Relatório do Trabalho de Ordenação e Estatísticas de Ordem

Técnicas de Programação Avançada — IFES — Campus Serra

Alunos: Antônio Carlos Durães da Silva,
Carlos Guilherme Felismino Pedroni,
Lucas Gomes

Prof. Jefferson O. Andrade

29 de outubro de 2019

Sumário

Lista de Códigos Fonte

Lista de Figuras

1 Introdução

Este documento refere-se aos testes de desempenho dos algoritmos requeridos para o Trabalho de Ordenação e Estatísticas de Ordem da disciplina de Técnicas de Programação Avançada.

1.1 Ambiente de desenvolvimento

Para o desenvolvimento do trabalho foi utilizada a linguagem Python e a biblioteca MatPlotLib para realizar o plot dos gráficos. Para edição do código-fonte foi utilizado o ambiente PyCharm Community e Jupyter Notebook. Para a escrita do relatório foi utilizado o Overleaf e TeXstudio.

Para execução dos testes foi utilizado uma máquina com as seguintes configurações:

- CPU: i7-8550U
- SO: Windows 10 (x64)
- LP: Python (v. 3.6.8)

2 Dados para Testes

Para que os algoritmos fossem testados utilizou-se os 24 arquivos de entrada disponibilizados pelo professor. Esses arquivos de dados possuem números crescentes de registros: 10, 25, 50, 75, 100, 250, 500, 750, 1k, 2.5k, 5k, 7.5k, 10k, 25k, 50k, 75k, 100k, 250k, 500k, 750k, 1M, 2.5M, 5M e 7.5M.

2.1 Estrutura do Arquivo de Dados

Os arquivos de dados estarão no formato CSV e cada linha será composta pelos seguintes campos:

1. Email (email) — **cadeia de caracteres**
2. Sexo (gender) — **caractere [“F”, “M”, “O”]**
3. Identificador da pessoa (uid) — **alfa-numérico; único**
4. Data de nascimento (birthdate) — **data - formato universal**[]
5. Altura [centímetros] (height) — **inteiro**
6. Peso [kilogramas] (weight) — **inteiro**

3 Implementação do Trabalho

O projeto se encontra na pasta SRC e está dividido em 2 pastas util e sort. sendo esse os principais arquivos para o trabalho:

- main.py – Contém o código para o programa principal (Main) e se encontra na raiz da pasta “src”. É responsável por conter a leitura, validação e tratamento de argumentos oriundos da linha de comando. Coordena qual algoritmo será chamado de acordo com os argumentos recebidos.
- pessoa.py – Contém a classe pessoa com seus atributos e seus métodos utilizados para comparação, foi feito um método de comparação para cada atributo do objeto. Este arquivo encontra-se na raiz da pasta “src”.

- metodos_ordenacao.py – Contém a implementação dos algoritmos de ordenação. Encontra dentro da pasta "sort".
- verificador_ordenacao.py – Contém a implementação de métodos responsáveis por verificar se uma determinada lista está ordenada, crescente ou decrescentemente. Encontra-se dentro da pasta "sort".
- comparador.py - Classe que armazena uma função de comparação generica, tal função compara dois objetos de mesmo tipo, de acordo com um atributo ou um conjunto deles. Encontra-se na pasta "util".
- csv_manipulador.py - Contém a implementação de métodos responsáveis pela manipulação e criação dos arquivos de tipo **csv**. Converte um objeto Pessoa em um arquivo CSV e vice-versa. Encontra-se dentro da pasta "util".
- graph_generator.py - Contém as funções responsáveis por manipular a criação e persistência de gráficos. Encontra-se dentro da pasta "util".

3.1 Programa Principal

A função principal é responsável por coordenar as chamadas às funções que recebem e validam os argumentos de entrada por linha de comando, que abre o arquivo de entrada, realiza sua leitura e conversão das linhas em objetos do tipo Pessoa; que inicia e finaliza a contagem de tempo, e por identificar se apenas um e qual algoritmo de ordenção será chamado ou se todos serão executados.

Ao final de cada execução com êxito, a aplicação principal imprime o algoritmo utilizado, a quantidade de dados ordenados, a duração da ordenação e chama a função responsável por conveter a lista ordenada em um novo CSV para o caminho de saída recebido via argumento.

3.2 Modelagem de Dados

Embora, por uma questão de produtividade e praticidade o grupo tenha escolhido a linguagem Python, de tipagem fraca e multiparadigma, a modelagem teve como foco principal o paradigma Orientado a Objetos, a fim de ter um código mais modularizado, de fácil manutenção e compreensão por todos integrantes da equipe. Dessa forma, foram criadas classes para os três componentes principais do trabalho: **Pessoa** (para representar os dados de entrada), **Comparador** e **Método de Ordenação**.

3.2.1 Comparador

A fim de criar um modelo genérico para comparar dois objetos por uma ou mais propriedades, criou-se uma classe abstrata, que quando extendida pode representar um comparador para qualquer tipo de dado. Esta classe contém apenas um método, o "compararCom", que recebe dois objetos de mesmo tipo e retorna:

- **-1**: Se o primeiro item possuir a chave anterior a do segundo
- **0**: Se o primeiro item possuir a chave igual a do segundo
- **1**: Se o primeiro item possuir a chave posterior a do segundo

3.3 Algoritmos de Ordenação

Para este trabalho de Ordenação e Estatísticas de Ordem foi pedido a implementação de sete algoritmos de ordenação diferentes. A saber: ordenação por seleção (*selection sort*), ordenação por inserção (*insertion sort*), *merge sort*, *quicksort*, *heapsort*, *timsort* e *introsort*. Os últimos 2 sendo escolhidos pelo grupo do trabalho.

3.3.1 Algoritmo SelectionSort

Um dos métodos de ordenação de implementação mais simples e fácil compreensão, a ordenação por seleção trabalha com a ideia de obter-se os menores elementos e inseri-los nas primeiras posições da lista de elementos.

De forma mais precisa, quanto menor o valor do elemento, mais à esquerda da lista ele ficará. Dessa maneira, busque o menor elemento e troque-o com o primeiro (início da lista à esquerda) da lista; busque o segundo menor item e troque-o com o elemento da segunda posição da lista; busque o i-ésimo menor elemento e troque-o com o i-ésimo elemento da lista [1].

Por percorrer N elementos ou N-1 elementos a cada iteração, o algoritmo é considerado de ordem quadrática.

Código Fonte .1 Implementação do algoritmo Selection Sort

```
def __selection_sort(
        comparador: Comparador[T], lista: List[T],
        inic: Optional[int] = None, fim: Optional[int] = None) -> List[T]:

   def min_indice(lista: List[T], ind_inic: int, ind_fim: int) -> int:

      indice_min = ind_inic

      # Busque o índice do menor elemento do índice de início ao de fim
      for i in range(ind_inic + 1, ind_fim + 1):

         if comparador.compararCom(lista[indice_min], lista[i]) > 0:
            indice_min = i

      return indice_min

   if inic is None:
      inic = 0

   if fim is None:
      fim = len(lista)

   else:
      fim += 1

   # Para cada índice do ínicio até o fim da lista
   for i in range(inic, fim):
      ind_min = min_indice(lista, i, fim - 1)

      # Troque a posição do item atual com a posição do menor item
      lista[i], lista[ind_min] = lista[ind_min], lista[i]

   return lista
```

3.3.2 Algoritmo InsertionSort

De complexicidade semelhante ao algoritmo anterior, a ordenação por inserção também possui como características uma rápida implementação e mecanismo simples de compreensão.

Seu funcionamento se resume em selecionar um **elemento A** na lista e ir trocando de posição com elementos de índices anteriores, enquanto A possuir valor inferior ao do i-ésimo item anterior ou até chegar à posição mais a esquerda da lista (início).

Cormem[2] (2002, p. 18-20.), aponta em sua obra que este algoritmo, em seu pior caso, tem complexidade quadrática, e que seu caso médio também pode atingir tal grandeza de operações.

A implementação abaixo teve como ponto de partida e base, o pseudo-código descrito na obra de Cormem.

