Comparativo de Algoritmos de Ordenação

Mestrado Profissional de Computação Aplicada
Disciplina: Algoritmos e Programação
Professora Valéria Farinazzo
Turma: [Turma 01A] - 2022/2
Alunos:
Marcos Antonio Speca Junior – 72256826
Tharles Maicon Freire dos Santos - 72256842

Índice

Resumo

O objetivo deste trabalho foi realizar a comparação de diferentes tipos de algoritmos de ordenação para vetores com diferentes tamanhos (mil elementos, dez mil elementos e cem mil elementos) e diferentes ordenações prévias (aleatório, crescente e decrescente). Os algoritmos testados foram: bolha, inserção, seleção, *mergesort* e *quicksort*.

Desenvolvemos um "notebook" em Python para realizar os testes e para cada execução do algoritmo, para cada tamanho e para cada tipo de ordenação prévia, nós gravamos os tempos de execução em um arquivo de log para futura análise.

Executamos 5 coletas gerais para cada combinação algoritmo, tamanho e tipo, para que não ocorresse nenhum desvio. Por fim consolidamos os números em um arquivo Excel e criamos alguns gráficos para a efetiva comparação e análise.

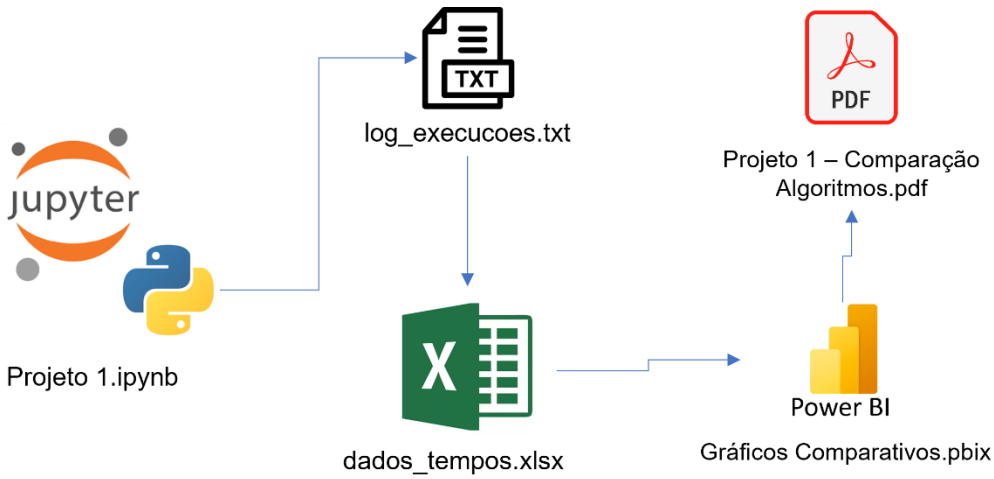
Método

Equipamento Utilizado

Para realizar os testes, utilizamos um equipamento com as seguintes configurações:

- Notebook Dell Inspiron 7580
- Processador Intel Core i7 1.80GHz 4 Cores 8 Logical Processors
- Memória RAM: 16GB
- Disco: SSD M2 500GB
- S.O.: Microsoft Windows 10 Pro

Diagrama de componentes



Os códigos e materiais estão disponíveis em: https://github.com/marcos-speca/mestrado-algoritmos-projeto1

Linguagem de programação e bibliotecas utilizadas

Os algoritmos foram desenvolvidos utilizando a linguagem de programação *Python* em um ambiente de *Jupyter Notebook*, abaixo as versões utilizadas:

- Python: versão 3.8.12
- Jupyter: versão 1.0.0
- Jupyter Server: versão 1.13.5
- Numpy: versão 1.21.5
- Pandas: versão 1.4.2

Para apoiar o desenvolvimento utilizamos algumas bibliotecas para o Python conforme abaixo:

```
import numpy as np
import pandas as pd
from datetime import datetime
import sys
import inspect
```

Cada biblioteca possui um propósito:

- numpy: auxilia na criação de vetores aleatórios
- pandas: auxilia na manipulação dos dados para consolidação e posterior gravação em arquivos texto e excel (utilizado nas análises).
- datetime: auxilia na gravação e cálculos de diferença de tempo de execução
- sys: biblioteca utilizada para alterar o parâmetro de limite de recursão do Python*
- inspect: biblioteca utilizada para verificar o tamanho da pilha.

