



UNIVERSIDADE FEDERAL DE JUIZ DE FORA



Relatório do trabalho de Estrutura de Dados II UFJF

Professor: Marcelo Caniato Renhe

Grupo: Bryan Carolino Muniz Barbosa (201965118A)
Gabriel Martins Quintana Vieira Marques (201835009)
Matheus Gomes Luz Werneck (201835037)
Pedro Henrique Almeida Cardoso Reis(201835039)

Juiz de Fora, 31 de Janeiro, 2021

Conteúdo

- 0 **Comentários** (Pág 2)
- 1 **Introdução** (Pág 2)
- 2 **Atividades realizadas por cada membro** (Pág 3)
- 3 **Estruturas de dados utilizadas** (Pág 4)
- 4 **Processamento de arquivos** (Pág 5-6)
- 5 **Algoritmos de ordenação** (Pág 6-13)
 - 4.1 Algoritmo de Ordenação Merge Sort
 - 4.2 Algoritmo de Ordenação Quick Sort
 - 4.3 Algoritmo de Ordenação Shell Sort
- 6 **Módulo de testes** (Pág 13-14)
- 7 **Análise dos algoritmos de ordenação** (Pág 15)
- 8 **Análise dos resultados** (Pág 15-22)
 - 8.1 Ambiente computacional
 - 8.2 Tabelas e gráficos
- 9 **Conclusões** (Pág 23)
- 10 **Referências** (Pág 23-24)

Comentários

1) Primeiramente é importante avisar que inicialmente pensamos em fazer o relatório em LaTeX, porém, como o relatório é uma tarefa que todos devem participar, acabou que achamos essa alternativa inviável pois nem todos membros do grupo eram familiarizados com esse editor de texto. Atrelado a esse problema de linguagem, tivemos também a questão de ter um período remoto que acaba sendo muito complicado com relação ao tempo, então decidimos optar pela utilização do Google Docs, tentando ao máximo manter a qualidade do relatório.

2) Para a implementação do código foi utilizado o Visual Studio Code além das extensões *IntelliSense* e *CodeSnap*. Usamos esta para printar as linhas de código que utilizamos no relatório além das linguagens C/C++ . Para os gráficos utilizamos o site Canva.

3) Utilizamos também na estruturação do nosso código várias pastas diferentes, cada uma com uma classe, para que o código pudesse ficar o mais organizado possível.

4) As fotos dos códigos das respectivas funções *StatisticalAnalysis()* e *PrintStatistic()* não foram colocados na parte de análise dos algoritmos de ordenação pois o tamanho da função *StatisticalAnalysis()* ficou muito grande, e como não seria possível colocá-la, também não colocamos a função *PrintStatistic()*.

1) Introdução

Este relatório é destinado para o primeiro trabalho da disciplina de Estrutura de Dados II, que teve como objetivo analisar e comparar o desempenho de diferentes algoritmos de ordenação aplicados a um conjunto de dados reais. Diante disso fomos capazes de:

- Manipular arquivos de texto;
- Compreender e implementar algoritmos de ordenação;
- Analisar o desempenho dos algoritmos implementados;

2) Atividades realizadas por cada membro

O trabalho foi realizado com uma média de duas a três reuniões por semana utilizando o programa de chamadas “Discord”, onde alguém compartilhava a tela e com isso eram feitas, juntos, algumas funcionalidades do código. Também utilizamos o aplicativo de mensagens “Whatsapp” para a troca de conhecimentos e ideias.

Além das atividades feitas em grupo, como por exemplo a divisão das classes e estruturas utilizadas no trabalho, decidimos dividir outras tarefas que restavam para cada membro do grupo, de forma que todos tivessem, no fim, um impacto significativo na realização do trabalho. A divisão quanto a implementação foi a seguinte:

- **Bryan:** Implementação do algoritmo Quicksort.
- **Gabriel:** Análise de algoritmos de ordenação, implementação do algoritmo Shellsort, comentários e formatação.
- **Matheus:** Pré-processamento dos dados, módulo de testes, comentários e formatação.
- **Pedro:** Análise de algoritmos de ordenação, implementação do algoritmo MergeSort, comentários.

A divisão quanto ao relatório foi a seguinte:

- **Bryan:** Revisão, quicksort, referências, gráficos para análise dos resultados e atividades realizadas por cada membro.
- **Gabriel:** Revisão, comentários, introdução, análise de algoritmos de ordenação, análise dos resultados, estruturação do relatório, atividades realizadas por cada membro, algoritmos de ordenação, shellsort, referências, tabelas e conclusão.
- **Matheus:** Revisão, pré-processamento dos dados, módulo de testes, estruturas de dados, análise dos resultados, atividades realizadas por cada membro, referências, conclusão.
- **Pedro:** Revisão, introdução, atividades realizadas por cada membro, análise de algoritmos de ordenação, referências, análise de resultados e implementação do algoritmo MergeSort.

3) Estruturas de dados utilizadas

Foram feitas algumas reuniões e pesquisas sobre quais estruturas de dados seriam melhores para o projeto. Com base nas referências encontradas, optamos por estruturar nosso código por meio de várias classes onde definimos em cada uma delas sua funcionalidade.

Para o armazenamento após o processamento do arquivo em formato csv criamos uma classe chamada *CovidInfo*, a qual é responsável por armazenar os dados de cada linha do csv. Diante disso, para salvar todo o conjunto dos dados processados foi utilizado o *Vector* de *CovidInfo*. O vector é uma implementação para um array dinâmico que consta com várias funções prontas que aceleram o desenvolvimento.

Entre os motivos para sua escolha está o fato de que ele é uma excelente e otimizada implementação do array dinâmico. Posto a isso, alguns dos algoritmos que foram implementados precisam acessar posições randômicas da estrutura de dados, como é o caso do Shell Sort e do Quick Sort.