Código Fonte .2 Implementação do algoritmo Insertion Sort

```
1  def __insertion_sort(
2        lista: List[T], comparador: Comparador[T],
3        inic: Optional[int] = 0, fim: Optional[int] = None) -> List[T]:
4
5     if fim is None:
6        fim = len(lista)
7
8     else:
9        fim += 1
10
11    # Para cada índice da posição inicial até o (tamanho - 1)
12    for i in range(inic, fim - 1):
13
14       item_atual = lista[i + 1]
15       j = i
16
17       while j >= inic and comparador.compararCom(lista[j], item_atual) > 0:
18          # Troque a posição do item atual com a posição do item anterior
19          lista[j + 1] = lista[j]
20          j -= 1
21
22       lista[j + 1] = item_atual
23
24    return lista
```

3.3.3 Algoritmo MergeSort

O merge sort é um dos algoritmos que usufrui do método "dividir para conquistar", abordagem que é caracterizada por aproveitar-se da divisibilidade do problema. Algoritmos desse grupo dividem a massa de dados em conjuntos menores, resolvem os sub-problemas e posteriormente agrupam suas soluções para compor a solução do problema único e inicial (Cormem[2], 2002, p. 21).

É comum encontrar esse método de ordenação sendo chamado de ordenação por mesclagem, mistura ou intercalação, esse último referente à forma como o algoritmo organiza e mescla as soluções dos sub-problemas.

A função "merge sort" abaixo divide a lista de dados ao meio, depois divide cada sublista ao meio e passa cada uma como parâmetro para chamada recursiva a si (linhas 4 a 8). Essa divisão continua ao meio é realizada até que cada sublista tenha tamanho um, ou seja, atingiu sua posição final (linha 2), nesse ponto (linha 9), a função "merge" é chamada para intercalar as sublistas.

O grupo optou com externalizar a função merge, pois notou que ela seria usada por outros algoritmos.

A função merge recebe a lista com todos valores (nem todos serão processados obrigatoriamente), um objeto comparador, o índice de início, da metade e do fim da lista a ser intercalada.

Nas linhas 14 e 15 é definida a quantidade de elementos que a sublista à esquerda e à direita terão. Nas linhas 19 e 20, embora pareça que as sublistas estejam criadas como

uma cópia dos elementos da lista original, tais subconjuntos são apenas uma lista de ponteiros para os itens da lista principal, uma vez que cada elemento é um objeto e não tipo primitivo. Por meio de conhecimento prévio da linguagem e por meio de depuração, o grupo confirmou que a cópia de elementos é uma cópia superficial.

A partir da linha 24, enquanto houver elementos na sublista à esquerda ou na à direita, compara-se o elemento atual (de índice **i**) do subconjunto à esquerda com o elemento atual (de índice **j**) do da direita, verifica qual dos dois elementos é o menor e o insere na posição atual (índice **i esq**) da lista principal.

Se a sublista à direita esvaziar-se, todos elementos restantes da sublista à esquerda são inseridos na lista principal (linhas 38 a 40).

Senão, a sublista à esquerda está vazia e a da direita ainda pode conter elementos não inseridos na lista principal, então todos itens restantes são inseridos (linhas 43 a 45).

Devido o mecanismo de divisão contínua e recursiva ao meio do conjunto, o algoritmo passa a incluir em seu número de operações um fator logarítmico, Cormem(p. 27 - 28) define sua complexidade como $\mathbf{\Theta(nlog_2 n)}$.

Código Fonte .3 Implementação do algoritmo Merge Sort

```
1 def __merge_sort(lista_original: List[T], cmp: Comparador[T], l: int, r: int):
2     if l < r:
3         # separando a lista no meio
4         m = (l + (r - 1)) // 2
5 
6         # separando a lista
7         MergeSort.__merge_sort(lista_original, cmp, l, m)
8         MergeSort.__merge_sort(lista_original, cmp, m + 1, r)
9         merge(lista_original, cmp, l, m, r)
```

Código Fonte .4 Implementação da função Merge

```
1  def merge(
2          lista: List[T], cmp: Comparador[T], i_esq: int, i_meio: int,
3          i_dir: int):
4     n1 = i_meio - i_esq + 1
5     n2 = i_dir - i_meio
6
7     # Criação de lista somente com a referência de cada item, não há duplicatas
8     # de usuários!
9     sublista_esq = [lista[i_esq + i] for i in range(n1)]
10    sublista_dir = [lista[i_meio + i + 1] for i in range(n2)]
11
12    i = j = 0
13
14    while i < n1 and j < n2:
15
16       if cmp.compararCom(sublista_esq[i], sublista_dir[j]) < 1:
17          lista[i_esq] = sublista_esq[i]
18          i += 1
19
20       # Se não, insira o item da sublista direita e aponte para o próx item
21       else:
22          lista[i_esq] = sublista_dir[j]
23          j += 1
24
25       i_esq += 1
26
27    # copiando os elementos restantes da lista_esquerda[], caso exista
28    for it in range(i, n1):
29       lista[i_esq] = sublista_esq[it]
30       i_esq += 1
31
32    # copiando os elementos restantes da lista_direira[], caso exista
33    for it in range(j, n2):
34       lista[i_esq] = sublista_dir[it]
35       i_esq += 1
```

3.3.4 Algoritmo HeapSort

Segundo Knuth (p. 144), o algoritmo de ordenação por monte (heap) foi descoberto pelo cientista da computação John William Joseph Williams, em meados da década de 60.

Este algoritmo tem como sua principal estrutura um monte. Por se tratar de uma estrutura binária, o heap costuma ser representado por uma árvore binária.

O grupo decidiu simular a estrutura de uma árvore binária com a própria lista de dados. As linhas 6 e 7 são responsáveis por criar a árvore ordenada. A variável **i** inicia com o índice que divide a lista ao meio (quando ímpar), devido a construção da árvore utilizar o índice 2×i + 1 (linha 4 do heapify) para definir quem será o nó filho da esquerda ou direita da raiz.

Após criar a árvore ordenada, tem-se que a raiz da árvore (último elemento da lista) armazena o elemento de maior valor, dessa forma, a função heapsort percorre toda a lista, do fim para o início, substitui o primeiro elemento da lista com o elemento de índice **i** e reconstroi a árvore sem o elemento raiz. Sendo assim, na primeira iteração, o maior elemento estará na primeira posição, na segunda iteração o segundo maior elemento estará na primeira posição e o maior na última, esse processo segue até que toda árvore esteja ordenada.

Código Fonte .5 Implementação da função Heapify

```
def __heapify(
        lista: List[T], i: int, cmp: Comparador[T], tam: int):

    esq = 2 * i + 1
    dir = esq + 1
    maior = -1

    if esq < tam and cmp.compararCom(lista[esq], lista[i]) > 0:
        maior = esq

    else:
        maior = i

    if (dir < tam and cmp.compararCom(lista[dir], lista[maior]) > 0):
        maior = dir

    if maior != i:
        lista[i], lista[maior] = lista[maior], lista[i]
        HeapSort.__heapify(lista, maior, cmp, tam)
```

Código Fonte .6 Implementação do algoritmo Heap Sort

```
def __heapsort(lista: List[T], comparador: Comparador[T]) -> List[T]:

    # Método baseado no capítulo 6.3 do livro 'Algoritmos' (T. H. Cormen)
    tam = len(lista)

    for i in range(tam // 2, -1, -1):
        HeapSort.__heapify(lista, i, comparador, tam)

    for i in range(tam - 1, 0, -1):
        lista[i], lista[0] = lista[0], lista[i]
        HeapSort.__heapify(lista, 0, comparador, i)

    return lista
```

3.3.5 Algoritmo QuickSort

Descoberto por Charles Antony Richard Hoare, em 1961 [3], o algoritmo Quick Sort assemelha-se ao Merge e ao Heap Sort em alguns aspectos, como ser de natureza divisão e conquista, possui trechos recursivos (nesta implementação, também há versões iterativas, menos comuns) e complexidade.

O quick tem como essência um método auxiliar chamado de partição ou particionador. A função "partição" recebe a lista, um comparador, um índice de início e um de fim (determinam o espaço de verificação e ordenação).

A ideia principal do algoritmo é eleger um elemento da lista como pivô e organizar todos itens menores que ele à sua esquerda e todos valores superiores à sua direita na lista.

Segundo Cormem (2002, p. 122), o método faz jus ao título de ordenação rápida, pois encontra-se comumente mais próximo do melhor caso do que do pior, performando com a complexidade de $\Theta(nlog_2n)$, no melhor e médio caso; e $\Theta(n^2)$, no pior cenário.

