Relatório do Trabalho de Ordenação e Estatísticas de Ordem

Técnicas de Programação Avançada — Ifes — Campus Serra

Alunos: David Vilaça, Harã Heique, Larissa Motta.
Prof. Jefferson O. Andrade

14 de outubro de 2019

Sumário

Lista de Códigos Fonte

Lista de Figuras

1 Introdução

A ordenação é um dos problemas algorıtmicos mais fundamentais da ciência da computação. Já foi afirmado que até 25% de todos os ciclos de CPU são usados para ordenação, o que fornece um grande incentivo para estudos adicionais e otimização sobre ordenação. Dado isso o trabalho consiste na implementação de algoritmos de ordenação, assim como análises de suas performances.
Este documento tem o propósito de servir de relatório do Trabalho de Ordenação e Estatísticas de Ordem da disciplina de Técnicas de Programação Avançada, descrevendo sete dos nove algoritmos propostos sendo cinco deles obrigatórios, os quais são: selection sort, insertion sort, merge sort, quicksort e heapsort, e dois dos quatro restantes a critério de escolha do grupo, os quais são: introsort, timsort, smoothsort e patience sort.
Baseando-se nisso, os algoritmos escolhidos para o implementados neste trabalho foram:

1. Selection Sort;
2. Insertion Sort;
3. Merge Sort;
4. Quicksort;
5. Heapsort;
6. Introsort;
7. Timsort.

1.1 Ambiente de Desenvolvimento

O desenvolvimento do trabalho foi através da linguagem de programação *Python (versão 3.6.8+)* juntamente com o editor de código fonte *Visual Studio Code (versão 1.35.1+)*, como ambiente de desenvolvimento, cujo Sistema Operacional utilizado foi *Linux (distribuições Ubuntu 18.04 LTS e Mint 19.02 Cinnamon)*. A máquina utilizada para realizar os testes de análise dos algoritmos foi: *I7 3rd generation 16GB RAM(4x4GB)*. Todo código está hospedado no GitHub.
Para implementação dos algoritmos e aplicação como todo foram utilizadas somente bibliotecas standards da própria linguagem *Python*, não havendo a necessidade de utilizações de pacotes externos comumente provenientes do *PIP* package installer.

2 Dados para Testes

Os arquivos de dados foram fornecidos pelo professor através de um diretório compartilhado no GOOGLE DRIVE, no formato CSV e são compostos pelos seguintes campos:

- Email (email); tipo: texto;
- Sexo (gender); tipo: caractere; valores válidos: M, F, O;
- Identificador de Usuário (uid); tipo: alfa-numérico; único;
- Data de nascimento (birthdate); tipo: data; formato: ISO-8601;
- Altura, em centímetros (height); tipo: inteiro;
- Peso, em quilogramas (weight); tipo: inteiro.

3 Implementação do Trabalho

Como dito anteriormente, o trabalho foi implementado na linguagem *Python*, utilizando o ambiente de desenvolvimento *Visual Studio Code*. Os principais arquivos/módulos do projeto são:

- **build.py** – Contém o código para o programa principal, que interage com o usuário, além de um voltado para testes simples dos algoritmos para o desenvolvedor e também um código para realizar a chamada do script de análise dos dados.
- **sort_collection.py** – Contém a implementação dos algoritmos de ordenação.
- **Person.py** – Classe responsável por representar a entidade Pessoa (Person em inglês).
- **ExecutionType.py** – É uma classe do tipo Enum utilizada no módulo de build.py da aplicação principal que define o tipo de execução selecionada.
- **handler_person.py** – Contém o código de implementação da manipulação dos dados acerca dos objetos da classe Person. Toda parte de lógica, "hard code", está presente neste módulo, onde o build.py faz chamada de suas funções.
- **analisys_person.py** – Contém toda lógica do script para realizar análise acerca da performance de todos os sete algoritmos implementados.

3.1 Programa Principal

O programa principal é trivial na interação entre o usuário e a aplicação desenvolvida, onde ele é responsável pelas seguintes etapas:

1. Processamento dos argumentos passadas na linha de comando inseridos pelo usuário.
2. Ler os dados de entrada de um arquivo CSV passado na linha de comando e extraí-los para uma vetor da classe Person.
3. Executar um algoritmo escolhido pelo usuário passado na linha de comando.
4. Coletar o tempo de execução de processamento em CPU do algoritmo de ordenação selecionado.
5. Gravar os dados de saída ordenados em um arquivo CSV.
6. Reportar o tempo final de execução do processo de ordenação, indicando também o algoritmo selecionado e a quantidade de registros ordenados.

A Figura 1 demonstra um exemplo da chamada de execução da aplicação de uma entrada dos argumentos na linha de comando realizada pelo usuário. Note que tanto o argumento de *–input* quanto *–output* são obrigatórios e indicam respectivamente os nomes dos arquivos de entrada e saída. Já o argumento *–algorithm* determina o algoritmo de ordenação a ser executado.

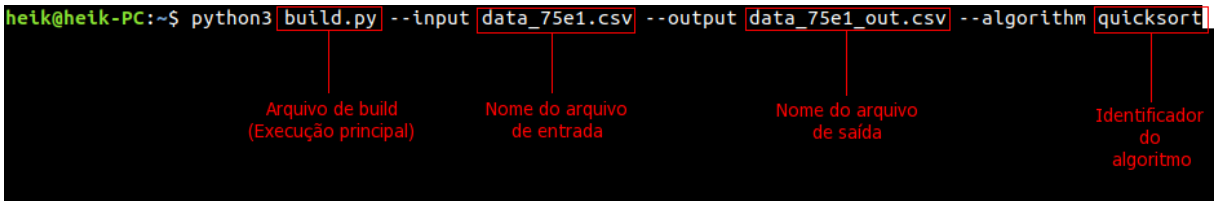

Figura 1: Exemplo de execução da aplicação na linha de comando.

Já a Figura 2 demonstra como o resultado de saída que é apresentado após a execução do programa definida pela última etapa programa principal discutido acima.

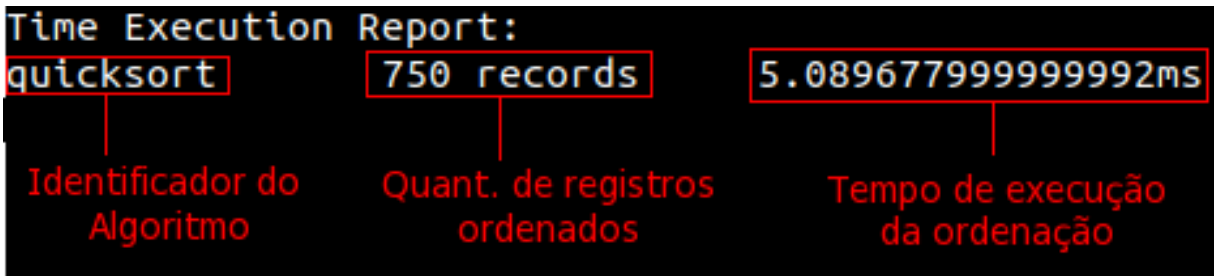

Figura 2: Exemplo do resultado de saída da execução do programa principal.

O Código Fonte 1 mostra a função principal, chamada de *main*, onde foi optado seguir a ideia da arquitetura proposta, ou seja, de ter um programa principal que recebe como parâmetro o algoritmo de ordenação a ser aplicado, como é mostrado na Figura 1.

```
1  def main():
2    print("Processing...")
3    # Definição dos argumentos passados no CLI.
4    args =
     ↪  handler_person.CLI_definition(list(sort.ALGORITHMS_SORTING_CHOICES.keys()))
5
6    # Pega os nome do arquivo de entrada e saída. Caso estejam errados é lançado
     ↪  uma exceção
7    input_filename: str = handler_person.check_filename(args.input)
8    output_filename: str = handler_person.check_filename(args.output)
9
10   # Extrai os objetos Person do arquivo.csv
11   lst_person: List[Person] = handler_person.readCSV(input_filename)
12
13   # Mensura o tempo de execução da CPU do processo de ordenação feita pelo
     ↪  algoritmo passado como argumento proveniente do CLI
14   start_time: float = handler_person.get_cpu_time()
15   sort.sort(lst_person, args.algorithm) if (args.algorithm != None) else
     ↪  sort.sort(lst_person)
16   finish_time: float = handler_person.get_cpu_time()
17
18   # Escreve o arquivo.csv de saída ordenado no mesmo modelo do arquivo.csv de
     ↪  entrada
19   handler_person.writeCSV(lst_person, output_filename)
20
21   # Reporta o tempo final de execução do processo de ordenação
22   clear_screen()
23   chosen_algorithm: str = args.algorithm if (args.algorithm != None) else
     ↪  'quicksort'
24   handler_person.report_time(chosen_algorithm, lst_person.__len__(),
     ↪  start_time, finish_time)
```

Código Fonte 1: Função principal main().

Importante ressaltar é que todas as funções chamadas na *main()* são provenientes do módulo *handler_person.py*, que basicamente contém toda lógica comumente chamada "hard code"de execução da aplicação. Outro ponto é que caso não seja passado o algoritmo de ordenação na linha de comando pelo usuário, dado que este é um argumento opcional, o algoritmo default de ordenação é o *quicksort*, como é mostrado na linha 15 e 23 do Código Fonte 1. Entretanto na linha 15 é mais abstrato, pois quando não é passado como argumento a chave do algoritmo de ordenação é chamado internamente dentro da função *sort(lst_person)* do módulo *sort_collection.py* o algoritmo *quicksort*. O Código Fonte 2 mostra este caso, onde algoritmo de ordenação que é executado de acordo com opção escolhida.

```python
def sort(collection: list, algorithm_identifier: str = 'quicksort') -> None:
    '''
        Call a determined method by the algorithm identifier passed as argument
        and sort the collection passed by reference.
    '''

    if algorithm_identifier == 'selectsort':
        selectsort(collection)
    elif algorithm_identifier == 'insertsort':
        insertsort(collection)
    elif algorithm_identifier == 'mergesort':
        mergesort(collection)
    elif algorithm_identifier == 'quicksort':
        quicksort(collection)
    elif algorithm_identifier == 'heapsort':
        heapsort(collection)
    elif algorithm_identifier == 'introsort':
        introsort(collection)
    elif algorithm_identifier == 'timsort':
        timsort(collection)
    elif algorithm_identifier == 'smoothsort':
        raise NotImplementedError("Smoothsort is not implemented yet. :/")
    elif algorithm_identifier == 'patiencesort':
        raise NotImplementedError("Patiencesort is not implemented yet. :(")
    else:
        quicksort(collection)
```

Código Fonte 2: Função de execução do algoritmo escolhido sort().

