Relatório Projeto 3 AED 2023/2024

Nome: Eduardo Luís Pereira Marques
Nº Estudante: 2022231584
PL (inscrição): PL1
Email: eduardo.marques07g@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.

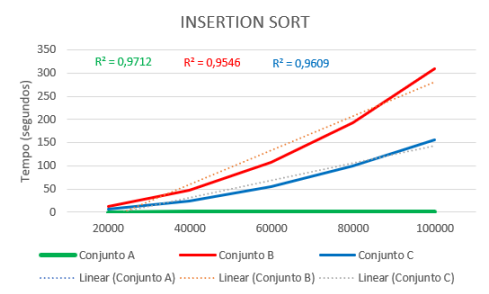
1. Planeamento

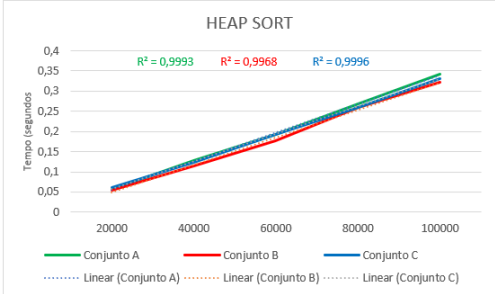
	Semana 1	Semana 2	Semana 3	Semana 4	Semana 5
Insertion Sort		x			
Heap Sort		x	x		
Quick Sort			x	x	
Finalização Relatório					x

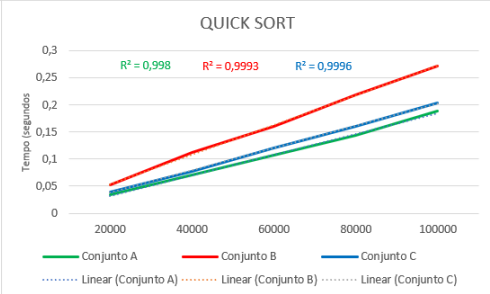
2. Recolha de Resultados

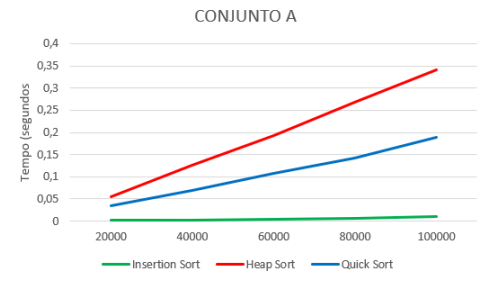
	INSERTION SORT				
	20000	40000	60000	80000	100000
Conjunto A	0,002	0,003	0,00502	0,007	0,01
Conjunto B	12,37624	47,50025	106,79769	191,86756	309,18119
Conjunto C	6,03343	23,83203	55,07442	100,43036	156,0761
	HEAP SORT				
	20000	40000	60000	80000	100000
Conjunto A	0,056	0,127	0,19211	0,26851	0,34191
Conjunto B	0,0539	0,11409	0,17629	0,25937	0,32204
Conjunto C	0,061	0,12435	0,19189	0,26011	0,33145
	QUICK SORT				
	20000	40000	60000	80000	100000
Conjunto A	0,0339	0,07	0,107	0,143	0,188
Conjunto B	0,05251	0,11185	0,161	0,21929	0,27235
Conjunto C	0,03911	0,078	0,12085	0,16011	0,20442

