Relatório Projeto 3 AED 2023/2024

Nome: Miguel Teixeira de Pina Monteiro Pereira Nº Estudante: 2022232552
PL (inscrição): PL4 Email: miguelmpereira0409@gmail.com

IMPORTANTE:

- As conclusões devem ser manuscritas... texto que não obedeça a este requisito não é considerado.
- Texto para além das linhas reservadas, ou que não seja legível para um leitor comum, não é considerado.
- O relatório deve ser submetido num único PDF que deve incluir os anexos. A não observância deste formato é penalizada.

1. Planeamento

	Semana 1	Semana 2	Semana 3	Semana 4	Semana 5
Insertion Sort		x			
Heap Sort			x		
Quick Sort				x	
Finalização Relatório				x	x

2. Recolha de Resultados *(tabelas)*

Insertion Sort

Nº de elementos	Conjunto A	Conjunto B	Conjunto C
10000	0,00235486	9,705354929	4,947041035
20000	0,004858971	39,8543539	20,08842707
30000	0,007869959	94,493402	43,66070294
40000	0,010300875	159,8595679	81,47531104
50000	0,011536121	249,3629518	129,475508
60000	0,015351772	345,0888908	188,0035539
70000	0,018368959	481,6163421	188,0035539
80000	0,019521952	620,3606141	406,8946249
90000	0,022445202	820,918988	514,1060169
100000	0,024760008	993,317606	600,2640259

Heap Sort

Nº elementos	Conjunto A	Conjunto B	Conjunto C
10000	0,090102196	0,075163841	0,081778049
20000	0,202577829	0,160340071	0,199109793
30000	0,283567905	0,254032135	0,277602911
40000	0,403826952	0,362287283	0,388051033
50000	0,491112947	0,446171999	0,504893064
60000	0,599418163	0,555109262	0,625593185
70000	0,707826853	0,648571014	0,737818241
80000	0,815196037	0,74855876	0,860481977
90000	0,933841944	0,862257004	1,012173176
100000	1,06458807	0,953294992	1,127898693

Quicksort

Nº elementos	Conjunto A	Conjunto B	Conjunto C
10000	0,0259881019	0,036128998	0,031386852
20000	0,0538609027	0,077115059	0,066071749
30000	0,0818250179	0,124202013	0,10256505
40000	0,1137640476	0,170887232	0,136456251
50000	0,1555750370	0,20638895	0,191430807
60000	0,1812579631	0,258950949	0,225508213
70000	0.2067010402	0,302306175	0,243608236
80000	0.2363388538	0,350661039	0,334372997
90000	0.2630991935	0,389600992	0,375334263
100000	0.2946121692	0,45359683	0,429516077

Conjuntos Ascendentes

Nº elementos	Quicksort	HeapSort	Insertion Sort
10000	0,0259881019	0,090102196	0,00235486
20000	0,0538609027	0,202577829	0,004858971
30000	0,0818250179	0,283567905	0,007869959
40000	0,1137640476	0,403826952	0,010300875
50000	0,1555750370	0,491112947	0,011536121
60000	0,1812579631	0,599418163	0,015351772
70000	0.2067010402	0,707826853	0,018368959
80000	0.2363388538	0,815196037	0,019521952
90000	0.2630991935	0,933841944	0,022445202
100000	0.2946121692	1,06458807	0,024760008