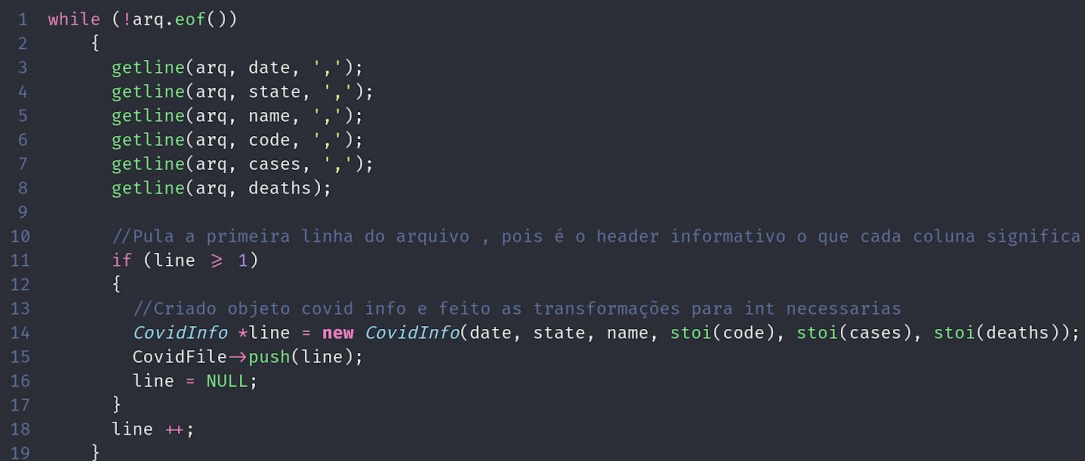
Com base nesses fatos previamente apontados, o *vector* foi uma escolha certa, pois mesmo que o custo de adicionar um elemento e deletar seja linear, a operação de acesso a posições aleatórias é constante. Por conta disso, optamos por utilizar um array dinâmico e não uma Lista Encadeada, cujo custo para acessar um elemento é linear.

```
class CovidInfo
{
private:
public:
    string date;
    string state;
    string city;
    float code;
    int cases;
    int deaths;
    int totalCases;
    CovidInfo();
    CovidInfo(string date, string state, string city, int code, int cases, int deaths);
    CovidInfo(string date, string state, string city, int code, int cases, int totalCases, int deaths);
    ~CovidInfo();
}
```

Figura 1: (Classe para armazenamento de cada linha do csv)

4) Processamento dos arquivos

Foi utilizada a biblioteca *fstream* para fazer o processamento do arquivo *brazil_covid19_cities.csv* e também o armazenamento na memória. Com essa biblioteca manipulamos o arquivo e usamos a função *getline()* para obter cada registro que estão separados por “,” ao final de cada linha. Em sequência, foi feita a transformação de cada elemento para seu tipo correspondente e depois o salvamento de cada linha obtida em um *vector CovidInfo*.

A screenshot of a code editor showing C++ code for processing a CSV file. The code uses `getline` to read each line of the file, splits it by commas, and converts the resulting strings to integers for cases and deaths. It also includes comments in Portuguese explaining the steps, such as skipping the header line and creating a `CovidInfo` object for each data row.

```
1 while (!arq.eof())
2 {
3     getline(arq, date, ',');
4     getline(arq, state, ',');
5     getline(arq, name, ',');
6     getline(arq, code, ',');
7     getline(arq, cases, ',');
8     getline(arq, deaths);
9
10    //Pula a primeira linha do arquivo , pois é o header informativo o que cada coluna significa
11    if (line ≥ 1)
12    {
13        //Criado objeto covid info e feito as transformações para int necessarias
14        CovidInfo *line = new CovidInfo(date, state, name, stoi(code), stoi(cases), stoi(deaths));
15        CovidFile->push(line);
16        line = NULL;
17    }
18    line ++;
19 }
```

Figura 2: (Código do processamento do arquivo)

Para gerar o arquivo *brazil_covid_19_cities_processado.csv* no pré processamento, foi utilizado o algoritmo ShellSort para fazer a ordenação por (Cidade, Estado) e Data. Após o ordenação, o mesmo é salvo em um formato em que cada coluna representa um dado , sendo esses dados: *data, cidade, Estado, código, casos diários, casos totais e número de mortes*.

```
date,state,city,code,dailyCases,totalCases,deaths,
2020-03-27,AC,Acrelândia,120001,0,0,0,
2020-03-28,AC,Acrelândia,120001,0,0,0,
2020-03-29,AC,Acrelândia,120001,2,2,0,
2020-03-30,AC,Acrelândia,120001,4,6,0,
2020-03-31,AC,Acrelândia,120001,1,7,0,
2020-04-01,AC,Acrelândia,120001,1,8,0,
2020-04-02,AC,Acrelândia,120001,0,8,0,
2020-04-03,AC,Acrelândia,120001,1,9,0,
2020-04-04,AC,Acrelândia,120001,0,9,0,
2020-04-05,AC,Acrelândia,120001,0,9,0,
```

Figura 3: (Csv gerado após o pre-processamento)

Para o processamento de arquivos foi utilizado o csv disponibilizado no site Kaggle por *FONTES[1]*. O arquivo é dividido em linhas, sendo que cada linha representa os dados de infecção de Covid19 em uma cidade em uma determinada data.

5) Algoritmos de ordenação

Os algoritmos de ordenação são métodos utilizados para ordenar dados. Foram criados para facilitar a organização de informações (Listas telefônicas, dicionários, contas bancárias, etc) que caso não estivessem ordenadas seriam extremamente difíceis de serem utilizadas.

A ordenação de dados pode ser feita de vários modos distintos, porém nem todos são eficientes. Para um tipo de ordenação ser significativo, deve-se levar em consideração diversos fatores e os principais são: Comparações, movimentações e tempo.

Temos atualmente os chamados métodos simples e métodos eficientes. O primeiro, respectivamente, é ideal para ordenar vetores pequenos, por serem métodos com implementações pequenas e fáceis de entender. Os algoritmos do método simples, possuem complexidade $C(n) = O(n^2)$, ou seja, requerem $O(n^2)$ comparações. Alguns exemplos desses algoritmos simples são:

- Insertion sort;
- Selection sort;
- Bubble sort;
- Comb sort;

Já os métodos eficientes são mais complexos e requerem um menor número de comparações, sendo assim ideais para trabalhar com grande quantidades de dados. Os algoritmos do método eficiente possuem complexidade $C(n) = O(n \log n)$. Alguns exemplos desses algoritmos eficientes são:

- Quick sort;
- Merge sort;
- Shell sort;
- Heapsort;

5.1) Algoritmo de ordenação Merge Sort

Um dos algoritmos escolhidos para fazer a ordenação é o Merge Sort. A escolha do MergeSort foi feita após algumas pesquisas e encontramos alguns pontos interessantes sobre ele:


