Algoritmos e Estruturas de Dados

Sorting Methods

Professores:

Tomás Oliveira e Silva (tos@ua.pt)
Pedro Lavrador (plavrador@ua.pt)

Pedro Sobral, 98491 – 1/3%

André Freixo, 98495 – 1/3%

Marta Fradique, 98626 – 1/3%

29/01/2021

Índice

1 - Introdução

No âmbito da unidade curricular de AED, foi-nos apresentada a realização deste trabalho prático, sendo este relatório o resultado do problema "Sorting Methods". Todo o código fonte e informações deste trabalho prático podem ser encontradas neste repositório do GitHub[1] (mais informações, ler README.md do repositório).

O trabalho prático, consiste essencialmente em estudar os tempos de execução de uma série de rotinas de ordenação. Os algoritmos de ordenação foram implementados em C, sendo os mesmos fornecidos pelos docentes da unidade curricular.

A linguagem de programação C, é uma linguagem muito poderosa, pois dá ao programador um controlo íntegro de todo o processo programado, sendo uma linguagem onde o programador tem de lidar com todos os pormenores, torna-se consideravelmente eficiente e otimizada. Desta forma, conseguiremos execuções mais eficientes, pois toda a implementação é feita em C, como já referido anteriormente.

Com a realização deste trabalho prático, esperamos veemente alargar os nossos conhecimentos em C, e principalmente em conhecimento sobre algoritmos de ordenação. Esperamos também conseguir concluir com êxito todos os objetivos que são propostos no início (em comentário) do programa sorting_methods.c.

[1]Por motivos de privacidade o repositório encontra-se privado, para visualização é favor entrar em contacto com os autores do trabalho prático.

2 - Introdução ao Problema

2.1 - Resumo

A realização deste trabalho visa de forma genérica, estudar e tirar conclusões à cerca de diversas estratégias de ordenação. Desta forma, iremos correr o ficheiro *sorting_methods.c*, de modo a que todos os resultados provenientes dessa execução sejam guardados num ficheiro novo *output.txt*, e todos os gráficos e conclusões à cerca das rotinas de ordenação serão feitas a partir dessa fonte de informação.

2.2 – Compilação e Execução

Para compilar o programa é necessário à partida ter um compilador de C instalado na máquina, por exemplo o *gcc*.

Posto isto, para compilar usamos o *makefile*:

```
make sorting_methods
```

O programa está capacitado com um teste, para executar o teste basta passar a seguinte linha de código:

```
./sorting_methods -test
```

Para correr o programa, e guardar os resultados dos tempos de execução, num ficheiro .txt, executamos a seguinte linha:

```
./sorting_methods -measure | tee output.txt
```

3 - Resultados

Como já referido anteriormente, os algoritmos e toda a implementação foi fornecida pelos docentes da unidade curricular, sendo então aqui no ponto 3 do relatório, que iremos expor os nossos resultados para cada rotina de ordenação.

3.1 – Bubble Sort

Bubble sort é um algoritmo de ordenação bastante simples, este fundamenta-se em trocar a ordem dos elementos até estes estarem numa ordem correta. Ainda, o *bubble sort* em termos de complexidade computacional, no melhor caso é de **O(n)**, no médio de **O(n^2)** e no seu pior caso de **O(n^2)** também.

Por exemplo:

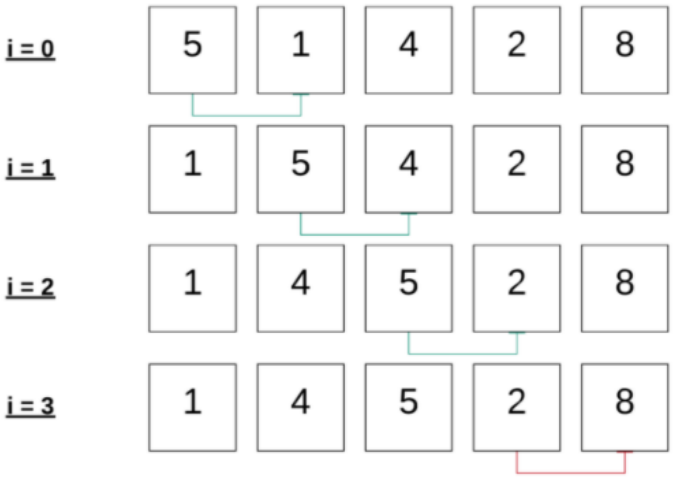

Figura 1- Demonstração funcionamento do Bubble Sort

A tabela com os tempos de execução para o *Bubble Sort* é a seguinte (Fig.2):

```
# bubble_sort
#      n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
      10 5.090e-07 8.030e-07 5.781e-07 6.397e-08
      13 8.180e-07 1.570e-06 1.381e-06 1.559e-07
      16 1.508e-06 1.882e-06 1.689e-06 8.603e-08
      20 1.845e-06 2.328e-06 2.080e-06 1.138e-07
      25 1.540e-06 2.778e-06 2.329e-06 3.448e-07
      32 1.695e-06 3.244e-06 2.535e-06 4.974e-07
      40 2.444e-06 4.224e-06 3.325e-06 5.050e-07
      50 3.136e-06 5.003e-06 3.694e-06 4.501e-07
      63 4.271e-06 6.647e-06 5.168e-06 5.660e-07
      79 6.410e-06 1.065e-05 7.421e-06 8.219e-07
     100 1.001e-05 1.570e-05 1.172e-05 1.189e-06
     126 1.436e-05 2.715e-05 1.812e-05 2.682e-06
     158 2.060e-05 3.290e-05 2.458e-05 2.664e-06
     200 3.062e-05 4.536e-05 3.345e-05 2.358e-06
     251 4.453e-05 5.641e-05 4.764e-05 2.146e-06
     316 6.504e-05 8.420e-05 6.896e-05 3.910e-06
     398 9.752e-05 1.321e-04 1.052e-04 8.303e-06
     501 1.465e-04 1.916e-04 1.542e-04 9.586e-06
     631 2.196e-04 2.586e-04 2.273e-04 5.008e-06
     794 3.321e-04 4.045e-04 3.444e-04 1.340e-05
    1000 5.118e-04 5.551e-04 5.231e-04 7.217e-06
    1259 7.953e-04 8.450e-04 8.112e-04 9.852e-06
    1585 1.254e-03 1.444e-03 1.285e-03 2.820e-05
    1995 2.014e-03 2.409e-03 2.093e-03 7.273e-05
    2512 3.310e-03 4.796e-03 3.633e-03 3.673e-04
    3162 5.637e-03 6.934e-03 6.126e-03 3.485e-04
    3981 9.734e-03 1.187e-02 1.045e-02 5.560e-04
    5012 1.721e-02 1.890e-02 1.762e-02 2.712e-04
    6310 3.054e-02 3.699e-02 3.169e-02 1.278e-03
    7943 5.324e-02 6.587e-02 5.658e-02 2.943e-03
   10000 9.101e-02 1.034e-01 9.320e-02 2.365e-03
```

Figura 2 - Tabela Bubble Sort

O gráfico com os tempos de execução para o *Bubble Sort*, é o seguinte:

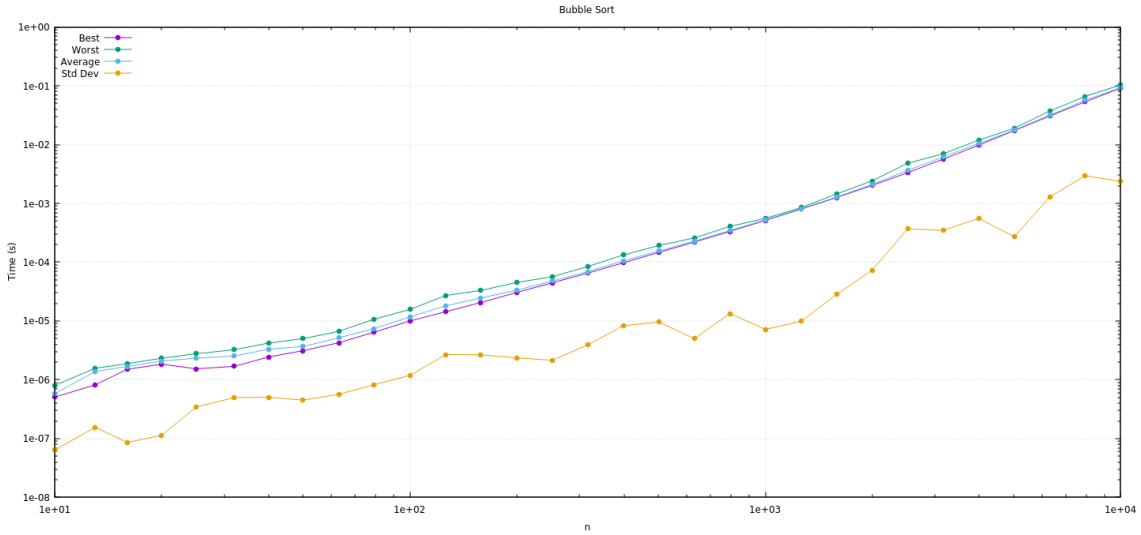

Figura 3 - Gráfico Bubble Sort

Ao analisamos o gráfico acima, observamos que o *Bubble Sort*, cresce de forma exponencial de aproximadamente $n2$ (pois como podemos observar na figura enquanto no eixo do x a função aumenta uma unidade no eixo do x aumenta duas), sendo que também se torna mais linear á medida que o n cresce. Quanto ao desvio padrão, este oscila de forma razoável contendo alguns picos.
Criamos ainda um outro gráfico para representarmos o *least squares fit,* a partir desta reduzimos a uma determinada escala o número de informação e observar nitidamente a tendência da curva. No *Bubble Sort* os valores crescem de forma n^2.

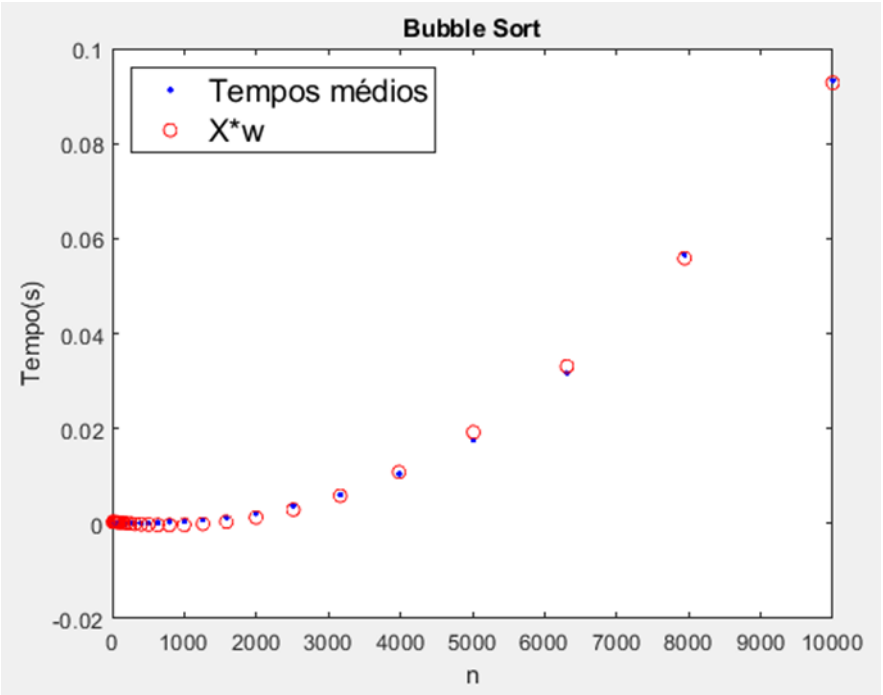

Figura 4 -Least Square Fit Bubble Sort

Ainda, ao analisarmos o gráfico com todas as rotinas de ordenação, podemos concluir que o *Buble Sort* tem o pior tempo de execução. Uma das maiores desvantagens desta rotina é a quantidade exacerbada de tempo que demora a fazer a ordenação, ainda o tempo aumenta de forma exponencial á medida que o número de dados a analisar também aumenta.

3.2 – Shaker Sort

O algoritmo *Shaker Sort* muda os elementos da direita para a sua posição correta. Ora, o algoritmo inicialmente funciona da esquerda para a direita, onde os itens são comparados dois a dois, se o da direita for maior eles trocam e assim sucessivamente, assim no fim da primeira iteração o maior número vai estar colocado na última posição. Na próxima iteração, o *array* é percorrido na direção oposta até à primeira posição, os itens também são comparados e invertem de posição se for necessário.

Ainda, o *shaker sort* em termos de complexidade computacional, no melhor caso é de **O(n)**, no médio de **O(n^2)** e no seu pior caso de **O(n^2)** também.

