

Análise de Complexidade de Tempo do Método Heap Sort

Eduardo Costa de Paiva

eduardocspv@gmail.com

Frederico Franco Calhau

fredericoffc@gmail.com

Gabriel Augusto Marson

gabrielmarson@live.com

Faculdade de Computação
Universidade Federal de Uberlândia

16 de dezembro de 2015

Lista de Figuras

2.1	Complexidade de custo do método Heap Sort (Vetor Aleatório)	11
2.2	Complexidade de tempo do método Heap Sort (Vetor Aleatório)	12
2.3	Complexidade de tempo do método Heap Sort com mínimos quadrados (Vetor Aleatório)	12
2.4	Complexidade de custo do método Heap Sort (Vetor Ordenado Crescente)	13
2.5	Complexidade de tempo do método Heap Sort (Vetor Ordenado Crescente)	13
2.6	Complexidade de tempo do método Heap Sort com mínimos quadrados (Vetor Ordenado Crescente)	14
2.7	Complexidade de custo do método Heap Sort (Vetor Ordenado Decrescente)	14
2.8	Complexidade de tempo do método Heap Sort (Vetor Ordenado Decrescente)	15
2.9	Complexidade de tempo do método Heap Sort com mínimos quadrados (Vetor Ordenado Decrescente)	15
2.10	Complexidade de custo do método Heap Sort (Vetor Parcialmente Ordenado Crescente)	16
2.11	Complexidade de tempo do método Heap Sort (Vetor Parcialmente Ordenado Crescente)	16
2.12	Complexidade de tempo do método Heap Sort com mínimos quadrados (Vetor Parcialmente Ordenado Crescente)	17
2.13	Complexidade de custo do método Heap Sort (Vetor Parcialmente Ordenado Decrescente)	17
2.14	Complexidade de tempo do método Heap Sort (Vetor Parcialmente Ordenado Decrescente)	18
2.15	Complexidade de tempo do método Heap Sort com mínimos quadrados (Vetor Parcialmente Ordenado Decrescente)	18

Lista de Tabelas

3.1	Vetor Aleatorio	19
3.2	Vetor Ordenado Crescente	20
3.3	Vetor Ordenado Decrescente	20
3.4	Vetor Parcialmente Ordenado Crescente	21
3.5	Vetor Parcialmente Ordenado Decrescente	21

Lista de Listagens

1.1	HeapSort.py	7
1.2	testeGeneric.py	8
1.3	monitor.py	9
A.1	testdriver.py	24

Sumário

Lista de Figuras	2
Lista de Tabelas	3
1 Introdução	6
1.1 Diretório	6
1.2 Códigos de programas	7
2 Gráficos	11
3 Tabelas	19
4 Análise	22
5 Citações e referências bibliográficas	23
Apêndice	24
A Códigos extensos	24
A.1 testdriver.py	24

Capítulo 1

Introdução

Este documento foi feito com o intuito de exibir uma análise do algoritmo Heap Sort com relação a tempo. Além disso, será feita uma comparação da curva de tempo do que se espera do algoritmo, ou seja, $O(n \lg n)$ com o caso prático.

1.1 Diretório

Dada a seguinte organização das pastas, utilizamos o arquivo testdriver.py, executando, uma função conveniente por vez. Para mais informações vá até ao apêndice.

OBS.: É necessário instalar o programa tree pelo terminal. Isso pode ser feito da seguinte maneira.

```
> sudo apt-get install tree
```

A seguir é mostrada a organização das pastas sendo que os diretórios significativas para o projeto são Codigos e Relatorio além do raiz:

```
tree --charset=ASCII
.
|-- Codigos
|   |-- Bubble
|   |   `-- __pycache__
|   |-- Bucket
|   |   `-- __pycache__
|   |-- Counting
|   |   `-- __pycache__
|   |-- Heap
|   |   `-- __pycache__
|   |-- Insertion
|   |   `-- __pycache__
|   |-- Merge
|   |   `-- __pycache__
|   |-- Quick
|   |   `-- __pycache__
|   |-- Radix
|   `-- Selection
|       `-- __pycache__
```

```

|-- Other
|-- Plot
|-- __pycache__
`-- relatorio
    |-- imagens
    |   |-- Bubble
    |   |-- Bucket
    |   |-- Counting
    |   |-- Heap
    |   |-- Insertion
    |   |-- Merge
    |   |-- Quick
    |   |-- Radix
    |   `-- Selection
    |-- Relatorio_Bubble
    |-- Relatorio_Bucket
    |-- Relatorio_Counting
    |-- Relatorio_Heap
    |-- Relatorio_Insertion
    |-- Relatorio_Merge
    |-- Relatorio_Selection
    `-- Resultados
        |-- Bubble
        |-- Bucket
        |-- Counting
        |-- Heap
        |-- Insertion
        |-- Merge
        |-- Quick
        `-- Selection

```

48 directories

1.2 Códigos de programas

Seguem os códigos utilizados na análise de tempo do algoritmo Heap Sort.

1. HeapSort.py: Disponível na Listagem 1.1.

Listagem 1.1: HeapSort.py

```

1 @profile
2 def trocaElementos(A, x, y):
3     aux = A[y]
4     A[y] = A[x]
5     A[x] = aux
6
7 @profile
8 def maxHeapify(A, n, i):
9     esquerda = 2*i + 1 #Pq o indice começa de 0
10    direita = 2*i + 2 #Pq o indice começa de 0
11
12    if esquerda < n and A[esquerda] > A[i]:
13        maior = esquerda
14    else:

```

```

15     maior = i
16     if direita < n and A[direita] > A[maior]:
17         maior = direita
18     if maior!=i:
19         trocaElementos(A,i,maior)
20         maxHeapify(A,n,maior)
21
22 @profile
23 def constroiMaxHeap(A,n):
24     for i in range(n // 2, -1, -1):
25         maxHeapify(A,n,i)
26
27
28 @profile
29 def heapSort(A):
30     n = len(A)
31     constroiMaxHeap(A,n)
32     m = n
33     for i in range((n-1), 0, -1):
34         trocaElementos(A,0,i)
35         m = m - 1
36         maxHeapify(A,m,0)
37
38 #lista = [13,46,17,34,41,15,14,23,30,21,10,12,21]
39 #heapSort(lista, 13)
40 #print(lista)