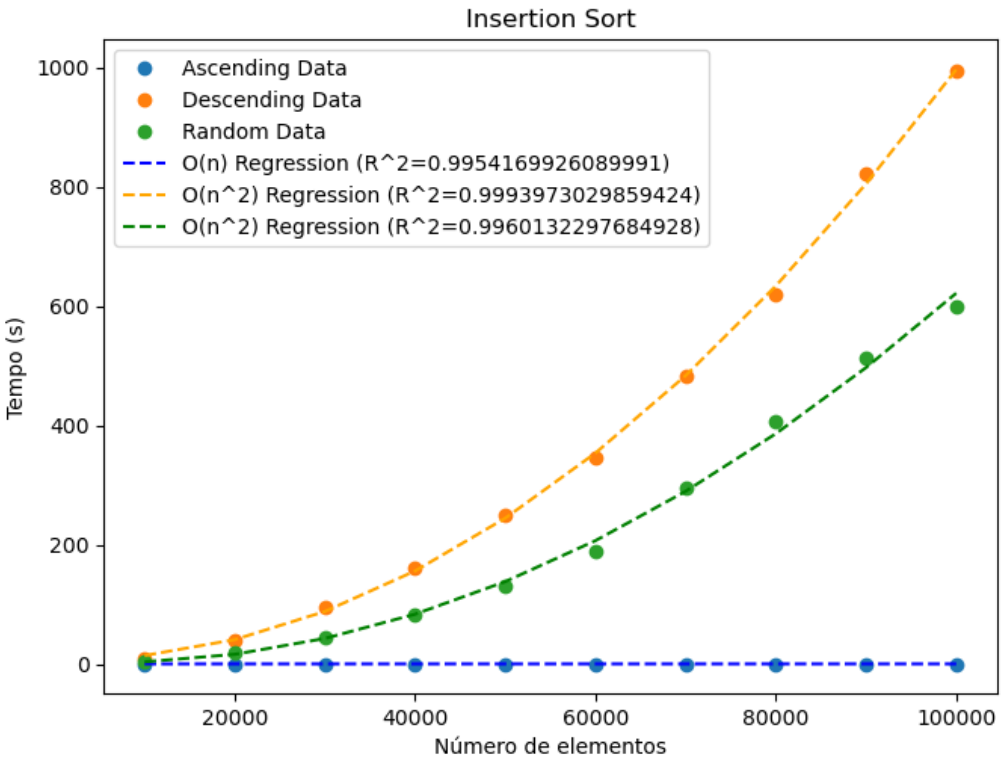
Conjuntos Descendentes

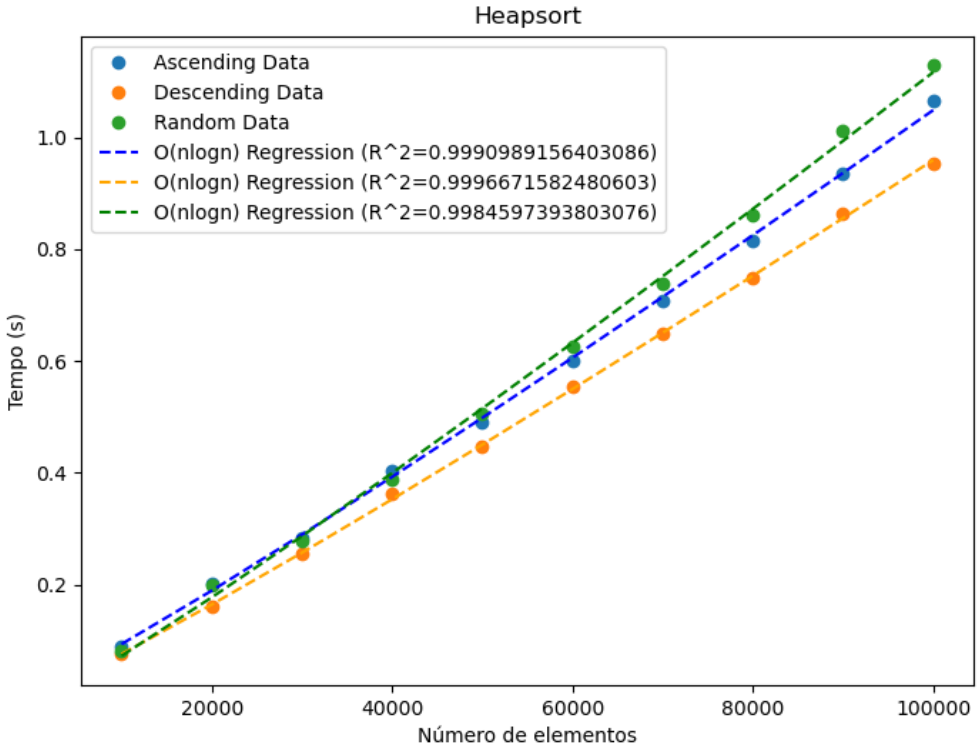
Nº elementos	QuickSort	HeapSort	Insertion Sort
10000	0,036128998	0,075163841	9,705354929
20000	0,077115059	0,160340071	39,8543539
30000	0,124202013	0,254032135	94,493402
40000	0,170887232	0,362287283	159,8595679
50000	0,20638895	0,446171999	249,3629518
60000	0,258950949	0,555109262	345,0888908
70000	0,302306175	0,648571014	481,6163421
80000	0,350661039	0,74855876	620,3606141
90000	0,389600992	0,862257004	820,918988
100000	0,45359683	0,953294992	993,317606