Código Fonte .7 Implementação do algoritmo Quick Sort

```
1  def __quicksort(
2        lista: List[T], cmp: Comparador[T], l: int, r: int) -> List[T]:
3     if l < r:
4        q = particao(lista, cmp, l, r)
5        QuickSort.__quicksort(lista, cmp, l, q - 1)
6        QuickSort.__quicksort(lista, cmp, q + 1, r)
7
8     return lista
```

Código Fonte .8 Implementação da função Partição

```
1  def particao(
2           lista: List[T], cmp: Comparador[T], i_esq: int, i_dir: int) -> int:
3     pivo = lista[i_dir]
4     i = i_esq - 1
5
6     for ind in range(i_esq, i_dir):
7
8        # Se o item atual for menor/igual que o pivô, troque-o c/ o pivô
9        if cmp.compararCom(lista[ind], pivo) <= 0:
10          i += 1
11          lista[i], lista[ind] = lista[ind], lista[i]
12
13    lista[i + 1], lista[i_dir] = lista[i_dir], lista[i + 1]
14
15    return i + 1
```

3.3.6 Algoritmo IntroSort

O intro ou introspective sort é um algoritmo híbrido que mescla os métodos base quick sort e heap sort[4]. É possível encontrar implementações que englobam um tericeiro

algoritmo, geralmente adicionado para tratar conjunto de dados pequenos (menos que cem elementos).

Neste caso, optou-se por incluir o insertion sort como algoritmo para conjunto de até 32 elementos.

A função Intro Sort recebe a lista a ser ordenada, um objeto comparador, um índice de início e de fim, e o número máximo de recursão desejado.

Por englobar o método Insertion, define-se um número máximo de elementos para aplicá-lo, tal número não deve ser muito grande, pois perde o melhor artifício da ordenação por inserção, nem pode ser muito pequeno, pois aproxima-se de uma ordenação de um único elemento, podendo gerar muitas chamadas adicionais (considerando que o método é recursivo). O grupo escolheu o tamanho máximo de 32 elementos.

A quantidade de níveis de recursão é definida por $\boldsymbol{lgn}$, ela é responsável por sinalizar quando o intro deve para de efetuar chamadas para si e começar a invocar o método heap sort (linhas 9 e 10).

Na linha 13, uma função auxiliar é chamada para verificar qual elemento entre o de índice inicial, índice central e índice final, é o item mediano entre os três. Este método é citado por Knuth (p. 122) como uma das formas para realizar uma troca de elementos a fim de reduzir o número de comparações.

Nas linhas (15 - 17), a função apenas chama o método de partição para obter um pivô, chama a si com a primeira metade da lista e depois com a segunda metade.

Código Fonte .9 Implementação da função Intro Sort

```
1  def __intro_sort(
2        lista: List[T], cmp: Comparador[T], i_ini: int, i_fim: int,
3        niveis_limite: float) -> List[T]:
4     tam = i_fim - i_ini
5
6     if tam <= IntroSort.__qtd_min:
7        return InsertionSort.ordenar(cmp, lista, i_ini, i_fim)
8
9     elif niveis_limite == 0:
10       return HeapSort.ordenar(cmp, lista)
11
12    else:
13       p = IntroSort.__mediana_3(lista, cmp, i_ini, i_ini + tam // 2, i_fim)
14       lista[p], lista[i_fim] = lista[i_fim], lista[p]
15       piv = particao(lista, cmp, i_ini, i_fim)
16       IntroSort.__intro_sort(lista, cmp, i_ini, piv - 1, niveis_limite - 1)
17       IntroSort.__intro_sort(lista, cmp, piv + 1, i_fim, niveis_limite - 1)
18
19    return lista
```

Código Fonte .10 Implementação da função de Mediana

```
1  def __mediana_3(
2        lista: List[T], cmp: Comparador[T], i_inic: int, i_meio: int,
3        i_fim: int) -> int:
4
5     item_1 = lista[i_inic]
6     item_2 = lista[i_meio]
7     item_3 = lista[i_fim]
8
9     cmp_i3_i1 = cmp.compararCom(item_3, item_1)
10    cmp_i1_i2 = cmp.compararCom(item_1, item_2)
11    cmp_i3_i2 = cmp.compararCom(item_3, item_2)
12
13    # se item_3 >= item_1 >= item_2 OU item_3 <= item_1 <= item_2:
14    if (cmp_i3_i1 > -1 and cmp_i1_i2 > -1) or (cmp_i3_i1 < 1 and cmp_i1_i2 < 1):
15       return i_inic
16
17    # se item_3 >= item_2 >= item_1 or item_3 <= item_2 <= item_1:
18    if (cmp_i3_i2 > -1 and cmp_i1_i2 < 1) or (cmp_i3_i2 < 1 and cmp_i1_i2 > -1):
19       return i_meio
20
21    # se item_1 >= item_3 >= item_2 or item_1 <= item_3 <= item_2:
22    if (cmp_i3_i1 < 1 and cmp_i3_i2 > -1) or (cmp_i3_i1 > -1 and cmp_i3_i2 < 1):
23       return i_fim
```

3.3.7 Algoritmo TimSort

Descoberto por Tim Peter, o algoritmo que leva seu nome é um método híbrido entre o heap sort e o insertion sort. Tim, desenvolveu o método, a princípio para Python, hoje o algoritmo já integra a linguagem Java, o sistema operacional Android e muitos outros softwares [5].

O algoritmo apresenta o conceito de run, que é um subconjunto de tamanho pré-estabelecido. Uma lista de entrada pode ser composta por um ou mais runs. O tamanho de cada run varia, mas por usar a ordenação por inserção, tamanhos entre 16 e 128 elementos são recomendados.

Tim partiu da premissa que dentro de um conjunto de dados, há pequenos subconjuntos já ordenados. Após realizar a divisão dessas sublistas, basta ordenar cada uma (linhas 7 a 9) e ordená-las por intercalação ou mesclá-las (merge) para unificar seus elementos na lista final (linhas 19 a 23).

A complexidade do método de Peter é $\boldsymbol{\Theta(nlgn)}$ para todos os casos, exceto se a lista estiver ordenada, neste caso, é de $\boldsymbol{\Theta(n)}$ [6].

Código Fonte .11 Implementação da função Tim Sort

```
def __tim_sort(lista: List[T], cmp: Comparador) -> List[T]:

    tam_lista = len(lista)
    ult_ind = tam_lista - 1

    # Ordene cada run de acordo com o tamanho pré-estabelecido
    for i in range(0, tam_lista, TimSort.__tam_run):
        run_fim = i + TimSort.__tam_run - 1
        InsertionSort.ordenar(cmp, lista, i, min(run_fim, ult_ind))

    itens_ordenados = TimSort.__tam_run

    # Enquanto houverem elementos a serem ordenados
    while itens_ordenados < tam_lista:

        # Percorra os índices da lista, de 0 até seu último índice.
        # A cada iteração, incremente i em 2 vezes (devido merge com duas
        # sublistas) a qtd de itens ordenados
        for ind_esq in range(0, tam_lista, 2 * itens_ordenados):
            # Calcule os índices de esquerda, meio e direita p/ o merge
            ind_meio = min((ind_esq - 1 + itens_ordenados), ult_ind)
            ind_dir = min((ind_esq - 1 + 2 * itens_ordenados), ult_ind)
            merge(lista, cmp, ind_esq, ind_meio, ind_dir)

        # O número de itens ordenados é o dobro da quantidade anterior,
        # já que duas sublistas (runs) foram mescladas
        itens_ordenados *= 2

    return lista
```

4 Análise de Desempenho de Algoritmos

Com exceção dos algoritmos Selection e Insertion Sort, todos outros foram executados para todos arquivos de entrada. Para cada arquivo de dados todos algoritmos testados foram executados dez vezes, como recomendado na especificação.

Todos os gráficos e tabelas utilizados encontram-se no arquivo compactado no diretório "resultados" ou no repositório do grupo[1].

Para evitar que o excesso de dados no relatório resulte em poluição do documento, o grupo tomou a liberdade de não exibir as tabelas utilizadas na formação dos gráficos, contudo, como dito acima, todas estão acessíveis pelo repositório.

4.1 Análise individual

O grupo notou algumas peculiaridades de cada algoritmo a medida que o número de dados crescia. Por isso, decidiu-se que além de um comparativo entre todos algoritmos, cada

[1] https://github.com/duraes-antonio/TPA_trab2/

método teria o gráfico com sua duração mínima, média e máxima para toda quantidade de dados. Todas análise abaixo tiveram como base a compreensão e leitura das tabelas com a média e o valor de cada execução de cada algoritmo.