Limite de Recursão no Python

O Python possui por padrão um limite de pilha de recursão para evitar estouro de pilha e recursões infinitas[1].

Este limite pode ser alterado através da função "*setrecursionlimit*" existente na biblioteca sys.

```
# Para resolver o problema da recursão do Python
sys.setrecursionlimit(3000)
```

Apesar de disponível, na execução dos scripts no presente teste vimos que não foi necessário alterar o parâmetro pois o valor padrão da recursão (3000) já era suficiente para garantir a execução dos algoritmos de recursão, mesmo para os vetores grandes de cem mil elementos.

[1] https://docs.python.org/3/library/sys.html#sys.setrecursionlimit

Algoritmos de Ordenação em Python

Os algoritmos utilizados foram adaptados e ajustados da internet[2] para acelerar o desenvolvimento, visto que o objetivo principal do trabalho é a comparação dos resultados. Para cada algoritmo, criamos funções que recebem um vetor como parâmetro e realizam a ordenação.

Algoritmo Bolha (bubblesort):

```
# Função "Bubble Sort"
def bubble_sort(lista):
    # Percorre cada elemento da lista
    for i in range(len(lista)-1, 0, -1):
        # Flutua o maior elemento para a posição mais a direita
        for j in range(i):
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
```

Algoritmo Inserção (insertion):

```
# Função "Insertion Sort"
def insertion_sort(lista):
    # Percorre cada elemento da lista
    for i in range(1, len(lista)):
        k = i
        # Insere o pivô na posição correta
        while k > 0 and lista[k] < lista[k-1]:
            lista[k], lista[k-1] = lista[k-1], lista[k]
            k = k - 1
```

Algoritmo Seleção (selection):

```
# Função "Selection Sort"
def selection_sort(lista):
    # Percorre todos os elementos de L
    for i in range(len(lista)):
        menor = i
        # Encontra o menor elemento
        for k in range(i+1, len(lista)):
            if lista[k] < lista[menor]:
                menor = k
        # Troca a posição do elemento i com o menor
```

[2] Fonte: https://colab.research.google.com/drive/1rgdEwvn3wfD5-wUVE_9MuZvEdy7kjdNv?usp=sharing

```
        lista[menor], lista[i] = lista[i], lista[menor]
```

Algoritmo Merge Sort (merge):

```
# Função "Merge Sort"
def merge_sort(lista):
    if len(lista) > 1:
        meio = len(lista)//2
        LE = lista[:meio]   # Lista Esquerda
        LD = lista[meio:]   # Lista Direita

        # Aplica recursivamente nas sublistas
        merge_sort(LE)
        merge_sort(LD)

        # Quando volta da recursão inicia aqui!
        i, j, k = 0, 0, 0
        # Faz a intercalação das duas listas (merge)
        while i < len(LE) and j < len(LD):
            if LE[i] < LD[j]:
                lista[k] = LE[i]
                i += 1
            else:
                lista[k] = LD[j]
                j += 1
            k += 1

        while i < len(LE):
            lista[k] = LE[i]
            i += 1
            k += 1

        while j < len(LD):
            lista[k] = LD[j]
            j += 1
            k += 1
```

Algoritmo Quick Sort (quick):

```
# Função "Quick Sort"
def quick_sort(lista):
    if len(lista) <= 1:
        return lista
    # Consideramos o Pivô como elemento do meio da lista.
    m = lista[len(lista)//2]
    # Chamada recursiva
    return  quick_sort([x for x in lista if x < m]) + [x for x in lista if x
== m] + quick_sort([x for x in lista if x > m])
```

No algoritmo de Quick Sort, fizemos uma adaptação diferente do algoritmo encontrado na internet para ele pegar o elemento do meio do vetor como “pivô” (a versão encontrada pegava o primeiro elemento).

Função que executa e grava os tempos

Para possibilitar a execução e tomada de tempos de forma continuada para os diferentes tipos de algoritmos, tamanhos e tipos de ordenação prévia, desenvolvemos uma função que recebe como parâmetro o método de ordenação (algoritmo), o vetor a ser ordenado e o tipo de ordenação prévia como parâmetros.

Esta função irá chamar as funções específicas de cada algoritmo, guardando o tempo de início e tempo de fim da execução.

