

Relatório Projeto 3 AED 2023/2024

Nome: Daniel Coelho Pereira
PL (inscrição): 4

Nº Estudante: 2021237092
Email: uc2021237092@student.uc.pt

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					
Heap Sort					
Quick Sort					
Finalização Relatório					

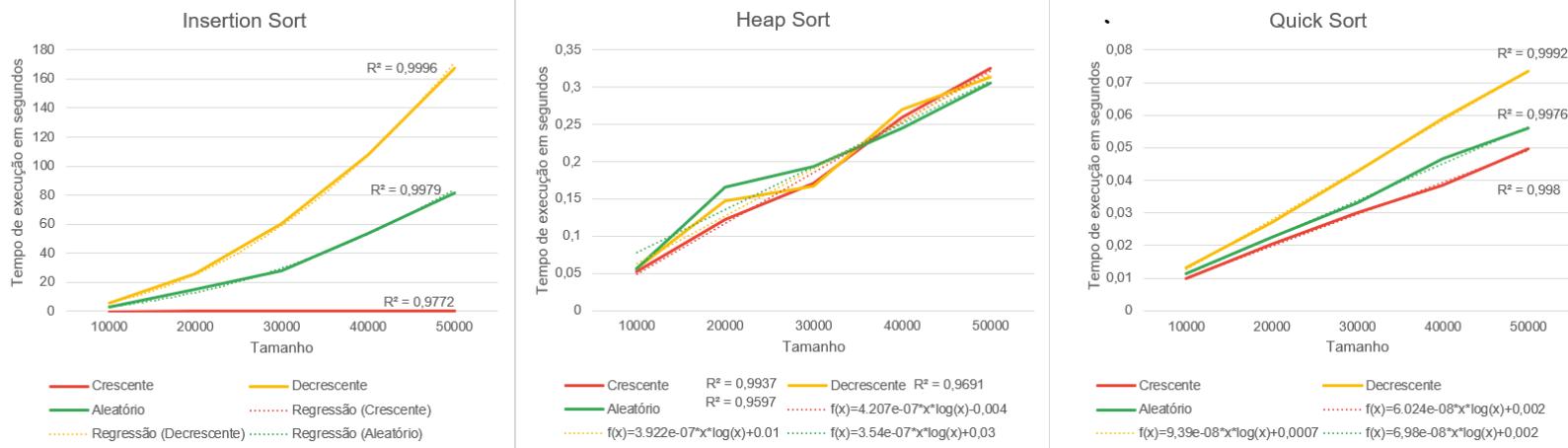
2. Recolha de Resultados

Tabela com registos temporais para cada algoritmo, em cada ordem e para cada tamanho, em segundos

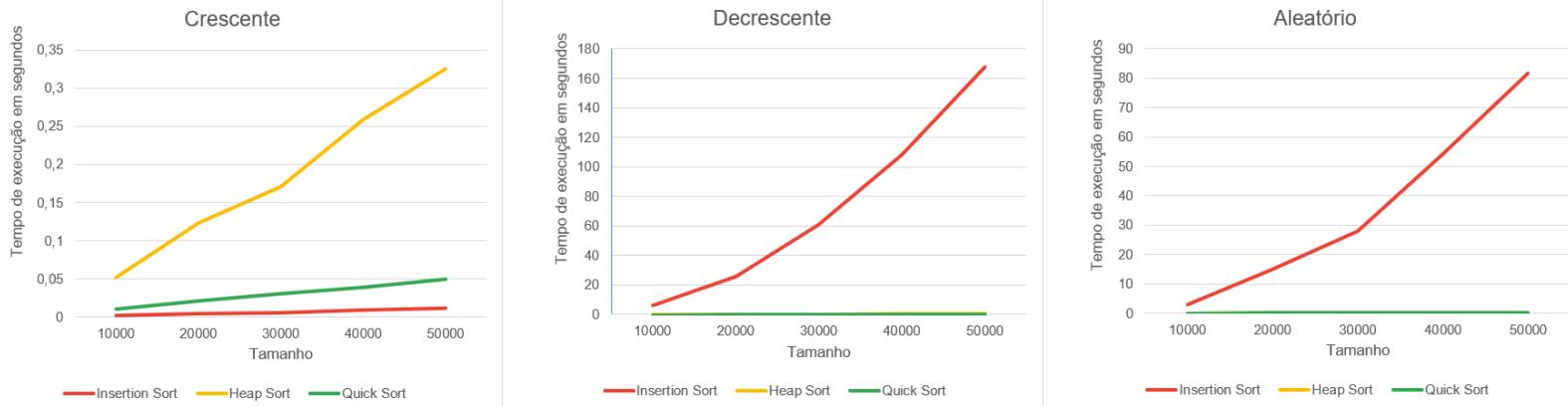
Algoritmo	Tamanho Ordem					
		10000	20000	30000	40000	50000
Insertion Sort	Crescente	0,001985	0,004322	0,00533	0,008985	0,011303
	Decrecente	5,987679	25,805359	60,651003	108,002830	167,640029
	Aleatório	3,00928	15,070705	28,052362	54,023916	81,668031
Heap Sort	Crescente	0,052194	0,122538	0,170544	0,258975	0,324799
	Decrecente	0,055841	0,147615	0,166883	0,269271	0,313172
	Aleatório	0,056865	0,165565	0,193132	0,244349	0,305511
Quick Sort	Crescente	0,009998	0,020327	0,030203	0,038381	0,049818
	Decrecente	0,01334	0,027005	0,042788	0,058987	0,073498
	Aleatório	0,011333	0,022661	0,033109	0,046669	0,055911