4.1.1 Análise gráfica - Selection Sort

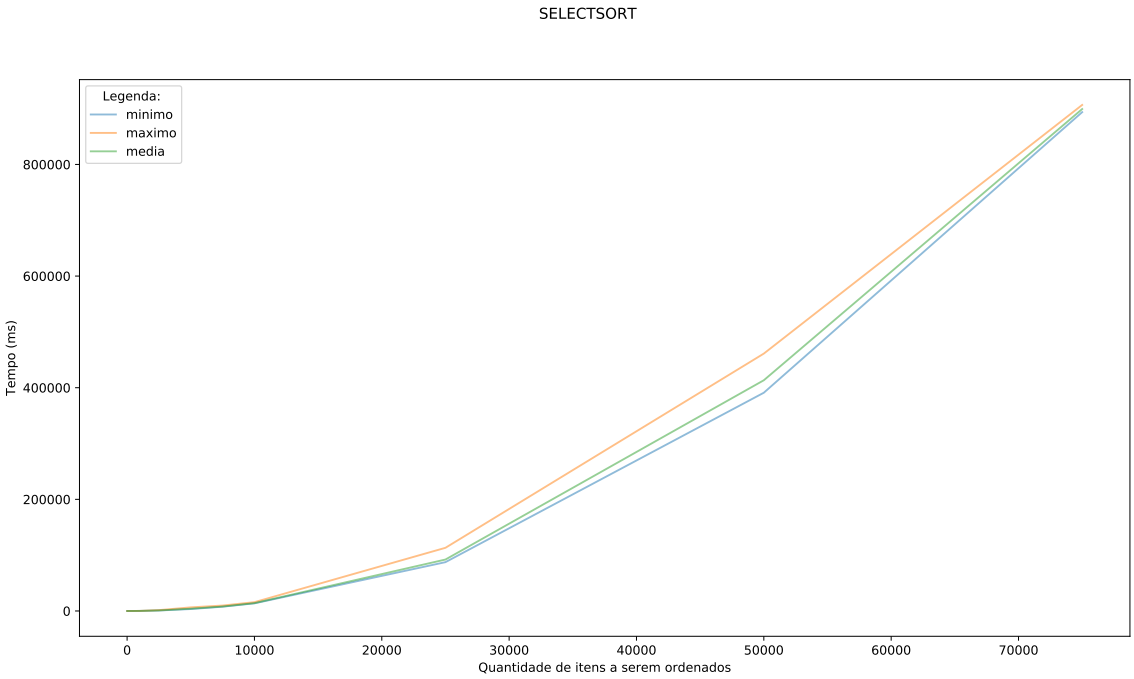

Figura 1: Gráfico de desempenho do algoritmo Selection Sort

O algoritmo começa a apresentar desvio considerável próximo a 25 mil registros. Neste caso, uma execução durou 113181 milessegundos, o que resultou em um desvio aproximado 22.7906% relação à média (92174). Todas outras execuções para esse conjunto de dados duraram entre 87 e 92 mil milessegundos.

Para 50 mil registros, o desvio se mantém, enquanto a média ficou em 413 mil ms, duas execuções ultrapassaram 450 mil ms, a máxima foi de 461 mil ms, apresentando desvio aproximado a 11.5556%.

Com 75 mil registros, o algoritmo apresenta baixa variação, inferior a 1%.

É importante notar que por qualquer uma das curvas (mínima, média ou máxima) o gráfico guarda semelhanças com a representação da curva de uma função quadrática. O aumento de inclinação da linha é pouco perceptível entre os arquivos com 10 20 mil registros, diferente

4.1.2 Análise gráfica - Insertion Sort

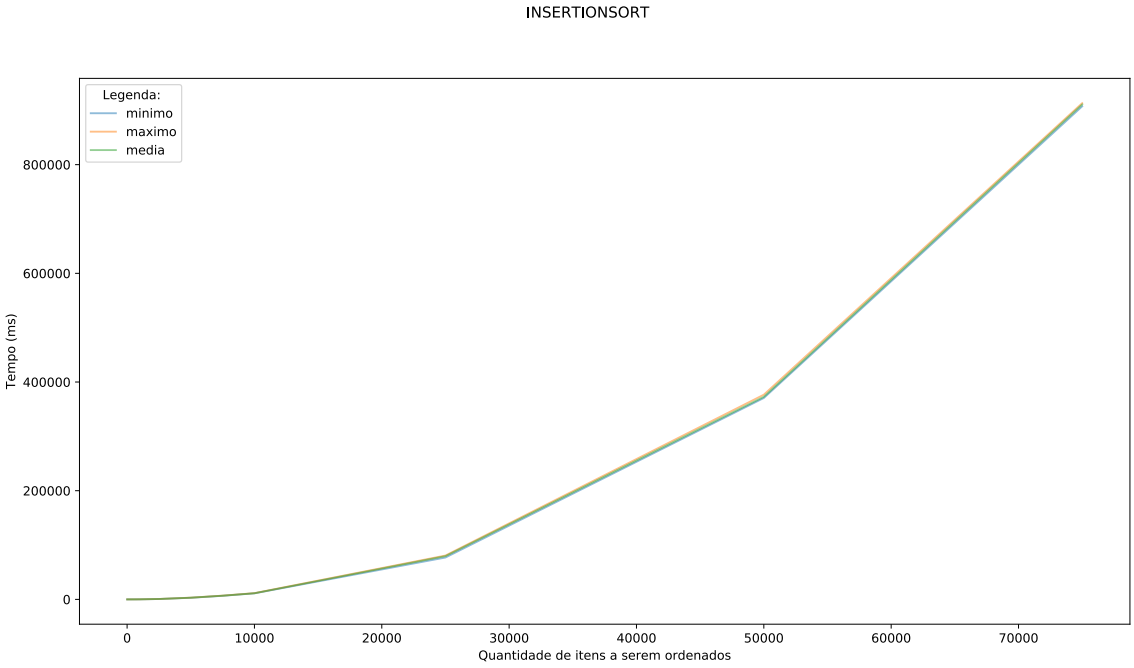

Figura 2: Gráfico de desempenho do algoritmo Insertion Sort

De todos os algoritmos, este foi o que apresentou menor variação entre as execuções, com suas durações mínima e máxima bem próximas à média.

Um ponto de destaque é o formato de sua linha, dando à impressão, que se houvesse uma continuação da bateria de testes, sua curva tenderia ao formato de uma parabola, representando graficamente sua complexidade.

4.1.3 Análise gráfica - Merge Sort

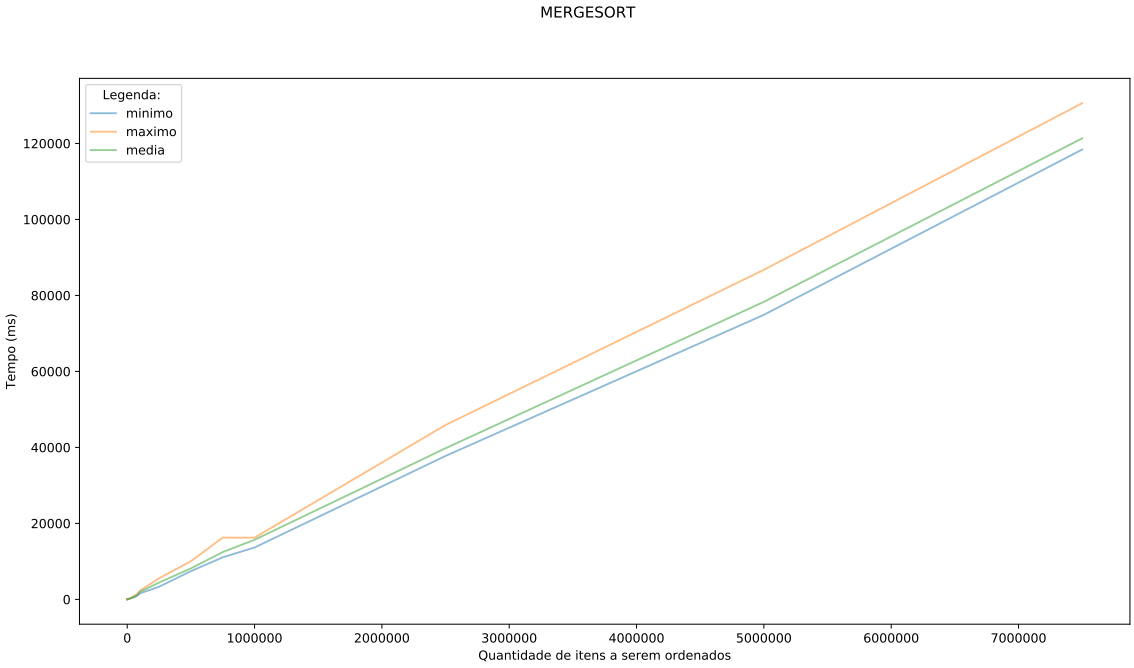

Figura 3: Gráfico de desempenho do algoritmo Merge Sort

O algoritmo apresenta execução rápida se comparado aos anteriores, o que ajuda a qualquer desvio absoluto mínimo ter impacto significativo como variação relativa.