- MergeSort é $O(n \log n)$;
- Indicado para aplicações que têm restrição de tempo;
- Fácil implementação;
- É estável;

Com base nos pontos citados acima chegamos a conclusão que o MergeSort seria uma boa escolha, já que não existe a possibilidade dos dados mudarem de posição ao fazer a ordenação e, durante a intercalação, os elementos com a mesma chave não terão sua posição trocada. Diante disso, o MergeSort executa os seguintes passos para sua execução:

- 1 - Dividir os dados, recursivamente, em subsequências, onde o vetor é dividido até sobrar um vetor com um elemento cada;
- 2 - Comparar duas metades recursivamente;

3 - Fazer a junção dos elementos;

Para a nossa implementação do MergeSort foram feitas poucas mudanças. Uma delas foi no vetor passado como parâmetro, onde usamos o vetor *CovidInfo* ao invés de um vetor com um tipo convencional para fazer a ordenação pelo número de casos. Além desta, foram acrescentadas duas variáveis para contabilizar o número de comparações e trocas entre os elementos do vetor



```
1  ///Funcao recursiva
2  void Sorting::mergeSortCases(vector<CovidInfo> &covidInfoList, int p, int r, int &comparisons, int &swaps)
3  {
4      if (p < r - 1)
5      {
6          int q = (p + r) / 2;
7          mergeSortCases(covidInfoList, p, q, comparisons, swaps); //chama de p a q
8          mergeSortCases(covidInfoList, q, r, comparisons, swaps); //chama de q a r
9          mergeCases(covidInfoList, p, q, r, comparisons, swaps);
10     }
11 }
12
```

Figura 3: (MergeSort)

```

1 void Sorting::mergeCases(vector<CovidInfo> &covidInfoList, int p, int q, int r, int &comparisons, int &swaps)
2 {
3
4     vector<CovidInfo> auxCovidInfoList;
5     int i = p;
6     int j = q;
7
8     while (i < q && j < r)
9     {
10         comparisons++;
11
12         if (covidInfoList.at(i).totalCases < covidInfoList.at(j).totalCases)
13         {
14             auxCovidInfoList.push_back(covidInfoList.at(i));
15             i++;
16             swaps++;
17         }
18         else
19         {
20             auxCovidInfoList.push_back(covidInfoList.at(j));
21             j++;
22             swaps++;
23         }
24     }
25
26     while (i < q)
27     {
28         auxCovidInfoList.push_back(covidInfoList.at(i));
29         i++;
30         swaps++;
31     }
32
33     while (j < r)
34     {
35         auxCovidInfoList.push_back(covidInfoList.at(j));
36         j++;
37         swaps++;
38     }
39
40     for (i = p; i < r; i++)
41     {
42         covidInfoList.at(i) = auxCovidInfoList.at(i - p);
43     }
44 }

```

Figura 4: (Função merge)

5.2) Algoritmo de ordenação Quick Sort

O QuickSort, em seu processo de ordenação, trabalha por meio da divisão das entradas contidas em uma estrutura em duas partes: menores e maiores que um elemento pivô escolhido, funcionando na seguinte sequência:

1. Um elemento pivô ($E[k]$) é escolhido na estrutura E através de uma dada estratégia de pivoteamento, podendo ser das mais simples (escolher a última entrada da estrutura) até as complexas (como heurísticas evolutivas e algoritmos de aprendizado de máquina para a tomada da decisão).
2. Com o pivô escolhido, o algoritmo move elementos de maneira que elementos maiores fiquem à direita (seção que chamaremos intuitivamente de E') e elementos menores à esquerda do pivô

(seção que, também para estabelecer de uma associação intuitiva, chamaremos de E''), formando duas seções na estrutura, chamadas de partições. Cabe lembrar que, neste estágio, o algoritmo apenas dividiu elementos entre duas seções da estrutura, dentro das próprias seções estes permanecem desordenados entre si.

3. O passo seguinte são duas novas chamadas do mesmo algoritmo, tomando como parâmetro as duas seções formadas, respectivamente. O mesmo processo de escolha do pivô e movimentação de elementos para posições maiores ou menores em relação a do pivô ocorre. Assim, chamadas sucessivas são feitas, culminando na ordenação de toda a estrutura.

O Quicksort foi escolhido por sua boa classificação em publicações comparativas da área de computação em geral e, ao mesmo tempo, pela familiaridade dos membros do grupo com o algoritmo. Para a implementação do Quicksort foi desenvolvido um algoritmo que funciona da seguinte maneira:

1. Escolha de um elemento pivô tomando como base para a decisão a média entre a posição 0 e a posição final da estrutura (posição do último elemento);
2. As comparações de 'maior que' e 'menor que' das entradas em relação ao valor do pivô;
3. A troca das posições dos valores, quando atendido o critério;
4. E, por fim, chamadas recursivas com vistas a executar o processo repetidas vezes para cada uma das novas partições E' e E'' geradas dentro de E .

```

1 void Sorting::quicksortCases(vector<CovidInfo> &covidInfoList, int began, int end, i
  nt &comparisons, int &swaps)
2 {
3     int i, j;
4     CovidInfo aux;
5     CovidInfo pivot;
6     i = began;
7     j = end - 1;
8     //é feito um pivô
9     pivot = covidInfoList[(began + end) / 2];
10    while (i <= j)
11    {
12        comparisons++;
13        while (covidInfoList[i].totalCases < pivot.totalCases && i < end)
14        {
15            comparisons++;
16            i++;
17        }
18        comparisons++;
19        while (covidInfoList[j].totalCases > pivot.totalCases && j > began)
20        {
21            comparisons++;
22            j--;
23        }
24        if (i <= j)
25        {
26            //são realizadas as trocas
27            swaps++;
28            aux = covidInfoList[i];
29            covidInfoList[i] = covidInfoList[j];
30            covidInfoList[j] = aux;
31            i++;
32            j--;
33        }
34    }
35    if (j > began)
36        //é chamada a função novamente de forma recursiva
37        quicksortCases(covidInfoList, began, j + 1, comparisons, swaps);
38    if (i < end)
39        //é chamada a função novamente de forma recursiva
40        quicksortCases(covidInfoList, i, end, comparisons, swaps);
41 }

```

Figura 5: (Algoritmo de ordenação QuickSort)

5.3) Algoritmo de ordenação Shell Sort

Shellsort é um algoritmo eficiente, de complexidade $C(n) = O(n \log n)$, e é uma melhoria do Selection sort. O método de ordenação shell, utiliza um valor chamado gap, que é o distanciamento dos elementos da estrutura, e é com ele que os elementos serão rearranjados [5]. Foi feita a escolha do Shellsort porque é um método eficiente, com tempo de

processo relativamente rápido, é de fácil entendimento e sua implementação é simples.

