**UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE**
**INSTITUTO METRÓPOLE DIGITAL**

Jonas de Oliveira Freire Filho

Neylane Pereira Lopes

# Análise Empírica de Algoritmos de Ordenação

NATAL, RN
2021

Jonas de Oliveira Freire Filho

Neylane Pereira Lopes

# Análise Empírica de Algoritmos de Ordenação de Ordenação

Relatório técnico apresentado a disciplina Estrutura de Dados Básica I para obtenção de nota parcial da primeira unidade do semestre letivo 2021.1 do curso Bacharelado em Tecnologia da Informação pela Universidade Federal do Rio Grande do Norte.

Orientador: Prof.ª Dr. Selan Rodrigues dos Santos

NATAL, RN
2021

**Sumário**

# 1 INTRODUÇÃO

Este presente relatório tem como objetivo realizar uma análise empírica para sete algoritmos de ordenação estudados na disciplina de Estrutura de Dados Básica I pela Universidade Federal do Rio Grande do Norte (UFRN). Esses algoritmos irão ser executados em seis cenários diferentes, isto é, os algoritmos vão ordenar os elementos de um conjunto de dados em seis situações diferentes, um possível cenário seria um *array* com elementos em posições aleatórias, ou elementos parcialmente ordenados.

Um algoritmo pode ser entendido como qualquer procedimento com um conjunto de etapas bem definidas para resolver um problema computacional, que recebe algum valor de entrada e produz uma saída correspondente (CORMEN et al., 2012). Desse modo, os algoritmos de ordenação possuem um conjunto de etapas bem definidas para colocar os elementos de, por exemplo: um *array*, em uma ordem específica. Sendo assim, o *array* 'desordenado' é o valor de entrada e o *array* ordenado o valor de saída. E o problema computacional seria ordenar os elementos em uma ordem específica dado um conjunto de dados ordenáveis.

As etapas bem definidas que cada algoritmo possui cativa uma série de dúvidas, duas delas são: qual algoritmo é o mais eficiente? Será que diferentes algoritmos podem ter diferentes desempenhos em distintas situações, ou cenários? Pensando nisso, ao final desse relatório a equipe pretende ser capaz de responder esse e outros questionamentos orientado aos dados coletados e os estudos realizados sobre os principais algoritmos de ordenações.

Para analisar um algoritmo, e compará-los é necessário definir o que seria um algoritmo melhor que outro. Para este relatório, entende-se que o algoritmo mais rápido é o que tem melhor desempenho, ou seja, aquele que tiver o menor tempo de execução. Mas retifica-se que exitem outras análises de desempenho, como a quantidade de memória usada pelo algoritmo.

Os algoritmos de ordenação estudados em sala de aula remota, foram: *bubble sort*, *insertion sort*, *selection sort*, *shell sort*, *quick sort*, *merge sort* e *radix sort*. Pretende-se implementar cada um deles e executá-los em seis cenários sugeridos pelo docente orientador deste relatório, são eles: elementos em ordem Não Decrescente, Não Crescente, Aleatório, $25\%$ em Posição Definitiva, $50\%$ em Posição Definitiva e $75\%$ em Posição Definitiva.

Nas seções seguintes, vai ser explicado todo o processo de implementação do projeto, vai ser mostrado os dados coletados e os gráficos gerados e, por último, os resultados obtidos serão comentados e discutidos.

# 2 MÉTODO

Essa seção tem como objetivo explicar minuciosamente todo o processo de criação dos algoritmos e do arquivo executável, além de descrever todas as informações técnicas relacionadas a infraestrutura, ambiente, e a geração dos gráficos.

## 2.1 Materiais utilizados

### 2.1.1 Ambiente de Execução

Para realizar a análise foi necessário definir um ambiente de execução estável, ou, pelo menos, o ambiente menos instável de acesso totalmente gratuito. Este ambiente foi o Google Colaboratory, que nos permitiu acesso fácil e gratuito a GPUs para gerar os dados de cada cenário. O Colaboratory, ou simplesmente 'colab' disponibiliza um Jupyter Notebook em uma máquina virtual com uma quantidade de CPU e de memória estáveis, ou algo próximo disso. Apesar do Jupyter Notebook somente permitir a execução de códigos em Python $2$ ou $3$, a maquina virtual do colab usufrui de um sistema operacional com kernel linux, o que permitiu executar arquivos executáveis pelo notebook, ou seja, programas gerados em c++.

As demais informações técnicas do ambiente não é disponibilizada pela empresa, que afirma que a quantidade de recurso disponibilizado para cada máquina virtual depende da quantidade total de recursos não alocados no momento de conectar ao ambiente de execução.

### 2.1.2 Ferramentas e Linguagem de Programação

A linguagem de programação usada para implementar todos os algoritmos e todos os cenários, bem como o programa que gera os dados foi a linguagem c++$17$. E para o tratamento dos dados e para a geração dos gráficos a linguagem usada foi Python $3$.

## 2.2 Algoritmos de Execução

Foram estudados e criados $6$ algoritmos de ordenação, são eles: *bubble sort*, *insertion sort*, *selection sort*, *shell sort*, *quick sort*, *merge sort* e *radix sort*. Segue a explicação de cada um deles.

- ***Bubble Sort*** - Consiste na ideia de percorrer o vetor diversas vezes até que os dados estejam totalmente ordenados, de modo que a cada passagem flutua para o topo o maior elemento da sequência. Assim, cada um dos elementos do vetor é comparado ao seu elemento adjacente e são trocados se eles estiverem

na ordem errada, ou seja, se o primeiro elemento é maior que o segundo. Sua complexidade é $O(n^2)$.

- ***Insertion Sort*** - Funciona de maneira semelhante à ordenar as cartas de um baralho. Considerando que as cartas já estão ordenadas, então ao receber uma nova carta, devemos colocá-la na posição correta, de forma que as cartas obedeçam à ordenação. Desse modo, devemos comparar a nova carta com todas as outras cartas e se ela for maior do que a carta comparada, ela é colocada à direita, caso contrário, à esquerda. Da mesma forma, outras cartas são colocadas em seus devidos lugares.

  Portanto, na ordenação por inserção, cria-se dois subvetores, o da esquerda com elementos ordenados (inicialmente com um único elemento) e o da direita com os demais elementos. Desse modo, começamos a partir do segundo elemento e comparamos com o primeiro elemento, e depois o colocamos em um lugar correto do subvetor ordenado à esquerda daquela posição. Em seguida, realizamos esse processo para os elementos subsequentes. Sua complexidade é $O(n^2)$.

- ***Selection Sort*** - O algoritmo, que também usufrui da estratégia de criar dois subvetores, consiste em encontrar o menor elemento do subvetor não ordenado e coloca-lo na parte ordenada. Desse modo, a cada repetição, até os últimos dois elementos do vetor, o menor elemento do subvetor não ordenado é selecionado e movido para o subvetor ordenado. Sua complexidade é $O(n^2)$.

- ***Shell Sort*** - O método começa ordenando elementos distantes entre si e reduz progressivamente a distancia entre os elementos a serem comparados. Começando com elementos distantes, ele pode mover alguns elementos fora do lugar para a posição correta mais rapidamente do que uma simples troca de vizinho próximo. Sua complexidade é $O(n^2)$.

- ***Merge Sort*** - A estratégia adotada por esse algoritmo é divisão e conquista. Sua ideia básica é dividir um problema em vários subproblemas e resolve-las recursivamente. Quando um subproblema é resolvido, o algoritmo faz a união de seus resultados até a solução total. Assim, o *merge sort* divide o vetor de entrada em dois e depois chama recursivamente a si mesmo para cada metade, subdividindo-as em duas novas partes até ter um vetor com apenas um único elemento e, em seguida, une cada parte ordenando-as. Sua complexidade é $O(n \log n)$.

- ***Quick Sort*** - O algoritmo, que também usufrui da estratégia de divisão e conquista, seleciona um elemento, que é escolhido pelas medianas-dos-três, e particiona o vetor fornecido com base nele. De modo que, todos os elementos menores que o pivô (elemento escolhido) fique antes dele, e todos os elementos

maiores depois. Em seguida, os dois subvetores são ordenados recursivamente até que o vetor completo se encontre ordenado. Sua complexidade é $O(n \log n)$.

- ***Radix Sort*** - É um algoritmo de ordenação não comparativo. A ideia é fazer a classificação dígito a dígito, começando do dígito menos significativo ao dígito mais significativo. Assim, ao final, os elementos estarão ordenados. Sua complexidade é $O(nk)$.