3. Visualização de Resultados

-Gráficos para cada tipo de algoritmo



-Gráficos para cada conjunto de chaves



4. Conclusões

4.1 Tarefa 1

Este algoritmo de ordenamento revelou-se bastante fácil de implementar. Através da análise dos dados obtidos podemos constatar que se trata de um algoritmo bastante eficiente no que toca a listas crescentes, tendo nesse caso uma complexidade $O(N)$, melhor do que o Heap Sort e do que o Quick Sort para as mesmas listas. O mesmo não acontece para as listas desordenadas e muito menos para as listas decrescentes em que o algoritmo tem que fazer um elevado número de operações, o que leva a que a complexidade nesses casos seja $O(N^2)$.

4.2 Tarefa 2

Através da observação dos resultados percebemos que para os 3 conjuntos de chaves, o algoritmo tem sempre a mesma complexidade $O(N \log N)$, o que nos leva a inferir que a ordem inicial dos elementos tem pouca influência sobre os tempos do Heap Sort.

Um dos pontos negativos deste algoritmo é que, ao contrário do Insertion Sort, não é estavel, ou seja não preserva a ordem original das chaves iguais.

4.3 Tarefa 3

Pela análise do gráfico do Quick Sort, verificamos que os tempos de ordenamento da lista decrescente são superiores à desordenada e por sua vez superiores à crescente. Em comparação com o Heap Sort, o Quick Sort é mais eficiente. Em média a complexidade deste algoritmo é $O(N \log N)$, no entanto em casos extremos pode chegar a $O(N^2)$. Um fator que contribui a que isso aconteça é a má escolha do pivot. Para evitar isso, na implementação do Quick Sort, o pivot é escolhido através de uma mediana de 3.

Tal como o HeapSort, também é um algoritmo instável.

Anexo A - Delimitação de Código de Autor

O Insertion Sort foi desenvolvido por mim na primeira aula de apoio ao projeto com auxílio do material fornecido na aula e posteriormente melhorado.

O Heap Sort e o Quick Sort foram também desenvolvidos por mim com ajuda das referências abaixo.

Anexo B - Referências

<https://www.geeksforgeeks.org/heap-sort/>

<https://www.blogcyberini.com/2018/08/quicksort-mediana-de-tres.html>

Anexo C – Listagem Código desenvolvido em Python

Implementação do Insertion Sort

```
4 usages
8  def insertionSort(array):
9      for i in range(1, len(array)):
10         chave = array[i]
11         j = i - 1
12         while j >= 0 and chave < array[j]:
13             array[j + 1] = array[j]
14             j = j - 1
15         array[j + 1] = chave
16
17     return array
```

Implementação do Heap Sort

```
3 usages
20  def heapify(array, tamanho, i):
21      maior = i
22      esquerda = 2 * i + 1
23      direita = 2 * i + 2
24
25      if esquerda < tamanho and array[esquerda] > array[maior]:
26          maior = esquerda
27      if direita < tamanho and array[direita] > array[maior]:
28          maior = direita
29      if maior != i:
30          array[i], array[maior] = array[maior], array[i]
31          heapify(array, tamanho, maior)
32
33
3 usages
34  def heapSort(array):
35      tamanho = len(array)
36      for i in range(tamanho // 2 - 1, -1, -1):
37          heapify(array, tamanho, i)
38      for i in range(tamanho - 1, 0, -1):
39          array[i], array[0] = array[0], array[i]
40          heapify(array, i, i == 0)
41
42      return array
```