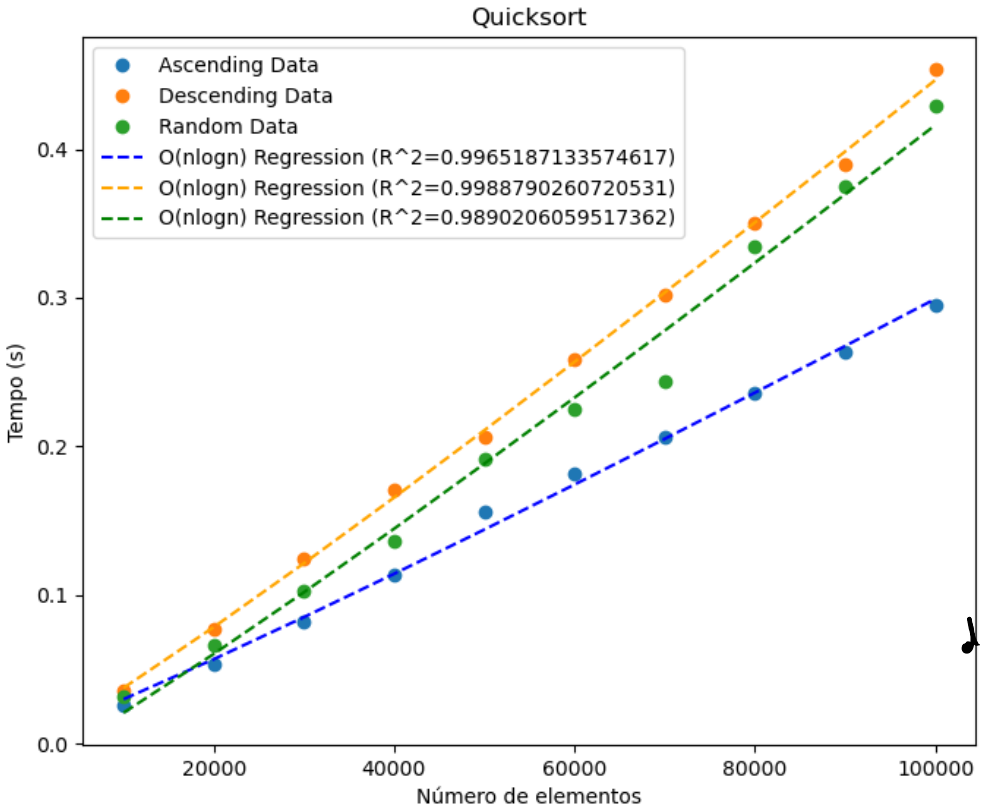
Conjuntos Aleatórios

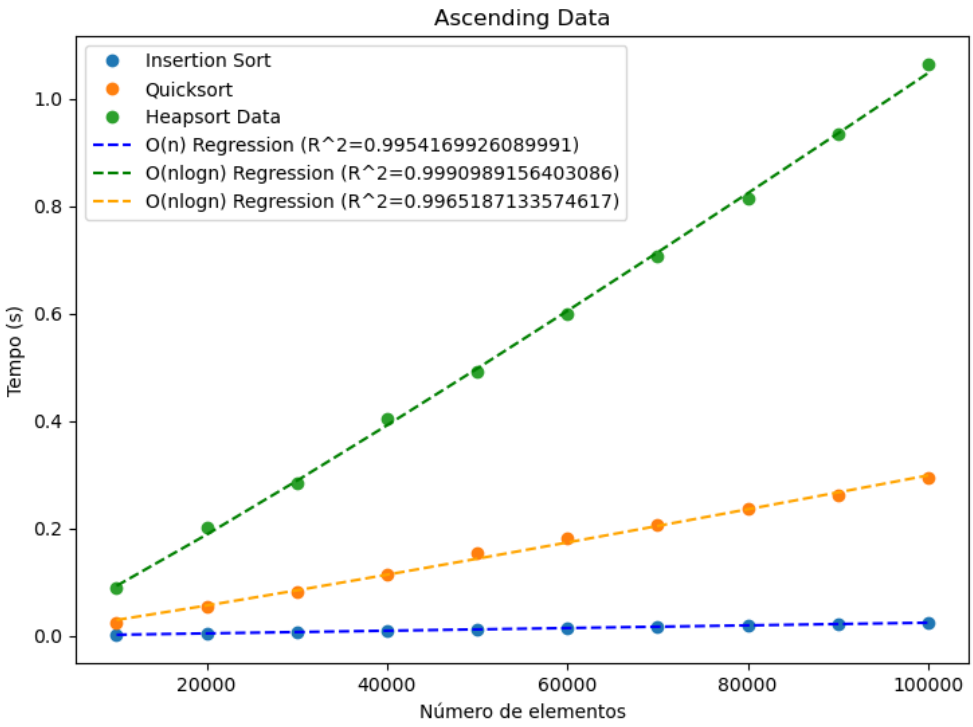
Nº elementos	QuickSort	HeapSort	Insertion Sort
10000	0,031386852	0,081778049	4,947041035
20000	0,066071749	0,199109793	20,08842707
30000	0,10256505	0,277602911	43,66070294
40000	0,136456251	0,388051033	81,47531104