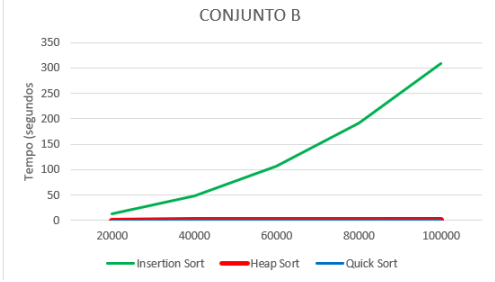
3. Visualização de Resultados

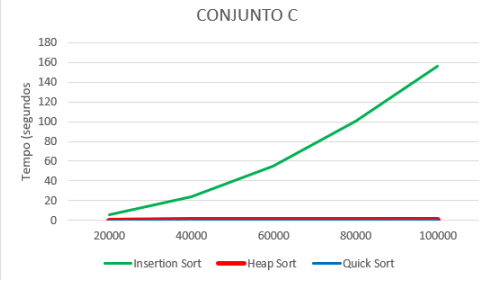










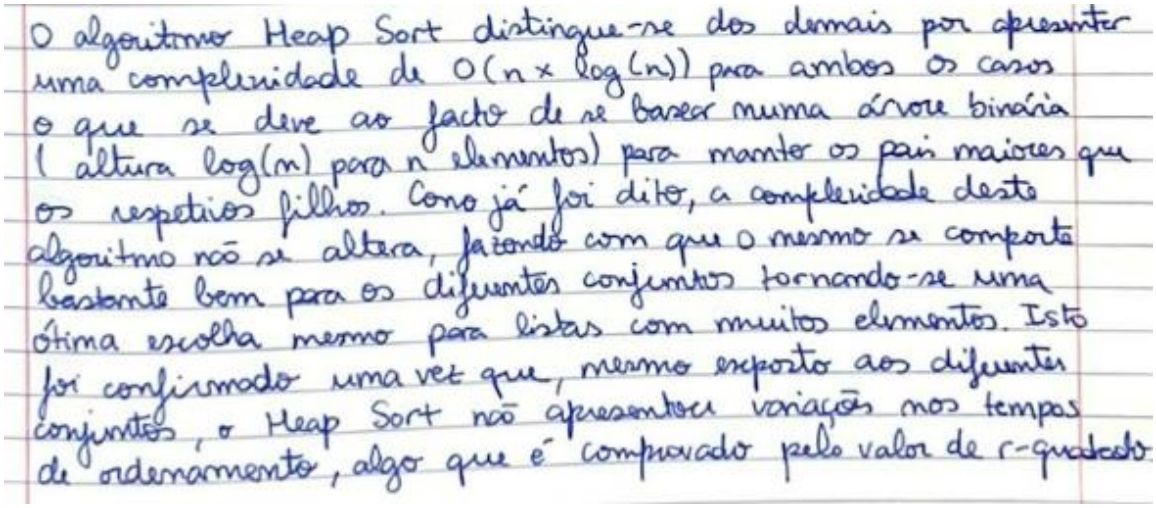


4. Conclusões

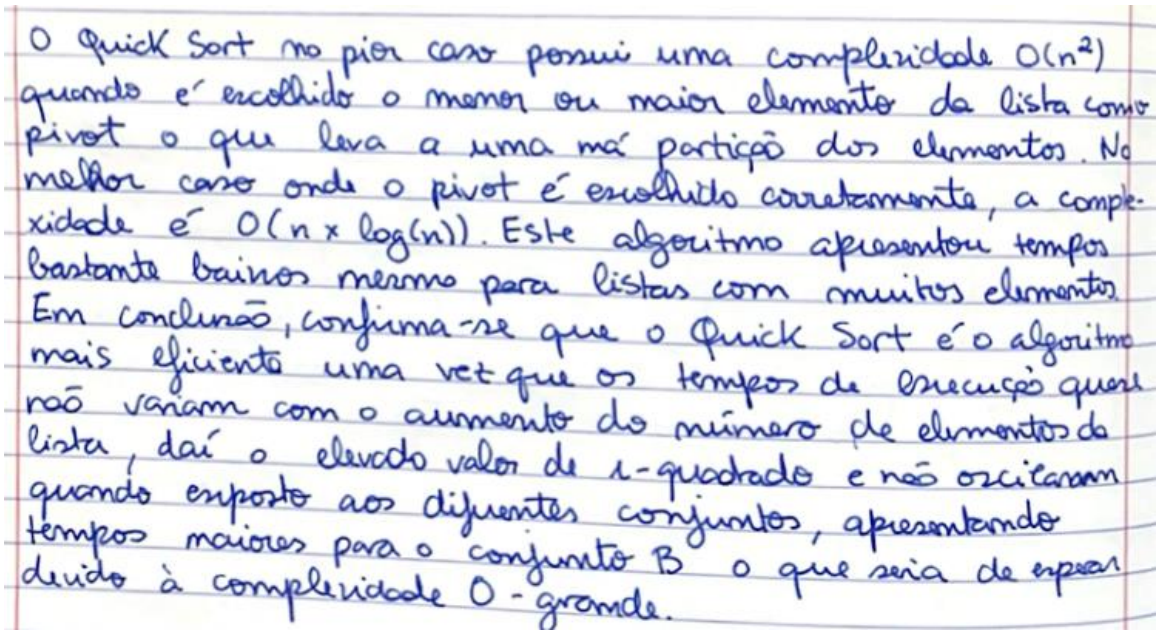
4.1 Tarefa 1

O algoritmo Insertion Sort apresenta uma complexidade de $O(n^2)$ no pior caso onde os elementos da lista se encontram inversamente ordenados, como é o caso do conjunto B. No melhor caso, a complexidade é $O(n)$, algo que acontece quando este algoritmo é imposto a uma lista que já se encontra ordenada corretamente. Em relação aos tempos de ordenamento é notório um enorme crescimento dos mesmos com o aumento dos elementos da lista, daí os elevados valores de n-quadrado. Não obstante, este algoritmo foi o que obteve melhores resultados no conjunto A devido ao reduzido número de comparações realizadas (n comparações). Após a análise dos gráficos, confirma-se que apresenta tempos maiores para o conjunto B e tempos bastante reduzidos para o conjunto A.

4.2 Tarefa 2


O algoritmo Heap Sort distingue-se dos demais por apresentar uma complexidade de O(n x log(n)) para ambos os casos o que se deve ao facto de se basear numa árvore binária (altura log(n) para n elementos) para manter os pais maiores que os respetivos filhos. Como já foi dito, a complexidade deste algoritmo não se altera, fazendo com que o mesmo se comporte bastante