Como se pode notar no Código Fonte 3 o arquivo de execução principal *build.py* também contém outros duas funções indicando tipos de execuções diferentes. Um deles é chamada de *run_without_args()*, o qual é utilizado para testes internos dos algoritmos de ordenação que o desenvolvedor/programador implementou. Já a outra forma de execução é invocada através de uma chamada de função, cujo nome é *run_analysis()*, o qual tem como principal objetivo a análise de um conjunto de dados de entrada, os quais seus resultados são persistidos em arquivos.csv de saída. Este tipo de execução será melhor discutido na Seção 4.

```
1  def main_without_args(lst_filenames: List[str]):
2      for filename in lst_filenames:
3          lstPerson: List[Person] = handler_person.readCSV(filename)
4          sort.quicksort(lstPerson)
5          print("Is sorted? %s" %(sort.isSorted(lstPerson)))
6          handler_person.show_people(lstPerson)
7
8  def run_analysis():
9      absolute_path: str =  os.path.dirname(os.path.abspath(__file__)) +
       ↪  "/files/input"
10     number_of_executions: int = 10
11     analisys_person.analyze(absolute_path, number_of_executions)
12
13 def choose_execution(execution_type: ExecutionType):
14     if (execution_type == ExecutionType.MAIN):
15         main()
16     elif (execution_type == ExecutionType.MAIN_WITHOUT_ARGS):
17         lstFileNames: list = [
18             "data_10e0.csv"
19         ]
20         main_without_args(lstFileNames)
21     elif (execution_type == ExecutionType.RUN_ANALYSIS):
22         run_analysis()
23     else:
24         raise Exception("Invalid execution type.")
25
26 def clear_screen():
27     try:
28         os.system('cls' if os.name=='nt' else 'clear')
29     except Exception:
30         pass
31
32 if __name__ == '__main__':
33     sys.tracebacklimit = 1
34     choose_execution(ExecutionType.MAIN)
```

Código Fonte 3: Outras formas de execução do módulo principal de execução build.py.

Note que a execução do algoritmo escolhido é selecionado na linha 34 do Código Fonte 3 através da mudança do classe enum chamada *ExecutionType.py*. Este enum é composto por três tipos de execuções diferentes, as quais já foram explicadas anteriormente. São elas:

- **MAIN** – Valor do enum que representa a execução principal a partir dos parâmetros passados na linha de comando pelo usuário;
- **MAIN_WITHOUT_ARGS** – Valor do enum que representa a execução voltada para testes internos do desenvolvedor sem utilização de comandos no terminal como é realizado na *main()*;

- **RUN_ANALYSIS** – Valor do enum que representa a execução voltada análise de dados de um conjunto de dados.

```
'''
    Class enum that represents a type of execution.
'''

from enum import Enum

class ExecutionType(Enum):
    MAIN = 1
    MAIN_WITHOUT_ARGS = 2
    RUN_ANALYSIS = 3
```

Código Fonte 4: Classe enum que representa a forma de execução no módulo principal build.py.

3.2 Leitura e Escrita de Arquivos

A manipulação dos arquivos são realizadas no módulo *handler_person.py*. Basicamente na leitura o arquivo é aberto e para cada linha do arquivo lido é instanciado um objeto da classe Person, este será discutido na sub-seção Estrutura de Dados, e adicionado em uma lista.

```python
def readCSV(input_filename: str) -> List[Person]:
    '''
        Read the content inside the filename passed as argument and transform
        to the Person objects.
    '''

    try:
        lstPerson: List[Person] = []
        with open(os.path.join(_INPUT_FILES_PATH, input_filename), 'r') as
        ↪  csvfile:
            dataCSV = csv.reader(csvfile, delimiter=',')

            # Ignora a primeira linha do arquivo, pois é o header
            next(dataCSV)

            # Percorre pelas linhas instanciando o objeto da classe Person
            ↪  armazenando dentro da lista
            for dataRow in dataCSV:
                lstPerson.append(Person(
                    dataRow[2], # id
                    dataRow[0], # email
                    dataRow[1], # gender
                    datetime.datetime.strptime(dataRow[3], "%Y-%m-%d"), #
                    ↪  birthdate
                    int(dataRow[4]), # height
                    int(dataRow[5]), # weight
                ))
    except FileNotFoundError as err:
        print(err)
        sys.exit(1)
    except IOError as err:
        print(err)
        sys.exit(1)
    except Exception as err:
        print(err)
        sys.exit(1)
    else:
        return lstPerson
```

Código Fonte 5: Leitura de arquivo através da função readCSV.

Já na escrita é aberto um arquivo, e para cada objeto lido em uma lista de objetos da classe Person é escrito no arquivo no mesmo formato do arquivo de entrada, porém de forma ordenada.

```python
def writeCSV(lst_person: List[Person], output_filename: str) -> None:
    '''Write the content in a CSV file for Person model.'''

    try:
        header: list = ["email", "gender", "uid", "birthdate", "height",
        ↪  "weight"]
        dataCSV: list = [header]
        for p in lst_person:
            dataCSV.append([
                p._email,
                p._gender,
                p._uid,
                p._birthdate.strftime("%Y-%m-%d"),
                p._height,
                p._weight
            ])
        with open(os.path.join(_OUTPUT_FILES_PATH, output_filename), 'w') as
        ↪  csvFile:
            writer = csv.writer(csvFile)
            writer.writerows(dataCSV)
    except IOError as err:
        print(err)
        sys.exit(1)
    except Exception as err:
        print(err)
        sys.exit(1)
```

Código Fonte 6: Escrita de arquivo através da função writeCSV.

3.3 Estruturas de Dados

Para modelar os dados que o programa deve manipular foi definida uma estrutura de dados, ou também chamada classe do Python, cujo nome é *Person*. Esta estrutura de dados é composta por seis campos, correspondentes as colunas dos arquivos CSV de entrada.

```
'''
    Class representing a Person
'''

from datetime import date, datetime

class Person:
    def __init__(self, id: str, email: str, gender: str, birthdate: date,
    ↪  height: int, weight: int):
        self._email: str = email
        self._gender: str = gender
        self._uid: str = id
        self._birthdate: date = birthdate
        self._height: int = height
        self._weight: int = weight

    @property
    def id(self) -> str:
        return self._uid

    @property
    def email(self) -> str:
        return self._email

    def compareTo(self, person) -> bool:
        '''
            Returns True if the self object(the caller) is less than the Person
            ↪  object passed as argument.
            Otherwise returns False
        '''
        if (self._uid < person._uid):
            return True
        return False

    def toString(self) -> str:
        '''A string representation to Person model'''
        return "UID: {0}\t Email: {1}\t Gender: {2}".format(self._uid,
        ↪  self._email, self._gender)
```

Código Fonte 7: Estrutura de dados `Person`.

O método de comparação da classe *Person*, chamado *CompareTo*, é responsável por comparar dois objetos desta classe, onde retorna o valor booleano *True* caso o objeto que chama o método (object caller) possua o valor do atributo *uid* menor que o objeto que é comparado, ou seja, o passado como argumento. Caso contrário retorna o valor booleano *False*. Foi adota esta forma de comparação dos objetos para se utilizar nos algoritmos de ordenação devido a simplicidade em relação as outras formas de comparação, os quais comumente retornam um valor do tipo *inteiro*.

3.4 Algoritmos de Ordenação

O Trabalho de Ordenação e Estatísticas de Ordem tem como requisito a implementação de sete algoritmos de ordenação diferentes. São eles: ordenação por seleção (*selection sort*), ordenação por inserção (*insertion sort*), *merge sort*, *quicksort*, *heapsort* e dois de quatro algoritmos que fica a critério de escolha do grupo, os quais os escolhidos foram: *introsort* e *timsort*.
Importante ressaltar que ambos algoritmos, tanto introsort quanto o timsort, são generalizações dos algoritmos obrigatórios, os quais possuem boa performance.

3.4.1 Algoritmo Selection sort

Segundo o Wikipédia o *Selection Sort* não possui melhor ou pior caso, tendo sua complexidade igual a $\Theta(n^2)$, tornando-o ineficiente em grandes vetores, sempre fazendo $(n^2 - n)/2$ comparações, independente do vetor está ordenado ou não.
A classificação de seleção é notada por sua simplicidade e possui vantagens de desempenho em relação a algoritmos mais complicados em determinadas situações, particularmente onde a memória auxiliar é limitada, visto que não utiliza um vetor auxiliar.

```
1  def selectsort(collection: list) -> None:
2
3    collection_size: int = collection.__len__()
4
5    for i in range(collection_size-1):
6        indiceMenor = i
7        k = i + 1
8
9        while (k < collection_size):
10           if (collection[k].compareTo(collection[indiceMenor])):
11               indiceMenor = k
12           k += 1
13
14       if (indiceMenor != i):
15           aux = collection[i]
16           collection[i] = collection[indiceMenor]
17           collection[indiceMenor] = aux
```

Código Fonte 8: Implementação do algoritmo Selection Sort.

3.4.2 Algoritmo Insertion sort

O *Insertion sort* é um algoritmo em que a ideia está em "inserir o elemento na posição correta do vetor". No algoritmo é percorrido as posições do vetor começando a partir da segunda posição, onde a cada nova interação o elemento da posição atual deve ser inserido na posição correta do sub-vetor ordenado à esquerda da dada posição, caso este esteja sendo ordenado de modo crescente.
Assim como o *Selection sort*, ele também é um algoritmo de ordem quadrática, possuindo complexidade assintótica igual a $\Theta(n^2)$ no pior caso e $\Theta(n)$ no melhor caso, sendo um dos mais eficientes desta ordem de complexidade.

```
def insertsort(collection: list) -> None:

  collection_size: int = collection.__len__()

  for i in range(1, collection_size):
      k: int = i-1
      aux: object = collection[i]

      while (k >= 0 and aux.compareTo(collection[k])):
          collection[k+1] = collection[k]
          k -= 1

      if (k+1 != i):
          collection[k+1] = aux
```

Código Fonte 9: Implementação do algoritmo Insertion Sort.

3.4.3 Algoritmo Merge sort

O *Merge Sorte* é um algoritmo do tipo divisão e conquista, que consiste em dividir o problema principal em subproblemas e resolve-los através da recursividade, unindo a resolução deles para solucionar o problema original.
Foi criado por John Von Neaumann em 1945 e funciona dividindo o vetor em n sub-vetores, contendo 1 elemento, depois mescla-se sub-vetores para produzir novos até restar apenas 1 com os elementos iniciais ordenados.
O algoritmo possui a complexidade de $\Theta(nlog_2n)$, onde ele é estável na maioria das implementações podendo ser interativa ou recursiva. A desvantagem do *Merge Sorte* é o gasto extra de memória com as cópias do vetor na chamada recursiva.

```
def mergesort(collection: list) -> None:

    collection_size: int = collection.__len__()

    if (collection_size > 1):
        half: int = collection_size // 2

        lst_aux_left: list = collection[:half]
        lst_aux_right: list = collection[half:]

        mergesort(lst_aux_left)
        mergesort(lst_aux_right)
        __merge(collection, lst_aux_left, lst_aux_right)
```

Código Fonte 10: Implementação do algoritmo Merge Sort.

```
def __merge(collection: list, lst_aux_left: list, lst_aux_right: list):
    i = j = k = 0

    while (i < lst_aux_left.__len__()) and (j < lst_aux_right.__len__()):
        if(lst_aux_left[i].compareTo(lst_aux_right[j])):
            collection[k] = lst_aux_left[i]
            i += 1
        else:
            collection[k] = lst_aux_right[j]
            j += 1
        k += 1

    for l in range(i, lst_aux_left.__len__()):
        collection[k] = lst_aux_left[l]
        k += 1

    for r in range(j, lst_aux_right.__len__()):
        collection[k] = lst_aux_right[r]
        k += 1
```

Código Fonte 11: Implementação do algoritmo Merge Sort(2).