```

2. testeGeneric.py Disponível na Listagem 1.2

Listagem 1.2: testeGeneric.py

```

1  ##adicionei - Serve para importar arquivos em outro diretório
2  ###  A CADA NOVO MÉTODO MUDAR O IMPORT,  A CHAMADA DA FUNÇÃO E O SYS.
    PATH
3
4  import sys
5  sys.path.append('/home/gmarson/Git/AnaliseDeAlgoritmos/Trabalho_Final
    /Codigos/Merge')
6  sys.path.append('/home/gmarson/Git/AnaliseDeAlgoritmos/Trabalho_Final
    ')
7
8  from monitor import *
9  from memoria import *
10
11 from MergeSort import *
12 import argparse
13
14 parser = argparse.ArgumentParser()
15 parser.add_argument("n", type=int, help="número de elementos no vetor
    de teste")
16 args = parser.parse_args()
17
18 v = criavet(args.n)
19 merge(v)
20
21
22
23 ## A EXECUÇÃO DESSE ARQUIVO EH ASSIM

```



```

24 ## NA LINHA DE COMANDO VC MANDA O NOME DO ARQUIVO E O TAMANHO DO
    ELEMNT0 DO vetor
25 ##EXEMPLO testeBubble.py 10
26 ##ele gera um vetor aleatório (criavet) e manda pro bubble_sort

```

3. monitor.py Disponível na Listagem 1.3

Listagem 1.3: monitor.py

```

1 # Para instalar o Python 3 no Ubuntu 14 ou 15
2 #
3 # sudo apt-get install python3 python3-numpy python3-matplotlib
    ipython3 python3-psutil
4 #
5
6 from math import *
7 import gc
8 import random
9 import numpy as np
10
11
12 from tempo import *
13
14 # Vetores de teste
15 def troca(m,v,n): ## seleciona o nível de embaralhamento do vetor
16     m = trunc(m)
17     mi = (n-m)//2
18     mf = (n+m)//2
19     for num in range(mi,mf):
20         i = np.random.randint(mi,mf)
21         j = np.random.randint(mi,mf)
22         #print("i= ", i, " j= ", j)
23         t = v[i]
24         v[i] = v[j]
25         v[j] = t
26     return v
27
28
29 def criavet(n, grau=-0.5, inf=-1000, sup=1000):
30     passo = (sup - inf)/n
31     if grau < 0.0:
32         v = np.arange(sup, inf, -passo)
33         if grau <= -1.0:
34             return v
35         else:
36             return troca(-grau*n, v, n)
37     elif grau > 0.0:
38         v = np.arange(inf, sup, passo)
39         if grau >= 1.0:
40             return v
41         else:
42             return troca(grau*n, v, n)
43     else:
44         return np.random.randint(inf, sup, size=n)
45     #return [random.random() for i in range(n)] # for bucket sort
46
47
48
49 #print(criavet(20))

```

```
50
51 #Tipo                                grau
52 #aleatorio                           0
53 #ordenado crescente                  1
54 #ordenado decrescente                -1
55 #parcialmente ordenado crescente     0.5
56 #parcialmente ordenado decrescente  -0.5
57
58
59 def executa(fn, v):
60     gc.disable()
61     with Tempo(True) as tempo:
62         fn(v)
63     gc.enable()
```

4. testdriver.py Referenciado no apêndice [A](#).

Capítulo 2

Gráficos

Seguem os Gráficos utilizadas no processo de análise do método Heap Sort:

1. Para um vetor aleatório

(a) Complexidade de custo do método Heap Sort disponível na lista de imagens [2.1](#).

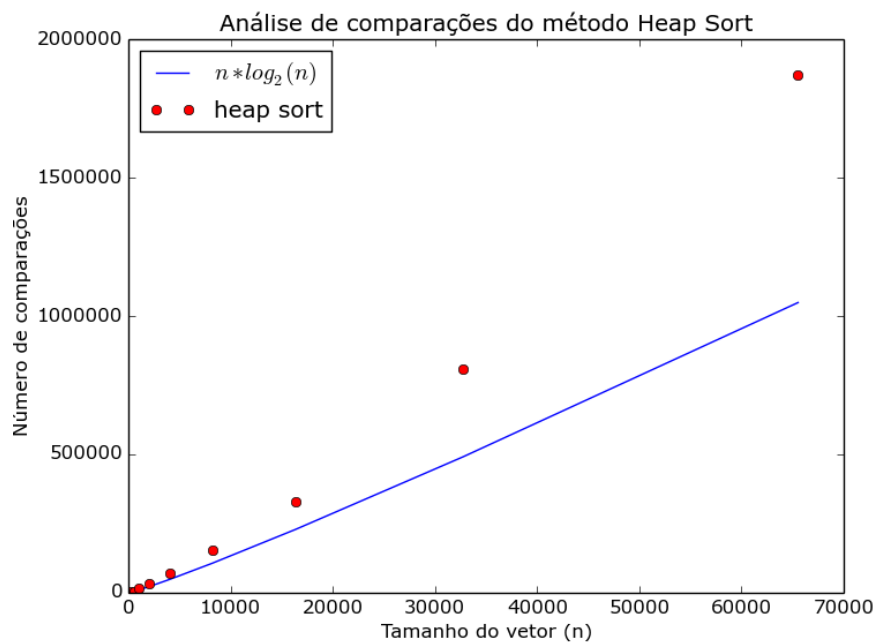


Figura 2.1: Complexidade de custo do método Heap Sort (Vetor Aleatório)

(b) Complexidade de tempo do método Heap Sort disponível na lista de imagens [2.2](#).

(c) Complexidade de tempo do método Heap Sort com mínimos quadrados disponível na lista de imagens [2.3](#).

