Algoritmo e Estrutura de Dados

Licenciatura em Engenharia Informática

Estudo de algumas Rotinas de Ordenação

Professores:

Tomás Oliveira e Silva

Pedro Lavrador.

Data: 22/01/2021

Trabalho realizado por:

Daniel Figueiredo, nº 98498, 33,33%

Eva Bartolomeu, nº 98513, 33,33%

Eduardo Fernandes, nº 98512, 33,33%

Índice

1. Introdução

No âmbito da cadeira de Algoritmos e Estruturas de Dados, foi-nos proposto um trabalho acerca do estudo de algumas rotinas de ordenação, mais concretamente sobre os seus tempos de execução.

Os algoritmos sugeridos para alvo de análise são: Bubble Sort, Shaker Sort, Insertion Sort, Shell Sort, Quick Sort, Merge Sort, Heap Sort, Selection Sort e Rank Sort.

De forma a obter, em cada rotina de ordenação, as tabelas com os tempos de execução, relacionados com o número de elementos a ordenar, corremos o código fornecido pelo professor. Para adquirir gráficos de tempos de execução, para cada algoritmo, usamos o gnuplot, que é um programa de linha de comando do linux. Finalmente, na construção dos gráficos da função aproximada dos tempos de execução, resolvemos trabalhar em matlab.

Os objetivos deste relatório são:

- Construir tabelas com tempos de execução para cada algoritmo de ordenação;
- Elaborar gráficos com tempos de execução para cada algoritmo de ordenação;
- Criar a função aproximada da função dos valores dos tamanhos do array a ordenar, em tempos médios, para cada algoritmo de ordenação;
- Tentar obter as melhores aproximações possíveis;
- Interpretar as tabelas e gráficos;
- Determinar complexidade computacional em cada rotina de ordenação;
- Saber identificar o melhor algoritmo de ordenação, perante um determinado tamanho do array;
- Saber identificar o melhor algoritmo de ordenação, perante os tempos de execução e o esforço para escrever e verificar o código real.

2. Algoritmos de Ordenação

2.1. Bubble Sort

O Bubble Sort é um algoritmo que percorre um determinado array, e vai trocando, ou não, os elementos adjacentes de forma a que fique em primeiro o elemento menor. Se durante este percurso não houve trocas, então o algoritmo deparou-se com um array já ordenado. Este caso é o caso considerado como o "best case", acabando assim com uma complexidade computacional de O(n).

Caso haja trocas, volta a fazer a comparação de elementos adjacentes e a trocá-los se não tiverem ordenados,mas só até ao penúltimo elemento do array. Depois, volta a verificar se houve trocas ou não, e assim sucessivamente.

O pior caso deste algoritmo, é quando o array está na ordem inversa da ordenação, correspondendo a uma complexidade de O(n^2).

Considera-se este algoritmo, dos piores algoritmos de ordenação, pois normalmente é muito dispendioso em nível de tempo.

2.2. Shaker Sort

Este algoritmo é igual ao outro, com a única diferença é que só é igual alternadamente, ou seja, em primeiro lugar compara elementos adjacentes de um array e troca-os se não estiverem ordenados, e caso não haja alterações para. Depois vai ao elemento mais pequeno do array, e percorre o array até ao início trocando os elementos juntos do array de maneira a respeitar a ordenação. Portanto este algoritmo vai intercalar este processo, com o processo do Bubble Sort.

Tal como no Bubble Sort, o melhor caso tem complexidade computacional de O(n) e o pior tem O(n^2).

2.3. Insertion Sort

O Insertion Sort é uma rotina que percorre um array do segundo elemento até ao início, depois do terceiro até ao primeiro, de seguida do quarto até ao primeiro, e assim sucessivamente. Neste percurso ao longo do array, troca-se os elementos adjacentes de maneira a ficar em primeiro os elementos mais pequenos.

O melhor caso tem complexidade computacional de O(n) e o pior tem O(n^2).

2.4. Shell Sort

O *Shell Sort* é um algoritmo que, em primeiro lugar, vai ordenar os elementos mais distantes do *array* e vai reduzindo, sucessivamente, o intervalo entre os elementos que foram ordenados, tendo em conta a sequência que foi utilizada. Podendo-se dizer que é uma versão generalizada do *Insertion Sort*.

O desempenho do *Shell Sort* depende do tipo de sequência usada para um determinado *array* de entrada. Podemos, então, observar o desempenho deste algoritmo no gráfico 9.

2.5. QuickSort

O *QuickSort* é um algoritmo do tipo *Divide and Conquer.* Este algoritmo escolhe um elemento como *pivot* e vai fazendo partições do *array* de entrada em torno do *pivot* escolhido.

Existem muitas versões de como se escolher o *pivot* que vai ser utilizado, tais como:

1 - Escolher o primeiro elemento*;*

2 - Escolher o último elemento*;*

3 - Escolher um elemento aleatório;

4 - Escolher o valor da mediana.

O processo chave deste algoritmo de ordenação é a partição. O objetivo destas partições é, que, dado um *array* e um elemento *X* do *array* como *pivot*, devemos colocar o elemento *X* na sua posição correta do *array* ordenado e colocar todos os elementos mais pequenos do que *X* antes desse elemento, e colocar os elementos que forem maiores que *X* à frente desse elemento.

Todas estas partições e a ordenação do *array* devem ser feitas em tempo linear, como iremos observar mais à frente (Gráfico 12).

2.6. Merge Sort

O *Merge Sort*, tal como o *QuickSort*, é um algoritmo do tipo *Divide and Conquer.* Este algoritmo é usado recursivamente, dividindo o *array* em dois ou mais *sub-arrays*, até ficar simples o suficiente para que seja resolvido diretamente.

As soluções dos *sub-arrays* são, posteriormente combinados, o que nos vai dar a solução ordenada do *array inicial.* Logo, de um modo geral, o *Merge Sort,* primeiramente divide o *array* em partes iguais e depois combina-as de uma maneira ordenada.

A complexidade espacial do *Merge Sort* é de O(n), visto que este algoritmo vai precisar de espaço extra e pode colocar algumas operações mais lentas.

O desempenho deste algoritmo pode ser observado no gráfico 15.

2.7. Heap Sort

O algoritmo *Heap Sort* é um algoritmo de ordenação generalista, e faz parte da família de algoritmos de ordenação por seleção. O algoritmo do *Heap Sort* usa um *max-heap* para fazer a ordenação. Para ordenar por ordem crescente em um array com valores entre *a[0]* e *a[n-1]*:

- Inicialmente, é construído o *max-heap* a partir da *tree* do array fornecido. Sendo *i=0, 1, … , n-1* os valores de *a[i]* são colocados no *max-heap*.
- Então, para *i=n-1, n-2, … ,0* é retirado o maior valor do *max-heap* e é guardado em *a[i]*. Depois de ter realizado estas tarefas o algoritmo termina.

Da forma que este algoritmo está construído o *max-heap* e o *array* podem partilhar da mesma memória.

2.8. Selection Sort

O *Selection Sort* é um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os *n-1* elementos restantes, até aos últimos dois elementos.

Em cada iteração vamos selecionar o menor valor do *array* que ainda não tenha sido ordenado, e colocá-lo na parte ordenada.

É um algoritmo simples e fácil de ser implementado em comparação aos restantes.

Ele é um dos mais rápidos na ordenação de vetores de tamanhos pequenos, sendo um dos mais lentos na ordenação de vetores de tamanhos grandes.

2.9. Rank Sort

Para cada elemento da lista a ser classificado, o algoritmo *Rank Sort* calcula o número total de elementos que são inferiores a esse número. Esse valor é chamado de *rank* do elemento e, para calculá-lo, o algoritmo precisa comparar o elemento com todos os outros valores da lista.

Em uma lista totalmente classificada em ordem numérica crescente, o *rank* de cada elemento será apenas a sua posição real na lista.

Finalmente, o algoritmo usa a classificação de cada elemento para colocá-lo em sua posição de classificação adequada.

3. Resultados

3.1. Tabela dos tempos de execução

Através dos ficheiros fornecidos pelo Professor, e executando o código C "sorting_methods.c", é gerado um ficheiro "output.txt". Este ficheiro contém as tabelas dos tempos de execução de cada método de ordenação.

```
# bubble_sort
#      n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
      10 5.320e-07 3.131e-06 1.676e-06 1.113e-06
      13 6.150e-07 7.610e-07 6.826e-07 3.409e-08
      16 7.570e-07 1.128e-06 8.944e-07 8.905e-08
      20 9.100e-07 1.139e-06 1.016e-06 5.281e-08
      25 1.162e-06 1.750e-06 1.335e-06 1.177e-07
      32 1.611e-06 2.276e-06 1.822e-06 1.241e-07
      40 2.186e-06 3.113e-06 2.533e-06 2.125e-07
      50 3.719e-06 4.629e-06 4.134e-06 1.980e-07
      63 4.498e-06 6.603e-06 5.507e-06 5.777e-07
      79 6.478e-06 9.875e-06 7.700e-06 8.957e-07
     100 9.794e-06 1.352e-05 1.099e-05 7.218e-07
     126 1.434e-05 1.830e-05 1.590e-05 8.880e-07
     158 2.096e-05 2.697e-05 2.306e-05 1.258e-06
     200 3.109e-05 3.567e-05 3.315e-05 1.031e-06
     251 4.572e-05 5.072e-05 4.802e-05 1.162e-06
     316 6.678e-05 1.211e-04 7.332e-05 1.055e-05
     398 9.960e-05 1.073e-04 1.029e-04 1.626e-06
     501 1.482e-04 1.859e-04 1.543e-04 4.475e-06
     631 2.302e-04 3.082e-04 2.430e-04 1.357e-05
     794 3.508e-04 4.455e-04 3.690e-04 1.815e-05
    1000 5.619e-04 8.110e-04 6.374e-04 6.246e-05
    1259 8.762e-04 1.089e-03 9.406e-04 4.962e-05
    1585 1.342e-03 2.002e-03 1.490e-03 1.468e-04
    1995 2.142e-03 3.452e-03 2.445e-03 3.512e-04
    2512 3.440e-03 4.014e-03 3.605e-03 1.099e-04
    3162 5.910e-03 7.011e-03 6.142e-03 1.985e-04
    3981 1.021e-02 1.128e-02 1.058e-02 2.162e-04
    5012 1.819e-02 2.027e-02 1.893e-02 3.929e-04
    6310 3.129e-02 3.914e-02 3.291e-02 1.354e-03
    7943 5.403e-02 9.116e-02 5.758e-02 6.174e-03
   10000 9.375e-02 1.250e-01 1.042e-01 6.889e-03
#------- --------- --------- --------- ---------
```

Tabela 1: Tempos de execução do Bubble Sort.

# shaker_sort				
# n	min time	max time	avg time	std dev
10	5.290e-07	1.050e-06	7.277e-07	1.970e-07
13	7.580e-07	1.148e-06	9.284e-07	1.037e-07
16	9.600e-07	1.354e-06	1.102e-06	8.720e-08
20	1.090e-06	1.482e-06	1.218e-06	7.445e-08
25	1.192e-06	1.672e-06	1.375e-06	1.055e-07
32	1.565e-06	1.974e-06	1.754e-06	9.247e-08
40	1.861e-06	2.329e-06	2.095e-06	1.110e-07
50	2.412e-06	3.163e-06	2.744e-06	1.768e-07
63	3.696e-06	6.726e-06	5.316e-06	7.194e-07
79	5.763e-06	7.481e-06	6.560e-06	4.454e-07
100	7.607e-06	1.153e-05	8.573e-06	7.858e-07
126	1.150e-05	2.118e-05	1.377e-05	1.835e-06
158	1.728e-05	2.854e-05	2.141e-05	2.830e-06
200	2.548e-05	3.807e-05	2.859e-05	2.521e-06
251	3.879e-05	5.938e-05	4.349e-05	4.438e-06
316	5.814e-05	8.772e-05	6.471e-05	7.109e-06
398	8.648e-05	1.246e-04	9.310e-05	7.555e-06
501	1.336e-04	1.953e-04	1.489e-04	1.428e-05
631	2.055e-04	3.097e-04	2.257e-04	2.274e-05
794	3.131e-04	4.701e-04	3.441e-04	3.330e-05
1000	4.977e-04	7.261e-04	5.479e-04	5.105e-05
1259	7.792e-04	1.101e-03	8.496e-04	7.449e-05
1585	1.238e-03	1.711e-03	1.358e-03	1.168e-04
1995	2.026e-03	2.832e-03	2.253e-03	2.032e-04
2512	3.353e-03	4.556e-03	3.711e-03	3.140e-04
3162	5.604e-03	8.048e-03	6.322e-03	5.595e-04
3981	9.408e-03	1.286e-02	1.050e-02	8.247e-04
5012	1.596e-02	2.024e-02	1.740e-02	9.998e-04
6310	2.669e-02	3.244e-02	2.916e-02	1.209e-03
7943	4.671e-02	5.253e-02	4.868e-02	1.381e-03
10000	7.609e-02	8.590e-02	8.092e-02	2.107e-03

Tabela 2: Tempos de execução do Shaker Sort.

# insertion_sort				
# n	min time	max time	avg time	std dev
10	4.990e-07	7.400e-07	5.727e-07	8.420e-08
13	6.670e-07	9.050e-07	7.204e-07	4.865e-08
16	6.650e-07	8.680e-07	7.154e-07	3.638e-08
20	6.970e-07	9.490e-07	7.781e-07	4.090e-08
25	7.310e-07	8.350e-07	7.809e-07	2.565e-08
32	8.340e-07	9.560e-07	8.822e-07	2.707e-08
40	9.130e-07	1.333e-06	1.090e-06	1.515e-07
50	1.094e-06	1.414e-06	1.230e-06	1.018e-07
63	1.394e-06	2.086e-06	1.646e-06	2.320e-07
79	1.946e-06	4.510e-06	3.185e-06	6.595e-07
100	3.131e-06	4.798e-06	3.455e-06	2.471e-07
126	3.614e-06	4.635e-06	4.073e-06	1.987e-07
158	4.770e-06	6.240e-06	5.218e-06	3.302e-07
200	6.889e-06	1.010e-05	7.533e-06	5.873e-07
251	1.077e-05	2.016e-05	1.392e-05	1.872e-06
316	1.531e-05	1.907e-05	1.656e-05	9.100e-07
398	2.317e-05	3.804e-05	2.523e-05	2.554e-06
501	3.585e-05	5.237e-05	4.009e-05	3.711e-06
631	5.542e-05	8.872e-05	6.284e-05	8.058e-06
794	8.540e-05	1.276e-04	9.225e-05	7.241e-06
1000	1.315e-04	1.938e-04	1.484e-04	1.615e-05
1259	2.053e-04	3.028e-04	2.265e-04	2.099e-05
1585	3.264e-04	4.695e-04	3.615e-04	3.422e-05
1995	5.101e-04	7.233e-04	5.644e-04	5.202e-05
2512	8.104e-04	1.190e-03	9.031e-04	9.135e-05
3162	1.272e-03	1.868e-03	1.432e-03	1.520e-04
3981	2.006e-03	2.876e-03	2.244e-03	2.235e-04
5012	3.190e-03	4.467e-03	3.598e-03	3.374e-04
6310	5.160e-03	7.188e-03	5.824e-03	5.018e-04
7943	8.033e-03	1.101e-02	8.994e-03	7.323e-04
10000	1.295e-02	2.158e-02	1.474e-02	1.460e-03
12589	2.045e-02	2.554e-02	2.210e-02	1.120e-03
15849	3.300e-02	3.806e-02	3.484e-02	1.283e-03
19953	5.276e-02	8.256e-02	5.725e-02	4.261e-03
25119	8.278e-02	1.012e-01	9.005e-02	4.221e-03