Houve variações consideráveis nas execuções com 1, 2.5, 5 e 7 milhões de registros.

Com um milhão de registros, a duração mínima (13653 ms) apresentou maior desvio em relação à média (15681 ms), 12.9328%.

Já com dois e meio milhões de registros, a duração máxima (45895 ms) apresentou maior desvio em relação à média (39776 ms), 15.3836%. Oito execuções apresentaram desvio abaixo ou próximo a 5%.

Com cinco milhões de registros, o maior desvio da média (78316 ms) é de 10.7628%, com a duração máxima de 86745 ms.

Por fim, com o arquivo de sete e meio milhões, o método se estabiliza, apresentando no máximo 7.61% de variação.

4.1.4 Análise gráfica - Heap Sort

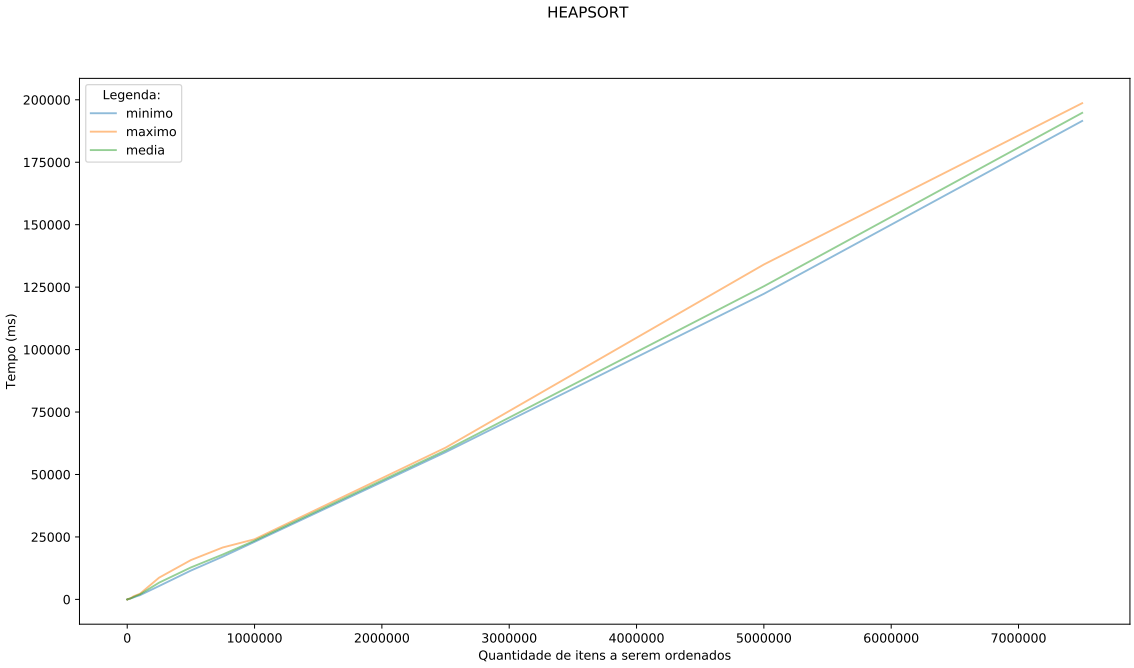

Figura 4: Gráfico de desempenho do algoritmo Heap Sort

É possível notar que próximo a três milhões de registros há o início de um desvio considerável entre o máximo e a média das execuções.

Isso ocorreu em virtude de uma execução das dez, com o arquivo de 5 milhões de registros, que levou 134062 milissegundos, enquanto que todas outras ficaram entre 122 mil e 126 mil ms. O máximo desviou cerca de 6.9544% em relação à média (125345).

A medida que a reta aproxima-se de 7 milhões de registros há uma convergência para média, o que se explica pelo valor máximo (198640) ter desviado apenas 1.9853% da média (194773).

4.1.5 Análise gráfica - Quick Sort

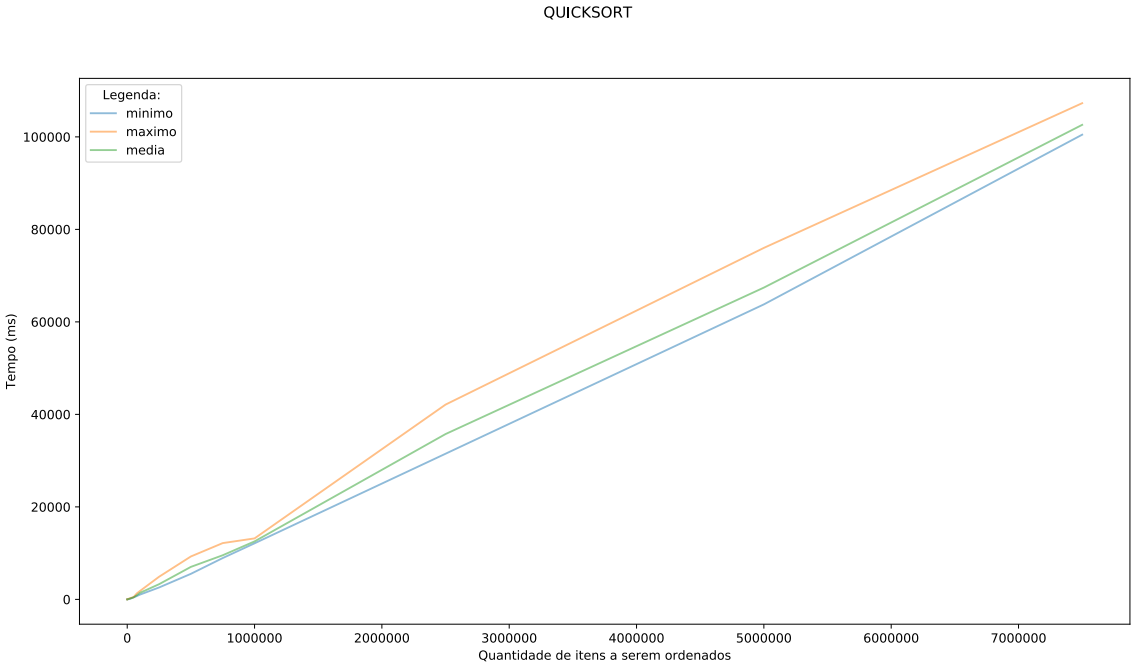

Figura 5: Gráfico de desempenho do algoritmo Quick Sort

Seu maior desvio foi com dois e meio milhões de registros, onde seu valor máximo (42115 ms) diferiu 17.8206% da média (35745 ms).

Já com cinco milhões de dados, o algoritmo apresentou desvio de 12.7336% de sua máxima (75996 ms) em relação a média (67412 ms). Todas outras execuções apresentaram desvio inferior a 5.39%.

Com o maior arquivo, o quick sort estabiliza-se apresentando no máximo 4.5632% de variação.

