

Computational Thinking using Python

Prof. Fernando Almeida

CHECKPOINT 5

Análise de dados através dos
Algoritmos de Ordenação

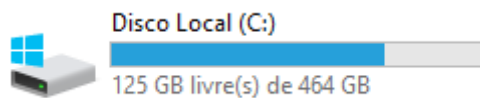


Melissa de Oliveira Pecoraro

RM: 98698



Configuração da Máquina:

- CPU-Processador: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz 2.20 GHz
- RAM instalada: 8,00 GB
- Produto BaseBoard 80BC



Tipo: Disco Local

Sistema de arquivos: NTFS

	Espaço usado:	364.441.038.848 bytes	339 GB
	Espaço livre:	134.440.996.864 bytes	125 GB

Capacidade: 498.882.035.712 bytes 464 GB



Unidade C:

Limpeza de Disco

Bubble Sort

Ordenação por flutuação

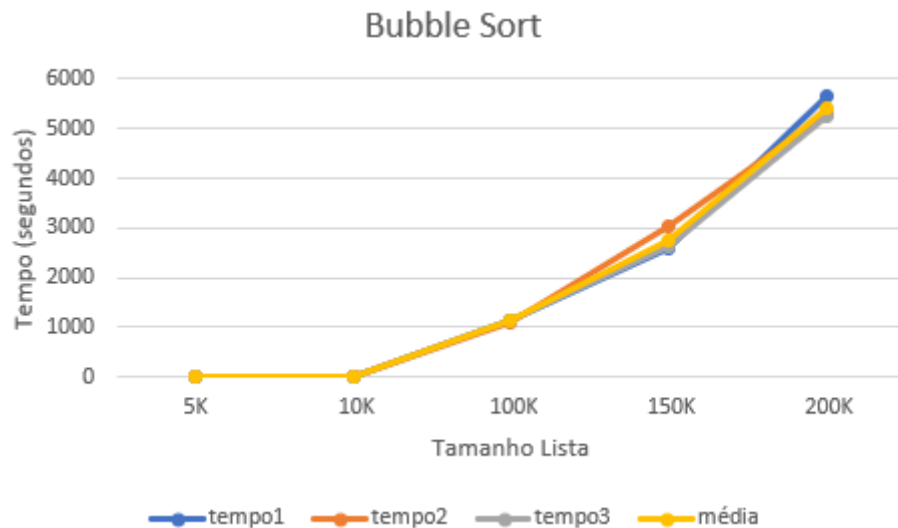
Esse algoritmo é o mais básico e consiste em realizar a ordenação de uma determinada lista de elementos, percorrendo-a várias vezes, e a cada passagem o maior elemento vai para o topo, esse processo acontece com a comparação de um elemento com o que está ao seu lado, e a troca ocorre se não estiverem ordenados até que a lista por completo se ordene.

```
#complexidade: O(n*2)
#bubble Sort (ordenação por flutuação)
def bubble_sort(lista):
    # obtém o tamanho da lista
    tam = len(lista)
    # variável para rastrear se houve alguma troca durante a iteração
    troca = False
    # loop externo para percorrer toda a lista
    for i in range(tam-1):
        # loop interno para comparar elementos adjacentes e fazer trocas se necessário
        for j in range(0, tam-i-1):
            # compara o elemento atual com o próximo e troca se estiverem na ordem errada
            if lista[j] > lista[j+1]:
                troca = True
                # realiza a troca de elementos
                lista[j], lista[j+1] = lista[j+1], lista[j]
        # verifica se houve trocas durante esta iteração
        if not troca:
            # se não houve trocas, a lista está ordenada e podemos interromper
            return
```

Com algoritmo Bubble Sort:

- Lista de 5.000 elementos
- Lista de 10.000 elementos
- Lista de 100.000 elementos
- Lista de 150.000 elementos
- Lista de 200.000 elementos

algoritmo	tamanho lista	tempo1	tempo2	tempo3	média
bubble sort	5K	2,65	2,585	2,65	2,62
bubble sort	10K	12,151	11,633	11,988	11,924
bubble sort	100K	1160,41	1120,276	1160,357	1147,01
bubble sort	150K	2608,509	3027,528	2623,444	2753,16
bubble sort	200K	5674,235	5275,666	5247,01	5398,97