Tabela 3: Tempos de execução do Insertion Sort.

```
# Shell_sort
#      n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
       10 5.230e-07 8.780e-07 7.006e-07 1.312e-07
       13 6.930e-07 1.153e-06 8.397e-07 1.210e-07
       16 8.070e-07 1.154e-06 9.694e-07 1.093e-07
       20 9.720e-07 1.130e-06 1.039e-06 3.480e-08
       25 1.106e-06 1.289e-06 1.189e-06 4.281e-08
       32 1.004e-06 1.900e-06 1.414e-06 2.874e-07
       40 1.253e-06 2.256e-06 1.694e-06 3.316e-07
       50 1.986e-06 2.867e-06 2.308e-06 1.610e-07
       63 2.314e-06 2.623e-06 2.463e-06 7.312e-08
       79 2.568e-06 3.395e-06 2.800e-06 1.741e-07
      100 3.049e-06 4.866e-06 3.496e-06 5.024e-07
      126 3.781e-06 5.201e-06 4.255e-06 3.884e-07
      158 4.944e-06 8.147e-06 5.505e-06 7.927e-07
      200 6.542e-06 1.183e-05 7.337e-06 1.077e-06
      251 8.508e-06 1.208e-05 9.453e-06 8.260e-07
      316 1.111e-05 1.734e-05 1.179e-05 8.524e-07
      398 1.464e-05 2.250e-05 1.569e-05 1.244e-06
      501 1.921e-05 2.651e-05 2.190e-05 2.239e-06
      631 2.513e-05 3.202e-05 2.612e-05 1.198e-06
      794 3.269e-05 4.088e-05 3.390e-05 1.369e-06
     1000 4.193e-05 5.456e-05 4.407e-05 2.320e-06
     1259 5.483e-05 6.810e-05 5.619e-05 1.728e-06
     1585 7.191e-05 7.421e-05 7.289e-05 4.915e-07
     1995 9.174e-05 1.311e-04 9.703e-05 7.359e-06
     2512 1.225e-04 1.596e-04 1.309e-04 9.217e-06
     3162 1.592e-04 2.137e-04 1.698e-04 1.260e-05
     3981 2.077e-04 2.512e-04 2.145e-04 9.136e-06
     5012 2.651e-04 3.074e-04 2.743e-04 6.667e-06
     6310 3.479e-04 4.285e-04 3.599e-04 1.286e-05
     7943 4.494e-04 6.200e-04 4.831e-04 3.941e-05
    10000 5.811e-04 6.570e-04 5.936e-04 1.215e-05
    12589 7.590e-04 1.328e-03 8.118e-04 9.730e-05
    15849 1.008e-03 1.321e-03 1.067e-03 7.072e-05
    19953 1.302e-03 2.487e-03 1.561e-03 3.051e-04
    25119 1.703e-03 3.383e-03 2.023e-03 4.275e-04
    31623 2.235e-03 4.372e-03 3.136e-03 5.964e-04
    39811 2.884e-03 6.447e-03 4.107e-03 1.097e-03
    50119 3.682e-03 5.062e-03 3.923e-03 3.040e-04
    63096 4.766e-03 6.174e-03 5.063e-03 3.028e-04
    79433 6.164e-03 1.147e-02 6.842e-03 1.203e-03
   100000 8.012e-03 9.148e-03 8.315e-03 2.650e-04
   125893 1.039e-02 1.135e-02 1.064e-02 1.911e-04
   158489 1.356e-02 2.363e-02 1.486e-02 1.950e-03
   199526 1.750e-02 2.097e-02 1.816e-02 5.941e-04
   251189 2.269e-02 2.948e-02 2.398e-02 1.594e-03
   316228 2.946e-02 3.753e-02 3.091e-02 1.651e-03
   398107 3.820e-02 4.445e-02 3.969e-02 1.295e-03
   501187 4.956e-02 5.617e-02 5.141e-02 1.411e-03
   630957 6.457e-02 1.134e-01 7.176e-02 9.648e-03
#------- --------- --------- --------- ---------
```

Tabela 4: Tempos de execução do Shell Sort.

```
# quick_sort
#       n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
       10 4.960e-07 5.960e-07 5.483e-07 2.848e-08
       13 6.300e-07 8.510e-07 7.979e-07 2.831e-08
       16 7.060e-07 9.880e-07 8.278e-07 1.005e-07
       20 8.240e-07 1.169e-06 9.906e-07 1.222e-07
       25 9.280e-07 1.250e-06 1.067e-06 1.005e-07
       32 1.092e-06 1.814e-06 1.410e-06 1.957e-07
       40 1.217e-06 1.989e-06 1.530e-06 2.088e-07
       50 1.461e-06 2.446e-06 1.865e-06 2.351e-07
       63 1.871e-06 2.673e-06 2.202e-06 2.153e-07
       79 2.418e-06 3.904e-06 2.785e-06 3.641e-07
      100 2.722e-06 4.410e-06 3.119e-06 4.577e-07
      126 3.594e-06 5.678e-06 4.229e-06 4.517e-07
      158 4.673e-06 7.268e-06 5.601e-06 6.990e-07
      200 6.056e-06 1.015e-05 7.284e-06 9.487e-07
      251 7.735e-06 1.043e-05 8.693e-06 6.474e-07
      316 9.030e-06 1.489e-05 1.117e-05 1.230e-06
      398 1.177e-05 1.879e-05 1.404e-05 1.736e-06
      501 1.612e-05 2.458e-05 1.905e-05 2.115e-06
      631 1.900e-05 2.747e-05 2.122e-05 2.011e-06
      794 2.437e-05 3.501e-05 2.678e-05 2.348e-06
     1000 3.176e-05 4.362e-05 3.454e-05 2.771e-06
     1259 4.127e-05 6.196e-05 4.485e-05 4.267e-06
     1585 5.360e-05 6.823e-05 5.679e-05 3.303e-06
     1995 6.867e-05 9.166e-05 7.324e-05 5.096e-06
     2512 8.850e-05 1.279e-04 9.590e-05 8.659e-06
     3162 1.158e-04 1.513e-04 1.219e-04 6.210e-06
     3981 1.491e-04 2.013e-04 1.583e-04 1.104e-05
     5012 1.926e-04 2.482e-04 2.027e-04 1.196e-05
     6310 2.494e-04 3.124e-04 2.628e-04 1.512e-05
     7943 3.187e-04 4.037e-04 3.352e-04 1.673e-05
    10000 4.171e-04 4.984e-04 4.341e-04 1.629e-05
    12589 5.272e-04 6.149e-04 5.443e-04 1.328e-05
    15849 6.791e-04 8.275e-04 7.021e-04 2.279e-05
    19953 8.787e-04 1.113e-03 9.163e-04 3.878e-05
    25119 1.130e-03 1.373e-03 1.183e-03 6.274e-05
    31623 1.455e-03 1.824e-03 1.536e-03 9.575e-05
    39811 1.880e-03 2.275e-03 1.968e-03 1.044e-04
    50119 2.423e-03 2.955e-03 2.537e-03 1.331e-04
    63096 3.120e-03 3.883e-03 3.264e-03 1.685e-04
    79433 4.036e-03 4.874e-03 4.220e-03 1.612e-04
   100000 5.150e-03 6.218e-03 5.415e-03 3.039e-04
   125893 6.653e-03 7.805e-03 6.934e-03 2.048e-04
   158489 8.911e-03 1.291e-02 1.028e-02 1.014e-03
   199526 1.142e-02 1.458e-02 1.243e-02 7.058e-04
   251189 1.484e-02 1.919e-02 1.631e-02 1.054e-03
   316228 1.917e-02 2.374e-02 2.087e-02 1.148e-03
   398107 2.353e-02 3.197e-02 2.636e-02 1.724e-03
   501187 2.976e-02 3.780e-02 3.162e-02 1.767e-03
   630957 3.837e-02 7.824e-02 4.518e-02 9.885e-03
   794328 4.950e-02 7.145e-02 5.476e-02 3.726e-03
  1000000 6.249e-02 9.387e-02 6.555e-02 4.158e-03
#------- --------- --------- --------- ---------
```

Tabela 5: Tempos de execução do Quick Sort.

```
# merge_sort
#       n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
       10 4.870e-07 9.250e-07 5.383e-07 9.996e-08
       13 6.990e-07 1.167e-06 8.633e-07 1.157e-07
       16 7.300e-07 9.810e-07 7.946e-07 5.828e-08
       20 7.960e-07 1.115e-06 8.660e-07 6.715e-08
       25 8.520e-07 1.324e-06 1.085e-06 1.047e-07
       32 9.300e-07 1.082e-06 9.914e-07 3.215e-08
       40 1.058e-06 2.152e-06 1.495e-06 3.152e-07
       50 1.421e-06 2.279e-06 1.662e-06 1.815e-07
       63 1.652e-06 2.860e-06 2.090e-06 3.271e-07
       79 2.182e-06 3.121e-06 2.599e-06 1.901e-07
      100 2.900e-06 4.058e-06 3.207e-06 2.706e-07
      126 3.594e-06 6.190e-06 4.481e-06 5.518e-07
      158 4.806e-06 7.980e-06 5.908e-06 7.303e-07
      200 6.096e-06 9.736e-06 7.498e-06 8.194e-07
      251 7.683e-06 1.227e-05 9.131e-06 1.043e-06
      316 1.031e-05 1.887e-05 1.265e-05 2.049e-06
      398 1.202e-05 2.020e-05 1.426e-05 1.831e-06
      501 1.520e-05 2.792e-05 1.764e-05 2.406e-06
      631 2.019e-05 2.905e-05 2.206e-05 1.856e-06
      794 2.647e-05 3.731e-05 2.852e-05 2.551e-06
     1000 3.396e-05 4.460e-05 3.596e-05 2.433e-06
     1259 4.393e-05 6.705e-05 4.781e-05 5.677e-06
     1585 5.778e-05 7.551e-05 5.991e-05 3.378e-06
     1995 7.427e-05 9.171e-05 7.746e-05 3.913e-06
     2512 9.730e-05 1.182e-04 1.001e-04 3.631e-06
     3162 1.254e-04 1.506e-04 1.298e-04 4.140e-06
     3981 1.609e-04 2.001e-04 1.662e-04 5.419e-06
     5012 2.091e-04 2.414e-04 2.146e-04 5.432e-06
     6310 2.758e-04 3.189e-04 2.812e-04 6.227e-06
     7943 3.531e-04 3.930e-04 3.593e-04 5.754e-06
    10000 4.561e-04 5.024e-04 4.638e-04 7.765e-06
    12589 6.017e-04 6.483e-04 6.089e-04 8.474e-06
    15849 7.696e-04 8.115e-04 7.777e-04 8.122e-06
    19953 9.892e-04 1.072e-03 1.003e-03 1.418e-05
    25119 1.305e-03 1.550e-03 1.340e-03 6.620e-05
    31623 1.666e-03 1.983e-03 1.701e-03 7.578e-05
    39811 2.138e-03 2.550e-03 2.186e-03 9.860e-05
    50119 2.810e-03 3.329e-03 2.853e-03 9.989e-05
    63096 3.583e-03 4.235e-03 3.623e-03 9.348e-05
    79433 4.595e-03 5.386e-03 4.634e-03 7.219e-05
   100000 6.021e-03 9.839e-03 6.289e-03 6.625e-04
   125893 7.685e-03 1.042e-02 8.177e-03 5.500e-04
   158489 9.827e-03 1.061e-02 9.907e-03 1.128e-04
   199526 1.284e-02 1.323e-02 1.289e-02 6.485e-05
   251189 1.636e-02 1.680e-02 1.643e-02 7.155e-05
   316228 2.091e-02 2.139e-02 2.101e-02 8.497e-05
   398107 2.728e-02 2.775e-02 2.735e-02 7.464e-05
   501187 3.470e-02 3.539e-02 3.482e-02 1.049e-04
   630957 4.438e-02 5.037e-02 4.472e-02 6.687e-04
   794328 5.775e-02 5.867e-02 5.791e-02 1.447e-04
  1000000 7.348e-02 7.682e-02 7.387e-02 5.952e-04
#------- --------- --------- --------- ---------
```

Tabela 6: Tempos de execução do Merge Sort.

```
# heap_sort
#       n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
       10 5.260e-07 8.930e-07 6.707e-07 1.378e-07
       13 8.150e-07 1.386e-06 1.052e-06 2.012e-07
       16 9.520e-07 1.398e-06 1.165e-06 1.240e-07
       20 9.880e-07 1.414e-06 1.120e-06 1.205e-07
       25 1.080e-06 1.621e-06 1.291e-06 1.475e-07
       32 1.324e-06 1.929e-06 1.676e-06 1.428e-07
       40 1.414e-06 2.273e-06 1.847e-06 2.530e-07
       50 2.110e-06 3.536e-06 2.619e-06 2.923e-07
       63 2.297e-06 3.649e-06 2.950e-06 4.169e-07
       79 2.897e-06 4.693e-06 3.756e-06 4.588e-07
      100 3.367e-06 5.729e-06 3.712e-06 4.010e-07
      126 4.386e-06 6.427e-06 4.782e-06 5.528e-07
      158 5.699e-06 9.159e-06 6.794e-06 9.548e-07
      200 7.411e-06 9.513e-06 7.849e-06 4.552e-07
      251 9.600e-06 1.360e-05 1.009e-05 8.035e-07
      316 1.309e-05 1.707e-05 1.407e-05 8.675e-07
      398 1.699e-05 1.893e-05 1.744e-05 3.850e-07
      501 2.217e-05 2.314e-05 2.256e-05 2.062e-07
      631 2.879e-05 3.243e-05 2.966e-05 9.017e-07
      794 3.638e-05 4.128e-05 3.724e-05 1.155e-06
     1000 4.664e-05 4.934e-05 4.779e-05 4.458e-07
     1259 5.986e-05 6.192e-05 6.054e-05 3.567e-07
     1585 7.755e-05 8.250e-05 7.854e-05 8.556e-07
     1995 1.006e-04 1.296e-04 1.025e-04 3.774e-06
     2512 1.302e-04 1.374e-04 1.317e-04 1.300e-06
     3162 1.684e-04 1.771e-04 1.696e-04 8.411e-07
     3981 2.179e-04 2.564e-04 2.219e-04 5.810e-06
     5012 2.813e-04 2.905e-04 2.835e-04 2.011e-06
     6310 3.633e-04 3.705e-04 3.653e-04 1.203e-06
     7943 4.696e-04 5.162e-04 4.750e-04 6.521e-06
    10000 6.067e-04 6.688e-04 6.107e-04 6.446e-06
    12589 7.840e-04 8.273e-04 7.879e-04 4.256e-06
    15849 1.014e-03 1.051e-03 1.018e-03 3.238e-06
    19953 1.309e-03 1.554e-03 1.344e-03 6.878e-05
    25119 1.691e-03 2.008e-03 1.731e-03 8.351e-05
    31623 2.190e-03 2.592e-03 2.231e-03 9.800e-05
    39811 2.830e-03 3.331e-03 2.864e-03 9.634e-05
    50119 3.662e-03 4.308e-03 3.694e-03 8.945e-05
    63096 4.738e-03 5.544e-03 4.763e-03 6.072e-05
    79433 6.119e-03 6.541e-03 6.148e-03 4.795e-05
   100000 7.928e-03 8.274e-03 7.958e-03 4.753e-05
   125893 1.028e-02 1.056e-02 1.031e-02 4.004e-05
   158489 1.332e-02 1.389e-02 1.339e-02 1.132e-04
   199526 1.727e-02 1.799e-02 1.738e-02 1.744e-04
   251189 2.240e-02 2.289e-02 2.245e-02 7.243e-05
   316228 2.899e-02 2.961e-02 2.906e-02 9.189e-05
   398107 3.750e-02 3.801e-02 3.758e-02 8.229e-05
   501187 4.849e-02 4.925e-02 4.859e-02 1.081e-04
   630957 6.260e-02 6.356e-02 6.273e-02 1.517e-04
#------- --------- --------- --------- ---------
```

Tabela 7: Tempos de execução do Heap Sort.

```
# rank_sort
#      n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
      10 4.800e-07 6.830e-07 5.700e-07 7.989e-08
      13 6.380e-07 8.720e-07 7.148e-07 8.882e-08
      16 7.750e-07 9.310e-07 8.210e-07 5.554e-08
      20 8.460e-07 1.290e-06 1.041e-06 9.107e-08
      25 9.200e-07 1.600e-06 1.143e-06 2.290e-07
      32 1.121e-06 1.739e-06 1.323e-06 2.250e-07
      40 1.311e-06 2.176e-06 1.694e-06 2.256e-07
      50 1.692e-06 2.746e-06 2.029e-06 3.040e-07
      63 2.785e-06 5.778e-06 3.282e-06 5.834e-07
      79 3.659e-06 6.596e-06 4.637e-06 9.522e-07
     100 5.045e-06 1.070e-05 6.028e-06 1.251e-06
     126 7.137e-06 1.171e-05 8.509e-06 1.144e-06
     158 1.072e-05 1.571e-05 1.225e-05 1.362e-06
     200 1.684e-05 2.639e-05 1.827e-05 1.569e-06
     251 2.607e-05 3.792e-05 2.804e-05 2.352e-06
     316 4.158e-05 5.506e-05 4.406e-05 2.836e-06
     398 6.500e-05 7.738e-05 6.753e-05 1.877e-06
     501 1.004e-04 1.179e-04 1.037e-04 2.704e-06
     631 1.583e-04 1.755e-04 1.629e-04 2.431e-06
     794 2.456e-04 2.898e-04 2.524e-04 6.776e-06
    1000 3.893e-04 4.086e-04 3.968e-04 3.982e-06
    1259 6.171e-04 6.668e-04 6.286e-04 7.361e-06
    1585 9.793e-04 1.058e-03 9.963e-04 1.148e-05
    1995 1.552e-03 1.862e-03 1.603e-03 7.599e-05
    2512 2.465e-03 2.928e-03 2.528e-03 1.015e-04
    3162 3.907e-03 4.620e-03 3.974e-03 1.042e-04
    3981 6.197e-03 6.750e-03 6.273e-03 6.378e-05
    5012 9.831e-03 1.022e-02 9.932e-03 6.289e-05
    6310 1.564e-02 1.620e-02 1.579e-02 1.004e-04
    7943 2.484e-02 2.549e-02 2.503e-02 1.189e-04
   10000 3.944e-02 4.020e-02 3.970e-02 1.472e-04
   12589 6.247e-02 6.363e-02 6.287e-02 2.348e-04
#------- --------- --------- --------- ---------
```

Tabela 8: Tempos de execução do Rank Sort.

```
# selection_sort
#      n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
      10 5.280e-07 9.510e-07 6.326e-07 5.989e-08
      13 7.410e-07 1.318e-06 9.582e-07 1.925e-07
      16 9.170e-07 1.259e-06 1.091e-06 1.015e-07
      20 1.117e-06 1.571e-06 1.336e-06 1.547e-07
      25 1.257e-06 1.514e-06 1.412e-06 6.455e-08
      32 1.718e-06 2.295e-06 1.900e-06 2.179e-07
      40 2.149e-06 3.081e-06 2.566e-06 3.193e-07
      50 3.113e-06 4.253e-06 3.666e-06 3.216e-07
      63 4.814e-06 5.894e-06 5.273e-06 3.217e-07
      79 6.619e-06 1.115e-05 8.280e-06 9.891e-07
     100 1.002e-05 1.425e-05 1.166e-05 1.180e-06
     126 1.533e-05 2.793e-05 1.674e-05 1.767e-06
     158 2.464e-05 3.778e-05 2.848e-05 3.824e-06
     200 3.990e-05 4.576e-05 4.079e-05 8.900e-07
     251 6.125e-05 6.989e-05 6.287e-05 1.392e-06
     316 9.697e-05 1.127e-04 9.871e-05 2.632e-06
     398 1.510e-04 1.621e-04 1.548e-04 1.418e-06
     501 2.389e-04 2.568e-04 2.428e-04 3.138e-06
     631 3.800e-04 4.013e-04 3.842e-04 4.265e-06
     794 6.035e-04 6.393e-04 6.082e-04 6.013e-06
    1000 9.601e-04 9.967e-04 9.652e-04 7.652e-06
    1259 1.526e-03 1.745e-03 1.555e-03 5.763e-05
    1585 2.425e-03 2.740e-03 2.457e-03 7.404e-05
    1995 3.850e-03 4.318e-03 3.873e-03 7.043e-05
    2512 6.116e-03 6.472e-03 6.132e-03 3.834e-05
    3162 9.706e-03 9.955e-03 9.726e-03 3.744e-05
    3981 1.541e-02 1.569e-02 1.543e-02 4.367e-05
    5012 2.445e-02 2.484e-02 2.448e-02 5.423e-05
    6310 3.879e-02 3.924e-02 3.884e-02 7.230e-05
    7943 6.151e-02 6.227e-02 6.162e-02 1.398e-04
#------- --------- --------- --------- ---------
```

Tabela 9: Tempos de execução do Selection Sort.