A tabela com os tempos de execução para o *Shaker Sort* é a seguinte (Fig.5):

```
# shaker_sort
#      n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
      10 5.240e-07 8.660e-07 6.580e-07 1.199e-07
      13 6.150e-07 1.045e-06 8.025e-07 1.503e-07
      16 8.990e-07 1.222e-06 1.009e-06 7.412e-08
      20 9.960e-07 1.419e-06 1.141e-06 8.882e-08
      25 1.303e-06 1.934e-06 1.540e-06 1.531e-07
      32 1.579e-06 3.519e-06 2.315e-06 5.537e-07
      40 2.241e-06 3.005e-06 2.633e-06 1.759e-07
      50 2.977e-06 4.048e-06 3.415e-06 2.588e-07
      63 4.164e-06 6.450e-06 5.153e-06 6.222e-07
      79 5.577e-06 8.947e-06 6.893e-06 8.447e-07
     100 9.407e-06 1.398e-05 1.127e-05 1.014e-06
     126 1.323e-05 2.255e-05 1.611e-05 2.034e-06
     158 1.863e-05 2.992e-05 2.177e-05 2.380e-06
     200 2.930e-05 4.500e-05 3.369e-05 3.153e-06
     251 4.160e-05 6.752e-05 4.859e-05 6.176e-06
     316 6.075e-05 8.627e-05 6.773e-05 5.709e-06
     398 9.197e-05 1.209e-04 9.891e-05 6.097e-06
     501 1.380e-04 1.807e-04 1.475e-04 8.350e-06
     631 2.106e-04 2.642e-04 2.227e-04 1.045e-05
     794 3.232e-04 4.021e-04 3.398e-04 1.459e-05
    1000 5.015e-04 5.961e-04 5.222e-04 1.663e-05
    1259 7.860e-04 9.308e-04 8.154e-04 2.312e-05
    1585 1.252e-03 1.490e-03 1.301e-03 3.994e-05
    1995 2.031e-03 2.288e-03 2.097e-03 4.122e-05
    2512 3.361e-03 3.782e-03 3.463e-03 7.156e-05
    3162 5.660e-03 6.264e-03 5.820e-03 1.157e-04
    3981 9.567e-03 1.035e-02 9.832e-03 1.635e-04
    5012 1.617e-02 1.745e-02 1.659e-02 2.664e-04
    6310 2.710e-02 2.856e-02 2.770e-02 3.263e-04
    7943 4.500e-02 4.700e-02 4.583e-02 4.478e-04
   10000 7.398e-02 7.646e-02 7.513e-02 5.769e-04
```

Figura 5 – Tabela Shaker Sort

O gráfico com os tempos de execução para o *Shaker Sort*, é o seguinte (Fig.6):

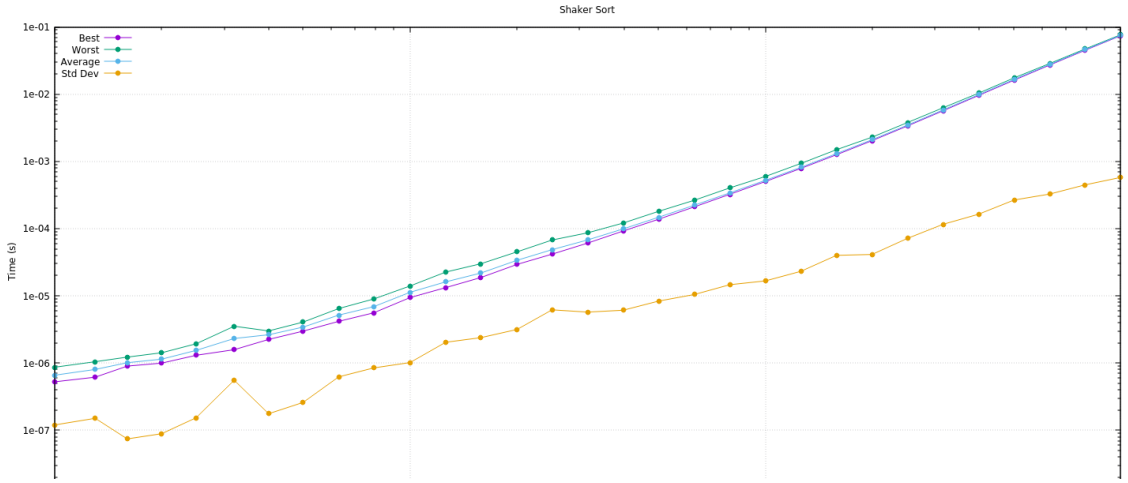

Figura 6- Gráfico Shaker Sort

Ao analisamos o gráfico acima, podemos concluir que o *Shaker Sort*, cresce de forma exponencial de aproximadamente $n2$ (pois como podemos observar na figura enquanto no eixo do x a função aumenta uma unidade no eixo do x aumenta duas). Quanto ao desvio padrão, este vai oscilando ao longo de *n,* com alguns desvios, mas não muito notórios.

O *Shaker Sort* é uma variação do *Bubble Sort*. A diferença entre eles é que o *Shaker Sort* percorre o *array* nas duas direções (direita para a esquerda e esquerda para a direita) ao contrário do *Bubble Sort* que só percorre numa direção, esta diferença faz com que o *Shaker Sort* apresente um tempo de execução ligeiramente melhor. Sendo assim, a próxima figura (Fig.8), apresenta a comparação entre as duas estratégias de ordenação.

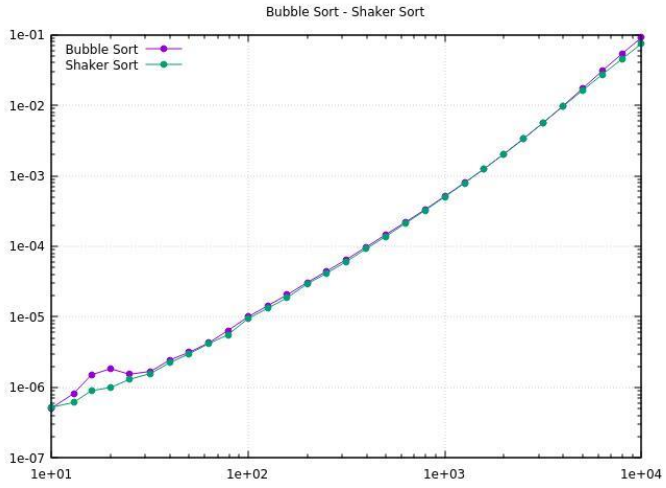

Figura 7 - Comparação entre o Bubble Sort e Shaker Sort

Ao compararmos diretamente estas duas rotinas, podemos concluir que a diferença a nível de tempo entre o *Shaker Sort* e o *Bubble Sort* é quase impercetível, tendo o *Shaker Sort* um tempo de execução ligeiramente melhor em alguns valores de *n* do gráfico.

Criamos ainda um outro gráfico para representarmos o *least squares fit,* a partir deste reduzimos o número de informação a uma determinada escala, de forma a observar nitidamente a tendência da curva. No *Shaker Sort* os valores médios acompanham todos a tendência de crescimento de **n^2,** o que comprova mais uma vez o crescimento desta rotina.

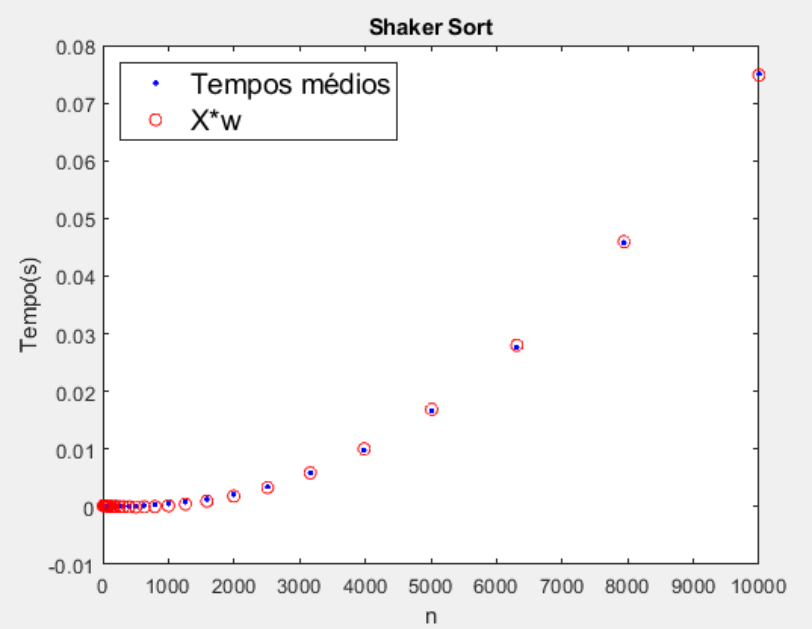

Figura 8 - Least Square Fit Shaker Sort

Ainda, ao analisarmos o gráfico com todas as rotinas de ordenação, podemos concluir que o *Shaker Sort* tem um desempenho medíocre, e existem outras implementações que podem ser usadas para uma melhor performance.

3.3 – Insertion Sort

Insertion sort é um algoritmo de ordenação bastante acessível, este consiste em dividir um *array* numa parte ordenada e numa desordenada, de seguida selecionamos um valor da parcela desordenada e colocamos esta na parte ordenada na sua respetiva posição. Para tal, efetuamos os seguintes passos: comparamos o elemento atual com o seu anterior, se o elemento atual for inferior vamos movê-lo uma posição e trocar os elementos, por exemplo, se *arr[i]* for menor que *arr[i-1]*, trocamos as posições um pelo outro.

Ainda, o *insertion sort* em termos de complexidade computacional, no melhor caso é de **O(n)**, no médio de **O(n^2)** e no seu pior caso de **O(n^2)** também.

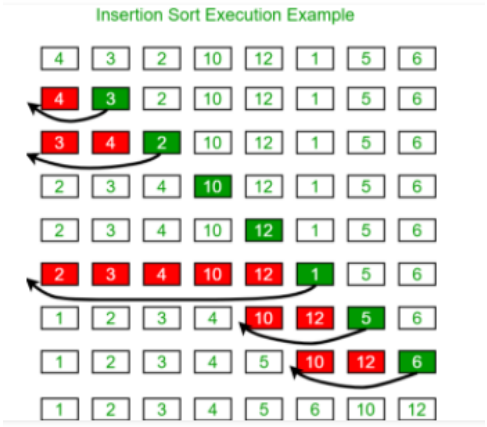

Figura 9 - Exemplo do funcionamento do Insertion Sort

A tabela com os tempos de execução para o *Insertion Sort* é a seguinte (Fig.9):

# insertion_sort				
#	n	min time	max time	avg time	std dev
#				
	10	4.820e-07	5.570e-07	5.043e-07	1.340e-08
	13	6.560e-07	7.960e-07	7.148e-07	4.577e-08
	16	5.860e-07	1.062e-06	7.251e-07	1.664e-07
	20	6.890e-07	1.165e-06	8.642e-07	1.677e-07
	25	7.410e-07	1.017e-06	8.689e-07	9.835e-08
	32	8.290e-07	1.268e-06	1.039e-06	1.567e-07
	40	9.330e-07	1.756e-06	1.236e-06	3.209e-07
	50	1.146e-06	1.735e-06	1.391e-06	1.842e-07
	63	1.431e-06	2.638e-06	1.777e-06	3.861e-07
	79	2.027e-06	2.838e-06	2.418e-06	2.176e-07
	100	3.311e-06	4.707e-06	3.851e-06	3.847e-07
	126	3.959e-06	5.336e-06	4.480e-06	3.179e-07
	158	5.091e-06	8.524e-06	5.839e-06	7.411e-07
	200	7.903e-06	1.932e-05	1.034e-05	2.512e-06
	251	1.076e-05	2.064e-05	1.205e-05	1.303e-06
	316	1.549e-05	3.219e-05	1.818e-05	2.562e-06
	398	2.405e-05	4.398e-05	2.902e-05	3.861e-06
	501	3.593e-05	7.575e-05	4.269e-05	6.889e-06
	631	5.347e-05	8.353e-05	6.090e-05	6.480e-06
	794	8.267e-05	1.114e-04	8.946e-05	6.649e-06
	1000	1.281e-04	1.688e-04	1.364e-04	8.176e-06
	1259	1.993e-04	2.545e-04	2.123e-04	1.256e-05
	1585	3.123e-04	3.869e-04	3.273e-04	1.456e-05
	1995	4.897e-04	5.896e-04	5.097e-04	1.721e-05
	2512	7.732e-04	9.129e-04	8.001e-04	2.242e-05
	3162	1.224e-03	1.482e-03	1.266e-03	4.337e-05
	3981	1.935e-03	2.165e-03	1.983e-03	3.652e-05
	5012	3.064e-03	3.495e-03	3.145e-03	6.885e-05
	6310	4.858e-03	5.610e-03	4.984e-03	1.310e-04
	7943	7.700e-03	8.545e-03	7.867e-03	1.684e-04
	10000	1.219e-02	1.310e-02	1.243e-02	2.031e-04
	12589	1.933e-02	2.051e-02	1.969e-02	2.691e-04
	15849	3.069e-02	3.229e-02	3.120e-02	3.653e-04
	19953	4.865e-02	5.076e-02	4.940e-02	4.466e-04
	25119	7.737e-02	7.969e-02	7.829e-02	5.223e-04

Figura 10 - Tabela Insertion Sort

O gráfico com os tempos de execução para o *Insertion Sort* é o seguinte (Fig.11):

