

Universidade Estadual do Maranhão (UEMA)  
Caixa Postal 09 – 65055-310 – São Luís – MA – Brazil

<sup>2</sup>Departamento de Engenharia de Computação  
Universidade Estadual do Maranhão (UEMA) – São Luís, SC – Brazil

Wesleson Souza Silva

## 1. Introdução

Este presente trabalho apresenta os teste com os piores e melhores casos de ordenação com 100, 1000, 10000, 1000000 e 10000000, a partir de um arquivo txt, onde a primeira linha é composta da quantidade de números que vão ser geradoras para serem ordenados e a segunda linha apresenta o nome do algoritmo que vai ser utilizado para o processo e implementação de alguns algoritmos de estrutura de dados como Bubble Sort, Insertion Sort, Shell Sort e Heapsort. Os dados como dia, mês, ano tempo de execução do algoritmo, caso, memória usada para o processo será salvo em um novo “**ARQUIVO.CSV**”.

## 2. Desenvolvimento

Esse projeto foi desenvolvido com a linguagem de programação Python, O primeiro passo dado no código foi a leitura de um arquivo .txt e o armazenamento da primeira linha para a criação das lista aleatórias que serão ordenadas.

```
10 | #Fazendo a leitura do arquivo txt
11 | f = open("file.txt", 'r')
12 | linha = int(f.readline())
13 |
14 |
```

Figura 1 - Parte do Código 1

Podemos ver acima que na linha 12 é usado o **int** antes do (**f.readline()**) para a primeira linha vim com inteiro e não como **string**.

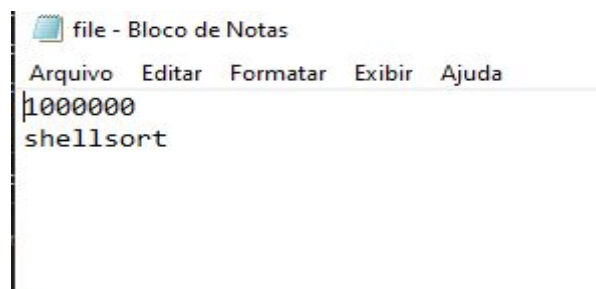


Figura 2 – Arquivo .txt utilizado

Após a leitura do arquivo.txt é feito a criação dos vetores que serão ordenados. Os vetores vão ser do tamanho determinado na linha 1 do txt, no código essa linha 1 é representada pela a variável linha do código abaixo.

```
14 |
15 | lista1=[]
16 | lista2=[]
17 | lista3=[]
18 | lista4=[]
19 | lista5=[]
20 | #lista melhor
21 | for i in range (linha):
22 |     lista1.append(random.randint(0,linha))
23 |
24 | #lista pior
25 | for i in range (linha):
26 |     lista2.append(random.randint(0,linha))
27 |
28 |
```

Figura 3 - Parte do Código 2

Em seguida será ilustrada a figura que apresenta **HeapSort** algoritmo de ordenação do melhor caso. O HeapSort Segundo Contribuidores da Wikipédia 2020 “O heapsort utiliza uma estrutura de dados chamada heap, para ordenar os elementos à medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada, lembrando-se sempre de manter a propriedade de max-heap.”

```
def heapify(self, array, n, i):
    #Seja i o índice de um dado elemento da heap. Podem
    # elementos a ele conectados (pai e filhos) através
    maior = i
    l = 2 * i + 1#direita
    r = 2 * i + 2#esquerda
    #Onde i é o inde atual, e o resulta da operação é o

    if l < n and array[i] < array[l]:
        maior = l

    if r < n and array[maior] < array[r]:
        maior = r

    if maior != i:
        array[i], array[maior] = array[maior], array[i]
        self.heapify(array, n, maior)
```

Figura 4 - Parte do Código 3

A próxima figura exibe a função do heapsort para o teste de pior caso, no pior caso a ordenação é do maior para o menor valor do vetor, nesse caso no trecho do código onde

ilustra o primeiro teste no caso o direito , a condição é posição [i] > posição [l] e da mesma forma é o esquerdo.

```
def heapify_pior(self, array, n, i):
    maior = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and array[i] > array[l]:
        maior = l

    if r < n and array[maior] > array[r]:
        maior = r

    if maior != i:
        array[i], array[maior] = array[maior], array[i]
        self.heapify_pior(array, n, maior)
```

Figura 5 - Parte do Código 4

Nessa parte do código é pego todas as informações da execução do código que estão na memória e salvo em um arquivo CSV.

```
with open('dados_gerados.csv', 'a', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["Data :", data, " Hora ", time, "HeapSort", "Tamanho do vetor", linha, " melhor caso ", Duracao, " memoria Ram ", me])
    #writer.writerow(["Data :", data1, " Hora ", time1, " bubblesort", "Tamanho do vetor", linha, " Melhor pior ", Duracao1, " memoria Ram "])
    file.close()
```