bem para os diferentes conjuntos tornando-se uma ótima escolha mesmo para listas com muitos elementos. Isto foi confirmado uma vez que, mesmo exposto aos diferentes conjuntos, o Heap Sort não apresentou variações nos tempos de ordenamento, algo que é comprovado pelo valor de r-quadrado

4.3 Tarefa 3


O Quick Sort no pior caso possui uma complexidade $O(n^2)$ quando é escolhido o menor ou maior elemento da lista como pivot o que leva a uma má partição dos elementos. No melhor caso onde o pivot é escolhido corretamente, a complexidade é O(n x log(n)). Este algoritmo apresentou tempos bastante baixos mesmo para listas com muitos elementos. Em conclusão, confirma-se que o Quick Sort é o algoritmo mais eficiente uma vez que os tempos de execução quase não variam com o aumento do número de elementos da lista, daí o elevado valor de r-quadrado e não oscilaram quando exposto aos diferentes conjuntos, apresentando tempos maiores para o conjunto B o que seria de esperar devido à complexidade O-grande.

Anexo A - Delimitação de Código de Autor

Desenvolvimento dos algoritmos Insertion Sort e Heap Sort com base na informação recolhida nos slides disponíveis, bem como a criação das funções para gerar os conjuntos e imprimir as listas.

Anexo B - Referências

Melhorar o algoritmo Quick Sort com ajuda encontrada no site: https://www.programiz.com/dsa/quick-sort