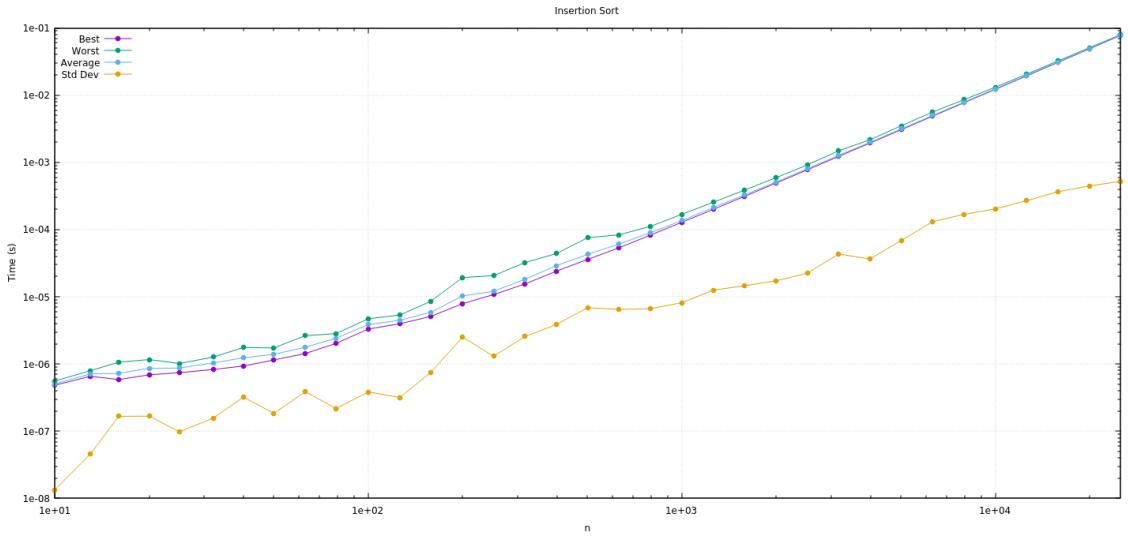

Figura 11 - Gráfico Insertion Sort

Ao analisamos o gráfico acima, podemos concluir que o *Insertion Sort*, cresce de forma exponencial de aproximadamente $n2$ (pois como podemos observar na figura enquanto no eixo do x a função aumenta uma unidade no eixo do x aumenta duas unidades). O desvio padrão vai oscilando ao longo do *n* de forma razoável contendo apenas alguns picos.

Criamos ainda um outro gráfico para representarmos o *least squares fit*, a partir deste reduzimos o número de informação a uma determinada escala, de forma a observar nitidamente a tendência da curva. No *Shaker Sort* os valores médios acompanham todos a tendência de crescimento de n^2, o que comprova mais uma vez o crescimento desta rotina.

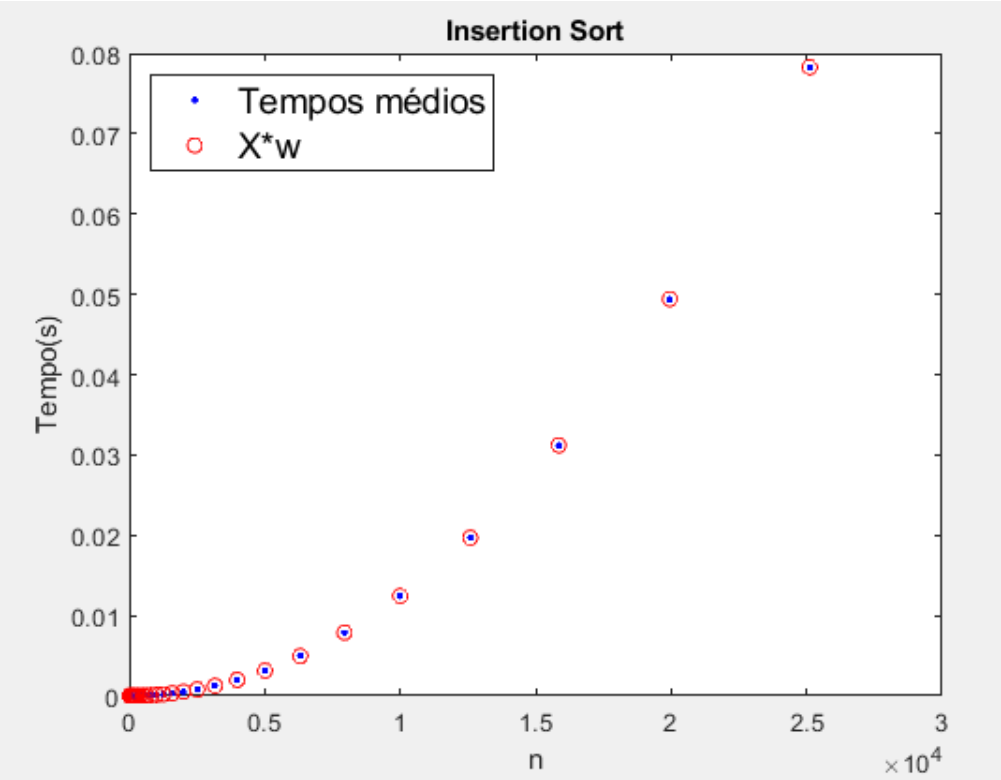

Figura 12 - Least Square Fit Insertion Sort

Ora, ao analisarmos o desempenho de todas as técnicas de *sorting* em conjunto podemos concluir que o *Insertion sort* apresenta um bom desempenho quando o número de ***n*** é reduzido. Assim, quando o *array* já se encontra parcialmente ordenado a técnica do *insertion sort* deve ser utilizada por apresentar o melhor desempenho, caso tenhamos um *array* maior ou menos ordenado é mais conveniente utilizar técnicas como o *merge sort.*

3.4 – Shell Sort

A estratégia de ordenação, *Shell Sort* (Fig.12), é de forma genérica uma sucessiva aplicação da estratégia de ordenação, *Insertion Sort*, onde são criados sub-arrays do array original. A ideia do *Shell Sort*, é que seja possível a troca de itens distantes, criamos o array-h para um grande valor de h, e vamos reduzindo o valor de h, até que este se torne 1

A nível da complexidade computacional desta rotina de ordenação, é um pouco manhosa, pois ainda não é conhecida para nenhum dos 3 casos (*best, worst* e *average*), depende muito da sequência de *strides* (passos) que são usadas. São conhecidos alguns valores para h, de modo a que o algoritmo seja **o(n^2),** por exemplo, quando h = 9 * 2^s - 9 * $2^{s/2}$ + 1 , para *s* par. No entanto é essencial voltar a ressalvar que a complexidade computacional depende muito do valor de h.

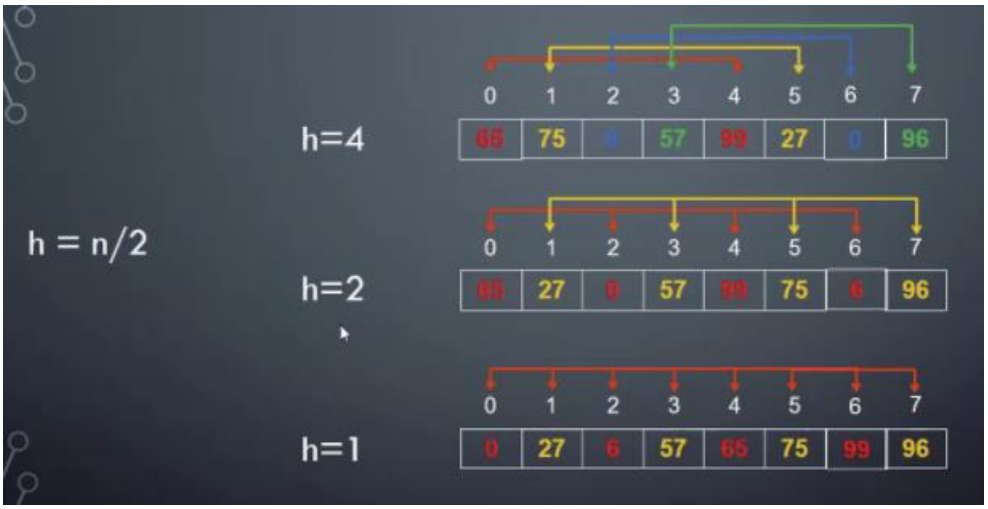

Figura 13 - Exemplo demonstrativo do funcionamento do Shell Sort

A tabela com os tempos de execução obtida foi a seguinte (Fig.13):

```
# Shell_sort
#       n  min time   max time   avg time    std dev
#------- --------- --------- --------- ---------
      10 5.280e-07 6.580e-07 5.961e-07 3.831e-08
      13 6.190e-07 1.074e-06 7.485e-07 1.594e-07
      16 6.640e-07 1.201e-06 8.641e-07 1.800e-07
      20 8.210e-07 1.119e-06 9.492e-07 8.542e-08
      25 9.270e-07 1.326e-06 1.110e-06 1.236e-07
      32 1.006e-06 1.909e-06 1.340e-06 3.297e-07
      40 1.306e-06 1.799e-06 1.567e-06 1.272e-07
      50 1.750e-06 2.005e-06 1.877e-06 5.917e-08
      63 2.021e-06 2.842e-06 2.362e-06 2.086e-07
      79 2.624e-06 4.604e-06 3.668e-06 5.687e-07
     100 3.500e-06 5.705e-06 4.005e-06 5.806e-07
     126 4.385e-06 5.851e-06 5.185e-06 3.498e-07
     158 5.528e-06 9.755e-06 6.454e-06 1.107e-06
     200 7.366e-06 1.116e-05 8.982e-06 9.066e-07
     251 9.099e-06 1.832e-05 1.265e-05 2.514e-06
     316 1.163e-05 2.043e-05 1.315e-05 1.059e-06
     398 1.537e-05 2.726e-05 1.825e-05 2.449e-06
     501 1.989e-05 3.292e-05 2.275e-05 2.681e-06
     631 2.604e-05 4.095e-05 2.986e-05 3.384e-06
     794 3.395e-05 5.434e-05 3.901e-05 5.113e-06
    1000 4.300e-05 6.490e-05 4.745e-05 5.010e-06
    1259 5.638e-05 7.910e-05 6.121e-05 5.212e-06
    1585 7.395e-05 9.707e-05 7.864e-05 5.986e-06
    1995 9.637e-05 1.272e-04 1.010e-04 6.680e-06
    2512 1.237e-04 1.687e-04 1.308e-04 9.244e-06
    3162 1.597e-04 2.053e-04 1.675e-04 1.021e-05
    3981 2.090e-04 2.592e-04 2.169e-04 1.044e-05
    5012 2.727e-04 3.507e-04 2.818e-04 1.402e-05
    6310 3.534e-04 4.418e-04 3.638e-04 1.614e-05
    7943 4.599e-04 5.450e-04 4.703e-04 1.449e-05
   10000 5.971e-04 6.974e-04 6.098e-04 1.721e-05
   12589 7.803e-04 8.838e-04 7.926e-04 1.456e-05
   15849 1.015e-03 1.235e-03 1.037e-03 3.346e-05
   19953 1.315e-03 1.493e-03 1.336e-03 2.509e-05
   25119 1.708e-03 1.916e-03 1.733e-03 2.507e-05
   31623 2.216e-03 2.565e-03 2.259e-03 4.791e-05
   39811 2.877e-03 3.235e-03 2.925e-03 4.571e-05
   50119 3.741e-03 4.166e-03 3.803e-03 5.679e-05
   63096 4.850e-03 5.587e-03 4.948e-03 1.081e-04
   79433 6.284e-03 7.031e-03 6.405e-03 1.231e-04
  100000 8.145e-03 8.773e-03 8.298e-03 1.302e-04
  125893 1.058e-02 1.130e-02 1.077e-02 1.591e-04
  158489 1.372e-02 1.451e-02 1.396e-02 1.819e-04
  199526 1.775e-02 1.882e-02 1.811e-02 2.317e-04
  251189 2.299e-02 2.709e-02 2.367e-02 7.942e-04
  316228 2.982e-02 3.137e-02 3.045e-02 3.593e-04
  398107 3.872e-02 4.071e-02 3.951e-02 4.557e-04
  501187 5.021e-02 5.337e-02 5.137e-02 6.918e-04
  630957 6.513e-02 6.788e-02 6.630e-02 6.442e-04
#------- --------- --------- --------- ---------
```

Figura 14 - Tabela Shell Sort

O gráfico obtido para esta estratégia de ordenação foi o seguinte (Fig.14):