O algoritmo Bubble Sort foi o mais complexo, demorado e trabalhoso, tentei começando com os valores passados no documento, mas acabou pegando muito tempo. Fazendo algumas tentativas consegui achar a melhor opção com esses tamanhos de lista: 5.000, 10.000, 100.000, 150.000 e 200.000, mesmo não tendo uma diferença tão grande entre as 5 listas, foi o que melhor se adequou e o que minha máquina aguentou. Com certeza esse algoritmo foi aquele que mais rendeu, em questão de demora de tempo, em fazer a troca de elementos da lista e ordená-la. É possível observar, e principalmente com esse algoritmo, que quanto maior a lista mais tempo foi gasto para ordená-la.

Existe uma diferença pequena entre as 3 execuções e com isso conseguimos ter garantia do tempo, executando mais de uma vez e calculando também a média.

Selection Sort

Ordenação por seleção

Um algoritmo um pouco mais eficiente que o Bubble Sort, mas ainda simples, ele ordena uma lista de elementos não ordenados e ele pega o primeiro menor número e passa para a primeira posição, depois o segundo menor número e passa para a segunda posição e assim sucessivamente, fazendo a troca com os números não ordenados e trocando somente o necessário, e esse processo se repete até que toda a lista esteja ordenada.

```

#complexidade: O(n*2)
#selection Sort (ordenação por seleção)
def selection_sort(lista):
    # percorre a lista
    for i in range(len(lista)):
        # assume que o índice do menor elemento é o índice atual
        min_index = i

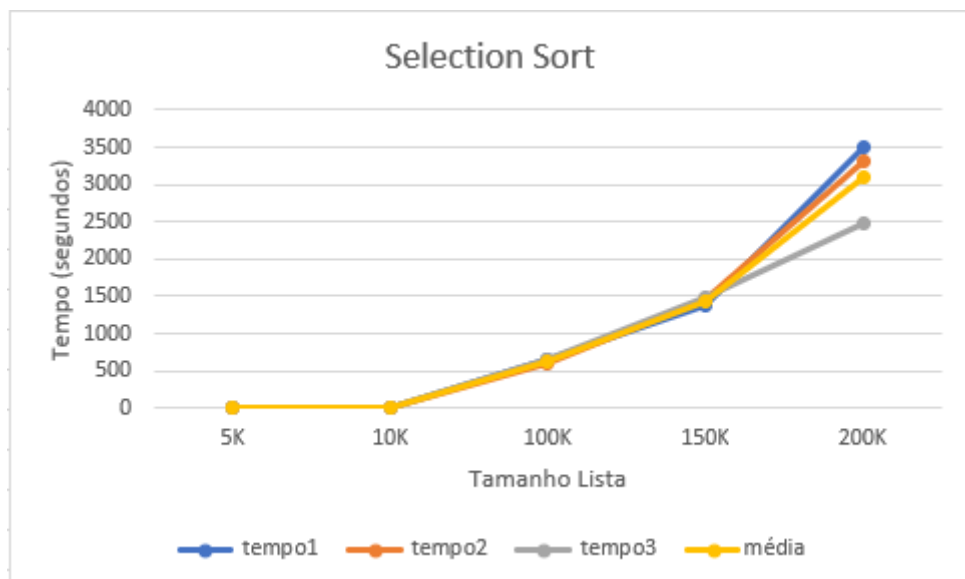
        # encontra o índice do menor elemento a partir do índice atual até o final da lista
        for j in range(i+1, len(lista)):
            #seleciona o menor elemento a cada iteração
            if lista[j] < lista[min_index]:
                min_index = j
        # troca o elemento atual com o menor elemento encontrado
        lista[i], lista[min_index] = lista[min_index], lista[i]

```

Com algoritmo Selection Sort:

- Lista de 5.000 elementos
- Lista de 10.000 elementos
- Lista de 100.000 elementos
- Lista de 150.000 elementos
- Lista de 200.000 elementos

algoritmo	tamanho lista	tempo1	tempo2	tempo3	média
selection sort	5K	1,415	1,564	2,141	1,706
selection sort	10K	5,141	4,866	5,593	5,2
selection sort	100K	653,776	597,178	657,037	635,997
selection sort	150K	1388,378	1450,714	1495,079	1444,723
selection sort	200K	3509,411	3301,682	2468,138	3093,076



O algoritmo Selection Sort fluiu bem mais que o Bubble Sort e foi possível observar também que ele talvez se assemelhe ao Insertion Sort em questão de tempo de execução. Ele funcionou bem com os tamanhos de lista passados.