2. Para um vetor ordenado crescente

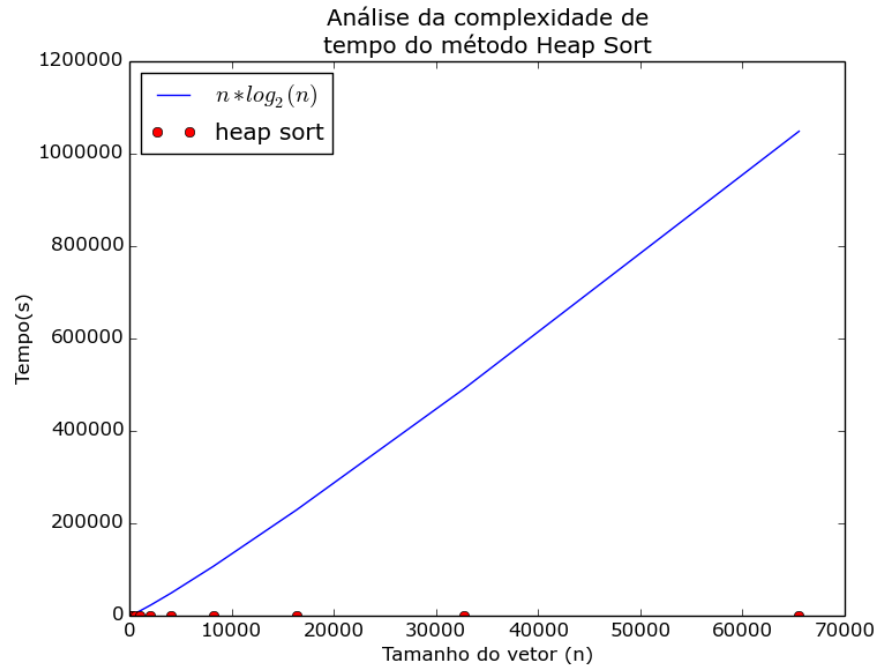


Figura 2.2: Complexidade de tempo do método Heap Sort (Vetor Aleatório)

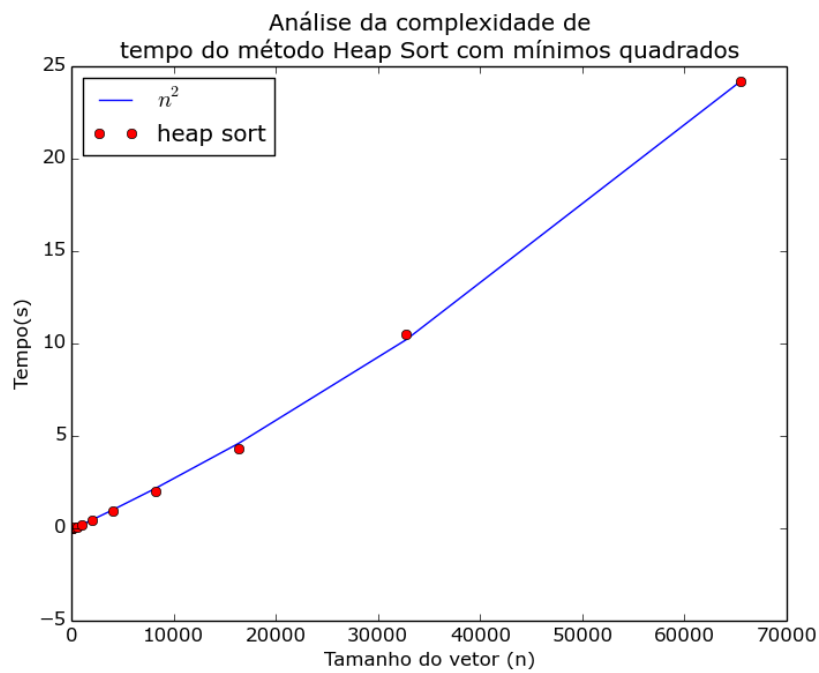


Figura 2.3: Complexidade de tempo do método Heap Sort com mínimos quadrados (Vetor Aleatório)

- (a) Complexidade de custo do método Heap Sort disponível na lista de imagens [2.4](#).
- (b) Complexidade de tempo do método Heap Sort disponível na lista de imagens [2.5](#).
- (c) Complexidade de tempo do método Heap Sort com mínimos quadrados disponível na lista de imagens [2.6](#).

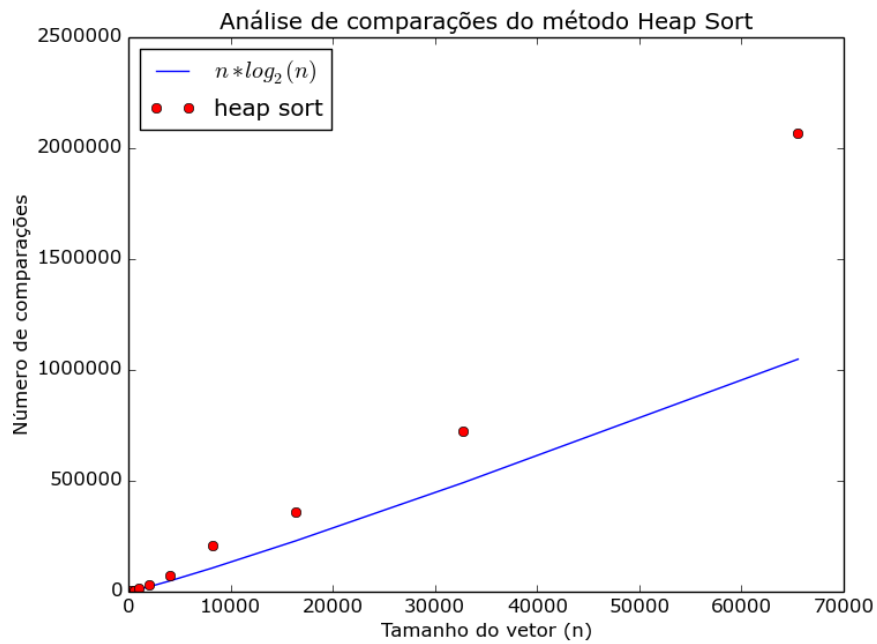


Figura 2.4: Complexidade de custo do método Heap Sort (Vetor Ordenado Crescente)

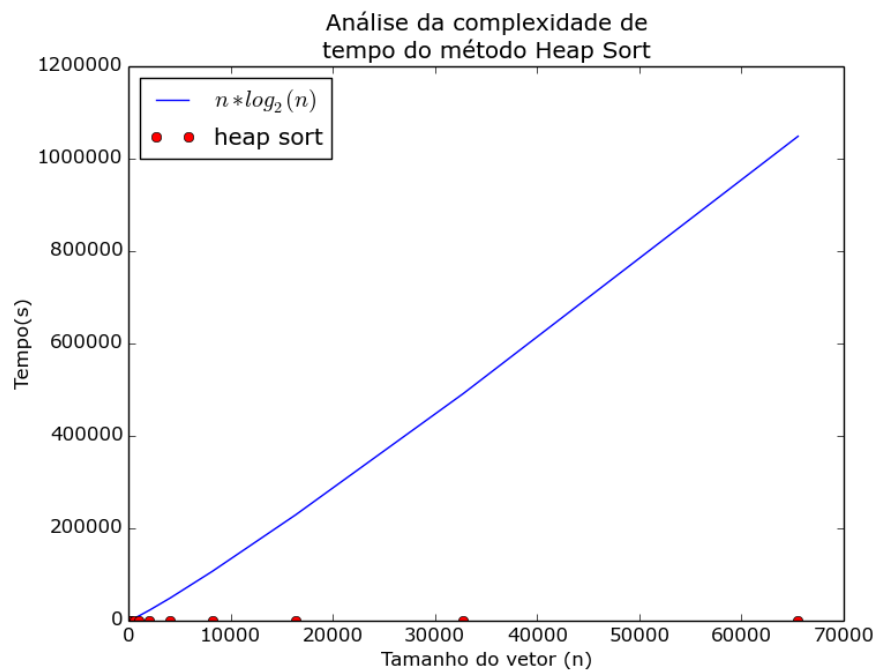


Figura 2.5: Complexidade de tempo do método Heap Sort (Vetor Ordenado Crescente)

