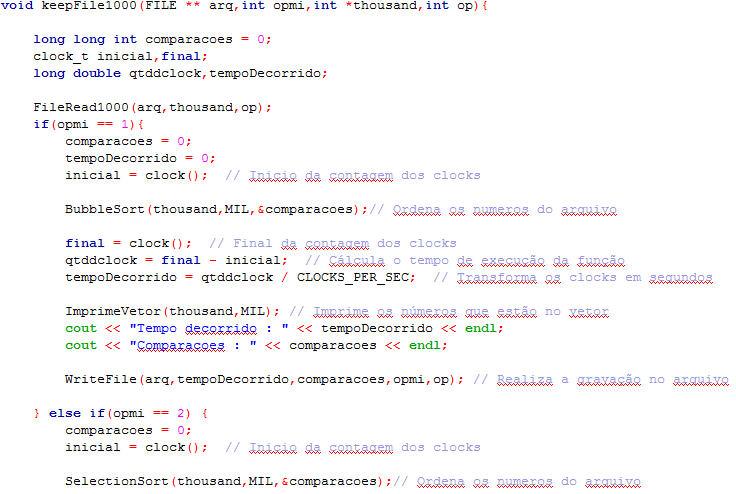
**APÊNDICE 11 –** Código da função MethodName



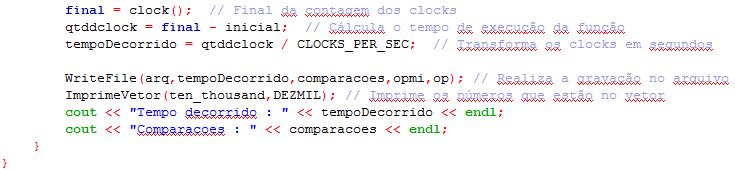
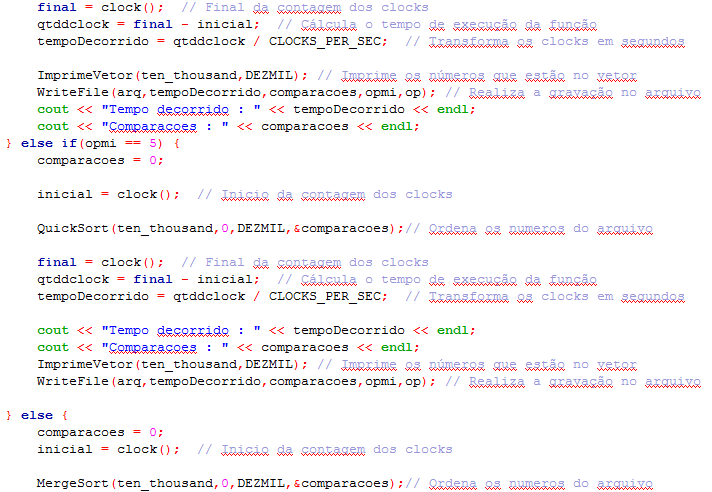
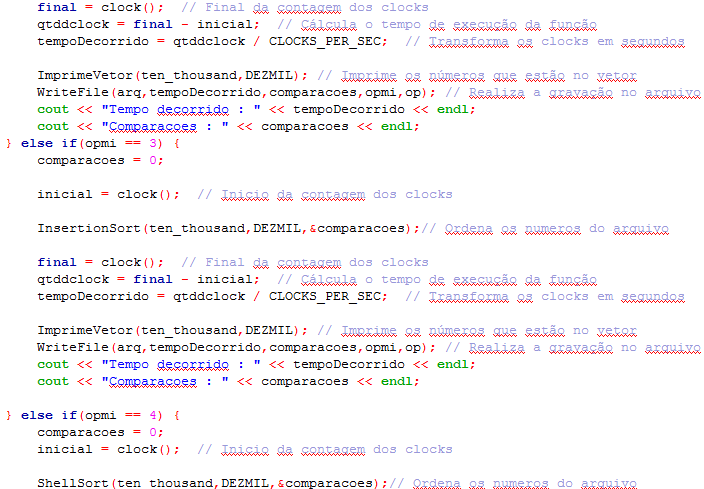
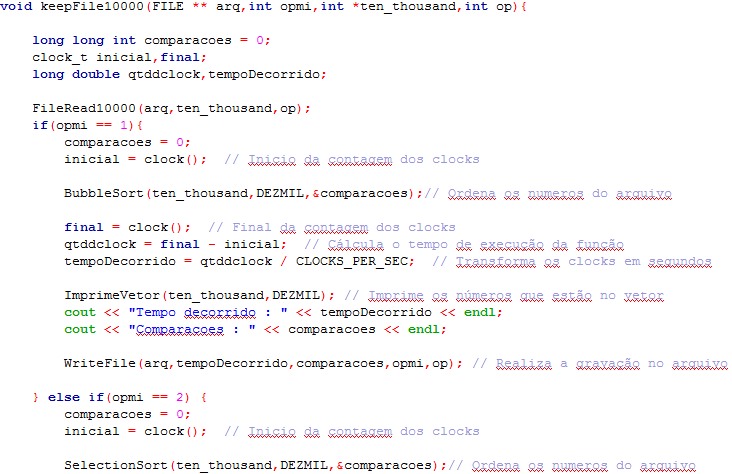
**APÊNDICE 12 –** Código da função ImprimeVetor



