universidade de aveiro

Algoritmos e Estruturas de Dados

Trabalho Prático 2

21/01/2021
Licenciatura em Engenharia Informática

Diana Elisabete Siso Oliveira, nº 98607, P2 (40%)
Miguel Rocha Ferreira, nº 98599, P2 (30%)
Paulo Guilherme Soares Pereira, nº 98430, P2 (30%)

Índice

1. Introdução

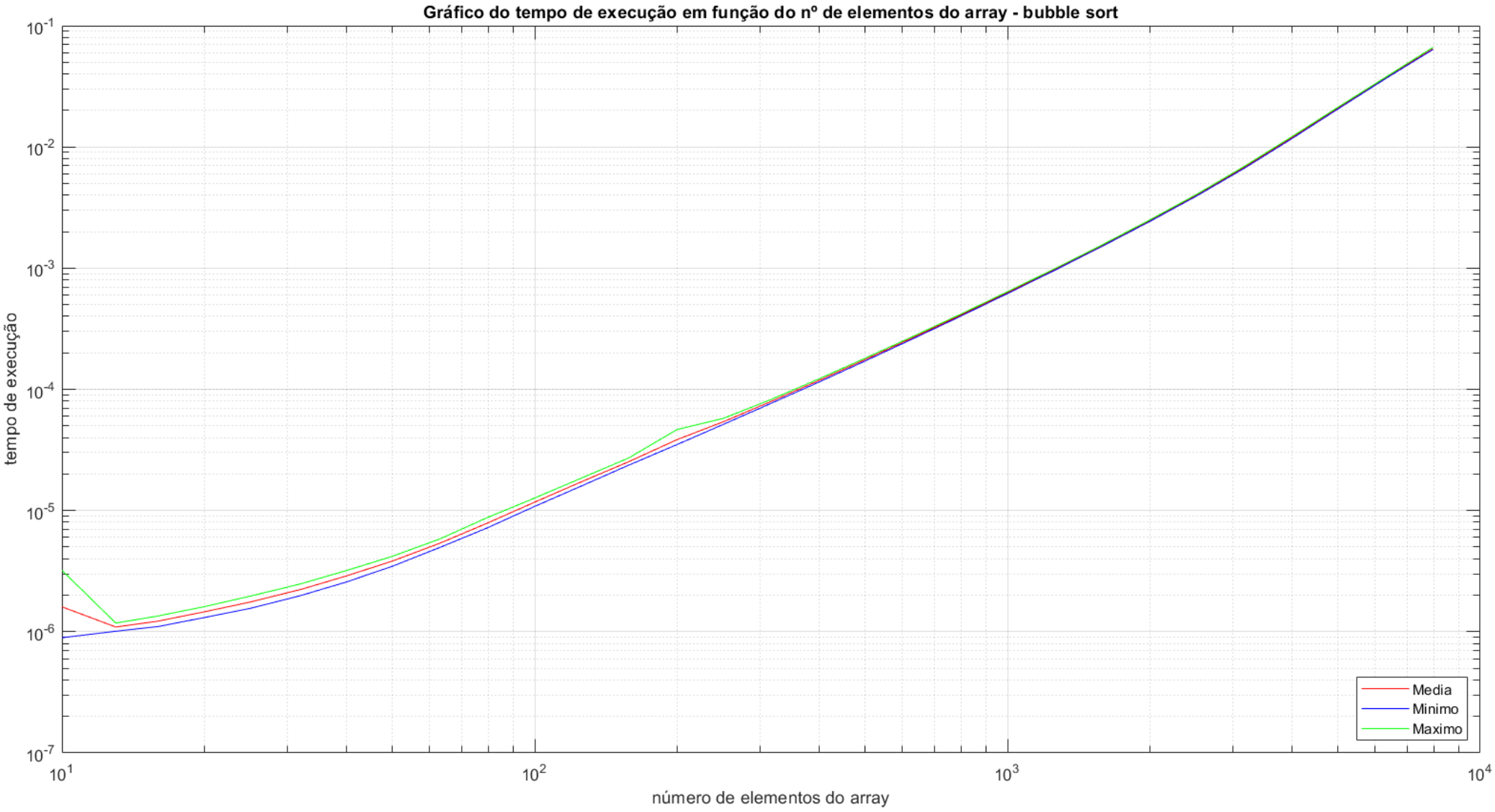
Com o desenvolvimento do presente projeto, visamos aprimorar os nossos conhecimentos sobre algoritmos de ordenação de forma a concluir qual algoritmo é mais eficiente.

Para tal, através de informações fornecidas previamente, é esperada a visualização do comportamento de cada algoritmo de ordenação no melhor caso, pior caso e caso médio através de tabelas e gráficos, assim como as análises do funcionamento de cada um deles e as comparações quanto às diferenças apresentadas.

Assim, contemplando as ideias base do enunciado, auxiliando-nos de código em MATLAB e após a compilação do programa *sorting_methods.c*, foram analisados os gráficos referentes a cada algoritmo e as respetivas conclusões.

2. Bubble Sort

n	min time	max time	avg time	std dev
10	8.920e-07	3.210e-06	1.605e-06	9.232e-07
13	1.009e-06	1.180e-06	1.094e-06	4.077e-08
16	1.105e-06	1.350e-06	1.224e-06	5.702e-08
20	1.309e-06	1.611e-06	1.459e-06	7.057e-08
25	1.557e-06	1.964e-06	1.757e-06	9.206e-08
32	1.988e-06	2.484e-06	2.232e-06	1.152e-07
40	2.575e-06	3.199e-06	2.888e-06	1.510e-07
50	3.469e-06	4.192e-06	3.829e-06	1.646e-07
63	4.959e-06	5.828e-06	5.372e-06	2.030e-07
79	7.133e-06	8.673e-06	7.806e-06	3.231e-07
100	1.082e-05	1.268e-05	1.173e-05	4.376e-07
126	1.603e-05	1.863e-05	1.737e-05	6.467e-07
158	2.366e-05	2.711e-05	2.526e-05	8.282e-07
200	3.501e-05	4.658e-05	3.844e-05	3.089e-06
251	5.152e-05	5.771e-05	5.415e-05	1.320e-06
316	7.652e-05	8.243e-05	7.933e-05	1.368e-06
398	1.141e-04	1.214e-04	1.177e-04	1.662e-06
501	1.719e-04	1.814e-04	1.765e-04	2.211e-06
631	2.612e-04	2.731e-04	2.670e-04	2.830e-06
794	3.992e-04	4.147e-04	4.067e-04	3.684e-06
1000	6.169e-04	6.391e-04	6.268e-04	5.121e-06
1259	9.592e-04	9.896e-04	9.740e-04	7.387e-06
1585	1.511e-03	1.556e-03	1.532e-03	1.090e-05
1995	2.413e-03	2.486e-03	2.448e-03	1.726e-05
2512	3.935e-03	4.061e-03	3.997e-03	2.927e-05
3162	6.631e-03	6.877e-03	6.753e-03	5.579e-05
3981	1.160e-02	1.204e-02	1.182e-02	1.047e-04
5012	2.068e-02	2.139e-02	2.104e-02	1.718e-04
6310	3.663e-02	3.766e-02	3.714e-02	2.284e-04
7943	6.371e-02	6.636e-02	6.455e-02	4.422e-04



Equação da reta correspondente ao valor médio: $y = 1.224e^{-09}x^2 + 1.774e^{-06}x - 0.0003777$

O algoritmo Bubble Sort é um dos algoritmos mais simples de ordenação sendo até a base para outros algoritmos como o Shaker Sort. Este algoritmo ordena numa única direção.

Dado um *array*, iremos comparar o seu primeiro elemento com o segundo elemento. Se o primeiro elemento for maior que o segundo elemento, eles irão trocar de posição e continuamos com a comparação. Em seguida compara o segundo elemento com o terceiro e aplica-se a mesma lógica, e procede se para a comparação entre o terceiro e quarto elemento e estas comparações acontecem até ao fim do array, ponto em que o array vai ser submetido novamente ao Bubble Sort até que durante uma das passagens do array não tenha nenhuma troca de elementos.

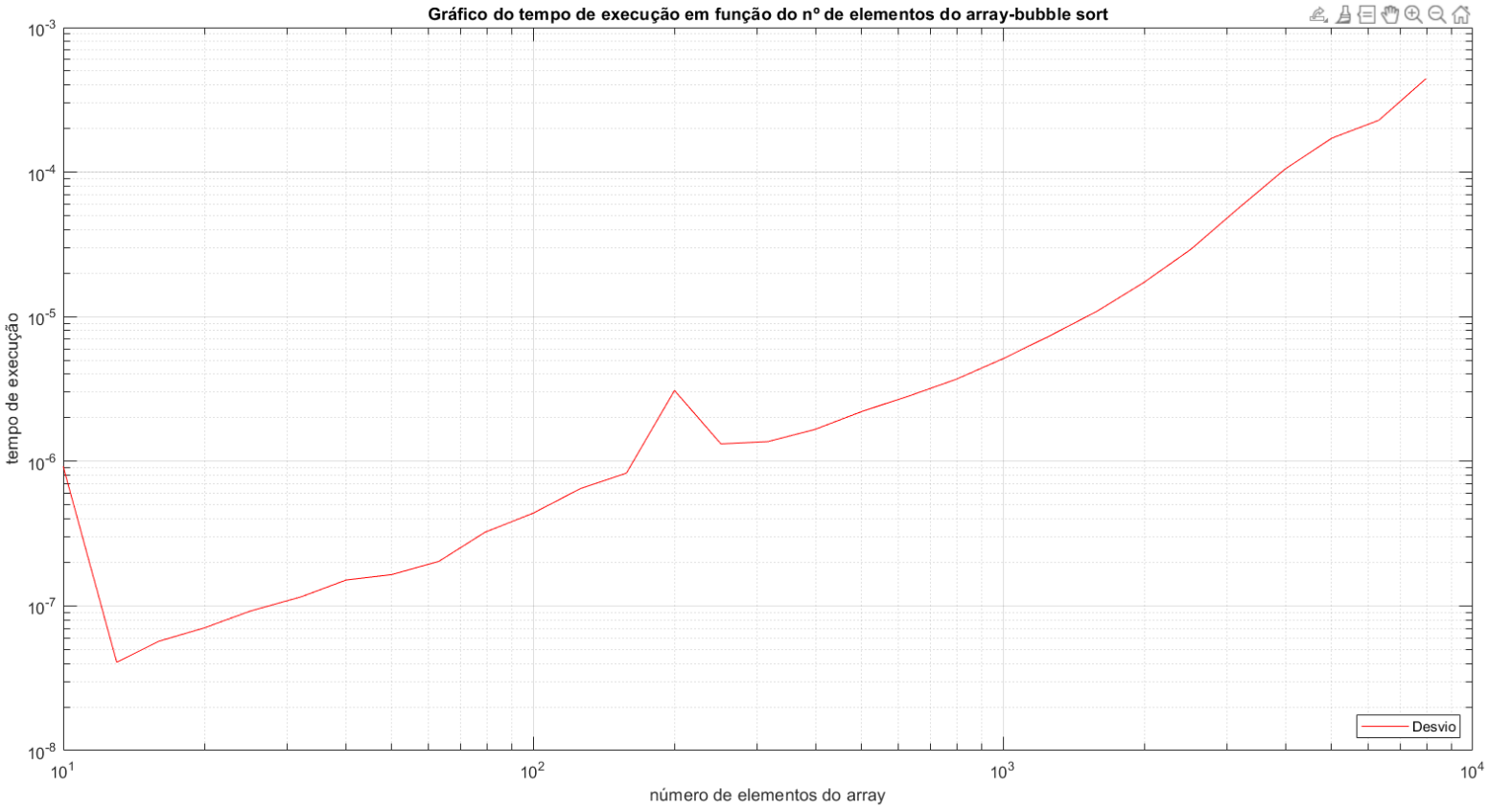
Em seguida é mostrado um exemplo de como funciona o Bubble Sort:

8	5	3	1	7
5	8	3	1	7
5	3	8	1	7
5	3	1	8	7
5	3	1	7	8
3	5	1	7	8
3	1	5	7	8
3	1	5	7	8
3	1	5	7	8
1	3	5	7	8
1	3	5	7	8
1	3	5	7	8
1	3	5	7	8
1	3	5	7	8
1	3	5	7	8
1	3	5	7	8
1	3	5	7	8

Este algoritmo sendo um algoritmo bastante simples tem como melhor caso uma complexidade (em Big O Notation) O(n), este caso acontece quando o array já está na ordem correta, para o pior e caso médio apresenta uma complexidade de $O(n^2)$.

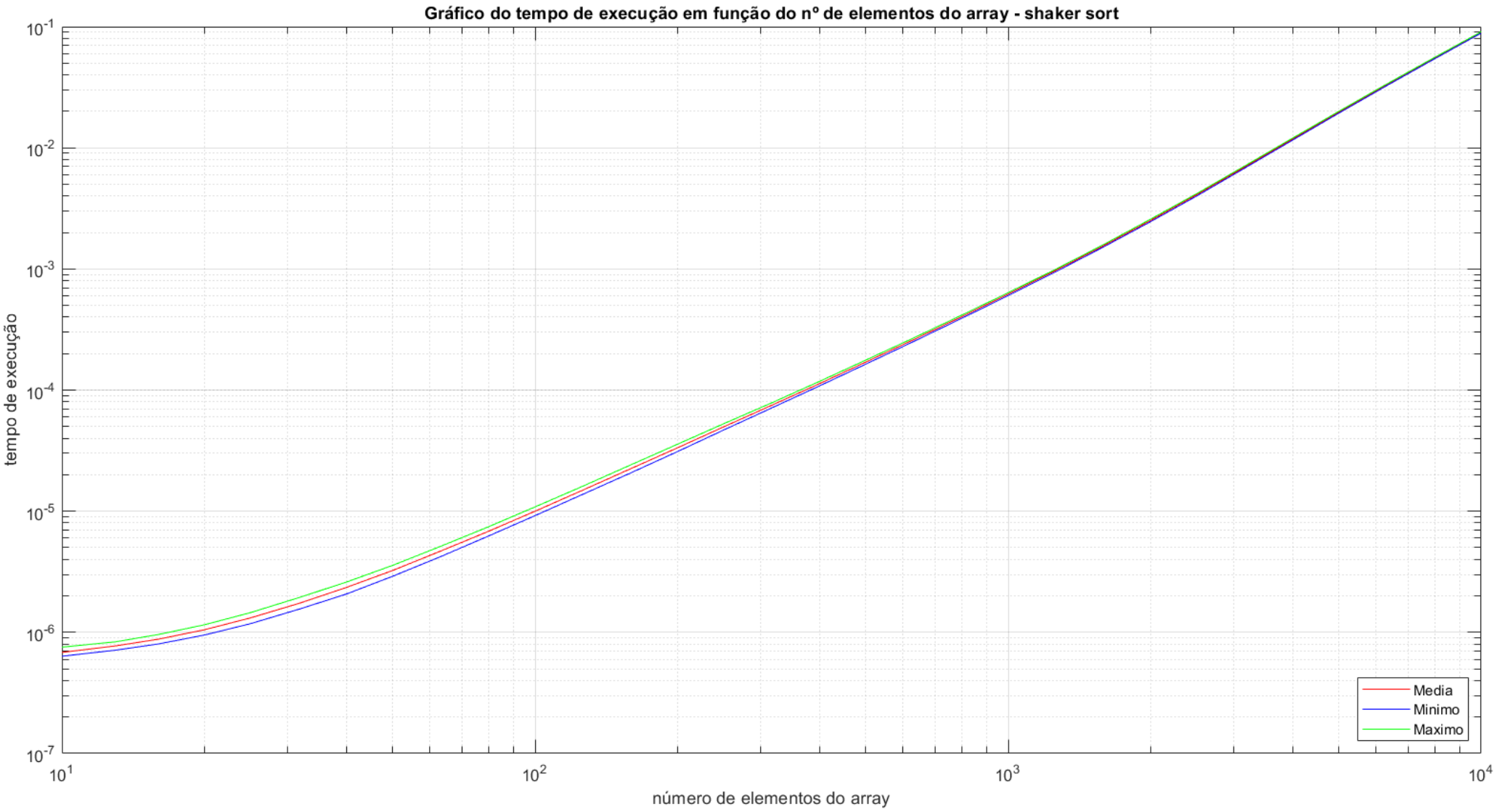
Por análise do gráfico podemos concluir que o tempo de execução varia de forma exponencial isto torna este algoritmo útil para casos em que o array é de pequena dimensão (até cerca de arrays com comprimento até 90) em que o tempo de execução não ultrapassa os 10^{-5} segundos.

Quanto ao desvio padrão, como é possível ver no gráfico seguinte, aumenta em relação ao tempo o que implica uma maior dispersão dos dados com o aumento do número de elementos o array, estes dados serão úteis mais tarde numa comparação entre os diferentes algoritmos.



3. Shaker Sort

n	min time	max time	avg time	std dev
10	6.350e-07	7.540e-07	6.829e-07	2.630e-08
13	7.130e-07	8.360e-07	7.731e-07	2.955e-08
16	8.020e-07	9.600e-07	8.784e-07	3.570e-08
20	9.500e-07	1.154e-06	1.050e-06	4.837e-08
25	1.177e-06	1.453e-06	1.314e-06	6.512e-08
32	1.570e-06	1.958e-06	1.754e-06	9.011e-08
40	2.076e-06	2.607e-06	2.354e-06	1.235e-07
50	2.903e-06	3.572e-06	3.240e-06	1.594e-07
63	4.200e-06	5.109e-06	4.667e-06	2.264e-07
79	6.143e-06	7.329e-06	6.724e-06	2.933e-07
100	9.191e-06	1.085e-05	9.996e-06	3.945e-07
126	1.376e-05	1.605e-05	1.489e-05	5.389e-07
158	2.041e-05	2.376e-05	2.211e-05	8.081e-07
200	3.105e-05	3.573e-05	3.335e-05	1.126e-06
251	4.701e-05	5.294e-05	4.991e-05	1.446e-06
316	7.102e-05	7.830e-05	7.473e-05	1.758e-06
398	1.076e-04	1.173e-04	1.123e-04	2.279e-06
501	1.642e-04	1.774e-04	1.708e-04	3.138e-06
631	2.516e-04	2.692e-04	2.605e-04	4.222e-06
794	3.878e-04	4.121e-04	3.999e-04	5.778e-06
1000	6.025e-04	6.365e-04	6.196e-04	8.060e-06
1259	9.455e-04	9.921e-04	9.694e-04	1.163e-05
1585	1.507e-03	1.577e-03	1.542e-03	1.662e-05
1995	2.442e-03	2.563e-03	2.500e-03	2.814e-05
2512	4.031e-03	4.215e-03	4.129e-03	4.368e-05
3162	6.795e-03	7.078e-03	6.939e-03	6.861e-05
3981	1.150e-02	1.195e-02	1.173e-02	1.065e-04
5012	1.943e-02	2.011e-02	1.977e-02	1.658e-04
6310	3.255e-02	3.358e-02	3.304e-02	2.393e-04
7943	5.388e-02	5.538e-02	5.465e-02	3.505e-04
10000	8.859e-02	9.081e-02	8.967e-02	5.252e-04



Equação da reta correspondente ao valor médio: $y = 1.468e^{-10}x^2 + 3.062e^{-08}x - 9.554e^{-06}$

O algoritmo *shaker sort* pode ser explicado tendo em base o *bubble sort*, uma vez que o seu modo de funcionamento é semelhante, possuindo a única diferença de que o *shaker sort* ordena em duas direções, enquanto que o *bubble sort* apenas ordena numa única direção. Observemos o funcionamento do *shaker sort* através do seguinte exemplo:

Dado um *array*, iremos comparar o seu primeiro elemento com o segundo elemento. Se o primeiro elemento for maior que o segundo elemento, eles irão trocar de posição e continuamos com a comparação. Note que a base da ordenação é a comparação entre elementos vizinhos.

4	8	2	9	1	7	3	0	5	6

Uma vez que o elemento 4 não é maior que o elemento 8, não será do nosso interesse continuar a comparar o elemento 4, pelo que iremos agora comparar o elemento 8 com o seu vizinho: o elemento 2. Como 8 é maior que 2, ambos os elementos irão trocar de posição.

4	8	2	9	1	7	3	0	5	6
4	2	8	9	1	7	3	0	5	6

Iremos continuar a usar o elemento 8 para comparação até que o elemento encontre um outro cujo valor seja maior, não havendo troca de posições. Nessa altura, escolhemos o elemento que possui um valor superior ao valor 8 e iremos comparar com o vizinho até encontrar novamente um elemento que possua um valor maior ou até chegarmos ao final do *array*. No exemplo apresentado, como o próximo passo irá bloquear o número 8 e passaremos a usar o número 9 para comparação, sendo este número o maior do *array*, não encontraremos um elemento com o valor maior, pelo que a comparação irá terminar quando chegarmos ao final do *array*.

4	2	8	1	7	3	0	5	6	9

A diferença entre o modo de ordenação do *bubble sort* e do *shaker sort* é possível de ser verificada agora: enquanto o *bubble sort*, ao chegar nesta etapa, começaria uma nova ronda de ordenação a partir do índice inicial, o *shaker sort*, quando chega ao final do *array*, escolhe o vizinho do último elemento e compara com os restantes elementos na direção oposta (em linguagem comum, de trás para a frente) com a condição contrária: se o valor for menor que o valor analisado, as posições trocam e altera-se e continuamos a comparar o mesmo elemento com os restantes até chegarmos ao início ou encontrarmos um valor

que o bloqueie, ou seja, que seja menor; se isso acontecer, alteramos o elemento de comparação.

4	2	8	1	7	3	0	5	6	9
4	2	8	1	7	3	0	5	6	9
4	2	8	1	7	3	0	5	6	9
4	2	8	1	7	0	3	5	6	9
4	2	8	1	0	7	3	5	6	8

. . . . .

0	4	2	8	1	7	3	5	6	9

Alcançado o início do *array* novamente, o algoritmo *shaker sort* escolhe o vizinho do primeiro elemento e irá realizar as comparações com os restantes elementos até ao penúltimo elemento do *array*, uma vez que a comparação com o último é desnecessária porque já sabemos que mais nenhum elemento presente no *array* tem um valor superior ao valor dele.

Note que nesta fase já temos o maior e o menor valores presentes no vetor nas suas posições corretas, pelo que não será necessário compará-los novamente com os restantes elementos. Podemos dizer então que há medida que vamos organizando o *array*, o segmento de elementos que iremos analisar diminuirá de tamanho.

0	4	2	8	1	7	3	5	6	9
0	2	4	8	1	7	3	5	6	9

. . . . .

0	2	4	1	7	3	5	6	8	9

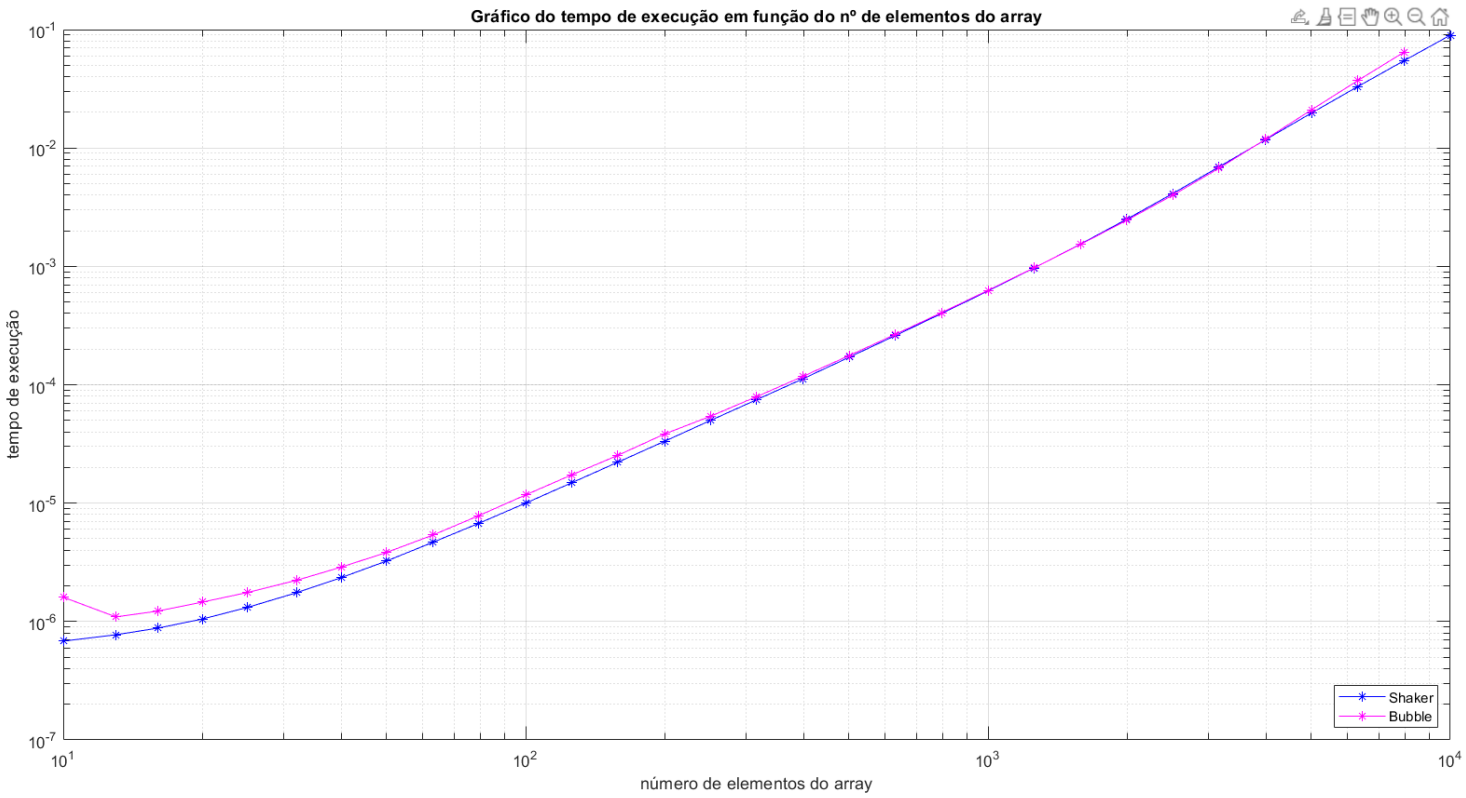
0	2	4	1	7	3	5	6	8	9
0	2	4	1	7	3	5	6	8	9
0	2	4	1	7	3	5	6	8	9
0	2	4	1	3	7	5	6	8	9

. . . . .