3. Para um vetor ordenado decrescente

- (a) Complexidade de custo do método Heap Sort disponível na lista de imagens [2.7](#).
- (b) Complexidade de tempo do método Heap Sort disponível na lista de imagens [2.8](#).
- (c) Complexidade de tempo do método Heap Sort com mínimos quadrados disponível na lista de imagens [2.9](#).

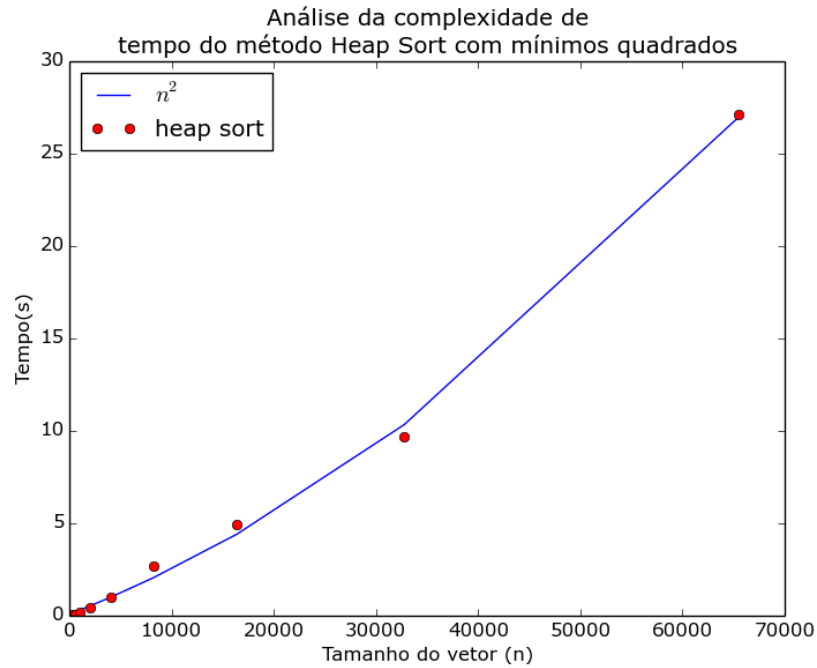


Figura 2.6: Complexidade de tempo do método Heap Sort com mínimos quadrados (Vetor Ordenado Crescente)

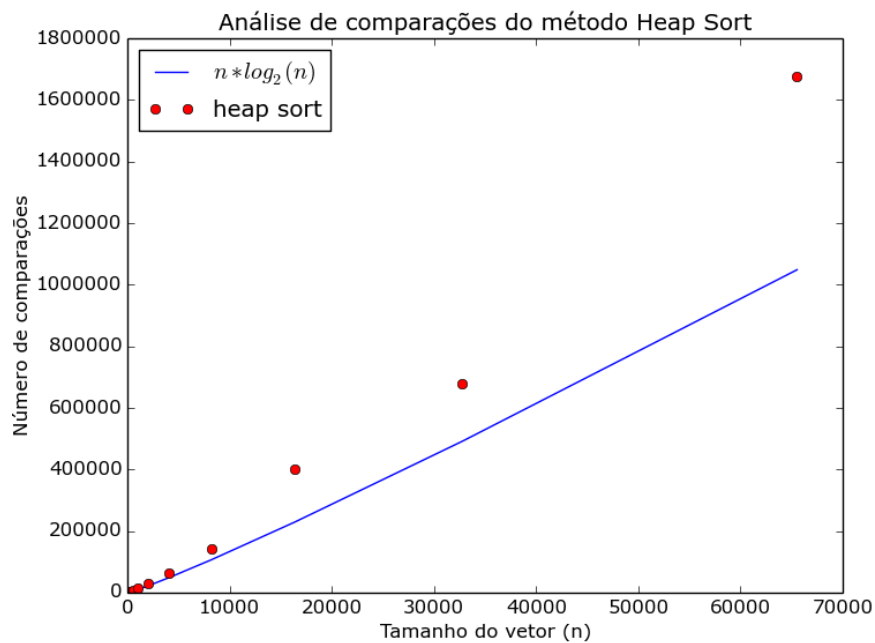


Figura 2.7: Complexidade de custo do método Heap Sort (Vetor Ordenado Decrescente)

4. Para um vetor parcialmente ordenado crescente

- (a) Complexidade de custo do método Heap Sort disponível na lista de imagens [2.10](#).
- (b) Complexidade de tempo do método Heap Sort disponível na lista de imagens [2.11](#).
- (c) Complexidade de tempo do método Heap Sort com mínimos quadrados disponível

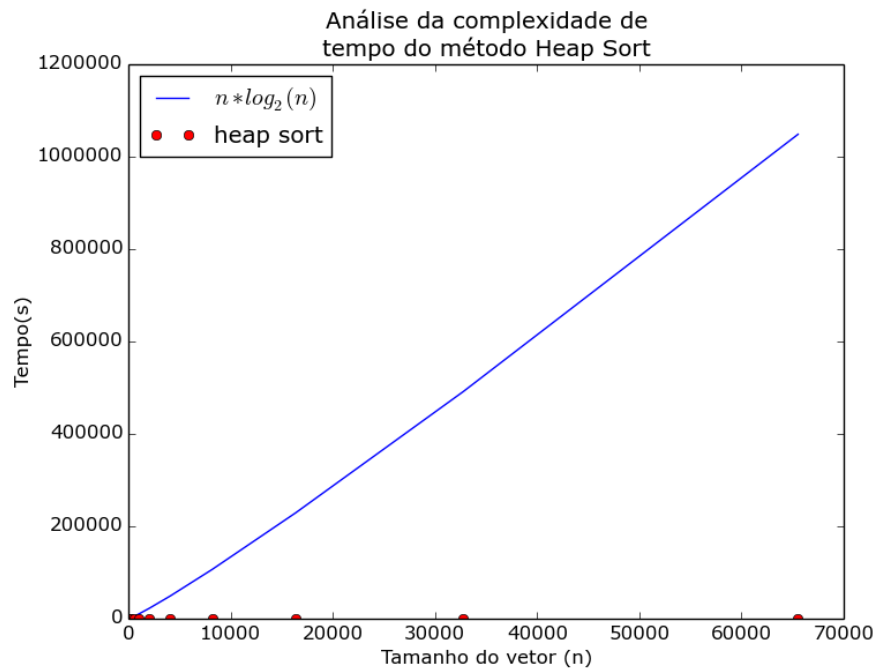


Figura 2.8: Complexidade de tempo do método Heap Sort (Vetor Ordenado Decrescente)