3.4.4 Algoritmo Quicksort

O *Quicksort*, assim como o *Merge Sort*, adota a estratégia de divisão e conquista. Ele foi inventado em 1960 por Tony Hoare enquanto trabalhava num projeto de tradução de máquina, tentando traduzir um dicionário de inglês para russo.
Funciona escolhendo um elemento chamado de pivô, depois particiona a vetor de forma que os elementos com valor menor que o pivô fique em posições anteriores e as de os de maior valor posteriores, recursivamente é aplicada a operação anterior aos sub-vetores de menor valores e de maior valores.
Possui uma complexidade no melhor caso e caso médio de $\Theta(nlog_2n)$, contém o laço interno simples, porém tem a escolha do pivô como caso crítico, podendo tornar a complexidade do algoritmo quadrática.

```
def quicksort(collection: list) -> None:
    __quicksort(collection, 0, len(collection) - 1)

def __quicksort(collection: list, left: int, right: int) -> None:
    if (left < right):
        partitionI: int = __partition(collection, left, right)
        __quicksort(collection, left, partitionI - 1)
        __quicksort(collection, partitionI + 1, right)
```

Código Fonte 12: Implementação do algoritmo Quicksort.

```
def __swap(collection: list, i: int, j: int) -> None:
    collection[i], collection[j] = collection[j], collection[i]
```

Código Fonte 13: Implementação da função swap).

```
def __partition(collection: list, left: int, right: int) -> int:
    pivot = collection[right]
    smallIndex: int = left - 1
    for i in range(left, right):
        if (collection[i].compareTo(pivot)):
            smallIndex += 1
            __swap(collection, smallIndex, i)
    __swap(collection, smallIndex + 1, right)

    return smallIndex + 1
```

Código Fonte 14: Implementação da função particion.

3.4.5 Algoritmo Heapsort

O *Heapsort* foi desenvolvido em 1964 por RW Floyd e JWJ Williams, utiliza a estrutura de dados heap para ordenar os elementos.
O algoritmo possui o mesmo principio que o *Selection Sort*, tem como ideia construir com o vetor uma max-heap, onde o primeiro nó é o de maior valor, trocar a raiz com o elemento de ultima posição do vetor, diminuir o tamanho em 1, rearranjar o max-heap e repetir o processo n-1 vezes.
Tem sua complexidade no pior e melhor caso de $\Theta(nlog_2n)$, onde comparado ao *Quicksort* não é tão rápido visto que seu laço interno realiza mais operações que o particionamento.

```
def heapsort(collection: list) -> None:
    __buildHeap(collection)
    for i in range(len(collection) - 1, -1, -1):
        __swap(collection, 0, i)
        __heapify(collection, i, 0)
```

Código Fonte 15: Implementação do algoritmo Heapsort.

```
def __buildHeap(collection: list) -> None:
    for i in range(math.floor(len(collection) / 2), -1, -1):
        __heapify(collection, len(collection), i)
```

Código Fonte 16: Implementação da função buildHeap.

```
def __heapify(collection: list, length: int, i: int) -> None:
    left: int = 2 * i
    right: int = 2 * i + 1
    largest: int = i

    if (left < length) and (collection[largest].compareTo(collection[left])):
        largest = left

    if (right < length) and (collection[largest].compareTo(collection[right])):
        largest = right

    if (largest != i):
        __swap(collection, i, la\emph{rgest)
        }__heapify(collection, length, largest)
```

Código Fonte 17: Implementação da função Heapify.

3.4.6 Algoritmo Introsort

Criado por David Musser em 1997 o *Introsort* é um algoritmo de ordenação híbrido baseado no *Quicksort* e no *Heapsort* tendo a complexidade média de $\Theta(nlog_2n)$.
Começa com o *Quicksort* e alterna para o *Heapsort* quando o pivô excede o limite de profundidade (log_2n), assim combina as partes boas de ambos os algoritmos.

```
def introsort(collection: list) -> None:
    n: int = len(collection)
    depthLimit: int = math.log(n, 2)
    pivot: int = n-1

    if n < 2:
        return
    if pivot > depthLimit:
        heapsort(collection)
        return

    introsort(collection[0:pivot])
    introsort(collection[pivot+1:n])
```

Código Fonte 18: Implementação do algoritmo Introsort.

3.4.7 Algoritmo Timsort

Inventado em 2002 por Tim Peters para ser usado em Python, e tem sido o algoritmo padrão desde a versão 2.3, o Timsort é um algoritmo hibrido, derivado do *Merge Sorte* e do *Insertion Sort*.
Basicamente o vetor é dividido em sub-vetores que são ordenados usando o *Insetion Sort* estável e esses sub-vetores ordenados são mesclados com o *MergeSort*.
Esse algoritmo possui a complexidade em seu melhor caso linear, ou seja $O(n)$, e seu pior caso tem complexidade de $\Theta(nlog_2n)$.

```python
def timsort(collection: list) -> None:
    lst_particoes: list = []
    size_collection: int = len(collection)
    RUN = 32 if size_collection > 32 else size_collection

    for i in range(0, size_collection, RUN):
        aux: list = collection[i:i+RUN]
        insertsort(aux)
        lst_particoes += aux

    size: int = RUN
    while size <= size_collection:
        for left in range(0, size_collection, 2 * size):
            middle: int = left + size - 1
            right: int = min(left + 2 * size, size_collection)
            __merge(collection, lst_particoes[left:middle],
            ↪  lst_particoes[middle:right])
        size = 2 * size
```

Código Fonte 19: Implementação do algoritmo Timsort.

4 Análise de Desempenho dos Algoritmos

Como é possível ver no Código Fonte 20, a linha 2 coleta o tempo do sistema imediatamente antes de executar o algoritmo de ordenação selecionado na linha de comando pelo usuário, e imediatamente após a execução do algoritmo de ordenação o tempo do sistema é coletado novamente, na linha 4. O tempo transcorrido, que é medido em milissegundos, como é demonstrado Código Fonte 21, é relatado ao final da execução do programa, o qual já foi apresentado na Figura 2.

```
1 # Mensura o tempo de execução da CPU do processo de ordenação feita pelo
  ↪ algoritmo passado como argumento proveniente do CLI
2     start_time: float = handler_person.get_cpu_time()
3     sort.sort(lst_person, args.algorithm) if (args.algorithm != None) else
      ↪ sort.sort(lst_person)
4     finish_time: float = handler_person.get_cpu_time()
```

Código Fonte 20: Trecho do código main() da coleta do tempo de processamento do algoritmo de ordenação selecionado.

```
1 def get_cpu_time() -> float:
2     '''Returns the CPU time in milliseconds(ms) of the current process.'''
3
4     return time.process_time() * 1000
```

Código Fonte 21: Função que coleta o tempo de CPU do processo.

Entretanto pode ocorrer variação no tempo de execução do algoritmo devido uma série de fatores externos do código implementado. Por isso, a forma mais segura de analisar a performance é executar cada código do algoritmo várias vezes para um mesmo arquivo de entrada e trabalhar com a média dos tempos coletados. As seções seguintes descrevem como foi feita a coleta dos dados de tempo de execução através do script criado e como esses dados foram analisados. Todos os testes foram realizados em um processador Intel Core i7 com 16Gb de RAM.

4.1 Coleta e Pré-processamento de Dados

Como dito anteriormente, cada algoritmo de ordenação – no caso o *selection sort*, *insertion sort*, *merge sort*, *quicksort*, *heapsort*, *introsort* e *timsort* – foi executado diversas vezes sobre cada um dos arquivos de dados e os tempos de execução reportados foram coletados e armazenados em uma estrutura de arquivos, que serão detalhadas nas subseções posteriores relativos ao script/código implementado para a realização da análise de dados.

4.1.1 Script Para Coleta dos Dados

O script foi implementado na linguagem Python e com o foco de somente o desenvolvedor da aplicação de utilizá-lo. O intuito é na coleta automatizada dos processamentos realizados dos arquivos de um dado diretório que é fornecido para o script. O Código Fonte 22

mostra como é feita a chamada do script, que possui todo o código implementado no módulo *analisys_person.py*.

```
1  def run_analysis():
2      absolute_path: str =  os.path.dirname(os.path.abspath(__file__)) +
       ↪  "/files/input"
3      number_of_executions: int = 10
4      analisys_person.analyze(absolute_path, number_of_executions)
5
6  def choose_execution(execution_type: ExecutionType):
7      if (execution_type == ExecutionType.MAIN):
8          main()
9      elif (execution_type == ExecutionType.MAIN_WITHOUT_ARGS):
10         lstFileNames: list = [
11             "data_10e0.csv"
12         ]
13         main_without_args(lstFileNames)
14     elif (execution_type == ExecutionType.RUN_ANALYSIS):
15         run_analysis()
16     else:
17         raise Exception("Invalid execution type.")
18
19 def clear_screen():
20     try:
21         os.system('cls' if os.name=='nt' else 'clear')
22     except Exception:
23         pass
24
25 if __name__ == '__main__':
26     sys.tracebacklimit = 1
27     choose_execution(ExecutionType.RUN_ANALYSIS)
```

Código Fonte 22: Chamada do script de análise dos dados.

Note na linha 27 do Código Fonte 22 que a chamada do script é feita na inicialização do módulo passando como argumento o tipo de execução (*RUN_ANALYSIS*), como é discutido na Subseção 3.1 onde é explicado as formas de execução do módulo principal da aplicação. Assim que a função *run_analysis()* é chamada note que na linha 2 é o diretório onde se encontram os arquivos e na linha 3 o número de execuções que serão feitas para coleta do tempo para cada arquivo do diretório. Por fim na linha 4 é feita a chamada da função que *analyze* do módulo *analisys_person* responsável pela análise dos dados.