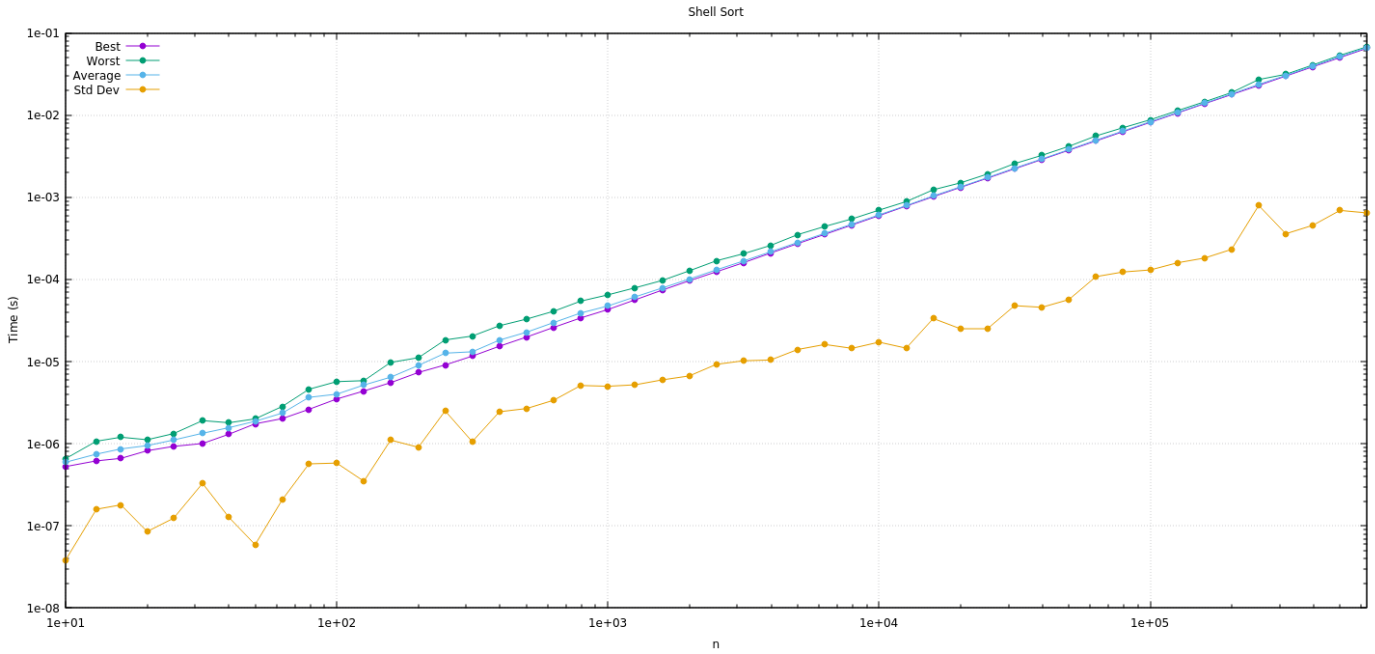

Figura 15 - Gráfico Shell Sort

Estando no gráfico os tempos numa escala logarítmica, podemos concluir que os mesmos crescem numa ordem de **O (nlog n)**, salvo a parte inicial onde isso não se verifica assim tão bem, porém à medida que o *n* cresce, este crescimento vai-se retratando cada vez mais. O desvio padrão vai oscilando ao longo do *n*, havendo momentos em é considerável.

Para justificar a complexidade computacional supra mencionada, recorremos a uma análise detalhada do gráfico da Fig.15, nesse sentido, no eixo do *x*, no intervalo entre 10^3 e 10^4, sendo que há um salto de uma unidade ao nível do expoente, em correspondência a esse mesmo salto, no eixo do y, no intervalo 10^{-4} e 10^{-3}, também se verifica um salto de um ao nível do expoente.

Como dito no início da introdução a esta estratégia de ordenação, o *Shell Sort*, passa por ser uma sucessiva aplicação do *Insertion Sort*. Sendo assim, a próxima figura (Fig.15), apresenta a comparação entre as duas estratégias de ordenação.

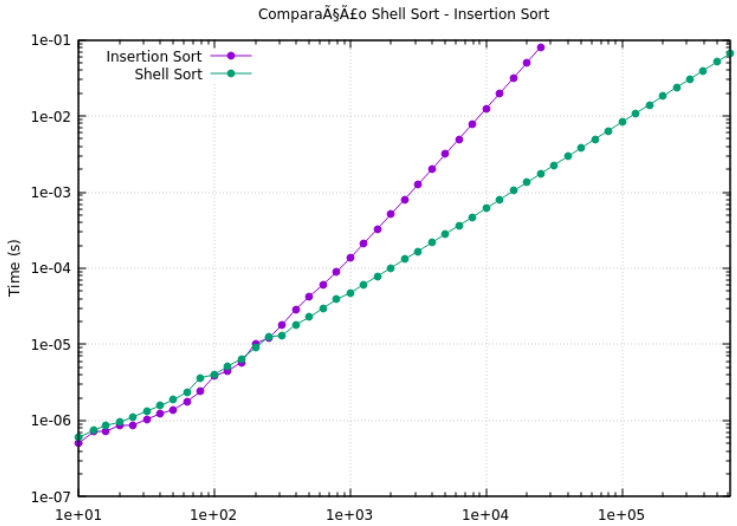

Figura 16 – Shell Sort / Insertion Sort

Na comparação direta entre as duas rotinas, podemos concluir que o *Shell Sort*, apresenta vantagens significativas a nível de tempo, comparado ao *Insertion Sort*, sobretudo para valores de *n* superiores 100.

Criamos ainda o seguinte gráfico, onde é feita a *least squares fit,* esta implementação é muito poderosa, pois permite reduzir numa certa escala o número de informação e observar de forma nítida a tendência da curva. No *Shell Sort* os valores médios acompanham todos a tendência de crescimento de $\mathbf{n^2}$**,** o que comprova mais uma vez o crescimento desta rotina.

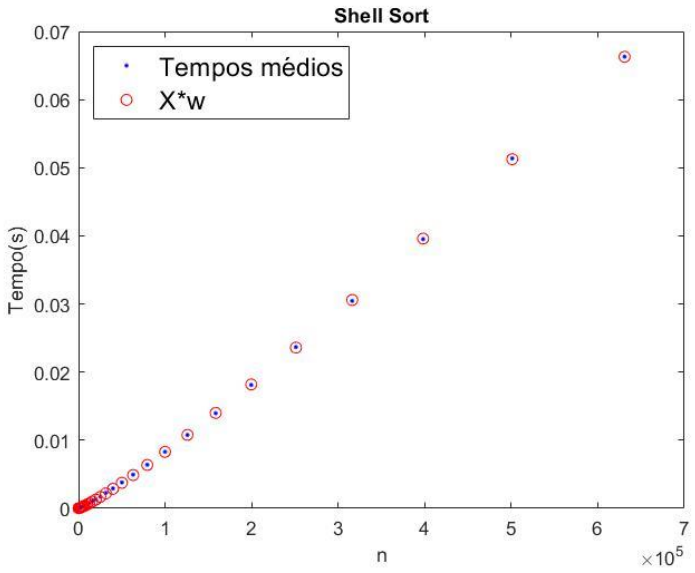

Figura 17 – Least Square Fit Shell Sort

3.5 – Quick Sort

A rotina de ordenação *Quick Sort* (Fig.17), é uma rotina que como o próprio nome indica é rápida, é implementada através de um método recursivo, que usa um algoritmo *Divide and Conquer*.

O algoritmo para ordenar *arrays* mais pequenos (menores que 20 a 30, discutível) usa a rotina *Insertion Sort*, pois torna-se mais eficiente, para *arrays* com tamanho maior, o algoritmo vai escolher um pivot, pivot esse que é escolhido aleatoriamente, neste contexto, faz-se uma passagem pelo *array*, e para a esquerda do pivot, colocamos os valores mais pequenos que o *array*, e para a direita os valores maiores, sendo a parte dos valores igual ao pivot discutível de que lado do pivot é que ficam. De seguida, é feita a mesma estratégia para cada sub-array, um da parte esquerda do pivot, outro da parte direita. O Quick Sort, tem uma complexidade computacional para o *Best e para o Average de* ***O(n log n),*** porém para *Worst* ***O(n^2).***

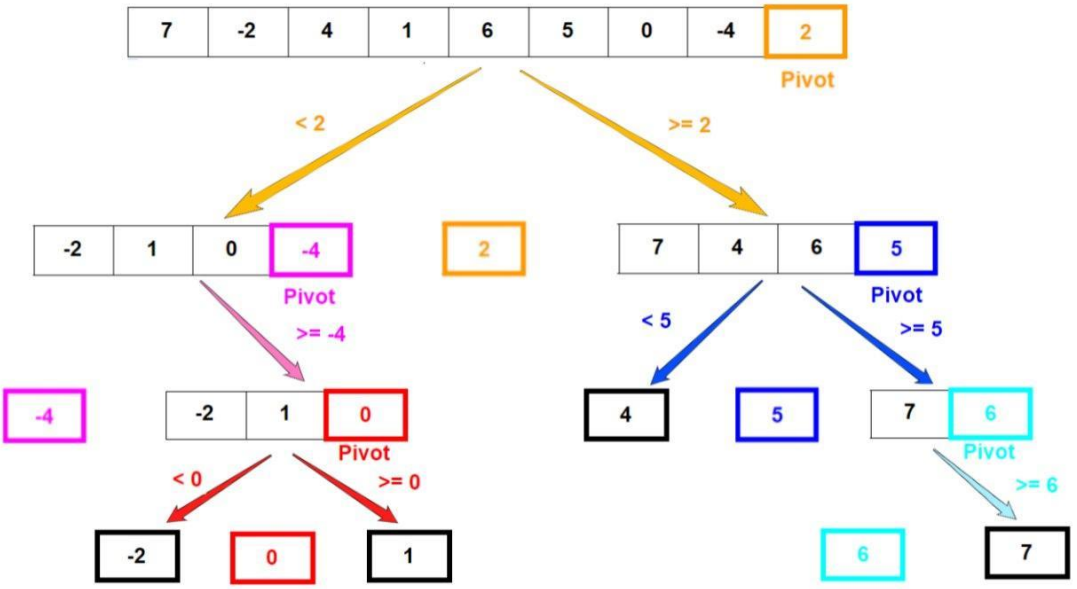

Figura 18 - Demonstração do funcionamento do Quick Sort

A tabela com os tempos de execução é a seguinte(Fig.18):

```
# quick_sort
#       n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
      10 4.900e-07 7.740e-07 5.507e-07 8.777e-08
      13 6.040e-07 8.320e-07 6.986e-07 8.941e-08
      16 5.670e-07 8.040e-07 6.654e-07 8.845e-08
      20 7.760e-07 9.370e-07 8.515e-07 4.575e-08
      25 7.890e-07 1.336e-06 9.929e-07 1.861e-07
      32 9.580e-07 1.742e-06 1.336e-06 2.863e-07
      40 1.099e-06 1.872e-06 1.392e-06 2.113e-07
      50 1.308e-06 1.963e-06 1.578e-06 2.054e-07
      63 1.652e-06 2.517e-06 1.984e-06 2.581e-07
      79 2.081e-06 3.529e-06 2.446e-06 4.253e-07
     100 2.983e-06 4.951e-06 3.852e-06 6.386e-07
     126 3.545e-06 6.017e-06 5.058e-06 6.049e-07
     158 5.254e-06 6.487e-06 5.791e-06 3.789e-07
     200 6.215e-06 9.136e-06 7.399e-06 7.276e-07
     251 7.194e-06 8.660e-06 7.772e-06 4.430e-07
     316 9.805e-06 1.707e-05 1.083e-05 9.511e-07
     398 1.258e-05 2.177e-05 1.492e-05 2.079e-06
     501 1.708e-05 2.519e-05 2.000e-05 2.146e-06
     631 2.136e-05 3.736e-05 2.414e-05 2.436e-06
     794 2.816e-05 4.380e-05 3.228e-05 3.936e-06
    1000 3.536e-05 5.589e-05 4.056e-05 4.973e-06
    1259 4.591e-05 7.028e-05 5.161e-05 5.893e-06
    1585 5.954e-05 7.816e-05 6.350e-05 4.335e-06
    1995 7.761e-05 1.016e-04 8.196e-05 5.038e-06
    2512 1.010e-04 1.295e-04 1.063e-04 6.346e-06
    3162 1.295e-04 1.665e-04 1.360e-04 6.701e-06
    3981 1.674e-04 2.188e-04 1.758e-04 9.690e-06
    5012 2.169e-04 3.453e-04 2.320e-04 2.185e-05
    6310 2.809e-04 3.396e-04 2.918e-04 1.139e-05
    7943 3.632e-04 4.290e-04 3.756e-04 1.234e-05
   10000 4.703e-04 5.525e-04 4.835e-04 1.436e-05
   12589 6.098e-04 7.138e-04 6.260e-04 1.563e-05
   15849 7.861e-04 9.095e-04 8.055e-04 1.831e-05
   19953 1.017e-03 1.184e-03 1.045e-03 2.754e-05
   25119 1.314e-03 1.482e-03 1.343e-03 2.311e-05
   31623 1.693e-03 1.955e-03 1.735e-03 3.632e-05
   39811 2.186e-03 2.384e-03 2.232e-03 3.240e-05
   50119 2.818e-03 3.141e-03 2.882e-03 4.699e-05
   63096 3.639e-03 4.004e-03 3.712e-03 5.497e-05
   79433 4.673e-03 5.134e-03 4.778e-03 7.490e-05
  100000 6.027e-03 6.533e-03 6.148e-03 9.176e-05
  125893 7.746e-03 8.329e-03 7.909e-03 1.085e-04
  158489 9.969e-03 1.061e-02 1.016e-02 1.329e-04
  199526 1.282e-02 1.358e-02 1.308e-02 1.668e-04
  251189 1.648e-02 1.741e-02 1.679e-02 2.011e-04
  316228 2.118e-02 2.220e-02 2.158e-02 2.262e-04
  398107 2.720e-02 2.839e-02 2.768e-02 2.772e-04
  501187 3.490e-02 3.644e-02 3.555e-02 3.458e-04
  630957 4.483e-02 4.655e-02 4.557e-02 3.859e-04
  794328 5.755e-02 5.950e-02 5.846e-02 4.467e-04
 1000000 7.388e-02 7.612e-02 7.498e-02 5.160e-04
#------- --------- --------- --------- ---------
```