50000	0,191430807	0,504893064	129,475508
60000	0,225508213	0,625593185	188,0035539
70000	0,243608236	0,737818241	294.9746460914612
80000	0,334372997	0,860481977	406,8946249
90000	0,375334263	1,012173176	514,1060169
100000	0,429516077	1,127898693	600,2640259

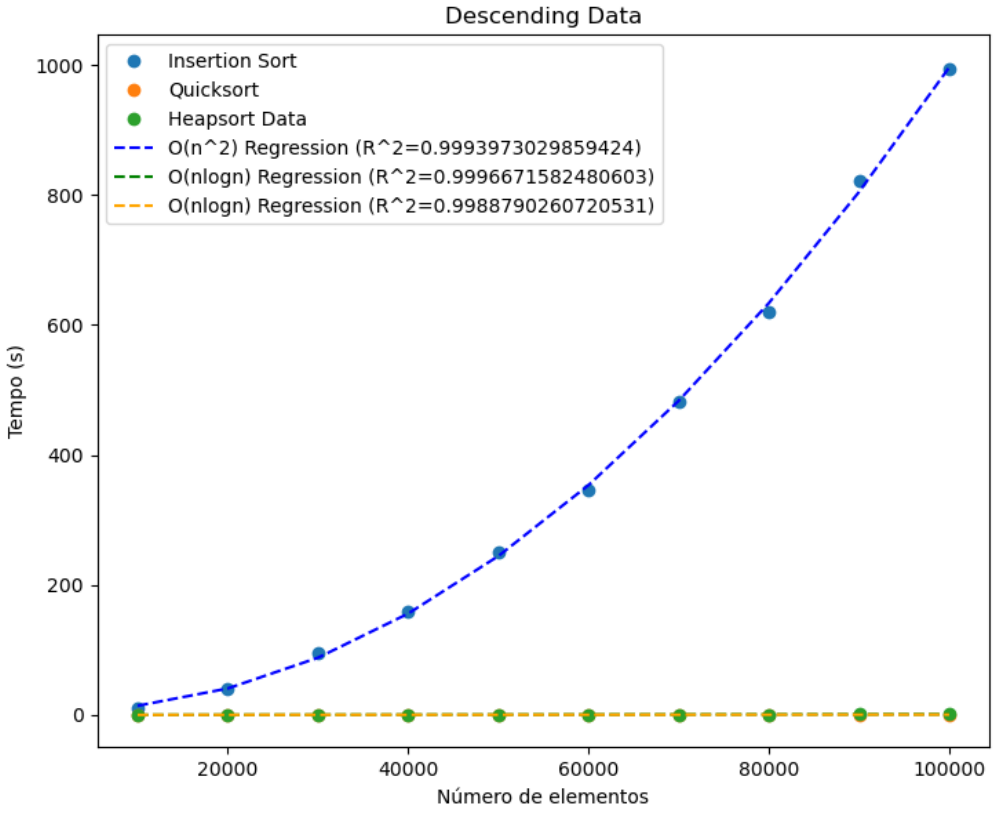
3. Visualização de Resultados *(gráficos)*