4.1.2 Análise dos dados através da função analyze

Esta função é a única de acesso pública dentro do módulo *analisys_person*, cujo objetivo é a realização da análise de dados acerca da classe *Person*, que é a estrutura de dados principal de análise. Basicamente a função recebe dois argumentos, estes são: o caminho absoluto do diretório que contém os arquivos necessários para análise e a quantidade de execuções que serão realizadas para cada arquivo. Quando terminada a análise cria um

diretório chamado *./src/files/analyze* que contém todos os arquivos gerados pela análise e retorna um valor booleano determinando se foi feita a correta análise. O Código Fonte 23 e Código Fonte 26 mostra a implementação da função *analyze()*, onde é possível notar que são feitas chamadas de várias funções com a notação duplo underline, que corresponde funções de visibilidade privada nas convenções de código da linguagem utilizada, o qual no caso do projeto é *Python*.

```
1  def analyze(directory_path: str, times_of_execution: int) -> bool:
2      '''
3          parses data from directory files passed as argument. When done, it
   ↪  creates a folder
4          named input/analyze, where it contains all the files generated by the
   ↪  analysis.
5          returns true if the algorithm can do the correct analysis. Otherwise
   ↪  returns false.
6      '''
7
8      hifens: str = "-" * 10
9      print("NUMBER OF EXECUTIONS BY ALGORITHM IS: %d\n" %(times_of_execution))
10     print("%sSTARTING ANALISYS%s" %(hifens, hifens))
11
12     dic_sorting_choices: Dict[str, str] = sort.ALGORITHMS_SORTING_CHOICES
13     lst_sorting_choices: List[str] = list(dic_sorting_choices.keys())
14     QUICKSORT_KEY = "quicksort"
15     filenames: List[str] = __get_all_filenames_from_directory(directory_path,
       ↪  True)
16
17     ## SETANDO O TIME OUT ATRAVÉS DO MAIOR ARQUIVO ##
18     print("\n%sCALCULATING TIMEOUT%s" %(hifens, hifens))
19     biggest_file_records: str = max(filenames, key=(lambda f:
       ↪  os.path.getsize(os.path.join(directory_path, f))))
20     lst_person: List[Person] = __readCSV_person(os.path.join(directory_path,
       ↪  biggest_file_records))
21     times_execution: float = __calculate_time_sort_algorithm(lst_person,
       ↪  QUICKSORT_KEY)
22     TIME_OUT: float = times_execution * 10
23     print("{0}timeout = {1}(ms){0}\n".format(hifens, TIME_OUT))
```

Código Fonte 23: Função analyze(1).

Note que no Código Fonte 23 a linha 12 é recuperado todos os tipos de algoritmos de ordenação em uma estrutura de dicionário, os quais são os sete mencionados anteriormente. Já na linha 15 é feita a chamada da função responsável pegar os nomes de todos os arquivos do diretório absoluto (*directory_path*) passado como argumento para serem lidos e analisados. Já entre as linhas 18 e 23 é o trecho do código responsável por calcular o tempo de *time-out* baseado no arquivo de maior tamanho, onde este também é o arquivo de maior quantidade de registros. Perceba que na linha 21 é chamada a função que realiza a leitura dos registros no arquivo e transformados em uma lista de objetos da classe *Person* e logo após calculado seu tempo de execução. O algoritmo executado para

definir o tempo de *time-out* do programa é *quicksort*. Na linha 22 é definido a variável *TIME_OUT*, cujo é o tempo de execução multiplicado por dez definindo um tempo limite para a término de execução de um algoritmo que chegue a este limite.

```
1 def __get_all_filenames_from_directory(path: str, orderble: bool = False) ->
  ↪ List[str]:
2     lst_filenames: List[str] = [f for f in os.listdir(path) if
      ↪  os.path.isfile(os.path.join(path, f))]
3     if orderble:
4         lst_filenames.sort()
5 
6     return lst_filenames
```

Código Fonte 24: Função que recupera o nome de todos os arquivos de um determinado diretório.

```
1 def __calculate_time_sort_algorithm(lst_person: List[Person], algorithm_key:
  ↪ str) -> float:
2     start_time: float = time.process_time() * 1000
3     sort.sort(lst_person, algorithm_key)
4     finish_time: float = time.process_time() * 1000
5 
6     return finish_time - start_time
```

Código Fonte 25: Função que calcula o tempo de execução do algoritmo de ordenação.

Após a definição do tempo de *time-out* são executados todos os algoritmos de ordenação igual ao número de vezes determinada pelo valor da variável *times_of_execution*. Ao término de execução de cada arquivo de entrada é criado um arquivo de saída para cada algoritmo contendo o valor do tempo de execução para cada execução feita, assim como a média final. Entre as linhas 5 e 22 do Código Fonte 26 correspondem ao trecho de código que realiza o tipo de análise citado.

```
# Registra os tempos médios de execução dos algoritmos de ordenação
dic_register_time_execution: Dict[str, Dict[int, float]] = {}

## CRIAR ARQUIVOS COM OS REGISTROS E MÉDIA FINAL P/ CADA ALGORITMO COM CADA REGISTRO RODANDO TIME_OF_EXECUTIONS VEZES ##
for filename in filenames:
    # Coloca os objetos em cache para ser em uma lista de Person
    lst_person: List[Person] = __readCSV_person(os.path.join(directory_path, filename))

    # Faz a análise para todos os algoritmos de ordenção presentes na lista
    for algorithm_key in lst_sorting_choices:

        print("{0} records running for {1}...".format(lst_person.__len__(), dic_sorting_choices[algorithm_key].upper()))

        dic_time_execution: float = __register_average_time_execution(lst_person, algorithm_key, times_of_execution, TIME_OUT)

        if algorithm_key in dic_register_time_execution:
            dic_register_time_execution[algorithm_key][len(lst_person)] = dic_time_execution
        else:
            dic_register_time_execution[algorithm_key] = { len(lst_person): dic_time_execution }

        print("Average time result: {0}ms".format(dic_time_execution["average"]))
        print(hifens*10)

## CRIAR ARQUIVOS COM A MÉDIA DE EXECUÇÃO FINAL E SUA QUANTIDADE DE REGISTROS ##
__register_average_times_per_num_executions(dic_register_time_execution)

## CRIAR UM ARQUIVO FINAL COM TODOS OS ALGORITMOS JUNTOS COM SEUS RESPECTIVOS
## TEMPOS MÉDIO DE EXECUÇÃO POR NÚMERO DE REGISTROS
__register_all_algorithms_average_times_per_num_executions(dic_register_time_execution, dic_sorting_choices, TIME_OUT)

print("%sANALISYS FINISHED%s" %(hifens, hifens))

return True
```

Código Fonte 26: Função analyze(2).

Todos esses arquivos são criados no diretório */src/files/analyze/average_times_each_record*, como é mostrado na linha 10 do Código Fonte 28.

```python
def __register_average_time_execution(lst_person: List[Person], algorith_key:
    str, times_of_execution: int, time_out: float) -> Dict[str, float]:
    # Realiza os cálculos de média
    dataCSV_execution_times: List[object] = []
    lst_time_execution: List[float] = []
    count_timeout_times = 0
    limit_of_timeout = 1

    for execution in range(1, times_of_execution + 1):
        __call_process_timeout(time_out, __calculate_time_of_sort_algorithm,
        [lst_person, algorith_key])
        time_execution: float = _queue_process.get_nowait() if not
        _queue_process.empty() else time_out
        dataCSV_execution_times.append([execution, time_execution])
        lst_time_execution.append(time_execution)

        if time_execution == time_out:
            count_timeout_times += 1
            print("\nTIME-OUTED\n")
            if count_timeout_times == limit_of_timeout:
                lst_time_execution += ([time_out] * (times_of_execution -
                limit_of_timeout))

                while execution <= time_execution:
                    execution += 1
                    dataCSV_execution_times.append([execution, time_out])

                break

    average_time: float = sum(lst_time_execution) / times_of_execution

    dic_result: Dict[str, float] = {
        "average": average_time,
        "max": max(lst_time_execution),
        "min": min(lst_time_execution)
    }
```

Código Fonte 27: Função de registro dos tempos médios execução para cada arquivo de entrada lido(1).

```
# Cria os dados para serem escritos no arquivo .csv de saída
header: list = ["Execution", "Average Time(ms)"]
final_result: list = ["Final Average", average_time]
dataCSV: list = dataCSV_execution_times
dataCSV.insert(0, header)
dataCSV.append(final_result)

# Seta o nome do arquivo e diretório, onde caso não esteja criado ainda é
↪ criado
filename: str = "{0}_{1}{2}".format(algorith_key, lst_person.__len__(),
↪ __EXTENSION_FILES)
directory: str = os.path.join(__OUTPUT_ANALYZE_FILES_PATH,
↪ "average_times_each_record")
os.makedirs(directory, exist_ok=True)

# Chama a função genérica que escreve um arquivo
__writeCSV_analisys_person(dataCSV, os.path.join(directory, filename))

return dic_result
```

Código Fonte 28: Função de registro dos tempos médios execução para cada arquivo de entrada lido(2).

Pode-se notar que os Código Fonte 27 e Código Fonte 28 são as partes mais "complexas" do script. Porém a função basicamente realiza a execução do algoritmo pela sua key, onde é passado como argumento no número de vezes definida pela variável *times_of_execution* correspondente ao trecho de código exibido pelo Código Fonte 27, além da criação do arquivo resultante dos dados coletados de cada processamento realizado correspondente ao treco do código exibido pelo Código Fonte 28.
Continuando a lógica no Código Fonte 26 na linha 25 é feita a chamada da função que cria sete arquivos, onde cada um contém um nome de um algoritmo de ordenação implementado. Neles são contidos o tempo médio de execução da cada quantidade de registros presentes nos arquivos, assim como seus valores máximos e mínimos. Todos estes arquivos são criados no diretório */src/files/analyze/average_times_numbers_records*, como é mostrado na linha 23 do Código Fonte 29.
O Código Fonte 29 mostra a implementação desta função, que utiliza os dados previamente coletados e registrados dentro da estrutura de dados dicionário (*dic_register_time_execution*) passado como argumento.

```
1  def __register_average_times_per_num_executions(dic_register_time_execution:
   ↪  Dict[str, Dict[int, object]]) -> None:
2      for algorithm_key in dic_register_time_execution.keys():
3          dic_algorithm_executions: Dict[int, Dict[str, float]] =
           ↪  dic_register_time_execution[algorithm_key]
4
5          # Pega as chaves que corresponde a número de registros e ordena de
           ↪  forma crescente
6          lst_num_records: List[int] = list(dic_algorithm_executions.keys())
7          lst_num_records.sort()
8
9          # Setando os dados para o CSV
10         header: list = ["Total Records", "Average Time Execution(ms)", "Minimum
           ↪  Value(ms)", "Maximum Value(ms)"]
11         dataCSV: list = [header]
12
13         for num_records in lst_num_records:
14             dataCSV.append([
15                 num_records,
16                 dic_algorithm_executions[num_records]["average"],
17                 dic_algorithm_executions[num_records]["min"],
18                 dic_algorithm_executions[num_records]["max"]
19             ])
20
21         # Seta o nome do arquivo e diretório, onde caso não esteja criado ainda
           ↪  é criado
22         filename: str = "{0}{1}".format(algorithm_key, __EXTENSION_FILES)
23         directory: str = os.path.join(__OUTPUT_ANALYZE_FILES_PATH,
           ↪  "average_times_numbers_records")
24         os.makedirs(directory, exist_ok=True)
25
26         # Escreve o arquivo de saída
27         __writeCSV_analisys_person(dataCSV, os.path.join(directory, filename))
```

Código Fonte 29: Função que registra número registros X tempo médio de execução para cada algoritmo de ordenação.