0	1	2	4	3	7	5	6	8	9

0	1	2	4	3	7	5	6	8	9
0	1	2	4	3	7	5	6	8	9
0	1	2	3	4	7	5	6	8	9
0	1	2	3	4	7	5	6	8	9
0	1	2	3	4	5	7	6	8	9
0	1	2	3	4	5	6	7	8	9

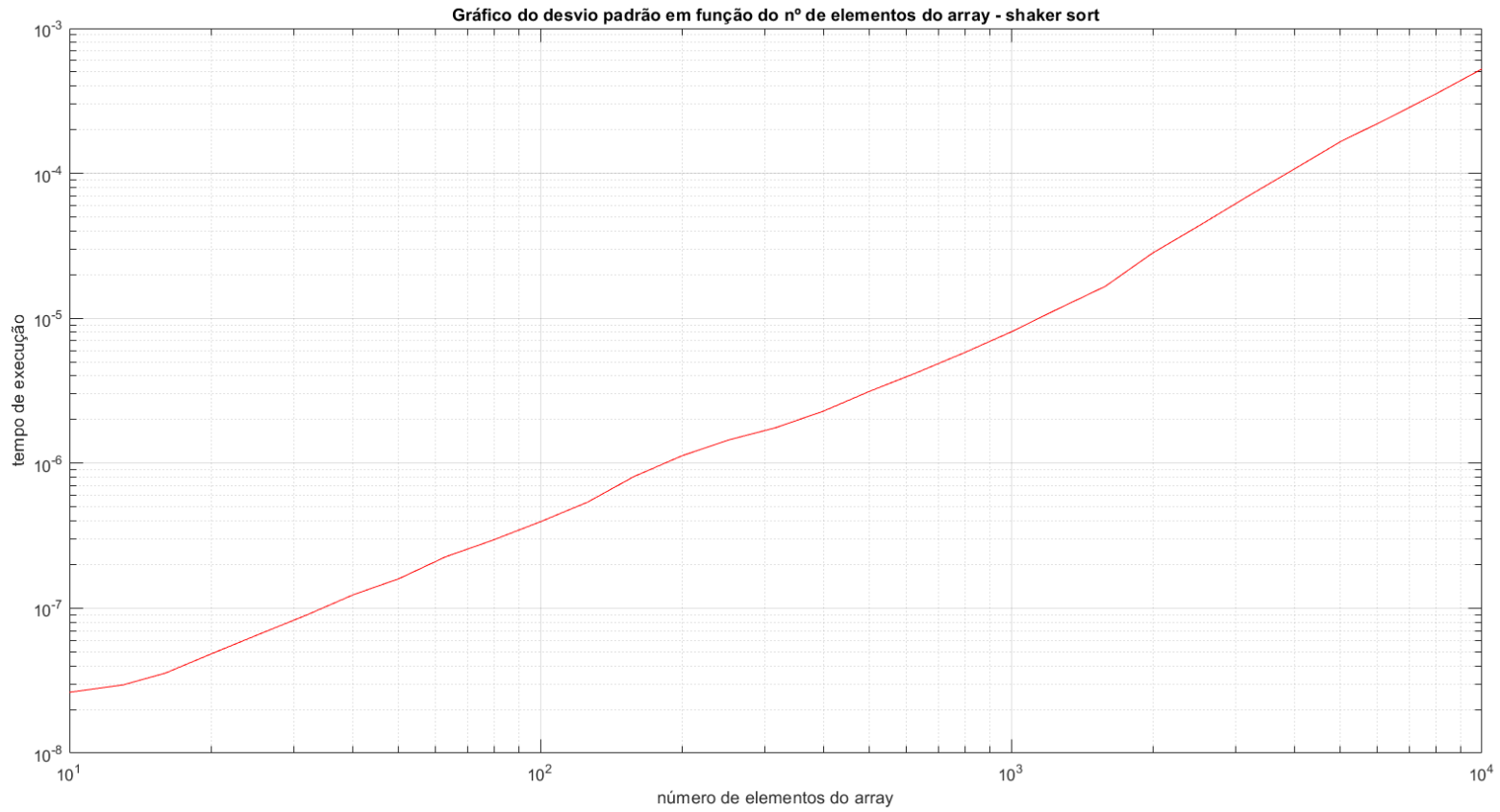
Podemos dizer que o *shaker sort* apresenta uma eficiência maior que a do *bubble sort*, uma vez que, para além de comparar elementos vizinhos na direção esquerda-direita, também os compara na direção direita-esquerda. Essa melhor eficiência é possível verificar para *arrays* com um número de elementos compreendidos nos seguintes intervalos de valores [10, 398] e [5012, 10000], sendo que na zona onde temos o intervalo [398, 5012] é possível observar uma aproximação entre as duas retas, chegando a existir interseção entre ambas. Note também que o começo da reta correspondente ao *shaker sort* não sofre um declínio como acontece com o *bubble sort*, a função é crescente no intervalo de valores [10, 10000].



Quanto à complexidade do algoritmo, apresenta complexidade Ω(n) no melhor caso (*Big Omega Notation*), complexidade $\theta(n^2)$no caso médio (*Big Theta Notation*) e complexidade $O(n^2)$ no pior caso (*Big O Notation*). Repare que as complexidades são semelhantes às complexidades do *bubble sort*.

Pela análise do gráfico do tempo de execução em função do número de elementos do *array* exclusivo do algoritmo *shaker sort*, podemos observar que quando o n se encontra compreendido entre 10 e 316 aproximadamente, o algoritmo apresenta uma instabilidade considerável em relação aos tempos de execução - a diferença entre o max time e o min time é acentuada -, ou seja, o algoritmo apresentará maior eficiência para *arrays* com comprimento maior que 361, mais precisamente, *arrays* com um número de elementos superior a 1585.

Em relação ao desvio padrão, é possível verificar que o mesmo aumenta quando o número de elementos no *array* aumenta, ou seja, a dispersão de valores em relação ao valor médio no caso do *shaker sort* aumenta ao longo do aumento do número de elementos no vetor. Posteriormente, analisaremos os desvios padrões de todos os algoritmos de forma a encontrar o mais estável.



4. Insertion Sort

n	min time	max time	avg time	std dev
10	5.980e-07	6.430e-07	6.190e-07	1.052e-08
13	6.360e-07	6.830e-07	6.578e-07	1.106e-08
16	6.740e-07	7.320e-07	7.000e-07	1.398e-08
20	7.260e-07	7.890e-07	7.534e-07	1.430e-08
25	7.970e-07	8.830e-07	8.341e-07	1.979e-08
32	9.130e-07	1.014e-06	9.559e-07	2.420e-08
40	1.051e-06	1.210e-06	1.115e-06	3.426e-08
50	1.261e-06	1.441e-06	1.340e-06	4.193e-08
63	1.566e-06	1.811e-06	1.679e-06	5.459e-08
79	2.010e-06	2.325e-06	2.161e-06	7.251e-08
100	2.741e-06	3.185e-06	2.939e-06	1.043e-07
126	3.771e-06	4.340e-06	4.051e-06	1.366e-07
158	5.323e-06	6.109e-06	5.701e-06	1.910e-07
200	7.743e-06	8.830e-06	8.250e-06	2.493e-07
251	1.144e-05	1.287e-05	1.212e-05	3.344e-07
316	1.716e-05	1.913e-05	1.813e-05	4.764e-07
398	2.610e-05	2.887e-05	2.742e-05	6.648e-07
501	4.003e-05	4.405e-05	4.205e-05	9.287e-07
631	6.210e-05	6.764e-05	6.489e-05	1.325e-06
794	9.669e-05	1.045e-04	1.006e-04	1.803e-06
1000	1.513e-04	1.624e-04	1.568e-04	2.558e-06
1259	2.378e-04	2.533e-04	2.453e-04	3.604e-06
1585	3.754e-04	3.952e-04	3.849e-04	4.808e-06
1995	5.892e-04	6.197e-04	6.045e-04	7.320e-06
2512	9.304e-04	9.720e-04	9.524e-04	1.006e-05
3162	1.473e-03	1.529e-03	1.501e-03	1.381e-05
3981	2.327e-03	2.420e-03	2.373e-03	2.140e-05
5012	3.690e-03	3.977e-03	3.763e-03	4.575e-05
6310	5.869e-03	7.446e-03	6.167e-03	3.430e-04
7943	9.288e-03	1.128e-02	9.679e-03	4.606e-04
10000	1.467e-02	1.501e-02	1.484e-02	8.180e-05
12589	2.324e-02	2.374e-02	2.349e-02	1.162e-04
15849	3.684e-02	3.753e-02	3.718e-02	1.625e-04
19953	5.844e-02	5.967e-02	5.897e-02	2.573e-04
25119	9.265e-02	9.417e-02	9.334e-02	3.436e-04

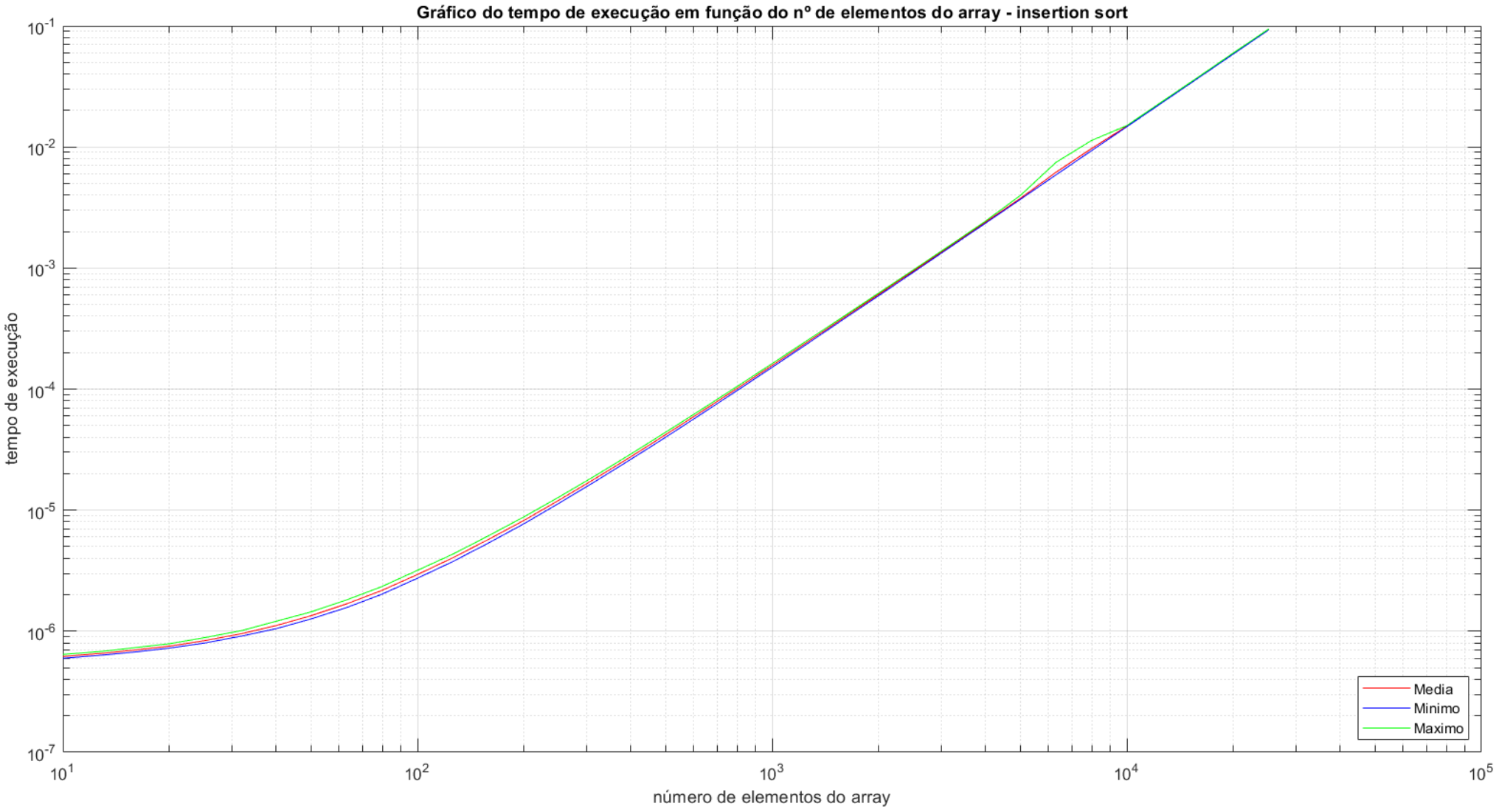


Gráfico do tempo de execução em função do nº de elementos do array - insertion sort
tempo de execução
número de elementos do array
Media
Minimo
Maximo

O Insertion Sort funciona de uma forma bastante simples pois é bastante intuitivo. Primeiro o array é dividido em duas partes, uma ordenada e a outra não ordenada, sendo os valores da parte não ordenada inseridos na parte ordenada no local correto.

Os passos consistem em:

1 - Iterar de array[1] até array [n];

2 - Comparar o valor atual com o anterior;

3 - Se o valor atual com o anterior comparar com os elementos anteriores movendo os valores maiores uma posição para abrir espaço para o elemento em troca

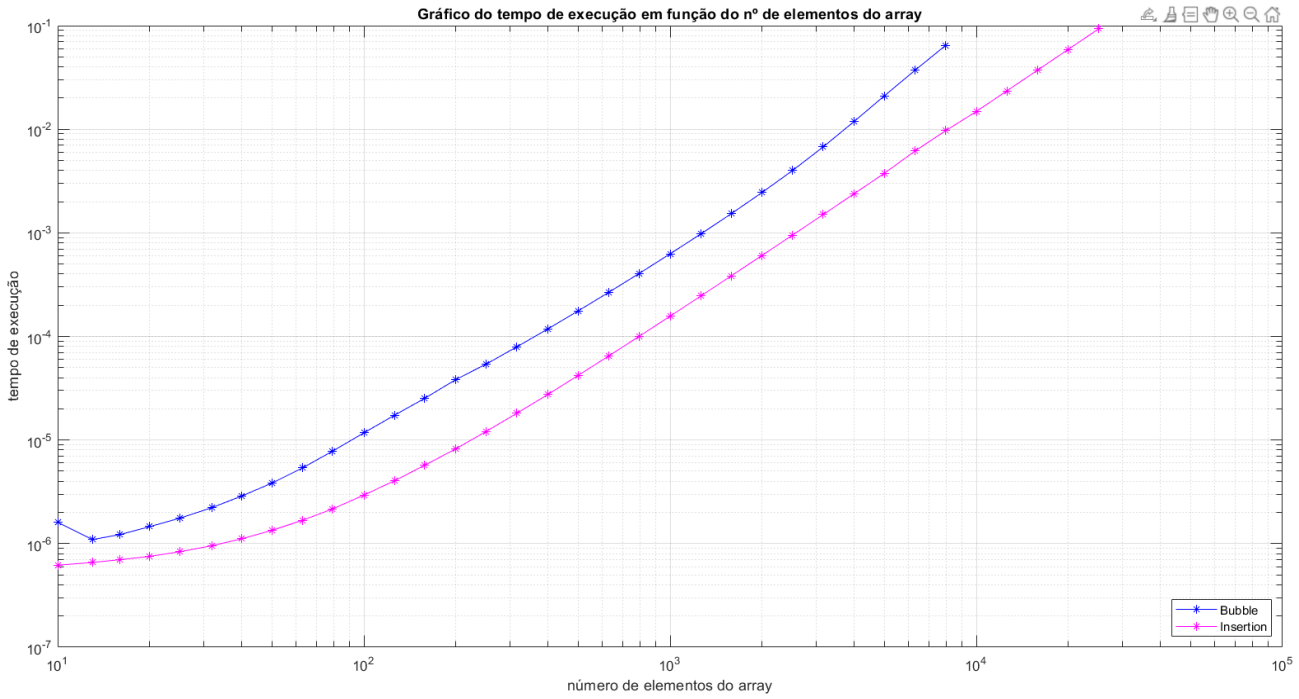
Em seguida ordenamos o seguinte array com uso deste algoritmo:

8	7	3	1	4

O azul simboliza o elemento atual.

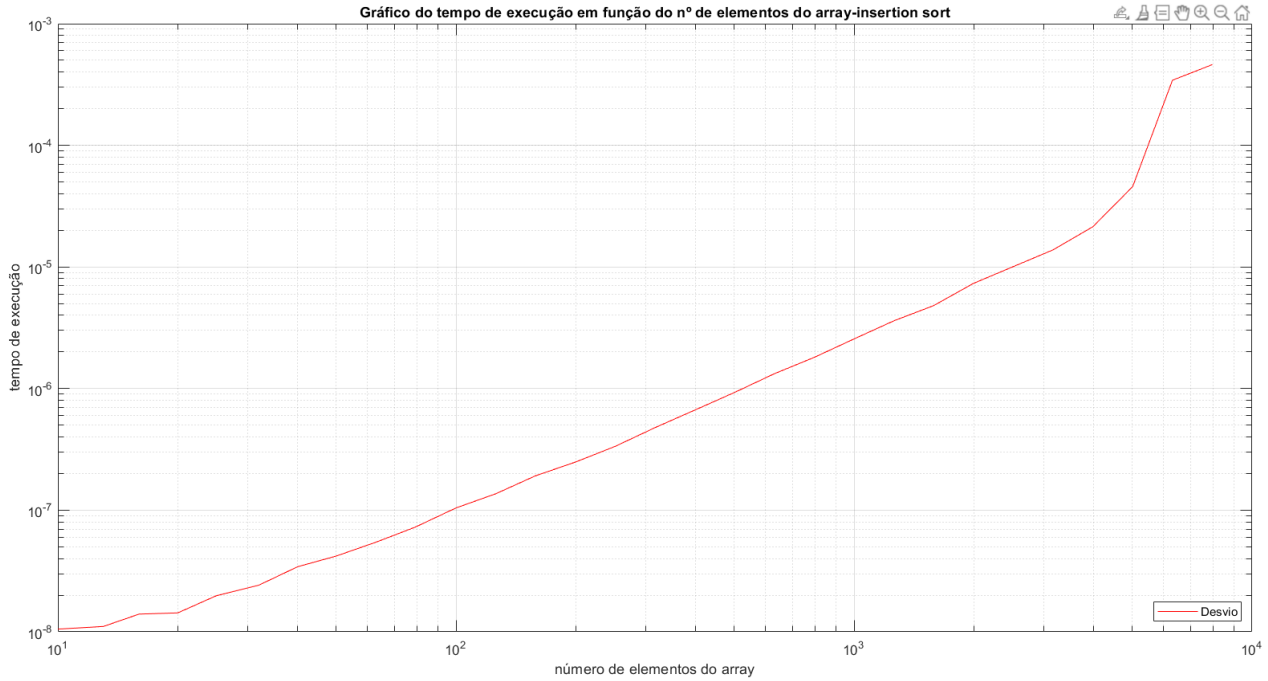
8	5	3	1	7
5	8	3	1	7
3	5	8	1	7
1	3	5	8	7
1	3	5	8	7

Relativamente à sua complexidade tem como melhor caso uma complexidade (em Big O Notation) O(n), este caso acontece quando o array já está na ordem correta, para o pior e caso médio apresenta uma complexidade de $O(n^2)$. Apesar de ter a mesma complexidade que algoritmos como o bubble sort as constantes multiplicativas por detrás da notação são pequenas ao ponto de se notar uma diferença relevante entre os dois. (como mostrado no gráfico seguinte)



Por análise do gráfico podemos concluir que o tempo de execução varia de forma exponencial, no entanto para arrays de tamanhos reduzidos (cerca de 130 elementos) é bastante útil e eficiente.

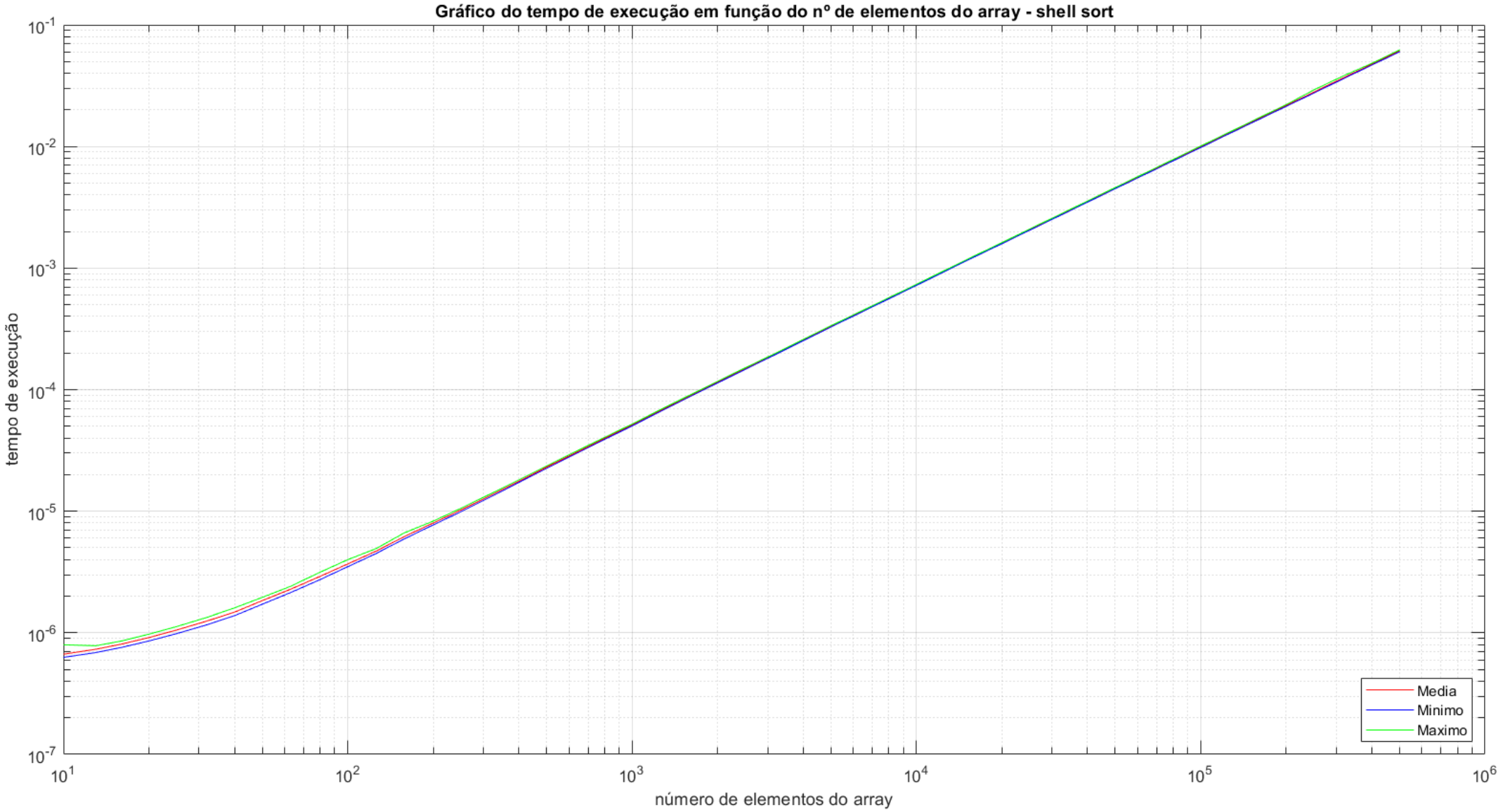
Quanto ao desvio padrão, como é possível ver no gráfico seguinte, aumenta em relação ao tempo o de forma mais ao menos linear numa fase inicial e crescendo de forma abrupta para valores muito elevados.



