Relatório Projeto 3 AED 2023/2024

Nome: Sérgio Lopes Marques Nº Estudante: 2022222096
PL (inscrição): 1 Email: sergiolmarques2004@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**	**X**

2. Recolha de Resultados *(tabelas)*

Insertion Sort

	Tempo (s)		
Tamanho do Array	**Conjunto A**	**Conjunto B**	**Conjunto C**
20000	0	22,22142	10,92342
40000	0	89,760407	43,825512
60000	0,0156252	201,44503	99,141271
80000	0,0156252	356,86431	177,53497
100000	0,015626	552,86871	278,14725

Heap Sort

	Tempo (s)		
Tamanho do Array	**Conjunto A**	**Conjunto B**	**Conjunto C**
20000	0,0937502	0,0939958	0,0934992
40000	0,2031174	0,1875076	0,203125
60000	0,3135393	0,2971869	0,3121758
80000	0,4375	0,4218748	0,4221885
100000	0,5781257	0,5468748	0,5468764

Quick Sort

	Tempo (s)		
Tamanho do Array	**Conjunto A**	**Conjunto B**	**Conjunto C**
20000	0,0468724	0,046886	0,0312481
40000	0,0781217	0,1093872	0,078125
60000	0,0937502	0,1875	0,1253109
80000	0,1409371	0,2496884	0,1562495
100000	0,1875002	0,3284216	0,2028286

Conjunto A

	Tempo (s)		
Tamanho do Array	**Insertion**	**Heap**	**Quick**
20000	0	0,0937502	0,0468724
40000	0	0,2031174	0,0781217
60000	0,0156252	0,3135393	0,0937502
80000	0,0156252	0,4375	0,1409371
100000	0,015626	0,5781257	0,1875002

Conjunto B

	Tempo (s)		
Tamanho do Array	**Insertion**	**Heap**	**Quick**
20000	22,22142	0,0939958	0,046886
40000	89,760407	0,1875076	0,1093872
60000	201,44503	0,2971869	0,1875
80000	356,86431	0,4218748	0,2496884
100000	552,86871	0,5468748	0,3284216

Conjunto C

	Tempo (s)		
Tamanho do Array	**Insertion**	**Heap**	**Quick**
20000	10,92342	0,0934992	0,0312481
40000	43,825512	0,203125	0,078125
60000	99,141271	0,3121758	0,1253109
80000	177,53497	0,4221885	0,1562495
100000	278,14725	0,5468764	0,2028286

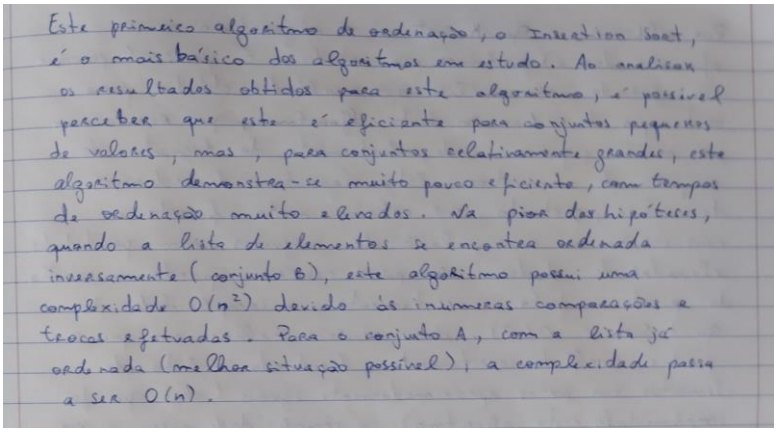
// usar o espaço que considerar necessário //

3. Visualização de Resultados *(gráficos)*

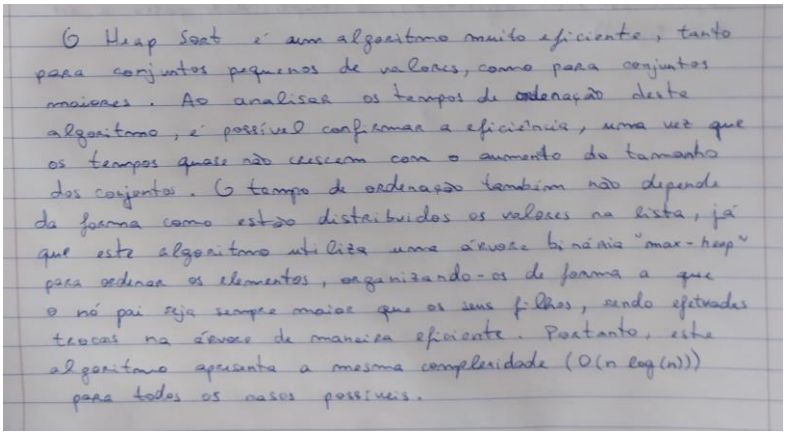
// usar o espaço que considerar necessário //