Por fim seguindo a lógica de execução do programa do Código Fonte 26 na linha 29 é feito o último e único arquivo, chamado *analisys_all_algorithms_average.csv*, que contém a junção dos arquivos gerados na etapa anterior discutida, ou seja, gerado um arquivo final com todos os algoritmos, contendo seus tempos de execuções médio em cada quantidade de registro dos arquivos lidos.
O Código Fonte 30 mostra a implementação desta função, que também utiliza os dados previamente coletados e registrados dentro da estrutura de dados dicionário (*dic_register_time_execution*) passado como argumento.

```
def __register_all_algorithms_average_times_per_num_executions(dic_register_time_execution: Dict[str, Dict[int, object]], dic_sorting_choices: Dict[str, str], timeout: float) -> None:

    if not dic_register_time_execution:
        return

    # Primeira linha contém as informações de número de registro em ordem crescente
    lst_keys_algorithms: List[dict] = list(dic_register_time_execution.keys())
    lst_num_records: List[int] = list(dic_register_time_execution[lst_keys_algorithms[0]].keys())
    lst_num_records.sort()

    dataCSV: list = []

    # Monta os dados para cada algoritmo em cada linha do dataCSV
    for algorithm_key in dic_register_time_execution.keys():
        data_row: List[object] = []

        # Adicionando os valores de média
        for num_records in lst_num_records:

            data_row.append(dic_register_time_execution[algorithm_key][num_records]["ave

        # Adicionando a qual algoritmo corresponde esses valores de média
        data_row.insert(0, dic_sorting_choices[algorithm_key])
        dataCSV.append(data_row)

    # Monta a header
    header: list = lst_num_records
    header.insert(0, "")
    dataCSV.insert(0, header)

    # Adicionando o valor de timeout
    dataCSV.append(["Timeout", timeout])

    filename: str = "analisys_all_algorithms_average{0}".format(__EXTENSION_FILES)

    # Escreve o arquivo de saída
    __writeCSV_analisys_person(dataCSV, os.path.join(__OUTPUT_ANALYZE_FILES_PATH, filename))
```

Código Fonte 30: Função que registra número registros X tempo médio de execução de todos os algoritmos de ordenação em um único arquivo.

4.2 Análise dos Resultados

A análise dos resultados foram realizadas através dos arquivos de saída obtidos na execução do script discutido na sub-seção anterior. As sub-seções posteriores mostrará as análises feitas em um conjunto de dados para cada algoritmo de ordenação e por fim uma análise detalhada de todos os algoritmos juntos em um mesmo gráfico.
Importante ressaltar que todo o conjunto de dados dos arquivos foram lidos 10 vezes para cada arquivo e mostrados em formas de tabelas e gráficos, cuja a quantidade de dados em cada arquivo são de: 10, 25, 75, 100, 250, 500, 750, 1000, 2500, 5000, 7500, 10000, 25000, 50000, 75000, 100000, 250000, 500000 1000000, 2500000, 5000000, 7500000. Já o tempo de execução medido em milissegundos (ms).
Como dito na anteriormente o cálculo do *Timeout* foi baseado no tempo de execução do algoritmo *Quicksort* ao ordenar a maior quantidade de dados, multiplicado por 10. Caso algum algoritmo atingisse esse tempo por 2 vezes seguidas estipulou-se que as demais execuções para aquela entrada também iriam atingir. Para a execução à qual será feita a análise dos resultados o *Timeout* foi igual à 912242.2771 milissegundos (pouco mais de 15 minutos).

4.2.1 Selection Sort

Selection sort			
Total Records	**Average (ms)**	**Minimum (ms)**	**Maximum (ms)**
10	0.0828504	0.069963	0.110183
25	0.1933319	0.149224	0.297292
50	0.5755221	0.416881	1.240977
75	1.0387112	0.851929	1.350753
100	2.5734205	1.5357	4.722715
250	11.6573716	10.139555	13.306141
500	31.9207071	29.020227	33.364834
750	62.8736719	60.121146	65.52321
1000	108.535861	106.540523	109.974755
2500	658.0068997	655.659046	660.589199
5000	2655.4927214	2649.686164	2660.817057
7500	5928.7287433	5918.321673	5937.188756
10000	10715.3564662	10587.402281	11168.398103
25000	74630.3859784	72349.647243	75721.90482
50000	336948.028985	335714.02217	339034.481072
75000	834708.8484096	829767.987117	845853.732132
100000	912242.2771	912242.2771	912242.2771
250000	912242.2771	912242.2771	912242.2771
500000	912242.2771	912242.2771	912242.2771
750000	912242.2771	912242.2771	912242.2771
1000000	912242.2771	912242.2771	912242.2771
2500000	912242.2771	912242.2771	912242.2771
5000000	912242.2771	912242.2771	912242.2771
7500000	912242.2771	912242.2771	912242.2771
TIMEOUT (ms)	**912242.2771**		

Tabela 1: Estatísticas dos tempos de execução do algoritmo Selection Sort.

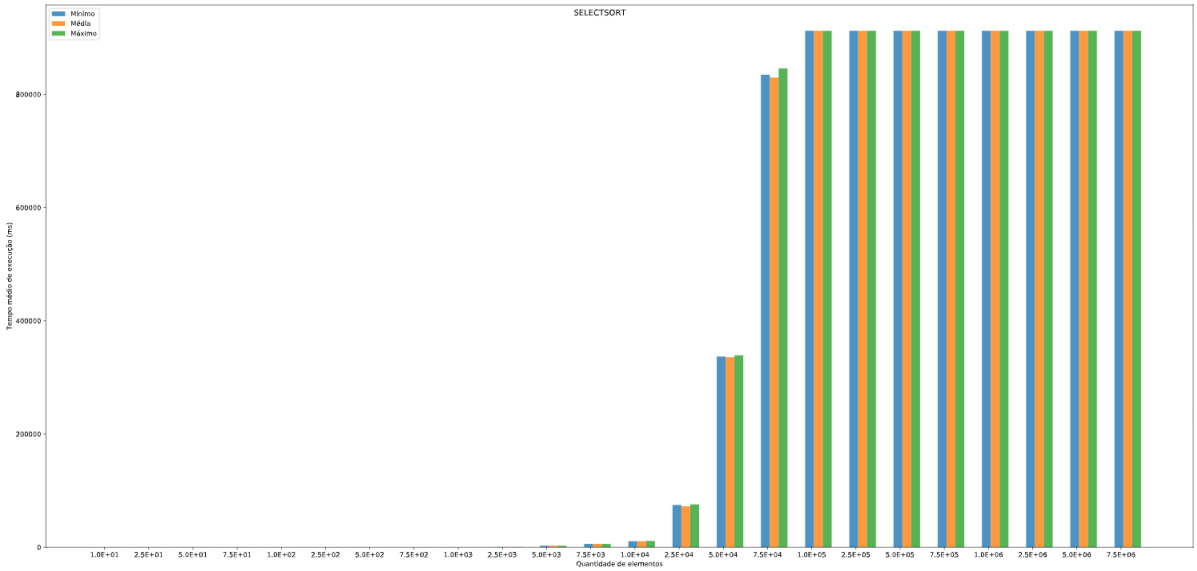

Figura 3: Gráfico do tempo de execução do Selection Sort.

O *Selection Sort* é um algoritmo com complexidade de $\Theta(n^2)$. Tem um bom desempenho na ordenação com baixa quantidade de dados, porém em torno de 7500 dados se torna lento demorando em torno de 6 segundos para efetuar a ordenação como é mostrado na Figura 3. Atinge o timeout com entradas a partir de 100000 dados, mostrado tanto na Tabela 1 quanto no gráfico Figura 3.

4.2.2 Insertion Sort

Insertion sort			
Total Records	**Average (ms)**	**Minimum (ms)**	**Maximum (ms)**
10	0.1120711	0.09474	0.178678
25	0.2032633	0.172568	0.257297
50	0.4413823	0.36414	0.504067
75	0.8520973	0.516946	1.858004
100	1.472551	1.247374	1.690347
250	6.7688062	3.972162	8.0986
500	20.0637704	18.353599	22.847153
750	37.060291	34.557909	39.435907
1000	65.166984	63.648203	66.486693
2500	379.323529	374.093351	381.275326
5000	1567.0939451	1562.351505	1572.023938
7500	3423.6631442	3411.568079	3434.211072
10000	6182.5665528	6169.440604	6198.802763
25000	41031.8505264	40101.032099	41858.999626
50000	194103.8127202	193071.471961	195896.895832
75000	533279.2042283	497810.514171	565872.946893
100000	912242.2771	912242.2771	912242.2771
250000	912242.2771	912242.2771	912242.2771
500000	912242.2771	912242.2771	912242.2771
750000	912242.2771	912242.2771	912242.2771
1000000	912242.2771	912242.2771	912242.2771
2500000	912242.2771	912242.2771	912242.2771
5000000	912242.2771	912242.2771	912242.2771
7500000	912242.2771	912242.2771	912242.2771
TIMEOUT (ms)	**912242.2771**		

Tabela 2: Estatísticas dos tempos de execução do algoritmo Insertion Sort.

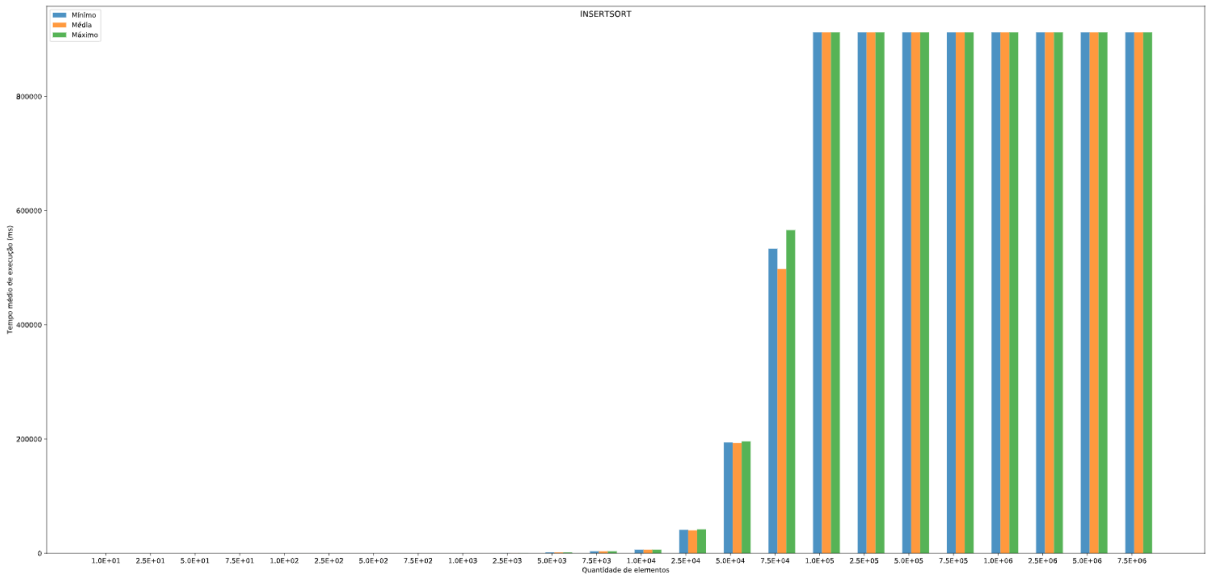

Figura 4: Gráfico do tempo de execução do Insertion Sort.