3.2. Gráficos dos tempos de execução

De forma a obter um gráfico dos tempos médios, para cada algoritmo de ordenação, usamos o gnuplot. Por exemplo, para construir o gráfico do Bubble Sort usamos os comandos retratados na Figura 1.

```
eva@eva-TUF-GAMING-FX504GD-FX80GD:~/Documentos/AED/A02/A02$ gnuplot

        G N U P L O T
        Version 5.2 patchlevel 2    last modified 2017-11-01

        Copyright (C) 1986-1993, 1998, 2004, 2007-2017
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'qt'
gnuplot> data_file="output.txt"
gnuplot> set title "Bubble Sort Times"
gnuplot> set logscale xy
gnuplot> set grid
gnuplot> set key left
gnuplot> set format xy "%0e"
gnuplot> set xlabel "n"
gnuplot> set ylabel "Tempo(s)"
gnuplot> plot data_file i 0 u 1:3 w p t 'Max Time' ,\
>     data_file i 0 u 1:4 w p t 'Avg Time' ,\
>     data_file i 0 u 1:2 w p t 'Min Time'
```

Figura 1: Comandos utilizados no gnuplot, para construir o gráfico dos tempos de execução do Bubble Sort

Fazemos os mesmos comandos para as restantes técnicas de ordenação, adequando o título do gráfico e o valor do plot que está a seguir ao i. Obtendo assim, um gráfico para cada algoritmo, em que o eixo das abscissas é o n e o das ordenadas representa o tempo em segundos. O símbolo '+' de cor roxa representa o tempo máximo de execução, o símbolo 'x' de cor verde simboliza o tempo médio, e por fim, o '*' de cor azul retrata o tempo mínimo.

Decidimos construir outro gráfico, em cada método de ordenação, onde se demonstra a aproximação da função de n (tamanho do array a ordenar) em y (tempo médio), para uma função conhecida do tipo A * n ^ 2 + B * n + C, ou do tipo A * nlog(n) + B*n + C .

Para tal, resolvemos criar um script "avg_time.sh"(Fig. 3). Ao executar este script é criado um ficheiro de nome "avg_time.txt", em que cada linha vai ter um valor de n, e de seguida, um valor médio de tempo de execução, isto para todas os tempos dos algoritmos apresentados em "output.txt".

Através do ficheiro “avg_time.txt” e do código matlab exposto em “leastSquaresFit.m” (Fig. 4), construímos nove gráficos (um para cada algoritmo). Cujo os eixos irão representar os valores de n e os tempos de execução, tal como é ilustrado nos respetivos gráficos. Os pontos vermelhos simbolizam os tempos médios de execução, e os círculos verdes os tempos da função aproximada.

Com este código matlab, fazemos também print do erro gerado em cada aproximação.

3.2.1. Bubble Sort

Analisando primeiramente os gráficos do Bubble Sort, vamos começar a interpretar o gráfico com apenas os tempos de execução (Gráfico 1).

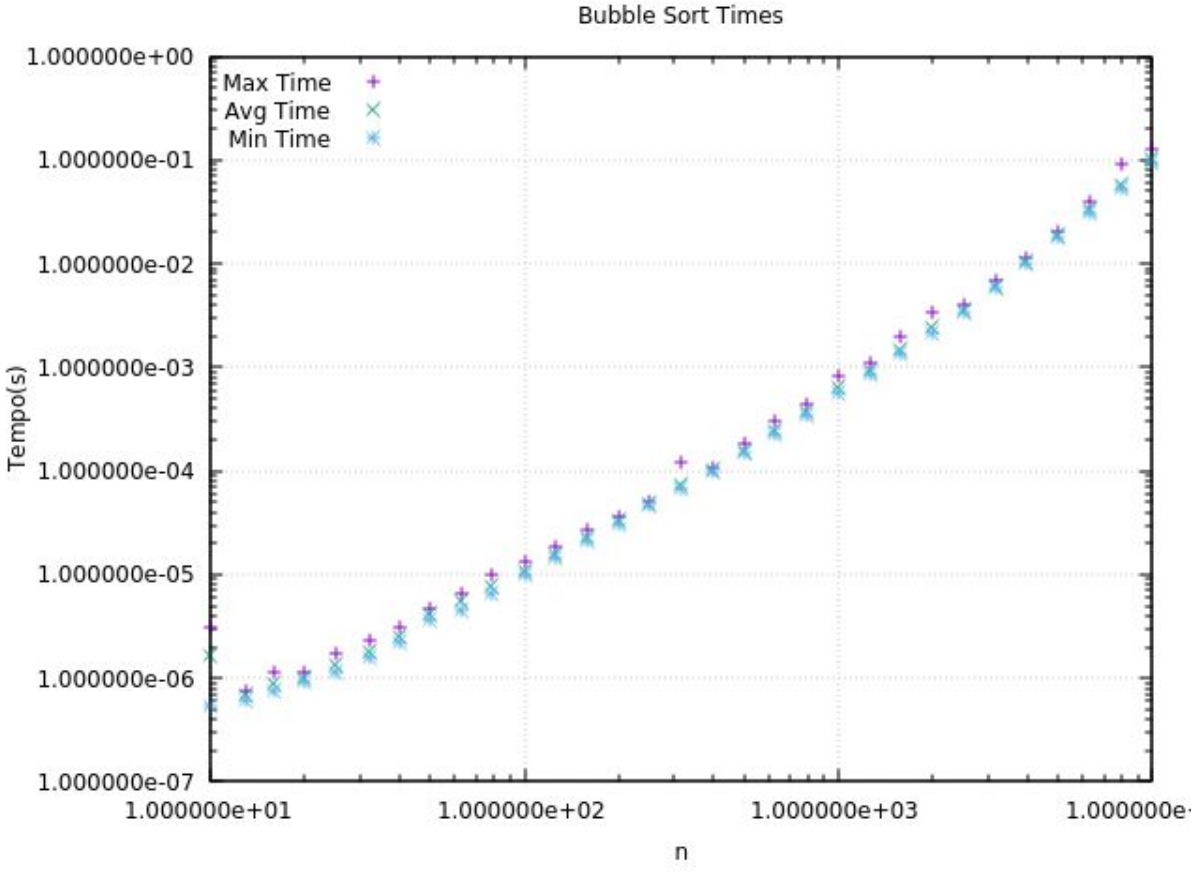

Gráfico 1: Tempos de execução do Bubble Sort

Neste gráfico é visível que nas funções, para todos os intervalos dos valores de n que sofrem um salto de expoente 1, vai corresponder, mais ou menos, a um salto de expoente 2 para os valores de tempos de execução. Por exemplo, neste gráfico visualizamos que no intervalo do eixo das abcissas de 10^2 a 10^3, os tempos sofrem aproximadamente uma deslocação de 10^{-5} a 10^{-3}.

Com base nesta análise, podemos dizer que estamos perante um algoritmo de complexidade computacional O(n^2). Portanto, à medida que o valor de n cresce, os tempos de execução, médio, mínimo e máximo, também crescem de forma quadrática, aparentando assim uma função de declive 2.

Observamos também que para valores de n maiores, os tempos mínimos e máximos não estão tão próximos dos tempos médios, como para valores de n menores. O que implica desvios padrões maiores para valores de n maiores, como podemos ver na tabela 1.

De seguida, vamos apresentar o outro gráfico, que apresenta a função de n em y e a função aproximada (Gráfico 2).

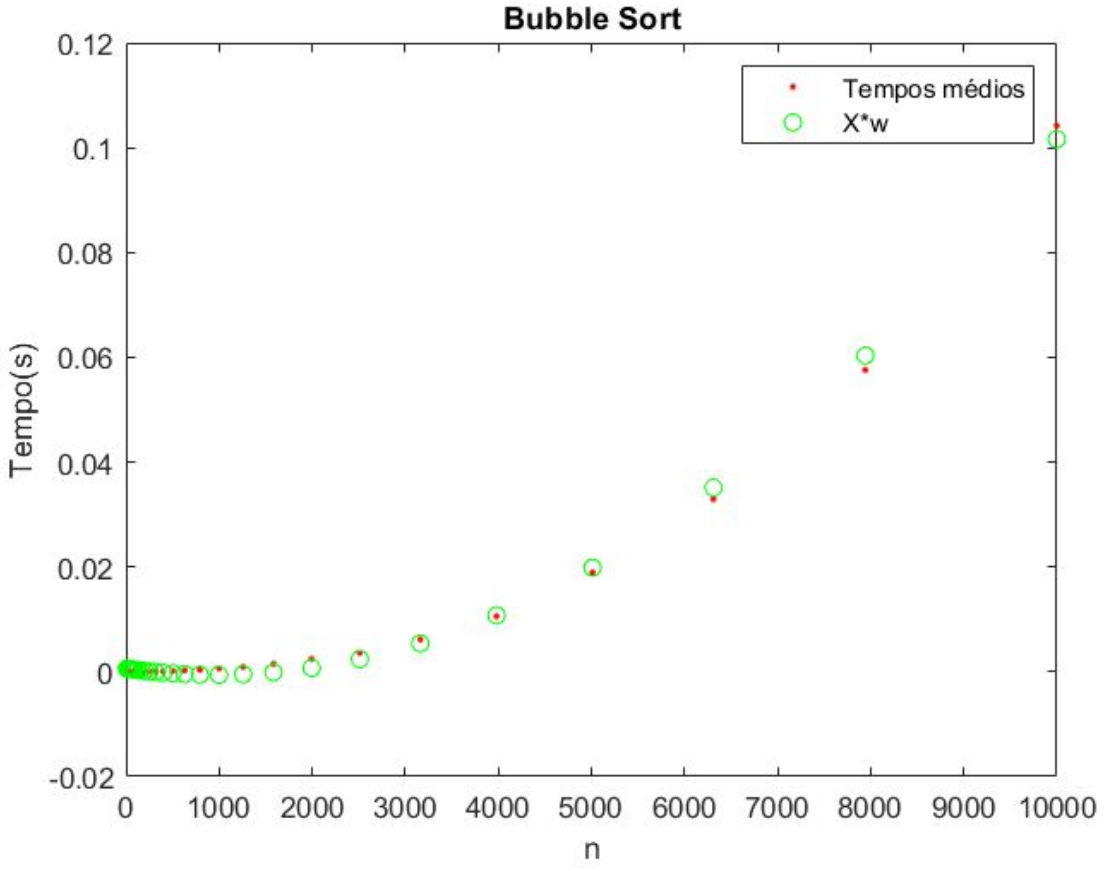

Gráfico 2: Função aproximada do Bubble Sort.

Nesta aproximação foi gerado um erro de 0.006016107346269, considerando-se assim uma aproximação razoável.

Fazendo zoom no gráfico no canto inferior esquerdo (Gráfico 3), observamos que a aproximação inicialmente vai melhorando, sendo muito boa em 3 pontos (aproximadamente nos intervalos 200 a 330 de n), mas depois vai piorando, voltando novamente a ser melhor só nos intervalos 4000 a 6500 de n.

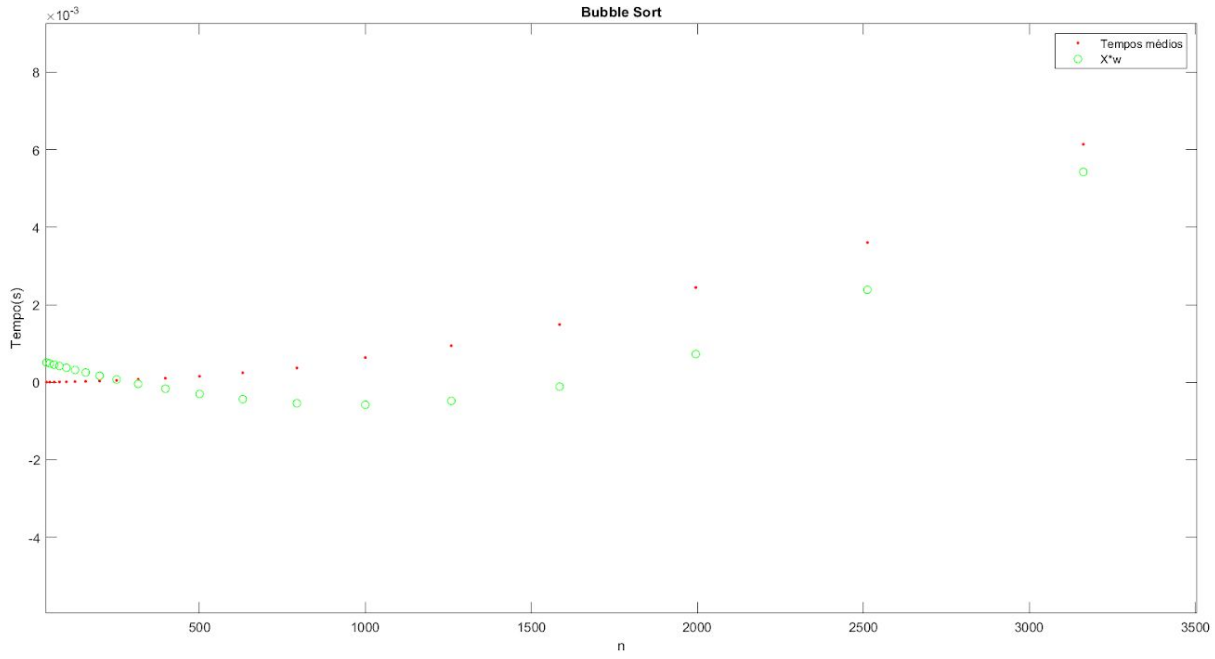

Gráfico 3: Zoom da função aproximada do Bubble Sort

3.2.2. Shaker Sort

Estudando agora o algoritmo Shaker Sort, podemos ver o seu gráfico dos tempos de execução no Gráfico 4.