Desse modo, a ordem dos algoritmos de ordenação de menor complexidade para os maior complexidade é: *radix sort*, *merge sort*, *quick sort*, *shell sort*, *insertion sort*, *selection sort* e *bubble sort*. Portanto, espera-se que os resultados dos testes realizados estejam de acordo com a complexidade de cada algoritmo.

## 2.3 Cenário dos *Arrays*

Foram estudados e criados sete cenários para executar os algoritmos de ordenação a fito de analisar e comparar profundamente o comportamento de cada algorítimo para cada cenário. Os cenários são: elementos em ordem Não Decrescente, Não Crescente, Aleatório, $25\%$ em Posição Definitiva, $50\%$ em Posição Definitiva e $75\%$ em Posição Definitiva. Segue a explicação de cada um deles.

- **Não Decrescente** - Neste cenário os elementos estão dispostos, tal que dado um elemento, o seu subsequente não é menor.
- **Não Crescente** - Neste cenário os elementos estão dispostos, tal que dado um elemento, o seu subsequente não é maior.
- **Aleatório** - Neste cenário os elementos estão dispostos de forma aleatória.
- $25\%$ **em Posição Definitiva** - Neste cenário $25\%$ dos elementos estão em sua posição original e os demais embaralhados.
- $50\%$ **em Posição Definitiva** - Neste cenário $50\%$ dos elementos estão em sua posição original e os demais embaralhados.
- $75\%$ **em Posição Definitiva** - Neste cenário $75\%$ dos elementos estão em sua posição original e os demais embaralhados.

## 2.4 Arquivo Executável

Para gerar os dados de cada cenário, foi criado um programa em c++ capaz de executar os testes de cada cenário. Este programa foi implementado para receber por parâmetro a quantidade de elementos inicial do vetor, a quantidade final de elementos

do vetor, a quantidade de amostras a serem coletadas, quais algoritmos de ordenação executar, quais cenários executar e a quantidade de execuções para cada amostra.

Para implementar este programa, foram criadas duas classes e todos os métodos necessários, uma para os algoritmos de ordenação e outra para os cenários. Além disso, uma *struct* para armazenar os dados da execução, isto é, os parâmetros que o programa vai receber. Desse modo, também foi criado uma função que para realizar a execução de tal forma que percorre cada cenário desejado para executar cada quantidade de elementos do array para cada algorítimo de execução e medir o tempo de ordenação $n$ vezes, tal que $n$ é a quantidade de execuções para cada amostra.

A biblioteca usada para mensurar o tempo foi a $std :: chrono$ e a unidade de tempo usada foi nanossegundos com duas casas decimais, a fito de obter maior precisão ao comparar os algoritmos.

## 2.5 Variáveis de Execução

Para executar o programa foi criado uma *struct* que armazenará todas as seis variáveis base para a execução do programa. Elas são responsáveis por definir os algoritmos, os cenários, a quantidade de amostras, além de outras variáveis. Segue a *struct* usada no programa para armazená-las.

Figura 1: *Struct* responsável por armazenar as variáveis de configuração.

```
struct RunningOpt{
    size_t min_sample_sz{100};     //!< Default 10^5.
    size_t max_sample_sz{100000}; //!< The max sample size.
    int n_samples{25};             //!< The number of samples to collect.
    short which_algs{1};           //!< Bit code for the chosen algorithms to run.
    short which_scenarios{1};      //!< Bit code for the chosen scenarios to run.
    short n_runs{5};               //!< Number of rounds for each size.

    size_type sample_step(void){
        return static_cast<float>(max_sample_sz-min_sample_sz)/(n_samples-1);
    }
};
```

Fonte: Autoria Própria

- **Quantidade de elementos inicial do *array*** - Como o programa executa os teste para várias quantidades de elementos detro de um *array*, essa variável determina a menor quantidade de elementos do *array* que vai ser ordenado pelos algoritmos.
- **Quantidade de elementos final do *array*** - Analogamente ao item anterior, essa variável determina a maior quantidade de elementos do *array* que vai ser ordenado pelos algoritmos.

- **Quantidade de amostras a serem coletadas** - Esta variável define a quantidade de diferentes tamanhos do *array* que os algoritmos vão ordenar. Em outras palavras, a quantidade de amostras que vão ser geradas. Por exemplo, se for atribuído o número 25 para essa variável, o programa irá executar os testes para $25$ tamanhos de *array* diferentes e igualmente espaçados.

- **Quais algoritmos de ordenação** - Define quais algoritmos de ordenação vão ser executados pelo programa. A metodologia adotada pela equipe para realizar essa definição foi monitorar os *bits* dessa variável, de forma que cada *bit* representa um algoritmo, e se este *bit* tiver o valor $1$ o algoritmo deverá ser executado, caso contrário, não deve ser executado. A atribuição dos *bits* para cada algoritmo se deu da seguinte forma: $2^0$ para o *insertion sort*, $2^1$ para o *selection sort*, $2^2$ para o *bubble sort*, $2^3$ para o *shell sort*, $2^4$ para o *quick sort*, $2^5$ para o *merge sort* e $2^6$ para o *radix sort*. Por exemplo, para executar todos os algoritmos pode-se atribuir o valor $127$, uma vez que sua representação binária é: $1111111$ e, como todos os *bits* são $1$ todos os algoritmos vão ser executados. Segue a implementação do *enum* para facilitar a atribuição dos valores.

Figura 2: *Enum* dos algoritmos.

```
enum algorithm_t {
            INSERTION = 1,
            SELECTION = 2,
            BUBBLE = 4,
            SHELL = 8,
            QUICK = 16,
            MERGE = 32,
            RADIX = 64,
            ALL_ALGORITHMS = 127,
        };
```

Fonte: Autoria Própria

- **Quais cenários** - Analogamente ao item anterior, essa variável define quais cenários serão executados pelo programa. A atribuição dos *bits* para cada cenário se deu da seguinte forma: $2^0$ para o não decrescente, $2^1$ para o não crescente, $2^2$ para o aleatório, $2^3$ para o $75\%$ em Posição Definitiva, $2^4$ para o $50\%$ em Posição Definitiva e $2^5$ para o $25\%$ em Posição Definitiva. Por exemplo, para executar todos os cenários pode-se atribuir o valor $63$, uma vez que sua representação binária é: $111111$ e, como todos os *bits* são $1$ todos os cenários vão ser executados. Segue a implementação do *enum* para facilitar a atribuição dos valores.

Figura 3: *Enum* dos cenários.

```
enum scenarios_t {
            NOTDECREASING = 1,
            NOTGROWING = 2,
            RANDOM = 4,
            _75PERINDEFINITIVEPOSITION = 8,
            _50PERINDEFINITIVEPOSITION = 16,
            _25PERINDEFINITIVEPOSITION = 32,
            ALL_SCENARIOS = 63,
        };
```

Fonte: Autoria Própria

- **Quantidade de execuções para cada amostra** - Define a quantidade de vezes que o algorítimo vai ordenar o *array* em um mesmo teste, ou seja, a quantidade de vezes que o algoritmo vai rodar para o mesmo array no mesmo cenário, possibilitando o cálculo do tempo médio de execução, a fito de reduzir possíveis instabilidades na coleta dos dados.

## 2.6 Execução do Programa

O programa criado consiste em calcular para cada cenário escolhido e para cada amostra diferente a média do tempo que cada algoritmo demora para ordenar o *array* para $n$ execuções. Segue uma das execuções realizadas pela equipe no colab.

Figura 4: Execução do cenário aleatório no colab.

```
%%script bash

./drive/MyDrive/colab/trabalho_02_edb_i/bin/prog 100 100000 25 127 4 5
```

Fonte: Autoria Própria

Nesta execução, a quantidade de elementos mínima do *array* é $100$, a quantidade máxima é $100000$ e a quantidade de amostras é $25$. Isto é, o programa vai executar os algoritmos para $25$ tamanhos de amostras diferentes que vão de $100$ elementos até $100000$ elementos, todos igualmente espaçados.

Note que o quarto parâmetro (número $127$) define qual(is) algoritmos(s) vão ser executados. Neste caso, serão todos, uma vez que todos os sete *bits* ($1111111$) estão com valor $1$. Já para os cenário, quinto parâmetro, somente o terceiro *bit* $000100$ é $1$ e

este refere-se exatamente ao cenário aleatório. Portanto, o programa vai executar o cenário aleatório para todos os sete algoritmo de ordenação.

O último parâmetro refere-se a quantidade de execuções que cada amostra vai ser executada para calcular a média dos tempos, que é o objeto de estudo deste relatório. Além dessa execução para o cenário aleatório, foram realizadas outras execuções para os demais cenários, somente alterando o quinto parâmetro.