O *Insertion Sort* assim como o *Selection Sort* possui a complexidade de $\Theta(n^2)$ no pior caso, levando vantagem sobre o Selection a partir da entrada com 100 linhas, porém o acompanha atingindo timeout com entradas a partir de 100000 registros, como é mostrado na Figura 4. Também começa ter perfomance ruim a partir de 7500 demorando em torno de 3,5 segundos para realizar o processo de ordenação, porém tem praticamente o mesmo custo de tempo quando se ordena 10000 registros em comparação ao *Selection Sort* com 7500 registros.

4.2.3 Merge Sort

Merge sort			
Total Records	**Average (ms)**	**Minimum (ms)**	**Maximum (ms)**
10	0.1683978	0.141905	0.279886
25	0.2838896	0.235523	0.375677
50	0.4861434	0.254198	0.572319
75	0.8254709	0.562473	1.180616
100	1.1987972	0.675328	1.493218
250	2.5644491	1.988769	4.136654
500	5.5758295	2.490089	7.169537
750	7.8216197	5.633572	9.598888
1000	10.3667951	7.792096	12.066417
2500	20.6688922	18.983586	22.557514
5000	38.687849	37.041548	40.14058
7500	55.9333884	54.173462	57.742492
10000	75.2054318	72.98262	78.254502
25000	198.7678439	197.796177	200.102851
50000	426.2855839	424.448011	432.385206
75000	675.0263716	670.581511	676.659618
100000	928.5264272	925.210533	931.294642
250000	2528.4812708	2525.267578	2534.629711
500000	5490.5929088	5478.462866	5503.902335
750000	8699.2297855	8686.683393	8719.084283
1000000	11764.5924389	11740.997449	11789.598829
2500000	32729.8362959	32688.09754	32762.559038
5000000	69787.5443332	69655.880595	70030.895013
7500000	108298.0473089	108180.681946	108431.106465
TIMEOUT (ms)	**912242.2771**		

Tabela 3: Estatísticas dos tempos de execução do algoritmo Merge Sort.

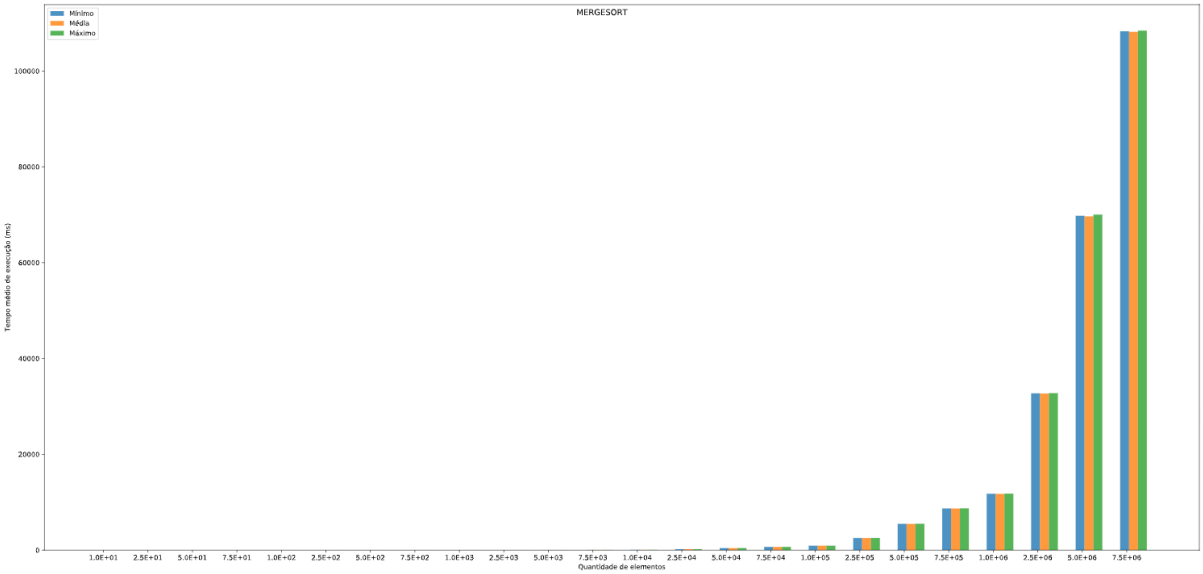

Figura 5: Gráfico do tempo de execução do Merge Sort.

O *Merge Sort* possui a complexidade de $\Theta(nlog_2n)$, conseguindo executar todas as entradas sem exceder o limite de timeout, como é mostrado na Figura 5, ficando atrás dos algoritmos anteriores apenas em pequenas entradas de dados, menores que 100 linhas. Começa a ter um tempo considerável de ordenação a partir de 250000 registros demorando em torno de 2,5 segundos para ordenação completa e em torno de 1 minuto e 40 segundos para a maior quantidade de registros, no caso é 7500000 registros.

4.2.4 QuickSort

Quicksort			
Total Records	**Average (ms)**	**Minimum (ms)**	**Maximum (ms)**
10	0.1131264	0.084512	0.12881
25	0.1672624	0.129569	0.191038
50	0.3126027	0.255308	0.412176
75	0.5062997	0.235879	0.88262
100	0.8331159	0.50738	1.035755
250	1.3462559	0.77812	1.85292
500	2.6332269	1.563893	3.493021
750	4.7538351	2.910883	7.61571
1000	5.2536694	3.530716	7.430321
2500	14.3068495	9.583293	18.039828
5000	27.3967887	22.305562	28.429353
7500	38.4377591	33.903364	40.604692
10000	52.2457401	47.523859	55.061196
25000	135.4775507	132.101726	137.543564
50000	323.8354155	319.502511	326.613223
75000	524.6261798	520.174226	548.018658
100000	666.4791089	664.179678	668.03862
250000	1926.386345	1918.338705	1938.996988
500000	4169.9041362	4157.286023	4183.067318
750000	6622.5045697	6612.971718	6633.949132
1000000	9224.7008268	9213.974946	9239.84031
2500000	26252.8115843	26182.726456	26300.664442
5000000	59293.2579544	59238.205888	59437.602533
7500000	92824.1835982	92687.164013	93067.550069
TIMEOUT (ms)	**912242.2771**		

Tabela 4: Estatísticas dos tempos de execução do algoritmo Quicksort.

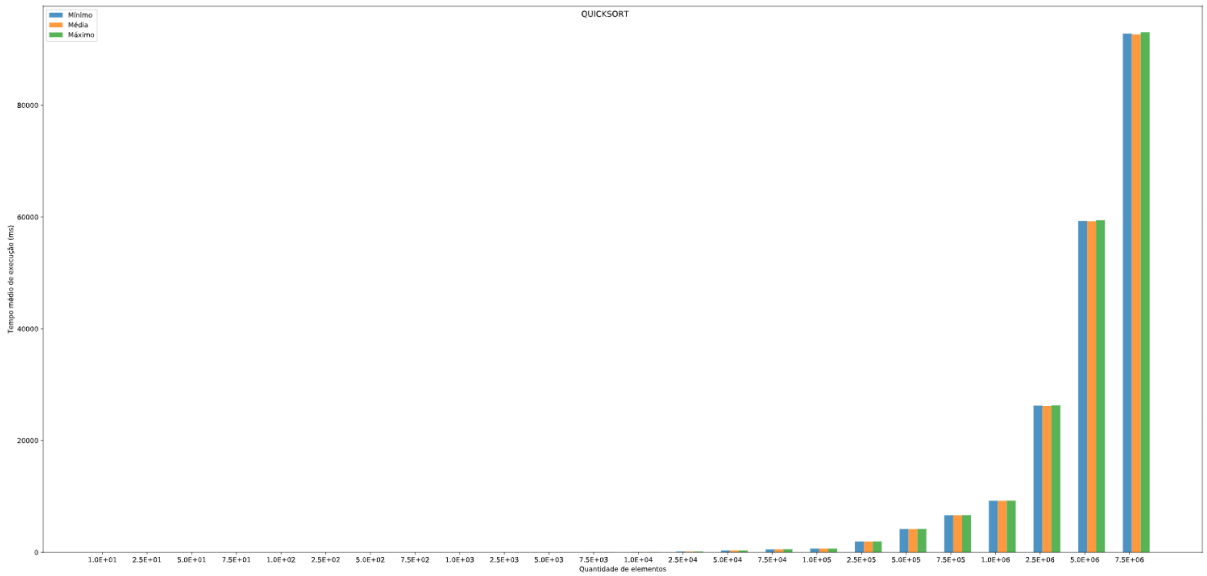

Figura 6: Gráfico do tempo de execução do Quicksort.

O *Quicksort* também possui a complexidade de $\Theta(nlog_2n)$, onde foi utilizado como base para medição de timeout, visto que possui um dos melhores desempenhos sobre os algoritmos de obrigatória implementação. Possui custo notável, pouco menos de 2 segundos, a partir de 250000 registros e com custo de tempo em torno de 1,5 minutos para o total de registros, no caso 7500000, como é mostrado na Figura 6.

4.2.5 Heapsort

Heapsort			
Total Records	**Average (ms)**	**Minimum (ms)**	**Maximum(ms)**
10	0.1755735	0.145922	0.227926
25	0.3187853	0.253909	0.463028
50	0.5954568	0.505377	0.707190999999999
75	0.8697918	0.599853	1.047813
100	1.1994068	0.896988	1.460343
250	2.5038601	1.476462	3.867148
500	6.4597693	3.37596	7.85595
750	9.5982234	5.864206	13.568797
1000	12.239323	7.196228	15.608808
2500	26.2358117	25.446467	27.431464
5000	50.0391093	45.276393	52.686928
7500	76.1907209	75.172519	78.205908
10000	101.9458537	96.780917	105.380381
25000	286.3624162	284.131473	293.115723
50000	642.4748843	639.431698	645.424742
75000	995.8367147	990.391292	1000.387384
100000	1392.6067732	1389.463902	1395.416374
250000	4098.1828184	4088.367742	4112.678678
500000	8686.0383563	8670.96148	8710.792671
750000	13792.3426109	13766.04422	13830.467842
1000000	19026.0666917	19005.03842	19089.855931
2500000	53582.9072272	53532.174368	53709.81947
5000000	116321.393643	116133.068901	116805.566337
7500000	182124.330677	181805.880857	182533.07936
TIMEOUT (ms)	**912242.2771**		

Tabela 5: Estatísticas dos tempos de execução do algoritmo Heapsort.

Figura 7: Gráfico do tempo de execução do Heapsort.

A complexidade do *Heapsort* $\Theta(nlog_2n)$, executando todos as entradas, em relação aos outros algoritmos com a mesma complexidade, porém com um desempenho menor em relação aos dois anteriores. Este já possui um desempenho mais considerável a partir de 100000 registros, com custo um pouco mais 1 segundo, e custo 2 vezes maior que os dois anteriores (quicksort e mergesort) para 7500000 demorando em torno 3 minutos, como é mostrado na Figura 7.