Já para o nosso shellsort foi escolhido o algoritmo de Knuth [6] que utiliza uma sequência $h = h * 3 + 1$ ou $\{1, 4, 13, 40, 121, 364, 1093, 3280, \dots\}$, o desempenho desse modelo se mostrou mais eficiente que o shell padrão com $gap = n/2$ ou $n/2^i$.

A implementação feita recebe o vetor junto com o seu tamanho e irá incrementar os contadores de comparações e trocas. O código começa então com $h = 1$ sendo o primeiro elemento que irá logo depois iniciar a sequência dos gaps, que serão então utilizados com as iterações do while. Após a montagem da sequência de distanciamentos, teremos outros três loops, um dentro do anterior. O primeiro irá atualizar o valor de h até chegar no elemento inicial "1", o segundo irá comparar os elementos "i" vezes utilizando o gap h . O terceiro e último irá checar se os elementos serão ou não trocados, caso eles sejam trocados e caso o valor "j" continue maior que o valor "h", o loop se dará novamente em direção contrária até que o primeiro caso em que não haja troca aconteça.

As posições dos contadores de trocas e comparações são mostradas na Figura 3, a troca de valores é realizada somente dentro do while, já as comparações são feitas antes e também dentro do while.

```

1 void Sorting::shellSortCases(vector<CovidInfo> &covidInfoList, int n, int &comparisons, int &swaps)
2 {
3     int h = 1;
4     int i, j;
5
6     while (h < n)
7     {
8         //O h inicial é calculado
9         h = h * 3 + 1;
10    }
11
12    while (h > 1)
13    {
14        //O valor de h é atualizado
15        h /= 3;
16
17        for (i = h; i < n; i++)
18        {
19            CovidInfo aux = covidInfoList.at(i);
20            j = i;
21            // São efetuadas comparações entre elementos com distanciamento h
22            comparisons++;
23            while (j >= h && aux.totalCases < covidInfoList.at(j - h).totalCases)
24            {
25                //covidInfoList[j] irá receber o valor de covidInfoList[j-h]. Troca
26                //Exemplo: {1,3,2}, covidInfoList[j] = 2 irá receber o valor de covidInfoList[j-h]=3, ficando assim {1,3,3}.
27                covidInfoList.at(j) = covidInfoList.at(j - h);
28
29                //O valor de j é atualizado
30                j -= h;
31                swaps++;
32                comparisons++; //voltará para o loop e irá comparar novamente
33            }
34            //covidInfoList[j] irá receber o vetor auxiliar. No caso do exemplo {1,3,3}, tal que aux=2, teremos {1,2,3}. Troca
35            covidInfoList.at(j) = aux;
36            swaps++;
37        }
38    }
39 }

```

Figura 6: (Algoritmo de ordenação Shellsort adaptado)

6) Módulo de testes

Para a implementação do módulo de testes foi criada a classe *Testing*, onde esta é responsável pela chamada das funções necessárias para realizar os testes. Posto a isso, essa classe interage com outras classes do sistema além de responder aos comandos do usuário.

O Módulo de testes conta com as seguintes funcionalidades:

- Pré-processamento do arquivo;
- Seleção de n registros aleatórios;
- Selecionar um dos algoritmos de ordenação para ordenar a instância com N registros previamente selecionadas;
- Escrita do resultado do algoritmo de ordenação no console ou em arquivo txt dependendo da escolha do usuário;

```
-----
Processamento dos Dados
O Arquivo processado sera salvo no arquivo : brazil_covid19_cities_processado.csv
-----
Digite [1] para começar o processamento do arquivo csv
Digite [2] ir para o modulo de testes(Somente se ja tiver o arquivo pre-processado salvo)
Digite [3] para ir para o modulo de estatisticas(Somente se ja tiver o arquivo pre-processado salvo)
Digite [0] para sair do programa
-----
```

```
-----
2
-----
Selecione o numero de instâncias para teste
Obs: (deve ser um numero entre 1 e 1400000)
-----
1400000
-----
Selecione o algoritmo de ordenacao:
-----
Digite [1] para MergeSort;
Digite [2] para ShellSort;
Digite [3] para QuickSort;
-----
```

```
-----
3
-----
Ordenação Finalizada
-----
Tempo de Processamento : 1.38017 segundos
Número de Comparações : 32293712
Número de Trocas : 11301972
-----
Selecione a saida dos resultados de teste:
-----
Digite [1] para escrever a saida em um arquivo txt;
Digite [2] para escrever a saida no console;
-----
```

Figura 7: (Fluxo do módulo de testes)

7) Análise dos algoritmos de ordenação

A análise dos algoritmos de ordenação foi pensado tendo como base o que foi especificado no trabalho. Foram utilizados registros randômicos de N tamanhos diferentes, 10.000, 50.000, 100.000, 500.000 e 1.000.000. E para cada valor de N foi gerado M conjuntos de registros randomizados que foram submetidos a três algoritmos de ordenação distintos.

Como a intenção era comparar os resultados obtidos pelos métodos de ordenação, pensamos em rodar os algoritmos para o mesmo registro randômico gerado em cada iteração, de forma que teríamos uma melhor comparação caso eles fossem submetidos aos mesmos casos.

As funções *StatisticalAnalysis()* e *PrintStatistics()* são responsáveis pela análise dos resultados dos algoritmos e estão com o modo de funcionamento bem detalhado no código de implementação. Para essas funções foi tentado ao máximo otimizar o código, porém, não encontramos maneira de melhorar mais do que foi implementado.

8) Análise dos resultados

8.1) Ambiente computacional

Os testes foram realizados em um sistema com as especificações da Tabela 1.

Sistema Operacional	LINUX MINT - Cinnamon 20
Processador	INTEL CORE 7 i7-9750H
Placa de Vídeo	NVIDIA GEFORCE GTX 1660 TI
Memória Ram	32GB DDR4
SSD	250GB

Versão Compilador	g++ 9.3.0
-------------------	-----------

Tabela 1: Especificações do sistema e ambiente computacional.