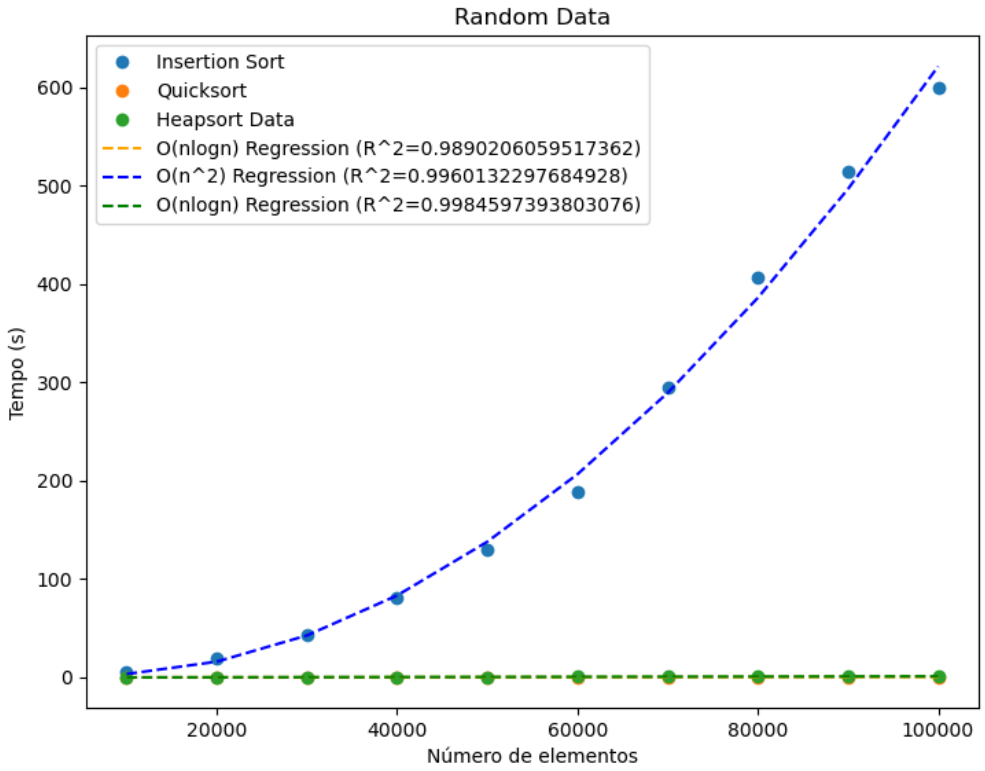

Insertion Sort
Ascending Data
Descending Data
Random Data
O(n) Regression (R^2=0.9954169926089991)
O(n^2) Regression (R^2=0.9993973029859424)
O(n^2) Regression (R^2=0.9960132297684928)
Tempo (s)
Número de elementos
0
200
400
600
800
1000
20000
40000
60000
80000
100000


Heapsort
Ascending Data
Descending Data
Random Data
O(nlogn) Regression (R^2=0.9990989156403086)
O(nlogn) Regression (R^2=0.9996671582480603)
O(nlogn) Regression (R^2=0.9984597393803076)
Tempo (s)
Número de elementos
0.2
0.4
0.6
0.8
1.0
20000
40000
60000
80000
100000