5. Shell Sort

n	min time	max time	avg time	std dev
10	6.280e-07	7.970e-07	6.666e-07	2.603e-08
13	6.880e-07	7.830e-07	7.326e-07	2.155e-08
16	7.580e-07	8.560e-07	8.072e-07	2.400e-08
20	8.570e-07	9.750e-07	9.141e-07	2.804e-08
25	9.840e-07	1.125e-06	1.054e-06	3.400e-08
32	1.165e-06	1.340e-06	1.250e-06	4.049e-08
40	1.384e-06	1.605e-06	1.480e-06	4.855e-08
50	1.717e-06	1.954e-06	1.838e-06	5.755e-08
63	2.144e-06	2.415e-06	2.284e-06	6.391e-08
79	2.706e-06	3.112e-06	2.878e-06	8.866e-08
100	3.497e-06	3.997e-06	3.684e-06	1.026e-07
126	4.496e-06	4.935e-06	4.704e-06	1.005e-07
158	5.921e-06	6.631e-06	6.186e-06	1.438e-07
200	7.723e-06	8.306e-06	8.006e-06	1.352e-07
251	9.951e-06	1.062e-05	1.028e-05	1.528e-07
316	1.304e-05	1.381e-05	1.340e-05	1.821e-07
398	1.715e-05	1.802e-05	1.756e-05	2.113e-07
501	2.256e-05	2.361e-05	2.307e-05	2.440e-07
631	2.957e-05	3.082e-05	3.015e-05	2.888e-07
794	3.857e-05	4.016e-05	3.927e-05	3.591e-07
1000	5.032e-05	5.209e-05	5.116e-05	4.036e-07
1259	6.618e-05	6.854e-05	6.721e-05	5.256e-07
1585	8.679e-05	8.954e-05	8.802e-05	6.278e-07
1995	1.132e-04	1.168e-04	1.148e-04	7.887e-07
2512	1.477e-04	1.521e-04	1.495e-04	9.785e-07
3162	1.921e-04	1.969e-04	1.941e-04	1.101e-06
3981	2.514e-04	2.579e-04	2.541e-04	1.451e-06
5012	3.283e-04	3.361e-04	3.317e-04	1.724e-06
6310	4.261e-04	4.356e-04	4.303e-04	2.203e-06
7943	5.540e-04	5.660e-04	5.594e-04	2.713e-06
10000	7.198e-04	7.342e-04	7.263e-04	3.353e-06
12589	9.393e-04	9.583e-04	9.478e-04	4.351e-06
15849	1.223e-03	1.247e-03	1.234e-03	5.921e-06
19953	1.583e-03	1.616e-03	1.598e-03	7.668e-06
25119	2.057e-03	2.105e-03	2.077e-03	1.076e-05
31623	2.669e-03	2.728e-03	2.694e-03	1.332e-05
39811	3.466e-03	3.545e-03	3.500e-03	1.878e-05
50119	4.505e-03	4.608e-03	4.549e-03	2.459e-05

63096	5.835e-03	5.979e-03	5.896e-03	3.377e-05
79433	7.565e-03	7.763e-03	7.643e-03	4.622e-05
100000	9.808e-03	1.009e-02	9.924e-03	6.413e-05
125893	1.274e-02	1.309e-02	1.289e-02	8.620e-05
158489	1.650e-02	1.701e-02	1.671e-02	1.171e-04
199526	2.136e-02	2.205e-02	2.165e-02	1.619e-04
251189	2.767e-02	2.938e-02	2.818e-02	3.460e-04
316228	3.594e-02	3.794e-02	3.658e-02	4.443e-04
398107	4.655e-02	4.811e-02	4.724e-02	3.878e-04
501187	6.022e-02	6.235e-02	6.117e-02	4.984e-04



Equação da reta correspondente ao valor médio: $y = 4.874e^{-14}x^2 + 9.885e^{-08}x - 0.0001342$

O algoritmo *shell sort* pode ser explicado tendo em base o *insertion sort*, porém com a diferença de que este algoritmo irá permitir que comparações entre elementos não vizinhos sejam realizadas. Em outras palavras, não iremos considerar que apenas um vetor terá que ser ordenado, mas sim vários vetores menores que serão segmentos do vetor principal e que irão decrescer em tamanho até que este tenha um comprimento igual a 1.

Começamos por definir o valor da variável *h*, variável que irá ditar o tamanho da primeira partição do vetor original em segmentos. Observe o exemplo:

h = 4

4	8	2	9	1	7	3	0	5	6

Como o valor de *h* é o 4, iremos comparar então o primeiro elemento com o elemento que se encontra quatro posições após o primeiro elemento.

4	8	2	9	1	7	3	0	5	6
1	8	2	9	4	7	3	0	5	6

Feita a primeira comparação, deslizamos uma casa para a direita e comparamos o segundo elemento com o elemento que se encontra quatro posições após o segundo elemento. De seguida comparamos o terceiro elemento com o elemento que se encontra quatro posições após o terceiro elemento. Realizamos essas comparações de 4 em 4 até chegarmos ao fim do *array*.

1	8	2	9	4	7	3	0	5	6
1	6	2	9	4	7	3	0	5	8
1	6	2	9	4	7	3	0	5	8
1	6	2	0	4	7	3	9	5	8

Terminado o processo de ordenação para um *h* igual a 4, decrementamos o *h* e repetimos o processo para *h* com o valor de 3 agora. Ao terminarmos de ordenar com *h* igual a 3, decrementamos novamente a variável e ordenamos. Realizamos sucessivas decrementações até que o valor do *h* seja igual a 1. Note que quando *h* igual 1, iremos ordenar seguindo a lógica do *insertion sort*.

h = 3

1	6	2	0	4	7	3	9	5	8

0	6	2	1	4	7	3	9	5	8

. . . . .

0	4	2	1	6	5	3	9	7	8

h = 2

0	4	2	1	6	5	3	9	7	8
0	4	2	1	3	5	6	9	7	8

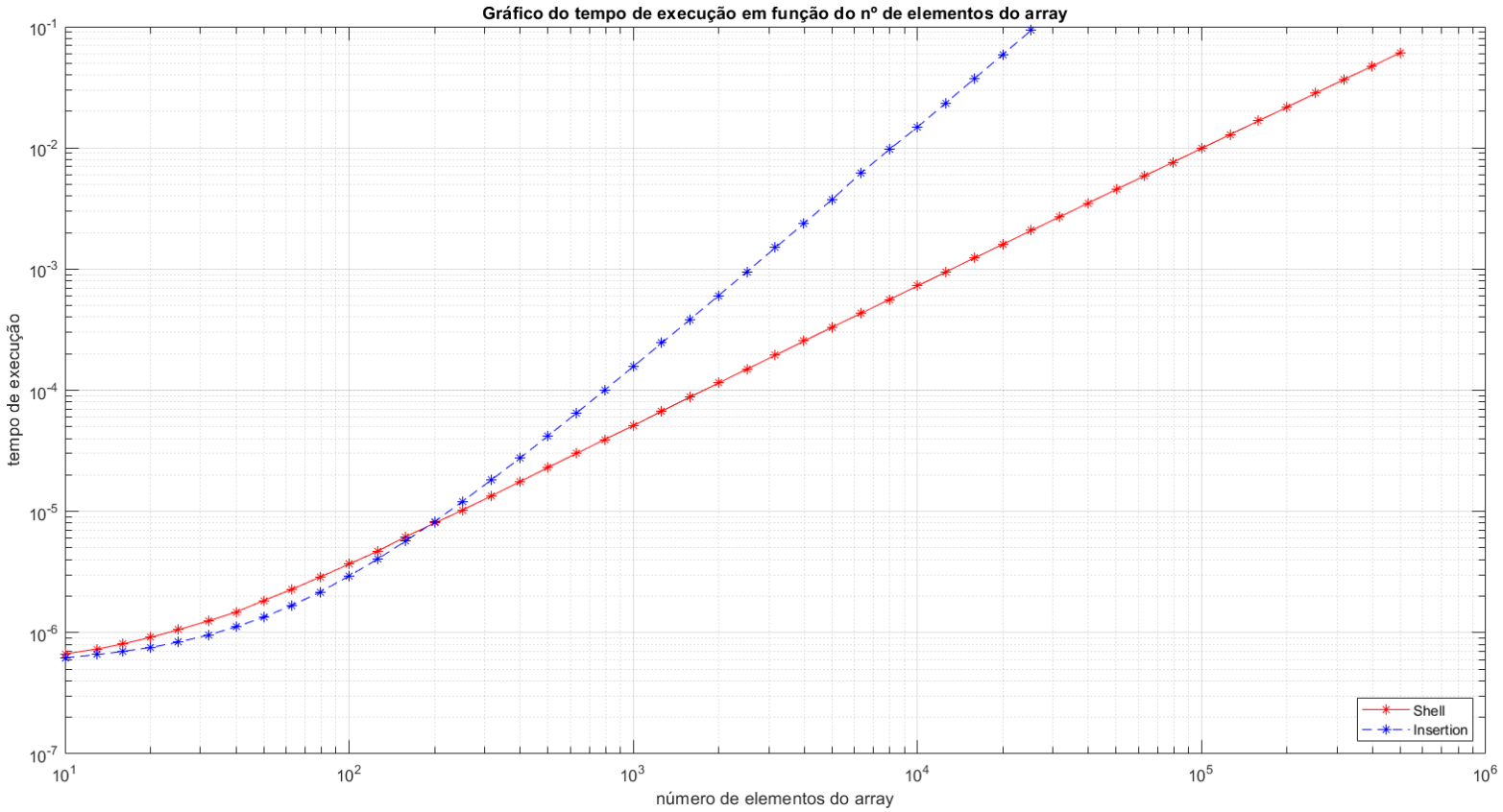
. . . . .

0	1	2	4	3	5	6	8	7	9

h=1 (*Insertion Sort*)

0	1	2	4	3	5	6	8	7	9
0	1	2	3	4	5	6	8	7	9

Podemos dizer que o *shell sort* apresenta uma eficiência maior que a do *insertion sort*, uma vez que, para além de comparar elementos vizinhos, também compara elementos com índices distantes. Essa melhor eficiência é possível verificar para *arrays* com um maior comprimento.

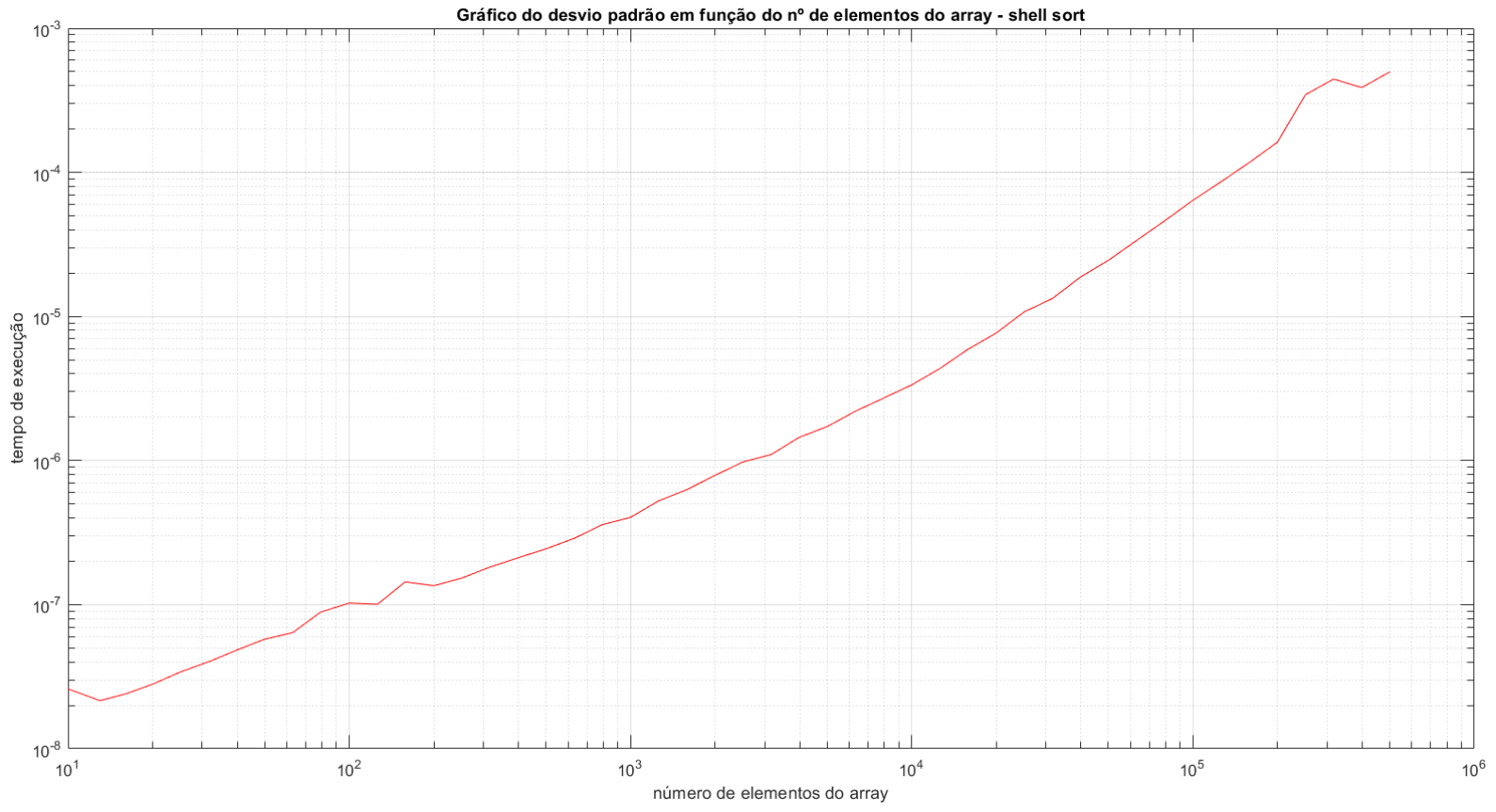


Observe que, embora o *insertion sort* apresente uma melhor eficiência numa fase inicial do gráfico, quando n tende para infinito, o melhor algoritmo de ordenação será o *shell sort*. Quando o nosso vetor possui cerca de 200 elementos, o tempo de execução de ambos os algoritmos são semelhantes (interseção de retas no gráfico).

Quanto à complexidade do algoritmo, os valores das mesmas ainda se encontram em processo de estudo. Atualmente, podemos dizer que apresenta complexidade $\Omega(nlog\ (n))$ no melhor caso (*Big Omega Notation*) para a maioria dos *arrays*, no caso médio (*Big Theta Notation*) o valor irá depender do *array* e $O(n^2)$ no pior caso (*Big O Notation*) para a maioria dos *arrays*.

Pela análise do gráfico do tempo de execução em função do número de elementos do *array* exclusivo do algoritmo *shell sort*, podemos observar que quando o n se encontra compreendido entre 10 e 251 aproximadamente, o algoritmo apresenta uma instabilidade considerável em relação aos tempos de execução - a diferença entre o max time e o min time é acentuada -, ou seja, o algoritmo apresentará maior eficiência para *arrays* com comprimento maior que 251.

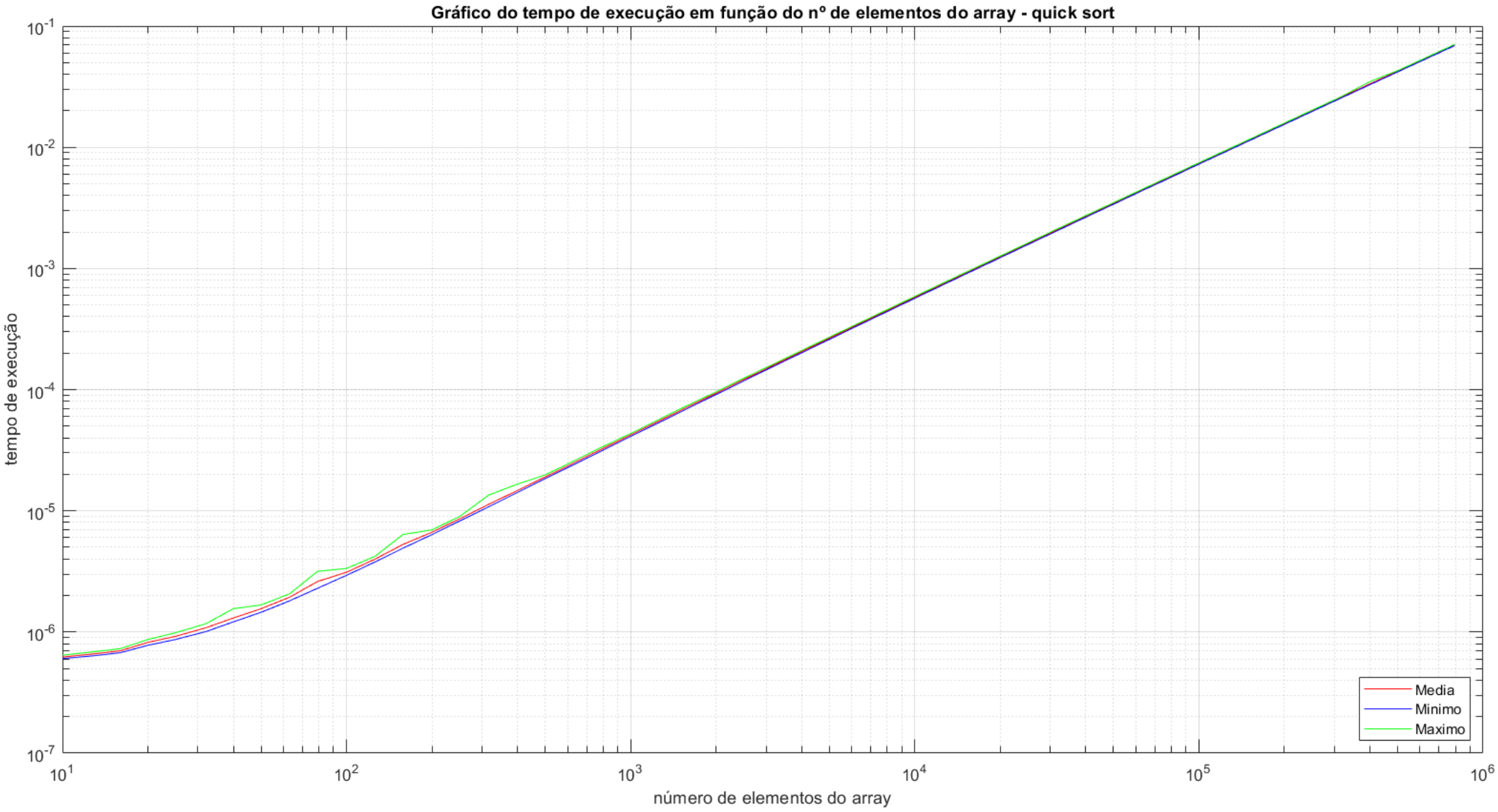
Em relação ao desvio padrão, é possível verificar que o mesmo aumenta quando o número de elementos no *array* aumenta, ou seja, a dispersão de valores em relação ao valor médio no caso do *shell sort* aumenta ao longo do aumento do número de elementos no vetor. Posteriormente, analisaremos os desvios padrões de todos os algoritmos de forma a encontrar o mais estável.



6. Quick Sort

n	min time	max time	avg time	std dev
10	6.030e-07	6.440e-07	6.217e-07	9.672e-09
13	6.380e-07	6.880e-07	6.616e-07	1.155e-08
16	6.750e-07	7.290e-07	6.998e-07	1.323e-08
20	7.770e-07	8.660e-07	8.210e-07	2.122e-08
25	8.670e-07	9.850e-07	9.204e-07	2.834e-08
32	1.008e-06	1.171e-06	1.084e-06	3.901e-08
40	1.213e-06	1.558e-06	1.306e-06	6.188e-08
50	1.453e-06	1.670e-06	1.556e-06	4.992e-08
63	1.806e-06	2.063e-06	1.934e-06	6.032e-08
79	2.294e-06	3.164e-06	2.608e-06	2.498e-07
100	2.934e-06	3.346e-06	3.113e-06	8.828e-08
126	3.785e-06	4.213e-06	3.994e-06	9.947e-08
158	4.918e-06	6.377e-06	5.282e-06	3.525e-07
200	6.370e-06	6.965e-06	6.658e-06	1.386e-07
251	8.270e-06	9.021e-06	8.624e-06	1.724e-07
316	1.079e-05	1.347e-05	1.131e-05	4.151e-07
398	1.411e-05	1.655e-05	1.464e-05	3.064e-07
501	1.844e-05	1.979e-05	1.906e-05	2.957e-07
631	2.408e-05	2.581e-05	2.486e-05	3.762e-07
794	3.145e-05	3.366e-05	3.243e-05	4.657e-07
1000	4.109e-05	4.344e-05	4.228e-05	5.528e-07
1259	5.352e-05	5.681e-05	5.511e-05	7.091e-07
1585	6.983e-05	7.402e-05	7.177e-05	8.840e-07
1995	9.095e-05	9.564e-05	9.332e-05	1.075e-06
2512	1.186e-04	1.248e-04	1.215e-04	1.346e-06
3162	1.545e-04	1.613e-04	1.578e-04	1.591e-06
3981	2.003e-04	2.091e-04	2.048e-04	2.042e-06
5012	2.601e-04	2.711e-04	2.657e-04	2.570e-06
6310	3.379e-04	3.507e-04	3.446e-04	3.086e-06
7943	4.374e-04	4.536e-04	4.460e-04	3.906e-06
10000	5.667e-04	5.872e-04	5.772e-04	4.946e-06
12589	7.339e-04	7.587e-04	7.470e-04	5.830e-06
15849	9.480e-04	9.790e-04	9.645e-04	7.673e-06
19953	1.225e-03	1.264e-03	1.246e-03	9.671e-06
25119	1.582e-03	1.632e-03	1.609e-03	1.164e-05
31623	2.040e-03	2.110e-03	2.076e-03	1.669e-05
39811	2.634e-03	2.715e-03	2.676e-03	1.967e-05
50119	3.392e-03	3.497e-03	3.448e-03	2.457e-05

63096	4.380e-03	4.497e-03	4.443e-03	2.864e-05
79433	5.624e-03	5.790e-03	5.718e-03	3.812e-05
100000	7.258e-03	7.453e-03	7.357e-03	4.805e-05
125893	9.324e-03	9.566e-03	9.461e-03	5.993e-05
158489	1.199e-02	1.230e-02	1.216e-02	7.579e-05
199526	1.541e-02	1.581e-02	1.563e-02	9.556e-05
251189	1.982e-02	2.032e-02	2.009e-02	1.201e-04
316228	2.545e-02	2.609e-02	2.579e-02	1.530e-04
398107	3.270e-02	3.466e-02	3.321e-02	3.168e-04
501187	4.189e-02	4.291e-02	4.246e-02	2.405e-04
630957	5.374e-02	5.503e-02	5.444e-02	3.014e-04
794328	6.896e-02	7.054e-02	6.980e-02	3.719e-04



Equação da reta correspondente ao valor médio: $y = 1.661e^{-14}x^2 + 7.368e^{-08}x - 0.0001118$

O algoritmo *quick sort* é, atualmente, o algoritmo mais eficiente na comparação de elementos com o objetivo de ordená-los segundo um determinado critério. O modo de funcionamento dele baseia-se na escolha de um elemento do *array*, que chamaremos de pivô, cuja funcionalidade será organizar os restantes elementos seguindo um critério de comparação. Por exemplo, se quisermos ordenar um *array* por ordem crescente, ao escolhermos um pivô, os elementos que forem maiores que esse pivô irão ficar à sua direita, enquanto que os elementos que forem menores que esse pivô ficarão à sua esquerda. Repare que, quando todas as comparações forem feitas, esse pivô estará na sua posição correta no *array* e as comparações que se seguem dão-se em dois segmentos do *array*, pois não tem qualquer relevância comparar um elemento que tenha ficado à esquerda do pivô com um elemento que tenha ficado à direita, se o elemento da esquerda foi menor que o pivô, então, automaticamente, será menor que qualquer elemento à direita do pivô. Observe o exemplo:

Escolhendo um pivô aleatório: 7.