## 2.7 Geração dos Gráficos

Tendo executado os cenários e gerado todos os dados, a equipe importou os *dataframes* no ambiente *notebook* colab para iniciar o processo de criação dos gráficos. Após tratar os dados e renomear as colunas dos *dataframes*, os gráficos de cada cenário foram criados, sendo um para os algoritmos *shell sort*, *insertion sort*, *selection sort* e *bubble sort*, que serão chamados de $O(n^2)$ e outro para os algoritmos *radix sort*, *merge sort* e *quick sort*, que serão chamados de $O(n \log n)$. Desse modo, foram criados um total de doze gráficos.

# 3 RESULTADOS

Essa seção tem como objetivo exibir e explicar os resultados coletados, mostrando os gráficos e tabela de cada um dos cenários.

## 3.1 Resultados do Cenário Não Decrescente

Segue uma amostra dos tempos de execução de cada algoritmo para o cenário não decrescente.

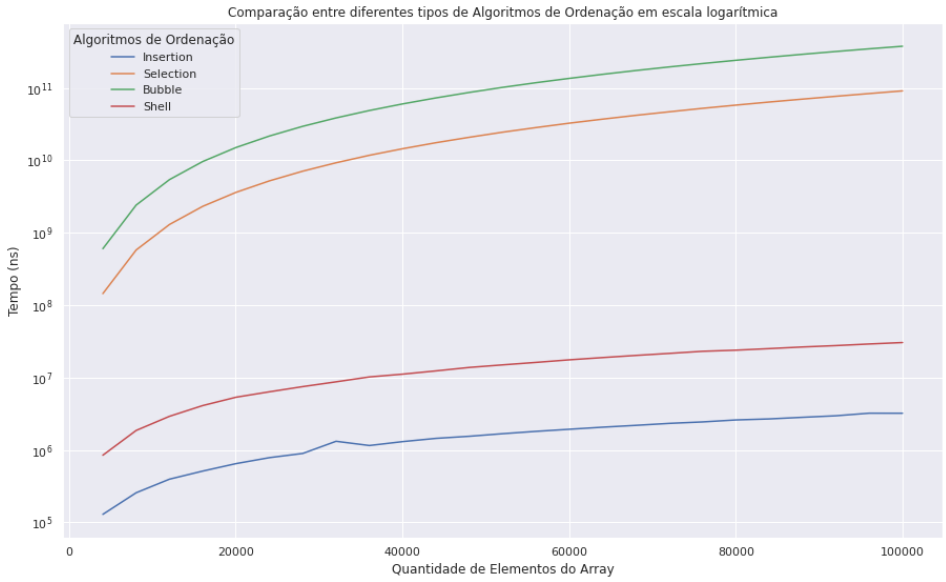
Figura 5: Amostra de dados do *datraframe* do cenário não decrescente.

| Quantidade de Elementos do Array | Insertion | Selection | Bubble | Shell | Quick | Merge | Radix |
|---|---|---|---|---|---|---|---|
| 4096 | 128668.2 | 1.442145e+08 | 6.020251e+08 | 844098.8 | 1622581.8 | 2648973.8 | 1500368.4 |
| 8092 | 255776.6 | 5.778142e+08 | 2.411658e+09 | 1856234.4 | 3498752.6 | 5362267.0 | 3011799.8 |
| 12088 | 393915.8 | 1.300325e+09 | 5.419876e+09 | 2910155.8 | 5830055.0 | 8204348.6 | 4554723.0 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 92008 | 2959210.8 | 7.676315e+10 | 3.191388e+11 | 27605479.4 | 53976855.2 | 65867099.2 | 35387455.6 |
| 96004 | 3212641.8 | 8.370143e+10 | 3.475630e+11 | 29053506.2 | 55216695.6 | 70004981.2 | 36781347.2 |
| 100000 | 3208047.2 | 9.116040e+10 | 3.771561e+11 | 30410682.6 | 57780816.6 | 74954341.0 | 38696686.2 |

Fonte: Autoria própria

Pode-se notar na Figura 5 que a tamanho mínimo da amostra foi $4096$ e que a máxima foi $100000$. Além disso, pode-se ter uma breve noção dos dados que foram coletados, como por exemplo: o *selection sort* e o *bubble sort* são ordem de grandeza maiores que os demais. Segue o gráfico com os algorítimos de complexidade $O(n^2)$.

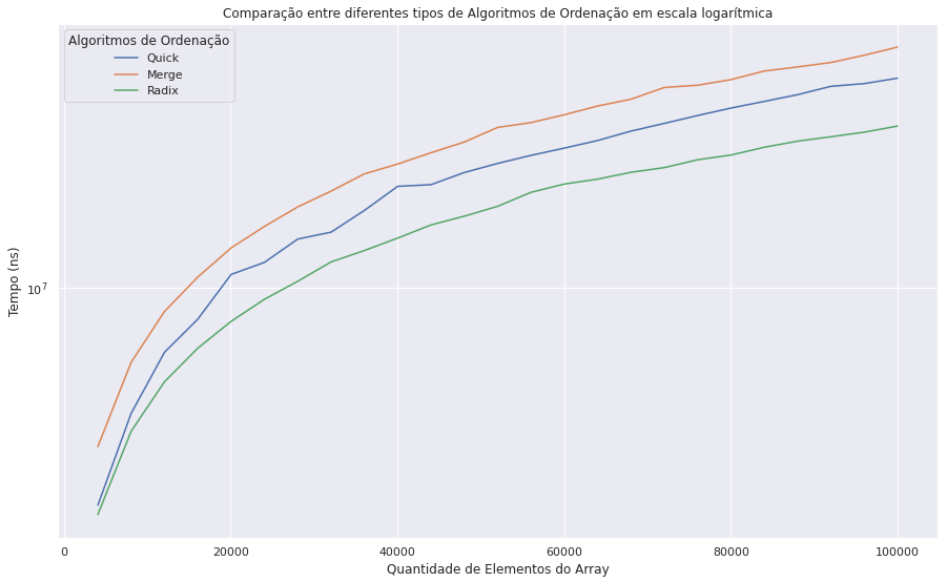
Figura 6: Gráfico para o cenário não decrescente de algoritmos com complexidade de $O(n^2)$.




Fonte: Autoria própria

Pode-se notar na Figura 6 que entre esses quatro algoritmos, os que tiveram pior performance foram, respectivamente, o *bubble sort* e o *selection sort*. E os que tiveram melhores desempenhos foram, respectivamente, os algoritmos *insertion sort* e o *shell sort*. Segue o gráfico com os algorítimos de complexidade $O(n \log n)$.

Figura 7: Gráfico para o cenário não decrescente de algoritmos com complexidade de $O(n \log n)$.




Fonte: Autoria própria

Neste gráfico, Figura 7, pode-se notar que entre os três algoritmos o que teve melhor desempenho foi o *radix sort*, e que o *quick sort* performou melhor que o *merge sort*. O porquê desse acontecimento será discutido no tópico 4.

Ademais, o que pode-se notar é que entre todos os algoritmos executados, o *insertion sort* se comportou diferente do esperado, esse comportamento será discutido no tópico 4.

## 3.2 Resultados do Cenário Não Crescente

Segue uma amostra dos tempos de execução de cada algoritmo para o cenário não crescente.

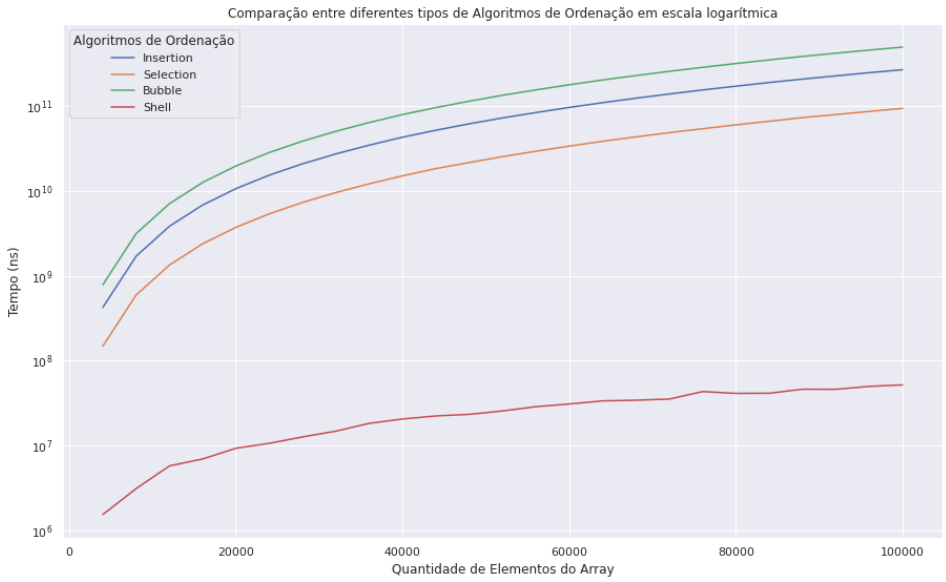
Figura 8: Amostra de dados do *datraframe* do cenário não crescente.