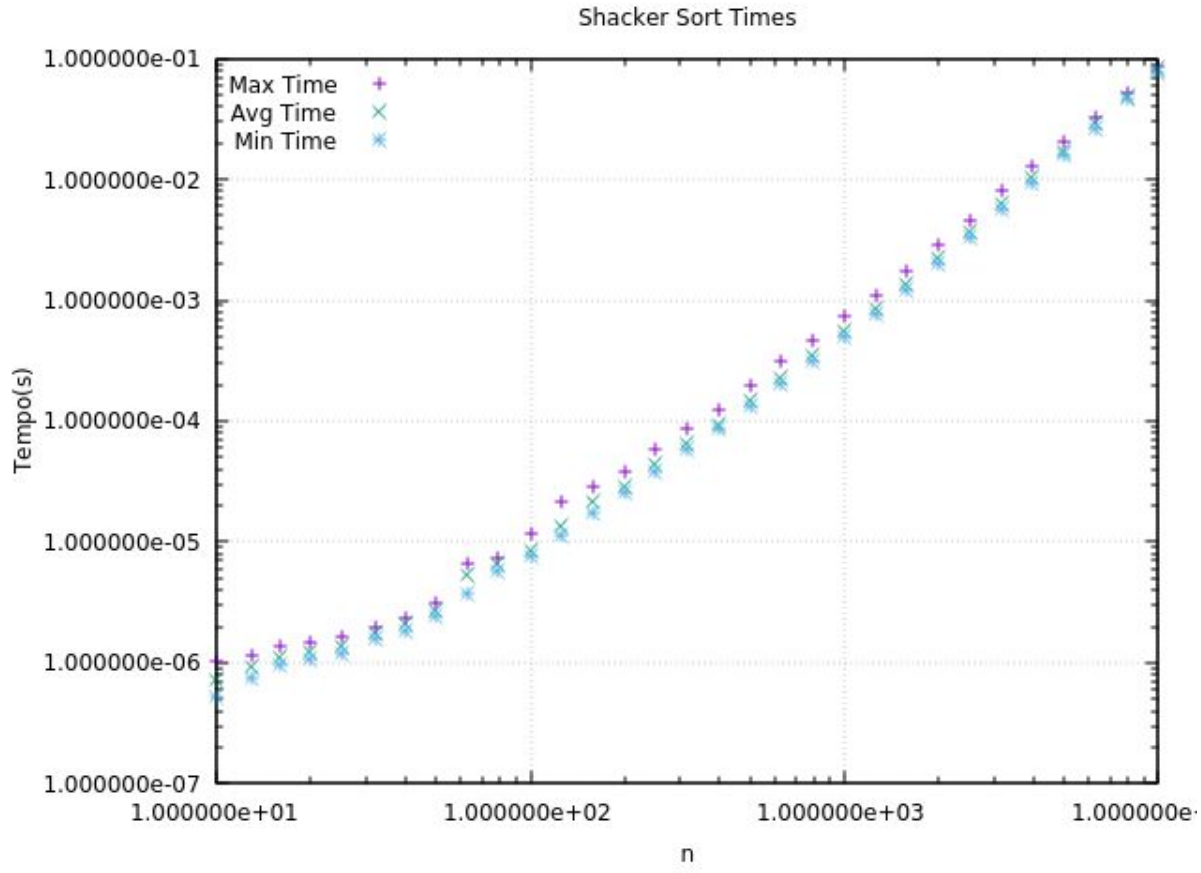

Gráfico 4: Tempos de execução do Shaker Sort.

No gráfico 4, tal como no outro algoritmo, observamos, mais ou menos, que as funções para todos os intervalos dos valores de n que avançam um salto de expoente 1, vai corresponder, a um avanço de expoente 2 para os valores de tempos de execução.

Podendo então dizer também que este algoritmo tem complexidade computacional O(n^2), isto é, à medida que o valor de n cresce, os tempos de execução, também crescem de forma quadrática com um declive cerca de 2.

Tal e qual como aconteceu no Bubble Sort, verificamos que para valores de n maiores, os tempos mínimos e máximos não estão tão próximos dos tempos médios, como para valores de n menores.

Para acabar de analisar o algoritmo Shaker Sort, no gráfico 5 é visível a função de n em y e a função aproximada.

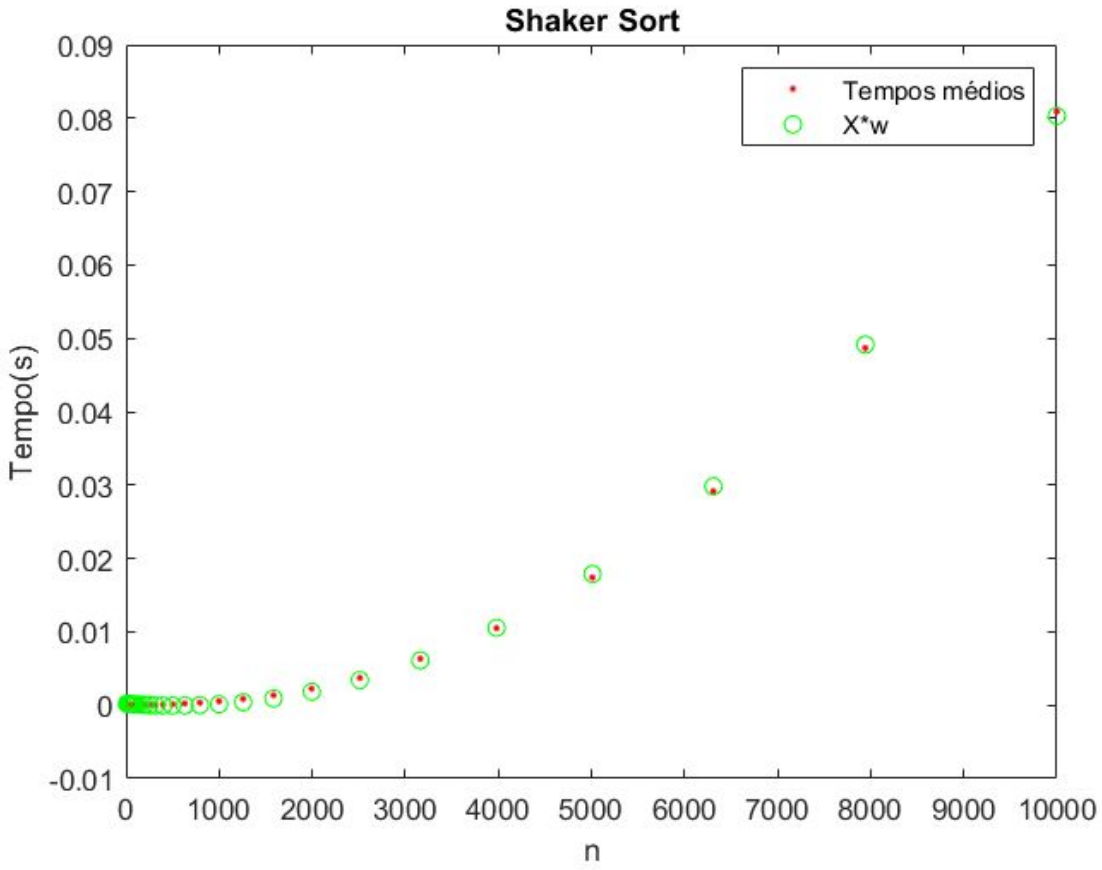

Gráfico 5: Função aproximada do Shaker Sort.

Nesta aproximação foi gerado um erro de 0.001666587187854, considerando-se assim uma aproximação boa.

Fazendo zoom no gráfico no canto inferior esquerdo(Gráfico 6), observamos que o processo de aproximação é equivalente ao Bubble Sort, mas este apresenta aproximações melhores. Inicialmente vai melhorando, aproximadamente nos intervalos 150 a 300 de n, posteriormente vai piorando, voltando novamente a ser melhor nos intervalos 3000 a 5000 de n.

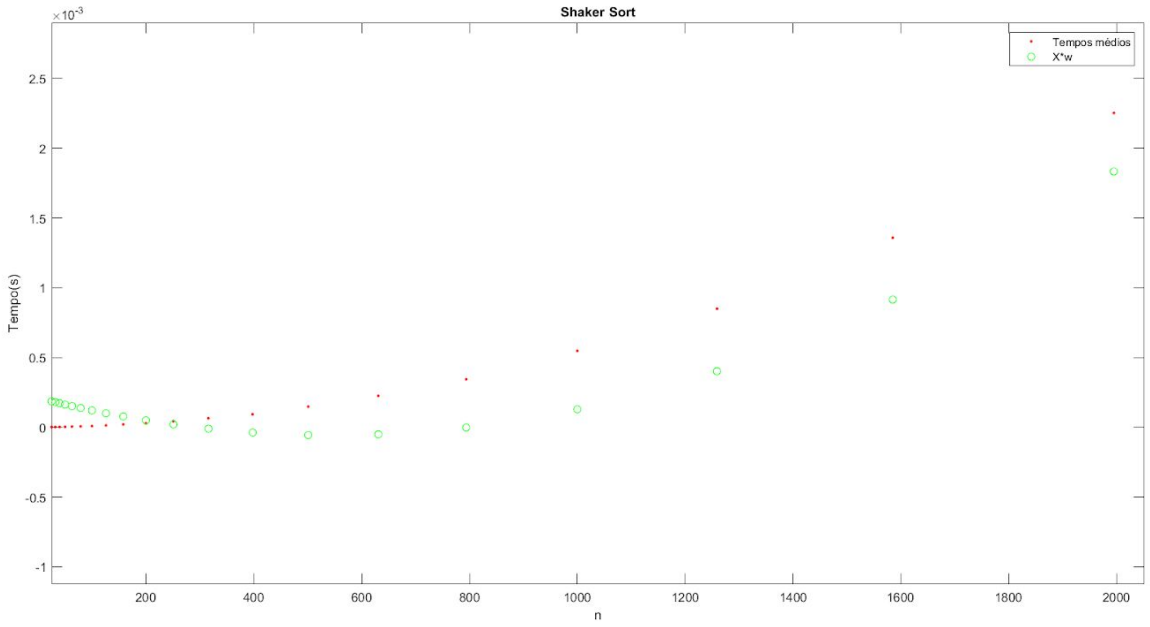

Gráfico 6: Zoom da função aproximada do ShakerSort

3.2.3. Insertion Sort

A seguir, mostramos o algoritmo Insertion Sort, começando pelo gráfico 7 com os tempos de execução deste algoritmo.

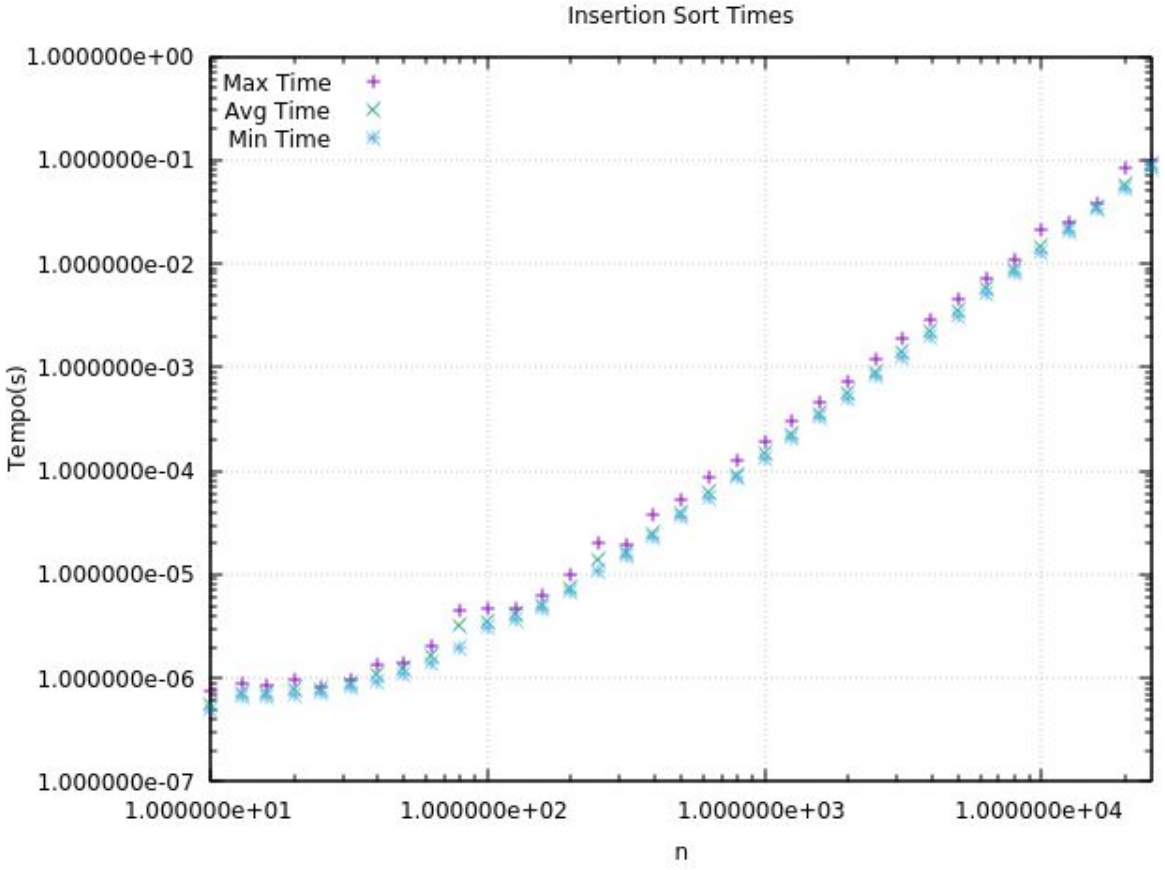

Gráfico 7: Tempos de execução do Insertion Sort.

Assim como os outros algoritmos analisados anteriormente, este apresenta também uma complexidade computacional O(n^2), portanto irá ter as mesmas conclusões que os outros. Só para relembrar, à medida que o valor de n cresce, os tempos de execução, também crescem de forma quadrática

Analisamos novamente, que para valores de n maiores, os tempos mínimos e máximos não estão tão próximos dos tempos médios, como para valores de n menores.

Para mostrar a função aproximada da função de n em y, temos o gráfico 8:

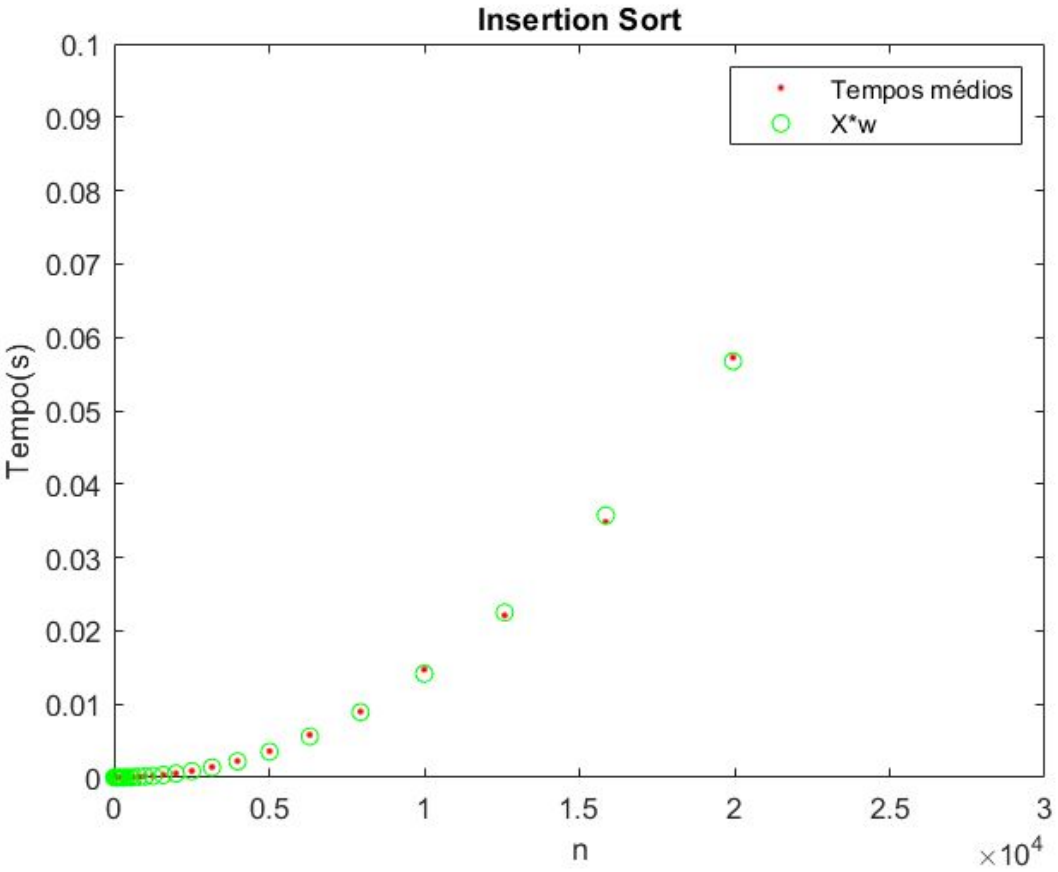

Gráfico 8: Função aproximada do Insertion Sort.

Nesta aproximação foi gerado um erro de 0.001278318328903, considerando-se assim uma aproximação boa.

Observamos que a aproximação é relativamente boa em todos os pontos, sendo pior nos últimos pontos, nos intervalos perto de 15000 a 23000 de n.