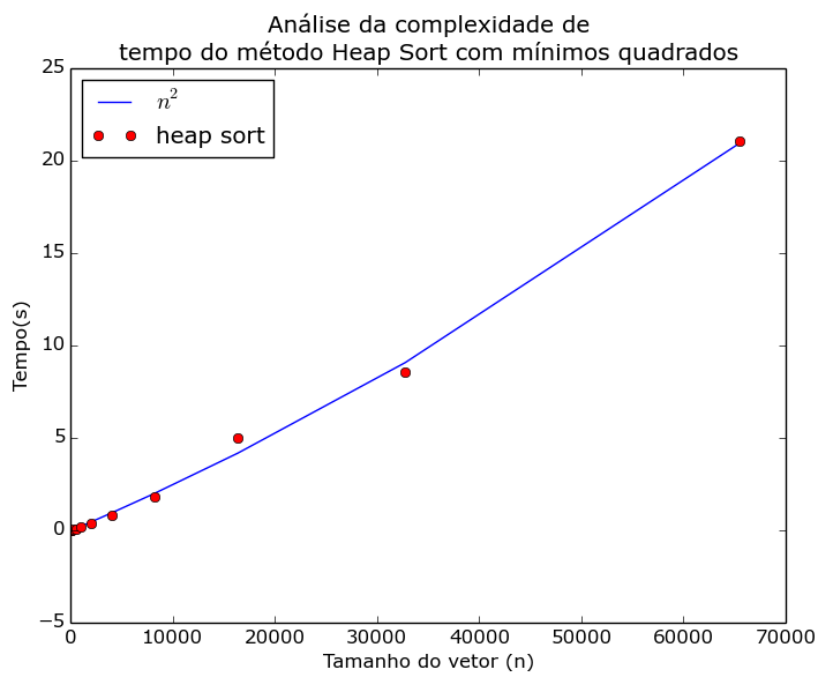


Figura 2.9: Complexidade de tempo do método Heap Sort com mínimos quadrados (Vetor Ordenado Decrescente)

na lista de imagens 2.12.

5. Para um vetor parcialmente ordenado decrescente

(a) Complexidade de custo do método Heap Sort disponível na lista de imagens 2.13.

(b) Complexidade de tempo do método Heap Sort disponível na lista de imagens

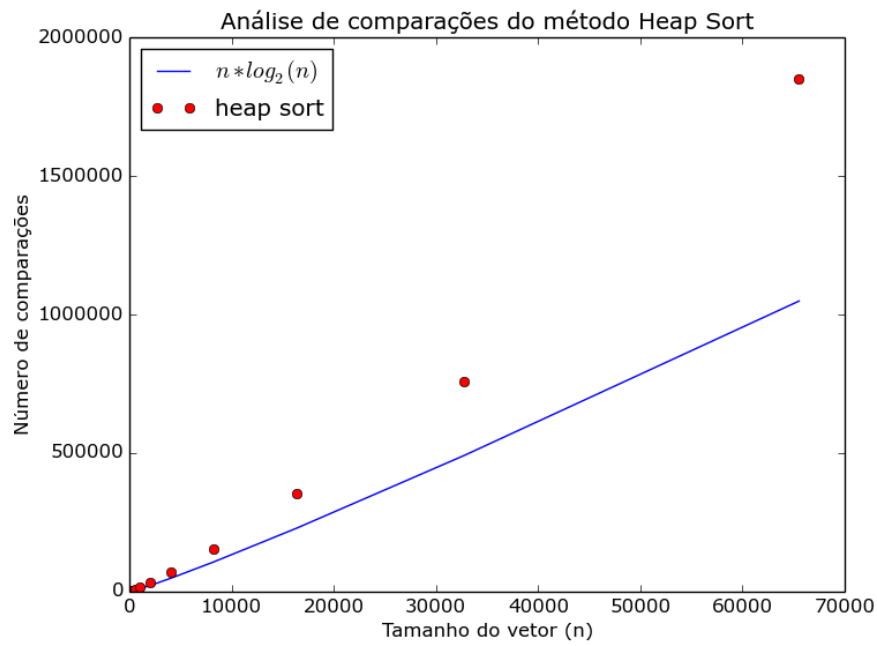


Figura 2.10: Complexidade de custo do método Heap Sort (Vetor Parcialmente Ordenado Crescente)



Figura 2.11: Complexidade de tempo do método Heap Sort (Vetor Parcialmente Ordenado Crescente)

2.14.

- (c) Complexidade de tempo do método Heap Sort com mínimos quadrados disponível na lista de imagens 2.15.

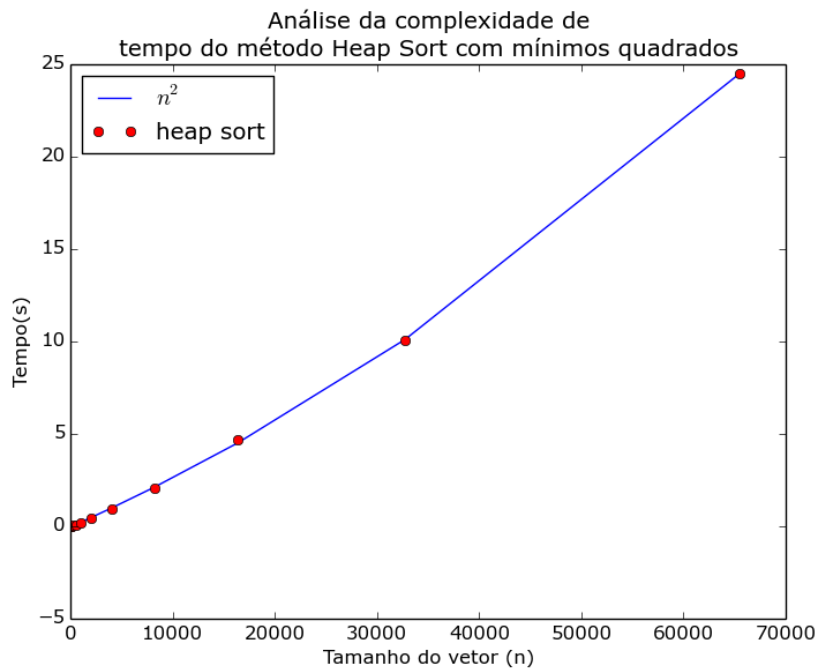


Figura 2.12: Complexidade de tempo do método Heap Sort com mínimos quadrados (Vetor Parcialmente Ordenado Crescente)