4	8	2	9	1	7	3	0	5	6
4	2	1	3	0	5	6	7	8	9

Escolhendo um pivô aleatório: 3

4	2	1	3	0	5	6	7	8	9
2	1	0	3	4	5	6	7	8	9

Escolhendo um pivô aleatório: 0

2	1	0	3	4	5	6	7	8	9
0	2	1	3	4	5	6	7	8	9

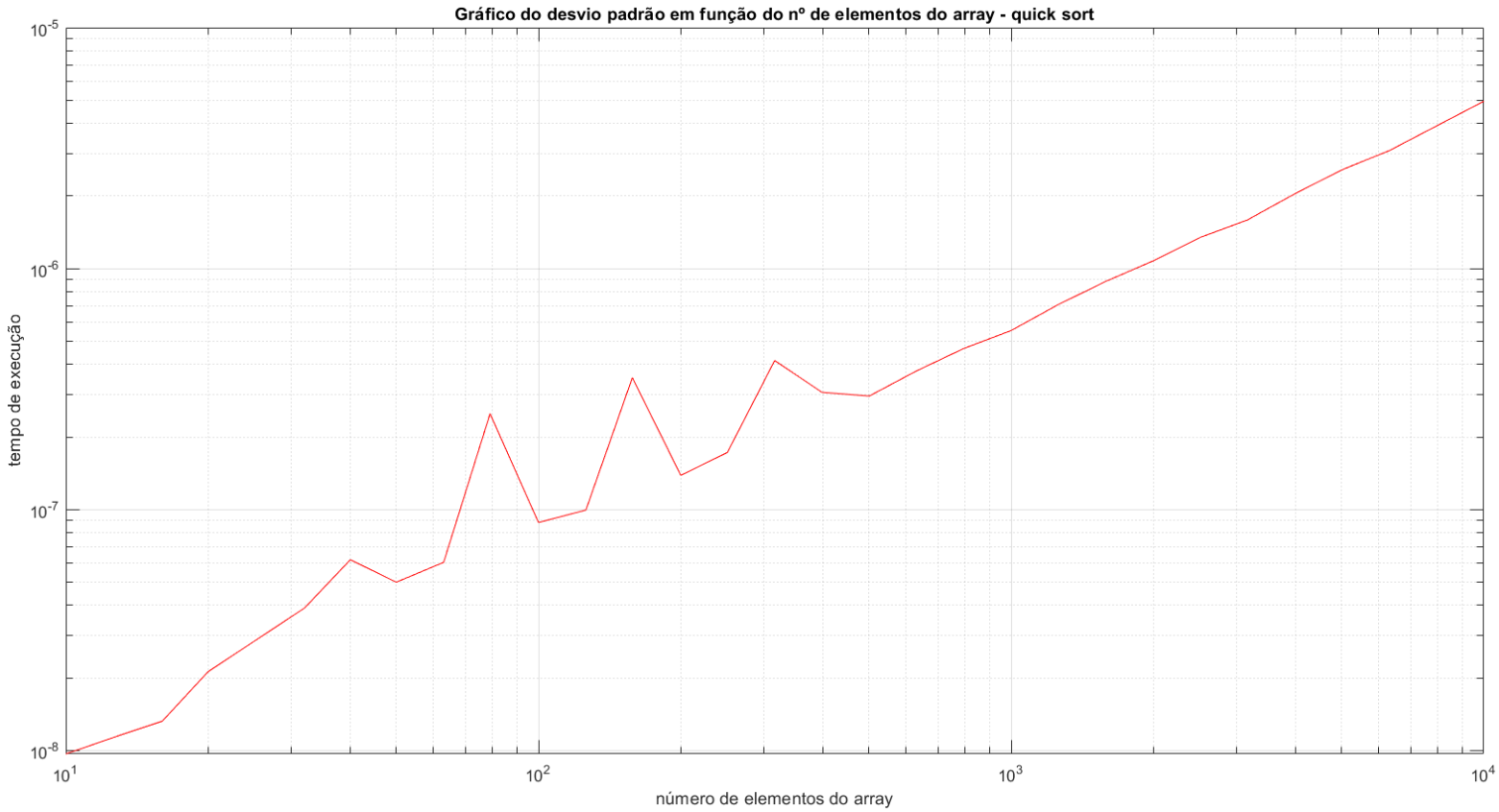
Escolhendo um pivô aleatório: 2

0	2	1	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

Quanto à complexidade do algoritmo, os valores das mesmas ainda se encontram em processo de estudo. Atualmente, podemos dizer que apresenta complexidade $\Omega(nlog\ (n))$ no melhor caso (*Big Omega Notation*), complexidade $\theta(nlog\ (n))$ no caso médio (*Big Theta Notation*) e complexidade $O(n^2)$ no pior caso (*Big O Notation*).

Pela análise do gráfico do tempo de execução em função do número de elementos do *array* exclusivo do algoritmo *quick sort*, podemos observar que quando o *n* se encontra compreendido entre 10 e 501 aproximadamente, o algoritmo apresenta uma instabilidade elevada em relação aos tempos de execução - a diferença entre o max time e o min time é acentuada. No entanto, para valores superiores a 501, ainda é possível verificar que o algoritmo não apresenta uma estabilidade segura, a diferença entre o min time e o max time ainda é significativa.

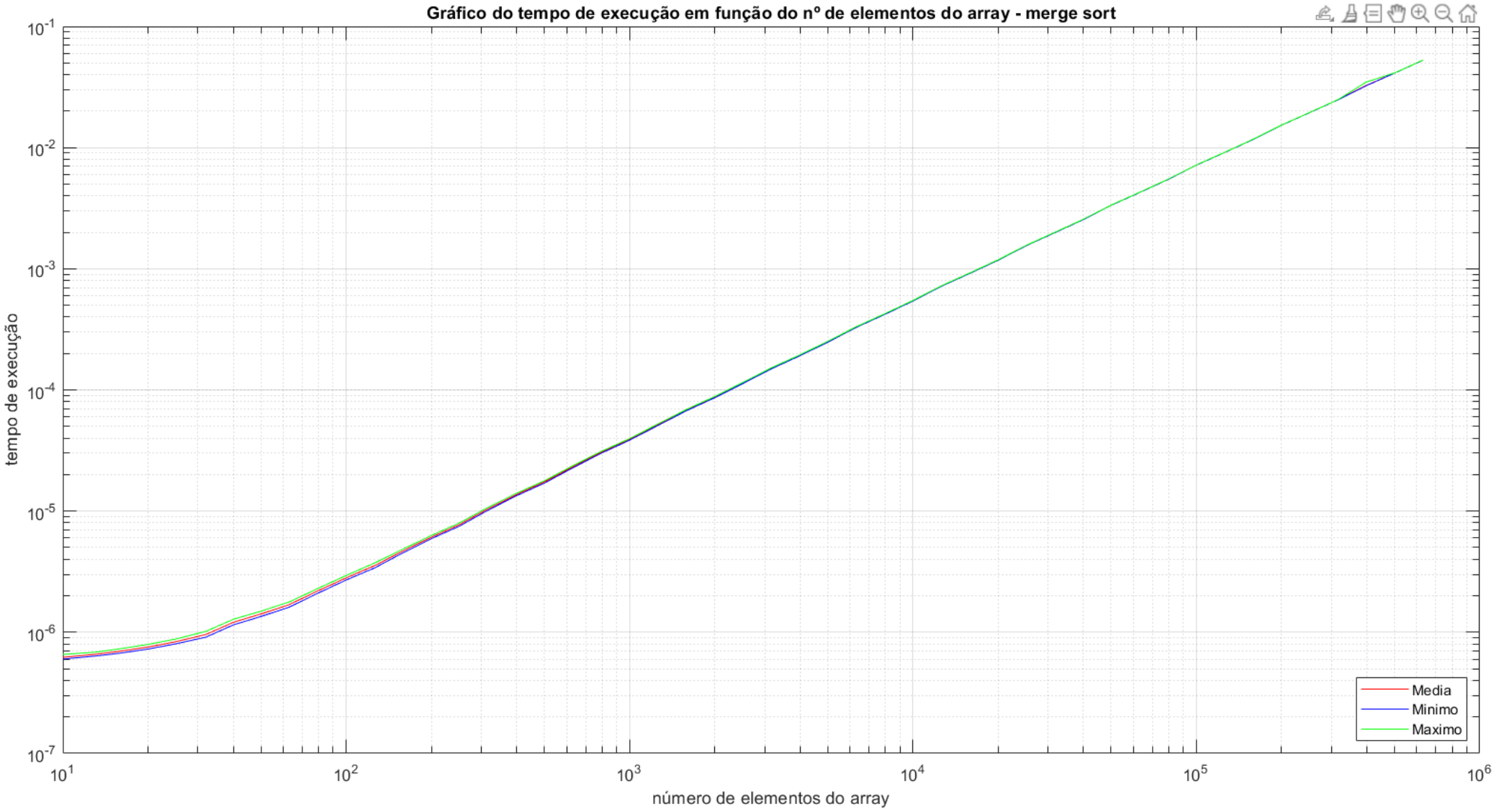
Em relação ao desvio padrão, é possível verificar que o mesmo aumenta quando o número de elementos no *array* aumenta, ou seja, a dispersão de valores em relação ao valor médio no caso do *quick sort* aumenta ao longo do aumento do número de elementos no vetor. É ainda notável a presença de picos e depressões acentuadas, consequência do algoritmo não ser um algoritmo estável. Posteriormente, analisaremos os desvios padrões de todos os algoritmos de forma a encontrar o mais estável.



7. Merge Sort

n	min time	max time	avg time	std dev
10	6.010e-07	6.560e-07	6.216e-07	1.121e-08
13	6.360e-07	6.840e-07	6.574e-07	1.164e-08
16	6.730e-07	7.310e-07	6.984e-07	1.316e-08
20	7.270e-07	7.940e-07	7.551e-07	1.516e-08
25	8.010e-07	8.780e-07	8.342e-07	1.881e-08
32	9.110e-07	1.017e-06	9.583e-07	2.495e-08
40	1.150e-06	1.280e-06	1.208e-06	2.975e-08
50	1.348e-06	1.491e-06	1.415e-06	3.354e-08
63	1.613e-06	1.779e-06	1.692e-06	3.913e-08
79	2.086e-06	2.285e-06	2.183e-06	4.705e-08
100	2.700e-06	2.940e-06	2.810e-06	5.699e-08
126	3.395e-06	3.728e-06	3.547e-06	7.589e-08
158	4.504e-06	4.832e-06	4.656e-06	8.011e-08
200	5.926e-06	6.295e-06	6.100e-06	8.941e-08
251	7.515e-06	7.982e-06	7.730e-06	1.091e-07
316	1.014e-05	1.071e-05	1.040e-05	1.361e-07
398	1.335e-05	1.397e-05	1.365e-05	1.471e-07
501	1.703e-05	1.780e-05	1.741e-05	1.838e-07
631	2.277e-05	2.368e-05	2.318e-05	2.121e-07
794	3.001e-05	3.105e-05	3.048e-05	2.363e-07
1000	3.846e-05	3.961e-05	3.899e-05	2.741e-07
1259	5.086e-05	5.232e-05	5.153e-05	3.236e-07
1585	6.724e-05	6.892e-05	6.797e-05	3.851e-07
1995	8.603e-05	8.801e-05	8.688e-05	4.573e-07
2512	1.129e-04	1.152e-04	1.139e-04	5.384e-07
3162	1.492e-04	1.520e-04	1.504e-04	6.028e-07
3981	1.909e-04	1.941e-04	1.922e-04	6.750e-07
5012	2.480e-04	2.520e-04	2.496e-04	8.958e-07
6310	3.285e-04	3.324e-04	3.302e-04	8.909e-07
7943	4.197e-04	4.247e-04	4.217e-04	1.148e-06
10000	5.419e-04	5.473e-04	5.443e-04	1.267e-06
12589	7.166e-04	7.227e-04	7.194e-04	1.399e-06
15849	9.144e-04	9.220e-04	9.177e-04	1.748e-06
19953	1.174e-03	1.184e-03	1.178e-03	2.179e-06
25119	1.552e-03	1.561e-03	1.556e-03	2.182e-06
31623	1.981e-03	1.995e-03	1.986e-03	2.851e-06
39811	2.538e-03	2.553e-03	2.544e-03	3.350e-06
50119	3.345e-03	3.360e-03	3.352e-03	3.259e-06

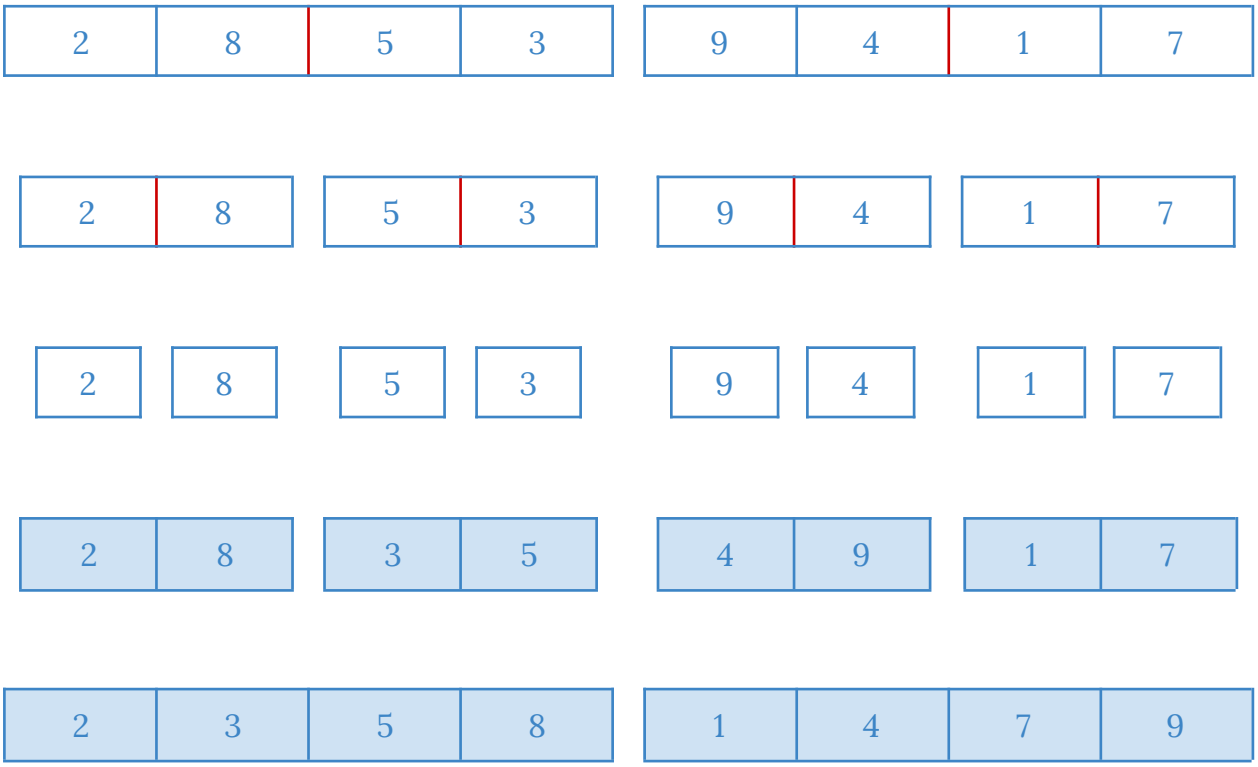
63096	4.261e-03	4.279e-03	4.269e-03	3.939e-06
79433	5.455e-03	5.478e-03	5.464e-03	4.917e-06
100000	7.165e-03	7.188e-03	7.175e-03	4.782e-06
125893	9.117e-03	9.145e-03	9.129e-03	6.014e-06
158489	1.166e-02	1.171e-02	1.168e-02	9.341e-06
199526	1.528e-02	1.533e-02	1.529e-02	9.419e-06
251189	1.945e-02	1.951e-02	1.947e-02	1.132e-05
316228	2.482e-02	2.490e-02	2.485e-02	1.500e-05
398107	3.245e-02	3.478e-02	3.264e-02	3.966e-04
501187	4.124e-02	4.135e-02	4.127e-02	2.147e-05
630957	5.266e-02	5.279e-02	5.271e-02	2.859e-05
794328	6.873e-02	6.892e-02	6.879e-02	3.739e-05



Equação da reta correspondente ao valor médio: $y = 1.766e^{-14}x^2 + 7.325e^{-08}x - 0.000106$

O *Merge Sort*, tal como o *Quick Sort*, é um algoritmo "*Divide and Conquer*". Neste algoritmo, o vetor inicial de elementos a ordenar vai ser dividido a meio. Cada metade vai ser também dividida a meio sucessivamente, geralmente usando recursão, até que restem apenas conjuntos elementares. De seguida, dos conjuntos mais pequenos para os maiores, estes vão sendo juntos de forma ordenada (vão sofrendo *merge*) e no final obtemos um conjunto ordenado.

Dividimos o vetor a meio e de seguida fazemos *merge* de forma ordenada.

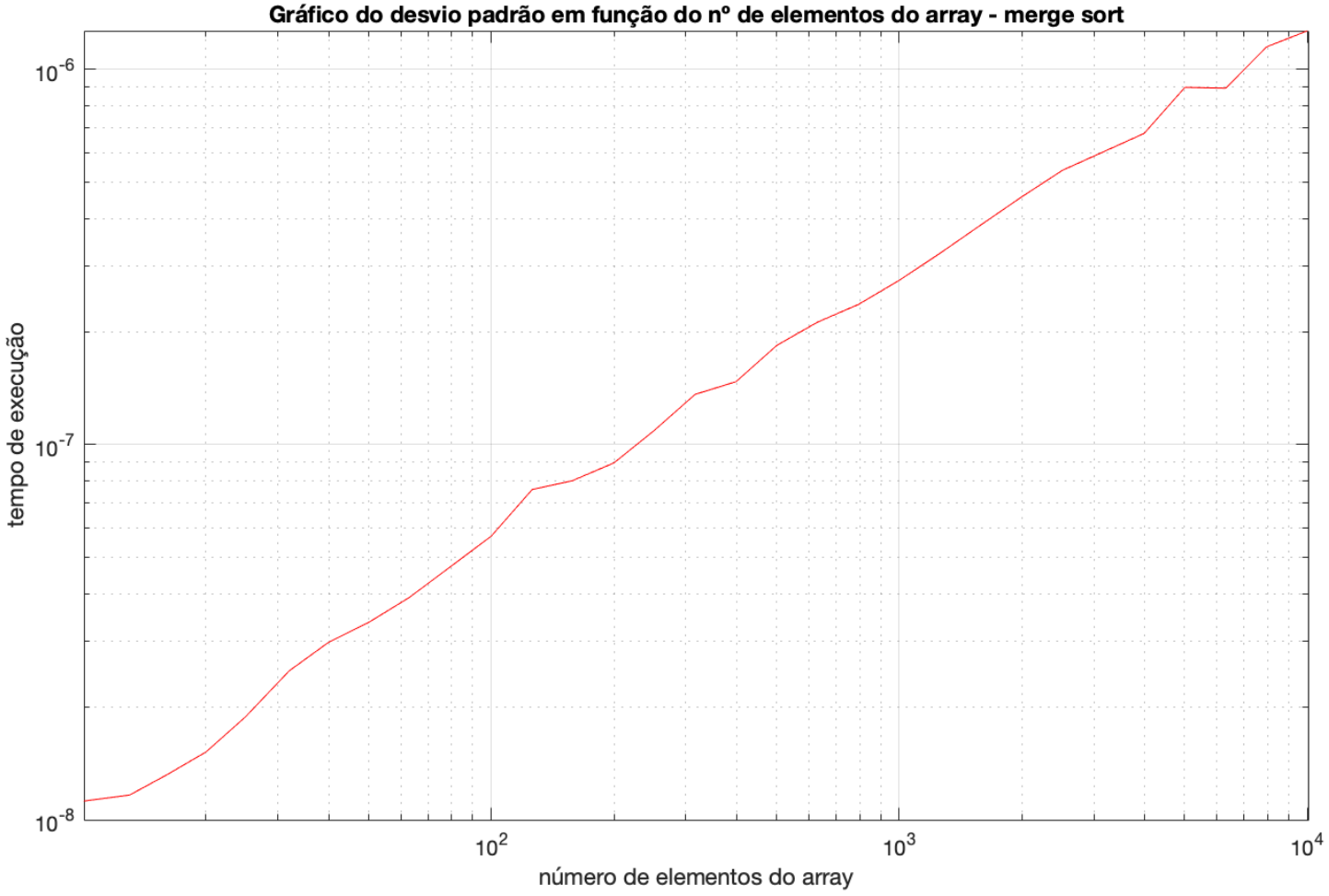


O vetor ordenado terá este aspeto:

Este algoritmo apresenta complexidade O(n*log*(n)) no pior caso (*Big O Notation*), θ(n*log*(n)) no caso médio (*Big Theta Notation*) e Ω(n*log*(n)) no melhor caso (*Big Omega Notation*).

Cada linha representa a relação do número de elementos do vetor com o tempo de execução no pior caso (linha a verde), no caso médio (linha vermelha) e no melhor caso (linha azul). Como as três linhas se mantêm muito próximas, indica que nos três casos a relação é muito idêntica, o que corrobora o facto de os três casos terem complexidade n*log*(n). É possível verificar também que as 3 linhas crescem exponencialmente até o X ter valor 300, aproximadamente, e que a partir daí crescem linearmente.

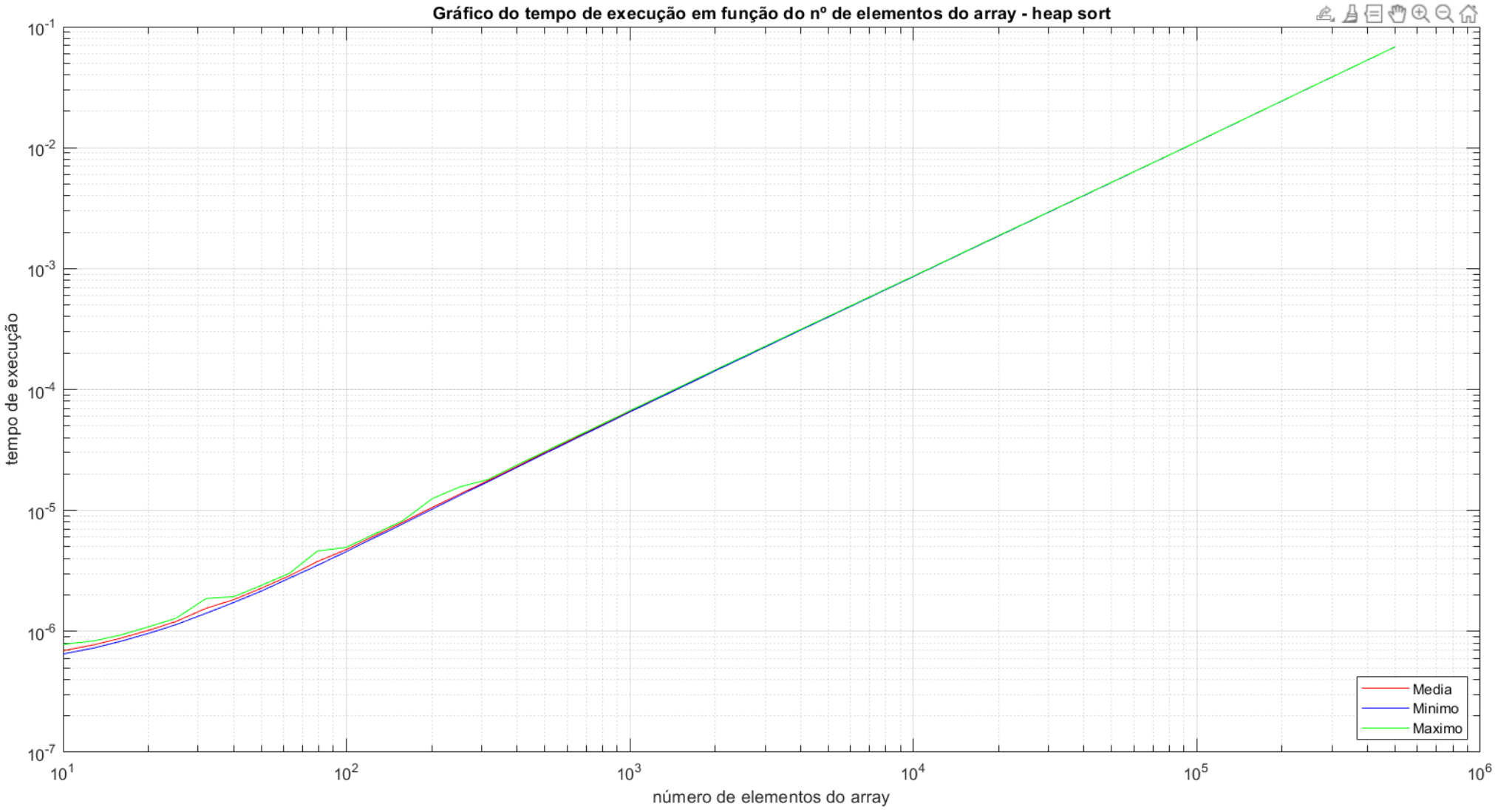
Em relação ao desvio padrão, é possível verificar que o mesmo aumenta quando o número de elementos no vetor aumenta, ou seja, a dispersão de valores em relação ao valor médio no caso do *merge sort* aumenta ao longo do aumento do número de elementos no vetor. Posteriormente, analisaremos os desvios padrões de todos os algoritmos de forma a encontrar o mais estável.