Após a execução do algoritmo, esta mesma função grava um "log" de execução em um arquivo de texto (log_execucoes.txt). Adotamos esta abordagem para não perder alguma informação caso o processo ficasse travado em alguma etapa, ou seja, com esta estratégia garantiríamos que o tempo já coletado fosse armazenado em um arquivo de logs caso uma execução futura parasse ou travasse.

Segue abaixo um trecho da função que verifica qual método selecionado, inicia a execução, grava os tempos e depois grava os dados de log no arquivo.

```
# Função que executa a ordenação e grava o log de execuções (para futura
comparação)
def exec_ordenacao(metodo,arr,tipo):

    log = None

    if metodo == "bubble":
        start_time = datetime.now()
        bubble_sort(arr)
        end_time = datetime.now()
        seconds = (end_time-start_time).total_seconds()

        log = metodo + ";" + tipo +  ";" + str(len(arr)) + ";" +
start_time.strftime("%Y-%m-%d %H:%M:%S") + ";" + end_time.strftime("%Y-%m-%d
%H:%M:%S") + ";" + str(seconds)
```

[… o código continua para os demais métodos ...]

Por último na função, existe uma verificação se há algum log para gravar e a função adiciona esta linha no arquivo.

```
    if(log):
        with open('log_execucoes.txt', 'a', encoding='utf-8') as f:
            f.write('\n')
            f.write(log)
        print(log)
```

Criação dos vetores para os testes

Para realização dos testes foi necessário a criação de 9 vetores conforme abaixo:

- Vetores de mil elementos: 1 Aleatório, 1 Crescente e 1 Decrescente
- Vetores de dez mil elementos: 1 Aleatório, 1 Crescente e 1 Decrescente
- Vetores de cem mil elementos: 1 Aleatório, 1 Crescente e 1 Decrescente

Utilizamos uma função "randint" da biblioteca "numpy" para a geração de números aleatórios[3], e laços simples de repetição para a geração dos vetores crescentes e decrescentes.

Nesta função definimos o valor máximo como múltiplo de 5 do tamanho do vetor para evitar muitas repetições.

```
# Vetores Aleatórios
a_1K = np.random.randint(1,5000,1000).tolist()
a_10K = np.random.randint(1,50000,10000).tolist()
a_100K = np.random.randint(1,500000,100000).tolist()

# Vetores Crescentes
c_1K =[]
c_10K =[]
c_100K =[]
for i in range(1,1001): c_1K.append(i)
for i in range(1,10001): c_10K.append(i)
for i in range(1,100001): c_100K.append(i)

# Vetores Decrescentes
d_1K = []
d_10K = []
d_100K = []
for i in range(1000,0,-1): d_1K.append(i)
for i in range(10000,0,-1): d_10K.append(i)
for i in range(100000,0,-1): d_100K.append(i)
```

A geração dos vetores é feita apenas uma vez por execução do script para garantir que os algoritmos trabalharam no mesmo vetor aleatório. Ao passar o vetor para cada execução nós criamos uma cópia para evitar que execuções futuras trabalhem em um vetor alterado ou ordenado.

[3] Função Randint https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html

Execução da coleta de tempos

Após a criação das funções e vetores conforme os tópicos anteriores, para cada tipo de algoritmo realizamos 9 execuções, considerando os diferentes tipos de ordenação prévia e tamanhos dos vetores.

Abaixo um exemplo para o algoritmo quick sort (para cada algoritmo de ordenação há um código similar):

```
# Teste Quick Sort

# Vetores 1K
qs_a_1K = a_1K.copy()
exec_ordenacao("quick",qs_a_1K,"Aleatório")

qs_c_1K = c_1K.copy()
exec_ordenacao("quick",qs_c_1K,"Crescente")

qs_d_1K = d_1K.copy()
exec_ordenacao("quick",qs_d_1K,"Decrescente")


# Vetores 10K
qs_a_10K = a_10K.copy()
exec_ordenacao("quick",qs_a_10K,"Aleatório")

qs_c_10K = c_10K.copy()
exec_ordenacao("quick",qs_c_10K,"Crescente")

qs_d_10K = d_10K.copy()
exec_ordenacao("quick",qs_d_10K,"Decrescente")

# Vetores 100K
qs_a_100K = a_100K.copy()
exec_ordenacao("quick",qs_a_100K,"Aleatório")

qs_c_100K = c_100K.copy()
exec_ordenacao("quick",qs_c_100K,"Crescente")

qs_d_100K = d_100K.copy()
exec_ordenacao("quick",qs_d_100K,"Decrescente")
```

Para cada chamada da função “exec_ordenacao”, uma linha no arquivo de log será gravada com os tempos de execução dos algoritmos.