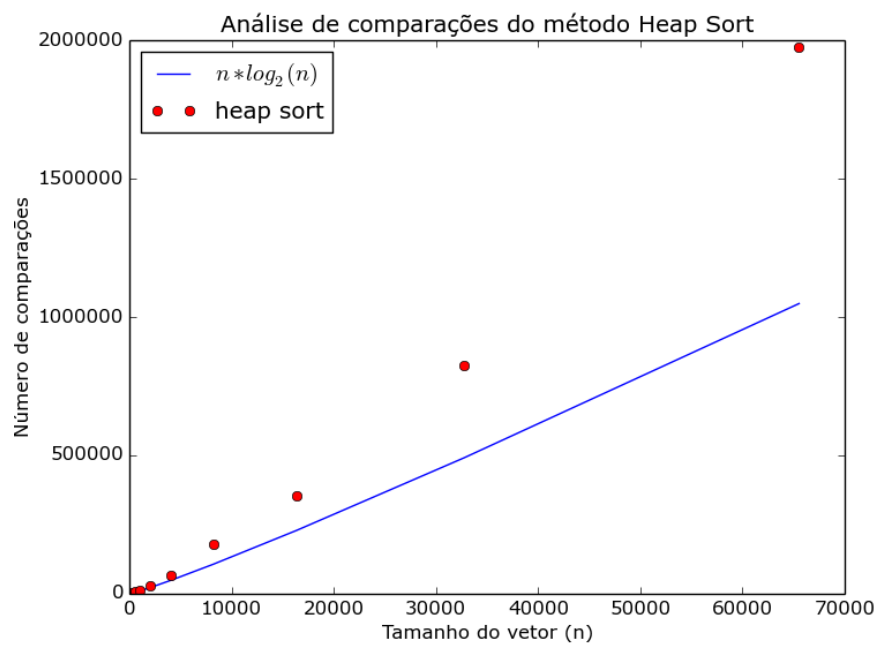


Figura 2.13: Complexidade de custo do método Heap Sort (Vetor Parcialmente Ordenado Decrescente)

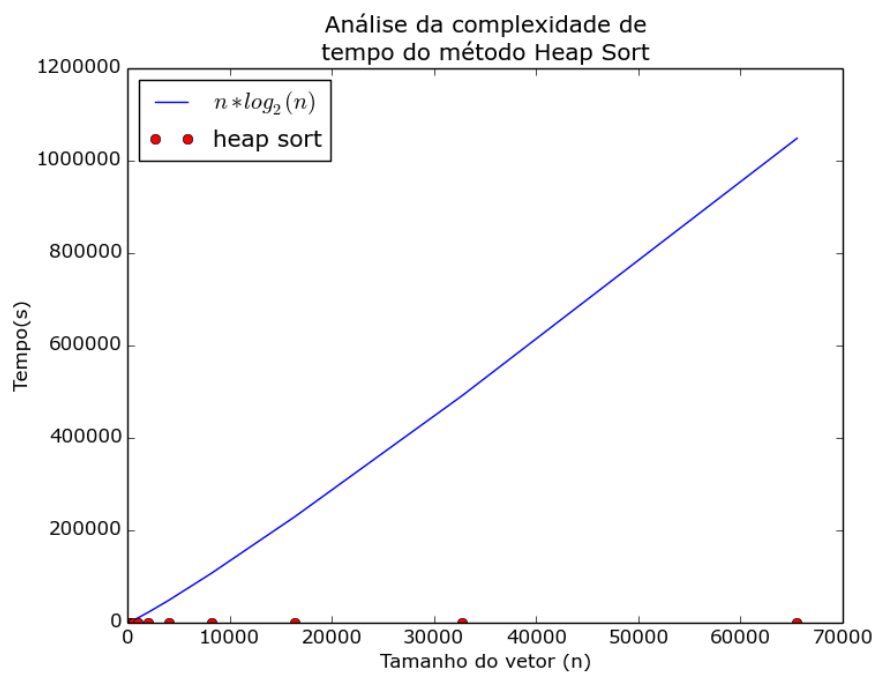


Figura 2.14: Complexidade de tempo do método Heap Sort (Vetor Parcialmente Ordenado Decrescente)

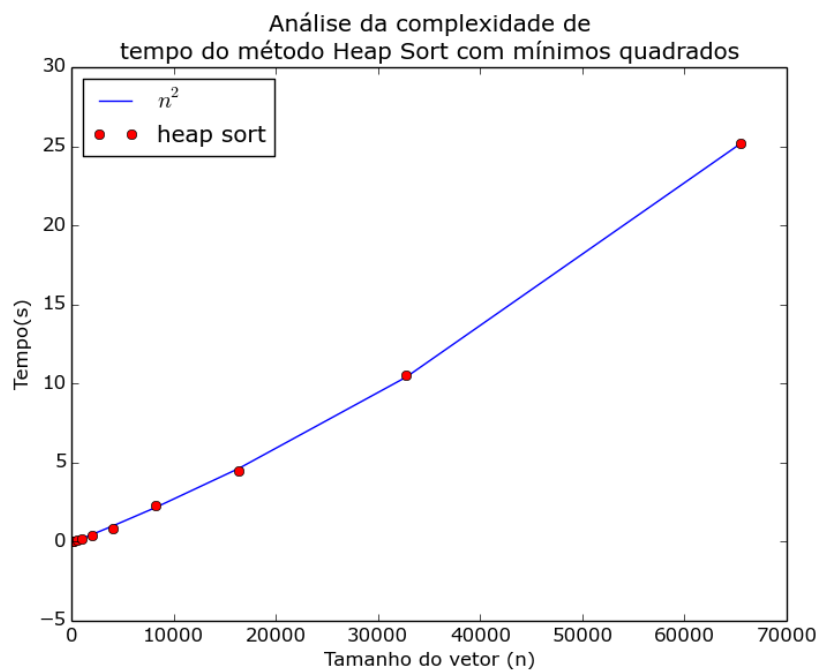


Figura 2.15: Complexidade de tempo do método Heap Sort com mínimos quadrados (Vetor Parcialmente Ordenado Decrescente)

Capítulo 3

Tabelas

Seguem as tabelas utilizadas para a análise do método Heap Sort.