3.2.4. Shell Sort

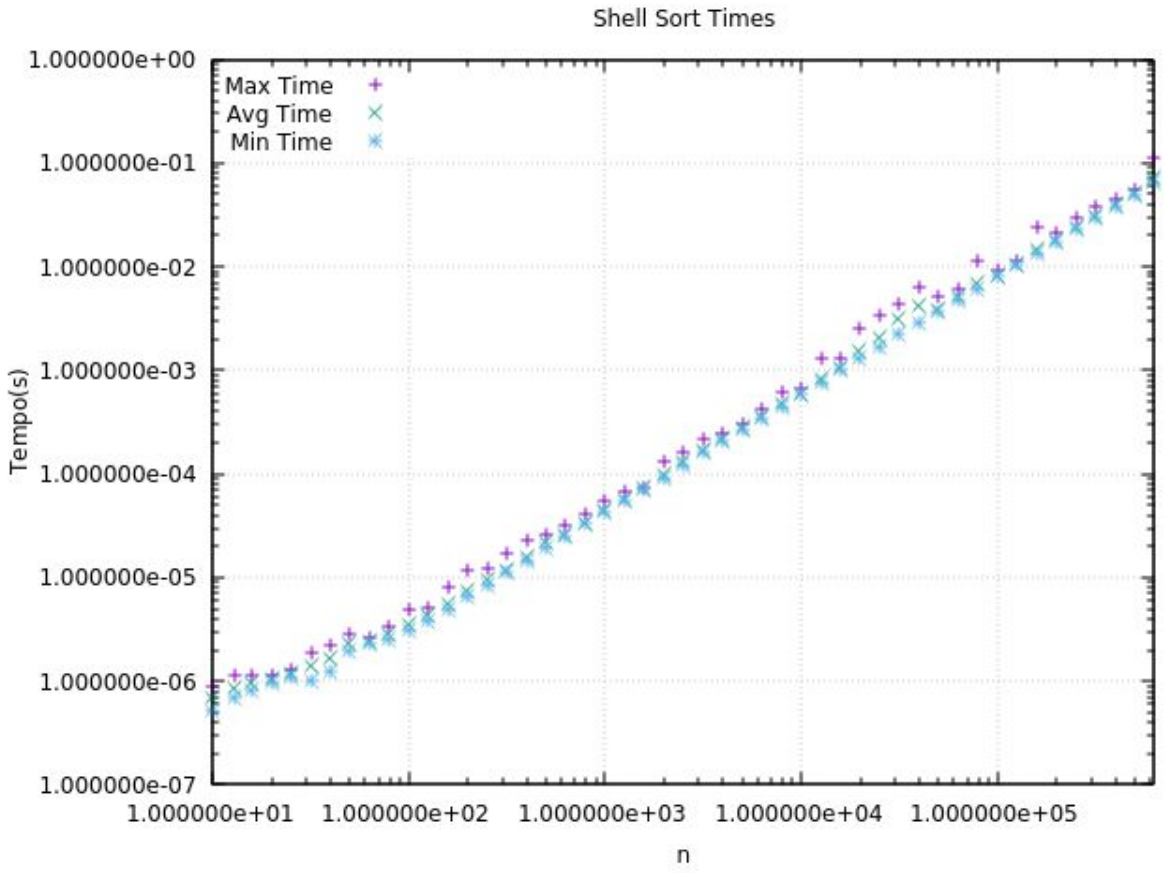

Gráfico 9: Tempos de execução do Shell Sort

Através da visualização do gráfico 9, podemos observar que a complexidade computacional do *Shell Sort* é de ordem "*n log n*" (O(n log n)), ou seja, à medida que o valor de *n* cresce, os valores dos tempos de execução também crescem exponencialmente com *n*.

Podemos observar que se trata de um algoritmo que cresce com O(n log n), para todos os casos (*worst, average* e *best*), visto que, por exemplo, no intervalo de 10^4 para 10^5, existe um salto de expoente 1, e para esse mesmo intervalo no eixo dos y, existe também para todos os casos, um salto de expoente 1, passando de valores um pouco inferiores a 10^{-3} para valores um pouco inferiores a 10^{-2}.

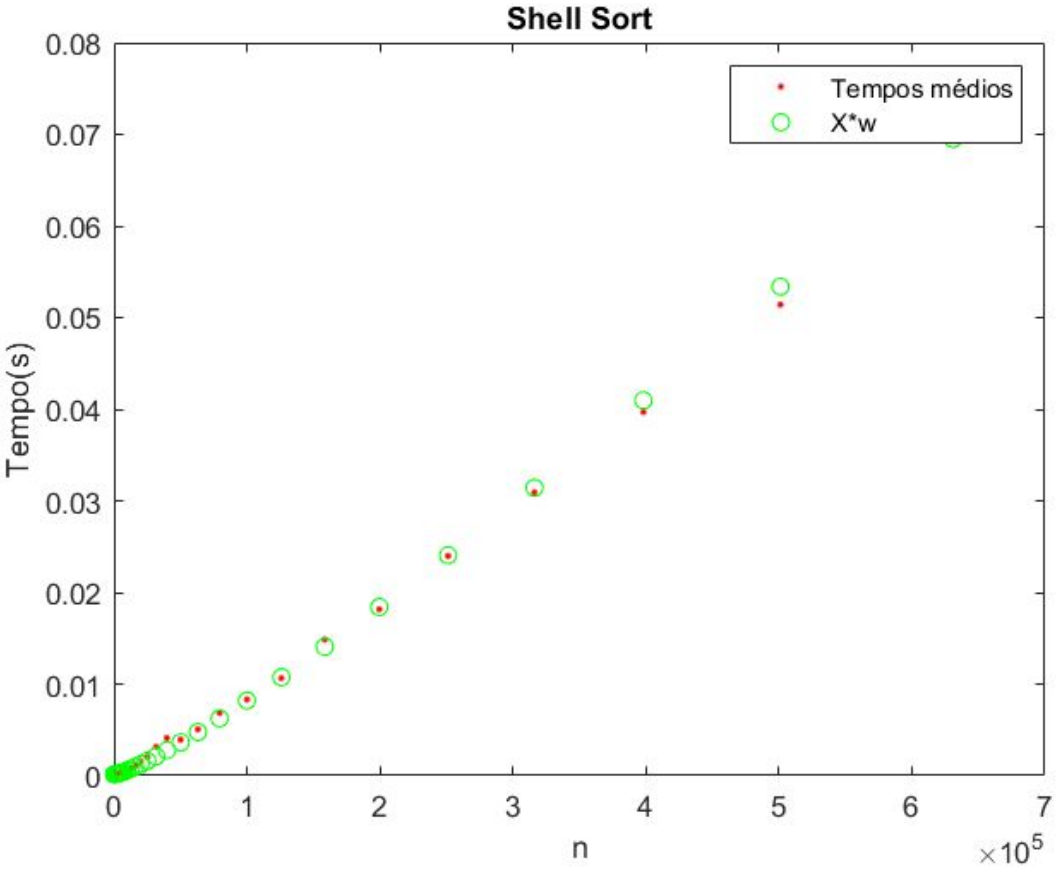

Gráfico 10: Função aproximada do Shell Sort

Nesta aproximação foi gerado um erro de 0.003973447184459, considerando-se assim uma boa aproximação, contudo pior que os algoritmos anteriores, como era de esperar, visto que, este algoritmo tem complexidade computacional de O(n log n), havendo mais discrepâncias a partir de valores de n superiores a 3* 10^5 **.**

Através do *zoom* no canto inferior esquerdo do gráfico 10, observamos que para o algoritmo *Shell sort*, existem várias incompatibilidades com a função utilizada, podendo observar esta conclusão através do gráfico 11, para valores um pouco inferiores a 0.5* 10^5 e para valores um pouco superiores a 1.5* 10^5, sendo estes os dois intervalos em que se verificam as piores discrepâncias.

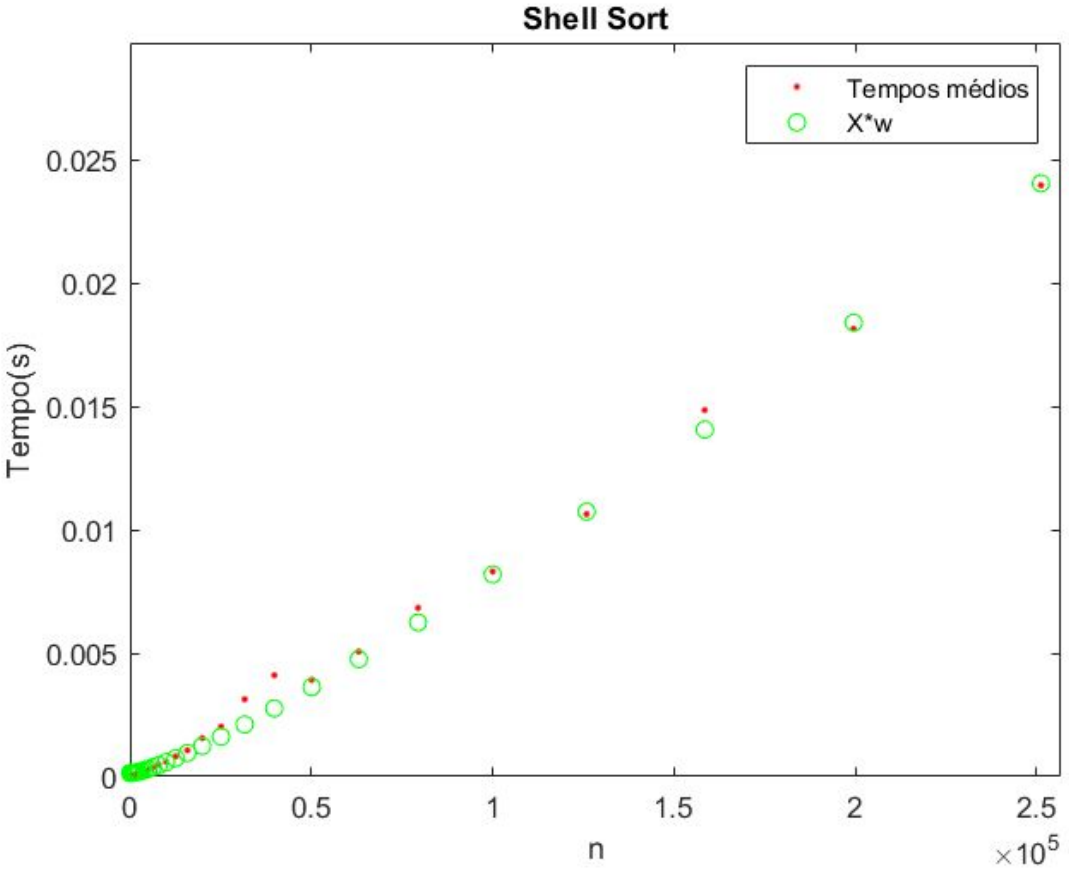

Gráfico 11: Zoom da função aproximada do Shell Sort

3.2.5. QuickSort

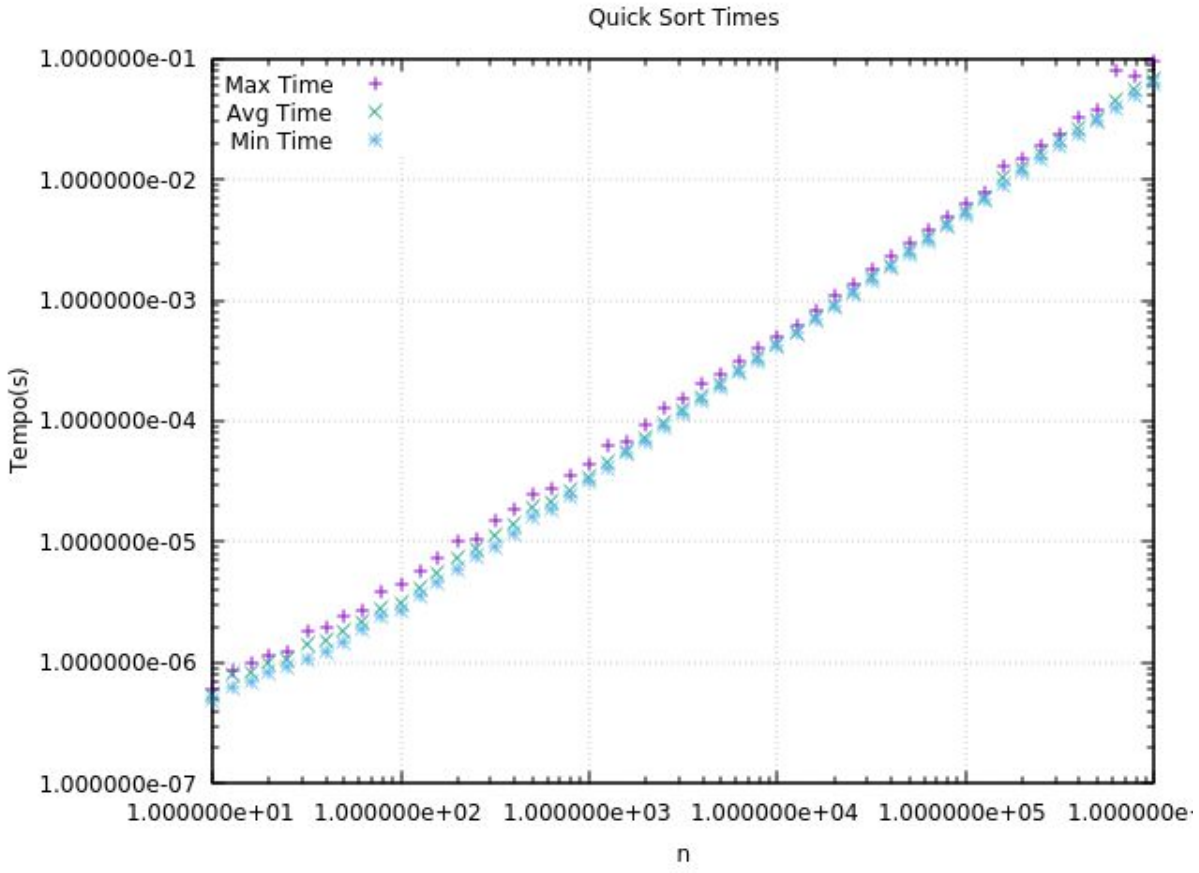

Gráfico 12: Tempos de execução do Quick Sort

Para o *QuickSort,* podemos observar que também se trata de um algoritmo que cresce exponencialmente à medida que o valor de *n* também cresce, podendo concluir que é um algoritmo de complexidade computacional de ordem "*n log n*" (O(n log n)).

Podemos observar essa conclusão através do gráfico 12, como por exemplo no intervalo de 10^4 para 10^5, existe um salto de expoente 1, e para esse mesmo intervalo no eixo dos y, existe também para todos os casos, um salto de expoente 1, passando de valores um pouco inferiores a 10^{-3} para valores um pouco inferiores a 10^{-2}.

Para este algoritmo podemos concluir que para valores de *n* até 10^3, conseguimos observar que os tempos de execução para todos os casos variam um pouco entre si, contudo para valores de *n* superiores a 10^3, para todos os casos os tempos são mais parecidos entre si, voltando a variar um pouco quando os valores de *n* começam a ser próximos de 10^6.

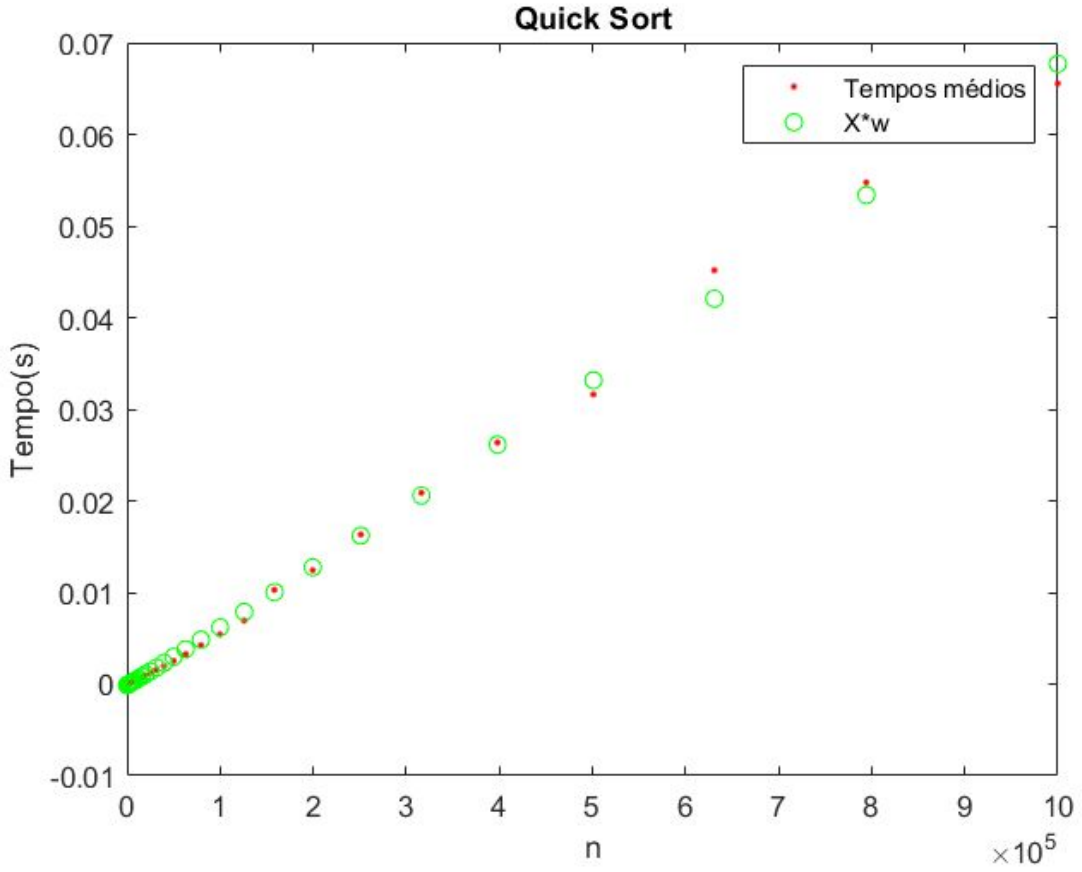

Gráfico 13: Função aproximada do Quick Sort

Nesta aproximação foi gerado um erro de 0.004675827185865, considerando-se assim uma boa aproximação, apresentando valores um pouco mais elevados que as anteriores, por ser também de complexidade computacional de O(n log n). Começando a notar-se mais discrepâncias para valores de n superiores a 5* 10^5 **.**