8. Heap Sort

n	min time	max time	avg time	std dev
10	6.490e-07	7.790e-07	6.905e-07	2.177e-08
13	7.330e-07	8.380e-07	7.794e-07	2.370e-08
16	8.280e-07	9.340e-07	8.783e-07	2.549e-08
20	9.610e-07	1.089e-06	1.019e-06	2.961e-08
25	1.135e-06	1.282e-06	1.205e-06	3.512e-08
32	1.412e-06	1.870e-06	1.553e-06	1.149e-07
40	1.732e-06	1.934e-06	1.828e-06	4.753e-08
50	2.150e-06	2.397e-06	2.271e-06	5.459e-08
63	2.756e-06	3.025e-06	2.885e-06	6.446e-08
79	3.517e-06	4.616e-06	3.782e-06	2.556e-07
100	4.574e-06	4.955e-06	4.758e-06	8.866e-08
126	5.974e-06	6.419e-06	6.189e-06	1.044e-07
158	7.735e-06	8.221e-06	7.978e-06	1.146e-07
200	1.021e-05	1.247e-05	1.054e-05	2.955e-07
251	1.332e-05	1.565e-05	1.366e-05	2.415e-07
316	1.729e-05	1.803e-05	1.764e-05	1.722e-07
398	2.257e-05	2.363e-05	2.300e-05	2.187e-07
501	2.953e-05	3.069e-05	3.001e-05	2.442e-07
631	3.840e-05	3.981e-05	3.897e-05	2.996e-07
794	4.995e-05	5.145e-05	5.060e-05	3.317e-07
1000	6.517e-05	6.690e-05	6.591e-05	3.793e-07
1259	8.451e-05	8.644e-05	8.530e-05	4.181e-07
1585	1.095e-04	1.119e-04	1.105e-04	5.084e-07
1995	1.423e-04	1.451e-04	1.435e-04	5.989e-07
2512	1.840e-04	1.872e-04	1.853e-04	6.825e-07
3162	2.381e-04	2.417e-04	2.396e-04	7.954e-07
3981	3.084e-04	3.125e-04	3.102e-04	9.216e-07
5012	3.980e-04	4.027e-04	4.001e-04	1.021e-06
6310	5.143e-04	5.194e-04	5.167e-04	1.171e-06
7943	6.654e-04	6.715e-04	6.681e-04	1.397e-06
10000	8.595e-04	8.664e-04	8.626e-04	1.595e-06
12589	1.112e-03	1.119e-03	1.115e-03	1.830e-06
15849	1.439e-03	1.448e-03	1.443e-03	2.025e-06
19953	1.857e-03	1.871e-03	1.863e-03	2.769e-06
25119	2.398e-03	2.410e-03	2.404e-03	2.632e-06
31623	3.103e-03	3.118e-03	3.110e-03	3.256e-06
39811	4.003e-03	4.022e-03	4.010e-03	3.815e-06
50119	5.166e-03	5.187e-03	5.175e-03	4.390e-06

63096	6.682e-03	6.703e-03	6.692e-03	4.674e-06
79433	8.625e-03	8.648e-03	8.636e-03	5.088e-06
100000	1.116e-02	1.119e-02	1.118e-02	6.852e-06
125893	1.448e-02	1.452e-02	1.450e-02	8.107e-06
158489	1.876e-02	1.881e-02	1.877e-02	1.007e-05
199526	2.430e-02	2.435e-02	2.432e-02	1.059e-05
251189	3.151e-02	3.157e-02	3.153e-02	1.196e-05
316228	4.073e-02	4.081e-02	4.076e-02	1.538e-05
398107	5.275e-02	5.282e-02	5.278e-02	1.656e-05
501187	6.823e-02	6.832e-02	6.826e-02	1.896e-05



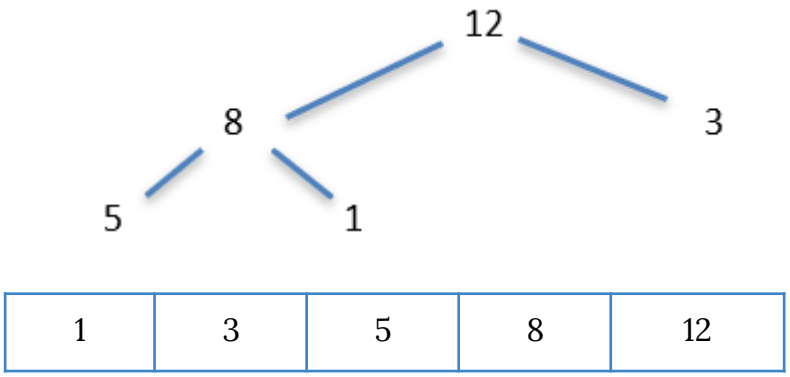
Equação da reta correspondente ao valor médio:$y = 5.227e^{-14}x^2 + 1.112x - 0.0001308$

O algoritmo Heap Sort usa como base uma estrutura do tipo Binary Heap, sendo um algoritmo similar ao Selection Sort no sentido em que encontramos primeiro o elemento máximo colocando-o no fim. Este processo é depois repetido para os restantes elementos.

Para explicar este algoritmo vamos começar por explicar em que consiste a estrutura Binary Heap, esta estrutura tem por base uma árvore binária completa (árvore em que todos os níveis estão completos exceto o último), nesta estrutura os elementos encontram se ordenados de forma a que o nó pai é maior que os seus filhos (max heap), esta estrutura pode ser representada num array em que estando o pai no índice i os filhos podem ser calculados por 2*i+1 (ou +2 pro filho da direita)

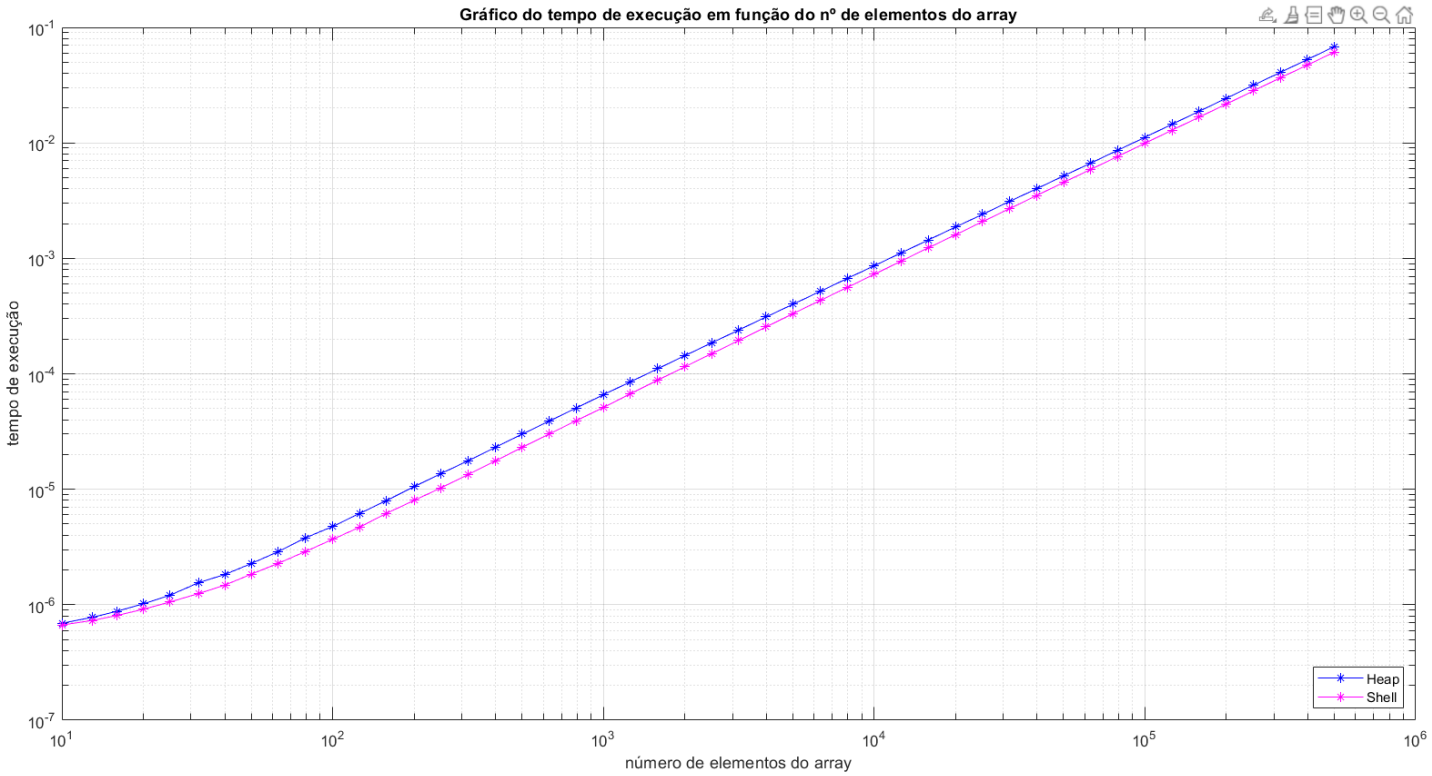
Após ser gerado o heap é criado um array removendo sucessivamente o maior elemento do heap colocando-o no último espaço vazio do array, assim podemos também perceber que o Heap Sort não requer espaço extra.

Num exemplo com os elementos 5,12,3,8,1 era gerada a seguinte heap e o seguinte array.

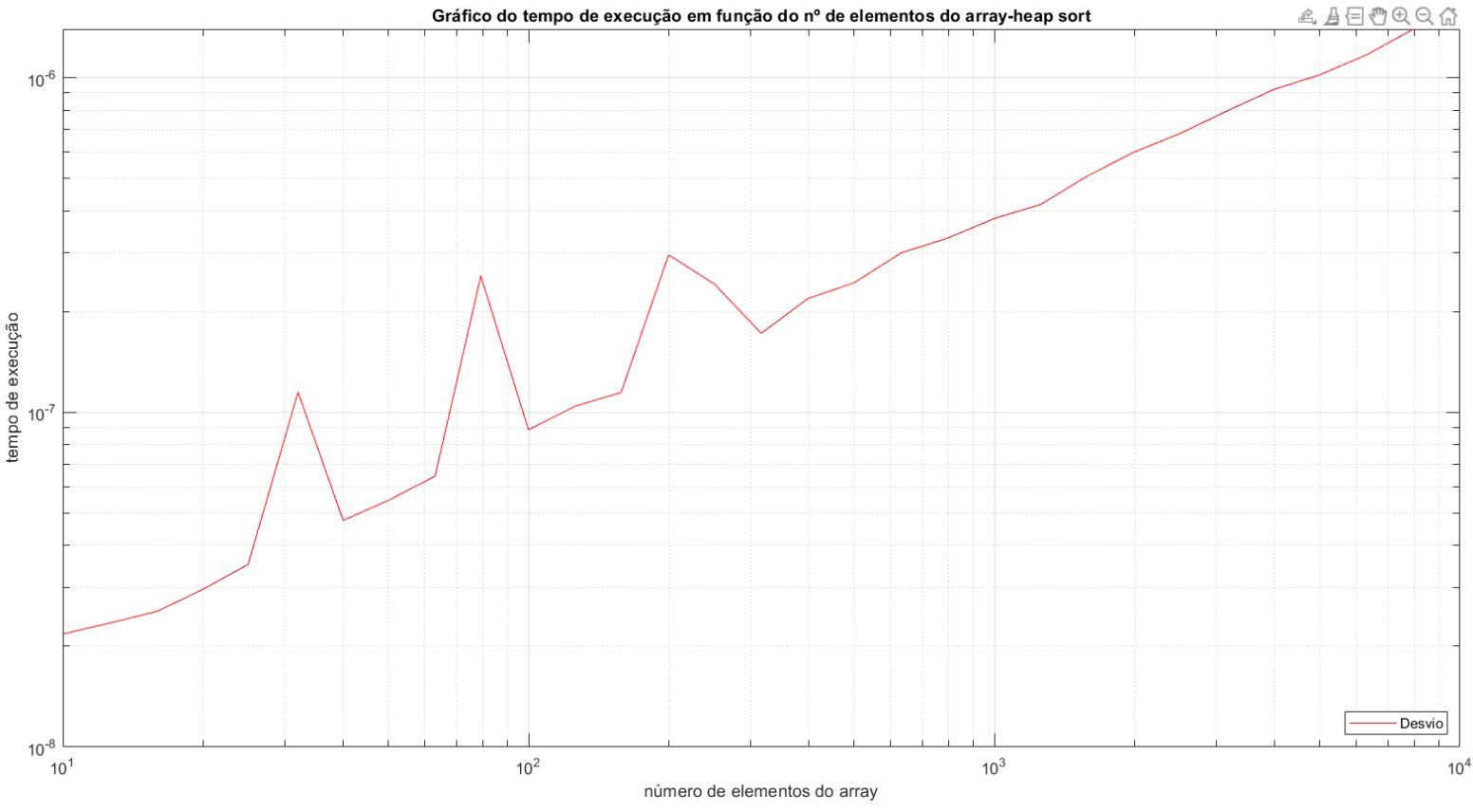


Relativamente à sua complexidade o caso médio, bem como o melhor e pior caso uma complexidade (em Big O Notation) O(nlog(n)).

Por análise do gráfico podemos concluir que o tempo de execução varia de forma exponencial, tendo em conta a sua complexidade computacional este método é bastante fiável embora não seja o melhor, sendo mais lento que o Shell Sort para todos os tamanhos de array.

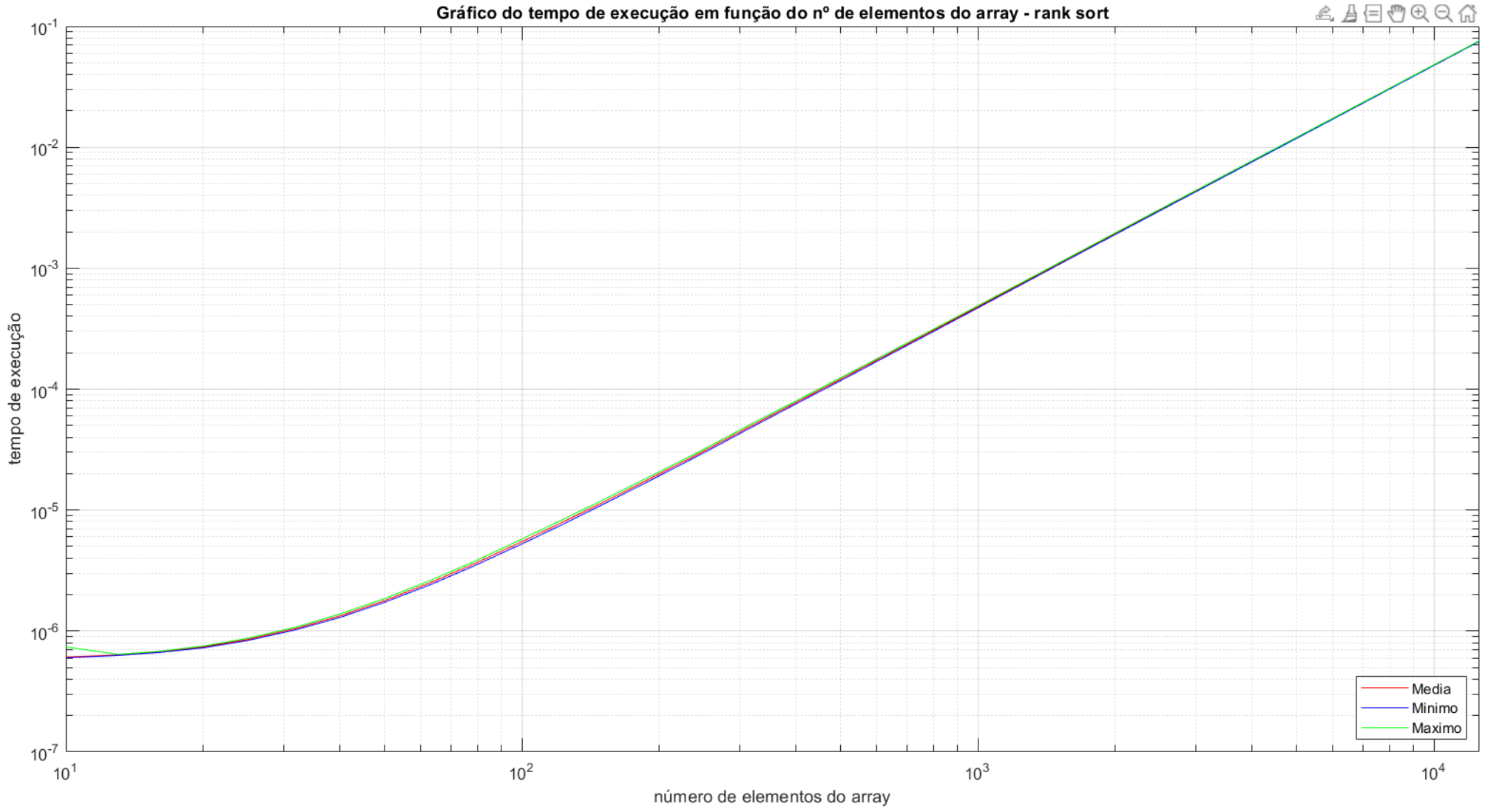


Quanto ao desvio padrão, como é possível ver no gráfico seguinte, aumenta em relação ao tempo o de forma mais ao menos linear (apesar de ter algumas irregularidades).



9. Rank Sort

n	min time	max time	avg time	std dev
10	6.010e-07	7.380e-07	6.093e-07	2.066e-08
13	6.280e-07	6.420e-07	6.343e-07	3.163e-09
16	6.640e-07	6.790e-07	6.714e-07	3.679e-09
20	7.260e-07	7.500e-07	7.375e-07	5.780e-09
25	8.340e-07	8.700e-07	8.502e-07	8.572e-09
32	1.026e-06	1.079e-06	1.051e-06	1.279e-08
40	1.299e-06	1.387e-06	1.338e-06	2.029e-08
50	1.729e-06	1.856e-06	1.785e-06	2.936e-08
63	2.416e-06	2.621e-06	2.513e-06	4.765e-08
79	3.488e-06	3.788e-06	3.629e-06	7.041e-08
100	5.247e-06	5.766e-06	5.482e-06	1.148e-07
126	7.968e-06	8.739e-06	8.325e-06	1.752e-07
158	1.219e-05	1.324e-05	1.269e-05	2.470e-07
200	1.918e-05	2.070e-05	1.990e-05	3.568e-07
251	2.984e-05	3.195e-05	3.085e-05	5.128e-07
316	4.746e-05	5.079e-05	4.903e-05	7.667e-07
398	7.507e-05	7.942e-05	7.716e-05	1.051e-06
501	1.180e-04	1.249e-04	1.213e-04	1.572e-06
631	1.870e-04	1.964e-04	1.916e-04	2.244e-06
794	2.958e-04	3.092e-04	3.023e-04	3.083e-06
1000	4.694e-04	4.887e-04	4.784e-04	4.550e-06
1259	7.472e-04	7.733e-04	7.605e-04	6.284e-06
1585	1.190e-03	1.227e-03	1.208e-03	8.935e-06
1995	1.888e-03	1.946e-03	1.915e-03	1.411e-05
2512	2.996e-03	3.081e-03	3.034e-03	1.967e-05
3162	4.745e-03	4.860e-03	4.800e-03	2.685e-05
3981	7.515e-03	7.681e-03	7.594e-03	3.764e-05
5012	1.190e-02	1.213e-02	1.201e-02	5.328e-05
6310	1.886e-02	1.918e-02	1.903e-02	7.485e-05
7943	2.991e-02	3.035e-02	3.012e-02	1.020e-04
10000	4.741e-02	4.805e-02	4.772e-02	1.459e-04
12589	7.517e-02	7.604e-02	7.561e-02	2.075e-04



Equação da reta correspondente ao valor médio: $y = 4.763e^{-10}x^2 + 9.921e^{-09}x - 1.038e^{-06}$

No *rank sort* vamos ter dois vetores.. Um com os elementos (*list*) e outro com o *rank* de cada elemento (*rank_list*), com as mesmas dimensões e inicializado a 0. Começamos por calcular o *rank* de cada elemento e colocar o valor em *rank_list*. O *rank* vai ser o número de vezes que esse determinado elemento vai ser maior do que os outros e vai ser incrementado de acordo com as comparações que se fizerem. No final, o vetor vai ser ordenado do menor *rank* para o maior.

Na seguinte tabela temos na primeira linha o vetor (*list*) e na segunda linha os *ranks* (*rank_list*).

2	8	5	3	9	4	1
0	1	0	0	0	0	0

Caso o valor do elemento a vermelho seja maior ou igual que o valor a verde, aumenta-se o rank no índice do número a vermelho. Caso seja menor, aumenta-se o rank no índice do número a verde.

2	8	5	3	9	4	1
0	2	1	0	0	0	0

2	8	5	3	9	4	1
0	3	2	1	0	0	0

2	8	5	3	9	4	1
0	3	2	1	4	0	0

2	8	5	3	9	4	1
0	4	3	1	5	2	0

2	8	5	3	9	4	1
1	5	4	2	6	3	0

No final, ordenamos o vetor de números por ordem de acordo com o *rank_list*.

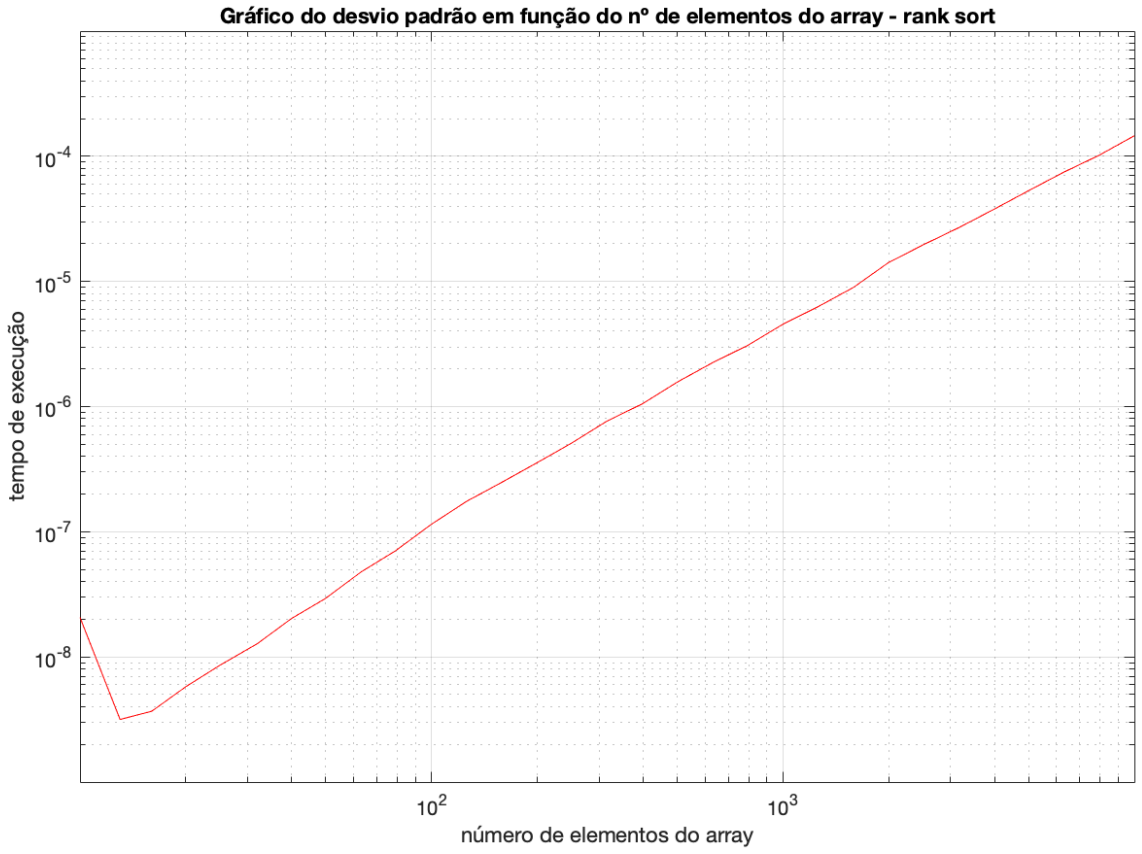
```
for (i=0; i<número de elementos no vetor;;i++)
{
    sorted_list[rank_list[i]]=list[i];
}
```

O vetor ordenado terá então este aspeto:

1	2	3	4	5	8	9

Este algoritmo apresenta complexidade $O(n^2)$ no pior caso (*Big O Notation*), $\theta(n^2)$ no caso médio (*Big Theta Notation*) e $\Omega(n^2)$ no melhor caso (*Big Omega Notation*).