Tabela 3.1: *Vetor Aleatorio*

Tamanho do Vetor	Comparações	Tempo(s)
32	221	0.002936
64	554	0.007282
128	1220	0.016775
256	2972	0.038711
512	6241	0.084794
1024	15566	0.204121
2048	31729	0.417880
4096	72772	0.966805
8192	152718	2.020991
16384	329817	4.330500
32768	809967	10.479563
65536	1975300	24.160150

Tabela 3.2: *Vetor Ordenado Crescente*

Tamanho do Vetor	Comparações	Tempo(s)
32	224	0.003360
64	570	0.007836
128	1301	0.017826
256	3045	0.041852
512	6879	0.093491
1024	15139	0.206329
2048	33154	0.447251
4096	75236	1.009909
8192	208341	2.714416
16384	359038	4.934060
32768	725481	9.676112
65536	2068944	27.122525

Tabela 3.3: *Vetor Ordenado Decrescente*

Tamanho do Vetor	Comparações	Tempo(s)
32	189	0.002377
64	472	0.005641
128	1137	0.013842
256	2799	0.035372
512	5920	0.073351
1024	13172	0.168761
2048	29450	0.370108
4096	65470	0.830505
8192	141369	1.795706
16384	399986	4.984716
32768	677610	8.592706
65536	1675479	21.039164

Tabela 3.4: *Vetor Parcialmente Ordenado Crescente*

Tamanho do Vetor	Comparações	Tempo(s)
32	215	0.003182
64	545	0.007835
128	1430	0.019694
256	3021	0.042281
512	6668	0.089422
1024	15339	0.211859
2048	33122	0.442760
4096	72154	0.962233
8192	153615	2.050023
16384	353799	4.667474
32768	758077	10.041427
65536	1850046	24.526060

Tabela 3.5: *Vetor Parcialmente Ordenado Decrescente*

Tamanho do Vetor	Comparações	Tempo(s)
32	204	0.002475
64	453	0.005813
128	1361	0.017676
256	2659	0.033628
512	6338	0.076890
1024	13982	0.176524
2048	30510	0.386215
4096	66639	0.856832
8192	177670	2.257010
16384	352288	4.448912
32768	826238	10.542908
65536	1975300	25.168916

Capítulo 4

Análise

Podemos observar que todas as curvas de todos os gráficos, exceto os de complexidade de tempo sem a interpolação dos mínimos quadrados (Gráficos 2.2, 2.5, 2.8, 2.11, 2.14), apresentaram uma correspondência forte com a curva da função $F(x) = x \lg(x)$, o que nos permite concluir que, dada a complexidade de tempo do algoritmo Heap Sort por $G(x)$ então $F(x) = c * G(x)$ sendo que c é uma constante maior que zero e $x > x_0$. Portanto, podemos verificar que pelo fato do algoritmo heapSort envolver n chamadas da função maxHeapify, que possui complexidade $O(\lg(n))$, sua complexidade é $O(n \lg(n))$.

Capítulo 5

Citações e referências bibliográficas

Apêndice A

Códigos extensos

A.1 testdriver.py

Listagem A.1: testdriver.py

```
1 # coding = utf-8
2 import subprocess
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import sys, shutil
6
7
8 ##PRA CADA NOVO METODO TEM QUE MUDAR
9 #Sys.path()
10
11 ## PARA CADA VETOR NOVO OU NOVO METODO TEM QUE MUDAR
12 #Para o executa_teste a chamada das funções e o shutil.move()
13 #para os plots a chamada das funções e o savefig
14
15 sys.path.append('/home/gmarson/Git/AnaliseDeAlgoritmos/Trabalho_Final/
    Codigos/Bucket') ## adicionei o código de ordenação
16 sys.path.append('/home/gmarson/Git/AnaliseDeAlgoritmos/Trabalho_Final/
    relatorio/Resultados/Bucket') ## adicionei o resultado do executa_teste
17
18
19 def executa_teste(arqteste, arqsaida, nlin, intervalo):
20     """Executa uma sequência de testes contidos em arqteste, com:
21         arqsaida: nome do arquivo de saída, ex: tBolha.dat
22         nlin: número da linha no arquivo gerado pelo line_profiler contendo
23             os dados de interesse. Ex: 14
24         intervalo: tamanhos dos vetores: Ex: 2 ** np.arange(5,10)
25     """
26     f = open(arqsaida, mode='w', encoding='utf-8')
27     f.write('#          n          tempo(s)\n')
28
29     for n in intervalo:
30         cmd = ' '.join(["kernprof -l -v", "testeGeneric.py", str(n)])
31         str_saida = subprocess.check_output(cmd, shell=True).decode('utf-8')
32         linhas = str_saida.split('\n')
33         #for i in linhas:
```