4.1.6 Análise gráfica - Tim Sort

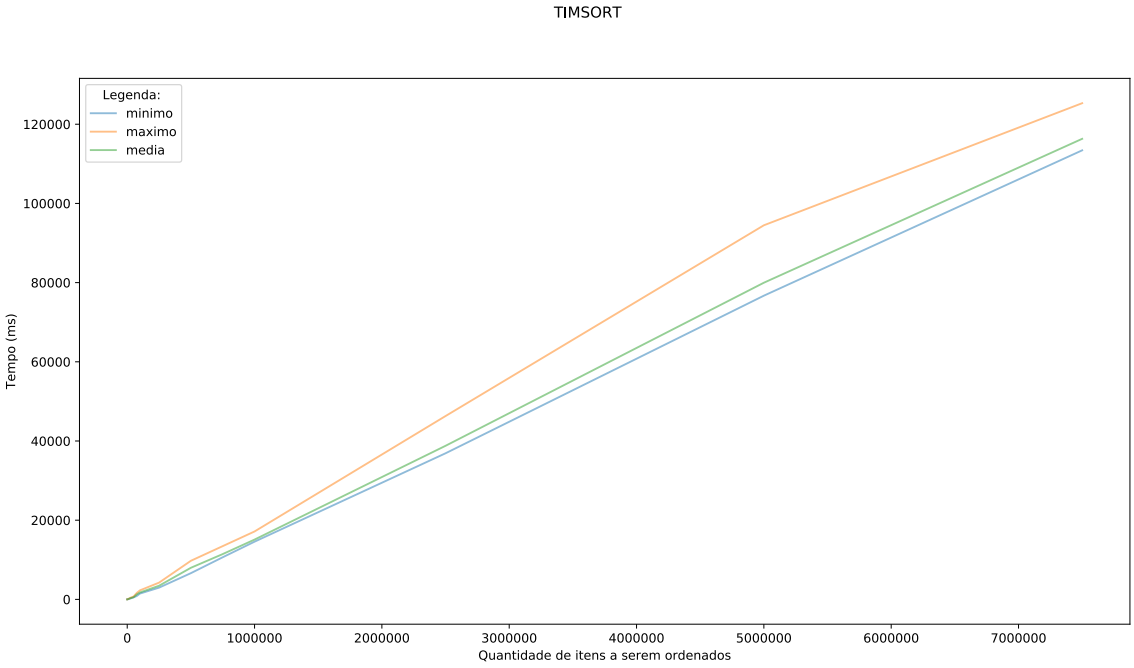

Figura 6: Gráfico de desempenho do algoritmo Tim Sort

...

4.1.7 Análise gráfica - Intro Sort

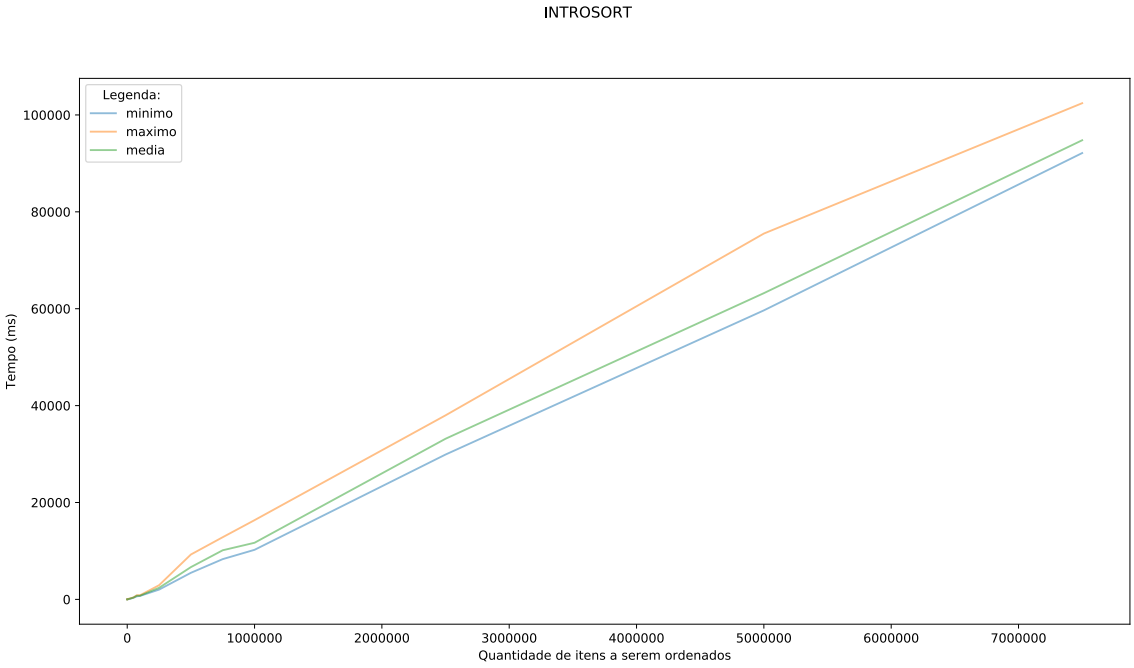

Figura 7: Gráfico de desempenho do algoritmo Intro Sort

4.2 Análise entre os Algoritmos

Considerando que os 24 arquivos de dados possuem grandezas distintas (dezena, centena, milhar, milhão) em relação ao número de registros, o grupo optou por trabalhar cada gráfico abaixo com os 4 arquivos de mesmo unidade. Por exemplo, um gráfico apenas para tratar dezenas (10, 25, 50 e 75), um para centenas, assim por diante.

O procedimento acima foi realizado para garantir melhor visibilidade dos gráficos e das variações dos algoritmos, por menor que forem.

4.2.1 Número de registros: 10 a 75

Por se tratar de uma quantidade ínfima de dados para um computador moderno, os testes com os arquivos com dezenas de dados apresentou anomalias. Como pode ser visto no gráfico abaixo, os únicos algoritmos que apresentaram valores diferentes de zero foram o Heap, Insertion e Selection Sort. Dois (Heap e Selection) desses apresentaram tempo de ordenação zero para 39 das 40 execuções (dez execuções para cada um dos quatro arquivos).

Talvez pela maneira que fora implementado, apenas o Insertion Sort apresentou valores não nulos para todas as execuções com 50 e 75 registros.

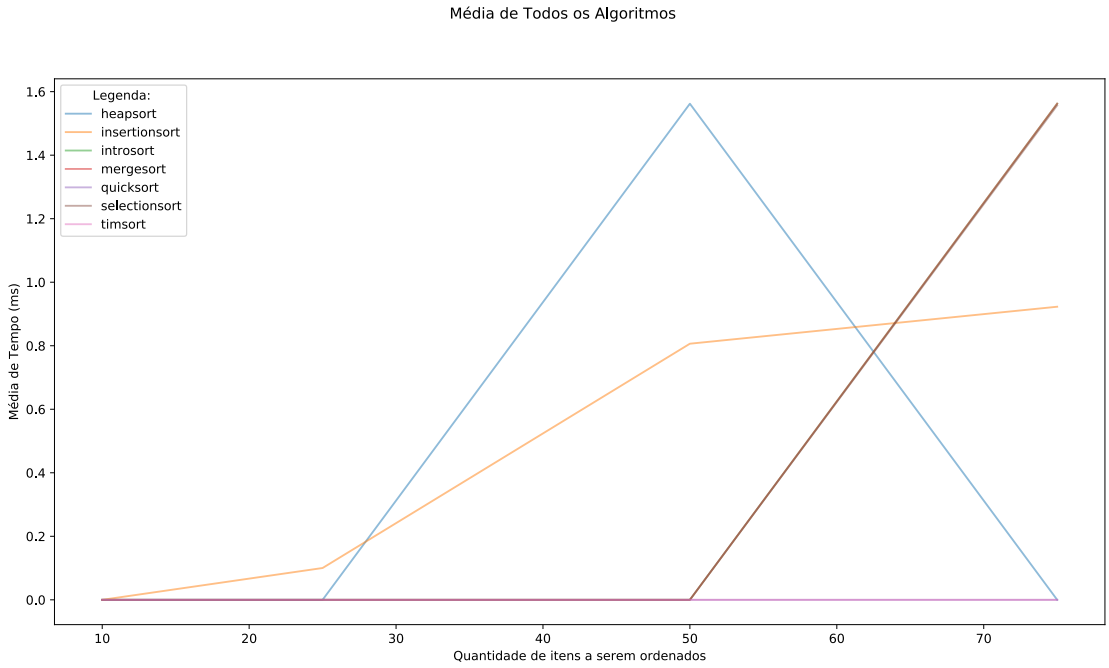

Figura 8: Gráfico com 10 a 75 registros

4.2.2 Número de registros: 100 a 750

Com arquivos de 100 a 750 registros, no gráfico abaixo é possível ver que:

- Os piores algoritmos, como já esperado, são os de ordem quadrática, selection e insertion, respectivamente
- Com cem registros, os melhores algoritmos são: heap, merge, tim e intro sort
- Com 250 registros, o quick sort passa a ser o algoritmo mais rápido para todo o gráfico
- A partir de 500 registros, o heap sort passa a apresentar o pior desempenho entre os 5 melhores; o merge sort compete com o intro sort; sendo o quick e o tim sort, os melhores
- No último arquivo, com 750 registros, os três mais rápidos são: quick, tim e intro sort, respectivamente