Observe que antes de cada execução, criamos a cópia do vetor e passamos esta cópia para a execução da função.

Consolidação dos dados

Por fim o último trecho do código consolida os tempos coletados nesta execução com os tempos de coletas anteriores em um arquivo de excel único que servirá de base para as análises.

```
# Abre o arquivo de log em um DF
df_log = pd.read_csv('./log_execucoes.txt',sep=';')

# Carrega os dados do Excel
df_dados = pd.read_excel('./dados_tempos.xlsx',sheet_name='dados')

# Busca o Ultimo ID da Coleta
ultima_coleta = max(df_dados['id_coleta'])
id_coleta = int(ultima_coleta)+1
df_log['id_coleta'] = id_coleta

#Junda os dois Data Frames
df_dados = pd.concat([df_dados,df_log])

# Grava Dados no arquivo Excel
df_dados.to_excel('dados_tempos.xlsx',sheet_name='dados', index=False)
```

Executamos ao todo 5 coletas completas, considerando todos os algoritmos, tamanhos e tipos de vetores foram 225 tomadas de tempo ao todo.

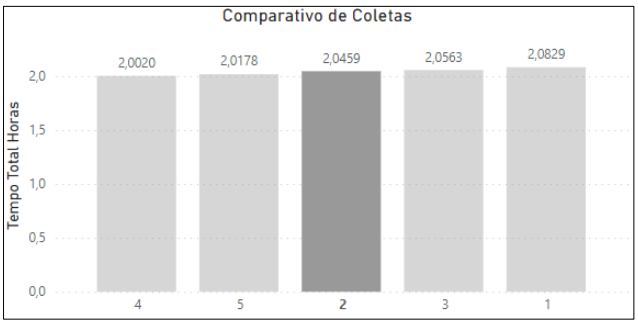
Análise dos Dados

Neste tópico iremos apresentar o comparativo dos tempos coletados, análises a respeito sobre a eficiência de cada algoritmo e por fim iremos discorrer sobre a análise assintótica dos algoritmos versus os tempos obtidos.

Coletas realizadas

Foram realizadas 5 coletas de tempos ao todo. Cada coleta consistiu em executar cada um dos 5 algoritmos de ordenação (bolha, seleção, inserção, mergesort e quicksort) para cada um dos 3 tipos de vetores, para cada um dos 3 tamanhos de vetores diferentes, portanto cada coleta gerou 45 registros de tempo, indicando o tamanho, tipo e algoritmo de ordenação utilizado.

Realizamos 5 coletas em diferentes momentos para eliminar possíveis variações significativas de dados.



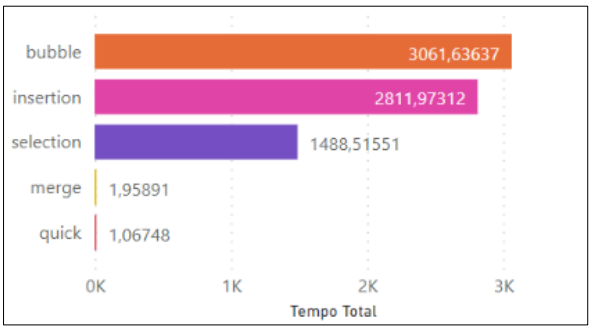
Como podemos ver conforme gráfico acima, os tempos totais (neste gráfico em horas) foram muito semelhantes para cada coleta, por isso decidimos selecionar a coleta com o tempo total médio para realizar as demais análises de comparação entre os algoritmos.

Nos tópicos a seguir detalharemos cada análise realizada.