A diferença de tempo entre as 3 execuções na lista de 200K, teve mais divergência, foi um pouco maior que o algoritmo passado. Obtiveram mais variações do que o Bubble Sort, tendo outros níveis de tempo nos gráficos e tabelas. Assim como o Selection e no Insertion Sort, a lista que mais diferenciou de uma pra outra na hora de executar e ordenar foi de 10.000 para 100.000. As execuções fluíram bem e não tiveram dentro dos tempos, diferenças de uma execução para outra relacionada a um determinado tamanho de lista, o que confirma a eficiência do algoritmo.

Insertion Sort

Ordenação por inserção

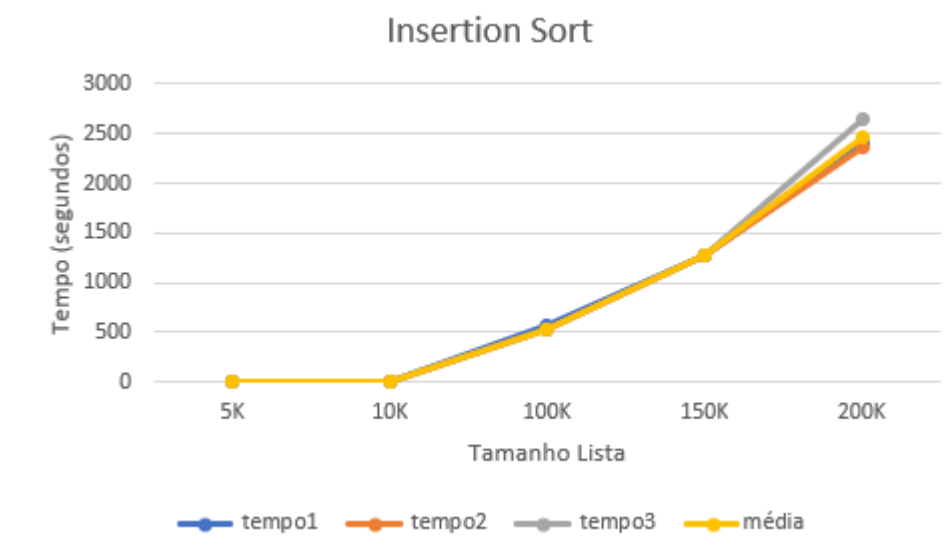
Este algoritmo é fácil de utilizar como o bubble e selection e é eficaz para listas que não possuem muitos elementos. Ele forma uma sequência de elementos ordenados, sendo um por vez, pegando determinado elemento e o inserindo em uma posição correta para que ocorra a ordenação de todos os elementos da lista.

```
#complexidade:  $O(n^2)$ 
#insertion Sort (ordenação por inserção)
def insertion_sort(lista):
    # percorre a lista começando do segundo elemento - índice 1
    for i in range(1, len(lista)):
        # pega o elemento atual para comparar - pivo
        pivo = lista[i]
        # define o índice anterior ao elemento atual
        j = i-1
        # move os elementos maiores que o pivo para uma posição a frente
        # até encontrar a posição correta para o pivo
        while j >= 0 and pivo < lista[j]:
            # move o elemento para frente
            lista[j+1] = lista[j]
            # move para o próximo elemento a esquerda
            j -= 1
        # insere o pivo na posição correta
        lista[j+1] = pivo
```

Com algoritmo Insertion Sort:

- Lista de 5.000 elementos
- Lista de 10.000 elementos
- Lista de 100.000 elementos
- Lista de 150.000 elementos
- Lista de 200.000 elementos

algoritmo	tamanho lista	tempo1	tempo2	tempo3	média
insertion sort	5K	1,254	1,191	1,097	1,18
insertion sort	10K	4,534	4,705	5,707	4,982
insertion sort	100K	567,948	521,88	527,577	539,135
insertion sort	150K	1276,233	1270,175	1274,578	1273,662
insertion sort	200K	2403,039	2354,811	2640,85	2466,23



O algoritmo Insertion Sort foi bem eficiente, mas ainda demorado dependendo do tamanho da lista, comparado com o Merge Sort.