4.2.6 Introsort

Introsort			
Total Records	**Average (ms)**	**Minimum (ms)**	**Maximum (ms)**
10	0.1913663	0.128251	0.255191
25	0.3132138	0.21104	0.453676
50	0.5783367	0.511087	0.654748
75	0.8004745	0.504561	1.00833
100	1.4468503	1.184897	1.67571
250	2.616117	1.991324	3.861411
500	5.4870136	3.810233	8.612736
750	9.494108	8.011285	11.391001
1000	11.977483	7.12971	16.722446
2500	27.0864828	23.500798	28.995656
5000	48.3541936	44.401008	51.172284
7500	74.9151763	70.195207	77.937171
10000	102.045273	96.609641	106.67345
25000	286.6707056	283.876831	291.938767
50000	642.351485	636.09602	647.24104
75000	997.1283355	994.653873	1000.057226
100000	1391.1397919	1384.663723	1399.502553
250000	4090.673953	4082.646179	4097.385903
500000	8673.748339	8660.165822	8688.858778
750000	13788.942246	13778.112922	13803.259985
1000000	19020.0503025	18969.66871	19211.619342
2500000	53421.5436551	53355.219444	53485.16455
5000000	116067.3083249	115887.290706	116424.960583
7500000	181901.4706732	181799.964798	182017.004876
TIMEOUT (ms)	**912242.2771**		

Tabela 6: Estatísticas dos tempos de execução do algoritmo Introsort.

Figura 8: Gráfico do tempo de execução do Introsort.

Introsort um algoritmo híbrido de complexidade $\Theta(nlog_2n)$, que em comparação aos demais teve o pior desempenho em entradas de dados bem pequenas, porém desempenho melhor para maior quantidade de registros, mas assemelhando-se ao *Heapsort*, possuindo custo também um pouco maior que 3 minutos para 7500000 registros, como é mostrado na Figura 8.

4.2.7 Timsort

Timsort			
Total Records	**Average (ms)**	**Minimum (ms)**	**Maximum (ms)**
10	0.1169616	0.082105	0.152678
25	0.2787946	0.236035	0.371082
50	0.4574584	0.373136	0.63281
75	0.6006387	0.401664	0.936791
100	0.959196	0.510155	1.596697
250	1.7633065	1.239567	2.461032
500	3.1132286	2.245794	3.66648
750	5.6125922	3.910906	6.854602
1000	6.6699417	4.336757	8.406006
2500	17.2783206	15.324658	18.815558
5000	28.6931781	27.005417	30.229143
7500	41.6140855	35.699249	43.530175
10000	56.1798351	55.485545	57.302899
25000	147.0193484	143.855217	149.003746
50000	304.9228902	300.288381	306.624137
75000	481.4233439	474.852455	483.100618
100000	660.0840526	657.874611	665.446689
250000	1738.3637821	1734.930637	1742.388434
500000	3763.9158751	3758.919784	3771.725333
750000	5909.8770187	5895.687127	5921.384677
1000000	7938.7660803	7929.640789	7955.674468
2500000	21634.3857313	21579.516191	21724.706607
5000000	46197.4054755	45964.787265	47429.030726
7500000	69415.6667284	69370.972999	69487.03633
TIMEOUT (ms)	**912242.2771**		

Tabela 7: Estatísticas dos tempos de execução do algoritmo Timsort.

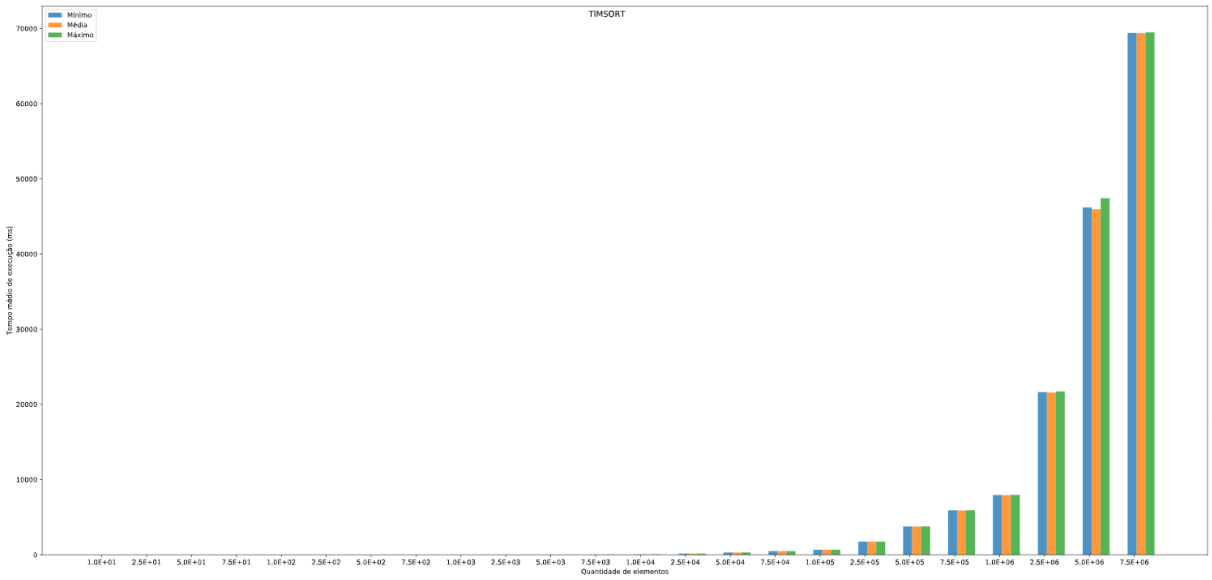

Figura 9: Gráfico do tempo de execução do Timsort.

O *Timsort* também tem a complexidade de $\Theta(nlog_2n)$ e assim como o *Introsort* é híbrido. Teve o melhor desempenho entre todos os algoritmos com entradas de dados grandes, superiores a 50000 linhas. Possuindo custo em torno de 1 minuto e 10 segundos para 7500000, como é mostrado na Figura 9.

4.2.8 Análise conjunta de todos os algoritmos de ordenação

Average of time execution of all algorithms							
Total Records	**Selection Sort**	**Insertion Sort**	**Merge Sort**	**Quicksort**	**Heapsort**	**Introsort**	**Timsort**
10	0.0828504	0.1120711	0.1683978	0.1131264	0.1755735	0.1913663	0.1169616
25	0.1933319	0.2032633	0.2838896	0.1672624	0.3187853	0.3132138	0.2787946
50	0.5755221	0.4413823	0.4861434	0.3126027	0.5954568	0.5783367	0.4574584
75	1.0387112	0.8520973	0.8254709	0.5062997	0.8697918	0.8004745	0.6006387
100	2.5734205	1.472551	1.1987972	0.8331159	1.1994068	1.4468503	0.959196
250	11.6573716	6.7688062	2.5644491	1.3462559	2.5038601	2.616117	1.7633065
500	31.9207071	20.0637704	5.5758295	2.6332269	6.4597693	5.4870136	3.1132286
750	62.8736719	37.060291	7.8216197	4.7538351	9.5982234	9.494108	5.6125922
1000	108.535861	65.166984	10.3667951	5.2536694	12.239323	11.977483	6.6699417
2500	658.0068997	379.323529	20.6688922	14.3068495	26.2358117	27.0864828	17.2783206
5000	2655.4927214	1567.0939451	38.687849	27.3967887	50.0391093	48.3541936	28.6931781
7500	5928.7287433	3423.6631442	55.9333884	38.4377591	76.1907209	74.9151763	41.6140855
10000	10715.3564662	6182.5665528	75.2054318	52.2457401	101.9458537	102.045273	56.1798351
25000	74630.3859784	41031.8505264	198.7678439	135.4775507	286.3624162	286.6707056	147.0193484
50000	336948.028985	194103.8127202	426.2855839	323.8354155	642.4748843	642.351485	304.9228902
75000	834708.8484096	533279.2042283	675.0263716	524.6261798	995.8367147	997.1283355	481.4233439
100000	912242.2771	912242.2771	928.5264272	666.4791089	1392.6067732	1391.1397919	660.0840526
250000	912242.2771	912242.2771	2528.4812708	1926.386345	4098.1828184	4090.673953	1738.3637821
500000	912242.2771	912242.2771	5490.5929088	4169.9041362	8686.0383563	8673.748339	3763.9158751
750000	912242.2771	912242.2771	8699.2297855	6622.5045697	13792.3426109	13788.942246	5909.8770187
1000000	912242.2771	912242.2771	11764.5924389	9224.7008268	19026.0666917	19020.0503025	7938.7660803
2500000	912242.2771	912242.2771	32729.8362959	26252.8115843	53582.9072272	53421.5436551	21634.3857313
5000000	912242.2771	912242.2771	69787.5443332	59293.2579544	116321.393643	116067.3083249	46197.4054755
7500000	912242.2771	912242.2771	108298.0473089	92824.1835982	182124.330677	181901.4706732	69415.6667284
TIMEOUT	912242.2771						

Figura 10: Tempo médio de execução dos algoritmos (ms).

Para a menor entrada de dados o algoritmo que possuiu o melhor desempenho foi o *Selection Sort*. O algoritmo com melhor desempenho médio foi o *Quicksort* que ordenou mais rápido as entradas entre 25 a 25000 linhas de dados. Para entradas a partir de 50000 linhas o que teve melhor desempenho foi o *Timsort*.

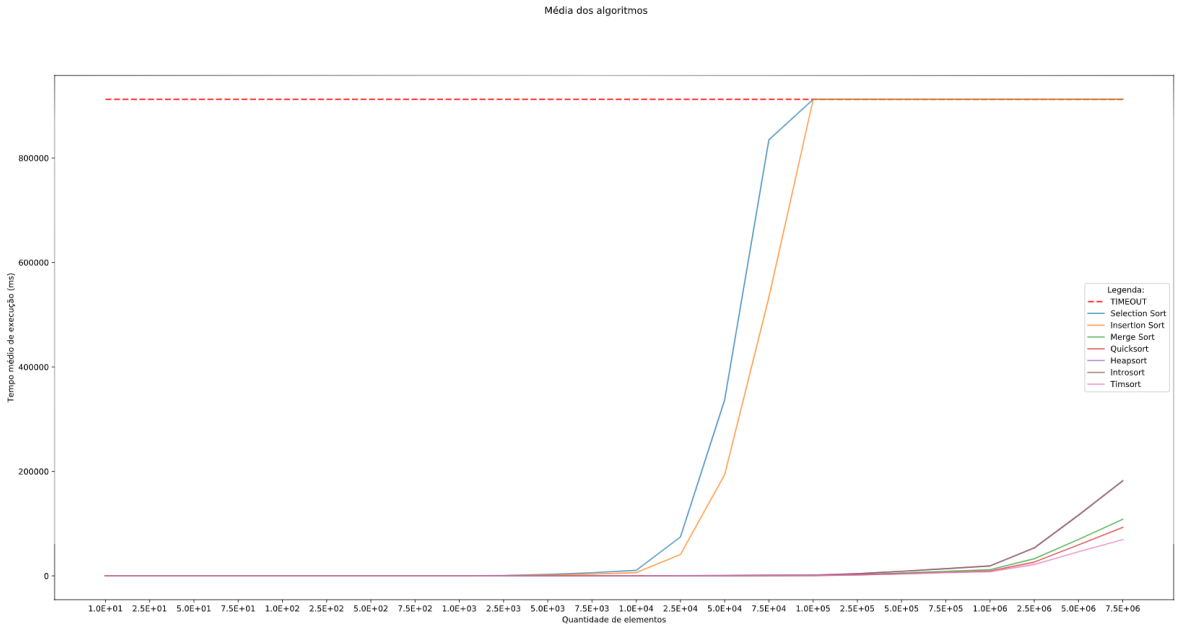

Figura 11: Gráfico do tempo médio de execução dos algoritmos em linha.