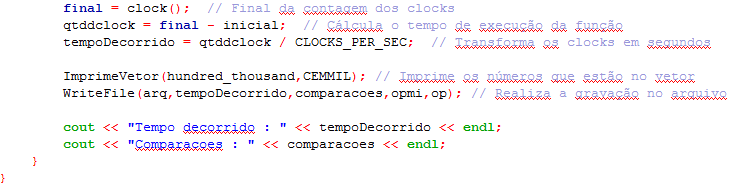
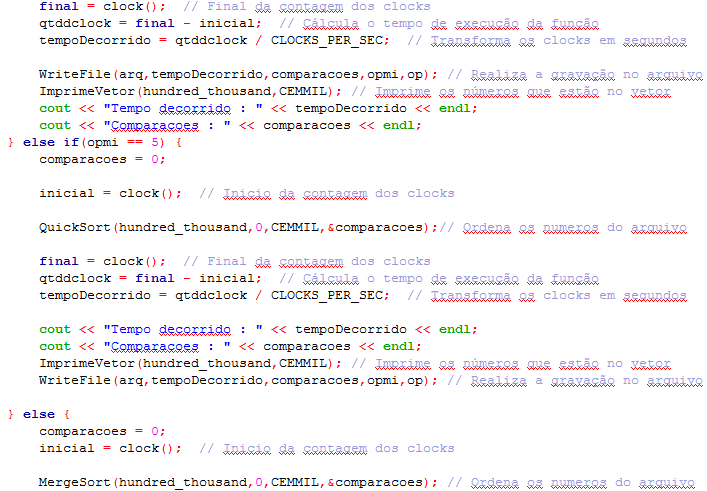
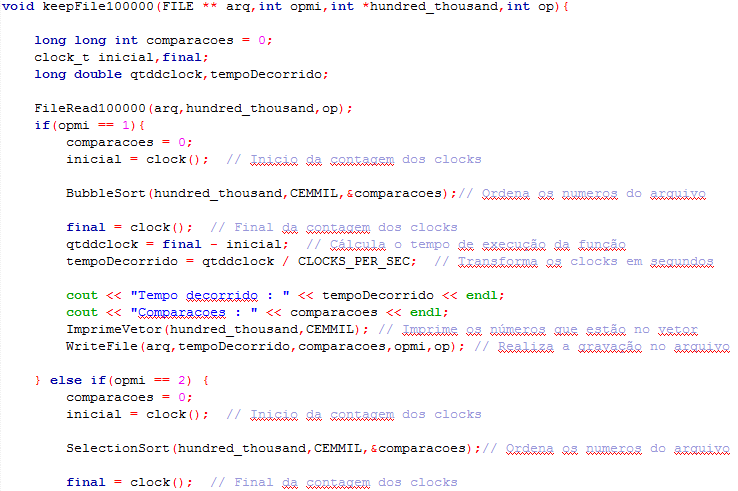
**APÊNDICE 13 –** Código da função KeepFile1000



**APÊNDICE 14 –** Código do KeepFile10000



**APÊNDICE 15 –** Código do KeepFile100000



**7 REFÊRENCIAS BIBLIOGRÁFICAS**

WIKIPÉDIA .Bubble Sort. Disponível em : <<https://pt.wikipedia.org/wiki/Bubble_sort> >. Acesso em 27/07/2021.

TREINAWEB. Conheça os principais algoritmos de ordenação. Disponível em :<<https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao>>. Acesso em 27/07/2021.

BLOG CYBERINI. Quicksort (análise e implementações). Disponível em : < <https://www.blogcyberini.com/2018/08/quicksort-analise-e-implementacoes.html> >. Acesso em 27/07/2021.

WIKIPÉDIA. Selection Sort. Disponível em :

< <https://pt.wikipedia.org/wiki/Selection_sort> >. Acesso em 27/07/2021.

WIKIPÉDIA. Shell Sort. Disponível em : < <https://pt.wikipedia.org/wiki/Shell_sort> >. Acesso em : 27/07/2021.

SOUSA, J. E. G et al. Algoritmos de Ordenação : Um Estudo Comparativo. Universidade Federal Rural do Semi-Árido, Curso de Ciência e Tecnologia. Disponível em :

< <https://periodicos.ufersa.edu.br/index.php/ecop/article/view/7082> >. Acesso em : 27/07/2021.