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

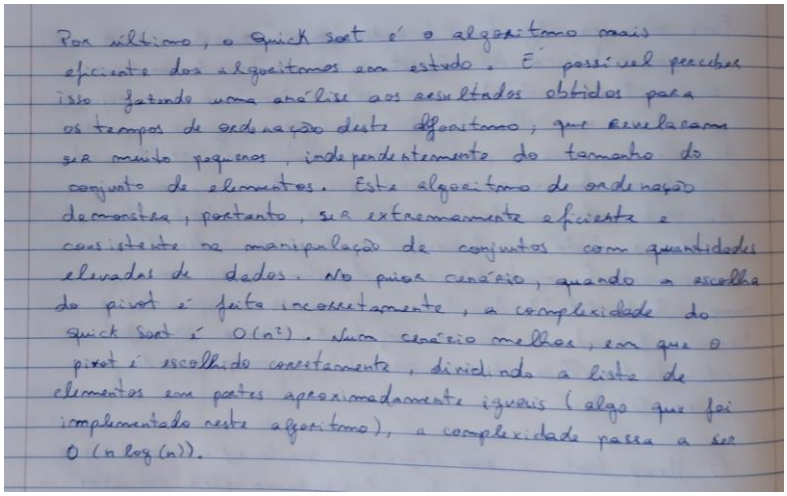
4.1 Tarefa 1


Este primeiro algoritmo de ordenação, o Insertion Sort, é o mais básico dos algoritmos em estudo. Ao analisar os resultados obtidos para este algoritmo, é possível perceber que este é eficiente para conjuntos pequenos de valores, mas, para conjuntos relativamente grandes, este algoritmo demonstra-se muito pouco eficiente, com tempos de ordenação muito elevados. No pior das hipóteses, quando a lista de elementos se encontra ordenada inversamente (conjunto B), este algoritmo possui uma complexidade $O(n^2)$ devido às inúmeras comparações e trocas efetuadas. Para o conjunto A, com a lista já ordenada (melhor situação possível), a complexidade passa a ser $O(n)$.

4.2 Tarefa 2


O Heap Sort é um algoritmo muito eficiente, tanto para conjuntos pequenos de valores, como para conjuntos maiores. Ao analisar os tempos de ordenação deste algoritmo, é possível confirmar a eficiência, uma vez que os tempos quase não crescem com o aumento do tamanho dos conjuntos. O tempo de ordenação também não depende da forma como estão distribuídos os valores na lista, já que este algoritmo utiliza uma árvore binária "max-heap" para ordenar os elementos, organizando-os de forma a que o nó pai seja sempre maior que os seus filhos, sendo efetuadas trocas na árvore de maneira eficiente. Portanto, este algoritmo apresenta a mesma complexidade ($O(n \log(n))$) para todos os casos possíveis.

4.3 Tarefa 3


Por último, o Quick Sort é o algoritmo mais eficiente dos algoritmos em estudo. É possível perceber isso fazendo uma análise aos resultados obtidos para os tempos de ordenação deste algoritmo, que revelam ser muito pequenos, independentemente do tamanho do conjunto de elementos. Este algoritmo de ordenação demonstra, portanto, ser extremamente eficiente e consistente na manipulação de conjuntos com quantidades elevadas de dados. No pior cenário, quando a escolha do pivot é feita incorretamente, a complexidade do Quick Sort é $O(n^2)$. Num cenário melhor, em que o pivot é escolhido corretamente, dividindo a lista de elementos em partes aproximadamente iguais (algo que foi implementado neste algoritmo), a complexidade passa a ser $O(n \log(n))$.

Anexo A - Delimitação de Código de Autor

O código desenvolvido neste projeto é da minha autoria. Contudo, a implementação dos algoritmos de ordenação foi baseada nos slides e vídeos disponibilizados pelo professor. Para a implementação do Quick Sort foi, ainda, retirada alguma informação do site cujo link se encontra de seguida.