Quicksort
Ascending Data
Descending Data
Random Data
O(nlogn) Regression (R^2=0.9965187133574617)
O(nlogn) Regression (R^2=0.9988790260720531)
O(nlogn) Regression (R^2=0.9890206059517362)
Tempo (s)
0.0
0.1
0.2
0.3
0.4
20000
40000
60000
80000
100000
Número de elementos


Ascending Data
Insertion Sort
Quicksort
Heapsort Data
O(n) Regression (R^2=0.9954169926089991)
O(nlogn) Regression (R^2=0.9990989156403086)
O(nlogn) Regression (R^2=0.9965187133574617)
Tempo (s)
Número de elementos
0.0
0.2
0.4
0.6
0.8
1.0
20000
40000
60000
80000
100000


Descending Data
Insertion Sort
Quicksort
Heapsort Data
O(n^2) Regression (R^2=0.9993973029859424)
O(nlogn) Regression (R^2=0.9996671582480603)
O(nlogn) Regression (R^2=0.9988790260720531)
Tempo (s)
Número de elementos
0
200
400
600
800
1000
20000
40000
60000
80000
100000



4. Conclusões *(as linhas desenhadas representam a extensão máxima de texto manuscrito)*

4.1 Tarefa 1

A complexidade do insertion sort para arrays ordenados por ordem crescente é O(n), o seu best case, sendo a complexidade do average case, quando os elementos encontram-se ordenados aleatoriamente, e o seu worst case, quando encontram-se ordenados em ordem decrescente, $O(n^2)$, comprovado pelos resultados obtidos. É um algoritmo estável, mantém a ordem relativa entre os elementos iguais, e também é "in-place", não necessita de espaço adicional além do que já está ocupado pelos próprios dados O(1). É muito eficiente para conjuntos pequenos e ordenados, sendo pouco eficiente para conjuntos grandes e desordenados (aleatoriamente ou decrescentemente).

4.2 Tarefa 2

A complexidade do Heap Sort é $O(n \log())$ para todos os casos, comprovado pelos resultados obtidos. Deve-se ao facto de que em cada caso realizamos operações na Heap Tree, max ou min. Não é um algoritmo estável, não mantém a ordem relativa entre elementos iguais, mas é "in-place", não necessita de espaço adicional além do que já está ocupado pelos próprios dados $O(1)$. A nível de performance mostra-se inferior ao Quick Sort.

4.3 Tarefa 3

Para todos os casos, o Quick Sort tem uma complexidade de $O(n \log(n))$, comprovado pelos resultados obtidos. Pode ir até $O(n^2)$, o seu worst case, quando a escolha do pivot é mal efetuada. Não é um algoritmo estável, não mantém a ordem relativa entre elementos iguais, apesar de ser "in-place", não necessita de espaço adicional além do que já está ocupado pelos próprios dados $O(1)$. É o melhor algoritmo para ordenamento em arrays com conjunto de elementos em ordem aleatória e decrescente, sendo ótimo para arrays grandes e elementos aleatórios. A nível de performance é superior ao Heap Sort.

Anexo A - Delimitação de Código de Autor

```
##################################################################
##########################################
##################################################      Heap      Sort
##################################################
##################################################################
##########################################
```

```
def heapify(array, N, i ):
largest = i #Inicializa o maior como raiz
left = 2*i +1
right = 2*i +2

# Se o left child for maior que a raiz
# até agora
if left < N and array[left] > array[largest]:
largest = left
# Se o right child for maior que a raiz
#a até agora
if right < N and array[right] > array[largest]:
largest = right

#Se o maior nao for a raiz
if largest != i:
array[i],array[largest] = array[largest],array[i] #Swap

#Fazer heapify recursivamente a
# sub-tree afectada
heapify(array,N, largest)

def heapSort(array):
N = len(array)

#Construir a maxheap
for i in range(N//2 -1, -1, -1):
heapify(array, N, i)

#Extrair elementos um por um
for i in range(N-1, 0, -1):
array[i],array[0] = array[0],array[i] # Swap
heapify(array, i, 0)
```

Anexo B - Referências

https://www.geeksforgeeks.org/insertion-sort/

https://www.geeksforgeeks.org/heap-sort/
https://www.geeksforgeeks.org/quick-sort/