Figura 19 - Tabela Quick Sort

O gráfico com os tempos de execução para o *Quick Sort*, é o seguinte (Fig.20):

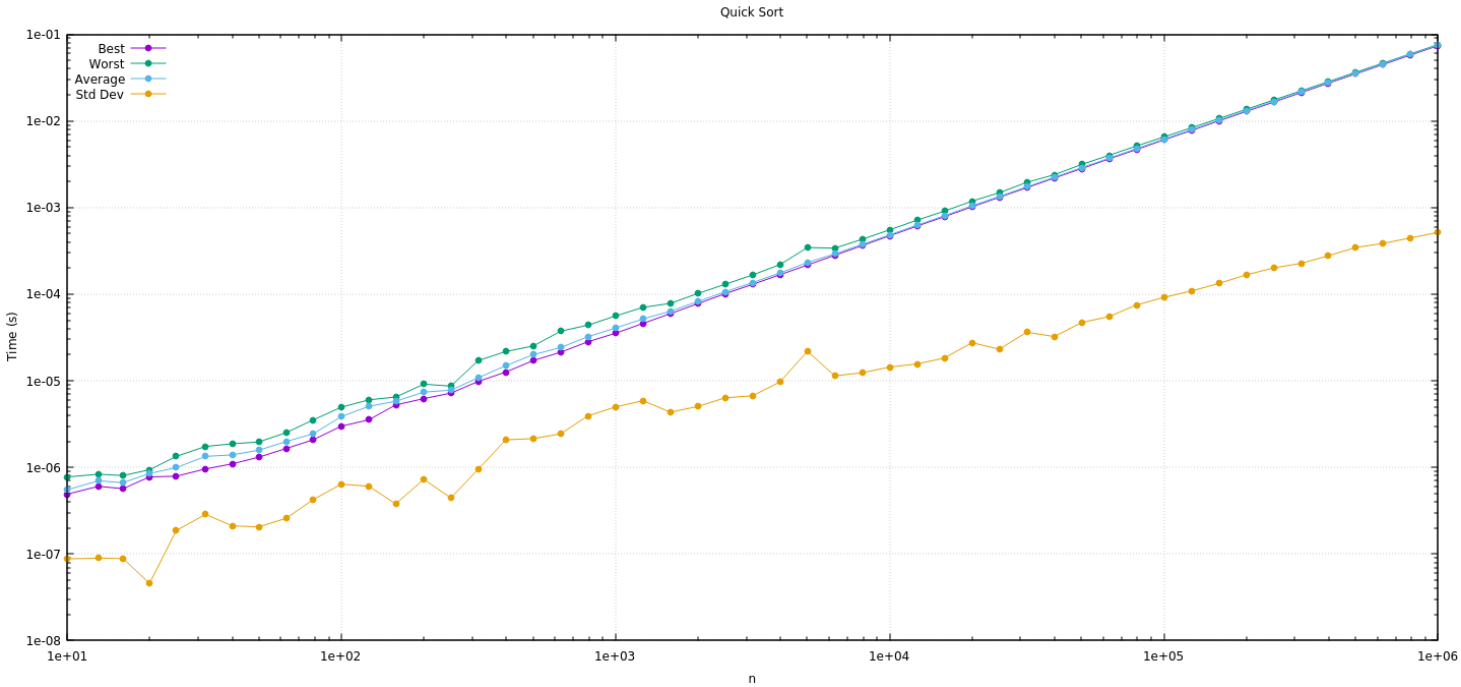

Figura 20 - Gráfico Quick Sort

Esta rotina, como se pode observar no gráfico, apresenta um crescimento temporal **O(nlog n)**, crescimento esse que no início apresenta algumas divergências, porém à medida que o n aumenta a reta torna-se mais linear, e também é notório realçar que as diferenças de tempos de execução entre *Best, Worst,* e *Average* tornam-se muito pequenas, como a variação do desvio padrão o comprova.

Esta conclusão é retirada através do gráfico da Fig.21, pois como podemos ver nos intervalos, 10^4 e 10^5 nos eixos de x, e de um pouco a baixo 10^{-3} e de 10^{-2}, em ambos existe um salto exponencial de 1, logo dessa forma fundamentamos a complexidade computacional do Quick Sort.

Criamos ainda o seguinte gráfico, onde é feita a *least squares fit,* esta implementação é muito poderosa, pois permite reduzir numa certa escala o número de informação e observar de forma nítida a tendência da curva. No *Quick Sort* os valores médios acompanham todos a tendência de crescimento de ***nlog(n).***

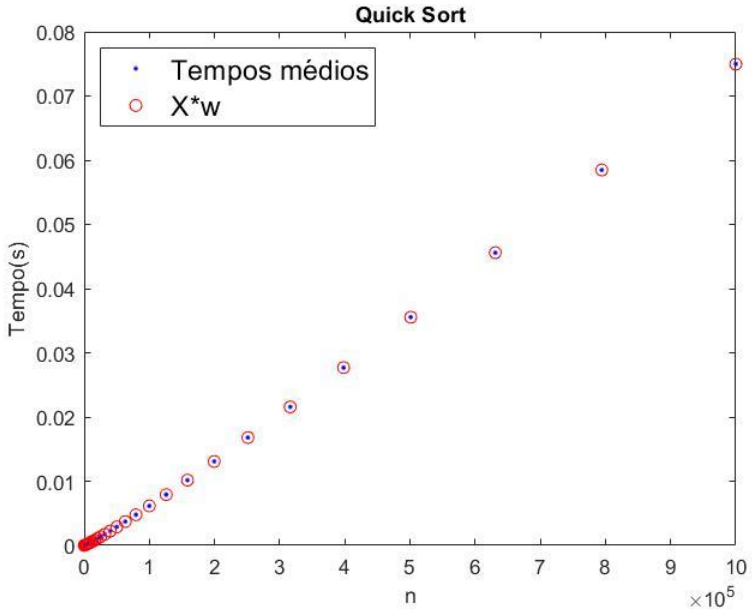

Figura 21 - least square Quick Sort

3.6 – Merge Sort

O *Merge Sort* (Fig.22) é um algoritmo de ordenação, que usa uma estratégia de *Divide and Conquer,* é implementado de forma recursiva, e em traços gerais pode ser descrita como: Primeiramente divide-se o *array* em 2 partes iguais (ou, parcialmente igual, se o tamanho do mesmo for um número ímpar), divide-se recursivamente até que fiquemos com cada elemento sozinho, e a partir desse momento, entra a parte de *Conquer*, onde se liga as partes dos sub-arrays ordenados. Esta rotina de ordenação tem uma complexidade computacional igual para os três casos, *Best, Avevage, e Worst*, **O(n log n)**, que iremos verificar mais a frente.

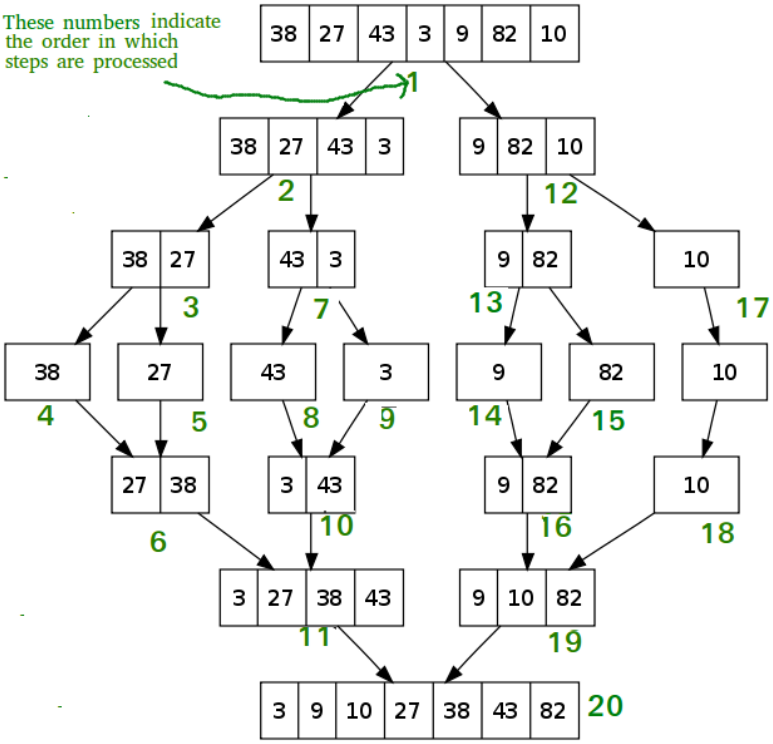

Figura 22 - Demonstração do funcionamento do Merge Sort

A tabela com os tempos de execução da rotina *Merge Sort*, é a seguinte (Fig.23):

```
# merge_sort
#       n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
       10 4.880e-07 6.350e-07 5.395e-07 5.059e-08
       13 6.230e-07 8.000e-07 6.536e-07 2.120e-08
       16 7.040e-07 1.094e-06 8.498e-07 1.525e-07
       20 7.100e-07 8.710e-07 8.172e-07 3.011e-08
       25 7.510e-07 1.013e-06 8.741e-07 8.367e-08
       32 8.070e-07 1.212e-06 1.001e-06 1.462e-07
       40 1.317e-06 1.552e-06 1.424e-06 5.827e-08
       50 1.276e-06 1.973e-06 1.561e-06 2.279e-07
       63 1.533e-06 1.990e-06 1.709e-06 1.359e-07
       79 2.022e-06 3.282e-06 2.352e-06 3.780e-07
      100 3.032e-06 4.201e-06 3.574e-06 2.862e-07
      126 3.342e-06 5.085e-06 3.820e-06 4.452e-07
      158 4.380e-06 5.198e-06 4.617e-06 1.944e-07
      200 5.332e-06 9.459e-06 6.929e-06 7.700e-07
      251 6.632e-06 8.272e-06 7.272e-06 4.271e-07
      316 9.014e-06 1.322e-05 1.021e-05 8.710e-07
      398 1.193e-05 2.137e-05 1.359e-05 1.401e-06
      501 1.501e-05 2.768e-05 1.785e-05 2.817e-06
      631 2.033e-05 2.966e-05 2.310e-05 2.084e-06
      794 2.573e-05 4.069e-05 2.895e-05 3.280e-06
     1000 3.300e-05 4.779e-05 3.720e-05 3.652e-06
     1259 4.345e-05 6.229e-05 4.795e-05 4.446e-06
     1585 5.743e-05 7.610e-05 6.107e-05 4.575e-06
     1995 7.333e-05 9.853e-05 7.683e-05 5.745e-06
     2512 9.613e-05 1.255e-04 9.946e-05 5.814e-06
     3162 1.243e-04 1.570e-04 1.303e-04 7.535e-06
     3981 1.590e-04 2.013e-04 1.648e-04 7.526e-06
     5012 2.063e-04 2.546e-04 2.139e-04 1.102e-05
     6310 2.730e-04 3.299e-04 2.802e-04 1.093e-05
     7943 3.485e-04 4.147e-04 3.554e-04 1.379e-05
    10000 4.496e-04 5.229e-04 4.587e-04 1.409e-05
    12589 5.951e-04 6.811e-04 6.033e-04 1.458e-05
    15849 7.596e-04 8.637e-04 7.694e-04 1.659e-05
    19953 9.750e-04 1.160e-03 9.924e-04 3.036e-05
    25119 1.289e-03 1.442e-03 1.301e-03 2.374e-05
    31623 1.645e-03 1.802e-03 1.660e-03 2.427e-05
    39811 2.109e-03 2.429e-03 2.137e-03 4.466e-05
    50119 2.778e-03 3.024e-03 2.799e-03 3.583e-05
    63096 3.539e-03 3.932e-03 3.569e-03 5.503e-05
    79433 4.532e-03 4.970e-03 4.575e-03 7.761e-05
   100000 5.950e-03 6.416e-03 6.006e-03 9.706e-05
   125893 7.572e-03 8.081e-03 7.645e-03 1.178e-04
   158489 9.685e-03 1.022e-02 9.780e-03 1.274e-04
   199526 1.268e-02 1.325e-02 1.279e-02 1.403e-04
   251189 1.615e-02 1.683e-02 1.631e-02 1.734e-04
   316228 2.061e-02 2.396e-02 2.090e-02 5.376e-04
   398107 2.694e-02 2.793e-02 2.719e-02 2.255e-04
   501187 3.424e-02 3.563e-02 3.462e-02 3.224e-04
   630957 4.382e-02 6.380e-02 4.787e-02 5.084e-03
   794328 5.705e-02 5.871e-02 5.768e-02 3.912e-04
  1000000 7.266e-02 7.435e-02 7.337e-02 3.781e-04
#------- --------- --------- --------- ---------
```

Figura 23 - Tabela Merge Sort

O gráfico com os tempos de execução da rotina Merge Sort, é a seguinte (Fig.24):