A Figura 11 possui uma visualização melhor em relação ao comportamento desses algoritmos em relação a sua complexidade. Note que pela curva do gráfico apresentado na Figura 11 é nítido que tanto o *Selection Sort* quanto o *Insertion Sort* são funções quadráticas e os demais algoritmos de ordenação são logarítmicas.

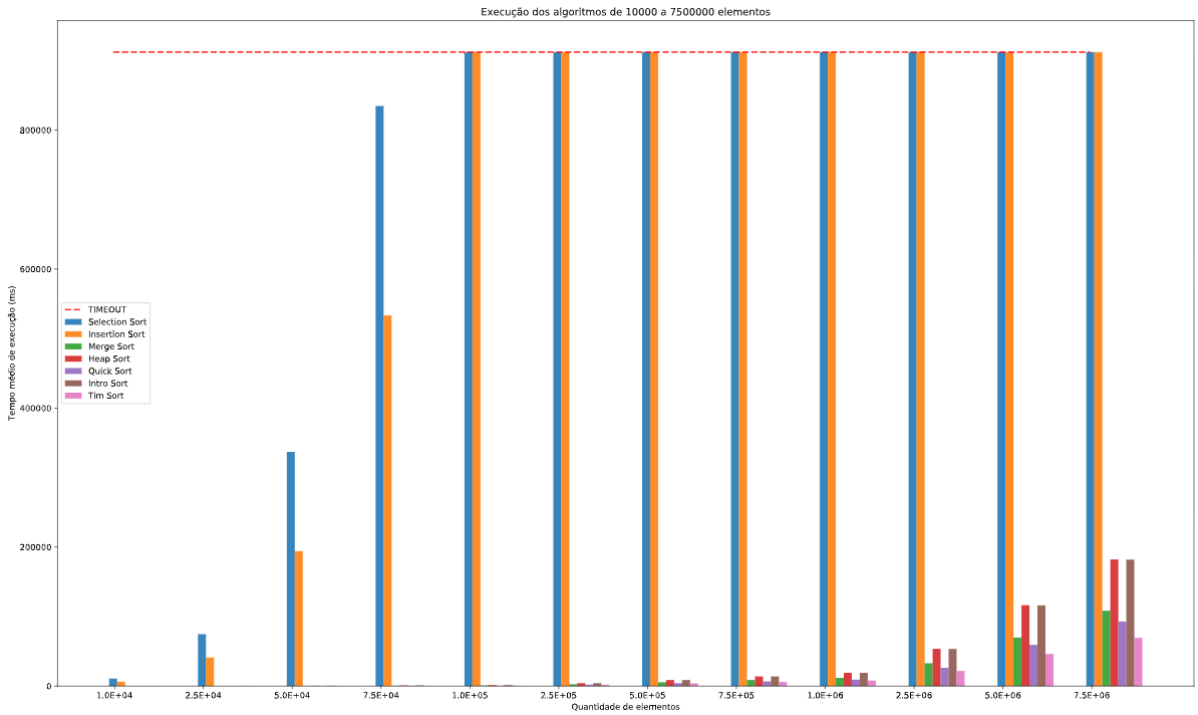

Figura 12: Gráfico do tempo médio em entradas de 10000 a 7500000 dados.

No intuito de uma melhor visualização das análises a Figura 12 mostra o tempo dos algoritmos com as entradas mais significantes, a partir de 10000 dados, onde o tempo é superior a 50 milissegundos. Percebe-se que os algoritmos *Selection Sort* e o *Isertion Sort* acabam mascarando o desempenho dos outros por isso foi separado em mais dois gráficos, Figura 13 e Figura 14.

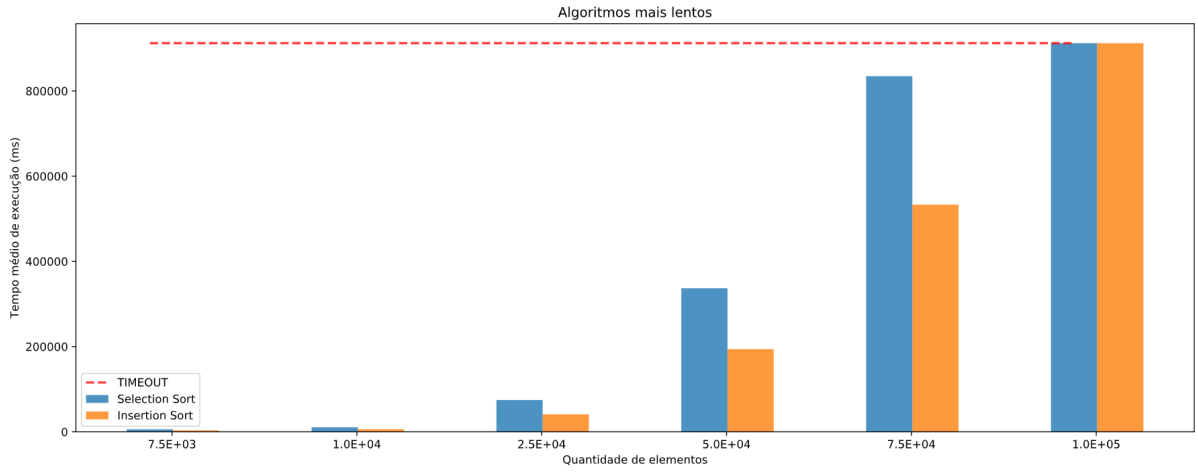

Figura 13: Gráfico do tempo médio de execução dos algoritmo mais lentos.

O *Selection Sort* e o *Isertion Sort* não são os mais eficientes, não executando oito entradas, tendo um desempenho bom na menor entrada, e um desempenho próximo aos outros até 75 registros.

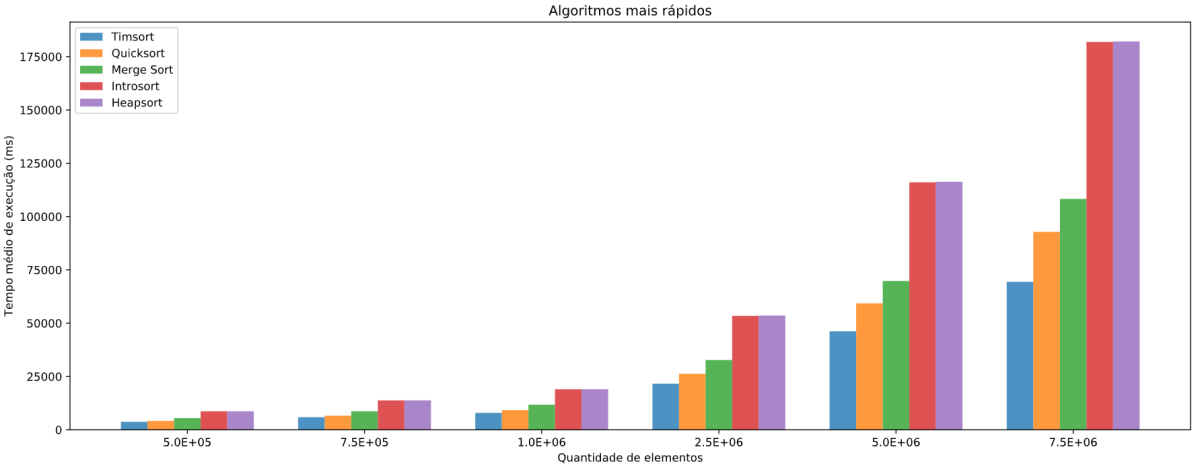

Figura 14: Gráfico do tempo médio de execução dos algoritmo mais rápidos.

Na Figura 14 é mostrado de forma mais clara o desempenho do *Timsort*, *Quicksort*, *Merge sort*, *Introsort* e o *Heapsort* para entradas maiores sem a interferência dos algoritmos com pior desempenho, onde o *Timsort* é em média o melhor algoritmo para as entradas maiores, e notável que tanto o *Heapsort* quanto o *Introsort* são os piores para maiores entradas de registros.

5 Referências Bibliográficas

- Leitura de arquivos csv:
  - https://pythonprogramming.net/reading-csv-files-python-3/
  - Códigos anteriores (programação 1 e 2 e estrutura de dados).
- Escrita de arquivos csv:
  - https://www.programiz.com/python-programming/working-csv-files
  - Códigos anteriores (programação 1 e 2 e estrutura de dados).
- Selection Sort:
  - https://pt.wikipedia.org/wiki/Insertion_sort
  - Códigos anteriores (programação 2 e estrutura de dados).
- Insertion Sort:
  - https://pt.wikipedia.org/wiki/Selection_sort
  - Códigos anteriores (programação 2 e estrutura de dados).
- MergeSort:
  - https://pt.wikipedia.org/wiki/Merge_sort
  - Códigos anteriores (programação 2 e estrutura de dados).
- QuickSort:
  - https://pt.wikipedia.org/wiki/Quicksort
  - https://www.cc.gatech.edu/classes/cs3158_98_fall/quicksort.html
  - Códigos anteriores em estrutura de dados.
- Heapsort:
  - https://pt.wikipedia.org/wiki/Heapsort
  - https://www.cc.gatech.edu/classes/cs3158_98_fall/heapsort.html
  - http://wiki.icmc.usp.br/images/c/c6/Ordenacao2.pdf
  - Códigos anteriores em estrutura de dados.
- Introsort:
  - https://en.wikipedia.org/wiki/Introsort
  - https://www.geeksforgeeks.org/introsort-or-introspective-sort
  - https://pt.wikipedia.org/wiki/Intro_sort
- Timsort:
  - Link: Timsort-the fastest sorting algorithm you've never heard of
  - https://pt.wikipedia.org/wiki/Timsort
- TimeOut do algoritmo de ordenação:

– Ajuda do colega de sala, Joel, de como parar o algoritmo após a execução do tempo limite. Entretanto não foi utilizada a mesma biblioteca que ele, dado que ele utilizou a metodologia de criação de threads, já no nosso projeto foi utilizado a metodologia de criação um processo novo, disponibilizado lib chamada *multiprocessing* do próprio *Python*.