Podemos observar pelo gráfico 14, que os valores da aproximação da função utilizada para o algoritmo *Quick sort*, vêm comprovar os valores obtidos do erro, visto que podemos observar muitas divergências, começando apenas a ir para valores parecidos à função utilizada, para valores de n superiores a 2.5* 10^5 **, voltando a haver divergências a partir de valores de n superiores a 5*** 10^5 **, como podemos observar pelo gráfico 13.**

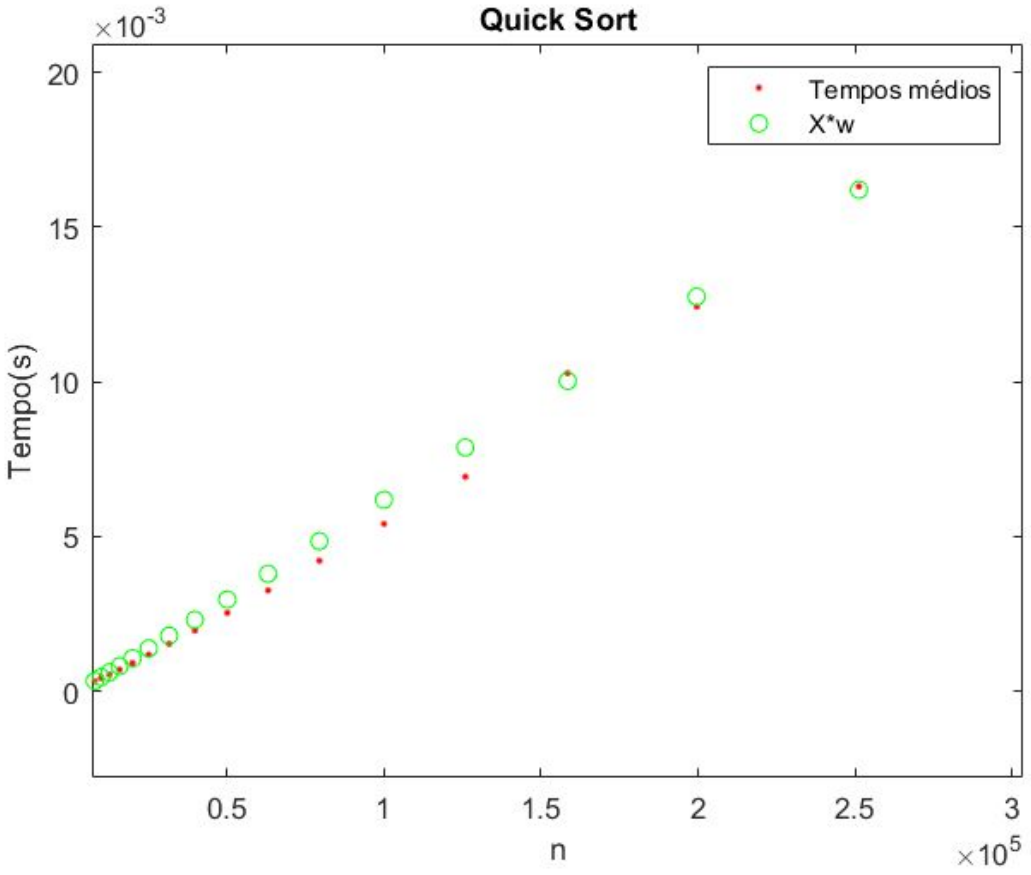

Gráfico 14: Zoom da função aproximada do QuickSort

3.2.6. Merge Sort

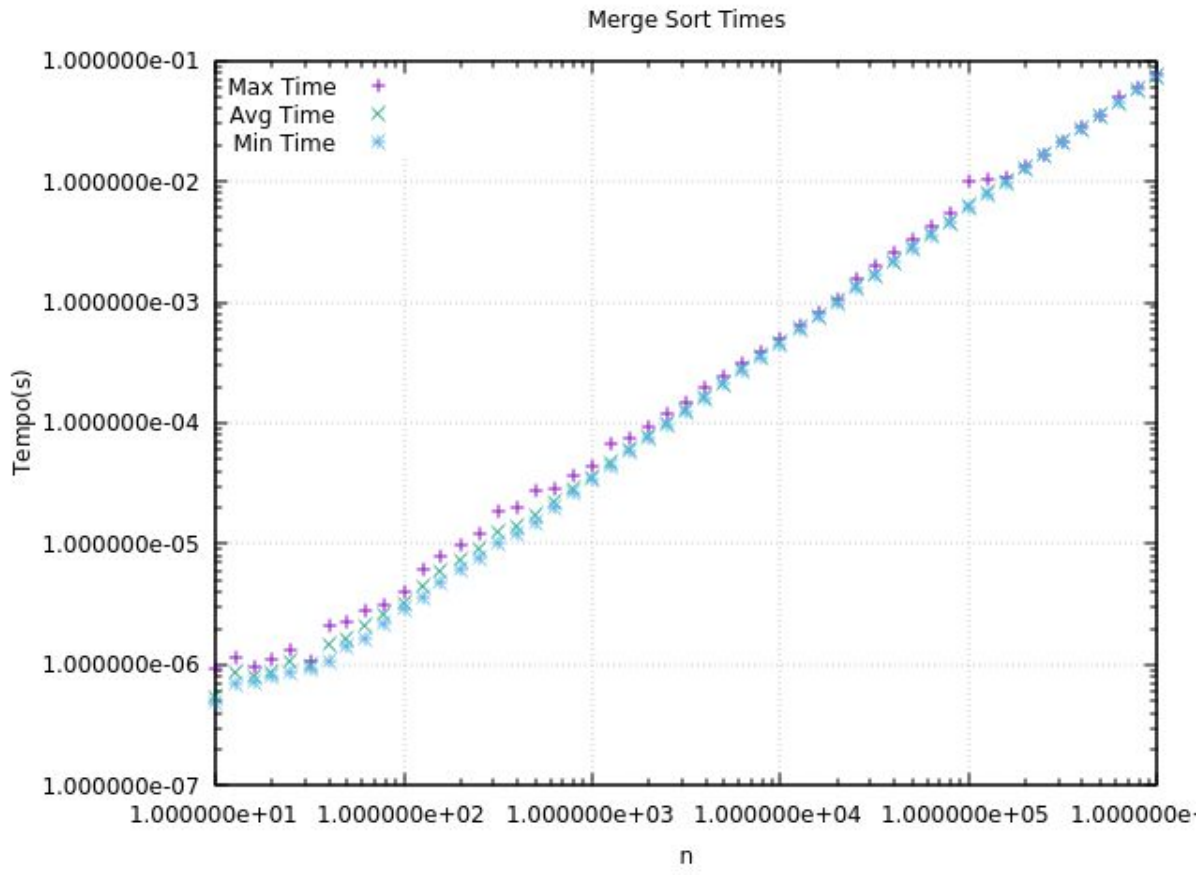

Gráfico 15: Tempos de execução do Merge Sort

No gráfico 15, podemos observar os tempos de execução do algoritmo *Merge sort,* onde podemos tirar conclusões acerca da complexidade computacional deste algoritmo.

Este que cresce exponencialmente à medida que o valor de *n* também cresce, podendo concluir que é um algoritmo de complexidade computacional de ordem "*n log n"* (O(n log n)).

Podemos observar essa conclusão através do gráfico 15, como por exemplo no intervalo de 10^4 para 10^5, existe um salto de expoente 1, e para esse mesmo intervalo no eixo dos y, existe também para todos os casos, um salto de expoente 1, passando de valores um pouco inferiores a 10^{-3} para valores um pouco inferiores a 10^{-2}.

Para este algoritmo podemos também chegar à conclusão que a partir de valores de *n* superiores a 10^3, os valores dos tempos de execução, para todos os casos, tomam valores muito próximos entre si.

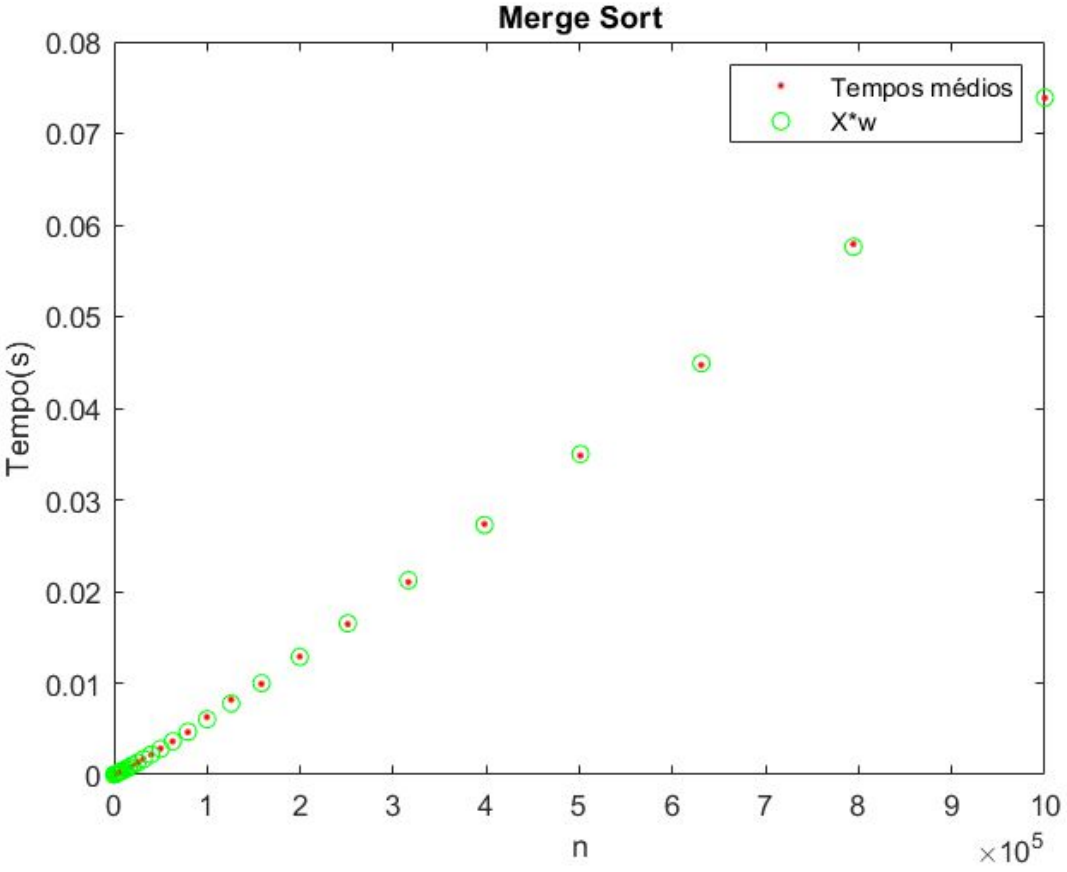

Gráfico 16: Função aproximada do Merge Sort

Nesta aproximação foi gerado um erro de 6.781947110418438e-04, considerando-se assim uma aproximação muito boa, obtendo-se valores inferiores aos obtidos anteriormente.

Para o *Merge sort*, não foi necessário fazer o *zoom,* pois podemos concluir através do gráfico 16 que as aproximações são muito boas, não havendo grandes divergências.

3.2.7. Heap Sort

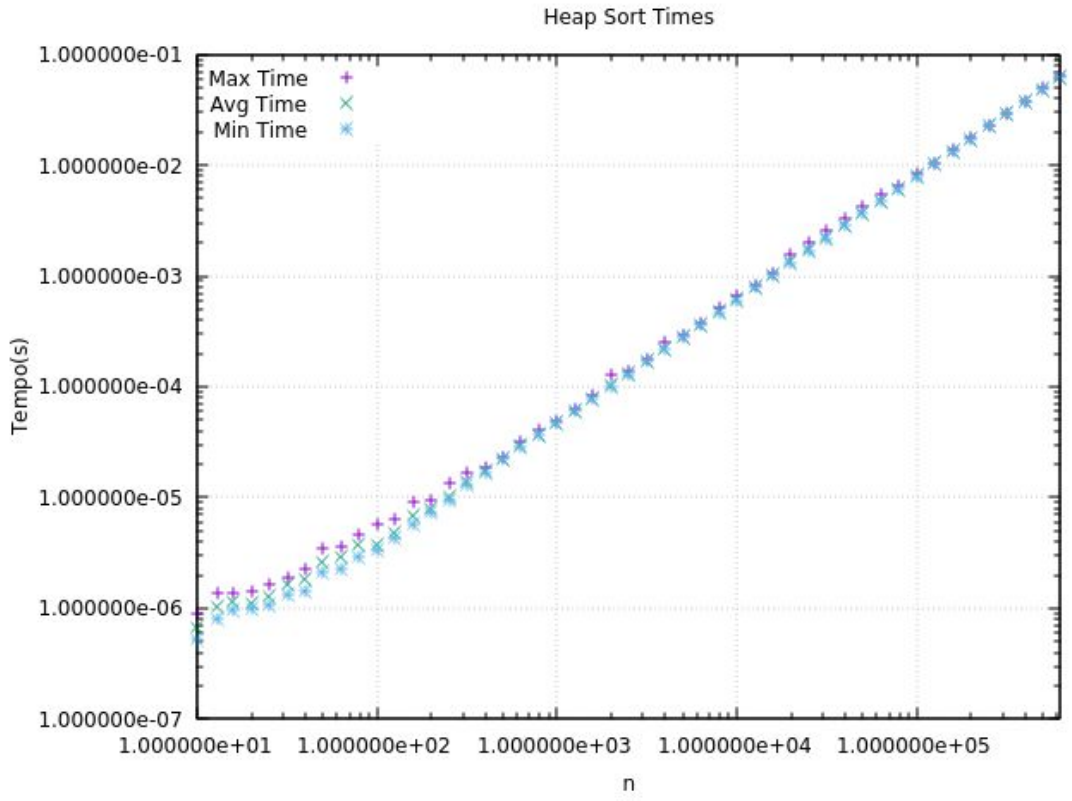

Gráfico 17: Tempos de execução do Heap Sort

Tal como por exemplo, o *Merge Sort* ou o *Quick Sort*, o *Heap Sort* é um algoritmo de ordenação que possui uma complexidade computacional *"n log n" (O(n log n)).*

No gráfico acima é possível verificar que o valor do tempo aumenta exponencialmente em função do valor de n.

No *Heap Sort,* a árvore é percorrida do topo para o final, para ser retirado um elemento da mesma. Sabendo que a altura da árvore de tamanho *n* é $log_2(n)$, na maior parte dos casos, significa que se o número de elementos duplicar a árvore ganha mais um nível de profundidade. Nesta seção do código a complexidade computacional é de *O(log n)*.

Como, este método é chamado *n* vezes, conclui-se que a complexidade computacional para reparar o *heap* é *O(n log n)*.

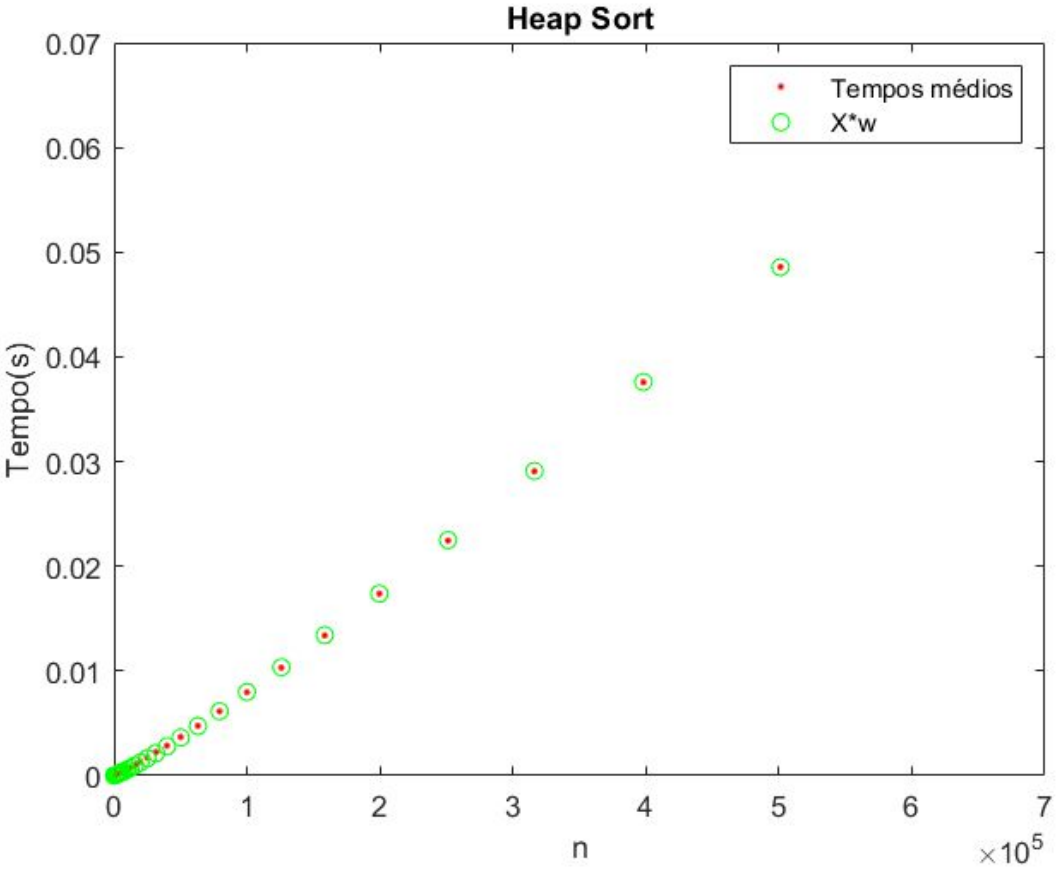

Gráfico 18: Função aproximada do Heap Sort

Nesta aproximação foi gerado um erro aproximado a 2.261971000825978e-04, sendo esta aproximação bastante positiva relativamente a algumas já referidas acima.