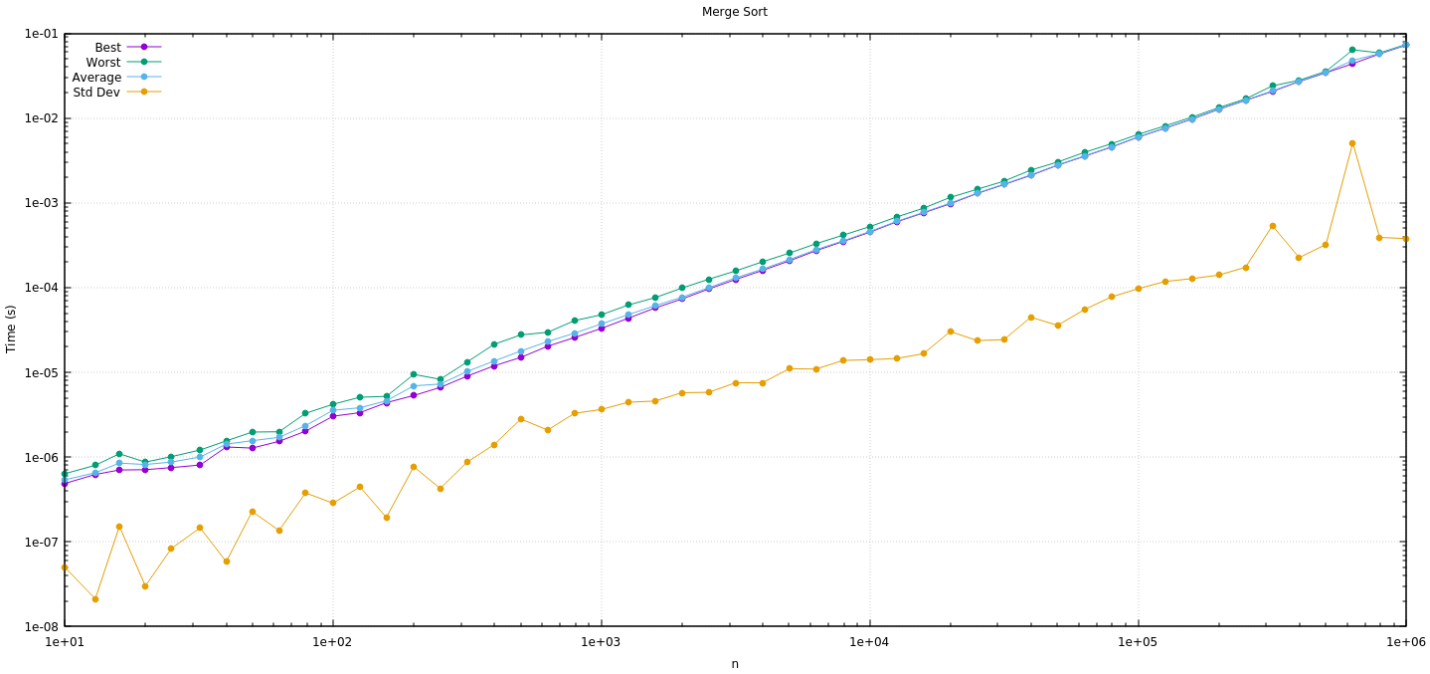

Figura 24 - Gráfico Merge Sort

De acordo com o gráfico da (Fig.24), a estratégia de ordenação, Merge Sort, apresenta no seu todo um crescimento na ordem de **O(nlog n)**, tal como nos outros algoritmos no início é mais linear que exponencial, no entanto à medida que o n aumenta o nível de crescimento exponencial acompanha esse aumento de forma cada vez mais assente. É visível que, por análise do gráfico, o Worst Case, tem algumas oscilações, para valores de n pequenos, e algumas para valores de n mais elevados.

O fundamento da complexidade computacional é retirada através do gráfico da Fig.25, pois como podemos ver nos intervalos, 10^4 e 10^5 nos eixos de x, e de um pouco a baixo 10^{-3} e de 10^{-2}, em ambos existe um salto exponencial de 1, logo dessa forma fundamentamos a complexidade computacional do Merge Sort.

Fizemos ainda uma comparação do Merge Sort com o Merge Sort, pois achamos pertinente a comparação entre estas 2 rotinas, ambas usam Divide and Conquer, e têm tempos de execução parecidos. Na comparação direta consegue-se ver que o Quick Sort é em toda a escala melhor a nível de tempo de execução que o Merge Sort, a diferença não é muito substancial, porém é notável.

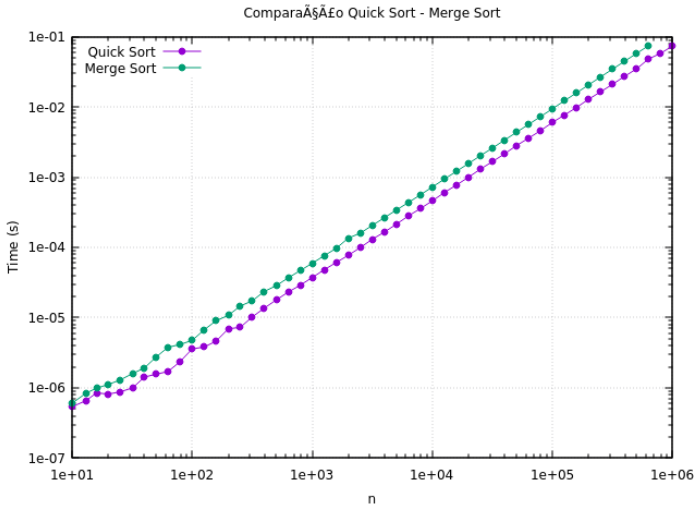

Figura 25 - Quick Sort / Merge Sort

Criamos ainda o seguinte gráfico, onde é feita a *least squares fit,* esta implementação é muito poderosa, pois permite reduzir numa certa escala o número de informação e observar de forma nítida a tendência da curva. No Merge Sort os valores médios acompanham quase todos todos a tendência de crescimento de **nlog(n)**, temos alguns pontos a não seguir tão a risca o crescimento "normal", no entanto o crescimento dos tempos de execução mantem-se.

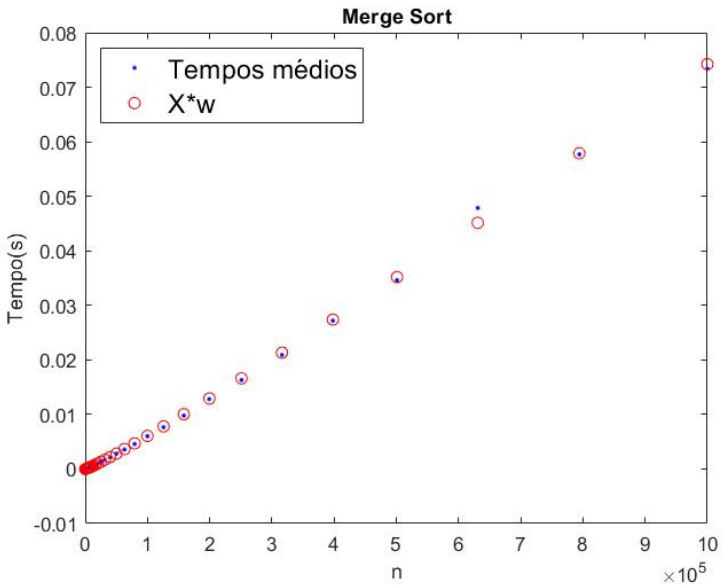

Figura 26 - Least Squares Fit Merge Sort

3.7 – Heap Sort

Um Max-Heap é uma árvore binária na qual o nó pai tem um valor superior aos seus respetivos nós filhos. O Heap Sort é uma técnica de ordenação baseada em comparação e que utiliza um Max-Heap. Para esta técnica ser utilizada começamos por criar um Max-Heap com os dados fornecidos sendo que, o maior item vai ficar guardado na raiz do Heap. De seguida vamos trocar o valor da raiz pelo último valor, retiramos o último elemento e voltamos a transformar a árvore binária num Max-Heap. Por fim repetimos este procedimento até o tamanho do array ser igual a 1. O Heap Sort, tem uma complexidade computacional para o *Best , o Average* e para o *Worst* igual a **O(n log n).**

A tabela com os tempos de execução para o *Heap Sort* é a seguinte (Fig.27):

```
# heap_sort
#      n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
      10 5.620e-07 6.390e-07 6.034e-07 1.858e-08
      13 7.480e-07 9.430e-07 8.359e-07 4.042e-08
      16 7.690e-07 1.321e-06 9.898e-07 2.059e-07
      20 8.960e-07 1.508e-06 1.116e-06 1.910e-07
      25 1.028e-06 1.685e-06 1.289e-06 2.446e-07
      32 1.287e-06 1.926e-06 1.602e-06 1.990e-07
      40 1.568e-06 2.636e-06 1.916e-06 3.484e-07
      50 2.241e-06 4.075e-06 2.743e-06 5.220e-07
      63 2.664e-06 4.727e-06 3.786e-06 5.928e-07
      79 3.651e-06 4.798e-06 4.133e-06 3.054e-07
     100 4.136e-06 5.828e-06 4.750e-06 4.093e-07
     126 5.565e-06 9.511e-06 6.684e-06 1.041e-06
     158 7.394e-06 1.140e-05 9.011e-06 8.666e-07
     200 9.443e-06 1.607e-05 1.075e-05 1.604e-06
     251 1.193e-05 2.161e-05 1.460e-05 2.671e-06
     316 1.522e-05 2.426e-05 1.726e-05 1.870e-06
     398 1.994e-05 3.213e-05 2.348e-05 2.704e-06
     501 2.591e-05 3.615e-05 2.855e-05 2.772e-06
     631 3.357e-05 4.917e-05 3.683e-05 3.812e-06
     794 4.365e-05 5.861e-05 4.667e-05 3.630e-06
    1000 5.545e-05 7.511e-05 5.944e-05 4.878e-06
    1259 7.195e-05 9.515e-05 7.610e-05 5.853e-06
    1585 9.318e-05 1.186e-04 9.711e-05 6.271e-06
    1995 1.211e-04 1.670e-04 1.351e-04 1.402e-05
    2512 1.531e-04 1.936e-04 1.594e-04 7.917e-06
    3162 1.976e-04 2.529e-04 2.052e-04 1.128e-05
    3981 2.560e-04 3.143e-04 2.633e-04 1.187e-05
    5012 3.303e-04 4.175e-04 3.402e-04 1.741e-05
    6310 4.267e-04 5.032e-04 4.342e-04 1.303e-05
    7943 5.518e-04 6.510e-04 5.579e-04 1.468e-05
   10000 7.128e-04 8.249e-04 7.216e-04 1.577e-05
   12589 9.219e-04 1.075e-03 9.350e-04 2.489e-05
```

Figura 27- Tabela Heap Sort

```
 15849 1.194e-03 1.332e-03 1.207e-03 2.096e-05
 19953 1.540e-03 1.761e-03 1.555e-03 2.591e-05
 25119 1.989e-03 2.265e-03 2.015e-03 4.141e-05
 31623 2.572e-03 2.821e-03 2.594e-03 3.469e-05
 39811 3.317e-03 3.659e-03 3.350e-03 5.057e-05
 50119 4.284e-03 4.764e-03 4.328e-03 7.569e-05
 63096 5.544e-03 6.232e-03 5.599e-03 1.079e-04
 79433 7.173e-03 7.714e-03 7.241e-03 1.056e-04
100000 9.295e-03 1.004e-02 9.387e-03 1.329e-04
125893 1.207e-02 1.284e-02 1.220e-02 1.915e-04
158489 1.565e-02 1.643e-02 1.580e-02 1.880e-04
199526 2.028e-02 2.116e-02 2.051e-02 2.747e-04
251189 2.629e-02 2.724e-02 2.658e-02 3.052e-04
316228 3.401e-02 3.517e-02 3.441e-02 3.464e-04
398107 4.401e-02 4.565e-02 4.452e-02 3.986e-04
501187 5.692e-02 5.841e-02 5.752e-02 3.650e-04
630957 7.348e-02 7.530e-02 7.416e-02 4.031e-04
#------- --------- --------- --------- ---------
```

Figura 28 – Tabela Heap Sort

O gráfico com os tempos de execução para o *Heap Sort* é a seguinte (Fig.26):

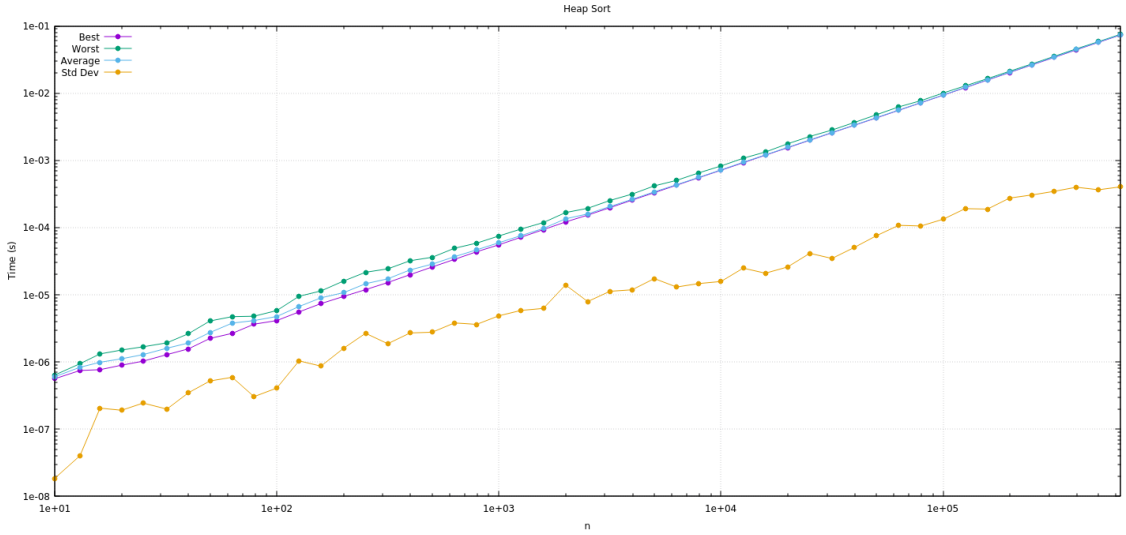

Figura 29- Gráfico Heap Sort

Ao analisamos o gráfico acima, observamos que o *Heap Sort*, cresce de forma exponencial de aproximadamente **nlog(n)**, sendo que tem alguns picos no ínicio.