8.2) Tabelas e Gráficos

Os testes foram realizados para três “M” diferentes e uma vez para cada valor de M. Pensamos que como já se tratava de uma média, ao fazer diversas vezes para um mesmo valor estaríamos contrariando a proposta do aumento da quantidade de conjuntos diferentes.

É esperado que quanto maior for a quantidade de “M” conjuntos diferentes mais preciso será o resultado. Escolhemos os valores 5 , 25 e 50, mostrados respectivamente nas Tabelas 2,3 e 4 e também nos Gráficos 1(a,b,c), 2(a,b,c) e 3(a,b,c), pois achamos que seriam valores de boa qualidade e cumpririam com o que foi demandado pelo projeto.

Média de comparações para M=5									
Valores de N	Tempo(s)			Comparações			Trocas		
	Quicksort	Mergesort	Shellsort	Quicksort	Mergesort	Shellsort	Quicksort	Mergesort	Shellsort
10.000	0.0055478	0.0329136	0.01246	167045	120515	200190	44962	133601	200190
50.000	0.0359108	0.200486	0.106434	946667	718533	1.43073e+06	270283	784447	1.43073e+06
100.000	0.0728714	0.419961	0.191129	1.99432e+06	1.53704e+06	2.86172e+06	603676	1.66891e+06	2.86172e+06
500.000	0.426823	2.12363	1.13793	1.08664e+07	8.83702e+06	1.60751e+07	3.6712e+06	9.47569e+06	1.60751e+07
1.000.000	1.03736	5.86016	3.02043	2.33412e+07	1.86751e+07	3.48224e+07	7.82297e+06	1.99514e+07	3.48224e+07

Tabela 1: Tabela para média de comparações de 5 conjuntos diferentes.

Média de comparações para M=25									
Valores de N	Tempo(s)			Comparações			Trocas		
	Quicksort	Mergesort	Shellsort	Quicksort	Mergesort	Shellsort	Quicksort	Mergesort	Shellsort
10.000	0.00628212	0.0379688	0.106447	167650	120470	206428	44887	133601	206428
50.000	0.0390516	0.232163	0.106447	965529	718439	1.36924e+06	269482	784447	1.36924e+06
100.000	0.0801559	0.458816	0.218301	1.98742e+06	1.53707e+06	2.83866e+06	604706	1.66891e+06	2.83866e+06
500.000	0.458995	2.43062	1.261	1.09104e+07	8.83748e+06	1.64542e+07	3.6684e+06	9.47569e+06	1.64542e+07
1.000.000	0.993172	5.55842	2.85312	2.30539e+07	1.86749e+07	3.52147e+07	7.84321e+06	1.99514e+07	3.52147e+07

Tabela 2: Tabela para média de comparações de 25 conjuntos diferentes.

Média de comparações para M=50									
Valores de N	Tempo(s)			Comparações			Trocas		
	Quicksort	Mergesort	Shellsort	Quicksort	Mergesort	Shellsort	Quicksort	Mergesort	Shellsort
10.000	0.00589838	0.0373703	0.0137611	167866	120471	204040	44831	133601	204040
50.000	0.0357953	0.214463	0.0945004	960827	718409	1.37028e+06	269811	784447	1.37028e+06
100.000	0.0781458	0.448707	0.208436	1.9972e+06	1.53716e+06	2.86458e+06	603865	1.66891e+06	2.86458e+06
500.000	0.484015	2.5409	1.32493	1.09432e+07	8.8375e+06	1.62478e+07	3.665e+06	9.47569e+06	1.62478e+07
1.000.000	1.03248	5.7848	2.92429	2.29592e+07	1.8675e+07	3.4713e+07	7.84222e+06	1.99514e+07	3.4713e+07

Tabela 3: Tabela para média de comparações de 50 conjuntos diferentes.

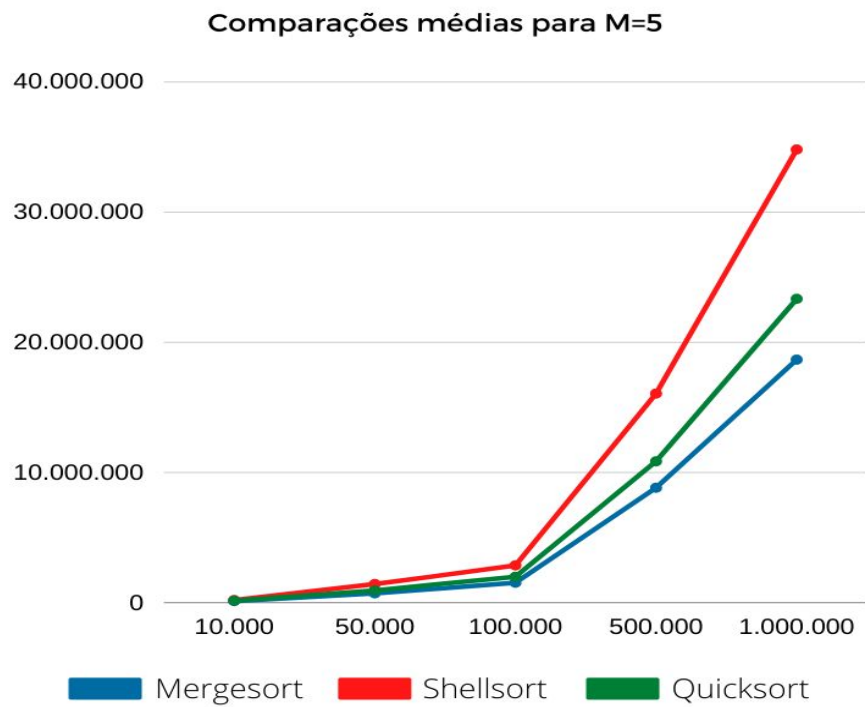


Gráfico 1a: Comparações médias.