| Quantidade de Elementos do Array | Insertion | Selection | Bubble | Shell | Quick | Merge | Radix |
|---|---|---|---|---|---|---|---|
| 4096 | 4.244516e+08 | 1.485396e+08 | 7.834614e+08 | 1548480.4 | 2643365.2 | 2655273.6 | 1215297.6 |
| 8092 | 1.696606e+09 | 5.938005e+08 | 3.133376e+09 | 3127101.4 | 5915426.0 | 5419364.0 | 2447108.0 |
| 12088 | 3.823405e+09 | 1.337755e+09 | 7.055994e+09 | 5778447.8 | 9831192.0 | 8238667.6 | 4679305.2 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 92008 | 2.248654e+11 | 7.885847e+10 | 4.156361e+11 | 46030304.4 | 90996528.0 | 68670347.4 | 35229750.4 |
| 96004 | 2.451908e+11 | 8.584239e+10 | 4.524354e+11 | 49878537.8 | 96335924.6 | 71255537.0 | 37006671.6 |
| 100000 | 2.657169e+11 | 9.312788e+10 | 4.905802e+11 | 51876809.4 | 100902980.2 | 73697002.6 | 39408618.0 |

Fonte: Autoria própria

Observa-se na Figura 8 a amostra de dados coletados, vai do tamanho $4096$ até o tamanho $100000$. Ademais, é possível notar antecipadamente, nos dados gerados, que os algoritmos *bubble sort* e *insertion sort* possuem ordem de gradeza maiores que os demais. Abaixo, tem-se o gráfico dos algoritmos de complexidade $O(n^2)$.

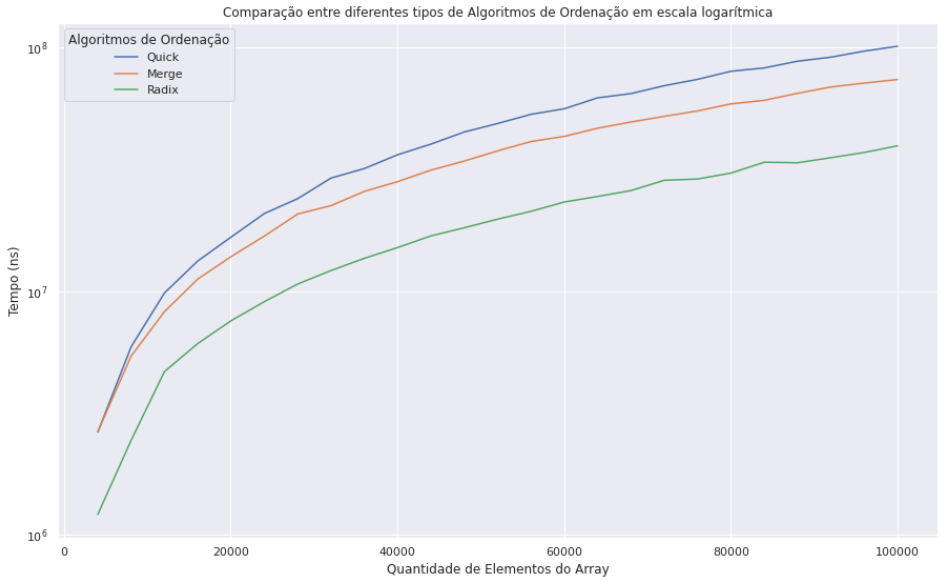
Figura 9: Gráfico para o cenário não crescente de algoritmos com complexidade $O(n^2)$.




Fonte: Autoria própria

Analisando o gráfico da Figura 9 nota-se que entre os quatro algoritmos, os que tiveram pior performance foram, respectivamente, *bubble sort* e *insertion sort*, como citado anteriormente. Já os que obteram melhor desempenho foram, respectivamente, os algoritmos *shell sort* e *selection sort*. Abaixo, tem-se o gráfico dos algoritmos de complexidade $O(n \log n)$.

Figura 10: Gráfico para o cenário não crescente de algoritmos com complexidade $O(n \log n)$.




Fonte: Autoria própria

No gráfico acima, Figura 10, percebe-se que os algoritmos que tiveram melhor desempenho foram, respectivamente, o *radix sort*, o *merge sort* e o *quick sort*.

## 3.3 Resultados do Cenário Aleatório

Segue uma amostra dos tempos de execução de cada algoritmo para o cenário aleatório.

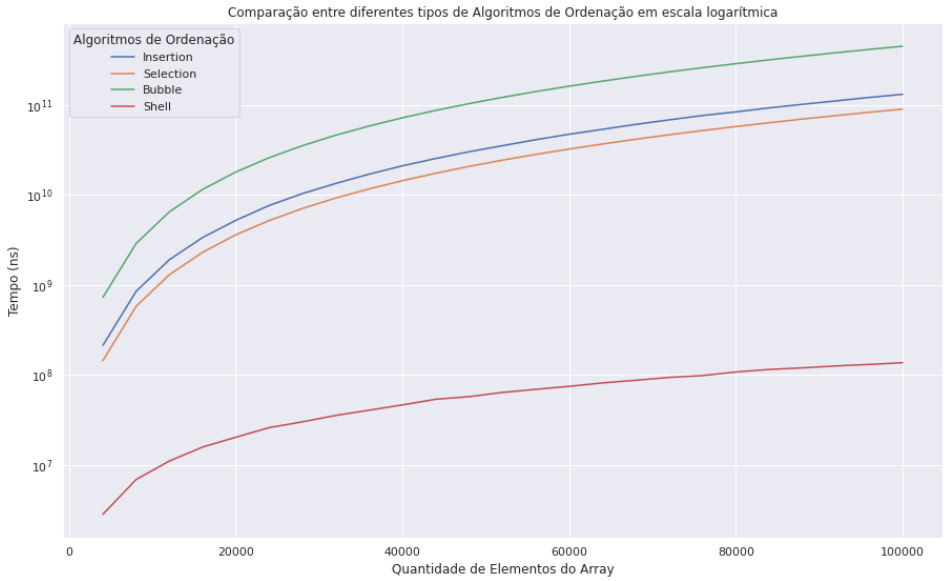
Figura 11: Amostra de dados do *datraframe* do cenário $100\%$ aleatório.

| Quantidade de Elementos do Array | Insertion | Selection | Bubble | Shell | Quick | Merge | Radix |
|---|---|---|---|---|---|---|---|
| 4096 | 2.125813e+08 | 1.442544e+08 | 7.291048e+08 | 2812963.8 | 2195210.2 | 3187757.0 | 1501626.6 |
| 8092 | 8.520856e+08 | 5.803891e+08 | 2.901296e+09 | 6870310.8 | 4706987.6 | 6677793.6 | 3021084.8 |
| 12088 | 1.917271e+09 | 1.303980e+09 | 6.534287e+09 | 11039149.4 | 7419705.4 | 10028836.2 | 5050919.6 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 92008 | 1.116985e+11 | 7.661041e+10 | 3.831646e+11 | 125912710.0 | 68168243.8 | 83399712.4 | 34782595.8 |
| 96004 | 1.219293e+11 | 8.343806e+10 | 4.173626e+11 | 130789906.8 | 72175465.4 | 86909168.4 | 37309188.8 |
| 100000 | 1.323231e+11 | 9.053148e+10 | 4.527920e+11 | 136746451.6 | 75735229.4 | 90889942.8 | 38224518.4 |

Fonte: Autoria própria

Repara-se na Figura 11 a amostra de dados coletados, vai do tamanho $4096$ até o tamanho $100000$. Ademais, é possível verificar antecipadamente, nos dados gerados, que os algoritmos *bubble sort* e *insertion sort* possuem ordem de gradeza maiores que os demais. Abaixo, tem-se o gráfico dos algoritmos de complexidade $O(n^2)$.