Criamos ainda o seguinte gráfico, onde é feita a *least squares fit,* esta implementação é muito poderosa, pois permite reduzir numa certa escala o número de informação e observar de forma nítida a tendência da curva. No *Heap Sort* os valores médios acompanham todos a tendência de crescimento de **n log(n)**.

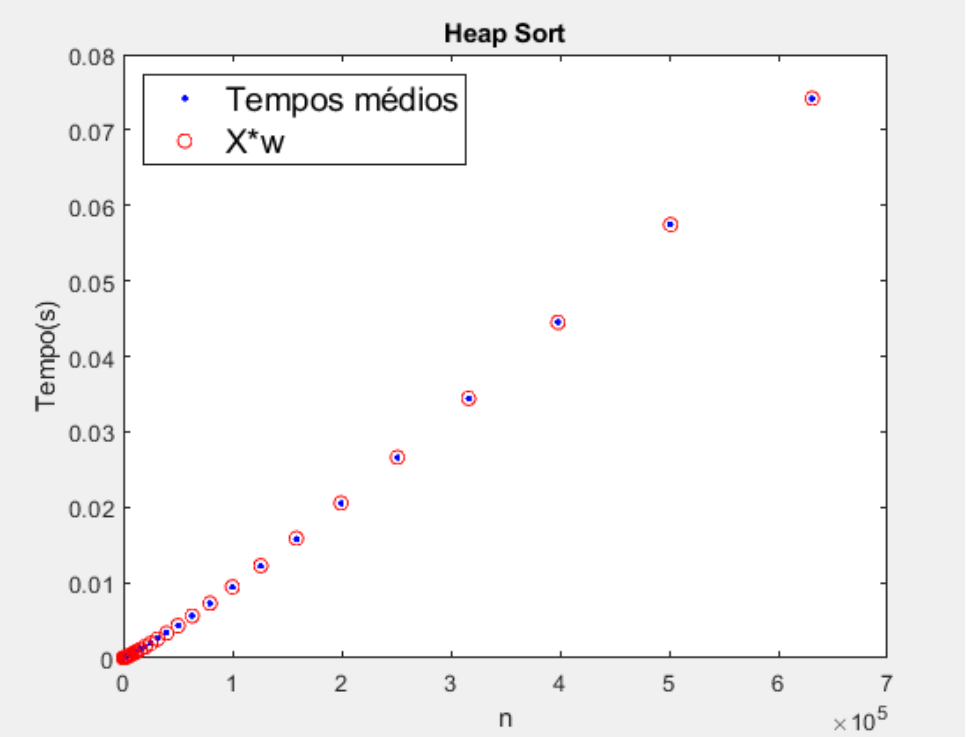

Figura 30- Least Square Fit Heap Sort

3.8 – Rank Sort

Um algoritmo paralelo é um algoritmo no qual pode ser executado diferentes partes cada um na sua vez em diferentes dispositivos processadores e no fim ser tudo novamente junto para assim se obter o resultado correto. O Rank sort é um simples algoritmo paralelo de ordenação no qual cada elemento de um array é comparado com todos os outros elementos do array com o intuito deverificar qual é o maior. O rank de um elemento é definido pelo número total de elementos menores que ele mesmo. A posição final de um elemento no array organizado é o seu rank. O Rank Sort, tem uma complexidade computacional para o *Best , o Average* e para o *Worst* igual a **O(n^2).**

A tabela com os tempos de execução para o *Rank Sort* é a seguinte (Fig.31):

```
# rank_sort
#      n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
      10 4.810e-07 5.120e-07 4.905e-07 7.729e-09
      13 5.250e-07 1.015e-06 8.230e-07 1.353e-07
      16 7.360e-07 1.113e-06 9.244e-07 1.432e-07
      20 9.950e-07 1.170e-06 1.046e-06 2.992e-08
      25 1.052e-06 1.297e-06 1.207e-06 5.416e-08
      32 1.280e-06 1.848e-06 1.472e-06 1.236e-07
      40 1.640e-06 1.928e-06 1.756e-06 6.426e-08
      50 2.271e-06 3.344e-06 2.703e-06 2.512e-07
      63 2.194e-06 4.086e-06 3.314e-06 5.756e-07
      79 4.942e-06 8.067e-06 6.457e-06 7.238e-07
     100 4.758e-06 8.959e-06 7.237e-06 1.058e-06
     126 7.773e-06 1.207e-05 9.165e-06 1.040e-06
     158 1.113e-05 1.935e-05 1.341e-05 1.786e-06
     200 1.696e-05 2.658e-05 1.902e-05 1.885e-06
     251 2.720e-05 5.048e-05 3.346e-05 5.086e-06
     316 4.170e-05 6.145e-05 4.576e-05 3.702e-06
     398 6.415e-05 8.776e-05 6.958e-05 5.167e-06
     501 1.006e-04 1.348e-04 1.067e-04 7.616e-06
     631 1.565e-04 1.972e-04 1.640e-04 7.228e-06
     794 2.459e-04 3.001e-04 2.566e-04 1.042e-05
    1000 3.895e-04 4.691e-04 4.040e-04 1.473e-05
    1259 6.173e-04 7.327e-04 6.356e-04 1.871e-05
    1585 9.799e-04 1.161e-03 1.008e-03 2.995e-05
    1995 1.550e-03 1.739e-03 1.586e-03 2.813e-05
    2512 2.463e-03 2.738e-03 2.511e-03 3.823e-05
    3162 3.911e-03 4.327e-03 3.982e-03 6.486e-05
    3981 6.207e-03 6.751e-03 6.328e-03 1.089e-04
    5012 9.842e-03 1.046e-02 1.000e-02 1.361e-04
    6310 1.562e-02 1.640e-02 1.584e-02 1.812e-04
    7943 2.484e-02 2.790e-02 2.543e-02 6.262e-04
   10000 3.941e-02 4.086e-02 3.989e-02 3.169e-04
   12589 6.259e-02 6.452e-02 6.330e-02 4.308e-04
```

Figura 31 - Tabela Rank Sort

O gráfico com os tempos de execução para o *Rank Sort* é a seguinte (Fig.32):

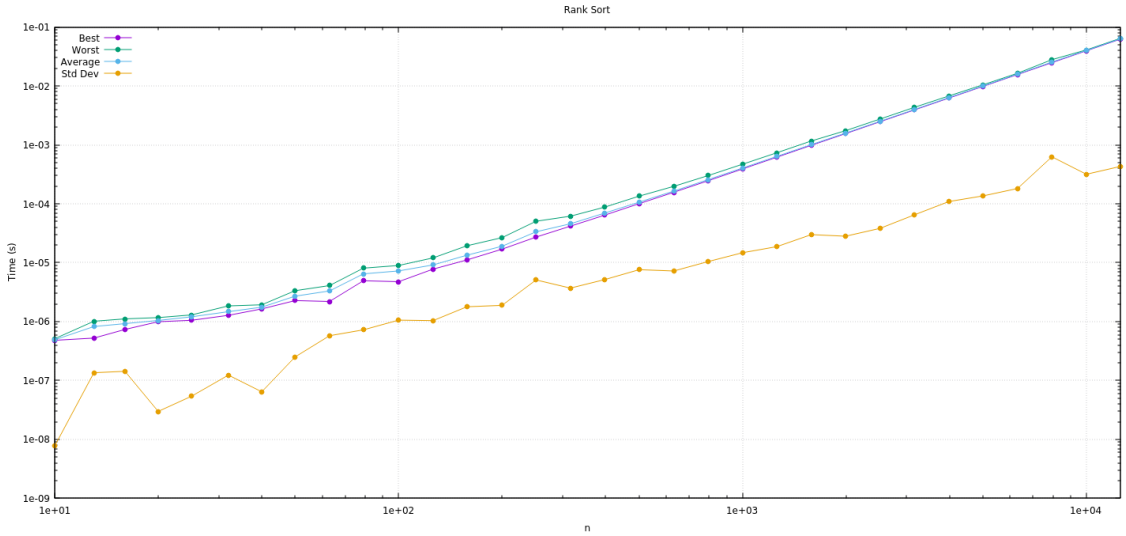

Figura 32 - Gráfico Rank Sort

Ao analisamos o gráfico acima, observamos que o *Rank Sort*, cresce de forma exponencial de aproximadamente $\mathbf{n^2}$.

Criamos ainda o seguinte gráfico, onde é feita a *least squares fit*, esta implementação é muito poderosa, pois permite reduzir numa certa escala o número de informação e observar de forma nítida a tendência da curva. No *Rank Sort* os valores médios acompanham todos a tendência de crescimento de $\mathbf{n^2}$.

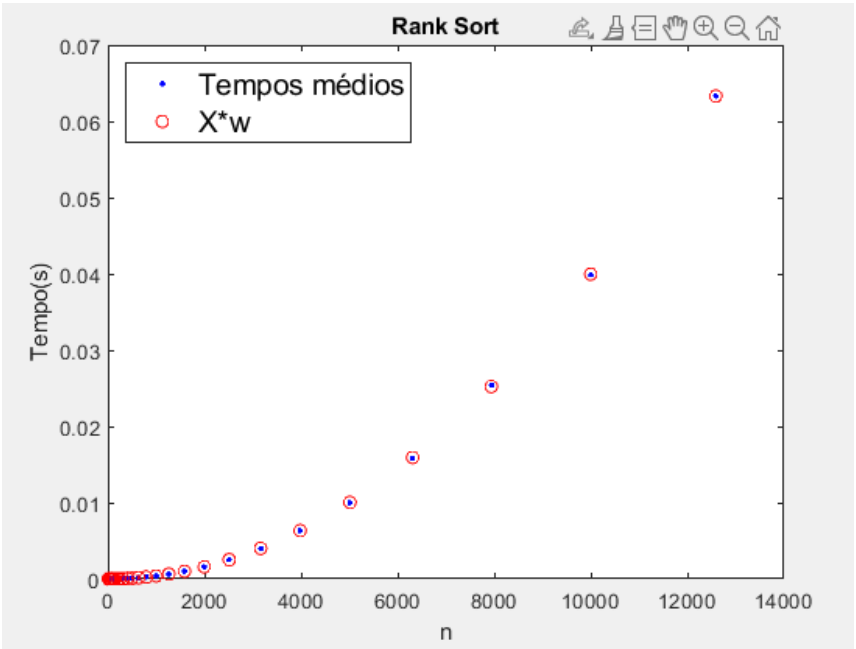

Figura 33 - Least Squares Fit Rank Sort

3.9 – Selection Sort

Selection Sort(Fig. 34) é um algoritmo que organiza um array ao repetidamente encontrar o elemento mínimo da parte não organizada e coloca-lo no início. Este algoritmo mantem dois sub-arrays num array, sendo estes o sub-array com a parte já organizada e o sub-array com a parte que ainda falta organizar. O Selection Sort, tem uma complexidade computacional para o *Best , o Average* e para o *Worst* igual a **O(n^2)**.