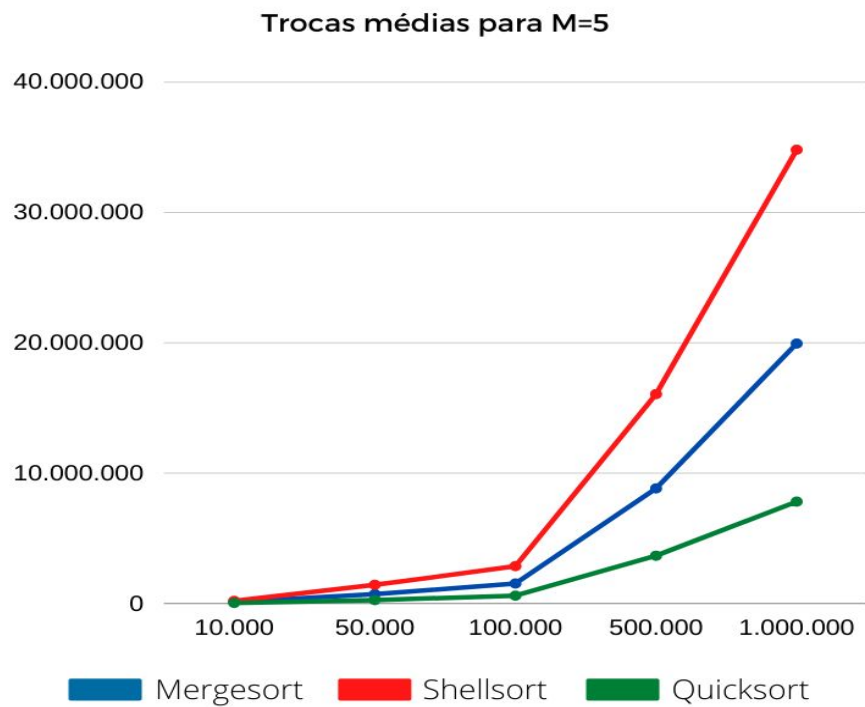


Gráfico 1b. Trocas médias

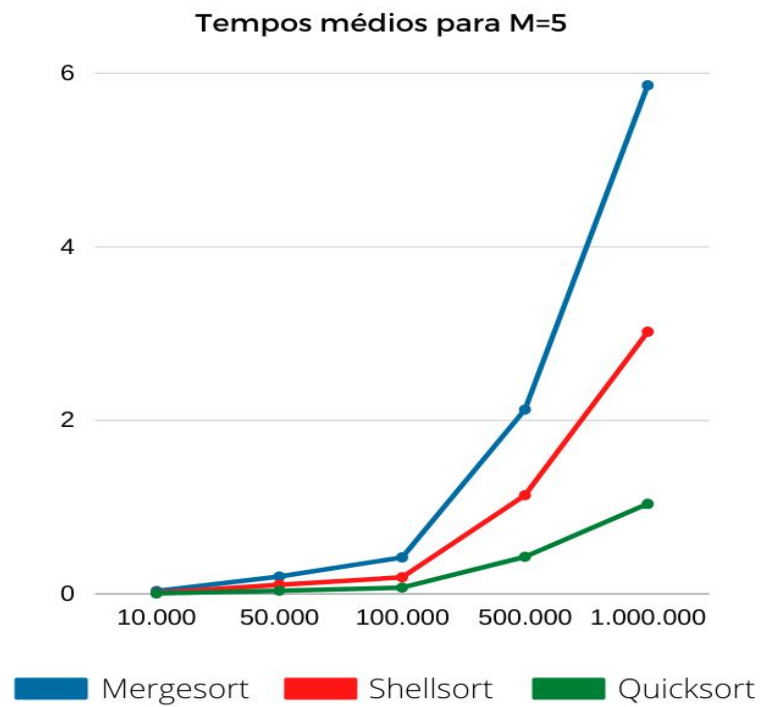


Gráfico 1c. Tempos médio

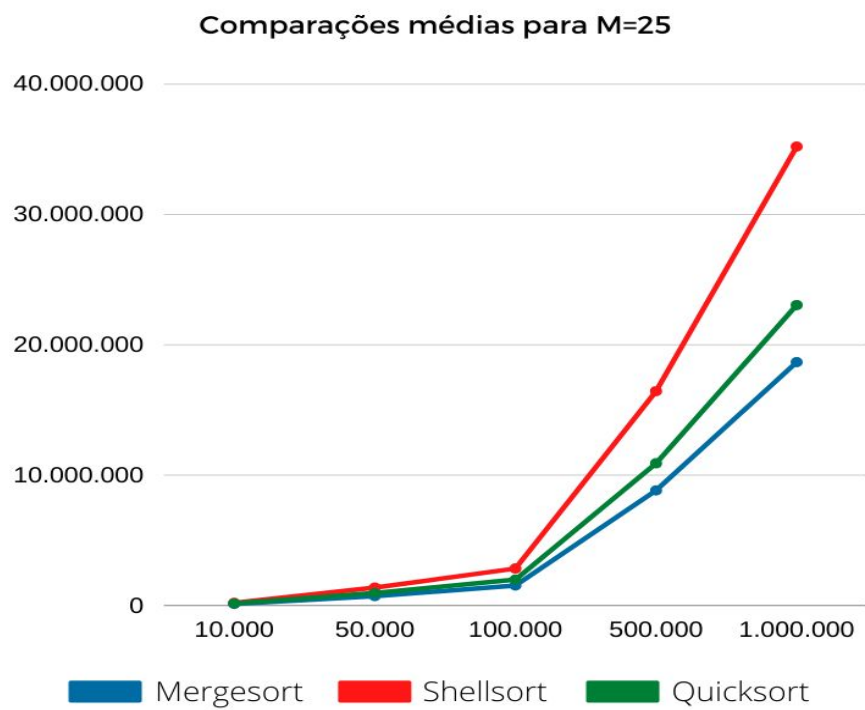


Gráfico 2a: Médias de comparações.