Comparação entre algoritmos

Tempos totais

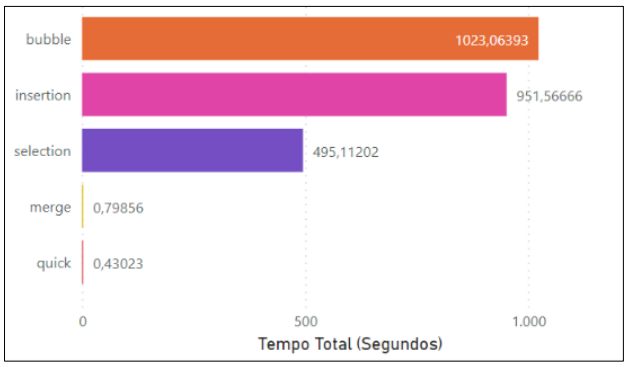
O gráfico abaixo apresenta a comparação de tempos totais (em segundos) entre os algoritmos:



- O algoritmo bolha é o mais demorado de todos, levando no total 3.062 segundos (51 minutos) para ordenar todos os 9 vetores.
- Os algoritmos que não usam recursão são os que levam mais tempo para ordenar os vetores (bolha, seleção e inserção).
- Os algoritmos Bolha e Inserção estão bem próximos, sendo os mais lentos; seguidos do algoritmo de seleção, que dentre os algoritmos que não utilizam recursão é o mais rápido.
- Os algoritmos que utilizam recursão são os mais rápidos e ordenaram os vetores em alguns segundos, sendo o algoritmo Quicksort o mais rápido de todos realizando a ordenação dos 9 vetores em um segundo.

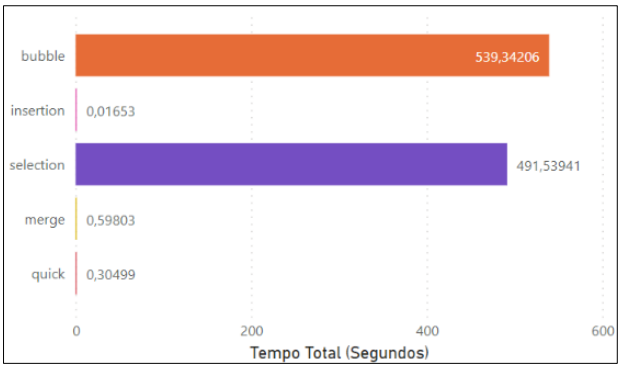
Vetores Aleatórios

Considerando apenas vetores aleatórios temos uma situação semelhante ao total, sendo o algoritmo Bolha sendo o mais lento e o Quicksort sendo o mais rápido:



Vetores Crescentes

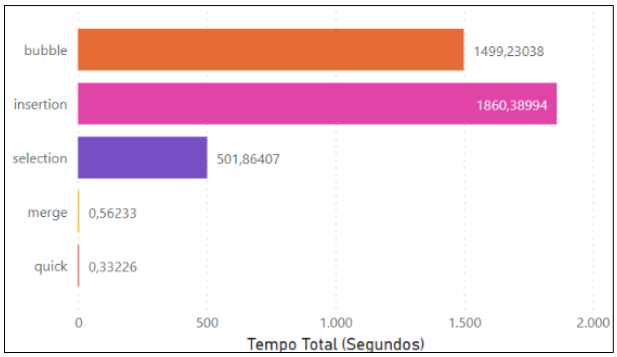
Já considerando vetores que já estejam pré-ordenados de forma crescente temos uma situação interessante, veja que o algoritmo de Inserção acaba sendo o mais rápido de todos:



Isso ocorre devido ao algoritmo de inserção que no seu melhor caso (vetor já ordenado), só vai realizar uma passagem no vetor, sem realizar nenhuma troca, portanto em seu melhor caso ele é o mais rápido.

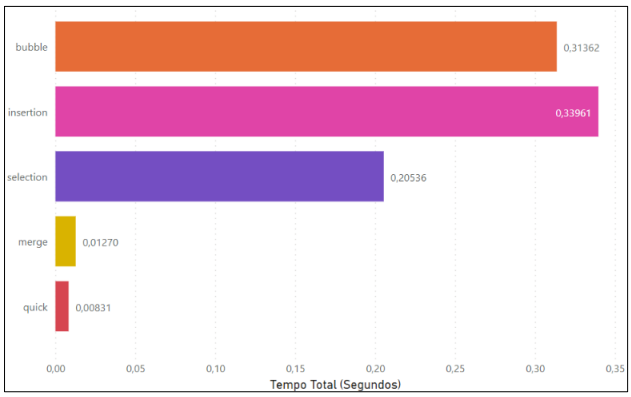
Vetores Decrescentes

No caso de vetores decrescentes (que podemos considerar como “pior caso”), o algoritmo de inserção é o que leva mais tempo para ordenar os vetores.