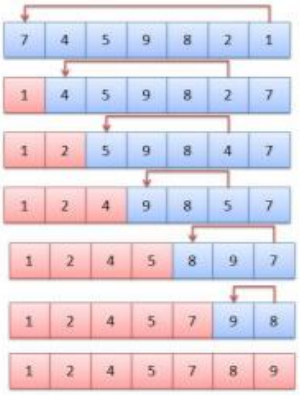

Figura 34 - Selection Sort

A tabela com os tempos de execução para o *Selection Sort* é a seguinte (Fig.35):

```
# selection_sort
#      n  min time  max time  avg time   std dev
#------- --------- --------- --------- ---------
      10 5.200e-07 5.510e-07 5.341e-07 7.063e-09
      13 6.160e-07 7.980e-07 7.578e-07 4.274e-08
      16 7.790e-07 9.400e-07 8.416e-07 5.609e-08
      20 9.500e-07 1.146e-06 1.013e-06 6.380e-08
      25 1.248e-06 1.334e-06 1.286e-06 2.291e-08
      32 1.669e-06 2.205e-06 1.896e-06 1.950e-07
      40 2.106e-06 2.898e-06 2.600e-06 2.153e-07
      50 3.030e-06 5.634e-06 4.502e-06 8.971e-07
      63 4.757e-06 7.121e-06 5.514e-06 6.885e-07
      79 6.640e-06 9.334e-06 7.936e-06 9.169e-07
     100 1.056e-05 1.878e-05 1.155e-05 1.081e-06
     126 1.631e-05 2.892e-05 1.947e-05 2.775e-06
     158 2.520e-05 3.967e-05 2.840e-05 3.917e-06
     200 4.011e-05 5.369e-05 4.366e-05 3.424e-06
     251 6.127e-05 7.673e-05 6.459e-05 3.683e-06
     316 9.701e-05 1.309e-04 1.015e-04 7.231e-06
     398 1.506e-04 1.780e-04 1.558e-04 4.842e-06
     501 2.389e-04 2.894e-04 2.457e-04 9.080e-06
     631 3.798e-04 4.268e-04 3.862e-04 8.825e-06
     794 6.033e-04 6.547e-04 6.079e-04 9.314e-06
    1000 9.599e-04 1.070e-03 9.702e-04 1.812e-05
    1259 1.526e-03 1.655e-03 1.540e-03 2.282e-05
    1585 2.424e-03 2.549e-03 2.440e-03 2.345e-05
    1995 3.850e-03 4.103e-03 3.886e-03 5.184e-05
    2512 6.116e-03 6.415e-03 6.164e-03 6.474e-05
    3162 9.706e-03 1.004e-02 9.775e-03 8.640e-05
    3981 1.541e-02 1.583e-02 1.552e-02 1.128e-04
    5012 2.445e-02 2.506e-02 2.463e-02 1.599e-04
    6310 3.879e-02 3.944e-02 3.898e-02 1.619e-04
    7943 6.154e-02 6.261e-02 6.192e-02 2.325e-04
```

Figura 35 - Tabela Selection Sort

O gráfico com os tempos de execução para o *Selection Sort* é a seguinte (Fig.36):

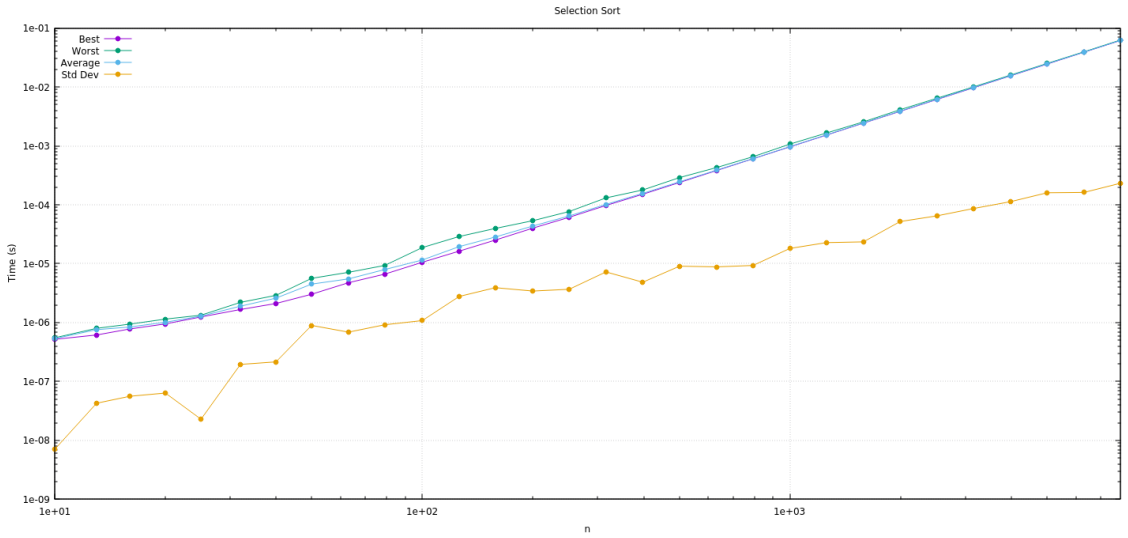

Figura 36 - Gráfico Selection Sort

Ao analisamos o gráfico acima, observamos que o *Selection Sort*, cresce de forma exponencial de aproximadamente $\mathbf{n^2}$, sendo que também se torna mais linear á medida que o ***n*** cresce.

Criamos ainda o seguinte gráfico, onde é feita a *least squares fit*, esta implementação muito poderosa, pois permite reduzir numa certa escala o número de informação e observar de forma nítida a tendência da curva. No *Selection Sort* os valores médios acompanham todos a tendência de crescimento de $\mathbf{n^2}$.

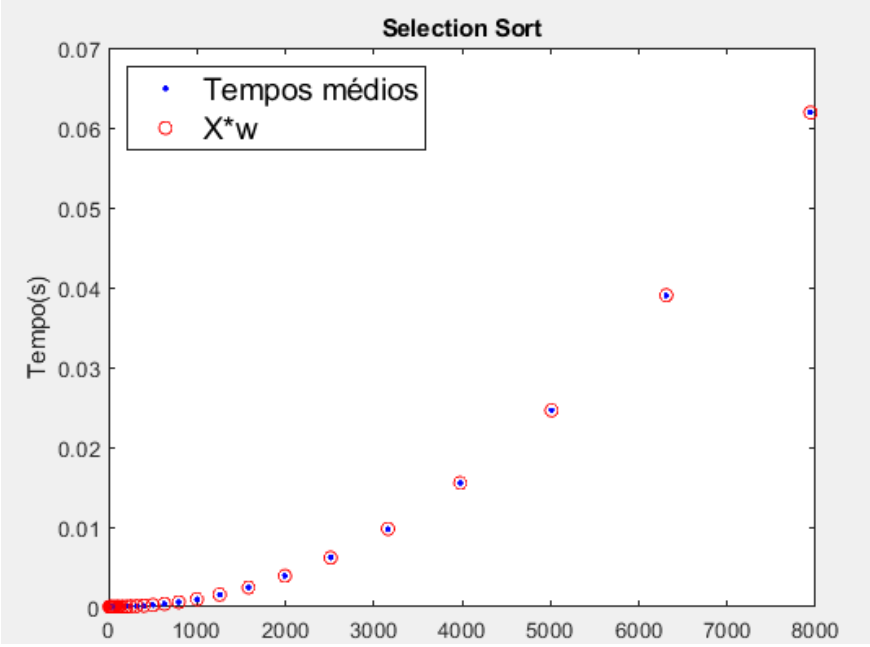

Figura 37 - Least Squares Fit Selection Sort

3.10 – Resultados Totais

No que toca à observação de todas as estratégias em simultâneo, criamos o seguinte gráfico (Fig.38), o gráfico é referente ao melhor tempo de todas as rotinas de ordenação.

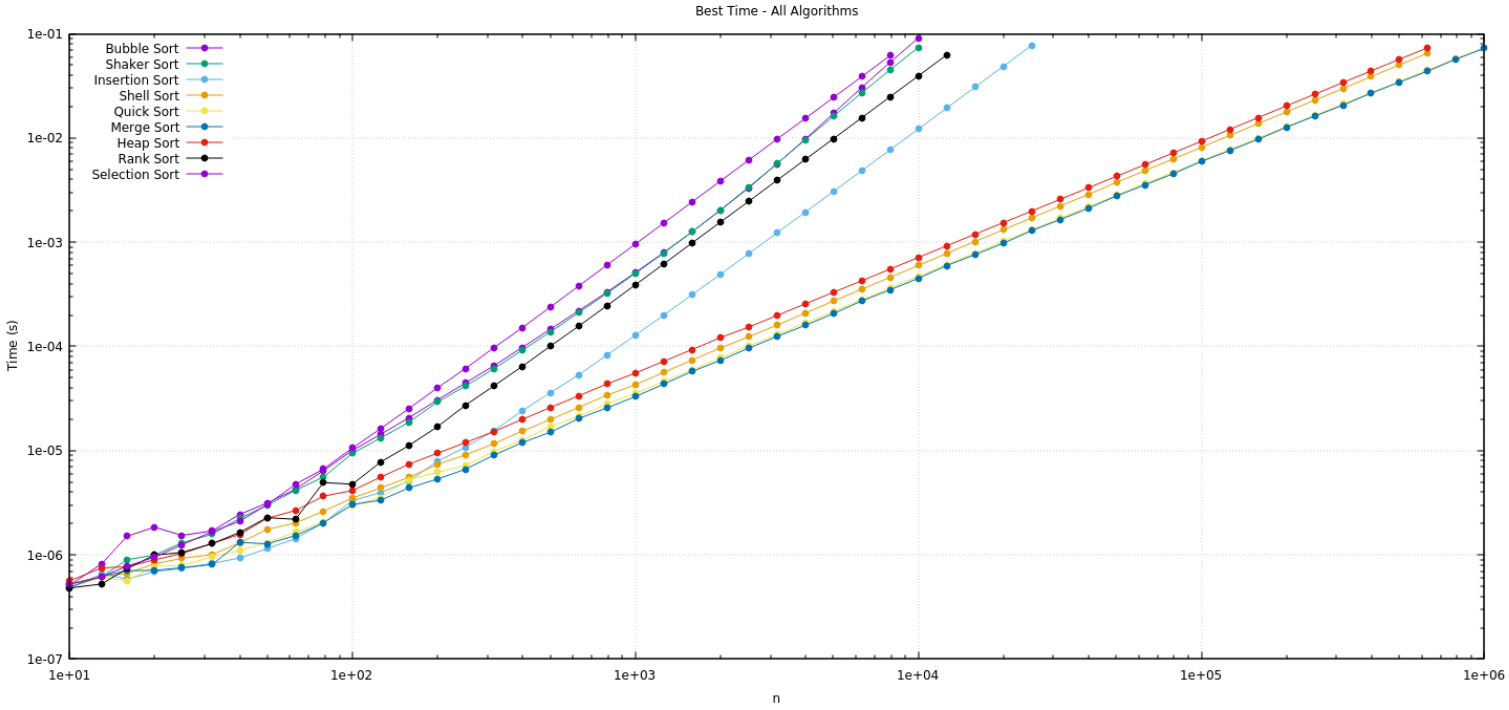

Figura 38- Melhor tempo de todas as estratégias

De acordo com o gráfico, em cima, podemos concluir que a melhor rotina de ordenação, pois é a mais rápida, é a estratégia *Quick Sort*. Pode parecer e até confundir-se um pouco com o gráfico com os tempos do Merge Sort, no entanto, o Quick Sort consegue ter tempos mais pequenos.

Com a análise deste gráfico, conseguimos retirar conclusões relativamente a um determinado "grupo" de rotinas de ordenação, estratégias como *Bubble Sort, Rank Sort, Shaker Sort*, entre outras, que são computacionalmente más, pois o seu tempo de execução para uma pequena quantidade de números (menores que 100), chegam a ser semelhantes com as restantes rotinas, porém para valores maiores que esses, são significativamente más rotinas, pois são muito lentas. Como é visível no gráfico, por exemplo enquanto o *Heap Sort*, consegue quase ordenar um milhão de valores, o *Rank Sort,* consegue ordenar pouco mais de dez mil números.

Nesta ordem de ideias, e em tom de conclusão podemos dizer que as melhores rotinas para uma grande escala de número são o Quick Sort, e o Merge Sort, sendo que o Quick Sort apresenta uma pequena vantagem em relação ao Merge Sort, porém também é importante referir que estas rotinas apresentam um grau de complexidade maior em relação à sua implementação. No que toca a ordenação de estruturas pequenas, a melhor opção é o Insertion Sort, quer pela sua fácil capacidade de implementação quer pela sua rapidez.

Ora, a técnica *Divide and Conquer* consiste em resolver um problema recursivamente através de três etapas. Esta técnica é bastante vantajosa pois permite nos resolver problemas com um elevado número de dificuldade, e usar algoritmos bastantes eficazes. Algumas das rotinas de ordenação que seguem o algoritmo *Devide and Conquer* são o *Quick Sort* e o *Merge Sort.* Como estas rotinas são as que apresentam o melhor tempo de execução podemos concluir que o algoritmo *Divide and Conquer* é verdadeiramente eficaz.

Conclusão

Após a realização deste trabalho prático pudemos compreender o funcionamento das variadas rotinas de ordenação, ainda conseguimos observar o seu modo de crescimento através de gráficos, comparar com os valores teóricos e comparar as diversas rotinas de ordenação de forma a perceber quais são as mais eficazes e entender porquê.

Concluímos ainda que os métodos mais eficazes são o *Quick Sort* e o *Merge Sort* e os menos vantajosos são o *Bubble Sort e o Shaker Sort,* sendo que para valores mais pequenos, arrays com tamanho até 20 ou 30 elementos o *Insertion Sort, é o mais competente.*

De forma geral cumprimos com os objetivos previamente definidos, complementamos os conceitos teóricos com os práticos.

Bibliografia

Silva, Tomás Oliveira e. Lecture notes: Algorithms and Data Structute (AED - Algoritmos e Estruturas de Dados), LEI, MIEC, 2020/2021