Figura 6 - Parte do Código 5

Dados Gerados HeapSort					
Data	Horário	Tamanho do vetor	Caso	Tempo de execução	Memória Ram
26/10/20	22:15:41	100	melhor	0:00:00.009801	3411316736
26/10/20	22:15:41	100	pior	0:00:00.016953	3411275776
26/10/20	22:15:50	1000	melhor	0:00:00.019907	3401121792
26/10/20	22:15:51	1000	pior	0:00:00.025759	3401105408
26/10/20	22:16:00	10000	melhor	0:00:00.207680	3416121344

26/10/20	22:16:00	10000	pior	0:00:00.196128	3413983232
26/10/20	22:16:13	100000	melhor	0:00:02.511545	3436462080
26/10/20	22:16:17	100000	pior	0:00:03.288515	3437129728
26/10/20	22:16:59	1000000	melhor	0:00:30.565256	3518124032
26/10/20	22:17:33	1000000	pior	0:00:34.242484	3518124032
26/10/20	22:25:10	10000000	melhor	0:06:12.262496	3313594368
26/10/20	22:31:22	10000000	pior	0:06:10.425706	3312525312

Tabela 01 - Dados Gerados 1

Nesta parte será apresentado o Insertion Sort o segundo algoritmo escolhido neste projeto. A ideia melhor caso por trás desse algoritmo é simplesmente percorra as posições do vetor iniciando com o índice 1. Em cada nova posição é como a nova carta que você recebeu, e será inserido no lugar correto no subvetor ordenado à esquerda daquela posição.

```
def InsertionSortM(lista1):
    # começamos o loop no segundo elemento
    for j in range(1, len(lista1)):
        key = lista1[j] #O próximo item que

        i = j-1 # o último item com o qual
        #agora continuamos movendo a chave p
        while (i > -1) and key < lista1[i]:
            lista1[i+1]=lista1[i] #move o ú
            i=i-1#observe o próximo item pa
        #okay i não é maior que key signific
        lista1[i+1] = key
```

Figura 7 - Parte do Código 6

Nesse trecho é feito a aplicação do pior caso do Insertion Sort, para esse caso foi feita a ordenação do maior para o menor. O trecho do código que define a ordenação do maior para o menor e a linha que faz o seguinte teste teste, `key < lista[i]`:

```
##pior caso Insertion sort###
def InsertionSortP(lista1):
    # começamos o loop no segundo elemento
    for j in range(1, len(lista1)):
        key = lista1[j] #O próximo item que

        i = j-1 # o último item com o qual
        #agora continuamos movendo a chave
        while (i > -1) and key > lista1[i]:
            lista1[i+1]=lista1[i] #move o ú
            i=i-1#observe o próximo item pa
        #okay i não é maior que key signific
        lista1[i+1] = key
```

Figura 8 - Parte do Código 7

Dados Gerados InsertionSort					
Data	Horário	Tamanho do vetor	Caso	Tempo de execução	Memória Ram
26/10/20	22:39:02	100	melhor	0:00:00.013596	3056443392
26/10/20	22:39:02	100	pior	0:00:00.096868	3056431104
26/10/20	22:39:14	1000	melhor	0:00:00.118419	3057414144
26/10/20	22:39:14	1000	pior	0:00:00.294460	3053477888
26/10/20	22:39:47	10000	melhor	0:00:10.644905	3070447616
26/10/20	22:39:58	10000	pior	0:00:21.620339	3072704512
26/10/20	23:02:27	100000	melhor	0:22:07.067587	3046301696
26/10/20	23:30:39	100000	pior	0:50:18.660156	3243298816
26/10/20		1000000	melhor		
26/10/20		1000000	pior		
26/10/20		10000000	melhor		
26/10/20		10000000	pior		

Tabela 02 - Dados Gerados 2

Iremos apresentar agora o algoritmo e resultados do Shellsort, a ordenação Shell funcionaliza da seguinte maneira, ele simplesmente faz uma quebra sucessiva da sequência a qual vai ser ordenada e implementa a ordenação por inserção na sequência obtida. Devido a sua complexidade possui excelentes desempenhos em N muito grandes, inclusive sendo melhor que o Merge Sort. A seguir é mostrado a ilustração o trecho de código do melhor caso, no melhor caso é feita a ordenação do menor para o maior índice do vetor. Esse o algoritmo passa várias vezes pelo vetor dividindo o grupo maior em menores. Nos grupos menores é aplicado o método da ordenação por inserção. Implementações do algoritmo.

```

342 def shellSort(nums):
343     h = 1
344     n = len(nums)
345     while h > 0:
346         for i in range(h, n):
347             c = nums[i]
348             j = i
349             while j >= h and c < nums[j - h]:
350                 nums[j] = nums[j - h]
351                 j = j - h
352                 nums[j] = c
353             h = int(h / 2.2)
354     return nums
355