Ao meu ver, ele é um algoritmo que teve resultados de execuções, tempo e média bem parecidos com o Selection Sort. Teve uma duração razoável para o parâmetro de lista passado e assim como já falei, no decorrer não tiveram muitas diferenças por conta de o valor dos tamanhos das listas talvez estar próximo. Assim como o Selection e no Insertion Sort, a lista que mais diferenciou de uma pra outras na hora de executar e ordenar foi de 10.000 para 100.000.

Podemos observar que ele foi bem mais próximo com os resultados do que o Selection Sort, os valores tiveram pouca variação, tanto de quantidade de execução, quanto de média.

Merge Sort

Ordenação por mistura

Dos 4 algoritmos, este é o mais eficiente para listas grandes. Com esse algoritmo a lista é dividida sempre em partes menores, ordenando essas partes e depois combinando elas (merge), fazendo assim a junção das partes ordenadas da lista para chegar à ordenação completa. Ele utiliza recursão, que consiste na parte de divisão e combinação dos elementos. Possui a estratégia de “dividir para conquistar”.

```
#complexidade: O(nlog2n)
#merge Sort (ordenação por mistura)
def merge_sort(lista):
    if len(lista)>1:
        #encontrando o meio da lista
        meio = len(lista) // 2 #parte inteira

        #fatiamento de listas
        L = lista[:meio]
        R = lista[meio:]

        #chamada recursiva
        merge_sort(L)
        merge_sort(R)

        #variaveis de controle
        # i - faz o controle da lista L
        # j - faz o controle da lista R
        # k - faz o controle da lista (anterior a chamada recursiva)

        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                lista[k] = L[i]
                i += 1
            else:
                lista[k] = R[j]
                j += 1
            k += 1

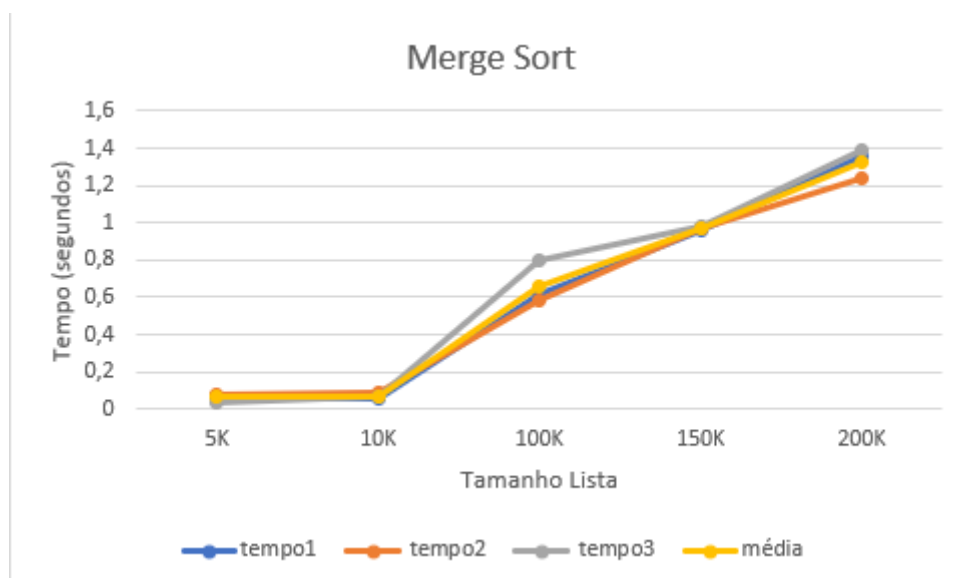
        #verificacao dos elementos da lista L
        while i < len(L):
            lista[k] = L[i]
            i += 1
            k += 1

        #verificacao dos elementos da lista R
        while j < len(R):
            lista[k] = R[j]
            j += 1
            k += 1
```


Com algoritmo Merge Sort:

- Lista de 5.000 elementos
- Lista de 10.000 elementos
- Lista de 100.000 elementos
- Lista de 150.000 elementos
- Lista de 200.000 elementos

algoritmo	tamanho lista	tempo1	tempo2	tempo3	média
merge sort	5K	0,073	0,081	0,037	0,063
merge sort	10K	0,056	0,084	0,067	0,069
merge sort	100K	0,612	0,579	0,802	0,664
merge sort	150K	0,963	0,97	0,978	0,973
merge sort	200K	1,353	1,236	1,386	1,325



O algoritmo Merge Sort foi o mais eficiente de todos e foi incrível ver que a mesma lista que para alguns algoritmos demoraram dias, horas e mais horas, para o Merge Sort foi em poucos segundos! A eficácia desse algoritmo é muito boa e traz uma melhor experiência para a ordenação e envolvimento com esses algoritmos. Mostra muito a "mágica" de ordenar listas de uma outra maneira, usando a recursividade.

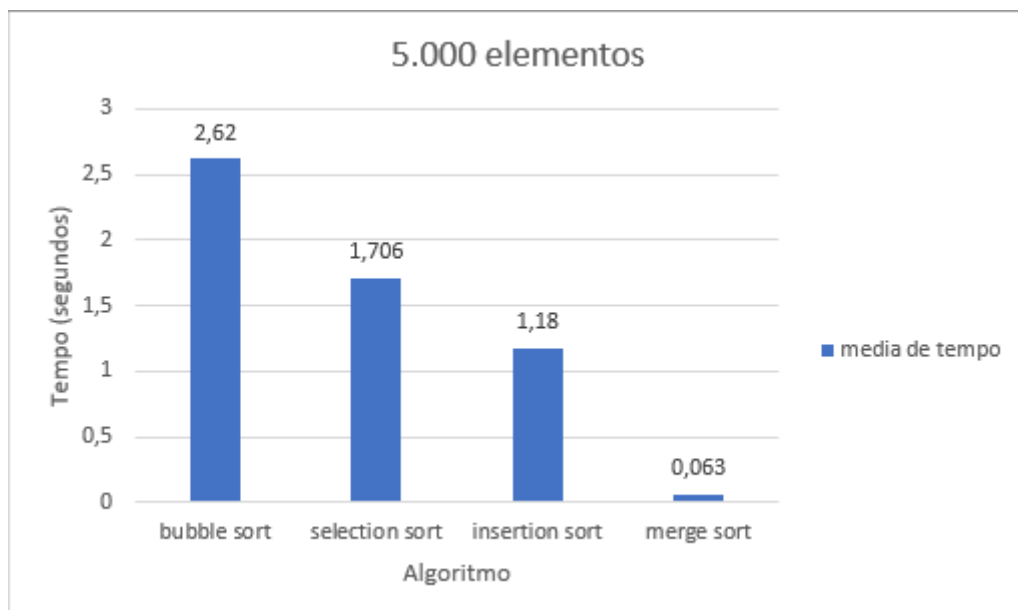
Não vi uma proximidade tão grande quanto o Insertion Sort entre as execuções e tamanhos de listas, mas o quão rápido executou esse algoritmo traz muita divergência em relação aos outros.

Com certeza esse algoritmo foi aquele que mais rendeu, em questão de rapidez e praticidade de tempo, em fazer toda a dinâmica com a troca de elementos da lista, com recursividade e ordená-la. Mesmo com uma lista que pro Bubble Sort estava demorada demais, para o Merge foi muito tranquilo e isso traz a eficiência do algoritmo como prova da sua lógica de ordenar, dos tempos gastos nas execuções e as médias.