Implementação do Quick Sort

```
1 usage
47 def particao(array, menor, maior):
48     meio = (menor + maior) // 2
49
50     arrayAux = [array[menor], array[meio], array[maior]]
51     arrayAux = insertionSort(arrayAux)
52     array[menor] = arrayAux[0]
53     array[meio] = arrayAux[1]
54     array[maior] = arrayAux[2]
55
56     array[meio], array[maior] = array[maior], array[meio]
57     pivot = array[maior]
58     i = menor - 1
59
60     for k in range(menor, maior):
61         if array[k] < pivot:
62             i += 1
63             array[i], array[k] = array[k], array[i]
64
65     array[i + 1], array[maior] = array[maior], array[i + 1]
66
67     return i + 1
```

```
5 usages
70 def quickSort(array, menor, maior):
71     if menor < maior:
72         indice = particao(array, menor, maior)
73         quickSort(array, menor, indice - 1)
74         quickSort(array, indice + 1, maior)
75
76     return array
```

Função para gerar chaves

```
1 usage
94 def gerarChaves(tamanho, inicio, fim):
95     listaDesordenada = [random.randint(inicio, fim) for _ in range(tamanho)]
96     listaCrescente = sorted(listaDesordenada)
97     listaDecrescente = sorted(listaDesordenada, reverse=True)
98     return listaCrescente, listaDecrescente, listaDesordenada
```

Função main

```
99 ► if __name__ == "__main__":
100     n = '''inserir tamanho'''
101     num_iteracoes = '''inserir numero de testes'''
102     tempos = {'Insertion Sort': {'Crescente': [], 'Decrescente': [], 'Desordenada': []},
103                'Heap Sort': {'Crescente': [], 'Decrescente': [], 'Desordenada': []},
104                'Quick Sort': {'Crescente': [], 'Decrescente': [], 'Desordenada': []}}
105
106     for _ in range(num_iteracoes):
107         listaCrescente, listaDecrescente, listaDesordenada = gerarChaves(n, inicio: 1, n)
108
109         inicio = time.time()
110         insertionSort(listaCrescente)
111         fim = time.time()
112         tempos['Insertion Sort']['Crescente'].append(fim - inicio)
113
114         inicio = time.time()
115         insertionSort(listaDecrescente)
116         fim = time.time()
117         tempos['Insertion Sort']['Decrescente'].append(fim - inicio)
118
119         inicio = time.time()
120         insertionSort(listaDesordenada)
121         fim = time.time()
122         tempos['Insertion Sort']['Desordenada'].append(fim - inicio)
123
124         inicio = time.time()
125         heapSort(listaCrescente)
126
127             fim = time.time()
128             tempos['Heap Sort']['Crescente'].append(fim - inicio)
129
130             inicio = time.time()
131             heapSort(listaDecrescente)
132             fim = time.time()
133             tempos['Heap Sort']['Decrescente'].append(fim - inicio)
134
135             inicio = time.time()
136             heapSort(listaDesordenada)
137             fim = time.time()
138             tempos['Heap Sort']['Desordenada'].append(fim - inicio)
139
140             inicio = time.time()
141             quickSort(listaCrescente, menor: 0, len(listaCrescente) - 1)
142             fim = time.time()
143             tempos['Quick Sort']['Crescente'].append(fim - inicio)
144
145             inicio = time.time()
146             quickSort(listaDecrescente, menor: 0, len(listaDecrescente) - 1)
147             fim = time.time()
148             tempos['Quick Sort']['Decrescente'].append(fim - inicio)
149
150             inicio = time.time()
151             quickSort(listaDesordenada, menor: 0, len(listaDesordenada) - 1)
152             fim = time.time()
153             tempos['Quick Sort']['Desordenada'].append(fim - inicio)
154
155             for method in tempos.keys():
156                 for order in tempos[method].keys():
157                     average_time = sum(tempos[method][order]) / num_iteracoes
158                     print(f"{method} - Lista {order}: {average_time:.6f} seconds")
```