Em outros algoritmos foi necessário recorrer à ferramenta de *zoom* para visualizar melhor o erro obtido porém, neste caso, achamos desnecessário devido ao facto de o erro obtido ser relativamente mais baixo e estando no gráfico a aproximação bem representada.

3.2.8. Selection Sort

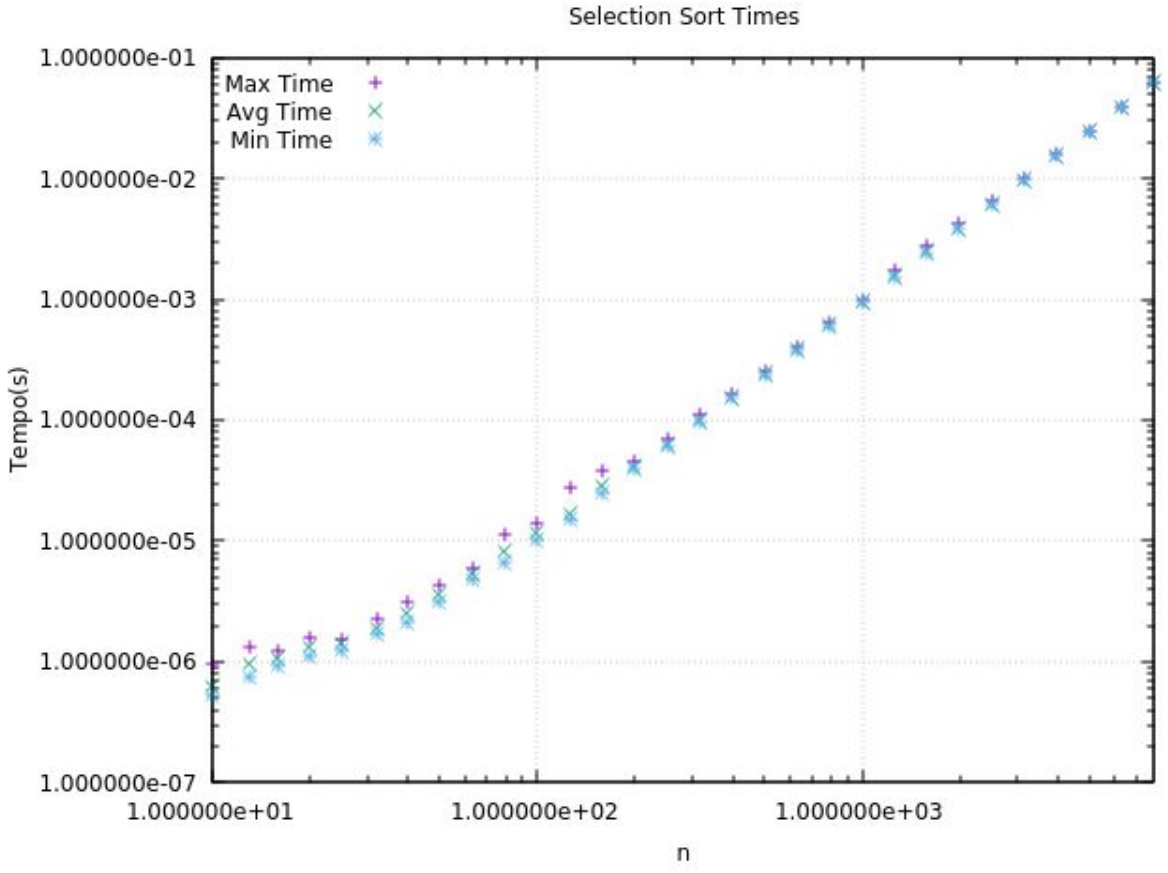

Gráfico 19: Tempos de execução do Selection Sort

No gráfico 19, tal como nos algoritmos *Bubble Sort, Shaker Sort* e *Insertion Sort*, observamos, mais ou menos, que as funções para todos os intervalos dos valores de n que avançam um salto de expoente 1, vai corresponder, a um avanço de expoente 2 para os valores de tempos de execução.

Então, conclui-se que este algoritmo tem complexidade computacional O(n^2), isto é, à medida que o valor de n cresce, os tempos de execução, também crescem de forma quadrática.

Como aconteceu nos algoritmos referidos acima, verificamos que para valores de n maiores, os tempos mínimos e máximos não estão tão próximos dos tempos médios, como para valores de n menores

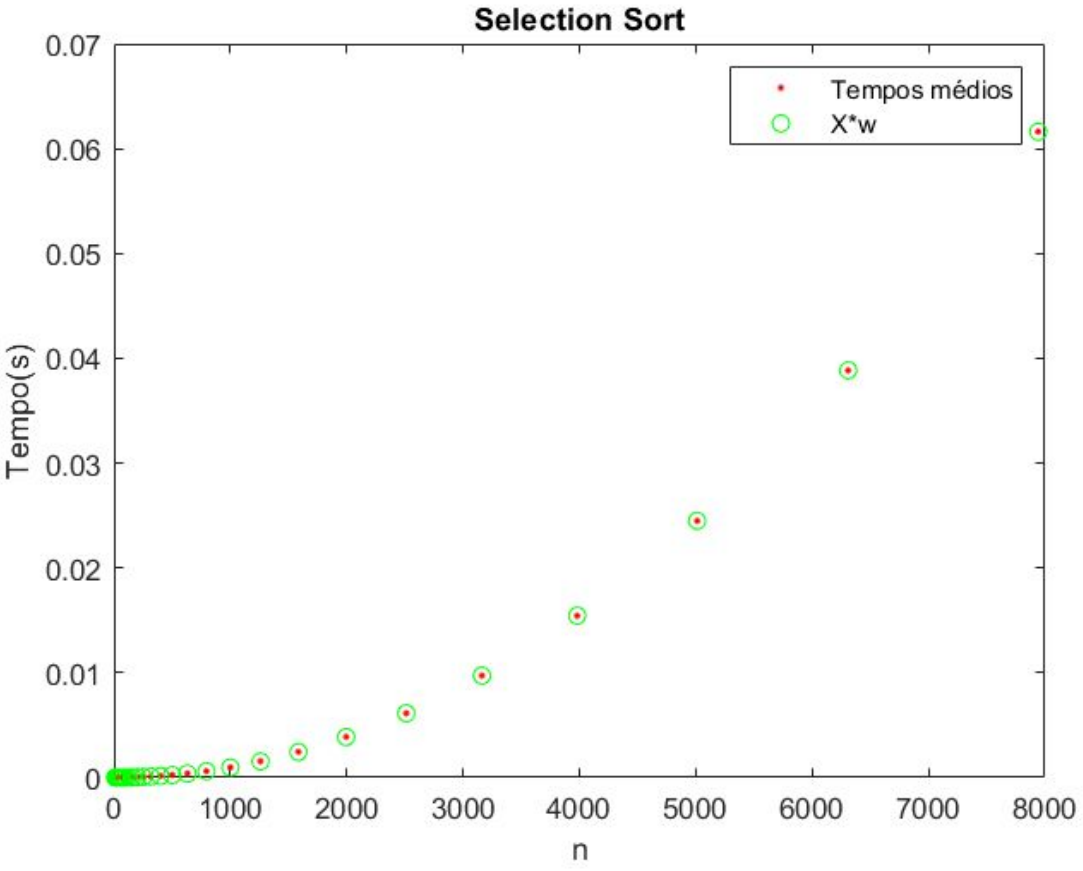

Gráfico 20: Função aproximada do Selection Sort

Nesta aproximação foi gerado um erro aproximado a 4.644135768901489e-05, sendo esta aproximação bastante positiva relativamente a algumas já referidas acima.

Também, neste algoritmo, achamos desnecessário recorrer à ferramenta *zoom* existente no Matlab devido ao facto de o erro obtido ser relativamente mais baixo e estando no gráfico a aproximação bem representada.

3.2.9. Rank Sort

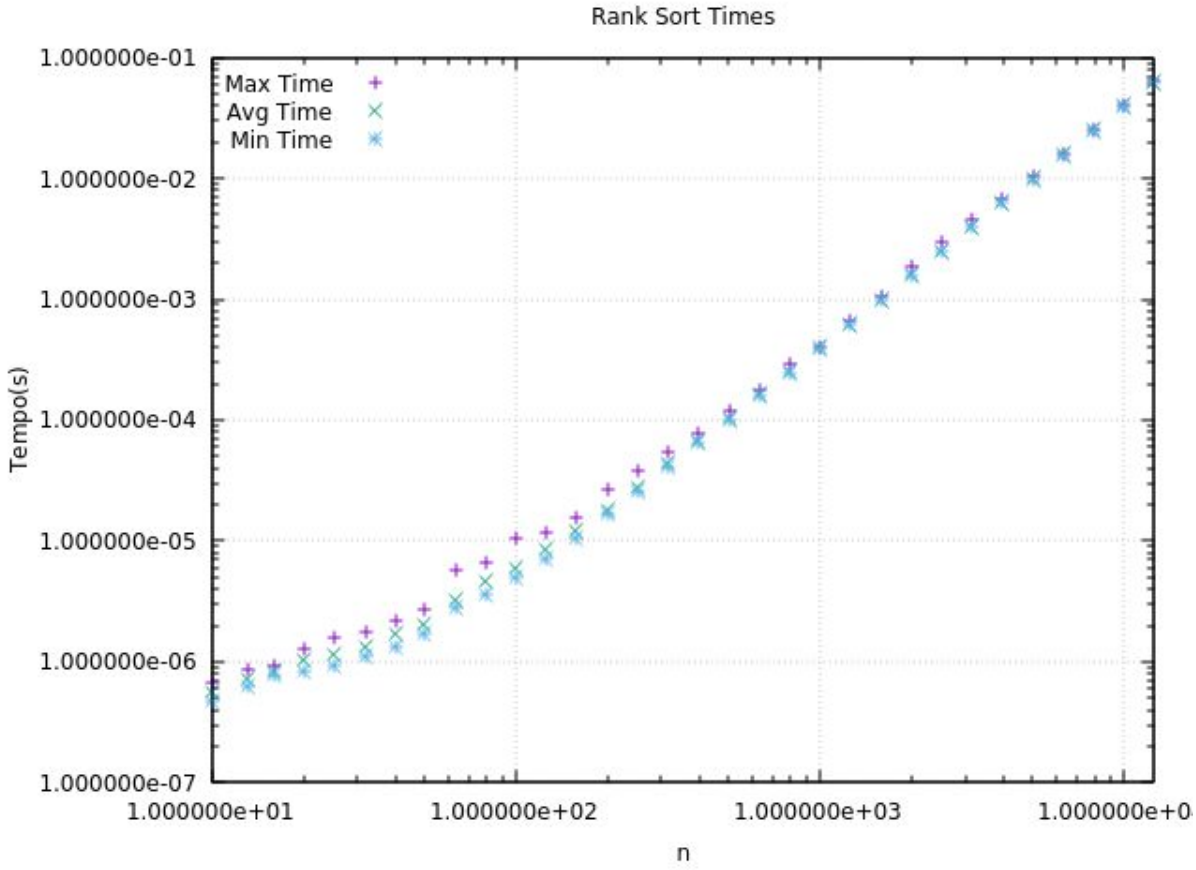

Gráfico 21: Tempos de execução do Rank Sort

Assim como o último algoritmo analisado, este apresenta também uma complexidade computacional O(n^2), portanto irá ter as mesmas conclusões que os outros. Ou seja, à medida que o valor de n cresce, os tempos de execução, também crescem de forma quadrática

Analisamos novamente, que para valores de n maiores, os tempos mínimos e máximos não estão tão próximos dos tempos médios, como para valores de n menores.

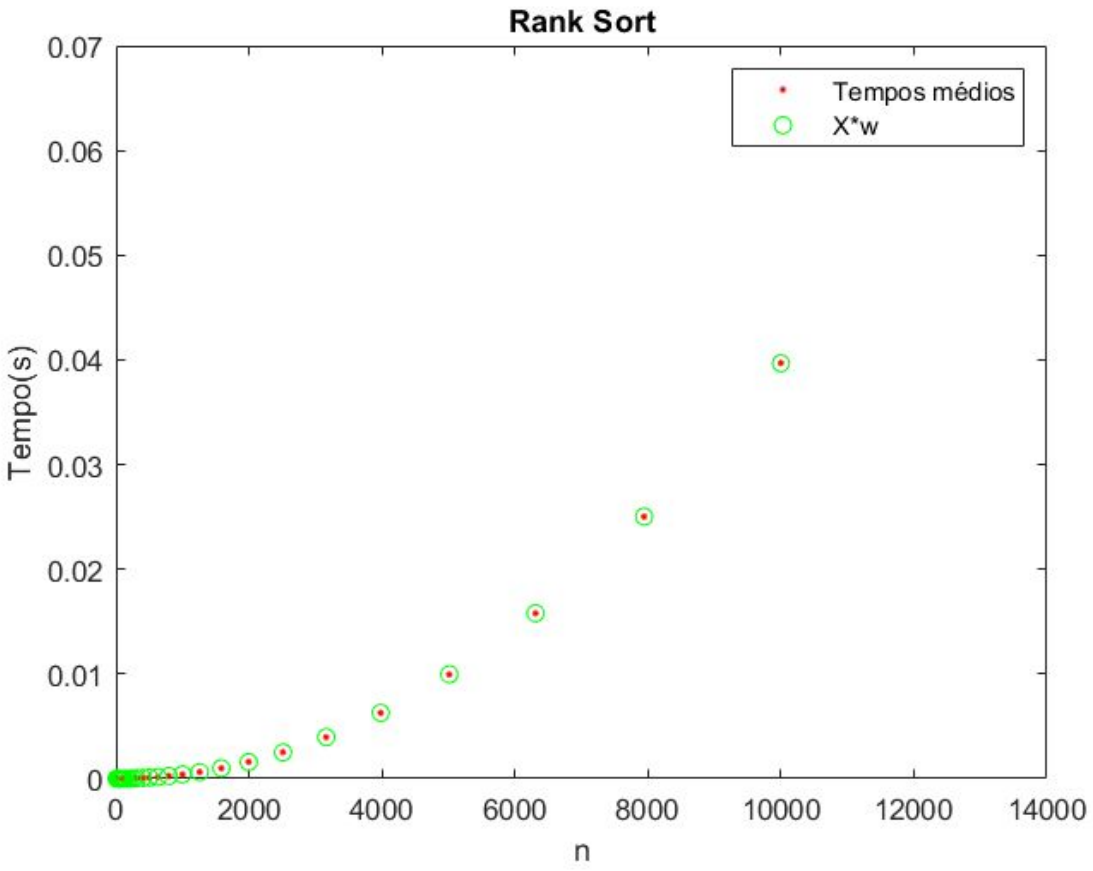

Gráfico 22: Função aproximada do Rank Sort

Nesta aproximação foi gerado um erro aproximado a 5.771224399770546e-05, sendo esta aproximação bastante positiva relativamente a algumas já referidas acima.

Neste algoritmos, como nos últimos apresentados, continuamos a achar desnecessário recorrer à ferramenta *zoom* existente no Matlab devido ao facto de o erro obtido ser relativamente mais baixo e estando no gráfico a aproximação bem representada.

3.3. Análise geral dos algoritmos de ordenação

3.3.1. Gráfico geral dos tempos de execução

Para a construção de um gráfico com os tempos médios de todos os algoritmos de ordenação, usamos no gnuplot os comandos expostos na Figura 2.

```
eva@eva-TUF-GAMING-FX504GD-FX80GD:~/Documentos/AED/A02/A02$ gnuplot

        G N U P L O T
        Version 5.2 patchlevel 2    last modified 2017-11-01

        Copyright (C) 1986-1993, 1998, 2004, 2007-2017
        Thomas Williams, Colin Kelley and many others

        gnuplot home:     http://www.gnuplot.info
        faq, bugs, etc:   type "help FAQ"
        immediate help:   type "help"  (plot window: hit 'h')

Terminal type is now 'qt'
gnuplot> data_file="output.txt"
gnuplot> set title "Average time"
gnuplot> set logscale xy
gnuplot> set grid
gnuplot> set key left
gnuplot> set format xy "%0e"
gnuplot> set xlabel "n"
gnuplot> set ylabel "Tempo(s)"
gnuplot> plot data_file i 0 u 1:4 w p t 'Bubble Sort'    ,\
>     data_file i 1 u 1:4 w p t 'Shaker Sort'    ,\
>     data_file i 2 u 1:4 w p t 'Insertion Sort',\
>     data_file i 3 u 1:4 w p t 'Shell Sort'     ,\
>     data_file i 4 u 1:4 w p t 'Quick Sort'     ,\
>     data_file i 5 u 1:4 w p t 'Merge Sort'     ,\
>     data_file i 6 u 1:4 w p t 'Heap Sort'      ,\
>     data_file i 7 u 1:4 w p t 'Rank Sort'      ,\
>     data_file i 8 u 1:4 w p t 'Selection Sort'
```

Figura 2: Comandos utilizados no gnuplot, para construir o gráfico dos tempos médios de execução

Gerando assim um gráfico (Gráfico 23), em que o eixo das abscissas é o n e o das ordenadas representa o tempo em segundos. O símbolo '+' de cor roxa representa os tempos médios de execução do Bubble Sort, o símbolo 'x' de cor verde simboliza os tempos do Shaker Sort, o '*' de cor azul retrata o Insertion, depois a delimitação de um quadrado de cor amarelo torrado representa o Shell, o quadrado pintado de amarelo corresponde ao Quick, a delimitação de um círculo de cor azul simboliza o Merge, posteriormente o círculo vermelho relata o Heap, o triângulo preto apresenta o Rank e por fim o triângulo roxo simboliza o Selection.