Relação dos Algoritmos com base nas Listas

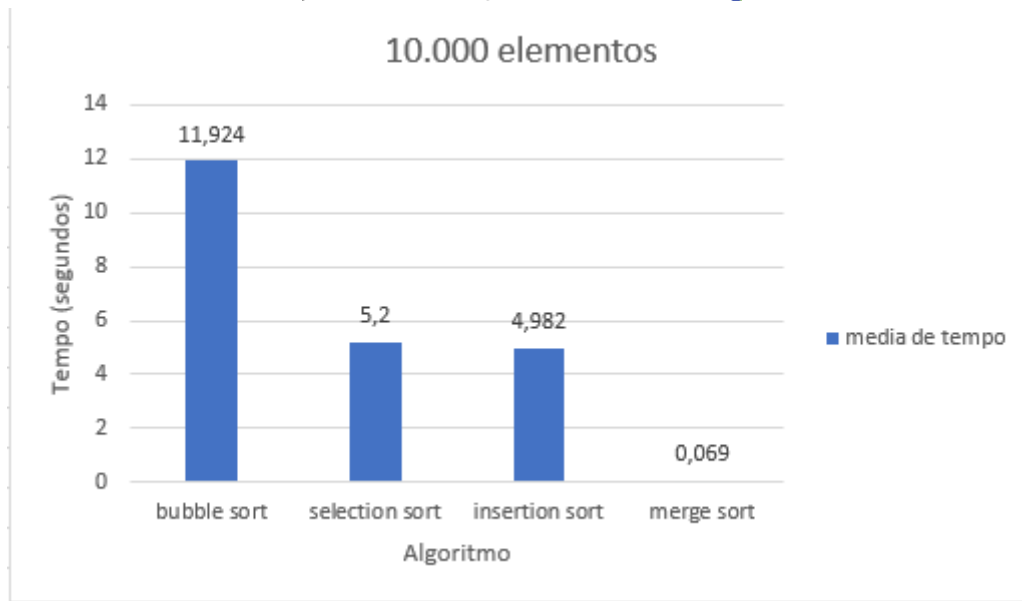
5.000 elementos

algoritmo	media de tempo
bubble sort	2,62
selection sort	1,706
insertion sort	1,18
merge sort	0,063



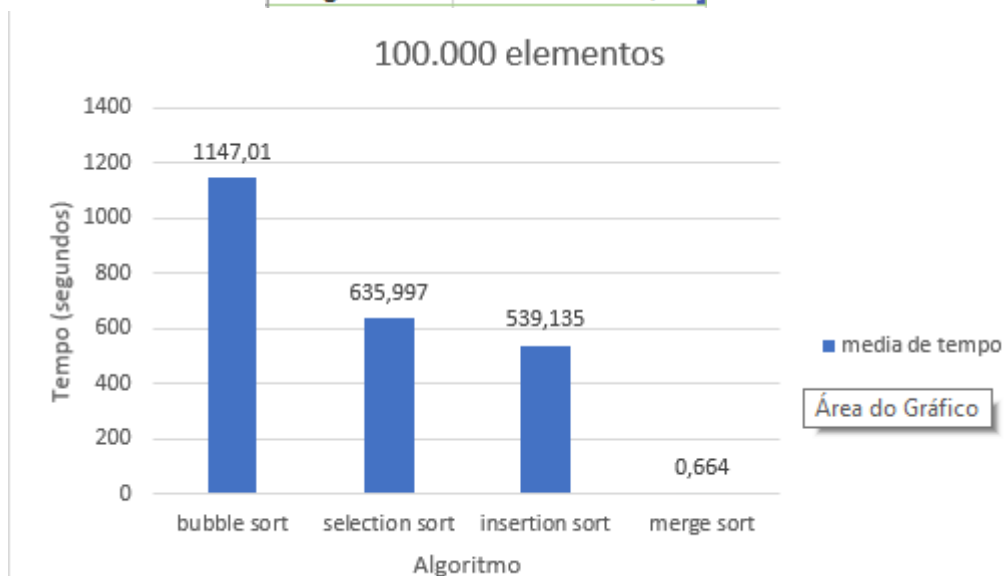
10.000 elementos

algoritmo	media de tempo
bubble sort	11,924
selection sort	5,2
insertion sort	4,982
merge sort	0,069