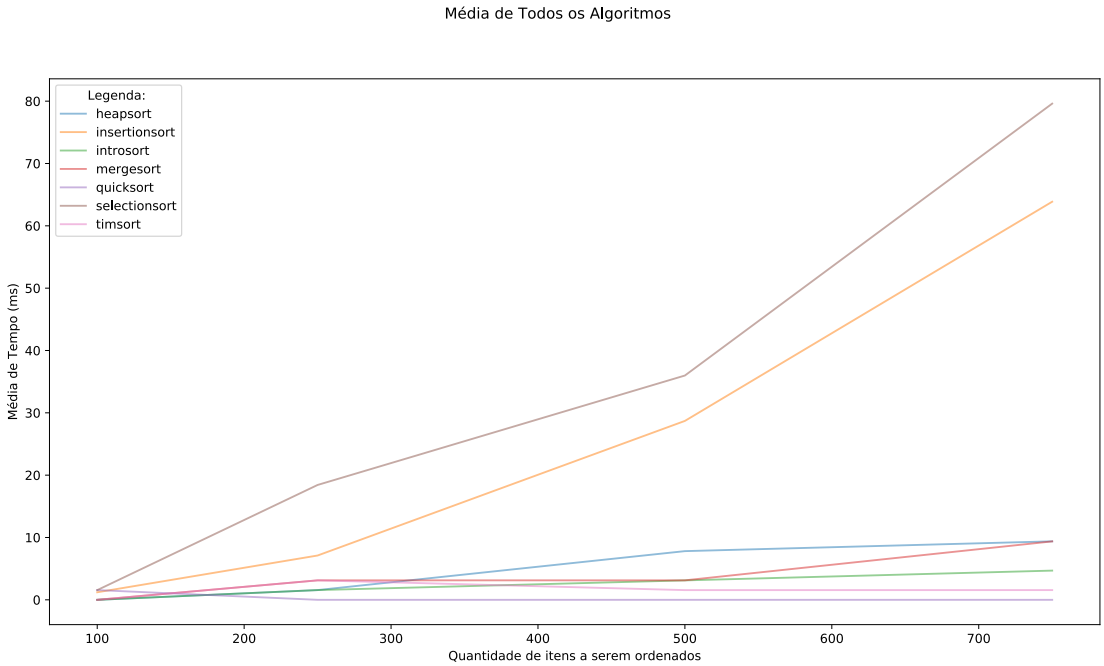

Figura 9: Gráfico com 10 a 75 registros

4.2.3 Número de registros: 1000 a 7500

Após atingir a ordem de milhar, observa-se que:

- O insertion e o selection sort se mantém os menos performáticos. Os cinco melhores algoritmos aproximam do tempo de duração zero.
- Com mil registros, tim, quick e intro ordenam os dados em cerca de 3 milessegundos, enquanto que o merge o faz em cerca de 6.2 ms e o heap, em 7.8
- Com 2500, os três algoritmos mais rápidos são quicksort (14.05 ms), introsort (17.33 ms) e timsort (20.3 ms)
- Ao atingir os 7500 registros, os três melhores permanecem sendo intro (56.24 ms), quick (64,04) e tim (67.16). Insertion sort tem sua duração em milissegundos aproximando-se do número de registros, enquanto selection já ultrapassa o tempo de 1 milessegundo por registro.

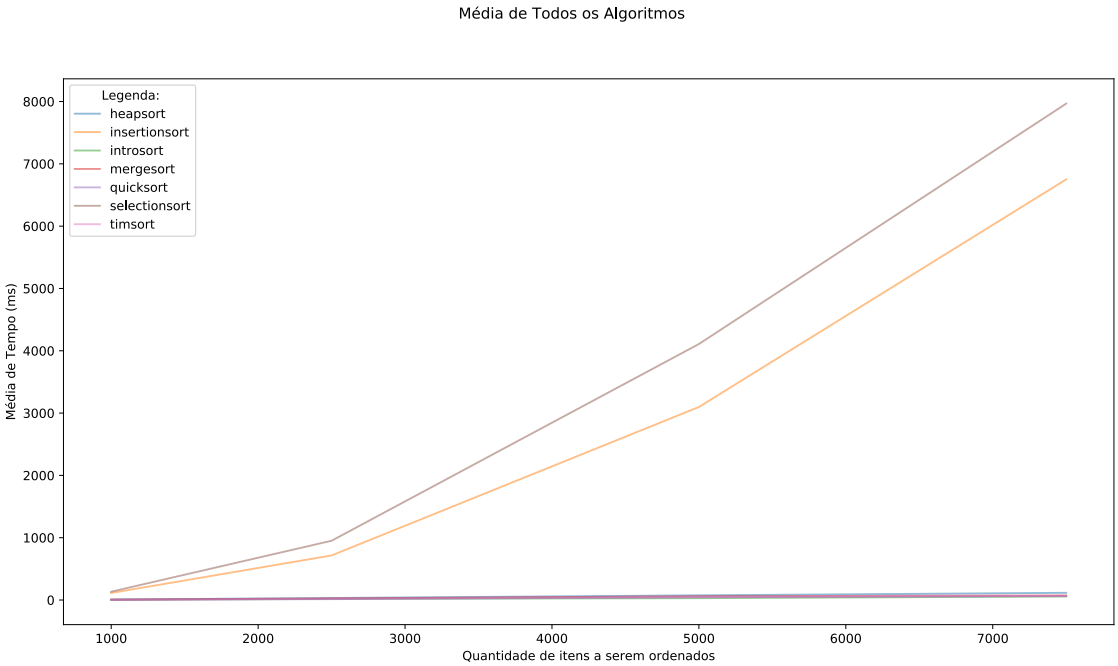

Figura 10: Gráfico com 1000 a 7500 registros

4.2.4 Número de registros: 10000 a 75000

No gráfico abaixo, com dezenas de milhares de registros, temos que:

- Selection e insertion sort começam extremamente próximos. Para 75 mil registros, o insertion sort (910683 ms) já leva em média 12.14 ms por registro e selection sort (899257), 11.99 ms
- Graficamente, é quase impossível distinguir qualquer dos cinco algoritmos melhores
- Intro sort consegue o posto de mais rápido durante todas execuções (750 ms para 75 mil de registros) e arquivos, seguido pelo quick sort (876 ms para 75 mil dados)
- Merge e tim sort competem pelo terceiro lugar. Merge consegue ser mais rápido com 25 mil (273.37 ms contra 309.3 ms) e 75 mil (1063.8 ms contra 1257.52 ms) dados, enquanto que tim prevalece nos 10 mil (103.07 ms contra 106.18 ms) e 50 mil (552.98 ms contra 615.46 ms) registros
- Heap sort tem sua média superada por todos quatro algoritmos anteriores, em todos quatro arquivos de dados

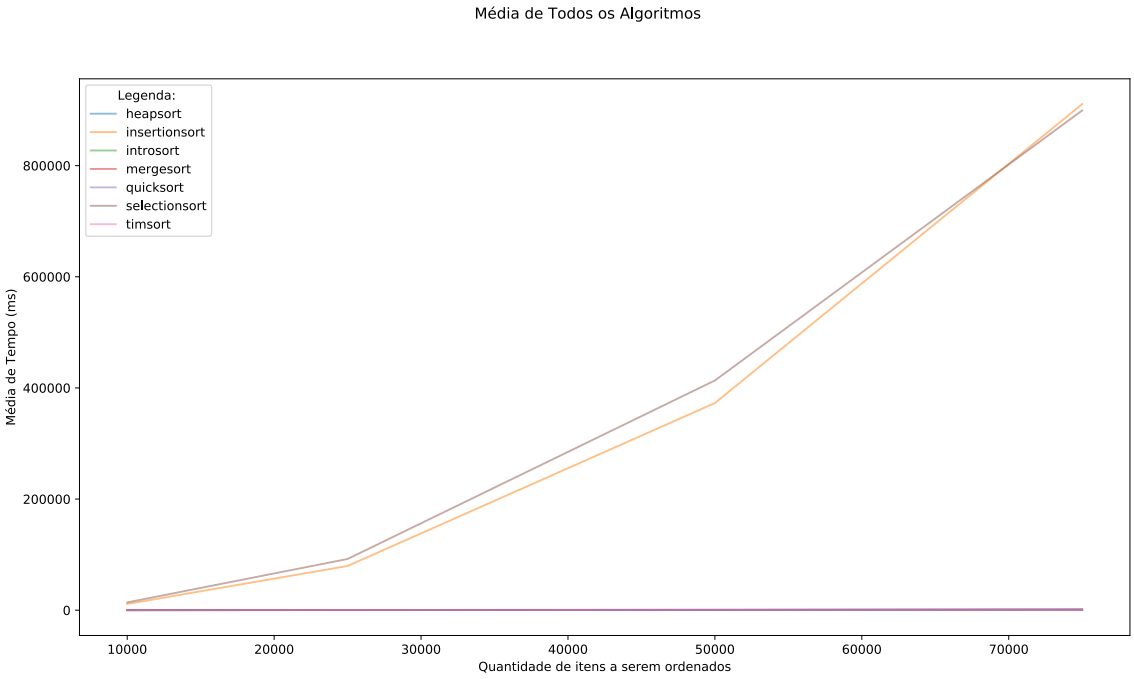

Figura 11: Gráfico com 10000 a 75000 registros

4.2.5 Número de registros: 100000 a 750000

Como algoritmos selection e insertion sort só tivera seus testes para arquivos com menos de cem mil registros, as linhas correspondentes a esses algoritmos não foram plotadas.