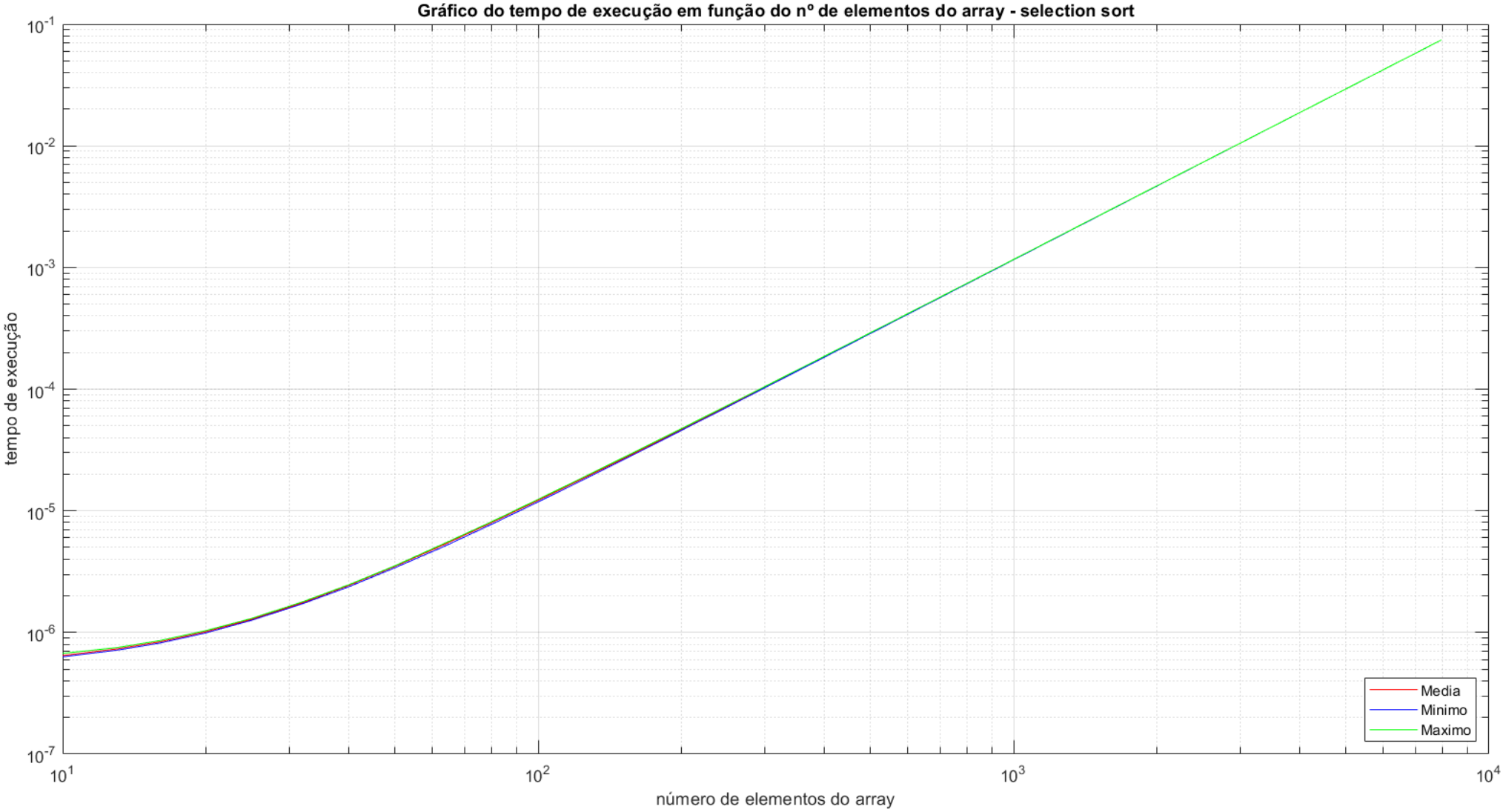
Cada linha representa a relação do número de elementos do array com o tempo de execução no pior caso (linha a verde), no caso médio (linha vermelha) e no melhor caso (linha azul). Como as três linhas se mantêm muito próximas, indica que nos três casos a relação é muito idêntica, o que corrobora o facto de os três casos terem complexidade n^2. É possível observar que para os primeiros elementos o tempo máximo de execução é superior à média e ao tempo mínimo, no entanto esta diferença deixa de existir para 13 elementos, aproximadamente, e as três linhas adotam agora o mesmo comportamento (crescem exponencialmente até 90, aproximadamente e após esse valor crescem linearmente).



Em relação ao desvio padrão, é possível verificar que o mesmo aumenta quando o número de elementos no *vetor* aumenta, ou seja, a dispersão de valores em relação ao valor médio no caso do *rank sort* aumenta ao longo do aumento do número de elementos no vetor. É ainda notável a presença de uma depressão acentuada, logo ao início, que é consequência do algoritmo não ser um algoritmo estável. Posteriormente, analisaremos os desvios padrões de todos os algoritmos de forma a encontrar o mais estável.

10. Selection Sort

n	min time	max time	avg time	std dev
10	6.310e-07	6.730e-07	6.464e-07	8.290e-09
13	7.150e-07	7.510e-07	7.322e-07	8.213e-09
16	8.180e-07	8.570e-07	8.370e-07	9.208e-09
20	9.920e-07	1.036e-06	1.014e-06	1.006e-08
25	1.261e-06	1.306e-06	1.283e-06	1.006e-08
32	1.725e-06	1.790e-06	1.757e-06	1.278e-08
40	2.377e-06	2.471e-06	2.439e-06	1.617e-08
50	3.405e-06	3.537e-06	3.496e-06	2.662e-08
63	5.035e-06	5.322e-06	5.213e-06	5.531e-08
79	7.600e-06	7.977e-06	7.844e-06	9.193e-08
100	1.183e-05	1.242e-05	1.218e-05	1.398e-07
126	1.842e-05	1.929e-05	1.892e-05	2.058e-07
158	2.866e-05	2.986e-05	2.931e-05	2.927e-07
200	4.571e-05	4.729e-05	4.651e-05	3.749e-07
251	7.182e-05	7.394e-05	7.289e-05	4.822e-07
316	1.140e-04	1.166e-04	1.153e-04	5.886e-07
398	1.811e-04	1.844e-04	1.826e-04	7.627e-07
501	2.879e-04	2.920e-04	2.897e-04	9.524e-07
631	4.580e-04	4.630e-04	4.602e-04	1.168e-06
794	7.275e-04	7.334e-04	7.300e-04	1.324e-06
1000	1.157e-03	1.164e-03	1.160e-03	1.569e-06
1259	1.840e-03	1.849e-03	1.844e-03	1.988e-06
1585	2.924e-03	2.936e-03	2.928e-03	2.504e-06
1995	4.643e-03	4.655e-03	4.647e-03	2.581e-06
2512	7.375e-03	7.389e-03	7.379e-03	2.631e-06
3162	1.170e-02	1.171e-02	1.171e-02	2.349e-06
3981	1.858e-02	1.859e-02	1.858e-02	3.009e-06
5012	2.948e-02	2.950e-02	2.949e-02	2.940e-06
6310	4.678e-02	4.679e-02	4.678e-02	3.543e-06
7943	7.418e-02	7.420e-02	7.419e-02	4.854e-06



Equação da reta correspondente ao valor médio: $y = 1.179e^{-09}x^2 + 2.581e^{-08}x - 3.594e^{-06}$

Este método de ordenação consiste em encontrar o elemento mais pequeno numa partição e colocá-lo numa outra partição já ordenada. Na primeira iteração, o mínimo é o primeiro elemento da partição não ordenada. De seguida, iremos percorrer a partição à procura de um elemento menor que esse e caso haja algum, esse passa a ser o mínimo. Quando todos os elementos forem percorridos, o mínimo trocará de posição com o primeiro elemento da partição não ordenada. Este elemento fará agora parte da partição ordenada.

2	8	5	3	9	4	1

O mínimo começa por ser o elemento 2, que é o primeiro elemento. Após a partição ser percorrida, percebemos que o elemento 1 é mais pequeno que o elemento 2, então passa esse a ser o mínimo.

2	8	5	3	9	4	1

O mínimo troca então de posição com o primeiro elemento da partição não ordenada, ficando este já ordenado.

1	8	5	3	9	4	2

A sequência repete-se até que todos os elementos estejam ordenados.

1	8	5	3	9	4	2
1	2	5	3	9	4	8
1	2	5	3	9	4	8
1	2	3	5	9	4	8
1	2	3	5	9	4	8
1	2	3	4	9	5	8

1	2	3	4	9	5	8

1	2	3	4	5	9	8

1	2	3	4	5	9	8

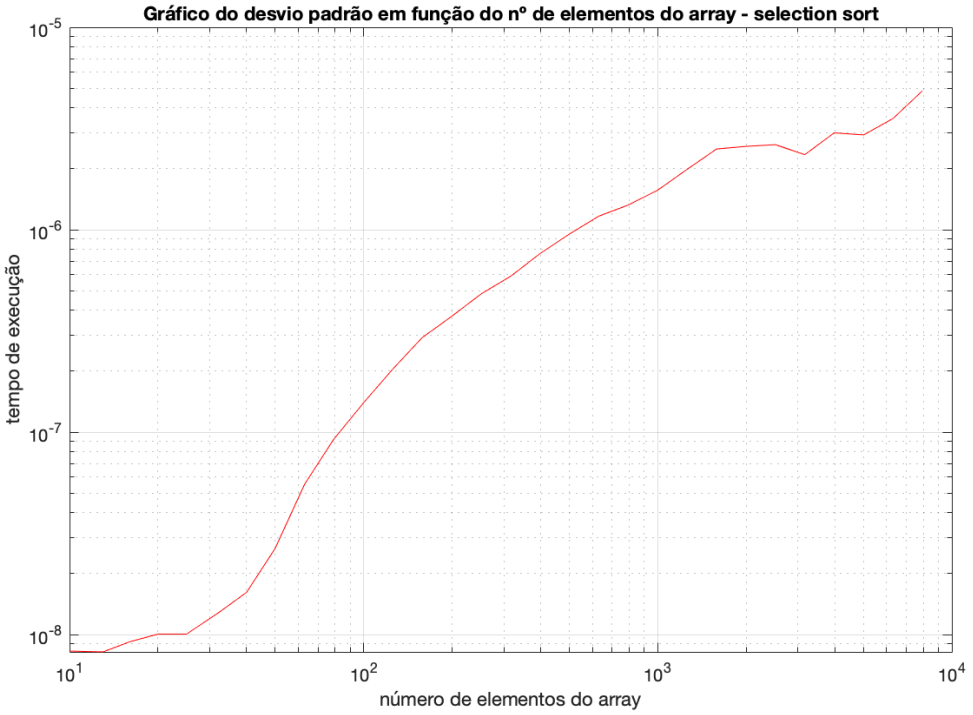
1	2	3	4	5	8	9

Este algoritmo apresenta complexidade $O(n^2)$ no pior caso (*Big O Notation*), $\theta(n^2)$ no caso médio (*Big Theta Notation*) e $\Omega(n^2)$ no melhor caso (*Big Omega Notation*).

Cada linha representa a relação do número de elementos do array com o tempo de execução no pior caso (linha a verde), no caso médio (linha vermelha) e no melhor caso (linha azul). Como as três linhas se mantêm muito próximas, indica que nos três casos a relação é muito idêntica, o que corrobora o facto de os três casos terem complexidade $n^{2.}$

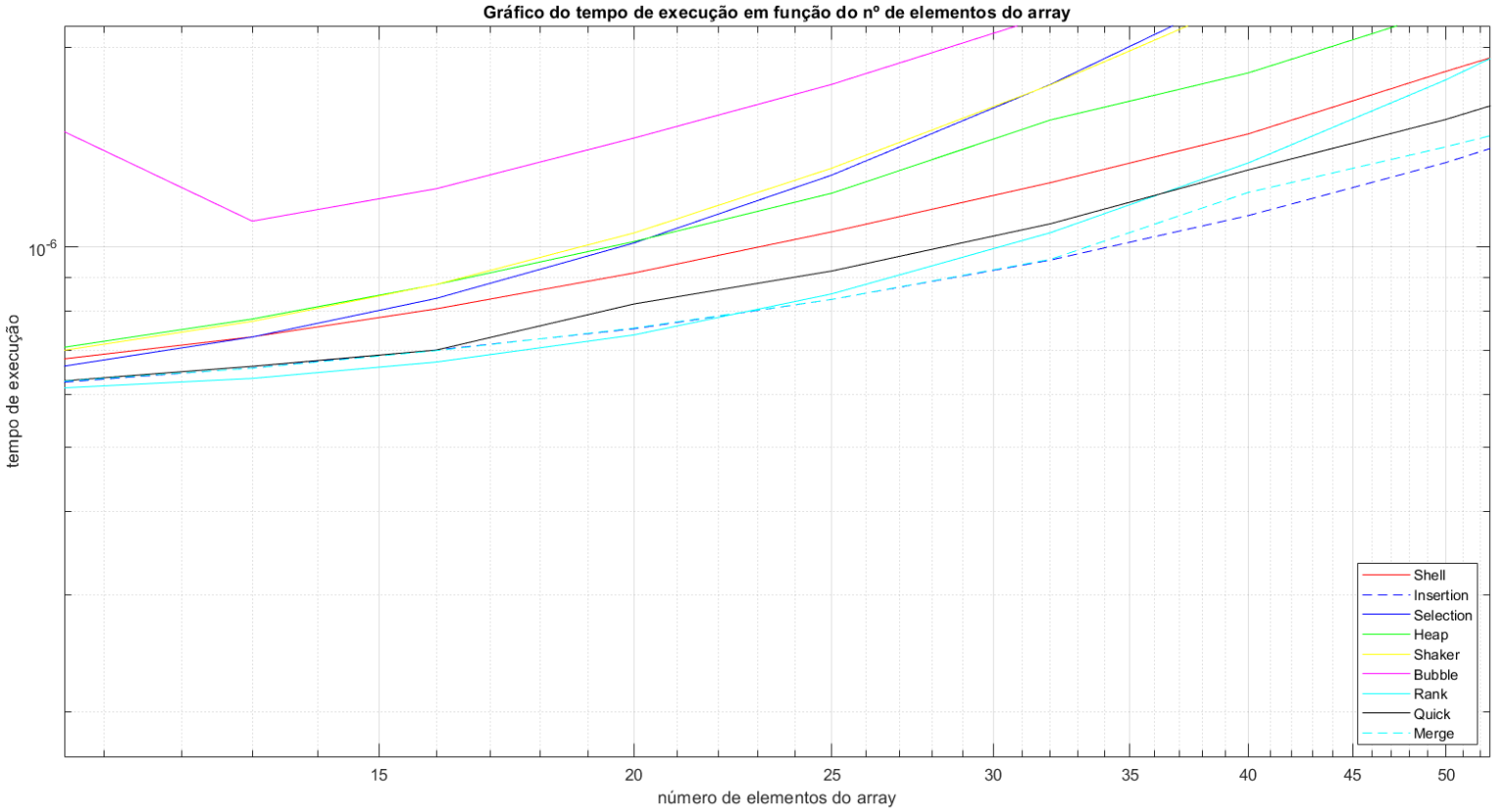
É possível verificar também que as 3 linhas crescem exponencialmente até o X ter valor 40, aproximadamente, e que a partir daí crescem linearmente.

Em relação ao desvio padrão, é possível verificar que o mesmo aumenta quando o número de elementos no *vetor* aumenta, ou seja, a dispersão de valores em relação ao valor médio no caso do *rank sort* aumenta ao longo do aumento do número de elementos no vetor. É ainda notável a presença de algumas pequenas depressões, que são consequência do algoritmo não ser um algoritmo estável. Posteriormente, analisaremos os desvios padrões de todos os algoritmos de forma a encontrar o mais estável.



11. Comparação entre Algoritmos

Na página seguinte encontra-se um gráfico com todas as funções referentes aos casos médios de cada algoritmo mencionado anteriormente (usaremos esse gráfico para as conclusões, porém iremos apresentar também os gráficos do melhor caso e do pior caso). Por observação direta do gráfico, sem necessidade de ampliação, notamos que o *bubble sort* claramente não é uma boa opção como algoritmo de ordenação, uma vez que durante todo o intervalo de valores que n assume, o seu tempo de execução é mais elevado que a maioria dos restantes algoritmos (até n igual a 63, o seu tempo de execução é superior a todos os restantes algoritmos). Outros algoritmos que também se destacam por possuírem um tempo de execução elevado a partir de determinado valor de n são o *shaker sort* (é um *bubble sort* bi-direcional) e o *selection sort*. É importante, antes de descartarmos estes algoritmos, verificarmos se a sua eficiência também é pouco razoável no começo, para valores de n pequenos.



A partir do gráfico ampliado, é possível verificar que o *shaker sort* e o *bubble sort* não apresentam um bom comportamento para *arrays* com um menor comprimento quando comparados com outros algoritmos analisados. É ainda verificável que o *shell sort* e o *heap sort* não apresentam também uma excelente eficiência em comparação com os restantes e isso verifica-se tanto para um n pequeno como para um n elevado.

Feita esta análise, descartamos então o *bubble sort*, o *selection sort*, o *shaker sort*, o *shell sort* e o *heap sort*, permanecendo com o *rank sort*, o *merge sort*, o *quick sort* e o *insertion sort*, os quais analisaremos num novo gráfico.

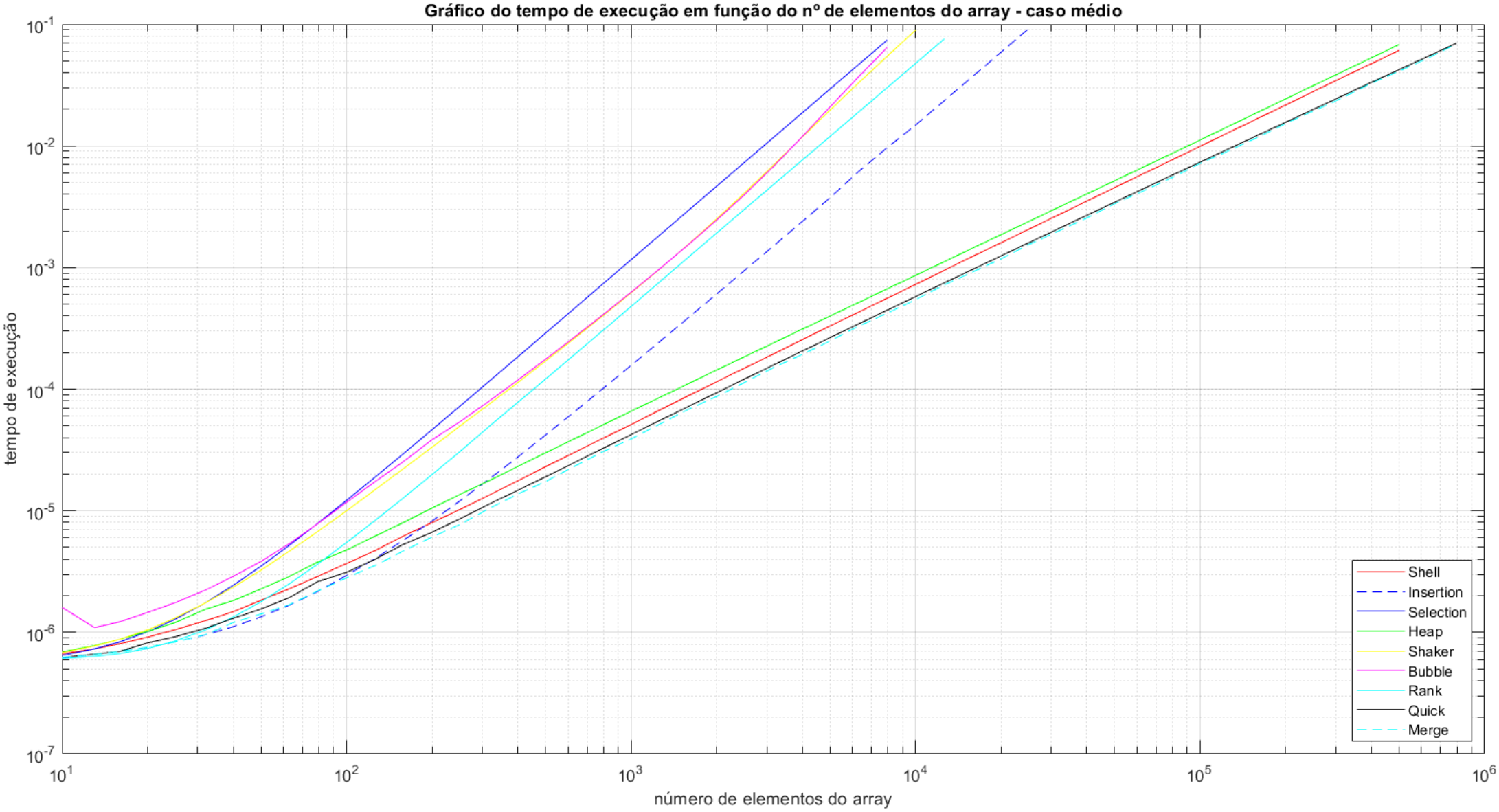


Gráfico do tempo de execução em função do nº de elementos do array - caso médio
tempo de execução
número de elementos do array
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
Shell
Insertion
Selection
Heap
Shaker
Bubble
Rank
Quick
Merge

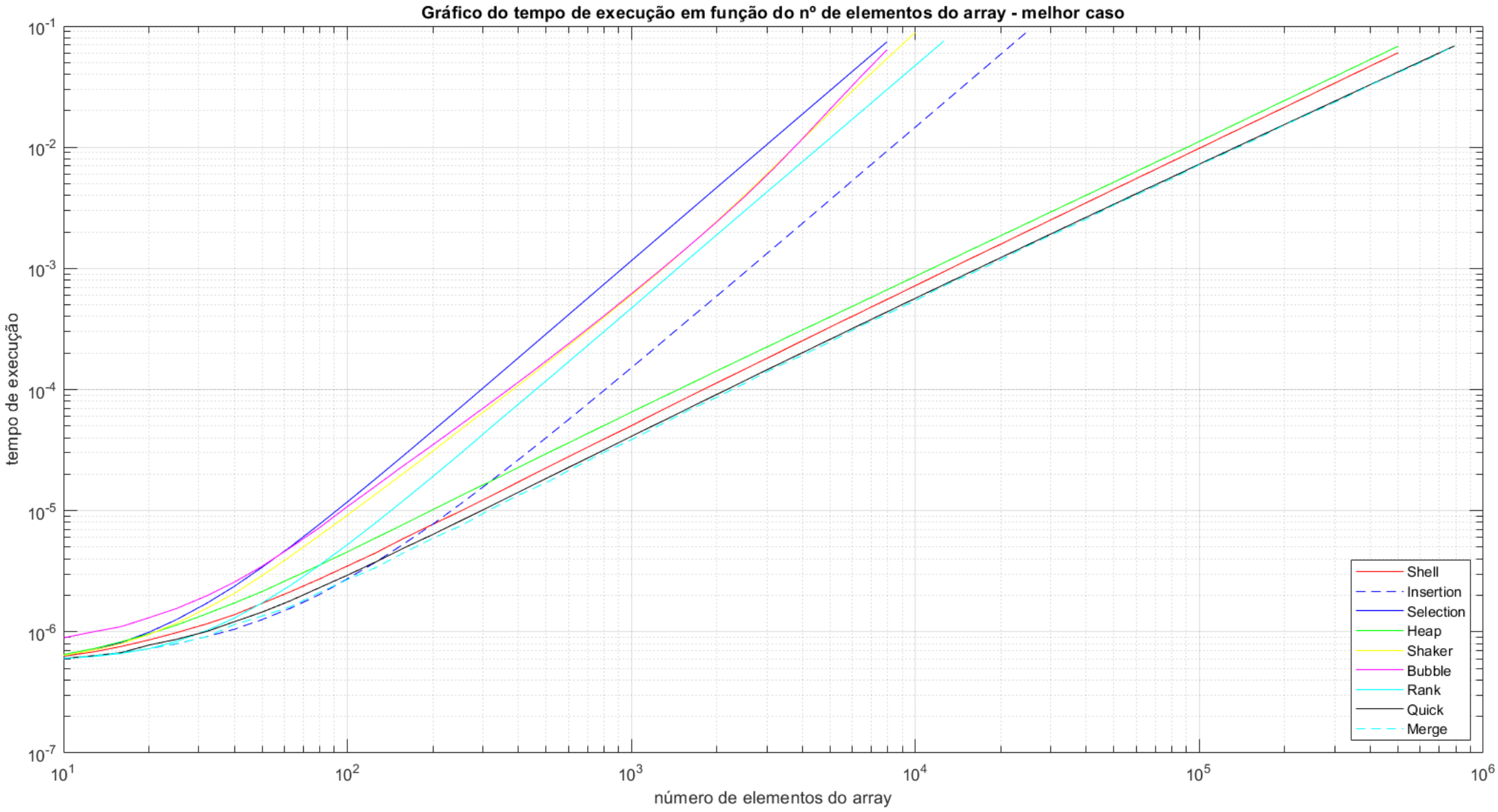


Gráfico do tempo de execução em função do nº de elementos do array - melhor caso
tempo de execução
número de elementos do array
Shell
Insertion
Selection
Heap
Shaker
Bubble
Rank
Quick
Merge