A.1

```
34     #     print(i)
35     #print (linhas)
36     unidade_tempo = float(linhas[1].split()[2])
37     tempo_total = float(linhas[3].split()[2])
38     #lcomp = linhas[nlin].split()
39
40     #print ("unidade tempo: ",unidade_tempo )
41     #print("lcomp: ",lcomp)
42     #print("tempo total",tempo_total)
43
44     #num_comps = int(lcomp[1])
45     str_res = '{:>8} {:>13} {:13.6f}'.format(n, tempo_total)
46     print(str_res)
47     f.write(str_res + '\n')
48 f.close()
49 shutil.move("tBucket_vetor_parcialmente_ordenado_decrescente.dat", "/
home/gmarson/Git/AnaliseDeAlgoritmos/Trabalho_Final/relatorio/
Resultados/Bucket/tBucket_vetor_parcialmente_ordenado_decrescente.
dat")
50
51 executa_teste("testeGeneric.py", "
tBucket_vetor_parcialmente_ordenado_decrescente.dat", 46, 2 ** np.
arange(5,15))
52
53 def executa_teste_memoria(arqteste, arqsaida, nlin, intervalo):
54     """Executa uma sequência de testes contidos em arqteste, com:
55     arqsaida: nome do arquivo de saída, ex: tBolha.dat
56     nlin: número da linha no arquivo gerado pelo line_profiler contendo
57     os dados de interesse. Ex: 14
58     intervalo: tamanhos dos vetores: Ex: 2 ** np.arange(5,10)
59     """
60     f = open(arqsaida,mode='w', encoding='utf-8')
61     f.write('#         n     comparações         tempo(s)\n')
62
63     for n in intervalo:
64         cmd = ' '.join(["kernprof -l -v ", "testeGeneric.py", str(n)])
65
66         str_saida = subprocess.check_output(cmd, shell=True).decode('utf-8')
67
68         linhas = str_saida.split('\n')
69         for i in linhas:
70             print(i)
71
72         print ("Linhas:",linhas[1])
73
74         unidade_tempo = float(linhas[1].split()[2])
75
76
77         str_res = '{:>8} {:>13} {:13.6f}'.format(n, n, n)
78         print(str_res)
79         f.write(str_res + '\n')
80     f.close()
81     #shutil.move("tSelection_memoria.dat", "/home/gmarson/Git/
AnaliseDeAlgoritmos/Trabalho_Final/relatorio/Resultados/Selection/
tSelection_memoria.dat")
82
83 #executa_teste_memoria("testeGeneric.py", "tSelection_memoria.dat", 14, 2
** np.arange(5,15))
```

```

84
85 def plota_teste1(arqsaida):
86     n, c, t = np.loadtxt(arqsaida, unpack=True)
87     #print("n: ",n,"\nc: ",c,"\nt: ",t)
88     #n eh o tamanho da entrada , c eh o tanto de comparações e t eh o
        tempo gasto
89     plt.plot(n, n ** 2, label='$n^2$') ## custo esperado bubble Sort
90     plt.plot(n, c, 'ro', label='selection sort')
91
92     # Posiciona a legenda
93     plt.legend(loc='upper left')
94
95     # Posiciona o título
96     plt.title('Análise de comparações do método da seleção')
97
98     # Rotula os eixos
99     plt.xlabel('Tamanho do vetor (n)')
100    plt.ylabel('Número de comparações')
101
102    plt.savefig('relatorio/imagens/Selection/
        selection_plot_1_ordenado_descrescente.png')
103    plt.show()
104
105
106
107 def plota_teste2(arqsaida):
108     n, t = np.loadtxt(arqsaida, unpack=True)
109     plt.plot(n, n , label='$n$')
110     plt.plot(n, t, 'ro', label='bucket sort')
111
112     # Posiciona a legenda
113     plt.legend(loc='upper left')
114
115     # Posiciona o título
116     plt.title('Análise da complexidade de \ntempo do método Bucket Sort')
117
118     # Rotula os eixos
119     plt.xlabel('Tamanho do vetor (n)')
120     plt.ylabel('Tempo(s)')
121
122     plt.savefig('relatorio/imagens/Bucket/
        bucket_plot_2_parcialmente_ordenado_decrescente.png')
123     plt.show()
124
125
126 def plota_teste3(arqsaida):
127     n, t = np.loadtxt(arqsaida, unpack=True)
128
129     # Calcula os coeficientes de um ajuste a um polinômio de grau 2 usando
130     # o método dos mínimos quadrados
131     coefs = np.polyfit(n, t, 2)
132     p = np.poly1d(coefs)
133
134     plt.plot(n, p(n), label='$n$')
135     plt.plot(n, t, 'ro', label='bucket sort')
136
137     # Posiciona a legenda
138     plt.legend(loc='upper left')
139

```

A.1

```
140     # Posiciona o título
141     plt.title('Análise da complexidade de \ntempo do método Bucket Sort
              com mínimos quadrados')
142
143     # Rotula os eixos
144     plt.xlabel('Tamanho do vetor (n)')
145     plt.ylabel('Tempo(s)')
146
147     plt.savefig('relatorio/imagens/Bucket/
              bucket_plot_3_parcialmente_ordenado_decrescente.png')
148     plt.show()
149
150 #plota_teste1("/home/gmarson/Git/AnaliseDeAlgoritmos/Trabalho_Final/
              relatorio/Resultados/Selection/tSelection_vetor_ordenado_decrescente.
              dat")
151 plota_teste2("/home/gmarson/Git/AnaliseDeAlgoritmos/Trabalho_Final/
              relatorio/Resultados/Bucket/
              tBucket_vetor_parcialmente_ordenado_decrescente.dat")
152 plota_teste3("/home/gmarson/Git/AnaliseDeAlgoritmos/Trabalho_Final/
              relatorio/Resultados/Bucket/
              tBucket_vetor_parcialmente_ordenado_decrescente.dat")
153
154
155 def plota_teste4(arqsaida):
156     n, c, t = np.loadtxt(arqsaida, unpack=True)
157
158     # Calcula os coeficientes de um ajuste a um polinômio de grau 2 usando
159     # o método dos mínimos quadrados
160     coefs = np.polyfit(n, c, 2)
161     p = np.poly1d(coefs)
162
163     plt.plot(n, p(n), label='$n^2$')
164     plt.plot(n, c, 'ro', label='bubble sort')
165
166     # Posiciona a legenda
167     plt.legend(loc='upper left')
168
169     # Posiciona o título
170     plt.title('Análise da complexidade de \ntempo do método da bolha')
171
172     # Rotula os eixos
173     plt.xlabel('Tamanho do vetor (n)')
174     plt.ylabel('Número de comparações')
175
176     plt.savefig('bubble4.png')
177     plt.show()
178
179 def plota_teste5(arqsaida):
180     n, c, t = np.loadtxt(arqsaida, unpack=True)
181
182     # Calcula os coeficientes de um ajuste a um polinômio de grau 2 usando
183     # o método dos mínimos quadrados
184     coefs = np.polyfit(n, c, 2)
185     p = np.poly1d(coefs)
186
187     # set_yscale('log')
188     # set_yscale('log')
189     plt.semilogy(n, p(n), label='$n^2$')
190     plt.semilogy(n, c, 'ro', label='bubble sort')
```

```
191
192     # Posiciona a legenda
193     plt.legend(loc='upper left')
194
195     # Posiciona o título
196     plt.title('Análise da complexidade de \ntempo do método da bolha')
197
198     # Rotula os eixos
199     plt.xlabel('Tamanho do vetor (n)')
200     plt.ylabel('Número de comparações')
201
202     plt.savefig('bubble5.png')
203     plt.show()
```