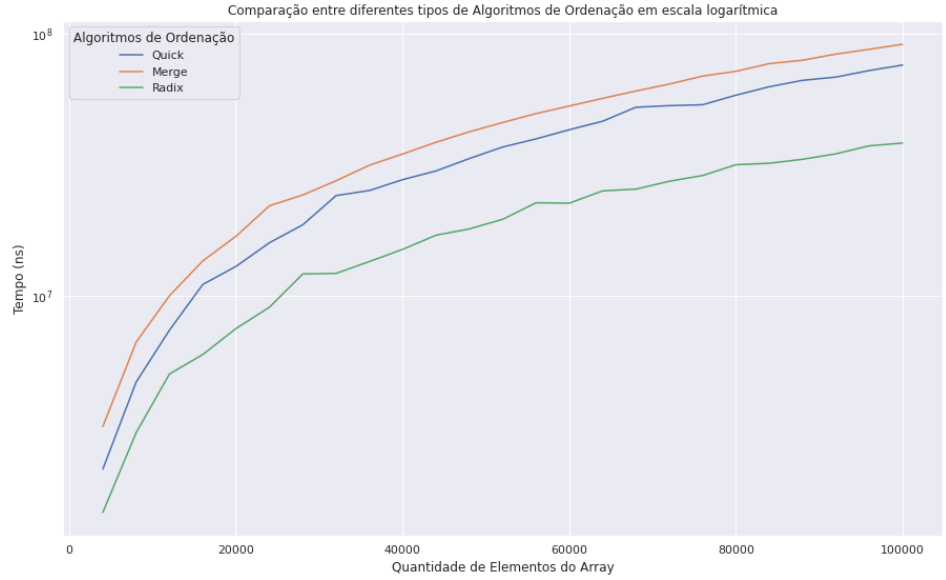
Figura 12: Gráfico para o cenário com elementos $100\%$ em posições aleatórias de algoritmos com complexidade $O(n^2)$.




Fonte: Autoria própria

Observando o gráfico da Figura 12 constata-se que entre os quatro algoritmos, os que tiveram pior desempenho foram, respectivamente, *bubble sort* e *insertion sort*, como citado anteriormente. Já os que obteram melhor performance foram, respectivamente, os algoritmos *shell sort* e *selection sort*. Abaixo, tem-se o gráfico dos algoritmos de complexidade $O(n \log n)$.

Figura 13: Gráfico para o cenário com elementos $100\%$ em posições aleatórias de algoritmos com complexidade $O(n \log n)$.




Fonte: Autoria própria

No gráfico acima, Figura 13, percebe-se que os algoritmos que tiveram melhor desempenho foram o *radix sort* e o *quick sort*. Assim, o *quick sort* performou melhor que o *merge sort*. O porquê desse acontecimento será discutido no tópico 4.

### 3.4 Resultados do Cenário $75\%$ em Posição Definitiva

Segue uma amostra dos tempos de execução de cada algoritmo para o cenário $75\%$ em posição definitiva.

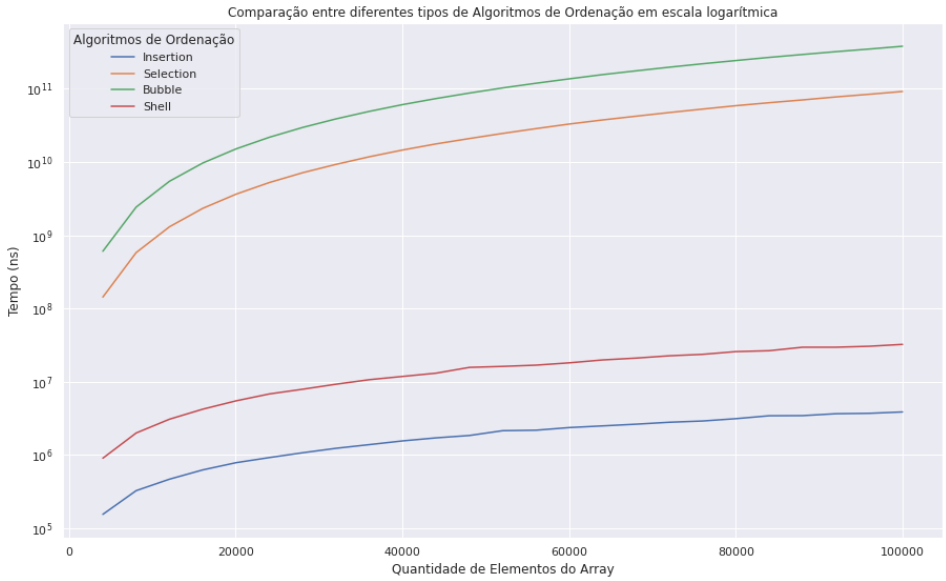
Figura 14: Amostra de dados do *datraframe* do cenário $75\%$ em posição definitiva.

| Quantidade de Elementos do Array | Insertion | Selection | Bubble | Shell | Quick | Merge | Radix |
|---|---|---|---|---|---|---|---|
| 4096 | 227049.6 | 1.438521e+08 | 6.038270e+08 | 952540.8 | 3.215086e+06 | 2672388.2 | 1532865.0 |
| 8092 | 412181.8 | 5.779027e+08 | 2.411879e+09 | 2126486.6 | 9.928109e+06 | 5435265.8 | 2994082.8 |
| 12088 | 613737.8 | 1.299219e+09 | 5.425901e+09 | 3229140.4 | 1.151457e+07 | 8269031.2 | 4540372.0 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 92008 | 4872001.4 | 7.663076e+10 | 3.191350e+11 | 30050371.0 | 4.114671e+08 | 66488265.0 | 34918964.4 |
| 96004 | 4985180.6 | 8.358164e+10 | 3.475976e+11 | 31647334.6 | 1.090503e+09 | 71180502.0 | 36686800.4 |
| 100000 | 5150542.6 | 9.071577e+10 | 3.773927e+11 | 32949202.8 | 1.262814e+09 | 73454013.6 | 39218355.4 |

Fonte: Autoria própria

Pode-se notar na Figura 14 que a tamanho mínimo da amostra foi $4096$ e que a máxima foi $100000$. Além disso, pode-se ter uma breve noção dos dados que foram coletados, como por exemplo: o *selection sort*, o *bubble sort* e o *quick sort* são ordem de grandeza maiores que os demais. Segue o gráfico com os algorítimos de complexidade $O(n^2)$.

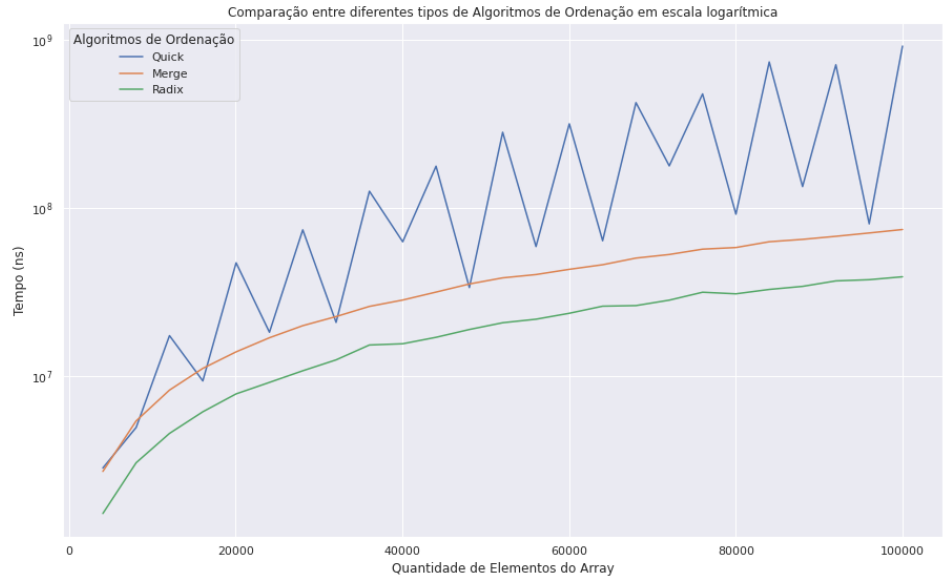
Figura 15: Gráfico para o cenário com elementos $75\%$ em posição definitiva de algoritmos com complexidade $O(n^2)$.




Fonte: Autoria própria

Pode-se notar na Figura 15 que entre esses quatro algoritmos, os que tiveram pior performance foram, respectivamente, o *bubble sort* e o *selection sort*. E os que tiveram melhores desempenhos foram, respectivamente, os algoritmos *insertion sort* e o *shell sort*. Segue o gráfico com os algorítimos de complexidade $O(n \log n)$.

Figura 16: Gráfico para o cenário com elementos $75\%$ em posição definitiva de algoritmos com complexidade $O(n \log n)$.