Anexo C – Listagem Código

```
import math
import statistics
import time
import sys
import random

sys.setrecursionlimit(10**6)

def generate_ascending_array(max_value):
return list(range(max_value + 1))

def generate_descending_array(max_value):
return list(range(max_value, -1, -1))

def generate_random_array(size, min_value, max_value):
return [random.randint(min_value, max_value) for _ in range(size)]



##################################################################
##########################################
##############################################        Insertion        Sort
##############################################
##################################################################
##########################################

def insertSort(array):

n = len(array)

for i in range(1,n):
temp = array[i]
j = i-1

while j>=0 and array[j]>temp:
array[j+1] = array[j]
```

```
j= j-1
array[j+1] = temp

##################################################################
##########################################
##############################################          Quick          Sort
################################################
##################################################################
##########################################


def quickSort(array, start, end):
#Verifica se o indíce start é menor que o índice end
if(start < end):
# Obtém o pivot_location usando a função partition
pivot_location = partition(array,start,end)
#Chama recursivamente quickSort para a partição da direita
quickSort(array,pivot_location+1,end)
#Chama recursivamente quickSort para a partição da esquerda
quickSort(array,start,pivot_location-1)
def partition(array, start, end):
#inicia o índice do pivot
i = start
#Seleciona o elemento pivot (neste caso o valor mediano do valor inicial, central e final do
array)
pivot = medianThree(array[start], array[start + (end - start) // 2], array[end])

#Itera pelo array do start ao end-1
for j in range(start, end):
#Se o actual elemento é menor ou igual ao pivot
if array[j] <= pivot:
#Troca o elemento actual com o elemento no índice i
array[i], array[j] = array[j], array[i]
#Move o índice pivot para a direita
i += 1

#Troca o elemento pivot com o elemento no índice i
array[i], array[end] = array[end] ,array[i]
#Devolve o índice pivot
return i

def medianThree( a, b, c):
return sorted([a, b, c])[1]

def getTimeQuickSortAscending(maxvalue):
```

```
lista = generate_ascending_array(maxvalue)
start_time = time.time()
quickSort(lista,0,maxvalue)
end_time = time.time()
elapsed_time = end_time - start_time
print(f'{elapsed_time};')


def getTimeQuickSortDescending(maxvalue):

lista = generate_descending_array(maxvalue)
start_time = time.time()
quickSort(lista,0,maxvalue)
end_time = time.time()
elapsed_time = end_time - start_time
print(f'{elapsed_time};')



def getTimeQuickSortRandom(maxvalue):

lista =generate_random_array(maxvalue,0,1000000)
start_time = time.time()
quickSort(lista,0,maxvalue)
end_time = time.time()
elapsed_time = end_time - start_time
print(f'{elapsed_time};')

def getTimeHeapSortAscending(maxvalue):
array = generate_ascending_array(maxvalue)
start_time = time.time()
heapSort(array)
end_time = time.time()
elapsed_time = end_time - start_time
print(f'{elapsed_time};')

def getTimeHeapSortDescending(maxvalue):
array = generate_descending_array(maxvalue)
start_time = time.time()
heapSort(array)
end_time = time.time()
elapsed_time = end_time - start_time
print(f'{elapsed_time};')
```

```
def getTimeHeapSortRandom(maxvalue):
array = generate_random_array(maxvalue, 0, 1000000)
start_time = time.time()
heapSort(array)
end_time = time.time()
elapsed_time = end_time - start_time
print(f'{elapsed_time};')


def getTimeInsertSortAscending(maxvalue):
array = generate_ascending_array(maxvalue)
start_time = time.time()
insertSort(array)
end_time = time.time()
elapsed_time = end_time - start_time