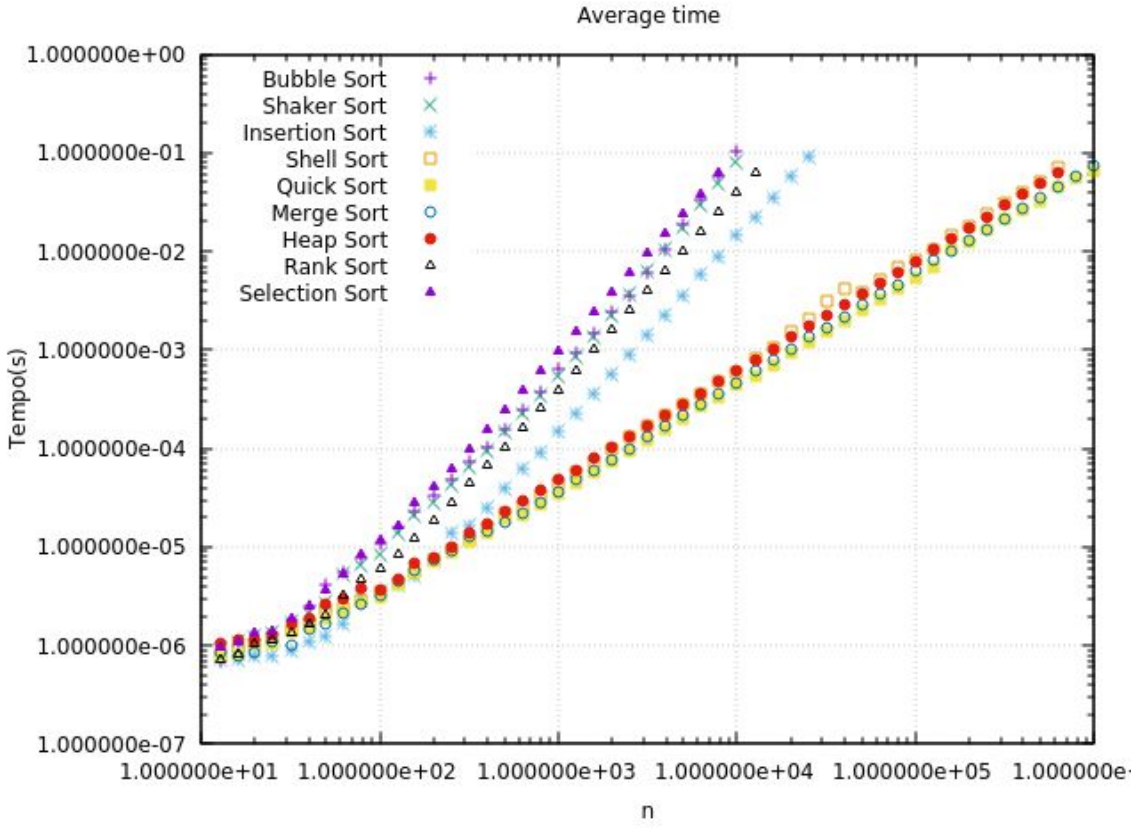

Gráfico 23: Tempos de execução de todos os algoritmos de ordenação

No gráfico 23 é bem visível quais são os algoritmos de complexidade O(n log n) e aqueles com complexidade O(n^2). Os primeiros são aqueles que aparentam um declive maior (2), ou seja, estão mais encostados ao eixo das abcissas, e os de O(n^2) apresentam um declive menor (1), estando mais juntos ao eixo das ordenadas.

Uma das grandes dúvidas que se colocam acerca de qual algoritmo utilizar para cada tamanho de *array*, é o facto de se escolher o algoritmo que permite obter um melhor tempo de execução.

Suponhamos, que o *array* com que se vai trabalhar tem tamanho 10, o melhor algoritmo é aquele que nos vai permitir ordenar o nosso *array* num menor tempo. Logo, para este caso específico, o melhor algoritmo a utilizar será o *Merge Sort*, como podemos observar no gráfico 24.

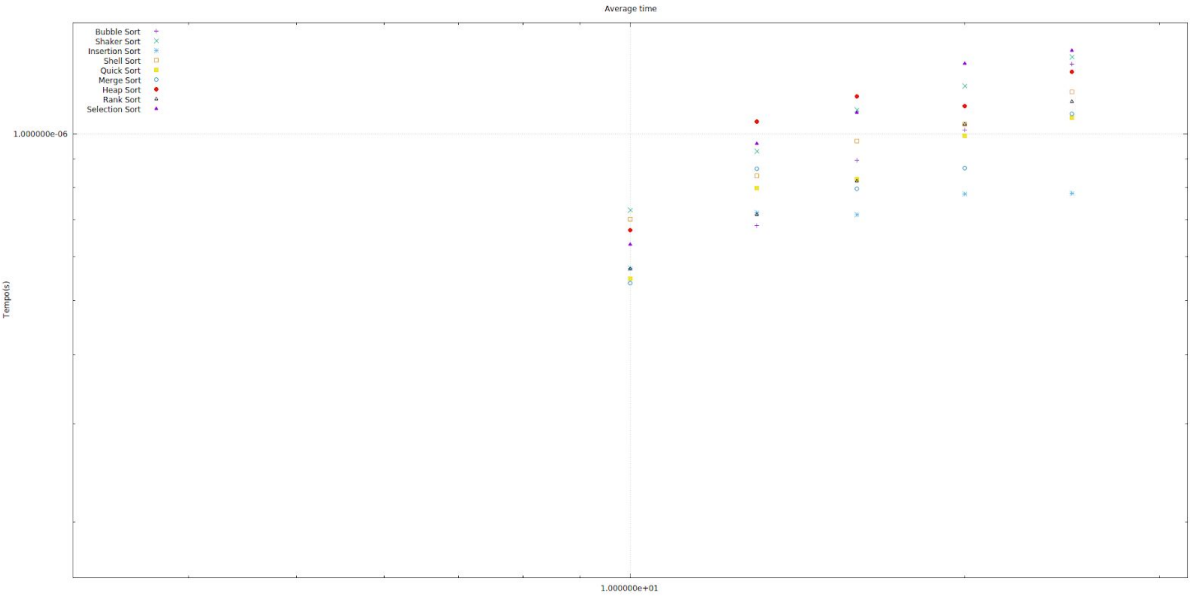

Gráfico 24: Tempos de execução do para n=1e01

Suponhamos agora, que o nosso *array,* aumentou de tamanho para 1000, qual será o melhor algoritmo a utilizar? Neste caso será o *Quick Sort*, como podemos comprovar no gráfico 25.

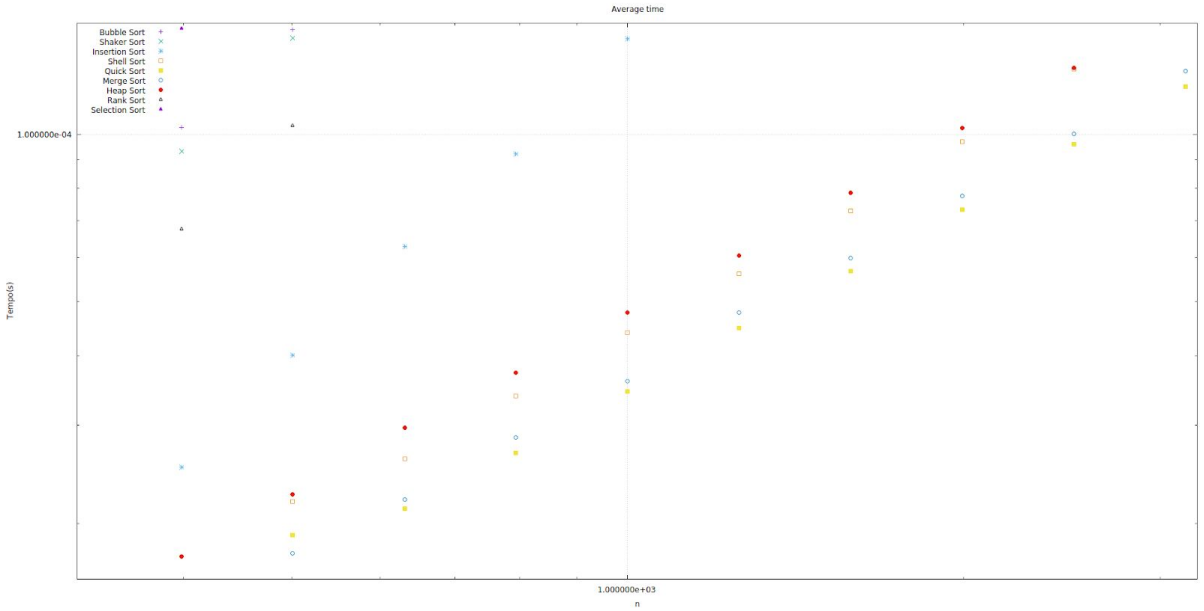

Gráfico 25: Tempos de execução do para n=1e03.

Quando o tamanho aumenta para 1 milhão de elementos, os algoritmos que têm complexidade computacional O(n^2) já não se devem utilizar pois já começam a ficar muito lentos. Para isso, dos algoritmos estudados neste trabalho, sobram-nos apenas quatro algoritmos, tendo estes O(n log n). Logo, para 1 milhão de elementos, o algoritmo escolhido para ordenar este *array* será o *Quick Sort*, como podemos observar no gráfico 26.

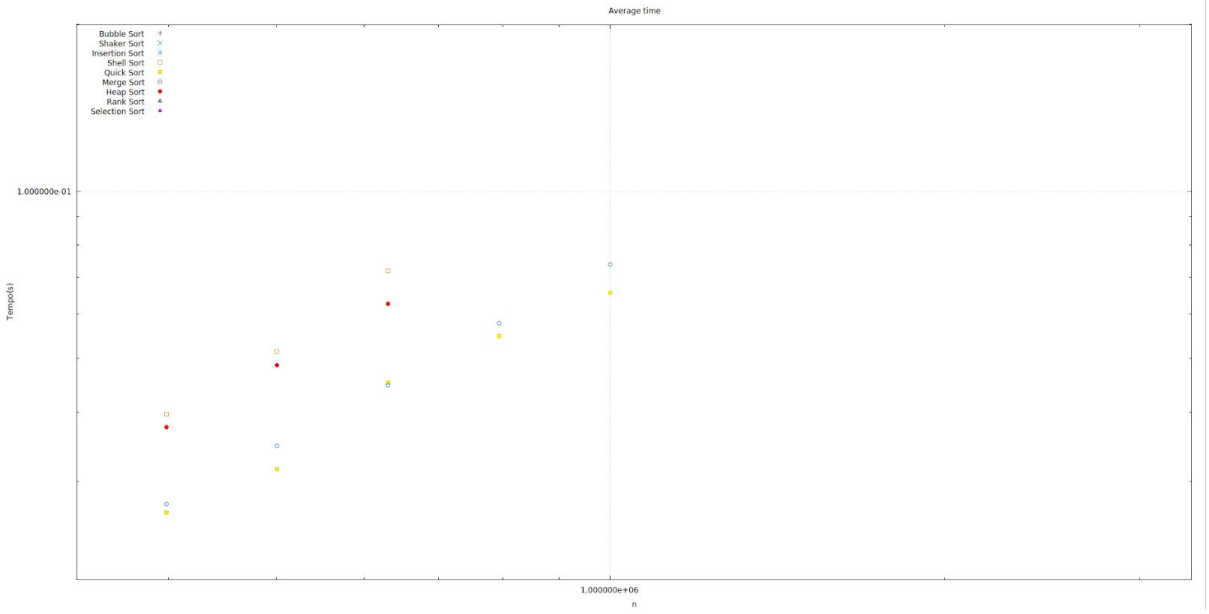

Gráfico 26: Tempos de execução do para n=1e06

Observamos também, que a partir de valores de *n* inferiores a 200 elementos, podemos observar que o algoritmo que tem melhor tempo de execução é o *Insertion Sort.* Para valores de *n* superiores a 200 elementos, o *Quick sort,* passa a ser, continuamente, o algoritmo que tem melhores tempos de execução.

3.3.2. Funções aproximadas aos algoritmos de ordenação

Para os algoritmos que são de ordem O(n^2), foi utilizada uma função de aproximação do tipo (A* n^2 + b*n + c), podendo observar através dos gráficos 4,7, 19 e 21, que se conseguiram aproximações boas aos algoritmos estudados, havendo mais discrepâncias no algoritmo *Bubble Sort (*Gráfico 1*)*.

Para os algoritmos que são de ordem O(n log n), podemos observar que, para os algoritmos *Merge Sort* e *Heap Sort (*Gráficos 15 e 17, respectivamente*),* se obtiveram aproximações boas, não diferindo muito da função utilizada para se fazer a aproximação, que foi do tipo (A * nlog(n) + B*n + C). Contudo, para os algoritmos *Shell Sort* e *Quick Sort (*Gráficos 9 e 12, respectivamente), observamos que se notaram mais discrepâncias comparando com os algoritmos *Merge Sort* e *Heap Sort.*

4. Escolha do algoritmo a utilizar

Neste trabalho foram analisados bastantes métodos para a ordenação de *arrays*.

Uns com tempos de execução mais rápidos, com complexidades computacionais variadas, quantidades de linhas de códigos bastante diferentes e com ocupação de memória também variada.

Sendo assim, com estes fatores todos em conta, é necessário uma avaliação dos mesmos em todos os métodos de forma a encontrar o melhor entre eles.

Sendo esta uma escolha tendo em base a nossa opinião, o que achamos mais relevante na escolha do melhor algoritmo a utilizar seria sem dúvida o tempo de execução do mesmo. Em geral, verificamos que o método de ordenação mais rápido é o *Quick Sort* pois, para ordenar *arrays* menores verifica-se que o mais rápido é o *Merge Sort,* como foi referido no tópico 3.3.1.

Um lado negativo deste método de ordenação seria o código a implementar. Verificamos que a quantidade de linhas de código necessárias à implementação do mesmo é consideravelmente superior às outras rotinas de ordenação.

5. Apêndice

```
grep "^[^#]" output.txt | awk '{print $1 " " $4}' > avg_time.txt
```

Figura 3: Código bash de avg_time.sh.

```
load avg_time.txt
idx = find(avg_time(:,1) == 10);
sizeAvg = size(avg_time);
idx(end+1) = sizeAvg(1);
strs = ["Bubble Sort" , "Shaker Sort", "Insertion Sort" , "Shell Sort" ,
    "Quick Sort" , "Merge Sort" , "Heap Sort" , "Rank Sort", "Selection Sort"];

for o = 1:(length(idx)-1)
    i = idx(o);
    seguinte = idx(o + 1);
    x = avg_time(i:seguinte,1); % extract the first column
    y = avg_time(i:seguinte , 2);% extract the second column

    if(o < 4 || o > 7)
        X = [ x.^2, x, 0*x+1 ];% build the X matrix
    else
        X = [ x.*log(x), x, 0*x+1 ];% build the X matrix
    end

    w = pinv(X)*y;% optimal solution (could also be written as w = X \ y;)
    e = y-X*w;
    format long;
    norm(e)
    figure;
    plot(x,y,'.r',x,X*w,'og');% plot the original data and its best least squares approximation
    title(strs(o));
    xlabel("n");
    ylabel("Tempo(s)");
    legend('Tempos médios','X*w')
end
```

Figura 4: Código matlab de leastSquaresFit.m.

6. Conclusão

Com este trabalho podemos concluir que fortalecemos os nossos conhecimentos acerca das várias rotinas de ordenação que fizeram parte deste trabalho prático.

Além disso, conseguimos perceber melhor quais as similaridades e as diferenças entre cada uma delas, abrangendo temas como a complexidade computacional e memória ocupada por cada um dos métodos permitindo também um aprofundamento intelectual sobre estes assuntos.

Ao decorrer deste trabalho apercebemo-nos também que ao calcular o desvio padrão de cada uma das funções aproximadas de cada método de ordenação, caso este desvio fosse consideravelmente pequeno, verificava-se que os tempos mínimos e tempos máximos de execução da rotina em específico aproximavam-se do tempo médio de execução da mesma.

Na nossa opinião, os objetivos principais deste trabalho prático foram atingidos com sucesso. **Conseguimos visualizar e comparar os diferentes tempos de execução das rotinas de ordenação.**