Fonte: Autoria própria

Neste gráfico, Figura 16, pode-se notar que entre os três algoritmos o que teve melhor desempenho foi o *radix sort*, e que o *merge sort* performou melhor que o *quick sort*. O porquê desse acontecimento também será discutido no tópico 4.

Ademais, o que pode-se notar é que entre todos os algoritmos executados, o *insertion sort* se comportou diferente do esperado, esse comportamento será discutido no tópico 4.

### 3.5 Resultados do Cenário $50\%$ em Posição Definitiva

Segue uma amostra dos tempos de execução de cada algoritmo para o cenário $50\%$ em posição definitiva.

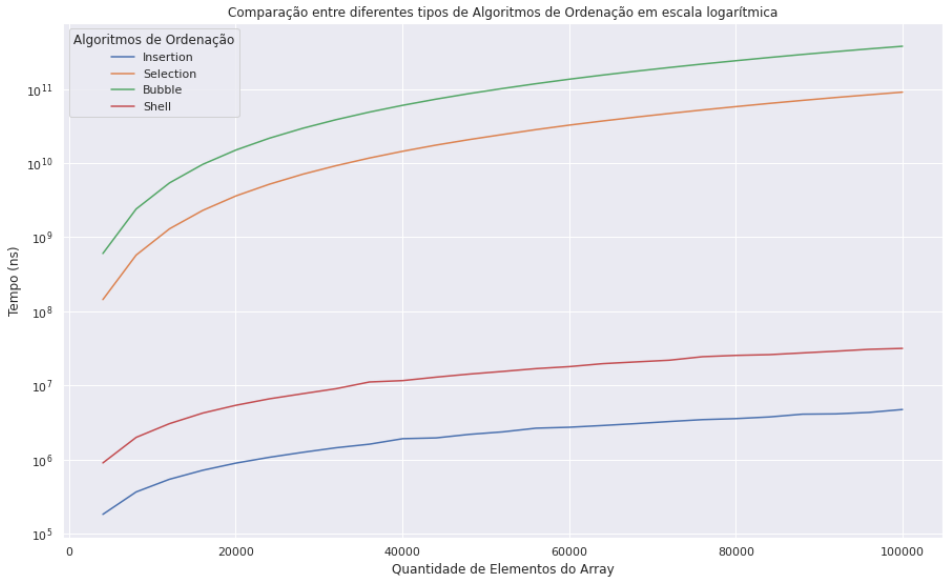
Figura 17: Amostra de dados do *datraframe* do cenário $50\%$ em posição definitiva.

| Quantidade de Elementos do Array | Insertion | Selection | Bubble | Shell | Quick | Merge | Radix |
|---|---|---|---|---|---|---|---|
| 4096 | 182069.6 | 1.437806e+08 | 6.017865e+08 | 899266.2 | 3.051750e+06 | 2643017.4 | 1525892.8 |
| 8092 | 365406.0 | 5.737515e+08 | 2.400617e+09 | 1990009.4 | 5.297528e+06 | 5400469.2 | 3011159.0 |
| 12088 | 541469.0 | 1.296523e+09 | 5.396203e+09 | 3049162.6 | 1.090928e+07 | 8315936.6 | 4525312.2 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 92008 | 4127938.6 | 7.666375e+10 | 3.189656e+11 | 28938127.4 | 3.708322e+08 | 66580665.2 | 35219622.8 |
| 96004 | 4323034.8 | 8.338473e+10 | 3.473497e+11 | 30672148.4 | 3.113311e+08 | 70163037.8 | 37024662.6 |
| 100000 | 4730879.8 | 9.056628e+10 | 3.767520e+11 | 31583721.4 | 1.125992e+09 | 73919512.4 | 37960169.6 |

Fonte: Autoria própria

Pode-se notar na Figura 17 que a tamanho mínimo da amostra foi $4096$ e que a máxima foi $100000$. Além disso, pode-se ter uma breve noção dos dados que foram coletados, como por exemplo: o *selection sort*, *bubble sort* e o *quick sort* são ordem de grandeza maiores que os demais. Segue o gráfico com os algorítimos de complexidade $O(n^2)$.

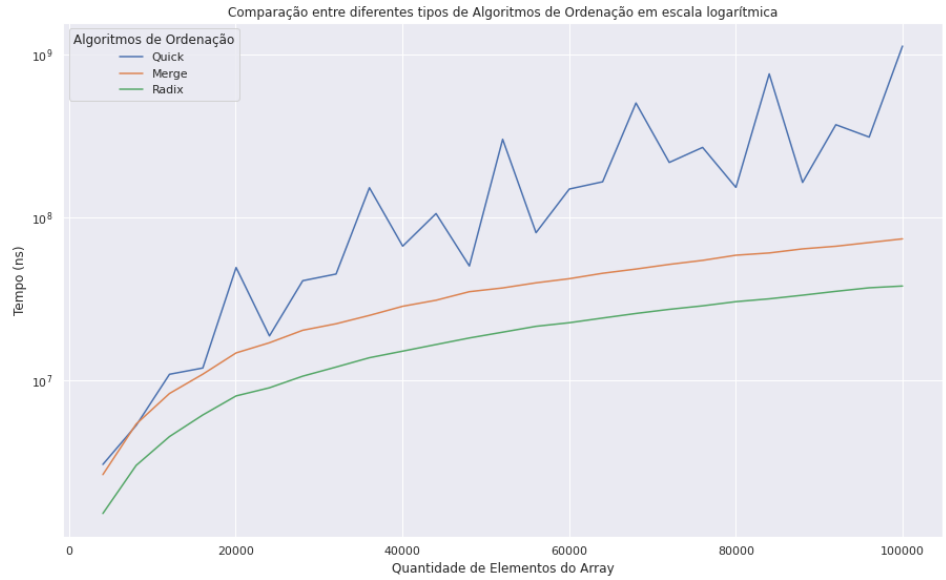
Figura 18: Gráfico para o cenário com elementos $50\%$ em posição definitiva de algoritmos com complexidade $O(n^2)$.




Fonte: Autoria própria

Pode-se notar na Figura 18 que entre esses quatro algoritmos, os que tiveram pior performance foram, respectivamente, o *bubble sort* e o *selection sort*. E os que tiveram melhores desempenhos foram, respectivamente, os algoritmos *insertion sort* e o *shell sort*. Segue o gráfico com os algorítimos de complexidade $O(n \log n)$.

Figura 19: Gráfico para o cenário com elementos $50\%$ em posição definitiva de algoritmos com complexidade $O(n \log n)$.




Fonte: Autoria própria

Neste gráfico, Figura 19, pode-se notar que entre os três algoritmos o que teve melhor desempenho foi o *radix sort*, e que o *merge sort* performou melhor que o *quick sort*. O porquê desse acontecimento será discutido no tópico 4.

Ademais, o que pode-se notar é que entre todos os algoritmos executados, o *insertion sort* se comportou diferente do esperado, esse comportamento será discutido no tópico 4.

### 3.6 Resultados do Cenário $25\%$ em Posição Definitiva

Segue uma amostra dos tempos de execução de cada algoritmo para o cenário $25\%$ em posição definitiva.

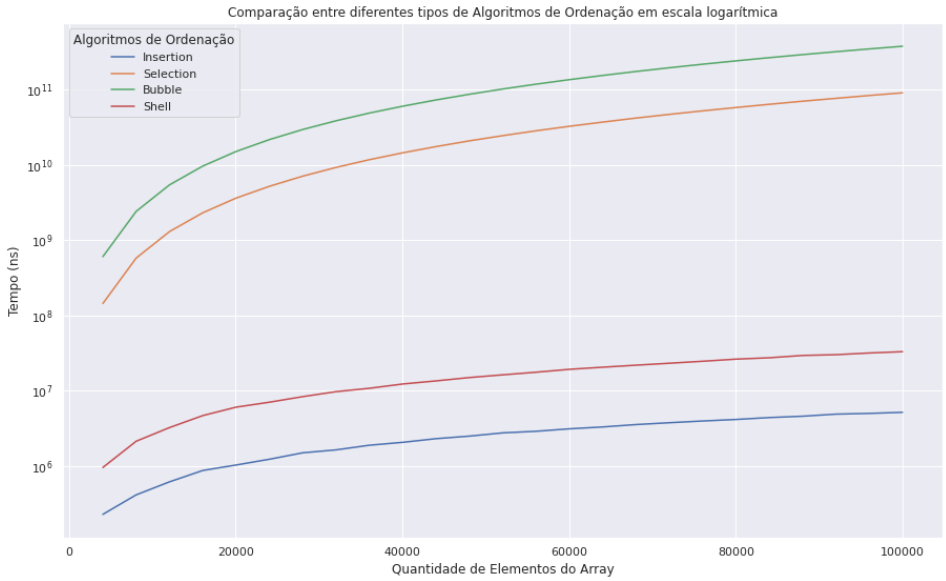
Figura 20: Amostra de dados do *datraframe* do cenário $25\%$ em posição definitiva.