Anexo B - Referências

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

Anexo C – Listagem Código

Insertion Sort

```
def insertionSort(lista):

    inicio = time.time()

    for i in range(1, len(lista)):

        valor = lista[i]

        j = i - 1
        while(j >= 0 and valor < lista[j]):
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = valor

    fim = time.time()
    tempo = fim - inicio

    return tempo
```

Heap Sort

```
def heap(lista, tam, valor):

    maior = valor

    esquerda = 2 * valor + 1
    direita = 2 * valor + 2

    if esquerda < tam and lista[maior] < lista[esquerda]:
        maior = esquerda

    if direita < tam and lista[maior] < lista[direita]:
        maior = direita

    if maior != valor:
        temp = lista[maior]
        lista[maior] = lista[valor]
        lista[valor] = temp
        heap(lista, tam, maior)

def heapSort(lista):

    inicio = time.time()

    tam = len(lista)

    for i in range(tam//2, -1, -1):
        heap(lista, tam, i)

    for i in range(tam - 1, 0, -1):
        temp = lista[0]
        lista[0] = lista[i]
        lista[i] = temp
        heap(lista, i, 0)

    fim = time.time()
    tempo = fim - inicio

    return tempo
```

Quick Sort

```python
def particao(lista, inicio, fim):

    primeiro = lista[inicio]
    meio = (inicio + fim) // 2
    intermedio = lista[meio]
    ultimo = lista[fim]

    if(primeiro <= intermedio <= ultimo) or (ultimo <= intermedio <= primeiro):
        indicePivot = meio
    elif(intermedio <= primeiro <= ultimo) or (ultimo <= primeiro <= intermedio):
        indicePivot = inicio
    else:
        indicePivot = fim

    pivot = lista[indicePivot]

    temp = lista[indicePivot]
    lista[indicePivot] = lista[fim]
    lista[fim] = temp

    indice = inicio - 1

    for j in range(inicio, fim):
        if lista[j] <= pivot:
            indice += 1
            temp = lista[j]
            lista[j] = lista[indice]
            lista[indice] = temp

    temp = lista[indice + 1]
    lista[indice + 1] = lista[fim]
    lista[fim] = temp

    return indice + 1
```

```python
def quickSort(lista, inicio, fim):

    tempoInicio = time.time()
    if inicio < fim:

        indicePivot = particao(lista, inicio, fim)

        quickSort(lista, inicio, indicePivot - 1)
        quickSort(lista, indicePivot + 1, fim)

    tempoFim = time.time()
    tempo = tempoFim - tempoInicio

    return tempo
```

Conjuntos

```python
def conjunto_A(tamanho):

    conjuntoA = [random.randint(0, tamanho) for i in range(tamanho)]
    conjuntoA.sort()

    return conjuntoA

def conjunto_B(tamanho):

    conjuntoB = [random.randint(0, tamanho) for i in range(tamanho)]
    conjuntoB.sort(reverse = True)

    return conjuntoB

def conjunto_C(tamanho):

    conjuntoC = [random.randint(0, tamanho) for i in range(tamanho)]

    return conjuntoC
```

Main

```python
def main():
    tamanhos = [20000, 40000, 60000, 80000, 100000]

    for tam in tamanhos:
        print("Tamanho dos Conjuntos: ", tam)

        print("== Insertion Sort ==\n")
        conjuntoA = conjunto_A(tam)
        conjuntoB = conjunto_B(tam)
        conjuntoC = conjunto_C(tam)
        tempo = insertionSort(conjuntoA)
        print("Tempo (conjunto A): ", tempo, "s")
        tempo = insertionSort(conjuntoB)
        print("Tempo (conjunto B): ", tempo, "s")
        tempo = insertionSort(conjuntoC)
        print("Tempo (conjunto C): ", tempo, "s")
        print("\n")

        print("== Heap Sort ==\n")
        conjuntoA = conjunto_A(tam)
        conjuntoB = conjunto_B(tam)
        conjuntoC = conjunto_C(tam)
        tempo = heapSort(conjuntoA)
        print("Tempo (conjunto A): ", tempo, "s")
        tempo = heapSort(conjuntoB)
        print("Tempo (conjunto B): ", tempo, "s")
        tempo = heapSort(conjuntoC)
        print("Tempo (conjunto C): ", tempo, "s")
        print("\n")

        print("== Quick Sort ==\n")
        conjuntoA = conjunto_A(tam)
        conjuntoB = conjunto_B(tam)
        conjuntoC = conjunto_C(tam)
        tempo = quickSort(conjuntoA, 0, len(conjuntoA) - 1)
        print("Tempo (conjunto A): ", tempo, "s")
        tempo = quickSort(conjuntoB, 0, len(conjuntoB) - 1)
        print("Tempo (conjunto B): ", tempo, "s")
        tempo = quickSort(conjuntoC, 0, len(conjuntoC) - 1)
        print("Tempo (conjunto C): ", tempo, "s")
        print("\n")

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