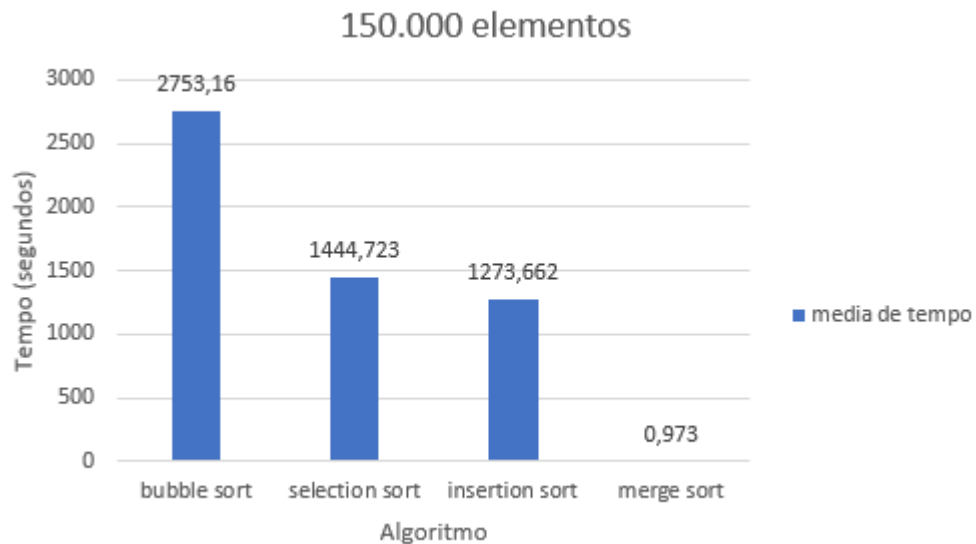
100.000 elementos

algoritmo	media de tempo
bubble sort	1147,01
selection sort	635,997
insertion sort	539,135
merge sort	0,664



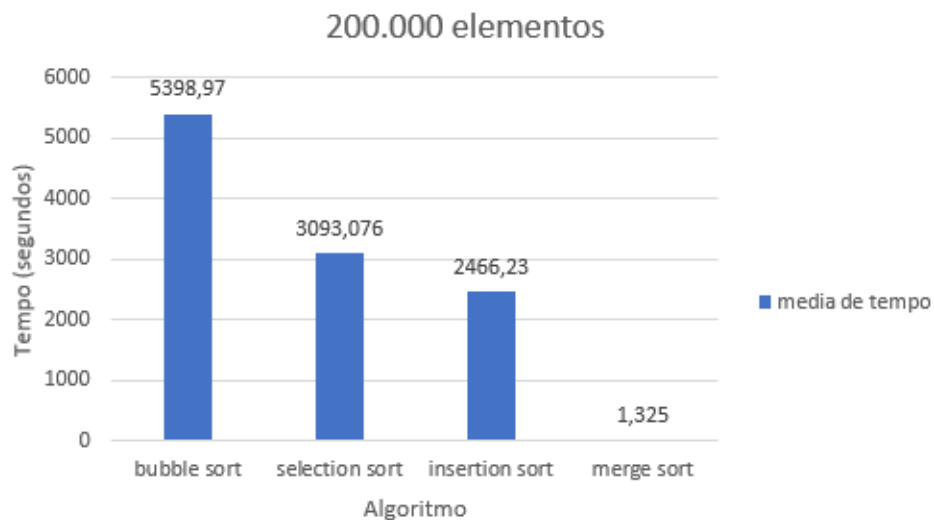
150.000 elementos

algoritmo	media de tempo
bubble sort	2753,16
selection sort	1444,723
insertion sort	1273,662
merge sort	0,973



200.000 elementos

algoritmo	media de tempo
bubble sort	5398,97
selection sort	3093,076
insertion sort	2466,23
merge sort	1,325



Conclusão

Com essa prova, foi possível desenvolver bem os conhecimentos e ir além daquilo que pudéssemos imaginar.

Um trabalho muito bem dedicado ao que temos aprendido nesse semestre na faculdade, na matéria de Python e esse tipo de prática traz uma ajuda muito grande não somente para a nossa vida acadêmica, mas para a vida profissional, lá fora.

No decorrer no desenvolvimento deste, pude perceber muito isso, que a forma como fomos atrás das informações e como nos dedicamos a desenvolver este projeto espelha naquilo que teremos como bagagem e é possível tirar isso de lição.

Com essa matéria e assunto abordado de algoritmos, algo que não é deixado de escanteio no mercado de trabalho, conseguimos desenvolver muito nosso aprendizado e a nossa forma de olhar para esse mundo. Foi bom aprofundar neste assunto de algoritmos de ordenação, com o intuito de desbravarmos do essencial da programação.