Vetores de mil elementos

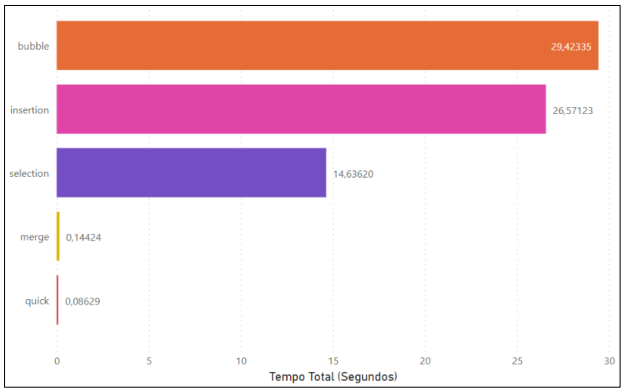
Comparando a performance de ordenação para vetores menores (mil elementos), todos os algoritmos realizaram a tarefa em menos de um segundo, o algoritmo que mais leva tempo para ordenar é o algoritmo de inserção, seguido dos algoritmos bolha, seleção merge e quicksort.



Vetores dez mil elementos

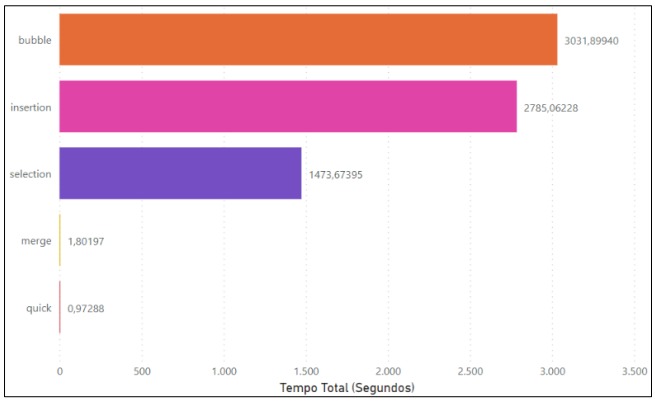
Para vetores médios, o tempo total dos algoritmos começa a seguir o padrão encontrado nos tempos totais, sendo o algoritmo bolha o mais lento e o quicksort o mais rápido.

Para estes vetores já é possível ver a grande diferença entre os algoritmos com recursão e os algoritmos sem recursão.



Vetores de cem mil elementos

Os maiores vetores testados foram os vetores de cem mil elementos, neste caso os algoritmos mais lentos foram bolha, inserção e seleção, seguidos pelos mais rápidos: merge e quicksort.



Eficiência dos Algoritmos

A tabela abaixo demonstra os diferentes tempos totais de ordenação, para cada algoritmo, tamanho de vetor e tipo de ordenação prévia.

Tipo	bubble	insertion	selection	merge	quick
Aleatório					
1000	0,10	0,13	0,08	0,00	0,00
10000	9,85	9,07	4,89	0,06	0,04
100000	1.013,11	942,37	490,14	0,74	0,39
Crescente					
1000	0,06	0,00	0,06	0,00	0,00
10000	5,18	0,00	4,79	0,04	0,02
100000	534,10	0,02	486,69	0,55	0,28
Decrescente					
1000	0,15	0,21	0,06	0,01	0,00
10000	14,39	17,50	4,96	0,05	0,02
100000	1.484,69	1.842,68	496,84	0,51	0,31

Considerando todos as variações de tamanhos de vetores e ordenações prévias, podemos concluir que o Quicksort é o algoritmo mais rápido para a ordenação dos vetores.

Outra conclusão relevante é a respeito do algoritmo inserção, onde dependendo da ordenação prévia do vetor ele pode ser o mais rápido (no melhor caso) e o mais lento (no pior caso).

No próximo tópico faremos uma análise a respeito da análise assintótica dos algoritmos versus os tempos coletados neste trabalho.