Anexo C – Listagem Código

```
import time
import random

def insertionSort(array):
    ini = time.time()
    for i in range(1, len(array)):
        key = array[i]
        previous = i - 1
        while previous >= 0 and key < array[previous]:
            array[previous + 1] = array[previous]
            previous = previous - 1
        array[previous + 1] = key
    fim = time.time()
    return format(fim - ini, ".5f")

def heapify(arr, n, i):
    maior = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[i] < arr[left]:
        maior = left
    if right < n and arr[maior] < arr[right]:
        maior = right
    if maior != i:
        arr[i], arr[maior] = arr[maior], arr[i]
        heapify(arr, n, maior)

def heapSort(arr):
    ini = time.time()
    n = len(arr)
    for i in range(n//2, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    fim = time.time()
    return format(fim - ini, ".5f")
```

```
def partition(array, first, last):
    lista = [array[first], array[(first+last)//2], array[last]]
    insertionSort(lista)
    array[first], array[(first+last)//2], array[last] = lista[0], lista[1], lista[2]
    pivot = array[(first+last)//2]
    array[(first+last)//2], array[last] = array[last], array[(first+last)//2]
    i = first - 1
    for j in range(first, last):
        if array[j] <= pivot:
            i = i + 1
            array[i], array[j] = array[j], array[i]
    array[i + 1], array[last] = array[last], array[i + 1]
    return i + 1

def quickSort(array, first, last):
    ini = time.time()
    if first < last:
        pi = partition(array, first, last)
        quickSort(array, first, pi - 1)
        quickSort(array, pi + 1, last)
    fim = time.time()
    return format(fim - ini, ".5f")

def calcularA(tamanho):
    conjuntoA = [random.randint(0, tamanho + 1) for i in range(tamanho)]
    conjuntoA.sort()
    return conjuntoA

def calcularB(tamanho):
    conjuntoB = [random.randint(0, tamanho + 1) for i in range(tamanho)]
    conjuntoB.sort(reverse = True)
    return conjuntoB

def calcularC(tamanho):
    conjuntoC = [random.randint(0, tamanho*2) for i in range(tamanho)]
    return conjuntoC

def printArray(array):
    print(3*" ",end="")
    for element in array:
        print(element,end=" ")
    print("\n")
```

```
def main():
    tamanhos = [20000,40000,60000,80000,100000]
    print("\n- INSERTION SORT -\n")
    for i in tamanhos:
        print("TAMANHO:",i,"\n")
        conjuntoA = calcularA(i)
        conjuntoB = calcularB(i)
        conjuntoC = calcularC(i)
        print(" - CONJUNTO A\n")
        tempo = insertionSort(conjuntoA)
        print(2*" ","Tempo:",tempo,"\n")
        print(" - CONJUNTO B\n")
        tempo = insertionSort(conjuntoB)
        print(2*" ","Tempo:",tempo,"\n")
        print(" - CONJUNTO C\n")
        tempo = insertionSort(conjuntoC)
        print(2*" ","Tempo:",tempo,"\n")
    print("- HEAP SORT -\n")
    for i in tamanhos:
        print("TAMANHO:",i,"\n")
        conjuntoA = calcularA(i)
        conjuntoB = calcularB(i)
        conjuntoC = calcularC(i)
        print(" - CONJUNTO A\n")
        tempo = heapSort(conjuntoA)
        print(2*" ","Tempo:",tempo,"\n")
        print(" - CONJUNTO B\n")
        tempo = heapSort(conjuntoB)
        print(2*" ","Tempo:",tempo,"\n")
        print(" - CONJUNTO C\n")
        tempo = heapSort(conjuntoC)
        print(2*" ","Tempo:",tempo,"\n")
```

```
    print("- QUICK SORT -\n")
    for i in tamanhos:
        print("TAMANHO:",i,"\n")
        conjuntoA = calcularA(i)
        conjuntoB = calcularB(i)
        conjuntoC = calcularC(i)
        print(" - CONJUNTO A\n")
        tempo = quickSort(conjuntoA,0,i - 1)
        print(2*" ","Tempo:",tempo,"\n")
        print(" - CONJUNTO B\n")
        tempo = quickSort(conjuntoB,0,i - 1)
        print(2*" ","Tempo:",tempo,"\n")
        print(" - CONJUNTO C\n")
        tempo = quickSort(conjuntoC,0,i - 1)
        print(2*" ","Tempo:",tempo,"\n")

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