```

Figura 9 - Parte do Código 8

Implementações do algoritmo do algoritmo no seu pior caso, no pior caso ele faz a ordenação do maior para o menor nesse caso a única parte que se é feita uma alteração o teste  $C > \text{NUM}[J - H]$ , para fazer a separação dos grupo e seguir com a inserção da ordenação

```

391 def shellSortP(nums):
392     h = 1
393     n = len(nums)
394     while h > 0:
395         for i in range(h, n):
396             c = nums[i]
397             j = i
398             while j >= h and c > nums[j - h]:
399                 nums[j] = nums[j - h]
400                 j = j - h
401                 nums[j] = c
402             h = int(h / 2.2)
403     return nums

```

Figura 10 - Parte do Código 9

Dados Gerados ShellSort					
Data	Horário	Tamanho do vetor	Caso	Tempo de execução	Memória Ram
27/10/20	12:39:41	100	melhor	0:00:00.059840	3158892544
27/10/20	12:39:41	100	pior	0:00:00.422444	3159003136
27/10/20	12:39:52	1000	melhor	0:00:00.162786	3159003136
27/10/20	12:39:52	1000	pior	0:00:00.403350	3185442816
27/10/20	12:39:52	10000	melhor	0:00:16.445437	3189059584
27/10/20	12:40:20	10000	pior	0:00:33.374106	3178901504
27/10/20	12:40:37	100000	melhor	0:33:23.386724	3178901504
27/10/20	12:40:37	100000	pior	0:33:23.386724	3178901504
27/10/20	13:14:13	1000000	melhor	0:33:23.386724	3190607872
27/10/20	13:46:07	1000000	pior	1:05:16.658868	2995814400

Tabela 03 - Dados Gerados 3

Vamos apresentar agora o algoritmo de ordenação bubbleSort esse algoritmo percorre a lista de itens ordenáveis do início ao fim, verificando a ordem dos elementos dois a dois, e trocando-os de lugar se necessário. Percorre-se a lista até que nenhum elemento tenha sido trocado de lugar na passagem anterior. No melhor caso é feita a ordenação do menor para o maior.



```

183 def bubbleSortmelhor(alist):
184     for passnum in range(len(alist)-1,0,-1):
185         for i in range(passnum):
186             if alist[i]>alist[i+1]:
187                 #irá trocar os itens i e j da
188                 temp = alist[i]
189                 alist[i] = alist[i+1]
190                 alist[i+1] = temp
191

```

Figura 11 - Parte do Código 10

No Pior caso do bubble sort é feito apenas a ordenação do maior para o menor para a geração dos dados.

```

125 def salvarbubblesortP():
126
127     def bubbleSortpior(alist):
128         for passnum in range(len(alist)-1,0,-1):
129             for i in range(passnum):
130                 #verificar se os elementos estão
131                 if alist[i]<alist[i+1]:
132                     #irá trocar os itens i e j da
133                     temp = alist[i]
134                     alist[i] = alist[i+1]
135                     alist[i+1] = temp
136

```

Figura 12 - Parte do Código 11

Dados Gerados BubbleSort					
Data	Horario	Tamanho do vetor	Caso	Tempo de execução	Memória Ram
28/10/20	18:32:30	100	melhor	0:00:00.009972	3588268032
28/10/20	18:32:31	100	pior	0:00:01.820118	3589324800
28/10/20	18:32:55	1000	melhor	0:00:00.013961	3572269056
28/10/20	18:32:56	1000	pior	0:00:00.454995	3572269056
28/10/20	18:32:56	10000	melhor	0:00:00.069973	3598790656
28/10/20	18:33:14	10000	pior	0:00:35.397901	3564732416
28/10/20	18:33:14	100000	melhor		
28/10/20	18:33:14	100000	pior	0:49:32.082646	3435880448

Tabela 04 - Dados Gerados 4

## Conclusão

Para a execução desse projeto é preciso que na segunda linha o algoritmo escolhido esteja escrito com “insertion”, “heapsort”, “shellsort”, ou “buble” para que seja feito o teste de qual algoritmo vai ser executado no projeto.

```
meuArquivo = open('file.txt','r')
tlinhas = (meuArquivo.readlines())

a="insertion"
b="heapsort"
c="shellsort"
d="bublle"
```

Figura 13 - Parte do Código 12

## Referências

Contribuidores da Wikipédia, "Heapsort," *Wikipédia, a enciclopédia livre*, <https://pt.wikipedia.org/w/index.php?title=Heapsort&oldid=59158436> (accessed agosto 25, 2020).

AZEREDO, Paulo A. (1996). *Métodos de Classificação de Dados e Análise de suas Complexidades*. Rio de Janeiro: Campus

WIRTH, Niklaus (1989). *Algoritmos e Estruturas de Dados*. Rio de Janeiro: Prentice-Hall do Brasil.

Veloso, Paulo; SANTOS, Clesio dos; AZEREDO, Paulo; FURTADO, Antonio (1986). *Estruturas de Dados* 4ª ed. Rio de Janeiro: Campus.

Bubble sort. (2019, setembro 30). *Wikipédia, a enciclopédia livre*. Retrieved 01:18, setembro 30, 2019 from [https://pt.wikipedia.org/w/index.php?title=Bubble\\_sort&oldid=56363566](https://pt.wikipedia.org/w/index.php?title=Bubble_sort&oldid=56363566).