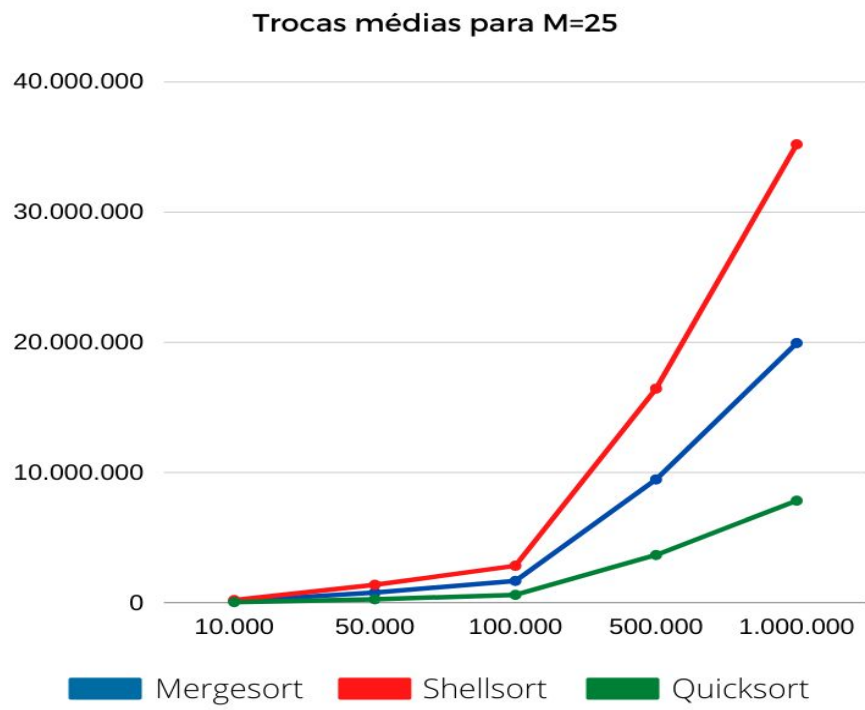


Gráfico 2b: Trocas médias

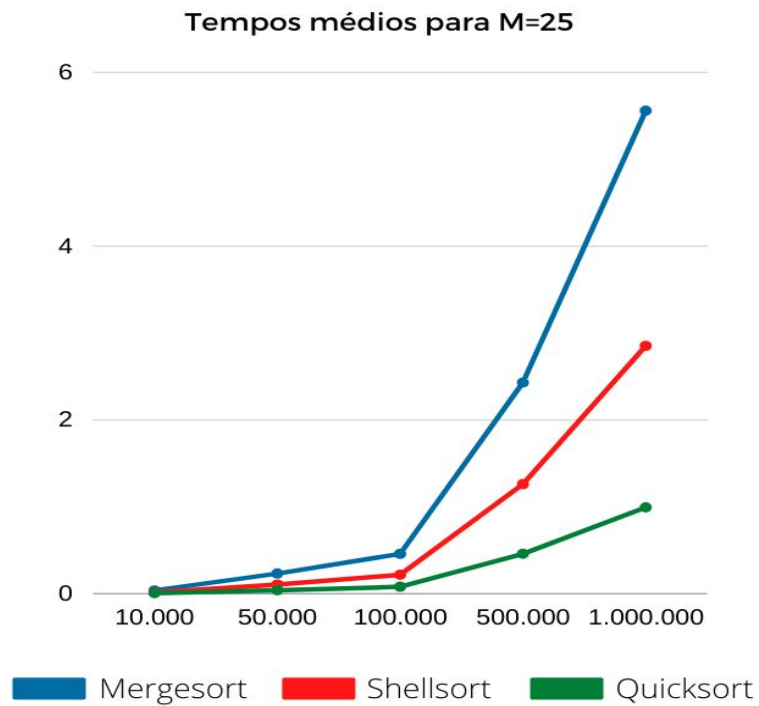


Gráfico 2c: Tempos médios

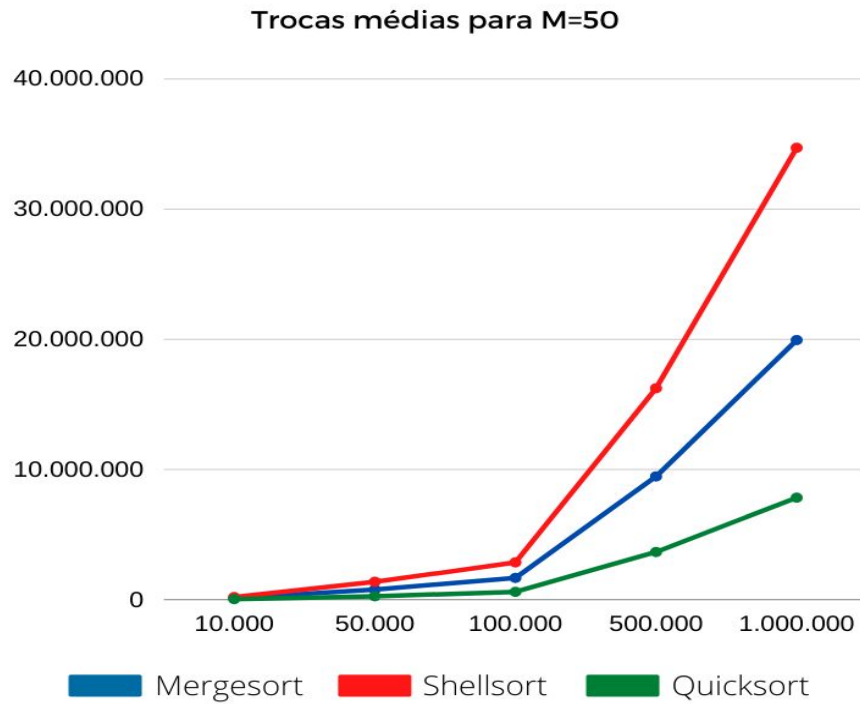


Gráfico 3a: Trocas Médias

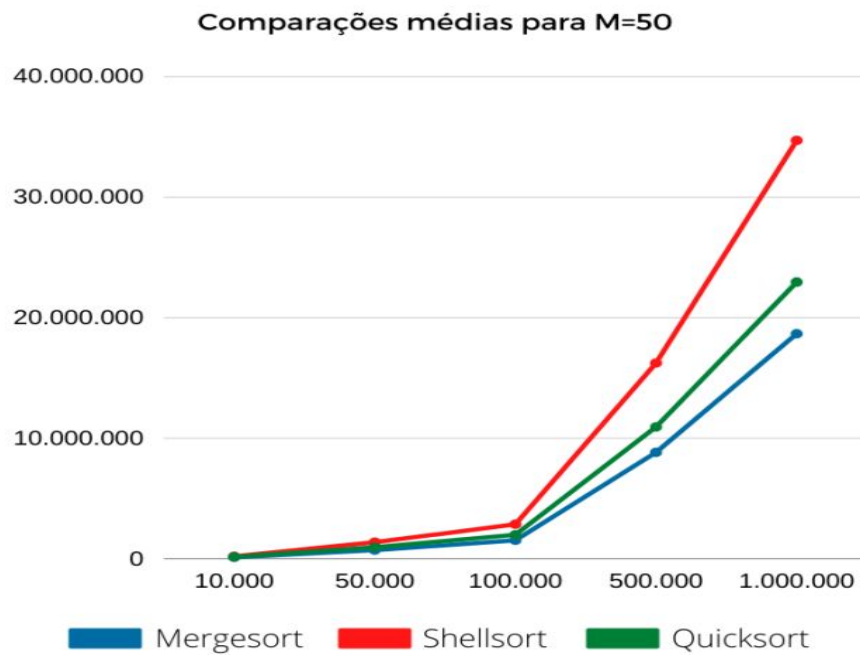


Gráfico 3b: Comparações Médias

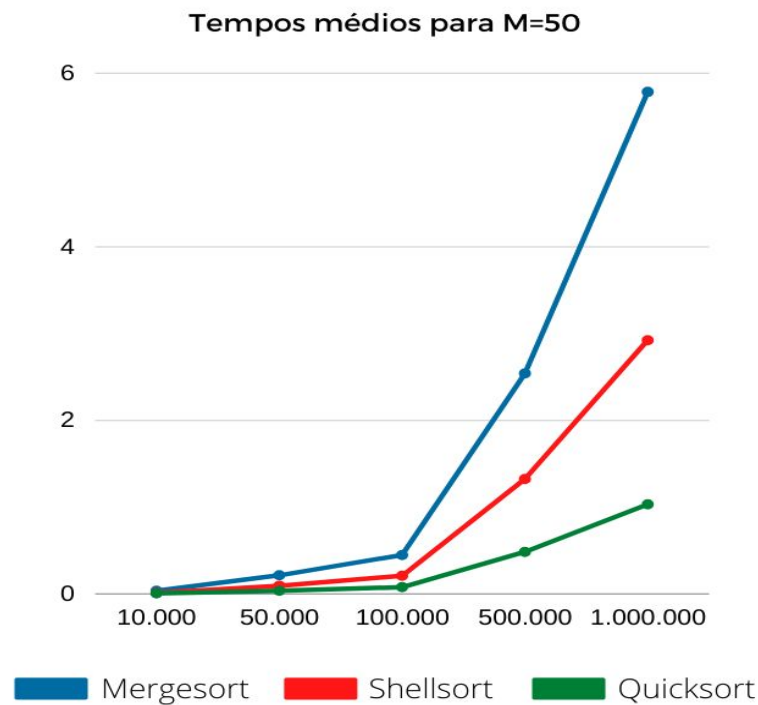


Gráfico 3c: Tempos médios

8.2) Análise das tabelas

Com os gráficos e tabelas podemos observar que os valores dos dados analisados não se diferem tanto à medida que o M aumenta. Este resultado já era esperado pois quanto mais elementos distintos diferentes, mais próximo da média final.

Quanto a cada algoritmo, o Quicksort foi o melhor em tempo para todos os valores distintos de M. Além disso, este teve o menor número de trocas e foi o segundo maior em número de comparações. O Shellsort foi o algoritmo que realizou mais comparações e trocas, e foi o segundo melhor método em questão de tempo. O Mergesort foi o algoritmo que apesar de ter tido a pior performance em questão de tempo, foi o algoritmo que obteve menos trocas e o segundo maior em número de comparações.

9) Conclusões

Após os testes e análises dos métodos de ordenação utilizados, concluímos que, como esperado, o Quicksort foi o algoritmo mais rápido em tempo de execução dentre os analisados. Ao subdividir o vetor e fazer inserções a partir de comparações com o pivô, o Quicksort garante um ótimo tempo de execução. Vale ressaltar que o mesmo fez o menor número de trocas, sendo esta uma operação custosa. Levando em consideração um cenário onde a instância a ser ordenada é muito grande e você precisa realizar a ordenação no menor tempo possível, o Quicksort acaba sendo uma excelente opção.

O Shellsort obteve um resultado mediano. O mesmo foi mais rápido que o Mergesort, mesmo sendo o algoritmo que mais fez trocas e comparações. Isso se deve pelo fato do ShellSort não usar chamadas recursivas como o Merge Sort, sendo que essa chamada tem um custo computacional alto. Logo, ele consegue ter um melhor desempenho que o Mergesort, mesmo fazendo mais trocas e comparações.