| Quantidade de Elementos do Array | Insertion | Selection | Bubble | Shell | Quick | Merge | Radix |
|---|---|---|---|---|---|---|---|
| 4096 | 227049.6 | 1.438521e+08 | 6.038270e+08 | 952540.8 | 3.215086e+06 | 2672388.2 | 1532865.0 |
| 8092 | 412181.8 | 5.779027e+08 | 2.411879e+09 | 2126486.6 | 9.928109e+06 | 5435265.8 | 2994082.8 |
| 12088 | 613737.8 | 1.299219e+09 | 5.425901e+09 | 3229140.4 | 1.151457e+07 | 8269031.2 | 4540372.0 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 92008 | 4872001.4 | 7.663076e+10 | 3.191350e+11 | 30050371.0 | 4.114671e+08 | 66488265.0 | 34918964.4 |
| 96004 | 4985180.6 | 8.358164e+10 | 3.475976e+11 | 31647334.6 | 1.090503e+09 | 71180502.0 | 36686800.4 |
| 100000 | 5150542.6 | 9.071577e+10 | 3.773927e+11 | 32949202.8 | 1.262814e+09 | 73454013.6 | 39218355.4 |

Fonte: Autoria própria

Pode-se notar na Figura 20 que a tamanho mínimo da amostra foi $4096$ e que a máxima foi $100000$. Além disso, pode-se ter uma breve noção dos dados que foram coletados, como por exemplo: o *selection sort*, *bubble sort* e o *quick sort* são ordem de grandeza maiores que os demais. Segue o gráfico com os algorítimos de complexidade $O(n^2)$.

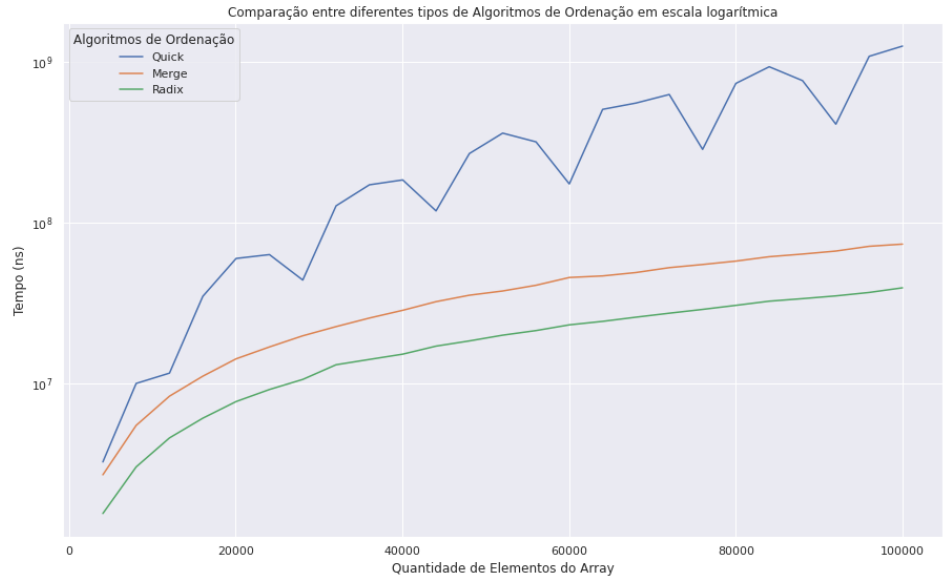
Figura 21: Gráfico para o cenário com elementos $25\%$ em posição definitiva de algoritmos com complexidade $O(n^2)$.




Fonte: Autoria própria

Pode-se notar na Figura 21 que entre esses quatro algoritmos, os que tiveram pior performance foram, respectivamente, o *bubble sort* e o *selection sort*. E os que tiveram melhores desempenhos foram, respectivamente, os algoritmos *insertion sort* e o *shell sort*. Segue o gráfico com os algorítimos de complexidade $O(n \log n)$.

Figura 22: Gráfico para o cenário com elementos $25\%$ em posição definitiva de algoritmos com complexidade $O(n \log n)$.




Fonte: Autoria própria

Neste gráfico, Figura 22, pode-se notar que entre os três algoritmos o que teve melhor desempenho foi o *radix sort*, e que o *merge sort* performou melhor que o *quick sort*. O porquê desse acontecimento será discutido no tópico 4.

Ademais, o que pode-se notar é que entre todos os algoritmos executados, o *insertion sort* se comportou diferente do esperado, esse comportamento será discutido no tópico 4.

# 4 DISCUSSÃO

Essa seção tem como objetivo discutir os resultados obtidos na seção anterior.

## 4.1 Comparando os algoritmos

Comparando os algoritmos de complexidade $O(n^2)$, tem-se que para todos os cenários o algoritmo *bubble sort* é o que possui o pior desempenho. Além disso, o *shell sort* e o *selection sort* tiveram o mesmo comportamento para todos os cenários. Ademais, *insertion sort* teve desempenho variado nos cenários não decrescente, $25\%$, $50\%$ e $75\%$ dos elementos em posição definitiva, pois ele foi privilegiado e por isso apresentou um ótimo desempenho, uma vez que, os vetores estavam subordenados, ver terceiro parágrafo da seção 4.4 para melhor compreensão de como os cenários $25\%$, $50\%$ e $75\%$ dos elementos em posição definitiva foram construídos. Para os demais cenários o desempenho do *insertion sort* foi similar ao do *selection sort*.

Comparando os algoritmos de complexidade $O(n \log n)$, tem-se que para todos os cenários o algoritmo *radix sort* é o mais eficiente. Ademais, o *merge sort* para os cenários não crescente, $25\%$, $50\%$ e $75\%$ dos elementos em posição definitiva teve performance superior ao *quick sort*, devido ao seu desempenho instável. Dessa forma, embora que na teoria o *merge sort* seja mais rápido *quick sort*, nem sempre isso ocorre.

Além disso, os dois algoritmos com os melhores tempo de execução foram, respectivamente:

1. Para o cenário não decrescente, o *insertion sort* e o *shell sort*.
2. Para o cenário não crescentes, o *radix sort* e o *shell sort*.
3. Para o *array* com $100\%$ dos elementos aleatórios, o *radix sort* e *quick sort*.
4. Para o *array* com $25\%$ de seus elementos em sua posição definitiva, o *insertion sort* e o *shell sort*.
5. Para o *array* com $50\%$ de seus elementos em sua posição definitiva, o *insertion sort* e o *shell sort*.
6. Para o *array* com $75\%$ de seus elementos em sua posição definitiva, o *insertion sort* e o *shell sort*.

De maneira geral, pode constatar que cada algoritmo tem um desempenho diferente dependendo do cenário em que ele está sendo utilizado. Dessa forma, o algoritmo mais eficiente a ser implementado em algum problema vai depender do cenário.

## 4.2 Como o algoritmo de decomposição de chave *radix sort* se compara com o melhor algoritmo baseado em comparação de chaves?

O algorítimo de ordenação *radix sort* baseado na decomposição de chave é levemente mais rápido que todos os algoritmos de ordenação baseado em comparação de chaves de complexidade $O(n \log n)$. Isso ocorre porque a complexidade do *radix sort* é $O(n)$.

## 4.3 Correspondência com as análises matemáticas

Todos os algoritmos apresentaram o comportamento esperado.

## 4.4 Vales e picos nos gráficos

Fazendo uma análise nos gráficos é notável que o algoritmo *quick sort* apresentou comportamento incomum para os cenários $75\%$, $50\%$ e $25\%$ de seus elementos em sua posição definitiva. Para estes cenários, os gráficos gerados apresentam grandes oscilações, cheios de picos e vales.

O primeiro motivo que cativa esse comportamento é sua a lógica, apesar de possuir um conjunto de etapas bem definidas o *quick sort* pode ter enormes variações, pois sua estratégia de ordenação consiste em criar dois subvetores dado um pivô. Desse modo, a performance do algoritmo depende diretamente da forma que os pivôs são selecionados. A equipe adotou a estratégia "medianas-dos-três"para determinar este valor, que consiste em simplesmente escolher a mediana do primeiro elemento com o ultimo elemento com o elemento do meio. Com essa estratégia é possível que dado uma variação de tamanho em uma sequência de elementos ordenáveis o *quick sort* tenha um padrão em seu comportamento no momento da ordenação, que foi exatamente o que aconteceu nesses casos.