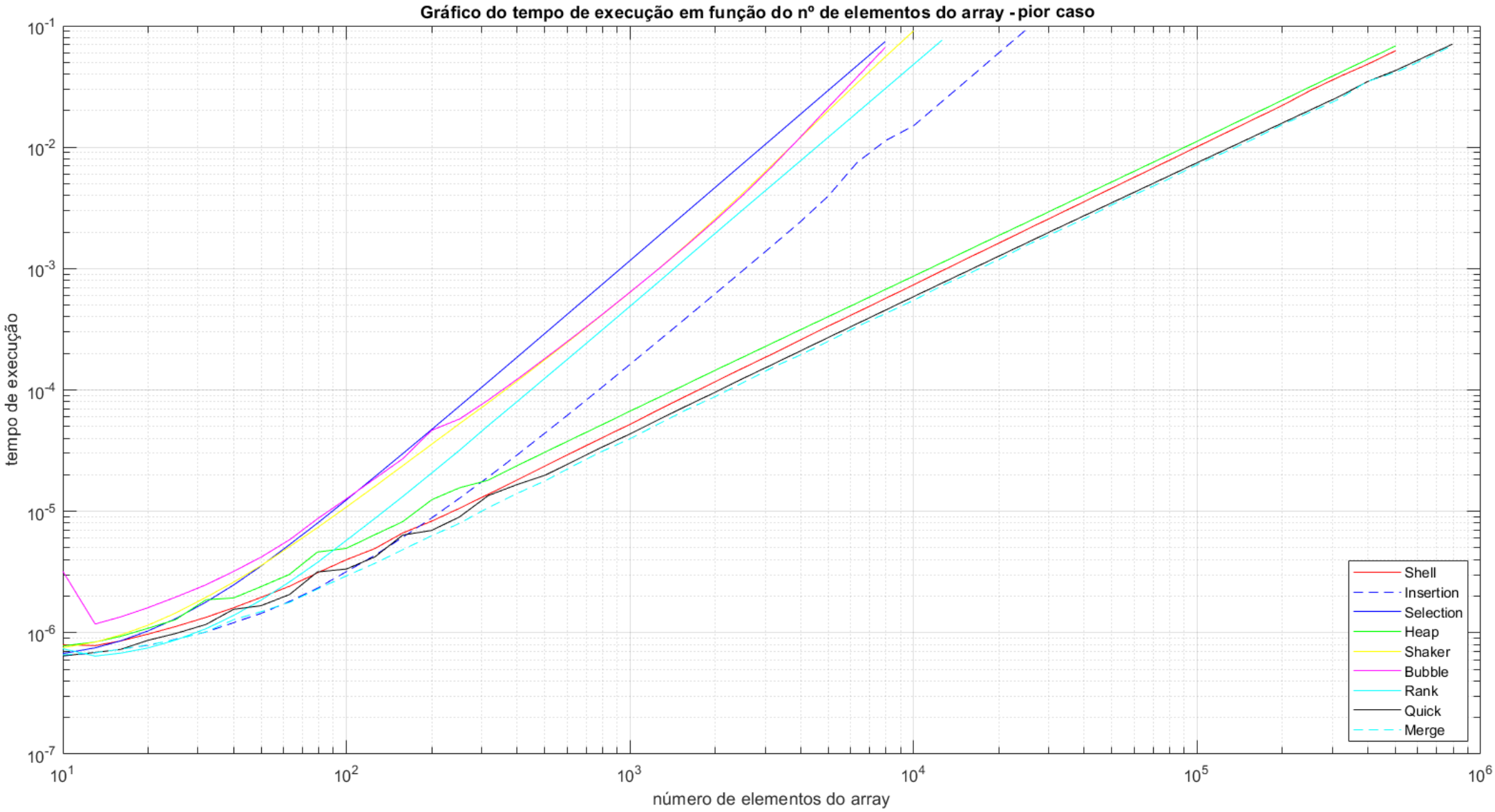


Gráfico do tempo de execução em função do nº de elementos do array - pior caso
tempo de execução
número de elementos do array
Shell
Insertion
Selection
Heap
Shaker
Bubble
Rank
Quick
Merge

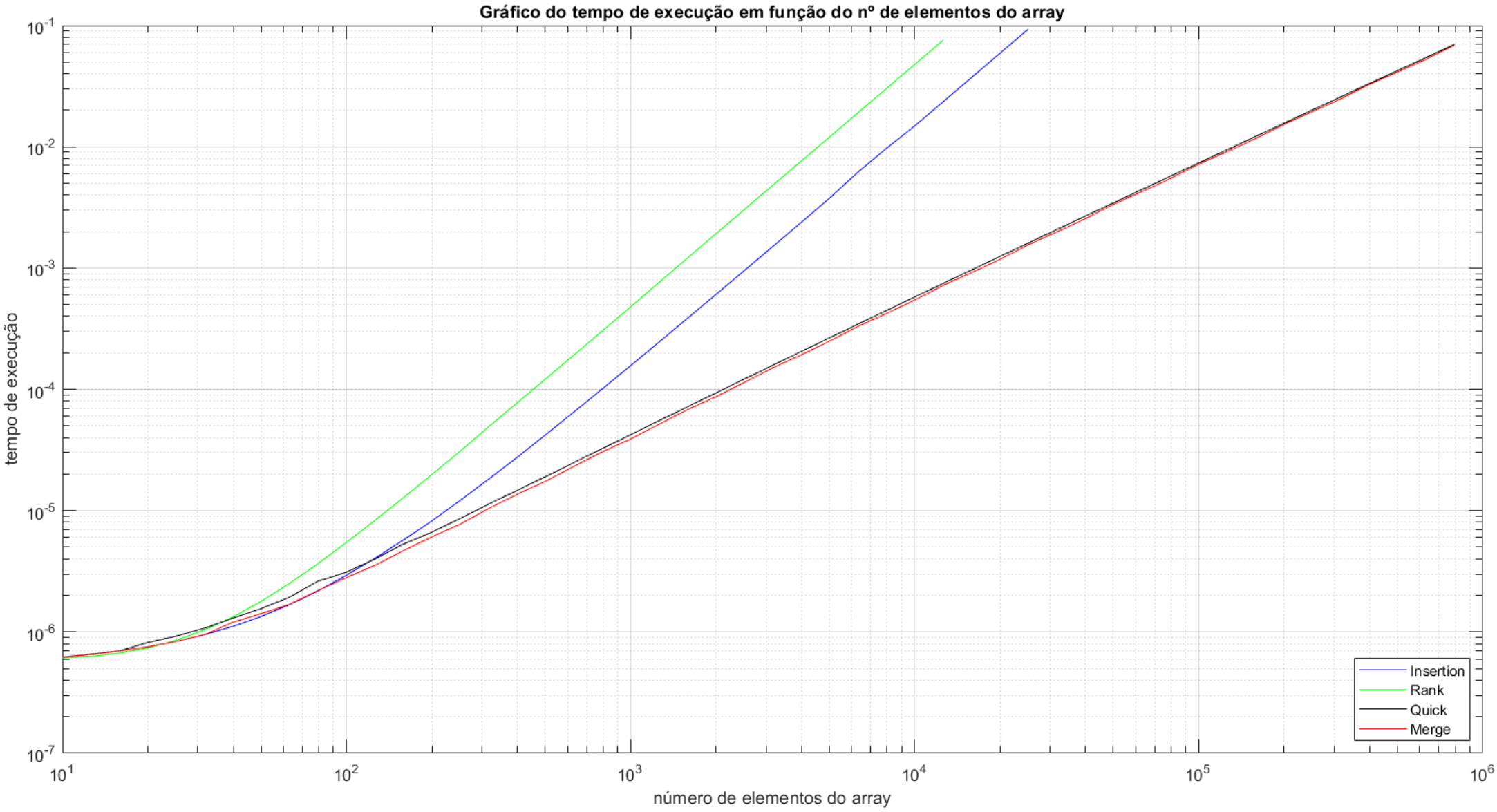
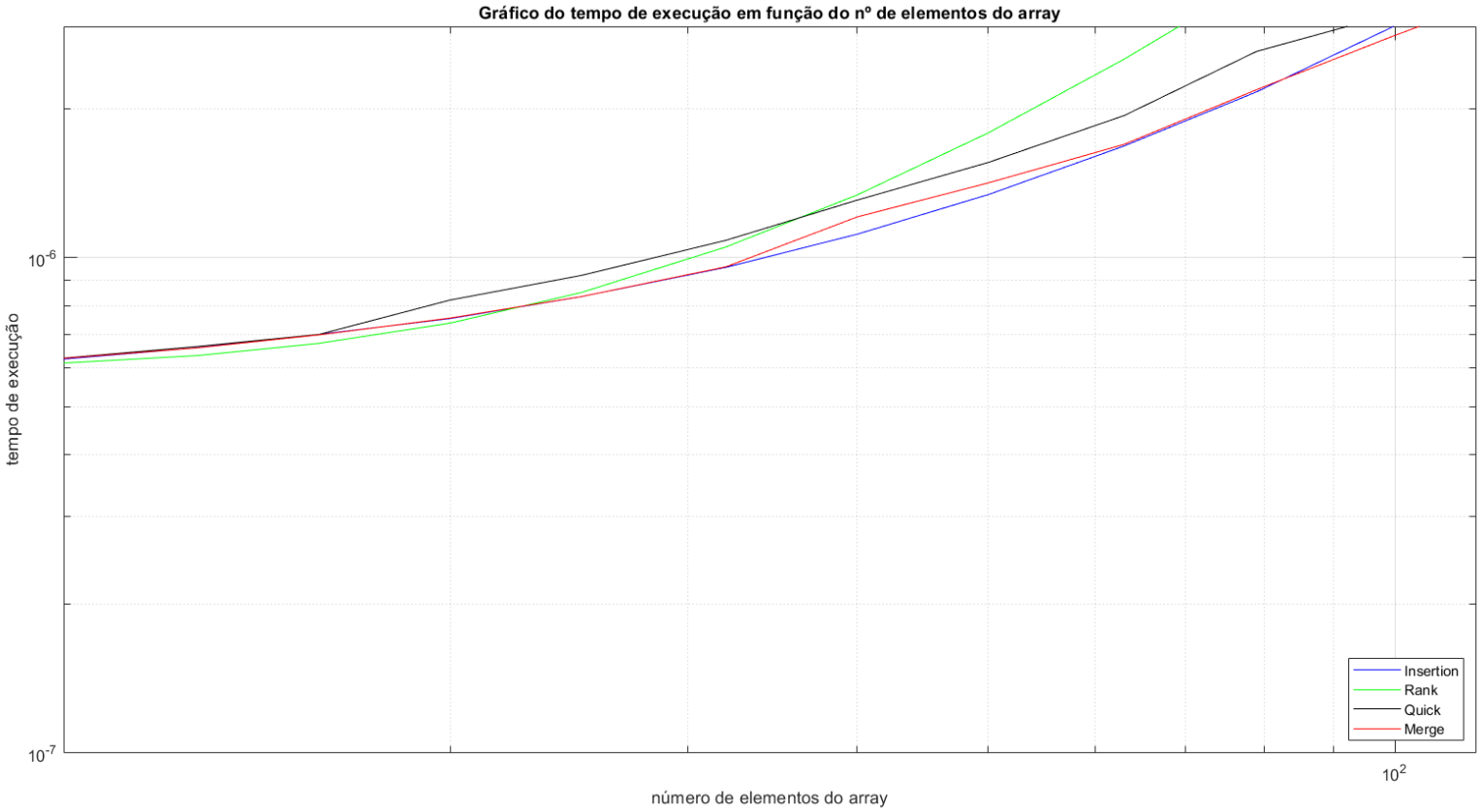


Gráfico do tempo de execução em função do nº de elementos do array
tempo de execução
número de elementos do array
Insertion
Rank
Quick
Merge

Com um novo gráfico contendo apenas os algoritmos de ordenação que não descartamos, conseguimos perceber que quando n tende para infinito, o *rank sort* e o *insertion sort* vão se provar menos eficientes que o *merge sort* e *quick sort*. Contudo, antes de descartarmos ambos os algoritmos, é necessário verificar o comportamento dos dois para valores de n pequenos.



É perceptível que o *rank sort* é o melhor algoritmo a ser implementado a nível de tempo de execução no intervalo de valores de n [10, 20], porém não nos devemos esquecer que é o pior algoritmo a nível de complexidade entre os quatro. Para um intervalo de [20, 32], o *insertion sort* e o *merge sort* apresentam uma grande semelhança nos tempos de execução, pelo que ambos são uma boa opção de implementação. Para um intervalo de [32, 79], o *insertion sort* é a melhor escolha de algoritmo de ordenação. Para n igual a 40, o *rank sort* deixa de ser uma opção viável quando comparada com todas as restantes, pelo que descartamos, assim como descartamos o *insertion sort* quando n assume valores superiores a 126. É relevante mencionar ainda que o *insertion sort* tem uma pior complexidade computacional quando comparado com o *merge sort* e o *quick sort*.

Resta, portanto, o *quick sort* e o *merge sort*, pelo que iremos analisar três gráficos de comparação entre ambos:

1) Comparação do melhor caso. ($\Omega(nlog\ (n))$ no *quick sort* e $\Omega(nlog\ (n))$ no *merge sort*)
2) Comparação do pior caso. ($\theta(nlog\ (n))$no *quick sort e* $\theta(nlog\ (n))$ no *merge sort*)

3) Comparação do caso médio. (O(n^2) no *quick sort* e O($nlog\ (n)$) no *merge sort*)

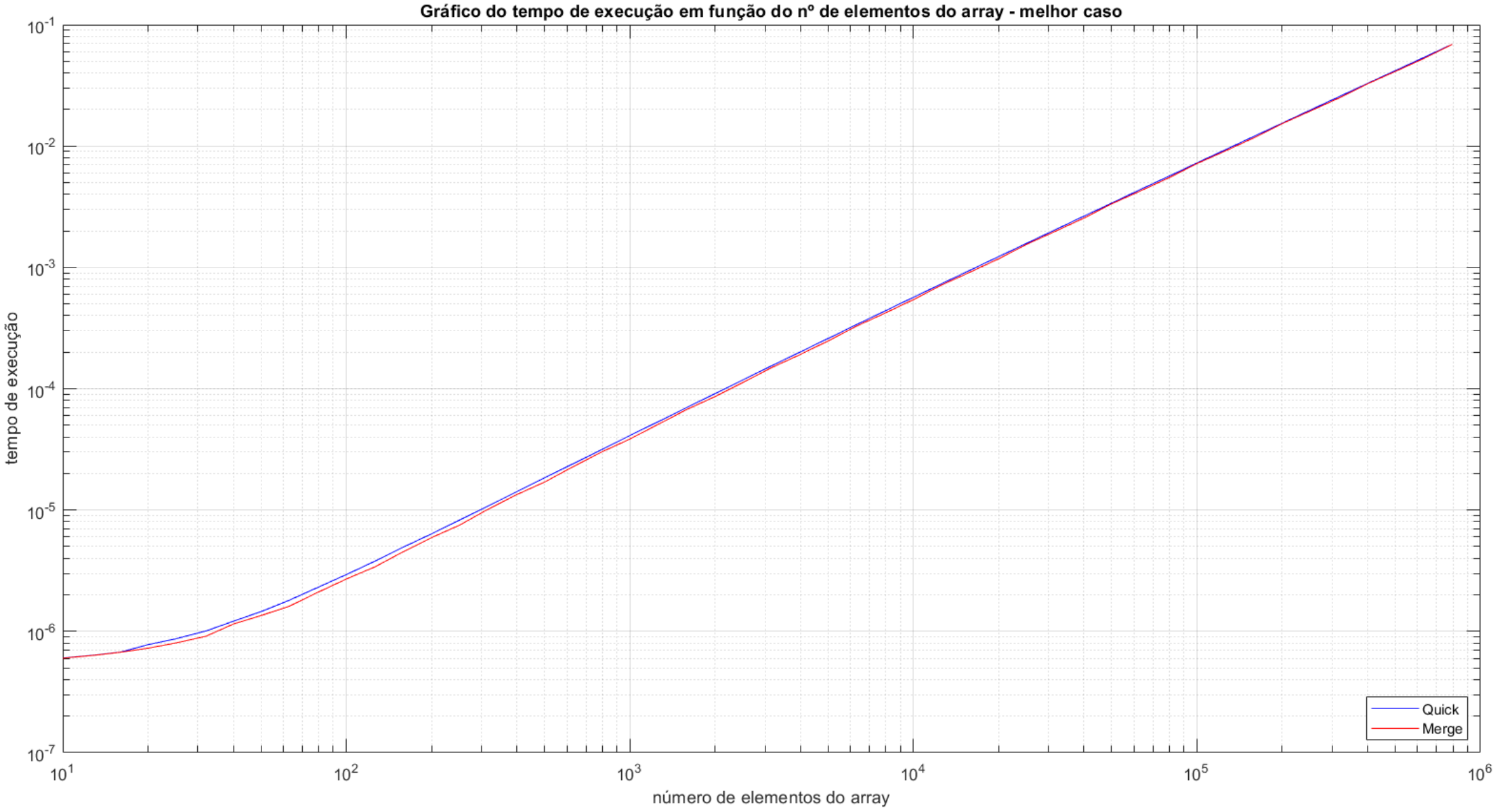


Gráfico do tempo de execução em função do nº de elementos do array - melhor caso
tempo de execução
número de elementos do array
Quick
Merge

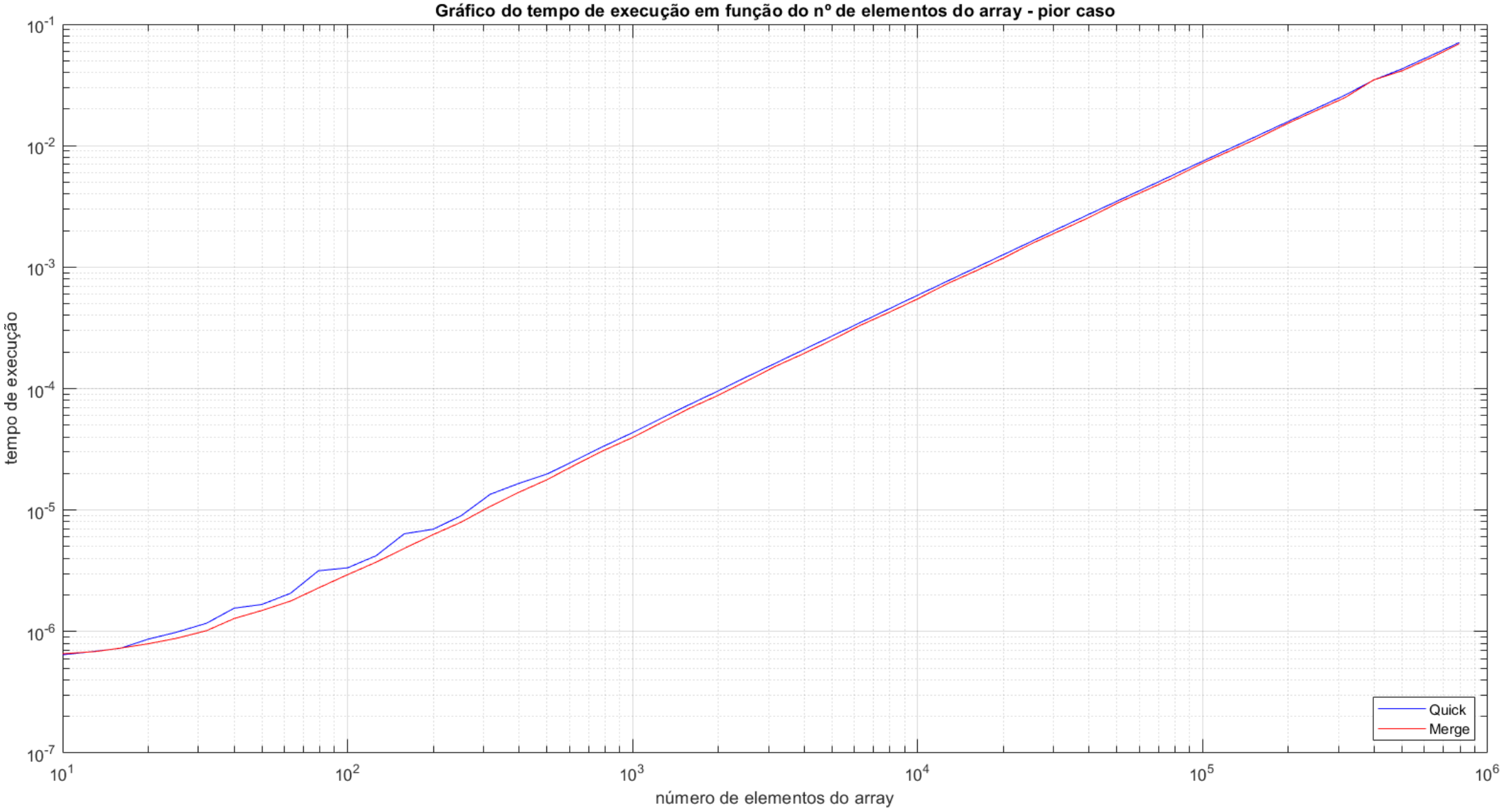


Gráfico do tempo de execução em função do nº de elementos do array - pior caso
tempo de execução
número de elementos do array
Quick
Merge
10^-1
10^-2
10^-3
10^-4
10^-5
10^-6
10^-7
10^1
10^2
10^3
10^4
10^5
10^6