O Merge Sort apresentou-se como um algoritmo pior que o esperado pelos membros do grupo. O mesmo apresentou um tempo pior que o do ShellSort, devido ao seu principal mecanismo ser a recursividade e seu número de trocas ser muito alto. Além disso, o Mergesort faz uso de uma estrutura de dados auxiliar, que pode ser uma grande desvantagem em algumas situações. Logo, concluímos que o Shellsort ainda apresenta um grande avanço quando comparado a algoritmos de complexidades n^2 , visto que ele é considerado uma melhoria do Selection Sort. Por conta disso, no uso do algoritmo deve-se analisar se os pontos citados na conclusão.

10) Referências

[1] FONTES, Rafael. Coronavírus (COVID 19) - Brazil Dataset. 2020. Disponível em: https://www.kaggle.com/unanimad/corona-virus-brazil?select=brazil_covid19_cities.csv (acessado em 08 de dezembro de 2020).

[2] DROZDEK, Adam. Estrutura de dados e algoritmos em C + +. 2. ed. São Paulo: Cengage Learning, 2016.

[3] VIANA, Daniel. Conheça os principais algoritmos de ordenação. Brasil. 26 de Dezembro, 2016. Disponível em: <https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao/> (acessado em 22 de janeiro de 2021).

[4]Wikipedia contributors, "Shellsort," *Wikipedia, The Free Encyclopedia*, <https://en.wikipedia.org/w/index.php?title=Shellsort&oldid=996646744> (accessed January 29, 2021).

[5] Souza, J. É. G., Ricarte, J. V. G. Lima, N. C. A. Algoritmos de Ordenação: Um estudo comparativo. In: ENCONTRO DE COMPUTAÇÃO DO OESTE POTIGUAR. 2017, Pau de Ferro, p. 169-170.

[6] Knuth, Donald E. "Sorting and searching." (1973).

[7] Toffolo, Túlio. Ordenação: Merge Sort [Ordenação: MergeSort](#).

[8] Wikipedia contributors, "Quicksort," *Wikipedia, The Free Encyclopedia*, <https://pt.wikipedia.org/wiki/Quicksort>.