Outro motivo que cativa esse comportamento é justamente a forma como esses cenários foram criados, pois, esse tipo de comportamento só aconteceu nesses cenários específicos. Os cenários $75\%$, $50\%$ e $25\%$ de seus elementos em sua posição definitiva rearranjam uma coleção de elementos de forma que, primeiramente, os elemento são ordenados, e depois um determinado percentual do conjunto é desordenado, tal que a cada dois elementos é efetuado a troca de posição com seu elemento subsequente. Sabendo disso, o cenário tem um padrão bem definido em sua construção, possibilitando o padrão observado nos gráficos para o algoritmo *quick sort*. E é por esses motivos que o *quick sort* se comportou de forma incomum para esses cenários.

# REFERÊNCIAS

CORMEN, T. H. et al. **Algoritmos: Teoria e Prática**. Rio de Janeiro: Elsevier Editora Ltda., v. 3, p. 17,30, 2012. Disponível em: <https://computerscience360.files.wordpress.com/2018/02/algoritmos-teoria-e-prc3a1tica-3ed-thomas-cormen.pdf>. Acesso em: 21.07.2021. Citado na página 5.

# APÊNDICES

# APÊNDICE A – Implementação dos algoritmos em C++

## A.1 Bubble Sort

```
1   template< typename RandomIt, typename Compare >
2   void bubble(RandomIt first, RandomIt last, Compare cmp)
3   {
4       for(RandomIt i=first; i<last; i++)
5       {
6           for(RandomIt j=first; j<last-1; j++)
7           {
8               if(cmp( *(j+1), *j ))
9               {
10                  std::swap(*j, *(j+1));
11              }
12          }
13      }
14  }
```

## A.2 Insertion Sort

```
1   template< typename RandomIt, typename Compare >
2   void insertion(RandomIt first, RandomIt last, Compare cmp)
3   {
4       for (RandomIt i = first+1; i < last; i++)
5       {
6           RandomIt j = i;
7
8           while (j > first && cmp(*j, *(j-1)))
9           {
10              std::swap(*j, *(j-1));
11              j--;
12          }
13
14          }
15  }
```

## A.3 Selection Sort

```
1   template< typename RandomIt, typename Compare >
2   void selection(RandomIt first, RandomIt last, Compare cmp)
3   {
4       for (RandomIt i = first; i < last-1; i++)
5       {
```

```
6             RandomIt smaller = i;
7             for (RandomIt j = i + 1; j < last; j++)
8             {
9                 if (cmp(*j, *smaller))
10                {
11                    smaller = j;
12                }
13            }
14            std::swap(*i, *smaller);
15            }
16    }
```

## A.4 Shell Sort

```
1     template< typename RandomIt, typename Compare >
2     void shell(RandomIt first, RandomIt last, Compare cmp)
3     {
4         int length = std::distance( first, last ),
5             interval, i, auxiliary, j;
6
7         while ( interval < length )
8         {
9             interval = interval*3+1;
10        }
11
12        while ( interval > 1 )
13        {
14            interval /= 3;
15            for ( i = interval; i < length; i++ )
16            {
17                auxiliary = *(first + i);
18                for ( j = i; j >= interval &&
19                    cmp(auxiliary, *(first+(j-interval))); j-=interval )
20                {
21                    *(first + j) = *(first + (j-interval));
22                }
23                *(first + j) = auxiliary;
24            }
25        }
26    }
```

## A.5 Merge Sort

```
1     template< typename FwrdIt, typename Compare >
2     void mergeSort( FwrdIt L, FwrdIt l_last,
3          FwrdIt R, FwrdIt r_last,
```

```
4           FwrdIt A, Compare cmp)
5       {
6           size_t sizeArrayLeft = std::distance( L, l_last ),
7                  sizeArrayRight = std::distance( R, r_last ),
8                  indexArrayLeft = 0, indexArrayRight = 0,
9                  indexMergedArray = 0;
10
11          while (indexArrayLeft < sizeArrayLeft &&
12                 indexArrayRight < sizeArrayRight)
13          {
14              if ( cmp( L[indexArrayLeft], R[indexArrayRight] ) )
15              {
16                  A[indexMergedArray] = L[indexArrayLeft++];
17              }
18              else
19              {
20                  A[indexMergedArray] = R[indexArrayRight++];
21              }
22              indexMergedArray++;
23          }
24
25          while ( indexArrayLeft < sizeArrayLeft )
26          {
27              A[indexMergedArray++] = L[indexArrayLeft++];
28          }
29
30          while ( indexArrayRight < sizeArrayRight )
31          {
32              A[indexMergedArray++] = R[indexArrayRight++];
33          }
34
35      }
36
37      template< typename RandomIt, typename Compare >
38      void merge(RandomIt first, RandomIt last, Compare cmp)
39      {
40          using myType = typename std::remove_reference
41                          <decltype(*std::declval<RandomIt>())>::type;
42
43          size_t length = std::distance( first, last );
44
45          if (length > 1)
46          {
47              size_t L_sz = length/2;
48              size_t R_sz = length - L_sz;
49
50              std::vector<myType> L( L_sz );
```

```
51              std::vector<myType> R( R_sz );
52
53              std::copy( first, first+L_sz, L.begin() );
54              std::copy( first+L_sz, last, R.begin() );
55
56              merge(L.begin(), L.end(), cmp);
57              merge(R.begin(), R.end(), cmp);
58
59              mergeSort( L.begin(), L.end(), R.begin(), R.end(), first, cmp );
60
61          }
62      }
```

## A.6 Quick Sort

```
1       template<class FwrdIt, class Compare>
2       FwrdIt partition(FwrdIt first, FwrdIt last, FwrdIt pivot, Compare cmp)
3       {
4           FwrdIt middle = first + (last-first)/2, fast = first;
5
6           if ( *first < *middle )
7           {
8               if( *middle < *(last-1) )
9               {
10                  pivot = middle;
11              }
12              else if ( *first < *(last-1) )
13              {
14                  pivot = last-1;
15              }
16              else
17              {
18                  pivot = first;
19              }
20          }
21          else
22          {
23              if( *(last-1) < *middle )
24              {
25                  pivot = middle;
26              }
27              else if ( *(last-1) < *first )
28              {
29                  pivot = last-1;
30              }
31              else
32              {
```

```
33                  pivot = first;
34              }
35          }
36
37          if ( pivot != last-1 )
38          {
39              std::swap( *pivot, *(last-1) );
40              pivot = last-1;
41          }
42
43          for (FwrdIt j = first; j < last; j++)
44          {
45              if (cmp(*j, *pivot))
46              {
47                  std::swap(*fast, *j);
48                  fast++;
49              }
50          }
51
52          std::swap(*fast, *pivot);
53          pivot = fast;
54
55          return pivot;
56      }
57
58      template<typename RandomIt, typename Compare>
59      void quick(RandomIt first,RandomIt last,Compare comp){
60          int length = std::distance( first, last );
61          if (length <= 1)
62          {
63              return;
64          }
65
66          RandomIt pivot = partition( first, last, last-1, comp );
67          quick( first, pivot, comp );
68          quick( pivot+1, last, comp );
69      }
```

## A.7 Radix Sort

```
1       template < typename FwrdIt, typename Comparator >
2       void radix( FwrdIt first, FwrdIt last, Comparator cmp )
3       {
4           using myType = typename std::remove_reference
5                           <decltype(*std::declval<FwrdIt>())>::type;
6
7           int index, exp=1;
```

```
8         size_t arraySize = std::distance( first, last );
9         std::vector <myType> auxiliary(arraySize);
10
11
12        std::copy( first, last, auxiliary.begin() );
13
14        myType largest = *first;
15        for (FwrdIt iterator=first+1; iterator<last; iterator++)
16        {
17            if ( cmp (largest, *iterator) )
18                largest = *iterator;
19        }
20
21        while ( largest/exp > 0 )
22        {
23            int count[10]={0};
24
25            for ( index=0; index< (static_cast<int> (arraySize)); index++ )
26                count[( *(first+index) / exp ) % 10]++;
27
28            for ( index=1; index<10; index++ )
29                count[index] += count[index - 1];
30
31            for ( index=arraySize-1; index>=0; index-- )
32                auxiliary[--count[( *(first+index) / exp) % 10]] =
33                                                    *(first+index);
34
35            for (index=0; index<(static_cast<int> (arraySize)); index++)
36                *(first+index) = auxiliary[index];
37
38            exp *= 10;
39        }
40    }
```