Análise Assintótica versus Tempos Obtidos

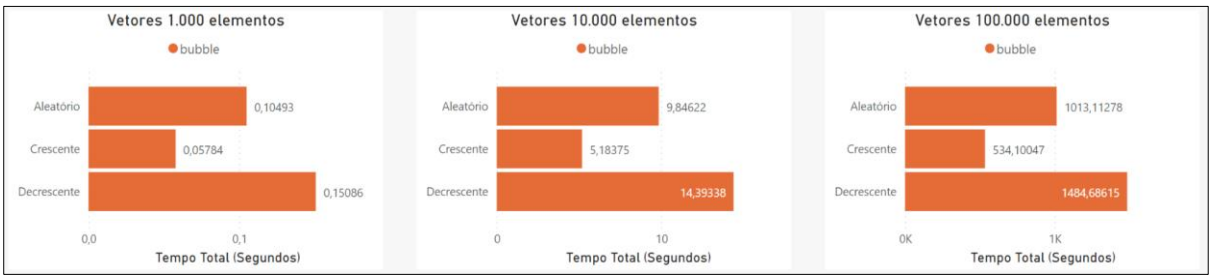
A análise assintótica nos permite simplificar a análise de complexidade de um algoritmo, para os algoritmos de ordenação temos a seguinte tabela:

Algoritmo	Melhor Caso	Pior Caso
Bolha (bubble)	O(n)	$O(n^2)$
Inserção (insertion)	O(n)	$O(n^2)$
Seleção (selection)	$O(n^2)$	$O(n^2)$
Merge Sort (merge)	O(n log n)	O(n log n)
Quick Sort (quick)	O(n log n)	$O(n^2)$*

* No caso do algoritmo Quicksort, o pior caso se refere ao caso de a arvore ficar totalmente desbalanceada.

Algoritmo Bolha

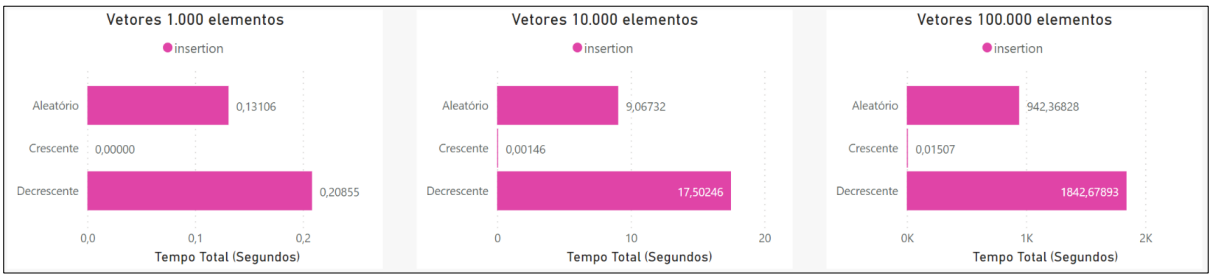
Considerando a análise assintótica do algoritmo em seu pior caso (vetor decrescente) e de seu melhor caso (vetor crescente), e comparando com os tempos obtidos (gráfico abaixo) podemos ver que os tempos condizem com a análise assintótica indicando que o algoritmo é mais rápido no melhor caso e mais demorado no pior caso.



Porém o algoritmo não apresentou diferenças significativas (quadráticas) entre o melhor e o pior caso.

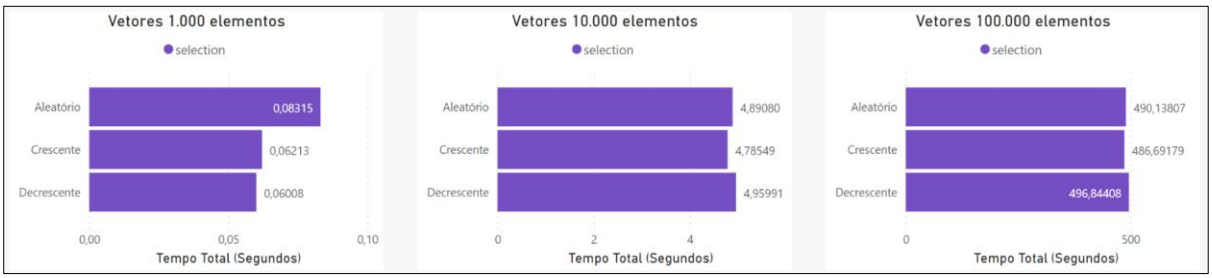
Algoritmo Inserção

Para o algoritmo de inserção, os tempos obtidos também refletem o melhor e o pior caso, e neste algoritmo o melhor caso foi extremamente rápido, confirmando a diferença entre O(n) do melhor caso e O (n^2) do pior caso.