Transparece que o heap sort se destaca dos outros quatro por sua menor eficiência, juntamente com a dupla merge e tim sort. Destacam-se introsort que permaneceu como melhor até levar 10138.76 ms para ordenar 750 mil registros e ser ultrapassado pelo quicksort, que o fez em 9539.94 ms.

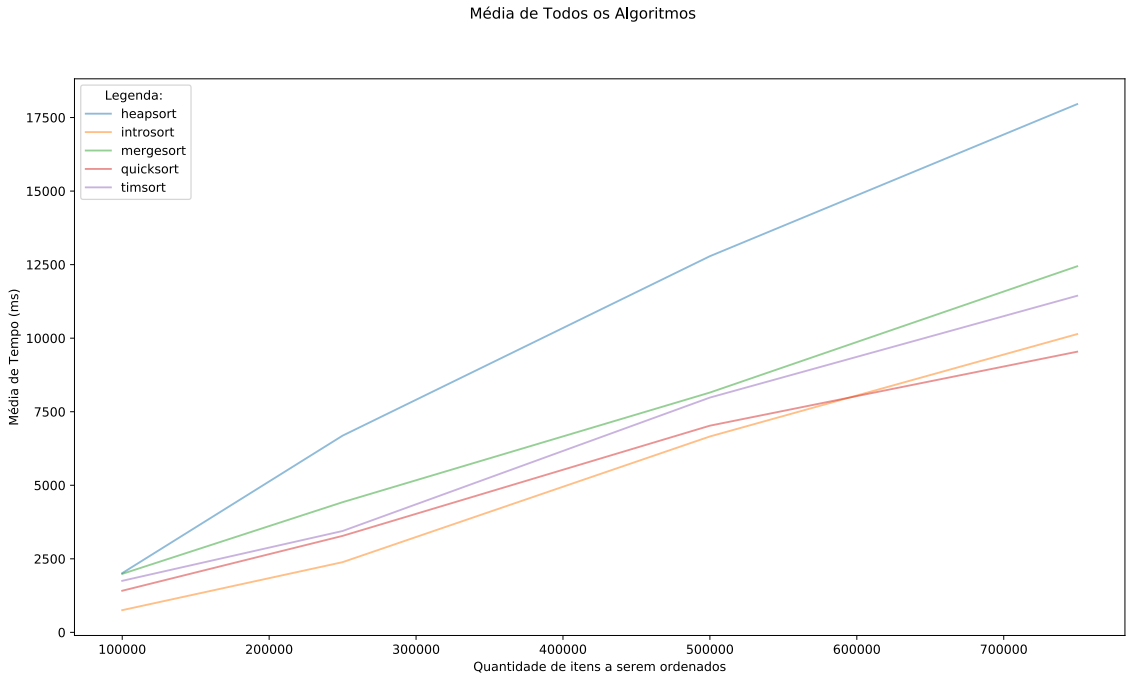

Figura 12: Gráfico com 100000 a 750000 registros

4.2.6 Número de registros: 1000000 a 7500000

De posse de mais de um milhão de dados reduz-se muito a possibilidade de algum algoritmo ter sido eficiente apenas por sorte.

As observações levantas pelo grupo foram:

- Embora a variação de tempo entre as execuções dos algoritmos tenha aumentado, ainda é possível ver uma disputa e troca de posições entre o merge e o tim sort.
- Como esperado, o quick e o intro sort saem como melhores métodos, com destaque para o intro que permanece durante todos arquivos de teste como o mais performático. Com um milhão de registros, o intro sort realizou a ordenação em 11687.73 ms, enquanto que quick levou 12537.14. Essa variância aumenta quando intro sort ordena 7.5 milhões de dados em 94778.95, e quick, em 102603.89, uma diferença de 7824.94 ms ou 7.8 segundos.
- É importante notar que os quatro algoritmos mais performáticos com o maior número de registros foram o quick sort e o merge sort, juntamente à suas derivações (intro e tim sort).
- O método híbrido **introsort** conseguiu superar os três algoritmos (quicksort, heapsort e insertion sort) que o compunha. Há também o caso do tim sort, que triunfou sobre os algoritmos que lhe deram origem (merge e insertion sort), para os casos com 1, 2.5 e 7.5 milhões.

4.3 Interrupção de testes (Timeout)

Como já dito anteriormente, o grupo não executou todos os casos de testes com todos algoritmos. O selection e insertion sort não processaram arquivos de quantidade maior ou igual a cem mil registros.

Ciente de que para cem mil registros, os algoritmos quick, heap e merge sort, em seus piores tempos de execução, tiveram duração de 1812 (1.81 seg), 2249 (2.24 seg) e 2291 milissegundos (2.29 seg), respectivamente, o grupo considerou plausível e viável determinar um limite máximo de cerca de 100 vezes o tempo gasto pelo merge sort como tempo aceitável para interromper a execução dos métodos insertion e selection sort.

Tomando como métrica o custo, em milessegundo por usuário, temos que:

- Para ordenar 100 dados, tomando sua melhor execução como base, de 0 ms, gastou-se 0 ms para ordenar cada usuário
- Para ordenar 750 dados, tomando sua melhor execução (69.0917 ms), gastou-se 0.0921 ms para ordenar cada usuário
- Para ordenar 7500 dados, tomando sua melhor execução (7671.3795 ms), gastou-se 1.0228 ms para ordenar cada usuário
- Para ordenar 75000 dados, tomando sua melhor execução (893709.5222 ms), gastou-se 11.9161 ms para ordenar cada usuário

Com base na média simples acima, selection sort levou 11.9161 ms para cada registro dos 75 mil. Supondo que esses custo se mantivesse fixo, isto é, não aumentasse com o aumento da lista. Apenas com uma multiplicação simples, teria-se que o custo para ordenar 100 mil dados, seriam gastos 1191910 milessegundos ou 19.8651 minutos, para uma execução.

Seriam 3.31 horas para realizar dez execuções, apenas para a ordenação por seleção. Ou seja, no mínimo o dobro, considerando ambos métodos de ordenação.

Dessa forma, tendo em vista que, com 50 mil elementos no conjunto, o algoritmo insertion sort levou 370596 milissegundos (370.6 seg ou 6.176 min), o que já excede o máximo de 229100 (100 * 2291) ms, optou-se por não realizar os testes para valores maiores ou iguais a 100 mil registros, considerando que o tempo de ordenação tende a aumentar quadraticamente com o número de dados.

Referências

[1] KNUTH, Donald Ervin. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3. ed. Reading, Massachusetts: Addison-wesley, 1981. 2 v. P. 138-140.

[2] CORMEN, Thomas H. et al. *Algoritmos: teoria e pratica*. Rio de Janeiro: Elsevier, 2002. 916 p.

[3] HOARE, C. A. R.. *Algorithm 64: Quicksort*. Communications Of The Acm, [s.l.], v. 4, n. 7, 1 jul. 1961. Association for Computing Machinery (ACM).

[4] MUSSER, David R.. Introspective Sorting and Selection Algorithms. Software: Practice and Experience, [s.l.], v. 27, n. 8, p.983-993, ago. 1997. Wiley. `http://dx.doi.org/10.1002/(sici)1097-024x(199708)27:83.0.co;2-#`.

[5] Christopher G. Healey. 2016. *Disk-Based Algorithms for Big Data*. CRC Press, Inc., Boca Raton, FL, USA. p. 34 - 35.

[6] Nicolas Auger, Vincent Jugé, Cyril Nicaud, Carine Pivoteau. *On the Worst-Case Complexity of TimSort*. 26th Annual European Symposium on Algorithms (ESA 2018), Aug 2018, Helsinki, Finland. pp.4:1–4:13, ff10.4230/LIPIcs.ESA.2018.4ff. ffhal-01798381f