print(f'{elapsed_time};')

def getTimeInsertSortDescending(maxvalue):
array = generate_descending_array(maxvalue)
start_time = time.time()
insertSort(array)
end_time = time.time()
elapsed_time = end_time - start_time
print(f'{elapsed_time};')

def getTimeInsertSortRandom(maxvalue):
array = generate_random_array(maxvalue, 0, 1000000)
start_time = time.time()
insertSort(array)
end_time = time.time()
elapsed_time = end_time - start_time
print(f'{elapsed_time};')


##################################################################
##########################################
##################################################  Main
##################################################
##################################################################
##########################################
def main():
print("QuickSort Ascending")
getTimeQuickSortAscending(10000)
getTimeQuickSortAscending(20000)
getTimeQuickSortAscending(30000)
```

```
getTimeQuickSortAscending(40000)
getTimeQuickSortAscending(50000)
getTimeQuickSortAscending(60000)
getTimeQuickSortAscending(70000)
getTimeQuickSortAscending(80000)
getTimeQuickSortAscending(90000)
getTimeQuickSortAscending(100000)

print("\n\n QuickSort Descending")

getTimeQuickSortDescending(10000)
getTimeQuickSortDescending(20000)
getTimeQuickSortDescending(30000)
getTimeQuickSortDescending(40000)
getTimeQuickSortDescending(50000)
getTimeQuickSortDescending(60000)
getTimeQuickSortDescending(70000)
getTimeQuickSortDescending(80000)
getTimeQuickSortDescending(90000)
getTimeQuickSortDescending(100000)

print("\n\n")



getTimeQuickSortRandom(10000)
getTimeQuickSortRandom(20000)
getTimeQuickSortRandom(30000)
getTimeQuickSortRandom(40000)
getTimeQuickSortRandom(50000)
getTimeQuickSortRandom(60000)
getTimeQuickSortRandom(70000)
getTimeQuickSortRandom(80000)
getTimeQuickSortRandom(90000)
getTimeQuickSortRandom(100000)









getTimeHeapSortAscending(10000)
getTimeHeapSortAscending(20000)
getTimeHeapSortAscending(30000)
getTimeHeapSortAscending(40000)
getTimeHeapSortAscending(50000)
getTimeHeapSortAscending(60000)
```

```
getTimeHeapSortAscending(70000)
getTimeHeapSortAscending(80000)
getTimeHeapSortAscending(90000)
getTimeHeapSortAscending(100000)

getTimeHeapSortDescending(10000)
getTimeHeapSortDescending(20000)
getTimeHeapSortDescending(30000)
getTimeHeapSortDescending(40000)
getTimeHeapSortDescending(50000)
getTimeHeapSortDescending(60000)
getTimeHeapSortDescending(70000)
getTimeHeapSortDescending(80000)
getTimeHeapSortDescending(90000)
getTimeHeapSortDescending(100000)
getTimeHeapSortRandom(10000)
getTimeHeapSortRandom(20000)
getTimeHeapSortRandom(30000)
getTimeHeapSortRandom(40000)
getTimeHeapSortRandom(50000)
getTimeHeapSortRandom(60000)
getTimeHeapSortRandom(70000)
getTimeHeapSortRandom(80000)
getTimeHeapSortRandom(90000)
getTimeHeapSortRandom(100000)


getTimeInsertSortAscending(10000)
getTimeInsertSortAscending(20000)
getTimeInsertSortAscending(30000)
getTimeInsertSortAscending(40000)
getTimeInsertSortAscending(50000)
getTimeInsertSortAscending(60000)
getTimeInsertSortAscending(70000)
getTimeInsertSortAscending(80000)
getTimeInsertSortAscending(90000)
getTimeInsertSortAscending(100000)


getTimeInsertSortDescending(10000)
getTimeInsertSortDescending(20000)
getTimeInsertSortDescending(30000)
getTimeInsertSortDescending(40000)
getTimeInsertSortDescending(50000)
getTimeInsertSortDescending(60000)
```

```
getTimeInsertSortDescending(70000)
getTimeInsertSortDescending(80000)
getTimeInsertSortDescending(90000)
getTimeInsertSortDescending(100000)

getTimeInsertSortRandom(10000)
getTimeInsertSortRandom(20000)
getTimeInsertSortRandom(30000)
getTimeInsertSortRandom(40000)
getTimeInsertSortRandom(50000)
getTimeInsertSortRandom(60000)
getTimeInsertSortRandom(70000)
getTimeInsertSortRandom(80000)
getTimeInsertSortRandom(90000)
getTimeInsertSortRandom(100000)



if __name__ == '__main__':
main()
```