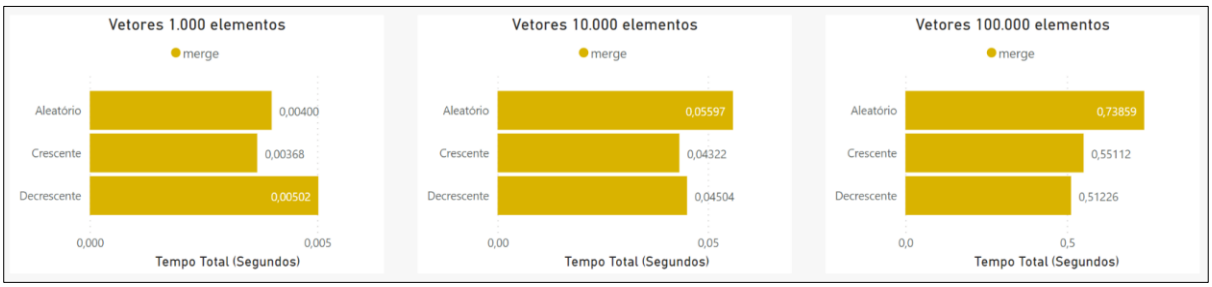
Algoritmo Seleção

Para o algoritmo de seleção, o pior caso e o melhor caso são os mesmos O(n^2) porque ele sempre irá percorrer o vetor para encontrar o menor elemento e fará isso para todos os elementos, e isso é o que podemos visualizar nos tempos obtidos.



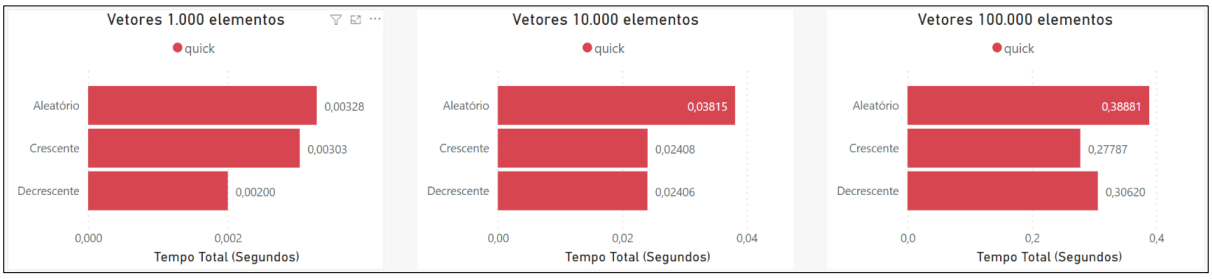
Algoritmo Merge Sort

No caso do merge sort, a análise assintótica indica que tanto para o melhor caso, quanto para o pior caso a complexidade será O(n log n). Os números obtidos ficaram compatíveis com a análise assintótica.



Algoritmo Quick Sort

O quick sort é o caso mais rápido de todos, porém em seu pior caso a complexidade pode ser O (n^2), porém ao garantirmos no próprio algoritmo utilizado neste trabalho para que o elemento "pivot" seja sempre o elemento do meio, a complexidade do algoritmo executando as ordenações foram semelhantes para o pior e o melhor caso, cuja complexidade é O (n log n).



Conclusão

Ao comparar os diferentes tipos de algoritmos de ordenação, considerando casos variados como vetores aleatórios, crescentes e decrescentes e também de tamanhos diferentes (mil, dez mil e cem mil elementos) entendemos que a análise assintótica foi condizente com os tipos obtidos.

Podemos citar alguns pontos relevantes dentro das análises realizadas:

- O algoritmo de bolha é realmente o mais lento, e condiz com sua análise assintótica mesmo que os tempos do pior e do melhor caso não sejam tão diferentes entre si.
- O algoritmo de inserção é o mais lento para os piores casos, porém o mais rápido para os melhores casos, sendo que os tempos evidenciaram as diferenças entre O(n) e O(n^2).
- O algoritmo de seleção teve os tempos semelhantes em todas as situações de vetores (aleatórios, crescentes e decrescentes), isso também condiz com a análise assintótica do algoritmo.
- Os tempos obtidos para o algoritmo merge sort também foi condizente em relação a sua análise assintótica, ficando bem próximos no melhor e no pior caso.
- Já o algoritmo Quicksort, o melhor algoritmo de todos foi eficiente em todos as situações (pior e melhor caso), porém vale ressaltar que o algoritmo utilizado neste teste minimiza a recursão desbalanceada no pior caso pois pega sempre o elemento do meio do vetor.