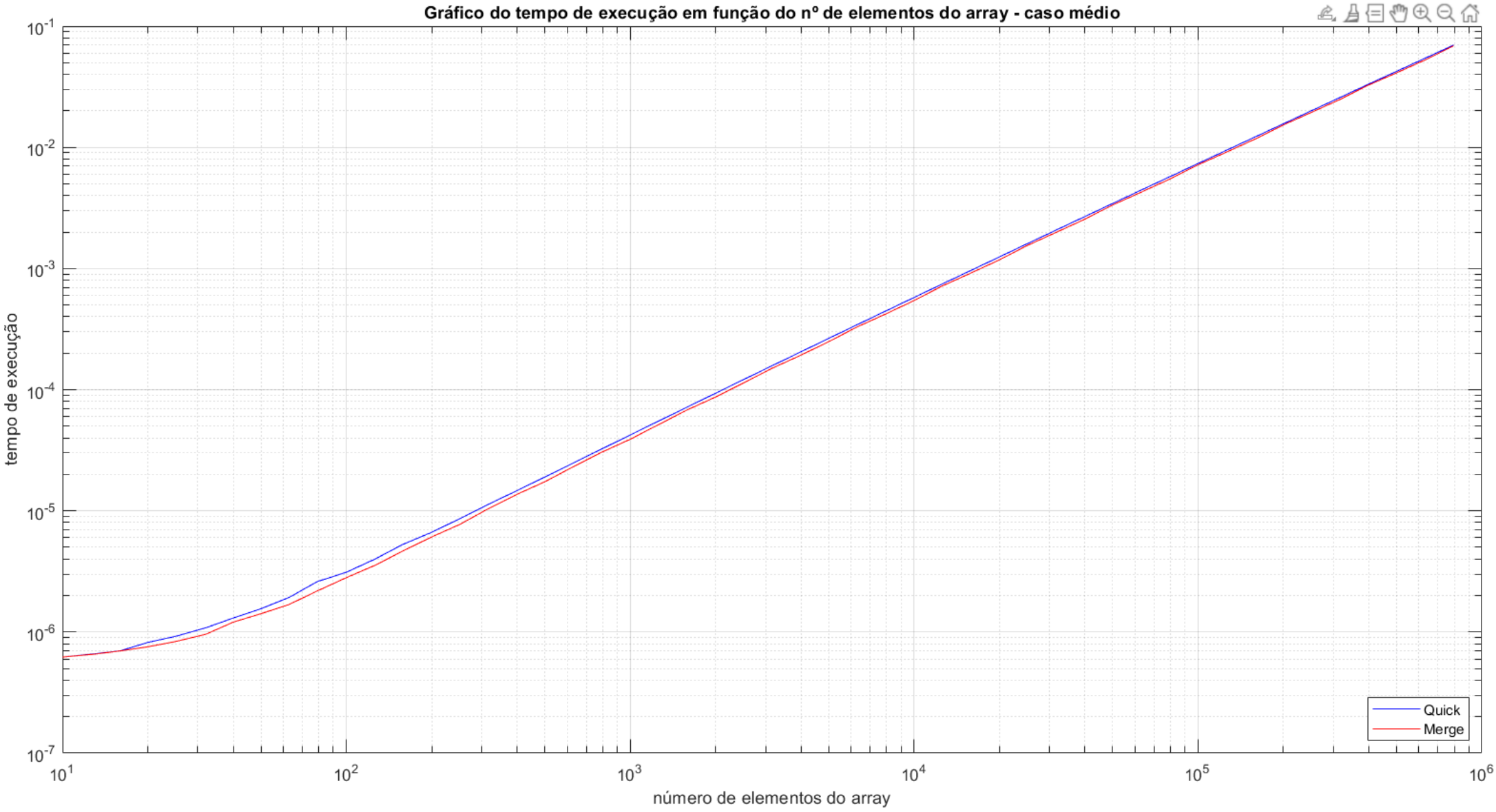
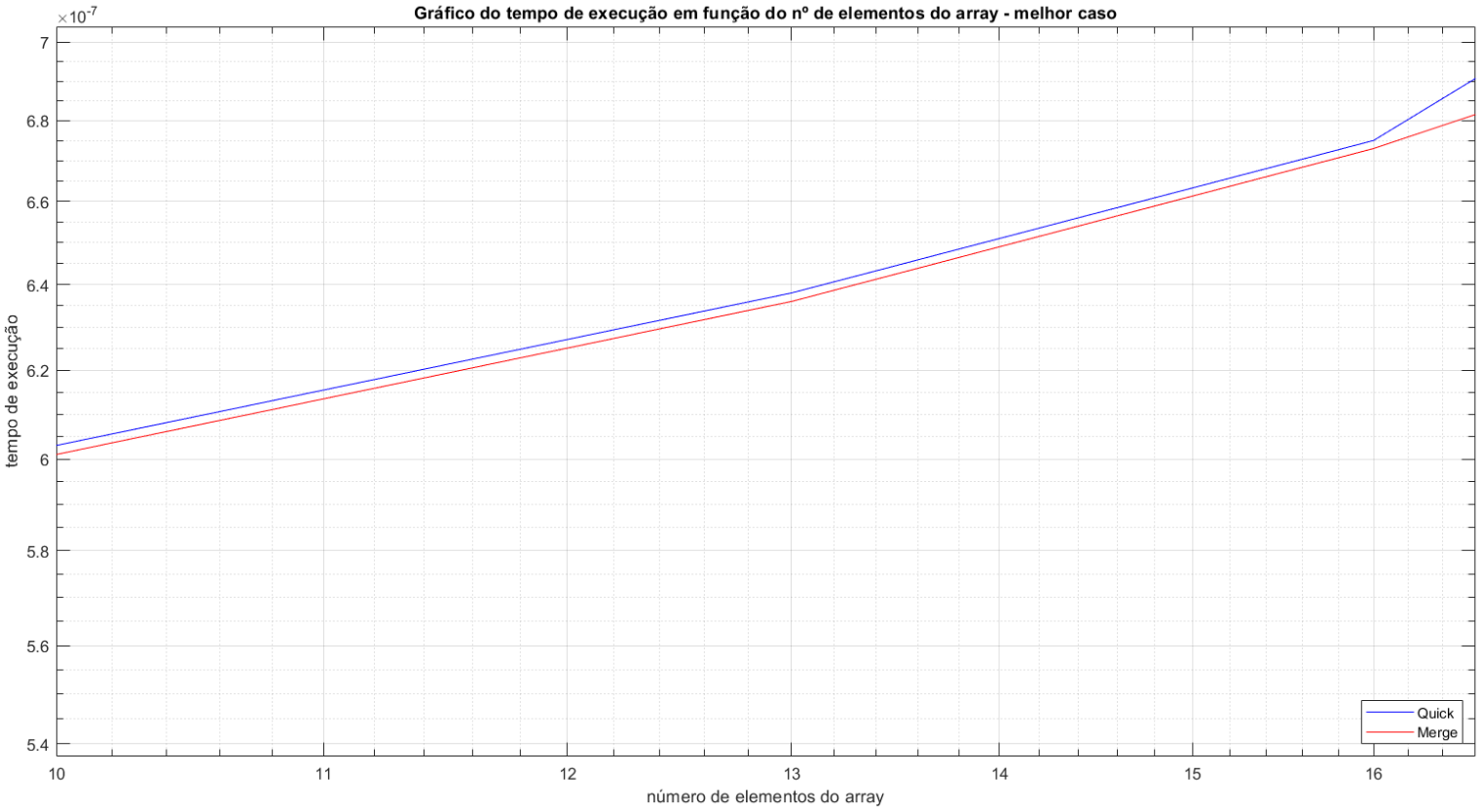
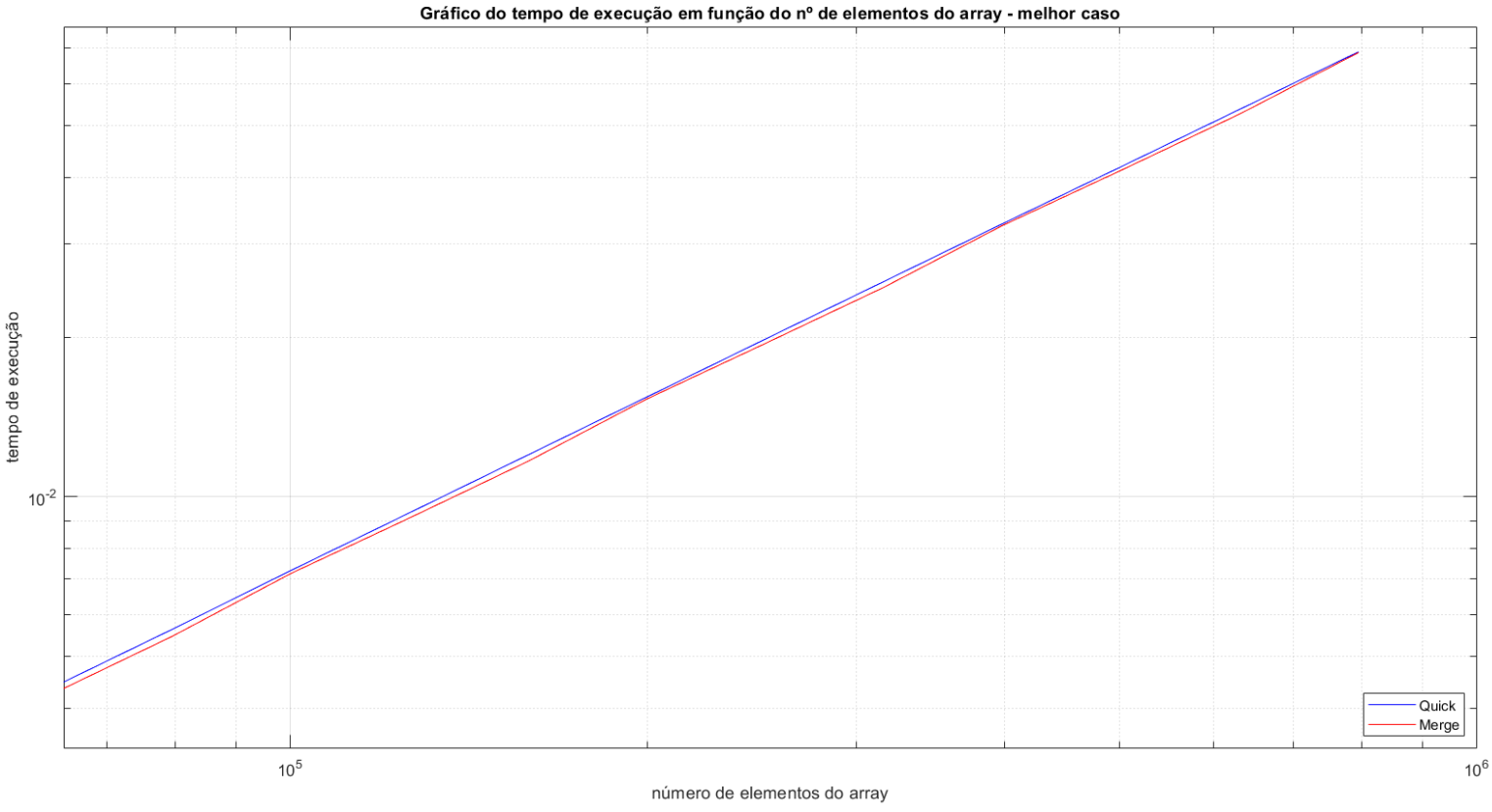


Gráfico do tempo de execução em função do nº de elementos do array - caso médio
tempo de execução
número de elementos do array
Quick
Merge
10^-1
10^-2
10^-3
10^-4
10^-5
10^-6
10^-7
10^1
10^2
10^3
10^4
10^5
10^6

Para o melhor caso:

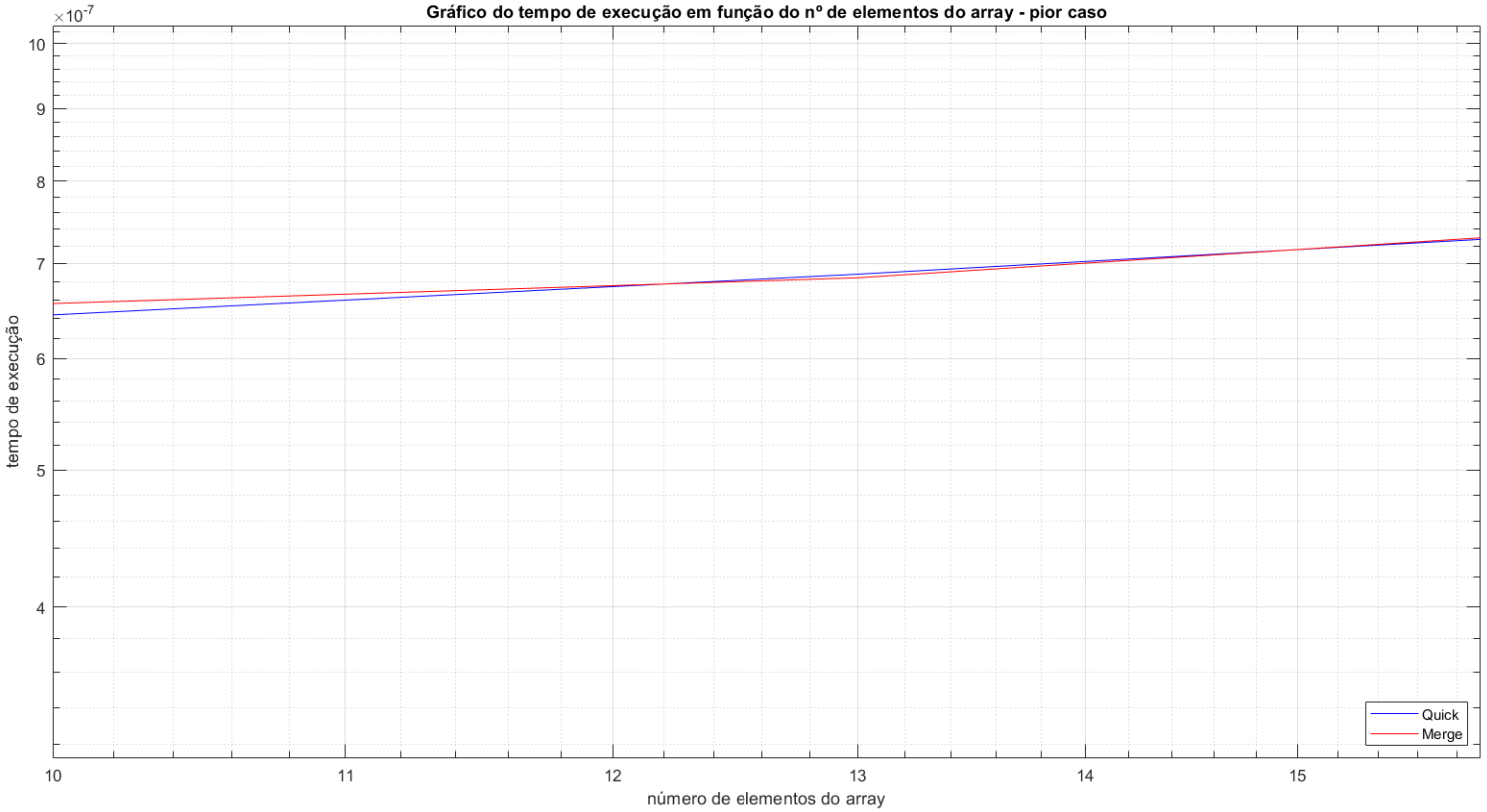
Apesar de, inicialmente, apresentarem valores muito próximos um do outro, através de uma ampliação do gráfico conseguimos perceber que o *merge sort* possui um melhor comportamento em relação ao *quick sort*. O mesmo pode ser dito em toda a continuidade das funções, pelo que concluímos que a melhor escolha, neste caso, é o *merge sort*.





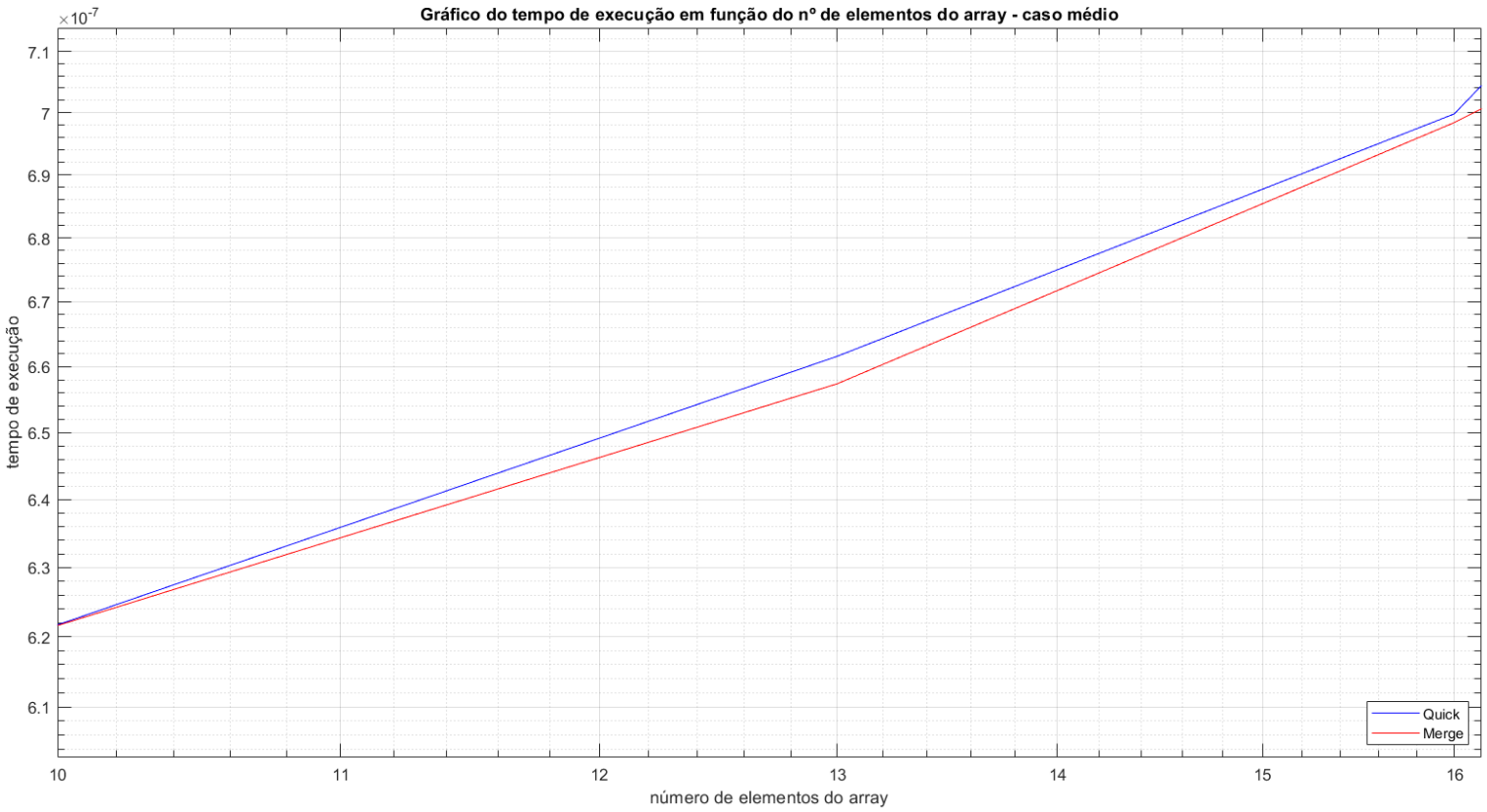
Para o pior caso:

É possível verificar, quando ampliamos o gráfico, que no intervalo [10, 12] aproximadamente, o *quick sort* apresenta uma maior eficiência. Contudo, não devemos esquecer que a complexidade do pior caso do *quick sort* é pior que a complexidade do pior caso do *merge sort*, portanto no intervalo [12, 794328] o melhor algoritmo de ordenação é o *merge sort*.



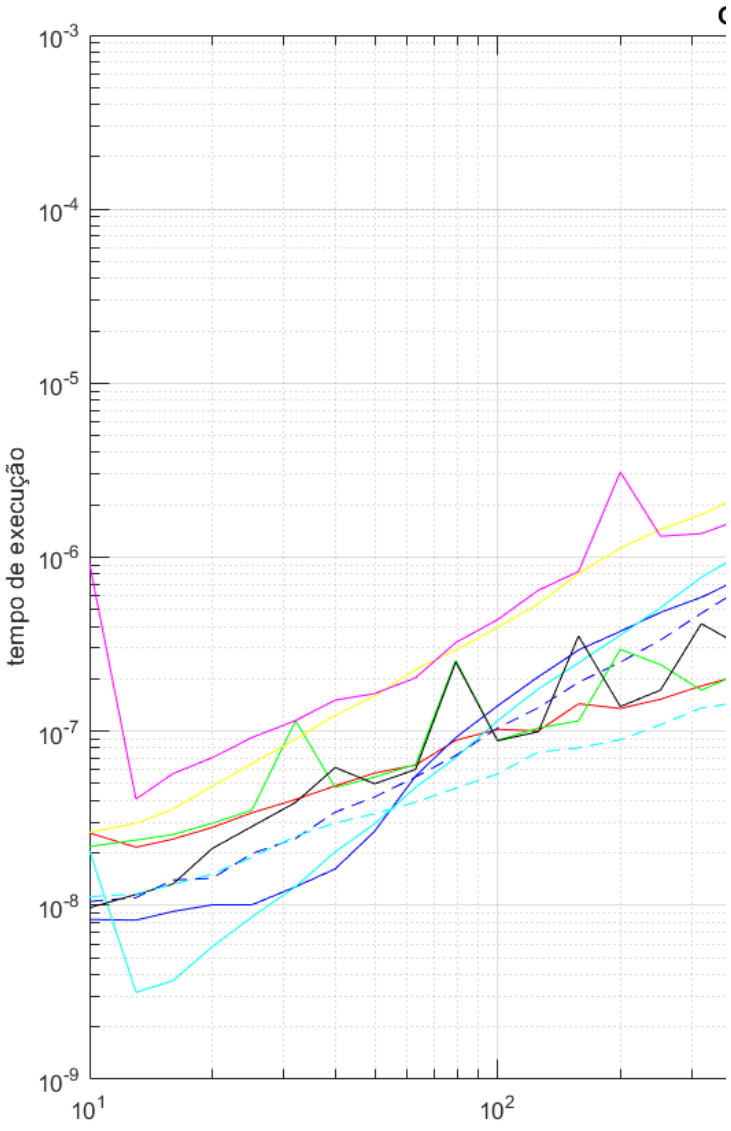
Para o caso médio:

É percetível que o *merge sort* apresenta um melhor comportamento quando *n* tende para valores grandes, todavia é necessário recorrer a uma ampliação do gráfico para avaliar a situação quando *n* assume valores menores. Ao fazê-lo, concluímos que em toda a gama de valores que *n* pode assumir, o *merge sort* é a melhor opção a escolher para obter um algoritmo de ordenação mais eficiente.



Pela análise do gráfico geral dos desvios (próxima página), conseguimos concluir que o algoritmo mais estável dentro dos nove algoritmos avaliados ao longo do presente relatório é o *shaker sort* e que, entre o *merge sort* e o *quick sort*, os algoritmos que apresentaram melhor eficiência, o *merge sort* apresenta um maior período de estabilidade, apesar de possuir um pico acentuado no final da reta, o que pode justificar os resultados obtidos do *merge sort* terem sido melhores que os do *quick sort*.

Num cenário ideal, o *quick sort* deveria ser mais eficiente que o *merge sort* quando trabalhamos com *arrays* de pequenas dimensões e o *merge sort* deveria ser mais eficiente que o *quick sort* quando trabalhamos com *arrays* de grandes dimensões. No entanto, esse resultado só foi possível de observar quando analisamos o gráfico de ambos os algoritmos no pior caso; nos restantes o *merge sort* sempre demonstrou um melhor desempenho.


tempo de execução
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
10^{1}
10^{2}

12. Anexo de Código (Matlab)

```
%Gráficos individuais de melhor
caso, pior caso e caso médio
%Gráficos individuais de desvio
padrão

fid =
fopen('nome_do_ficheiro.txt');
a=30
n = zeros(a,1);
media=zeros(a,1);
minimo=zeros(a,1);
desvio = zeros(a,1);
maximo=zeros(a,1);
for i=1:a
    linha = fgetl(fid);
    linha = split(linha);
    n(i) = str2double(linha{1});
    media(i)=str2double(linha{4});
   minimo(i)=str2double(linha{2});
   maximo(i)=str2double(linha{3});
   desvio(i)=str2double(linha{5});
end
%plot(n, desvio, 'r-');
plot(n,media, 'r-', n, minimo,
'b-', n, maximo, 'g-');
set(gca,'YScale', 'log','XScale',
'log');
legend('Media','Minimo','Maximo',
'Location','Southeast');
title('Título do Gráfico');
xlabel('número de elementos do
array');
ylabel('tempo de execução');
grid;
fclose(fid);

%Gráfico com todas as funções

shaker = fopen('shaker.txt');
shell = fopen('shell.txt');
quick = fopen('quick.txt');
selection = fopen('selection.txt');
rank = fopen('rank.txt');
merge = fopen('merge.txt');
heap = fopen('heap.txt');
insertion = fopen('insertion.txt');
bubble = fopen('bubble.txt');
a_shell_heap = 48;
a_shaker=31;
a_selection_bubble=30;
a_rank=32;
a_quick_merge=50;
a_insertion=35;
shaker_=zeros(31,1);
shell_=zeros(48,1);
quick_=zeros(50,1);
selection_=zeros(30,1);
rank_=zeros(32,1);
merge_=zeros(50,1);
heap_=zeros(48,1);
insertion_=zeros(35,1);
bubble_=zeros(30,1);
nshaker_=zeros(31,1);
nshell_=zeros(48,1);
nquick_=zeros(50,1);
nselection_=zeros(30,1);
nrank_=zeros(32,1);
nmerge_=zeros(50,1);
nheap_=zeros(48,1);
ninsertion_=zeros(35,1);
nbubble_=zeros(30,1);
for i=1:a_shell_heap
    linha = fgetl(shell);
    linha = split(linha);
   shell_(i)=str2double(linha{5});
  nshell_(i)=str2double(linha{1});
    linha = fgetl(heap);
    linha = split(linha);
    heap_(i)=str2double(linha{5});
   nheap_(i)=str2double(linha{1});
end
for i=1:a_shaker
     linha = fgetl(shaker);
     linha = split(linha);
  shaker_(i)=str2double(linha{5});
 nshaker_(i)=str2double(linha{1});
end
for i=1:a_selection_bubble
    linha = fgetl(selection);
    linha = split(linha);
selection_(i)=str2double(linha{5});
nselection_(i)=str2double(linha{1})
;
    linha = fgetl(bubble);
    linha = split(linha);
  bubble_(i)=str2double(linha{5});
 nbubble_(i)=str2double(linha{1});
end
for i=1:a_quick_merge
    linha = fgetl(quick);
    linha = split(linha);
   quick_(i)=str2double(linha{5});
  nquick_(i)=str2double(linha{1});
    linha = fgetl(merge);
    linha = split(linha);
   merge_(i)=str2double(linha{5});
nmerge_(i)=str2double(linha{1});
```

```
end
for i=1:a_rank
     linha = fgetl(rank);
     linha = split(linha);
    rank_(i)=str2double(linha{5});
   nrank_(i)=str2double(linha{1});
end
for i=1:a_insertion
     linha = fgetl(insertion);
     linha = split(linha);
insertion_(i)=str2double(linha{5});
ninsertion_(i)=str2double(linha{1})
;
end
plot(nshell_ ,shell_, 'r-',
ninsertion_, insertion_, 'b--',
nselection_,selection_, 'b-',
nheap_, heap_, 'g-',
nshaker_,shaker_, 'y-',
nbubble_,bubble_, 'm-', nrank_,
rank_,'c-',nquick_, quick_, 'k-',
nmerge_, merge_, 'c--');
set(gca,'YScale', 'log','XScale',
'log');
set(gcf,'color','w');
legend('Shell', 'Insertion',
'Selection','Heap', 'Shaker',
'Bubble', 'Rank', 'Quick',
'Merge','Location','Southeast');
title('TITULO_DO_GRÁFICO');
xlabel('número de elementos do
array');
ylabel('tempo de execução');
grid;
fclose(shaker);
fclose(quick);
fclose(bubble);
fclose(selection);
fclose(heap);
fclose(bubble);
fclose(rank);
fclose(merge);
fclose(insertion);
```